Providing Support for Evolvable Software Systems: An Agent Based Approach

Dott. Ing.

Christian Vecchiola

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering

ING-INF/05

University of Genoa

Faculty of Engineering
Department of Communication Computer and System Sciences

Advisors: Chiar.mo Prof. Ing. Massimo Paolucci
Chiar.mo Prof. Ing. Antonio Boccalatte

Date: February 20, 2007
Contents

Foreword ................................................................................................................ 7

Part I. Theory .......................................................................................................... 12

Current Trends in Software Development .............................................................13

1. A Short Story of Software Engineering...............................................................13

2. Methodologies for Software Development .......................................................20
   2.1 A Brief History of Software Life-Cycle Process Models..............................20
      2.1.1 Waterfall Model .....................................................................................20
      2.1.2 Spiral Model ...........................................................................................21
      2.1.3 Capability Maturity Model (CMM) and its Integration (CMMI) .......... 21
      2.1.4 Rational Unified Process (RUP) .............................................................22
   2.2 Agile Software Development .............................................................................23
   2.3 Model Driven Development .............................................................................25

3. Software Computing Models ................................................................................27
   3.1 From Paradigms to Computing Models .........................................................28
   3.2 Imitating Biology: Autonomic Computing and Beyond ..............................29
      3.2.1 The Rationale Behind Autonomic Computing ..........................................29
      3.2.2 Autonomic Computing Reference Model ...............................................31
      3.2.3 Emerging Key Applications of Autonomic Computing ..........................33
      3.2.4 One Step Beyond: Conscientious Software and Other Biological Models 34
   3.3 Handling Failures: Recovery Oriented Computing .........................................37
      3.3.1 Recovering Instead of Preventing ............................................................37
      3.3.2 Software Recovery Experiments ...............................................................38
      3.3.3 Automatic Diagnosis ...............................................................................39
      3.3.4 Fine Grained Partitioning and Recursive Restartability .......................40
      3.3.5 Reversible Systems for Operators ............................................................41
      3.3.6 Defense in Depth ......................................................................................41
      3.3.7 Redundancy ............................................................................................41
      3.3.8 Conclusions ............................................................................................42


5. Observations .......................................................................................................46
Contents

AGENTS AND MULTI-AGENT SYSTEMS................................................................................................................. 47
1. Introduction......................................................................................................................................................... 47
2. The Agent Abstraction .................................................................................................................................. 48
   2.1. The Notion of Agent ..................................................................................................................................... 48
   2.2. Agent Architectures ..................................................................................................................................... 51
      2.2.1. Deliberative Architectures .................................................................................................................. 52
      2.2.2. Reactive Architectures ...................................................................................................................... 53
      2.2.3. Hybrid or Layered Architectures ....................................................................................................... 54
      2.2.4. Practical Reasoning and Belief-Desire-Intention (BDI) Architectures .............................................. 55
      2.2.5. Considerations ....................................................................................................................................... 56
   2.3. Comparison: Agents, Objects, and the Others ......................................................................................... 56
      2.3.1. Agents vs. Objects ............................................................................................................................... 57
      2.3.2. Agent vs. Expert Systems .................................................................................................................. 59
3. The Environment Abstraction ....................................................................................................................... 59
   3.1. Characterization ......................................................................................................................................... 59
   3.2. Environment Models ................................................................................................................................. 60
   3.3. Interaction Facilities ................................................................................................................................... 63
4. Multi-Agent Systems ...................................................................................................................................... 64
   4.1. The Notion of Multi-Agent System ........................................................................................................ 64
   4.2. Properties of Multi-Agent Systems ......................................................................................................... 65
   4.3. Historical Perspective .................................................................................................................................. 66
5. Applications ...................................................................................................................................................... 68

AGENT ORIENTED SOFTWARE ENGINEERING ................................................................................................. 71
1. What is Agent Oriented Software Engineering? ............................................................................................. 71
2. The Nature of Complex Software Systems ..................................................................................................... 74
3. The Weapons of AOSE ..................................................................................................................................... 80
   3.1. Fundamental Techniques to Tackle Complexity ....................................................................................... 80
   3.2. Advantages of Agent-Oriented Abstractions ............................................................................................ 82
      3.2.1. Addressing Decomposition ................................................................................................................ 84
      3.2.2. Supporting Hierarchy and Organization ............................................................................................. 85
      3.2.3. Supporting Evolution and Adaptability ............................................................................................. 86
   3.3. Observations ................................................................................................................................................ 88
4. Agent-Oriented Methodologies ..................................................................................................................... 88
   4.1. What is a Methodology? ............................................................................................................................ 88
   4.2. Gaia ............................................................................................................................................................. 89
   4.3. MaSE .......................................................................................................................................................... 90
Contents

5. Observations .......................................................................................................................... 136

DISTRIBUTED PROGRAMMING WITH AGENTSERVICE ................................................................. 138

1. Introduction ........................................................................................................................... 138
2. Agent-Based Distributed Computing .................................................................................. 140
3. AgentService Distributed Computing Support .................................................................. 143
   3.1. Overview .......................................................................................................................... 143
   3.2. Messaging Service ......................................................................................................... 143
       3.2.1. Messaging System Infrastructure ............................................................................. 144
       3.2.2. Distributed Messaging Module ............................................................................. 146
       3.2.3. Message Serialization and Delivery ....................................................................... 147
       3.2.4. Observations .......................................................................................................... 149
   3.3. Mobility in AgentService .............................................................................................. 150
       3.3.1. Code Mobility ........................................................................................................... 150
       3.3.2. Mobile Agents and Mobile Agent Systems ............................................................. 155
       3.3.3. Related Works ......................................................................................................... 157
       3.3.4. AgentService Mobility Infrastructure .................................................................. 163
       3.3.5. Observations .......................................................................................................... 171
4. Building Complex Services: The Case of Load Balancing ................................................. 172
   4.1. Overview .......................................................................................................................... 172
   4.2. System Architecture: Virtual Platforms ........................................................................ 173
   4.3. Load Balancing Policies ............................................................................................... 174
   4.4. Observations .................................................................................................................. 175
5. Conclusions .......................................................................................................................... 175

PROGRAMMING INTERACTIONS: ONTOLOGIES AND PROTOCOLS .................................................. 179

1. Introduction ........................................................................................................................... 179
2. Ontologies ............................................................................................................................. 180
   2.1. Definition ........................................................................................................................ 180
   2.2. Languages for Ontologies ............................................................................................. 183
   2.3. Ontology Development ................................................................................................. 186
   2.4. Applications .................................................................................................................... 187
3. Ontologies and Multi-Agent Systems .................................................................................. 189
   3.1. The Need for a Meta-Ontology ....................................................................................... 189
   3.2. FIPA Recommendations for Ontologies ....................................................................... 190
       3.2.1. Meta Ontology for Agent Interaction ...................................................................... 190
       3.2.2. Agent Communication Language .......................................................................... 192
       3.2.3. Ontology Agent ....................................................................................................... 192
### Contents

3.3. State of the Art ................................................................................................................................. 193

4. Ontology Support in AgentService .............................................................................. 195
   4.1. The Object Model .......................................................................................................................... 196
       4.1.1. Classes and Interfaces vs. Schemas ............................................................................. 196
       4.1.2. The Ontology Class and its Services .......................................................................... 199
   4.2. Language Support ......................................................................................................................... 199
   4.3. Ontology Agent ............................................................................................................................. 201
   4.4. Ontology Development Process ................................................................................................. 202

5. Interaction Protocols ......................................................................................................................... 203
   5.1. Interaction Protocol Representations .......................................................................................... 203
   5.2. Related Works .............................................................................................................................. 205
   5.3. Design ........................................................................................................................................... 206
   5.4. Code Generation .......................................................................................................................... 210

6. Observations ................................................................................................................................. 212

Conclusions and Future Works ........................................................................................................... 215

Bibliography ................................................................................................................................. 222

List of Figures ................................................................................................................................. 245
Foreword


The thesis is about methodologies and tools to support software development for real life systems. The requirements of contemporary software systems are completely different from those in the past: every software product released by the industry, or conceived within academic research groups, is not anymore designed to operate alone. Software systems provide their services by relying on other software components which are developed by third parties and located everywhere on the planet. The advances in networking and the high availability of network bandwidth for almost everyone dramatically changed the role of the network not only in the IT community but almost for every user. Network connection, and in particular access to the Internet, is a common feature not only for computers but for a wide range of devices: mobile phones can browse the Web, video players can access the content of Internet TV, provide video on demand capabilities, and upgrade their firmware when a new version is released. This new scenario changes the concept of software we have today and the one we want to deliver tomorrow: every software application we design is conceived to be part of a bigger system upon which it depends in order to provide its services. Thanks to the development of web services technology and *service oriented architectures* (SOA) the integration of new features in software applications has become easier and the Internet, again, played an important role. Web services are software components which provide a well known set of functionalities and that are accessible through the web by using the HTTP protocol. The high availability of network bandwidth and the simple interaction pattern (based on SOAP and XML) offered by web services allowed their wide diffusion and use. It is reasonable to think that everyone is frequently connected and network
usage costs are becoming cheaper and cheaper. Hence, it is possible, from a design point of view, to actually exploit web services technology and include it as a component in the systems we design. When software projects require services that are not directly provided we can still compose different services or, better, create services which exploit other services, integrate their results and provide them according to our needs. These techniques are referred to web services orchestration and actually lead to the development and the diffusion of service oriented architectures: their wide use is what makes nowadays software an inherently dependable system.

System dependability introduces new challenges for software engineers. One of the most evident ones is security: dependable systems rely on third party code which has to be trusted or made not dangerous. Anyways, this is not the biggest challenge we have to face. The most difficult task is to design and implement software solutions that will be part of a dynamic, open, and complex system: we have to use approaches and tools allowing us producing flexible software applications. By the term flexible we mean able to adapt to the environment, and the environment in this case is represented by every third party component a software application uses or by every system which relies on in order to perform its activity. This environment is naturally evolvable since real life systems reflect advances in technology and are strongly influenced by end users preferences and expectations. For this reason, the components of these systems should be able to evolve themselves; that means able detect and adapt to changes and automatically upgrade when needed. In other words, they should be flexible. In order to support this scenario we have to design software applications as intelligent components which are able maintain themselves and provide them with the ability to interact with the system in an open and dynamic manner. Openness and dynamicity become fundamental in evolvable environments since they allow reacting to environment changes in a profitable manner. Research in this field has led to the formalization of autonomic computing, recovery oriented computing, and the more recent notion of conscientious software, which aim to provide models and techniques able to cope with the problems previously discussed.
Moreover, complexity, openness and dynamicity become even more important for large and ultra-large scale systems. At a larger scale we deal with systems of systems or software ecosystems: they are always running and comprise different software systems. The whole Internet is an example. In this scenario complexity increases dramatically and size is the most influencing factor: there are billions of lines of code, millions of users, thousands of interacting systems. Dynamicity is amplified too: ultra-large scale systems are continuously changing. If we take a snapshot of these systems we will have the same picture of a big metropolis: it never stops, there is always something being built, not working or being fixed, and its structure is not defined a priori but it grows under the influence of their inhabitants which are part of it. Ultra-large scale systems expose the same degree of dynamicity. As a consequence openness is augmented too and this effect is mostly due to the increased size and dynamicity. In order to cope and to properly handle these systems we need completely new methodologies because many of the hypotheses basing the current approaches are not satisfied in these contexts. A recent study aimed to define the requirements for ultra-large scale systems noticed that we will have to define completely new methodologies integrating notions and approaches from different research fields such as sociology, economy, game theory, etc.

The object of this thesis is not to provide a complete solution to support ultra-large scale software systems but to provide a valuable approach, a good methodology and an efficient tool to design and implement components (that are systems as well) for these systems. I found that agent technology provides useful and powerful abstractions to model these elements with the concepts of software agent and multi-agent systems. We can define a software agent as autonomous software entity able to expose a flexible behavior; as a consequence multi-agent systems are community of distributed software agents which interact by means of cooperation, negotiation and competition. Multi-agent systems are open structures characterized by distributed control, asynchronous computation, and fault tolerance. All these features make them an interesting abstraction to model open, complex and dynamic systems. Moreover, due to the fact that agents are intelligent software components able to dynamically interact with peers, multi-agent systems can
expose an emergent behavior. This property is useful to reason about – and to understand – complex systems, which normally expose an emergent behavior. These are the reasons which convinced me in using agent technology to model and implement evolvable systems.

During my doctoral studies I faced the problem from a theoretical point of view and then I tried to apply the outcomes of these studies to provide a practical way for developing evolvable software systems. This led to the design and the development of an agent programming framework called AgentService which is the second topic of this thesis. AgentService is comprehensive framework to develop distributed multi-agent systems which models agents as multi-behavioral autonomous software entities. The main idea behind the design of the framework is to provide a valuable tool to implement real systems: by starting from a theoretical model of the software agents users can define software agents, characterize their behavior, and program the interaction with peers. The framework assists designers in each step of the process and simplifies the development task and the deployment of multi-agent systems. AgentService allows developing highly dynamic, modular and customizable multi-agent systems: the software infrastructure which software agents rely on can be easily adapted to different scenarios and changing requirements. This is crucial when we want to provide programming support for evolvable software systems. For all these reasons the subject of this thesis falls into the field of Agent Oriented Software Engineering, better known as AOSE, which is the study and the application of agent-based techniques and methodologies to model and implement software systems.

The remaining of this thesis is divided in two parts and organized as follows: the first part, comprising chapters 1, 2, and 3 give an overview of the state of the art of the current trends in software development with a focus on agent oriented software development; the second part, along with the conclusions, will describe in detail the outcomes of my research activity. In particular, chapter 4 describes the AgentService programming framework, while chapter 5 and 6 discusses some advanced features of the framework.
which actually make AgentService an effective tool for agent oriented software development. An evaluation of the work of the past three years concludes the dissertation in Chapter 7.
Part I. Theory

This section introduces the art of software development and presents the fundamental concepts of agent oriented technology. The objective of this section is to define the context in which my doctoral studies developed. A concise, yet meaningful, description of the evolution of Software Engineering, along with hints on its future directions, is given in Chapter 1. Chapter 2 and 3 introduce agent technology and explain why Agent Oriented Software Engineering (AOSE) – the main subject of this dissertation – is a valuable approach for complex software system development.
1. A Short Story of Software Engineering

Today the “software” comprising the carefully planned interpretive routines, compilers, and other aspects of automatic programming are at least as important to the modern electronic calculator as its “hardware” of tubes, transistors, wires, tapes, and the like [Tukey58].

According to the Fred R. Shapiro [Shapiro00], librarian and etymologist at Yale Law School, this was the first mention of the word *software* with a computer-related meaning. It dates back to 1958 and appeared in an article of the *American Mathematical Monthly* written by John W. Tukey, statistician and co-inventor of the Fast Fourier Transform. The subject of the Tukey’s article was the teaching of concrete mathematics and he adopted the word *software* to describe all those automatic programming aids through which it was possible to apply mathematics. Some years later, in 1968, the term *Software Engineering* was used to entitle the first conference on software development and
maintenance sponsored by the NATO Scientific Committee [Naur69]. The expression
Software Engineering did not come along with a definition, but identified a need; quoting
from the original words of its inventor, F. L. Bauer, the general feeling was the following:

The whole trouble comes from the fact that there is so much tinkering with software. It is not made in a
clean fabrication process, which it should be. What we need is software engineering.

Today, the term Software Engineering still conveys the same idea, even if more formal and
structured definitions can be found. The IEEE Standard Computer Dictionary defines it as:

The application of a systematic, disciplined, quantifiable approach to development, operation, and
maintenance of software; that is, the application of engineering to software [IEEE90].

Again, according to Sommerville:

The specification, development, management, and evolution of software systems. Not constrained by
materials governed by physical laws or manufacturing processes. Theories, methods, and tools needed to
develop software. Evolving models of the real world [Sommerville96].

If we compare the most recent definitions with the original expression of Bauer we can
notice that today Software Engineering (SE) does not only concern software development
but its entire life cycle: from the identification of the problem domain to the deployment
and the maintenance of the software. This is a general misconception\(^1\): SE strongly
influences the practice of software development by providing models and methodologies
intended to efficiently deliver and maintain quality software solutions but it is not
software development. Such methodologies and computing models change according to
the advances in technology and Computer Science, and, more important, according to

\(^1\) An interesting discussion about this issue and the most common myths about Software Engineering can
be found at: http://www.cs.utexas.edu/~sahilt/research/SEMyths.html
what we identify by the term software. Whereas the idea of SE remained basically the same, the concept of software strongly changed over time. The definition provided by Tukey was absolutely practical and reflected the usage pattern of computer technology in the fifties, which was basically limited to computer scientists and not conceived for the whole public. According to the Princeton statistician, the term software identified all those aspects related with programming that were not considered hardware: system level routines (if any), compilers and interpreters, and specific purpose programs. In other words it referred to the intangible part of a computer system that makes it perform a specific task. Nowadays software is not anymore just a collection of algorithms, system level libraries and tools for development: software has become a service and the IT is more and more turning into a commodity like other industries did before (i.e. electricity) [Schwartz05]. Advances in technology made available computers to a wide range of users and many of them are not scientist or technicians. Advances in networking and the advent of the Internet made all these computers easily connected and encouraged the development of software for a wide range of market segments: from the automotive and the entertainment industry, to the mobile telephony. Mary Shaw [Shaw96] sketched an interesting picture of the evolution of Software Engineering from the fifties to the nineties. During this period:

.. the focus of the academic community (though not so much the industrial software development community) has shifted from simply writing programs to analyzing and reasoning about large distributed systems of software and data that come from diverse sources [Shaw96].

Figure 1 lays out the highlights of these shifts. Due to the constant diffusion of computer systems and the increase of computational power of the single machines the size and the complexity of software systems grew exponentially: we then moved from the programming any which way to the programming in the world style. Whereas originally the focus was on the single algorithm, at the end of the century the attention was captured by the architectural properties of software systems – which could spread over the planet – and the interaction patterns of their components. It is interesting to notice that in 1968 a
large-scale software system was characterized by at least thirty thousands deliverable instructions, more than ten software developers, and more than six months of development [Schwartz70]. Today these numbers characterize small size projects: the development time spans through years and we have millions of line of code written in high-level programming languages. The current focus is on systems of systems or software ecosystems and the future will be characterized by the study and the implementation of ultra large-scale systems (ULS) [Northrop05].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Specifications</strong></td>
<td>Programming any which-way</td>
<td>Programming in-the-small</td>
<td>Programming in-the-large</td>
<td>Programming in-the-world</td>
</tr>
<tr>
<td>Mnemonics, precise use of prose</td>
<td>Simple input-output specifications</td>
<td>Systems with complex specifications</td>
<td>Distributed systems with open-ended specs</td>
<td></td>
</tr>
<tr>
<td><strong>Design emphasis</strong></td>
<td>Emphasis on small programs</td>
<td>Emphasis on algorithms</td>
<td>Emphasis on system structure management</td>
<td>Emphasis on subsystem interactions</td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td>Representing structure, symbolic information</td>
<td>Data structures and types</td>
<td>Long-lived databases</td>
<td>Data &amp; computation independently create, come and go</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>Elementary understanding of control flow</td>
<td>Programs execute once and terminate</td>
<td>Program systems execute continually</td>
<td>Suites of independent processes cooperate</td>
</tr>
</tbody>
</table>

*Figure 1. Highlights in academic attention in software engineering [Shaw96]*

The increased dimensions of current software systems are just one of the causes that make them more complex. The massive spread of desktop computers in the nineties and subsequent the diffusion of high bandwidth connections to the Internet opened new possibilities and new challenges either to the academic community or the IT industry. The computational power and the network connection were not a problem anymore and this allowed rethinking the computing models in ways the were not possible before: these were the years when Grid Computing [Foster01] and Pervasive Computing or Ubiquitous Computing [Weiser91] established as research fields. It is interesting to notice how these
two computing models became reasonable paths to explore because it was actually possible to provide software implementations and find a market for them.

*Grid Computing* distributes processing across a parallel infrastructure. Throughput is increased by networking many heterogeneous resources across administrative boundaries to model a virtual computer architecture. For a computing problem to benefit from a grid, it must require either large amounts of computation time or large amounts of data, and it must be reducible to parallel processes that do not require intensive inter-communication. The successful application of this computing model is based on the assumption that computers belonging to the Grid are constantly connected to – or at least that they can easily access to – the Internet. Nowadays this computing model is widely studied and developed as witnessed by the various editions of the international conferences (*HPDC*\(^2\), *CCGrid*\(^3\), *IPDPS*\(^4\), etc.) on that subject and by the deployment of large scale Grids providing support in research fields other than Computer Science like NBIRN [Grethe05] and MediGrid [Montagnat03]. Moreover, the concept of *Utility Computing*, that is a computing model in which users pay to use the computing services of a deployed grid infrastructure and require a well defined quality of service, is being investigated and there is a market for it.

*Ubiquitous Computing* integrates computation into the environment, rather than having computers which are distinct objects. The term was originally coined in 1988 by Mark Weiser, researcher at the Xerox Palo Alto Research Center (PARC) who envisioned a future in which computer technology would have *disappeared into the background*, being

---


\(^3\) CCGrid is an acronym for IEEE International Symposium on Cluster Computing and the Grid.

constantly there and seamlessly providing us services like road signs and street lights do. According to Weiser the technology required for *Ubiquitous Computing* is fundamentally characterized by cheap low-power computers, software for ubiquitous applications, and the network that ties them together. Today we can easily inject small networked computers into any kind of object which is a requirement for *ambient intelligence*\(^5\). Thanks to wireless networks installed in public environment or buildings (museums, shopping city centers and train stations) people using their PDA or smart phones can be guided to discover services which they are interested in and organize better exploit the facilities offered by the environment. *Sensor networks*\(^6\) are a promising research field supporting ubiquitous computing. The October issue of *IEEE Pervasive Computing* focuses the attention on the advances made in the last ten years in deploying software services into the transportation field. In the editorial [Want06], Roy Want briefly summarizes the improvements of entertainment systems which are deployed in international flights: nowadays, while on board, we can take advantage of wide range of services. We can listen to music, watch movies, comedies, and documentaries by using the TV on demand system, call home by paying with our credit card and even talk with other passengers by using the messaging system. Soon it will be possible to access the Internet. We can make all these things while we are on our seat and the services are available for each user. These are illuminating examples because they put in evidence that software has deeply penetrated into society, that people are getting used to it, and that they start considering it as a *commodity* or a *service*.

\(^5\) The vision promoted by Philips Research seems to be the most adapt in this context. Ambient intelligence is explained as people living easily in digital environments in which the electronics are sensitive to their needs, personalized to their requirements, anticipatory of their behavior and responsive to their presence. See [http://www.research.philips.com/technologies/syst_softw/ami/index.html](http://www.research.philips.com/technologies/syst_softw/ami/index.html).

\(^6\) Sensor networks are networks consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations.
In the current scenario software is spread everywhere and more and more tasks are demanded to technology. For these reasons \textit{dependability} has become a fundamental requirement for software system. \textit{Dependability} is defined by IFIP WG-10.4 as the \textit{trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers}\textsuperscript{7}. This property has long been a critical requirement for space and defense systems, but today, it has become a requirement even for software. In the case of software there is no definition universally accepted; dependability is generally defined as a multi-attribute property including reliability, availability, safety, fault-tolerance, robustness, and security. More than before, providing useful computing models and development methodologies that support the development, the analysis, and the maintenance of dependable systems is an important issue. What makes such tasks harder is the fact that software systems are now \textit{dynamic} and \textit{evolvable}. We have then to provide useful methodologies which enable us to deliver software products in time and with a reasonable degree of trustworthiness. In this direction new software development methodologies like \textit{extreme programming} and \textit{agile software development} have been investigated. On the other hand, new computing models have to be conceived to make our software support evolution and expose dependability. The latest trends in \textit{Software Engineering} have been characterized by the diffusion of two computing models which could potentially provide these properties: \textit{Recovery Oriented Computing} and \textit{Autonomic Computing}.

The rest of this chapter will focus on these relatively new aspects of software Engineering. I will first present a brief explanation of the development methodologies and then I will concentrate on the computing models with a brief digression on what we expect for the future and what challenges we will have to face.

\textsuperscript{7} International Federation of Information Processing Working Group 10.4 (IFIP WG-10.4) on Dependable Computing and Fault Tolerance. More information available at \url{http://www.dependability.org/wg10.4/
2. Methodologies for Software Development

2.1. A Brief History of Software Life-Cycle Process Models

The problem of providing useful methodologies to develop software has always been a critical issue in Software Engineering which was initially born as a reaction to the lack of systemic methods for software development. The statement: “What we need is software engineering!” – expressed in 1968 by Bauer at the NATO Conference – clearly points out that the most important reason behind the software crisis was the absence of methodology for software development. He then advocated an engineering approach to the discipline of software development which implies a systematic approach to the problem. From that time numerous methodologies have been proposed. These methodologies consider the software life-cycle as a process and then provide models to structure and organize it. A brief review of the most successful software life-cycle process models are presented in this section.

2.1.1 Waterfall Model

One of the earliest software models for software development was the Waterfall Model. The term waterfall clearly identifies the nature of this model which is characterized by a sequential progress through five distinct and separated phases: Requirement Analysis, Design, Implementation, Integration, and Maintenance. In the same manner the water moves from the source of the waterfall down to its bottom part. The general assumption behind the model is that the time spent early on making sure that requirements and design are absolutely correct is very useful in economic terms since it will save much time and effort in subsequent phases. For this reason only when a phase is completed it is possible to move to the next one. Despite criticisms to this model, which can be found since 1970 [Royce70], the Waterfall Model is still widely used by large software development houses like the U.S. Department of Defense and the NASA. Its strength resides in its simplicity,
its structured approach, and on the fact it provides identifiable milestones in the development process. Critics to the Waterfall Model are concerned with the fact it does not provide any risk management strategy and it is not suitable for project wherein requirements change during the life-cycle.

2.1.2 Spiral Model

The limits of the Waterfall Model were clearly set out in 1970 by Winston Royce who in a report on his experiences of developing large software systems [Royce70] observed that the clear separation among software phases and their strictly sequential execution were not flexible enough to accommodate real life software project requirements. He then proposed a modified version of that model which introduced the concept of iteration of phases, which found its best expression in the Spiral Model introduced by Barry Boehm [Bohem88]. The Spiral Model was the first model to explain why the iteration matters. As originally envisioned, the iterations were typically 6 months to 2 years long and can be represented as the revolving cycles of a spiral: the process evolves by starting from the center of the spiral and moving towards its end, and each cycle develops around the previous results. Each phase of the spiral starts with a design goal: the alternatives for achieving it along with the constraints and the risks related to their adoption are evaluated and a solution is identified. The cycle ends with the client (who may be internal) reviewing the progress thus far and a commitment or a plan for the next cycle. Analysis and engineering efforts are applied at each phase of the project, with an eye toward the end goal of the project. The advantages of this model are to be found in the fact that it provides more realistic estimations as work progresses, it is more able to cope with the inevitable changes in requirements, and it anticipates the implementation phase.

2.1.3 Capability Maturity Model (CMM) and its Integration (CMMI)

The Capability Maturity Model for Software (CMM) [Paulk91] is the result of a study conducted by the Software Engineering Institute (SEI) on software process improvements. One of the most important results of this study was the fact that the inability to obtain
the expected productivity and quality standards was essentially due to the inability in managing the software process. In order to address this problem the CMM provides software organizations with guidance on how to gain control of their processes for developing and maintaining software and how to evolve toward a culture of software engineering and management excellence. It does this by presenting sets of recommended practices in a number of key process areas that have been shown to enhance software process capability. The CMM is based on knowledge acquired from software process assessments and extensive feedback from both industry and government. The CMM identifies five levels of maturity for software organizations, these are: Initial, Repeatable, Defined, Managed, and Optimizing. The progress from the Initial level to the Optimizing level leads the software organization to software engineering and management excellence. Each level establishes some structural, organizational, and behavioral properties that must be achieved and that are fundamental to move to the next level.

The CMM has often been criticized to be overly bureaucratic more focused on the process rather than on the substance. Due to excessive structuring it seems to be well suited for only bureaucratic organizations such as government agencies, large corporations and regulated monopolies. An attempt to improve the CMM is the Capability Maturity Model Integration (CMMI) [Chrissis03]. The goal of the CMMI project is to improve usability of maturity models for software engineering and other disciplines, by integrating many different models into one framework. Nonetheless, these models failed to diffuse widely and more successful commercial solutions – like the Rational Unified Process – are currently adopted.

2.1.4 Rational Unified Process (RUP)

The Rational Unified Process (RUP) [Kruchten03] is an iterative software development process framework created in 1998 by the Rational Software Corporation that is, since 2002, a division of IBM which released the latest version in 2005. As happened for the CMM the architects of RUP started to investigate on the causes of software projects failure and the outcome of this study was a system of software best practices they named
the *Rational Unified Process*. Differently from CMM and CMMI the process was designed with the same techniques used to design software – that are UML – and it has underlying object-oriented model. The RUP is a software product itself and it is maintained like a product: it is accessible through the web, it presents a modular structure, and it is characterized by continuous releases and integrations. This new conception of software process model makes the RUP suited for very different software organizations and never obsolete.

The *Rational Unified Process* captures many of modern software development’s best practices in a form suitable for a wide range of projects and organizations: *develop software iteratively, manage requirements, use component-based architectures, visually model software, continuously verify software quality, and control changes to software*. It models a software process along two dimensions: the horizontal dimension and the vertical dimension. The first (horizontal) dimension represents time and shows the lifecycle aspects of the process as it unfolds; it represents the *dynamic aspect* of the process expressed in terms of cycles, phases, iterations, and milestones. In the RUP, a software product is designed and built in a succession of incremental iterations. This allows testing and validation of design ideas, as well as risk mitigation, to occur earlier in the lifecycle. The second (vertical) dimension represents core process disciplines (or workflows), which logically group software engineering activities by their nature; it represents the *static aspect* of the process described in terms of process components: activities, disciplines, artifacts, and roles.

The RUP provides a rigorous yet flexible approach to model software processes and this is one of its key elements. Nowadays it is used by more than thousand companies, which are specialized in different application domains in both large and small projects.

### 2.2. Agile Software Development

The agile software development movement officially started on February 13, 2001 at The Lodge at Snowbird ski resort in the Wasatch Mountains of Utah. In that occasion seventeen experienced and recognized software development gurus met to talk and
discuss about some more flexible and effective approaches to software development practices adopted by them so far usually referred as light methodologies. The outcome of the meeting was the Manifesto of Agile Software Development that was signed by all the participants. It states the following:

\textit{We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:}

\begin{itemize}
    \item \textbf{Individuals and interactions} over processes and tools
    \item \textbf{Working software} over comprehensive documentation
    \item \textbf{Customer collaboration} over contract negotiation
    \item \textbf{Responding to change} over following a plan
\end{itemize}

\textit{That is, while there is value in the items on the right, we value the items on the left more [Agile01].}

The manifesto puts the emphasis on the elements that, according to the professional experience of the authors, have more value in leading a software project to its successful completion. The whole feeling behind it was a reaction to the established methodologies and practices that revealed to be too much structured, more focused on the process than on the result. The adjective agile was used to identify this new approach: agility is the capacity of moving quickly but decisively, of reacting to changing situations with speed and grace, of changing direction while maintaining the balance and the poise. In the same manner the new approach to software development should be more able to cope with changes rather than being stuck into a plan, based on the active collaboration with the final user rather than on negotiating contracts with customers and focused on delivering a result rather than producing useless documentation. In order to obtain this result the authors of the manifesto did not devise a comprehensive methodology but recommended some good practices that have demonstrated to be useful and effective. These methodologies basically cover team organization, software life cycle and development process, design issues, software testing, and relationship with the customer. They are explored in what can be considered the pillars of \textit{Agile Software Development: Dynamic}
Section 1 – Current Trends in Software Development

- Systems Development Method (DSDM) [Stapleton95], Extreme Programming (XP) [Beck00], Scrum [Schwaber06], Crystal Methods [Cockburn02], Feature Driven Design (FDD) [Palmer02], Adaptive Systems Development [Highsmith00], and Lean Software Development [Poppendieck03].

The Agile movement focuses on people, relationships, and uncertainty and the word ecosystem [Highsmith02] – rather than methodology – better characterizes its nature. According to the American Heritage Dictionary an ecosystem is defined as:

“organisms and their environment: a localized group of interdependent organisms together with the environment that they inhabit and depend on”

The Oxford English Dictionary extends this definition to include a constant interchange within the system, including both organic and inorganic elements. The word ecosystem conjures up a vision of living things and their interactions with each other. Within an organizational context, an ecosystem can then be thought of as a dynamic, ever-changing environment in which people and organizations constantly initiate actions and respond to each other’s actions. The word ecosystem focuses us on the dynamic interactions of individuals and teams rather than on the static lines on organization charts. This is the context in which agile practices like XP, Scrum, Crystal Methods, and the others give the best results. Such context is characterized by accelerated time schedule combined with significant risk and uncertainty that generate constant change during the project.

2.3. Model Driven Development

Model Driven Development (MDD) [Hailpern06] is a technique for addressing complex development challenges by dealing with complexity through abstraction: complex systems are modeled at different levels of specificity; as the development program proceeds, the model undergoes a series of transformations, with each transformation adding levels of specificity and detail.
The key elements of MDD are models, artifacts, and relationships, and transformations. A model is a representation of a system – or a portion of a system – from a specific point of view; it is defined by the collection of elements which are normally connected and interdependent. An artifact is a meaningful subset of the model according to some definition of meaningful and relationships institute mappings between artifacts in different models. Transformations are systematic modifications of a model and its set of affected relationships and capture the real essence of Model Driven Development. Almost all the activities of the software development process are performed through transformations: the system is implemented from specifications by subsequent model transformations; transformations can refine an abstract specification into a more detailed one or vice versa; different types of translation can be used to provide different implementations of the same model or to define a mapping between implementations; thanks to the formal specification of models translations can be automated. Moreover, traceability of artifacts is an inherent property of the system since at any level each artifact can be expressed as the chain of relationships required to obtain it from the initial model. This is one of the core values of MDD: it easily allows understanding the role of each component in the system at any stage, its function in the overall process, and provides a systematic way to implement changes.

Model Driven Architecture (MDA) [Kleppe03] is the most know framework supporting MDD. In order to support enterprise software development and deployment the Object Management Group (OMG) created a conceptual framework that separates business-oriented decisions from platform decisions to allow greater flexibility when architecting and evolving these systems. This framework along with the standards that help realizing it is what the OMG calls Model Driven Architecture; it is based on well established OMG standards as Meta Object Factory (MOF), Unified Modeling Language (UML), XML Metadata Interchange (XMI) and Common Warehouse Meta-model (CWM) and can be implemented on all the most important enterprise platforms (CORBA, J2EE, .NET and web-based platforms). MDA uses UML to describe models whose formal definition – the meta-model
– is expressed through MOF. MDA provides two different visions of a systems trough the Platform Independent Model (PIM) and the Platform Specific Model (PSM). The PIMs provide formal specifications of the structure and function of the system that abstracts away technical details. How the functionality specified in a PIM is realized is specified in a platform-specific way in the PSM, which is derived from the PIM via some transformation. Transformations between models allow refinement of the model at the major level of detail or its abstraction to a more general view. The clear separation between the specification of system and its realization allows its implementation on different platform with traceability either between models and models and realizations or between different realizations.

The main goal of MDD is then to provide a way to cope with complexity, to maintain a flexible and well structured knowledge of the system through abstractions, and to simplify the process of implementing changes. Despite the general acceptance of the approach, some criticism concerns redundancy (there are multiple representations of the same thing), potential round trip problems (due to the difficulty to keep synchronized the model with the code), and the suspect that sometimes it just moves the complexity elsewhere rather than throwing it away.

3. Software Computing Models

While software development methodologies describe processes leading to the deployment of software applications from the analysis of requirements, software computing models provides abstractions and rules defining how to structure software and organize computation. Software computing models and software development methodologies are complementary. The best software computing model we can use to solve a problem looses its effectiveness if it is not supported by a methodology that actually allows implementing it. On the other hand, the most effective methodology becomes useless if software deployed is based on a wrong computing model.
3.1. From Paradigms to Computing Models

The early computing model where mostly programming paradigms. The first organic approach to code structuring and organization was structured programming. This paradigm was introduced by Edgar Dijkstra [Dijkstra68, Dijkstra72] in 1968 as a reaction to the contemporary programming style. Structured programming puts the emphasis on control structures which define a rigid and foreseeable execution flow. Structured programming introduced the concept of invariant subsequently revisited by Bertrand Meyer with the design by contract [Meyer97]. Invariants are conditions that can be attached to the code in correspondence of control structures and establish properties which should be always satisfied for a proper execution. Invariants are useful to check the correctness of the implementations of a given algorithm. The subsequent evolution has been imperative programming which has been the reference paradigm first of the Algol [Naur63] and then the Pascal [Wirth71] programming languages. Imperative programming modeled a software application as a collection of functions or routines which perform specific tasks. The composition of routines – through serial or nested calls – defines the overall organization of a software application and allows the construction of complex software systems. It is interesting to investigate on the use of the term imperative. It recalls the famous expression divide et impera which defines a technique used to write programs from specifications: the problem is divided into smaller ones and algorithms and routines are designed to solve them. As the complexity of software systems increased new paradigms were required and objects replaced routines.

The object-oriented paradigm is basically composed by three disciplines which cover the major phases of the software life-cycle: object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP). The object-oriented paradigm represents

---

8 This expression identifies a technique used by the Romans to govern the Empire. The overall land was divided into small provinces headed by a consul. The consul had full powers in the province and was directly responsible to the emperor.
and implements a software system as a set of interacting objects. Objects have a state defined by the value of their attributes and a behavior defined by the methods (routines) they define. The emphasis has moved from the tasks to perform (routines) to the domain which defines the collection and characterization of the objects in the system. The domain represents the connection with the real system and the software system and its modeling is crucial for the success of software projects. Even if the first formalization of the object oriented paradigm appeared in 1967 with the Simula language [Nygaard81], it developed and become widely used in the last two decades of the century with the availability of professional object oriented programming languages like C++ and Java. Today object-oriented programming is the most widely used programming paradigm and more complex abstractions – like software components [Syzpersky02] – are built on this paradigm.

This brief summary is an example of what pointed out by Mary Shaw [Shaw96]: in the evolution described before we can notice how the main issue addressed by the programming paradigms shifted from the simple algorithm to the properties of software systems. The latest computing models have made a step further since they reason about global properties of software and software systems. Thanks to the advances in software engineering problems are solved and more complex ones, due to the increased dimensions of software systems, arise. These changes are normally followed by the introduction of new paradigms and computing models. The next two sections will explore two major trends arose in the last five years that are: Autonomic Computing and Recovery Oriented Computing.

### 3.2. Imitating Biology: Autonomic Computing and Beyond

#### 3.2.1 The Rationale Behind Autonomic Computing

Autonomic Computing is a concept bringing together many fields of Computer Science with the purpose of creating computing systems which are self-adaptive. The term Autonomic
Computing was first used by IBM in 2001 [Horn01] to describe computing systems that are said to be self-managing [Kephart03]. By the term self-managing four fundamental properties of the systems were identified: self-configuring, self-optimizing, self-healing, and self-protecting. Later on, this definition has been extended in order to include additional, even if secondary, features and autonomic systems were simply identified as self-* systems. Computing systems have reached a level of complexity where the human effort required getting the systems up and running and keeping them operational is getting out of hand. Hence Autonomic Computing tries to cope with this problem by introducing into software systems self-management properties. In order to instrument computing systems with this property it imitates biology and takes as a reference model the nervous system. The human nervous system is the master controller of the body and regulates its functions without conscious awareness and involvement. For example, it takes care of unconscious reflexes that are all those bodily adjustments that do not require our attention such as the size of the pupil, the digestive functions of the stomach and intestines, the rate and depth of respiration and dilatation or constriction of the blood vessels. In the same manner an autonomic software system takes care of itself and embed facilities that enable itself to tune their performance according to its state and the environment, to detect internal failures and fix them, and to change and evolve its structure according to the context.

The reason why the autonomic option could be a possible solution can be expressed by quoting from Kephart and Chess [Kephart03]:

The term autonomic computing is emblematic of a vast and somewhat tangled hierarchy of natural self-governing systems, many of which consist of myriad interacting, self-governing components that in turn comprise large numbers of interacting, autonomous, self-governing components at the next level down.

\footnote{In this case the asterisk refers to everything; that means every property that can be imagined. A self-* system is then self-adaptive, self-managing, ...}
The enormous range in scale, starting with molecular machines within cells and extending to human markets, societies, and the entire world socio-economy, mirrors that of computing systems, which run from individual devices to the entire Internet. Thus, we believe it will be profitable to seek inspiration in the self-governance of social and economic systems as well as purely biological ones.

The comparison between nature and software systems is definitely appropriated. As Kephart pointed out [Kephart03]:

The difficulty of managing today’s computing systems goes well beyond the administration of individual software environments. The need to integrate several heterogeneous environments into corporate-wide computing systems, and to extend that beyond company boundaries into the Internet, introduces new levels of complexity.... As systems become more interconnected and diverse, architects are less able to anticipate and design interactions among components, leaving such issues to be dealt with at runtime. Soon systems will become too massive and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge. And there will be no way to make timely, decisive responses to the rapid stream of changing and conflicting demands.

Hence a dynamic, reactive, and pro-active self-managing computing system is required.

3.2.2 Autonomic Computing Reference Model

The vision promoted by IBM defines an autonomic system as self-managing that means:

- **Self-configuring.** An autonomic computing system configures itself according to high-level goals, i.e. by specifying what is desired, not necessarily how to accomplish it. This can mean being able to install and set itself up based on the needs of the platform and the user.

- **Self-optimizing.** An autonomic system should be able to detect performance degradation in system behaviors and intelligently perform self-optimization functions.
• **Self-healing.** An autonomic system must be aware of potential problems and should have the ability to reconfigure itself to continue to function smoothly.

• **Self-monitoring.** An autonomic system must have the ability to dynamically adjust its resources based on its state and the state of its execution environment.

A fundamental requirement for autonomic systems to be self-managing is **self-awareness** [Hariri06]. Self-awareness means that an autonomic system knows itself and is aware of its state and its behaviors. Moreover **contextual awareness**, **reactivity** and **pro-activity** are implicitly derived from the previous list, whereas **openness** is a strongly desired feature.

In order to operatively deploy systems with the previously described properties IBM introduced the concept of **autonomic component** which represents the basic block of an autonomic system. Autonomic components are composed by two elements: the **managed element** and the **autonomic manager**. While the managed element basically represents the software component or (legacy) system that need to be managed, the core of the innovation resides in the autonomic manager which, like a regulator in a control system, controls the managed element. Figure 2 illustrates the structure of an autonomic component with a particular attention to the elements constituting the autonomic manager. In order to control the managed element, the autonomic manager interacts with it through a management interface that – by following the analogy with control systems – contains the **sensors** and the **actuators**. Inside the manager the five constituting elements implement the autonomic control loop which is normally referred as **MAPE-K**. The acronym stands for **Monitor, Analyze, Plan, Execute**, and **Knowledge** which respectively are the four activities performed by the manager and its knowledge base: while the activities can be pipelined, sequential, or concurrent all of them can access and modify the knowledge base of the manager, which often represents the architectural model of the managed element. The autonomic manager is a software component that ideally can be configured by human administrators through high-level goals and uses the monitored data from sensors and internal knowledge of the system to plan and execute, based on these high-level goals, the low-level actions that are necessary to achieve these goals.
The structure previously described provides just a reference since autonomic managers can be hierarchically composed or manage more than one element. If necessary, autonomic managers can cooperate in order to satisfy a system level goal or maintain a global property. This cooperation allows the system evolving according to changes in the environment and other requirements.

Parashar, Hariri et al. [Hariri06] proposed a slightly different architecture for autonomic systems based on two nested control loops: a global control loop controls the long term goals and system wide properties, while the single MAPE-K loops represent the local control loops.

3.2.3 Emerging Key Applications of Autonomic Computing

After the publishing of the manifesto of *Autonomic Computing* this computing model has been applied and some key areas have emerged as interesting scenarios for autonomic
systems. These research fields range from power management, data centers, Grid Computing clusters, and Ubiquitous Computing.

In the case of power management and ubiquitous computing great advantages can be obtained in applying the autonomic paradigm to the management of sensor networks. Khargharia et al. [Khargaria06] demonstrated that it is possible to save great amount of power by using autonomic management systems. Autonomic routing is another interesting application of autonomic computing to sensor networks: the challenges in this field are represented by the development of dynamic routing algorithms providing a low consumption of power and a relatively fast delivery. Self-awareness, context-awareness, and self-optimization can be profitably employed to design this kind of algorithms.

Autonomic techniques can be profitably applied even in the case of the data centers, clusters and computing grids: these are essentially wide-area high-performance heterogeneous distributed clusters of computers used to run anything from scientific to business applications for many differing users. The complexity is not only given by the difficulty in maintaining such a geographically distributed system, but in addition it is expected to provide an agreed quality of service (QoS); which could be formalized in a Service Level Agreement (SLA). These systems are intrinsically dynamic: system load and uptimes have to be continuously monitored and taken into account while allocating and using resources. This is an interesting challenge for Autonomic Computing which has been mostly used for dynamic resource management and system administration and some practical implementation have been deployed: the AutoMate project [Argawal03] is an interesting work in this direction.

3.2.4 One Step Beyond: Conscientious Software and Other Biological Models

The last edition of OOPSLA conference hosted the first workshop on Ultra-Large Scale (ULS) systems. According to Northrop et al. [Northrop05], ULS systems will be the future challenge of software engineering and the workshop was a first attempt to express ideas,
to investigate the state of the art, and present some preliminary work. By following the path opened by Autonomic Computing other biological models have been taken as examples to inspire new computing models and software systems. I found particularly interesting the notions of conscientious software [Gabriel06] and of commensalistic software system [Fleissner06] which are directly inspired by existing biological systems.

The term conscientious software has been introduced by Richard P. Gabriel and Ron Goldman and identifies the need for software to...

...grow up and become responsible for itself and its own future by participating in its own installation and customization, maintaining its own health, and adapting itself to new circumstances, new users, and new uses [Gabriel06].

In order to achieve this goal the authors propose a new computing model based on the combination of two elements: an autopoietic and allopoietic system. The authors expect that programming systems will become as living which use feedback, internal visibility, and continual noticing as part of their self-generating nature: many of the feedback loops in living systems are involved with the regulation and the production of components, proteins, and other biochemical material regulating, among the other things, the regulators just mentioned. This condition identifies the concept of autopoiesis. Autopoietic systems are systems that are defined as unities, as networks of productions of components, that recursively through their interactions, generate and realize the network that produces them and constitute, in the space in which they exist, the boundaries of the network as components that participate in the realization of the network [Maturana81].

Conversely, allopoietic systems are systems that produce something other than themselves. They are expression of allopoiesis which is, in other words, the abstraction of the manufacturing process. The majority of our software systems can be considered allopoietic but when a complex, programmed system needs to live for a long time, living becomes the ultimate goal. This
is a fundamental property of autopoiesis. Hence the challenge of conscientious software is about structuring a system which needs to recursively generate, realize, and produce itself as well as correctly produce something than itself that is the reason why they have conceived. Different solutions, based on completely autopoietic or allopoietic systems have been presented but it has been noticed that models and tools valuable to express and implement autopoiesis are not effective to implement allopoiesis and vice versa. For this reason the only possible solution is to devise a system that comprises the two properties and that clearly separates the tasks between the two components. As it normally happens in virtual-machine based program, the garbage collector which is responsible of managing the program memory is not part of the program itself and resides into the virtual machine core but it is needed from the managed application to run properly.

Another interesting notion borrowed from nature is the one of commensalistic symbiosis [Paracer00] that is a biological relationship between a symbiont, a species that is vulnerable or unable to survive on its own, and a host species with an effective self-preservation and defensive systems. Whereas the host species is neither advantaged nor disadvantaged by the relationship, the symbiont benefits from the host. This is the kind of relation among clown fishes and anemones which respectively play the roles of symbiont and host species: the clown fishes are immune to the poisonous tentacles of anemones and by living among them they are protected from potential predators. By recalling the work of Gabriel and Goldman, Fleissner and Beniassad propose a commensalistic software system [Fleissner06] that is a self-managing software architecture based on commensalistic symbiosis. The system consists of host components, symbiont components, and a nervous system. The autopoietic part of the system is comprised of host components and the nervous system, which are implemented in a specialized robust programming language. Symbiont components are allopoietic and produce a computational result or provide a service to users. Any general-purpose programming language can be used for their implementation. Since the symbiont components are fragile, they establish a symbiotic relationship with autopoietic host components. Host components are attached to the nervous system and are designed to
keep themselves along with any associated symbiont alive and healthy. This association is commensalistic, since the fragile symbiont components benefit, while the host components are unaffected. The systems proposed by Fleissner and Beniassad fundamentally preserves the structure devised by Gabriel and Goldman but put a strong emphasis on the internal organization of the participating components and introduces the concept of nervous systems that is the interface through which host components and symbionts interact.

These models are promising abstractions for shaping the next generation of software systems and lay their foundations on autonomic computing since they are self-managing systems.

### 3.3. Handling Failures: Recovery Oriented Computing

#### 3.3.1 Recovering Instead of Preventing

Recovery Oriented Computing (ROC) is a computing model explored by Patterson et al [Patterson02] which focuses on designing and implementing software systems able to quickly recover from failures without interrupting the service they are providing or by maximizing the system availability. According their inventors it can be described as follows:

*Recovery Oriented Computing (ROC) takes the perspective that hardware faults, software bugs, and operator errors are facts to be coped with, not problems to be solved. By concentrating on Mean Time to Repair (MTTR) rather than Mean Time to Failure (MTTF), ROC reduces recovery time and thus offers higher availability [Patterson02].*

As clearly pointed out by Northrop et al. [Northrop05] failures are inevitable in large, complex, and always running software systems. For this reason instead of trying to implement failure safe systems, Patterson et al. concentrated on the system ability to quickly recover from failures. Failures constitute a large portion of system administration problems, for this reason ROC not only promotes a more feasible computing model but
also helps reducing the total cost of ownership and allows deploying more robust software systems. As in the case of Autonomic Computing, Recovery Oriented Computing is focused on enforcing a global property of software systems and in order to face a practical problem. Moreover, ROC is fundamental requirement to design and to implement dependable software systems: in order to trust services delivered and computing results we have to be reasonably sure that failures will be properly handled. As pointed out by Fox and Patterson [Fox05] ROC also serves as a prerequisite for Autonomic Computing:

Although autonomic computing sets forth worthy goals, substantial work remains on more fundamental issues before it can truly succeed. We must fully understand problems and solutions before we can automate them, and the first steps must be to develop new tools that work hand-in-hand with operators, letting them bring their expertise to bear using metaphors and actions that match their view of system operation rather than the system’s underlying organization.

The substantial work on ROC has been produced by The Berkeley/Stanford Recovery-Oriented Computing (ROC) Project headed by David Patterson and Armando Fox. The outcome of their work has resulted into the definition of six ROC techniques along with real case studies demonstrating their effectiveness. These techniques have been devised by looking at other fields (disaster management, human error studies, and civil engineering) and which are examined in the following paragraphs,

3.3.2 Software Recovery Experiments

Recovery experiments cause failure in order to test their recovery properties. They are needed not only in the development lab but also in the field in order to collect information when faults occur in a given systems with its unique combination of hardware, software, and firmware. Software recovery experiments made by using FIG (Fault Injection in Glibc) – a lightweight, extensible tool for triggering and logging error at the application/system boundary – demonstrated that even mature, reliable programs
have misdocumented interfaces and poor error recovery mechanisms. By using FIG four successful practices for recovery have been noticed:

- **Resource Pre-allocation.** Resource pre-allocation moves failures due to lack of resources into the initialization phase of the systems. It has been noticed that sometimes encountering a failure at program initialization seems more desirable than doing so in the middle, when the system will is more likely to be disrupted by abnormal termination of an operation.

- **Graceful Degradation.** Techniques that offer partial service in the face of failures allow the service to degrade gracefully. Such techniques, whereby errors lead to reduced functionality than outright failures, postpone downtime until an operator can fully recover the system.

- **Selective Retry.** Selective retry is the process of trying again those actions which led to failure by choosing them according to a specific selection criterion. Some errors could be transient; in this case selective retry – with a reasonable delay – can solve the problem. The successful application of this technique resides in the ability of identifying the severity of the error.

- **Process Pools.** Systems using process pools can better handle failures because they just restart the processes which encountered failures without bringing down the entire system.

The techniques previously examined by studying the behavior of some professional applications (Apache HTTPD Server, MySQL Server, ...) when failure is injected into the system.

### 3.3.3 Automatic Diagnosis

Automatic diagnosis is the process which led a system to automatically detect the failure and eventually help the operator to fix the problem. Automatic diagnosis reduces the MTTR because the diagnosis is automatically performed by the system either as a
routine or when failure occurs: the data collected by the diagnosis is ready to be used by the operator which saves time in fixing the problem.

There are different methods to instrument a software system with this feature: classical error determination starts by making an accurate model of the system. Symptoms are recorded and a variety of statistical techniques are used to identify a suspect behavior or component. This kind of approach presents two drawbacks: first, many systems change frequently and it can be difficult – and really computational expensive – to maintain an update model especially for large systems; second, this kind of models normally captures the logical structure of the system and not their dynamic properties. It is possible to adopt a different approach which does not create a model of the systems but simply track changes of its structure over time, traces and reports problems, and, when failures occur, by using data-mining techniques identifies the origin of the problem. This approach has been implemented with Pinpoint that automates diagnosis for the J2EE middleware: it has been demonstrated that the overhead of tracing is generally contained and appropriate parameter setting can contain the number of false positives and discover a good number of hits.

3.3.4 Fine Grained Partitioning and Recursive Restartability

Fine grained partitioning is the act of partitioning the hardware components of the system – or its software modules – in way that allows them to be isolated from the remaining structure. Once isolated, these elements can be fixed to recover from failure. Isolation allows partial restart of the system and reduced downtimes. Fine grained partitioning facilitates graceful degradation of systems in case of failures: whereas system reboots reclaim all system resources and returns software to its initial state, partial restart maintains the system running with degraded performance. By allowing partial restart at multiple hierarchical levels it is possible to obtain an incredibly flexible software infrastructure which is able to isolate the smallest component of the systems encompassing the failure and restart it. This property is identified as recursive restartability, and experiments with the Mercury satellite ground station showed that a
fine grained partitioned and recursively restartable infrastructure allows dramatically reducing the MTTR.

### 3.3.5 Reversible Systems for Operators

In most cases the origin of faults reside into the work of operators hence it is desired to instrument the system with the capability of rolling back to a specific state, correct the mistake done by the operator, and recover back with modified state. In other words is needed is a reversible system which basically provide the undo operation for operators and an intelligent redo. To support retroactive repair and recovery from operator error, Patterson et al. introduced a three step undo process called “the three R’s” : Rewind, Repair, and Replay which basically implements the behavior described before. In the rewind step the system state is reverted to its contents before the error occurred. In the repair step the operator can perform any action he/she thinks being useful to fix the error (installing patches, correcting the bug, ...). Finally, in the replay step, the undo system re-executes all user interactions with the system, reprocessing them with the changes made during the repair step. This technique has been tested with a departmental undoable email system and encouraging results have been obtained.

### 3.3.6 Defense in Depth

Defense in depth implies that independent modules can provide backup defense that can improve reliability [Fox00]. By providing backup defense each module is able to quickly recover from failure to a valid state with the minimum loss of data. Defense in depth techniques lead to more robust systems that do not expose a single point of failure, and this property increase system reliability. This technique is normally applied into military devices but also in computer systems.

### 3.3.7 Redundancy

Redundancy is a common technique to cope with failure. By providing additional components it is more difficult to make a specific service completely unavailable: thanks
to the redundant architecture, as soon as a specific component stops operating properly the system can exclude it and route request to other instances of the same component.

3.3.8 Conclusions

Recovery Oriented Computing provides techniques which allow computing systems to quickly recover from failure in order to improve dependability and to lower the total cost of ownership. Mistakes made by people, software bugs, hardware failures are facts and not problems to solve, because they just happen. For this reason ROC techniques does not prevent errors from happening but provide ways to cope with them which are inevitable.

4. Challenges for the Future: Ultra-Large Scale Systems

Since when the term Software Engineering was coined, we have witnessed a constant increase in complexity and size of software systems. The advent of the Internet and its broad diffusion opened new possibilities and introduced new challenges for software engineers; it contributed the spread of distributed systems which play today a predominant role in Information Technology (IT). The advances in Ubiquitous Computing have injected software applications into any kind of device, and large interconnected systems whereby the number of components is unknown and continuously change are becoming an integral part of our life. Hence, the expectations for the future are characterized by a continuous progress in this direction.

In June 2006 the Software Engineering Institute (SEI)\(^{10}\) completed a study commissioned by the U.S. Department of Defense (DoD) whose subject was to investigate on the

\(^{10}\) The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense located at the Carnegie Mellon University.
properties of the next generation software systems and to provide suggestions about the research roadmap to follow in order to support these systems. DoD has the goal of *information dominance* – to achieve and exploit superior collection, fusion, analysis, and use of information to meet mission objectives – and it is always been anticipatory of the trends of mainstream software engineering. For this reason it is worth looking at the result of this study.

Northrop et al. [Northrop06] characterize the next generation software systems as increasingly complex systems composed by thousands of platforms, sensors, and decision nodes connected through heterogeneous wired and wireless networks.

*These systems will push far beyond the size of today’s systems and systems of systems by every measure: number of lines of code; number of people employing the system for different purposes; amount of data stored, accessed, manipulated, and refined; number of connections and interdependencies among software components; and number of hardware elements. They will be ultra-large scale (ULS) systems [Northrop06].*

The sheer scale, the high degree of interconnection, and the strong interaction with the human counterpart are the most important element of these systems:

*ULS systems will necessarily be decentralized in a variety of ways, developed and used by a wide variety of stakeholders with conflicting needs, evolving continuously, and constructed from heterogeneous parts. People will not just be users of a ULS system; they will be elements of the system. Software and hardware failures will be the norm rather than the exception. The acquisition of a ULS system will be simultaneous with its operation and will require new methods for control. ... Consequently, ULS systems will place unprecedented demands on software acquisition, production, deployment, management, documentation, usage, and evolution practices [Northrop05].*

Northrop et al. observed that the requirements of these systems are unlikely to be addressed adequately by incremental research within established categories: new
conceptions of the nature of these systems and new ideas how to develop them need to be conceived.

We will need to look at them differently, not just as systems or systems of systems, but as socio-technical ecosystems [Northrop05].

The new conception poses new challenges. The study produced by the SEI identified some key areas – and the related technologies – to explore in order to start research for ULS. These areas comprise:

- Human Interaction.
- Computational Emergence.
- Design.
- Computational Engineering.
- Adaptive System Infrastructure.
- Adaptable and Predictable System Quality
- Policy, Acquisition, and Management.

Whereas current research fields can provide useful elements to deal with these systems, the overall approach need to be completely reviewed and cannot be just an evolution of the central engineering approach:

The basic premise underlying the research agenda presented in this document is that beyond certain complexity thresholds, a traditional centralized engineering perspective is no longer adequate nor can it be the primary means by which ultra-complex systems are made real. Electrical and water systems are engineered, but cities are not—although their forms are regulated by both natural and imposed constraints. Firms are engineered, but the overall structure of the economy is not—although it is regulated. Ecosystems exhibit high degrees of complexity and organization, but not through engineering. The protocols on which the Internet is based were engineered, but the Web as a whole was not
engineered—although its form is constrained by both natural and artificial regulations. In this report, we take the position that the advances needed for ULS systems require a change in perspective, from the satisfaction of requirements through traditional, rational, top-down engineering to their satisfaction by the regulation of complex, decentralized systems [Northrop05].

The picture that better characterize the nature of ultra-large scale systems is that of a city:

Designing and building most of today’s large systems can be compared to designing and constructing a single, large building or a single infrastructure system (such as for power or water distribution). In contrast, ULS systems will operate at levels of complexity more similar to cities. At first it might seem that designing and building a city is simply a matter of designing and building a large number of buildings. However, cities are not conceived or built by individual organizations, but rather by the actions of many individuals acting locally over time. The form of a city is not defined in advance by specifying requirements; rather, a city emerges and changes over time through the loosely coordinated and regulated actions of many individuals. The factors that enable cities to be successful, then, include both extensive infrastructures not present in individual buildings as well as mechanisms that regulate local actions to maintain coherence without central control. These mechanisms include government organizations and policies, city planning, streets and transportation systems, communication and emergency services, and distribution of food and consumer goods, to name a few. Moreover, it is not feasible to design and build a city in one attempt. People, companies, communities, and organizations decide to build parts of cities for their own purposes. Cities grow and thrive based on cultural and economic necessities, and, although some aspects of a city are designed and constructed in a local context, most elements that make up the essence of a city arise from more global policies and mechanisms, such as zoning laws, building codes, and economic incentives designed to encourage certain sorts of growth and construction [Northrop05].

This picture, along with the vision lying behind it, is one of the best expressions of the change in trends of the last years. This change is mostly due the increase in complexity and in scale of software systems, but also witnesses their deep penetration into the society.
5. Observations

The brief review presented in this chapter does not aim to be a complete analysis of the evolution of Software Engineering in the last fifty years; neither wants to provide an in-depth study of the state of the art. The main objective of this introduction is to communicate to the reader how the discipline of Software Engineering evolved and to highlight the most important causes of this change. This brief story is functional to understand the nature of software systems at present times and to explain which kind of problems have to be addressed in providing software support for them.

To this aim, some otherwise important methodologies, programming paradigms, and phenomena not functional to the discussion, have been omitted or simply cited. In particular, functional programming and logic programming as well as the impact of the open source phenomenon were not considered. A complete trace of the evolution of Software Engineering can be found by chronologically looking at the proceedings of the editions of ICSE\textsuperscript{11}, ICSM\textsuperscript{12}, and OOPSLA\textsuperscript{13} which are the leading conferences in this field. Finally, I also deliberately avoided to discuss about the agent oriented technology and its role in this scenario. This issue will be the main subject of the next chapter and of the remaining part the thesis.

\textsuperscript{11} International Conference on Software Engineering.

\textsuperscript{12} International Conference on Software Maintenance.

\textsuperscript{13} Object Oriented Programming Languages and Applications.
1. Introduction

Agent-oriented technology encompasses abstractions, methodologies, and tools which characterize the agent-oriented approach and allow implementing software solutions where the organization and the conceptual units reflect the fundamental elements of the agent-oriented mindset. In order to apply the agent technology it is important to be familiar with the concepts of agent and multi-agent system, which are the pillars of the theory of agents. By using such abstractions we cast the problem domain into a collection of heterogeneous interacting agents – that is an agent-based design of the system – and we implement it into agent-based system that is normally represented by one or more multi-agent systems.

In this chapter I will introduce the concepts of software agent and multi-agent system, discuss their features, and provide references for a complete characterization of the subject. A brief overview of the practical applications of such technology will conclude the chapter.
2. The Agent Abstraction

2.1. The Notion of Agent

The agent concept is the basic building block of the agent oriented mindset. Agents are the conceptual units in which problems are decomposed and upon which systems are built. Despite its importance, there is no agreement about a universal accepted definition of what an agent really is [Franklin96]. Nonetheless, some properties are widely recognized to be fundamental for its characterization. Russel and Norvig [Russel95a] identified in the interaction with the environment a distinguishing feature of agents:

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.

A more precise, yet general, definition considers autonomy, situatedness, and goal-oriented behavior to be distinctive properties of agents:

A system situated within an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future [Franklin96].

An agent is a computational system which is long-lived, has goals, sensors and effectors, decides autonomously which actions to take in the current situation to maximize progress towards its (time-varying) goals [Maes97].

Patty Maes [Maes97] enriches the previous definition by contextualizing it in the case of software agents:

A software agent is a particular type of agent, inhabiting computers & networks, assisting users with computers-based tasks.

Wooldridge and Jennings [Wooldridge95] point out that, even though it can refer to either a hardware-based or software-based computer system, the term agent is usually
adopted to identify the latter. They, then, introduce the concept of intelligent agent which is:

A computer system situated in some environment that is capable of flexible autonomous action in order to meet its design objectives [Wooldridge97].

The concepts of autonomy and situatedness are, then, essential to define software agents. Autonomy characterizes agents as able to act without the direct intervention of humans (or other agents) and having direct control over its own actions and internal state. Situatedness implies that agents have a sensorial activity within the environment and that have the capability to perform actions able to change the environment state. Moreover, Wooldridge et al. introduced the concept of flexibility; this is the most interesting feature because it allows separating the concept of agent, as intelligent entity. There are many examples of systems that are already situated and autonomous: any control system, which must monitor an environment performing actions to modify it as conditions change (examples range from a very simple thermostat to a very complex reactor control system); software daemons, which monitor a software environment performing actions to modify it as conditions change (an example is the UNIX xbiff program monitoring incoming e-mails). Such examples cannot be considered intelligent software agent because they do not exhibit high-level skills [Shoam91], in other words, they are not able to expose a flexible behavior. According to Wooldridge et al. a flexible behavior is characterized by the three following properties: reactivity, pro-activity, and social ability.

Reactivity implies that:

Agents should perceive their environment and respond in a timely fashion to change that occur in it in order to satisfy their design objectives [Wooldridge99].

Pro-activity implies that:
Agents should not simply act in response to their environment; they should be able to exhibit opportunistic, goal-directed behavior and take the initiative where appropriate [Wooldridge99].

Social ability implies that:

Agents should be able to interact, when appropriate, with other artificial agents and humans in order to complete their own problem solving and to help others with their activities [Wooldridge99].

Social ability focuses the attention on another important element of the agent oriented mindset: interactions. They are expressed in three different forms: cooperation, competition, and negotiation. By using these politics agents perform their activity and coordinate by themselves. The vision matured so far corresponds to what Wooldridge and Jennings call the weak notion of agent [Wooldridge95]. They also identified a strong notion of agent which characterize an agent as

a computer system that, in addition to having the properties identified above, is either conceptualized or implemented using concepts that more usually applied to humans [Wooldridge95].

Researchers following this direction characterize agents using mentalistic notions, such as knowledge, belief, intention, and obligation [Shoam91]. Some AI researchers went further by considering emotional agents [Bates92, Bates94]. Moreover, other researchers emphasize other properties like adaptability – the ability to modify its behavior over time in response to changing environmental conditions or an increase in knowledge – and mobility – the ability to change its physical location to enhance its problem solving. Finally, radically different points of view can also be found. Nwana [Nwana96] suggested that the main properties of an agent should be identified in a set of three: autonomy, cooperation, and learning. He also provided a classification of agents into four different types: collaborative, interface, collaborative learning, and smart agents.
In the following I will consider only the weak notion of agent that is, among the other things, the least contentious one. Such notion will be the reference for the remaining of this thesis.

2.2. Agent Architectures

The notion of agent identifies the properties characterizing a software artifact as an agent, but does not provide any suggestion about how to design and, then, implement it. Agent architectures, on the other hand, provide software engineers with a formal specification, or a design model, that can be taken as starting point for implementing software agents.

In literature it is possible to find different definitions of agent architectures. According to Pattie Maes, an agent architecture can be described as follows:

[A] particular methodology for building [agents]. It specifies how... the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact. The total set of modules and their interactions has to provide an answer to the question of how the sensor data and the current internal state of the agent determine the actions... and future internal state of the agent. An architecture encompasses techniques and algorithms that support this methodology [Maes91].

Kaelbling gives a slightly different vision but with the same substantial meaning:

[A] specific collection of software (or hardware) modules, typically designated by boxes with arrows indicating the data and the control flow among the modules. A more abstract view of an architecture is as a general methodology for designing particular modular decompositions for particular tasks [Kaelbling91].

Jennings and Wooldridge [Wooldridge95] provide a more practical definition:

We consider the issues surrounding the construction of computer systems that satisfy the properties specified by agent theorists. This is the area of agent architectures.
All the previous definitions put the emphasis on the fact that an agent architecture provides a mean to create an agent. In particular, Maes and Kaelbling focus the attention on the fact that an architecture should define the internal organization of an agent and describe the process which determines its activity. This point of view, even if in a broad sense, is the same expressed by Jennings and Wooldridge. Hence, we can define agent architectures as computational models, whose implementations expose the characterizing properties of software agents.

Whereas it is possible to describe an agent through conceptual functions and provide a formal model able to express all the most diffused agent architectures, I will not use such approach. Conversely, I will briefly describe the features of these architectures by pointing out the practical advantages and drawbacks in adopting them. For readers interested in a global, unifying, and formal model I suggest [Wooldridge99] as a reference.

### 2.2.1. Deliberative Architectures

Deliberative Architectures constitute the classical AI approach for building software agents. They consist in modeling a software agent as a particular type of knowledge based system. Wooldridge and Jennings characterize a deliberative agent architecture as follows:

*We define a deliberative agent or agent architecture to be one that contains an explicitly represented, symbolic model of the world, and which decisions (for example about what actions to perform) are made via logical (or at least pseudo-logical) reasoning, based on pattern marching and symbolic manipulation [Wooldridge95].*

Deliberative architectures maintain a symbolic representation of the environment, and the evolution of the agent behavior is controlled by some automatic process (logical deduction or planning). Such vision has been strongly inspired by AI planning which is the sub-field of AI concerning itself on what to do – that is which sequence of actions or plans to devise and apply – in order to obtain a goal.
Deliberative architectures allow agents to maintain a state. These agents have some internal data structures that are used to record information about the environment. The benefits of this architecture are the elegant realization and the accurate semantics. On the other hand some drawbacks are evidenced in very computational difficulties, complexity in representing the perceptive information in symbolic knowledge, and complexity in managing dynamic environments.

2.2.2. Reactive Architectures

Reactive architectures have been the second landmark in the field of agent architectures. They basically developed as a reaction against the unattended premises of deliberative architectures in handling real-world problems. As a result, reactive architectures avoid the use of an explicit symbolic model representing the environment, but the agent course of action is determined in response to some external stimulus.

Perhaps, the most famous architecture embodying these principles is the subsumption architecture [Brooks86, Brooks91a, and Brooks91b]. A subsumption architecture agent is a collection of tasks accomplishing behaviors. Each behavior is a finite state machine that continually maps perceptual input to action output. In some implemented versions of the subsumption architecture, this mapping is achieved via situation → action rules, which simply determine an action to perform on the basis of the agent’s current state. Each behavior generates suggestions with respect to which action to perform; the overall decision about which action to perform is determined by interactions between the behaviors.

The subsumption architecture is very easy to understand, to develop, and it has been successfully applied in real scenarios. Despite this, some drawbacks limit its applicability: it requires a great amount of information about the environment, it is quite impossible to take decisions based on non-local information, it is rather difficult to engineer agents to fulfil a specific task, and it is very difficult to build up learning systems.
2.2.3. Hybrid or Layered Architectures

Hybrid architectures combine aspects of either deliberative architectures or reactive ones. Typically, these architectures are organized into software layers. The layers may be arranged vertically (so that only one layer has access to the agent’s sensors and effectors) or horizontally (so that all layers have access to sensor input and action output) as shown in Figure 3.

![Figure 3. Layered Agent Architectures](image)

In the horizontal layering the software layers are directly connected to the sensory input and action output, and managed in a parallel way. Actually, each layer acts like an agent itself, producing suggestions like what action to perform. In general, these kinds of architectures need a mediator to solve the conflicts among the software layers. The advantages in using this kind of agent architecture can be evidenced in the simplicity in programming several agent behaviors. On the other hand, the difficulties are evidenced in the implementation of a reliable mediator (with \( N \) levels performing \( M \) actions each the mediator has to manage \( M \times N \) interactions).

In the vertical layering, the agent activities are managed by distinct layers. There are two types of vertical layer architecture: one-pass control and two-pass control. The distinction between them resides in the information and control flow. One-pass control
architectures transfer the information and the control data from the bottom level, where the events occurring in the environment are caught, through the top level, where the data is provided to the effectors. Two-pass control architectures transfer the information from the bottom to the top, whereas it moves control data in the opposite direction, that is, from the top to the bottom.

2.2.4. Practical Reasoning and Belief-Desire-Intention (BDI) Architectures

Practical reasoning architectures are modeled on, or inspired by, the theory of practical reasoning that is the kind of pragmatic reasoning that used by humans to decide what to do. Practical reasoning has long been an area of study by philosophers, who are interested in developing theories that can account for human behavior. Typically, such theories make use of a folk psychology, whereby human behavior is understood by the attribution of attitudes such as beliefs, desires, intentions, and so on. Human behavior can be thought of as arising through the interaction of such attitudes. Practical reasoning architectures are modeled on theories of such interactions.

The Belief-Desire-Intention (BDI) [Rao95] architecture is probably the most known and the most influential among practical reasoning architectures. BDI agents are characterised by a mental state with three components: beliefs, desires, and intentions. Intuitively, beliefs correspond to information that the agent has about its environment. Desires represent options available to the agent — different possible states of affairs that the agent may choose to commit to. Intentions represent states of affairs that the agent has chosen and has committed resources to. An agent’s practical reasoning involves repeatedly updating beliefs from information in the environment, deciding which options are available, filtering these options to determine new intentions, and acting on the basis of these intentions.

The advantages in using this kind of agent architecture can be evidenced in an intuitive model and a fine functional distribution. In contrast, the main drawback can be found in the complex realization of efficient systems.
2.2.5. Considerations

The BDI architecture and the hybrid architectures are the most adopted or mimicked agent architectures. While deliberative and reactive architectures propose a pure approach, the other ones demonstrated to be more flexible and then, more adaptable to different contexts.

It can also be observed that, even though it is possible to find full BDI architecture compliant implementations – like AgentSpeak [Weerasooriya95] and Jadex [Braubach05] – many agent programming frameworks, like the one presented in this thesis, propose more general models supporting the concept of multi-behavioral activity. Such models are general enough to implement different agent architectures but require major effort for the software engineer who has to actually implement the desired infrastructure.

2.3. Comparison: Agents, Objects, and the Others

The introduction of the agent abstraction can raise doubts about its necessity. The perplexity mostly springs from considering the agent concept to be just a specific case of other programming abstractions such as objects, components, or expert systems. On the other hand, as previously seen, the concept itself of software agent lacks of a precise, complete, and formal definition; it has been characterized in terms of qualities this entity should have. As demonstrated by Franklin and Graesser [Franklin96], there is a wide variety on the properties that should be ascribed to agents, and sometimes no agreement on, at least, a subset of them. In this dissertation I decided to adopt the definition given by Jennings and Wooldridge [Wooldridge97] that is one of the most widely used. Even in this case people often fail to recognize the peculiarity of the agent concept because of the implementations of the agent abstractions hardly succeed in expressing through software those qualities – for example pro-activeness, social ability, and, above all, intelligence – which characterize it. Hence, when looking at practical examples of what a software agent is, the general feeling is to consider them just something nothing much different from the specific abstraction used to implement it. According to the specific technology adopted to implement it, a software agent has been considered nothing that
different from an object or an expert system. In this section I will briefly summarize what makes a software agent different from these abstractions.

2.3.1. Agents vs. Objects

The confusion between agents and objects is probably the most diffused and the most difficult one to eliminate. This is probably due to the fact that at present time object oriented technology is the most preferred technology for multi-agent systems. Hence, multi-agent systems are often seen as distributed concurrent object-oriented applications, and agents as complex, and sometimes defined active, objects.

In order to clarify the distinction we first focus on the essence of objects. Objects are defined as computational entities that encapsulate some state, are able to perform actions on this state through methods, and communicate by message passing. State, methods, and message passing are then the fundamental elements characterizing the object entity. The first observation that can be done concerns autonomy: object encapsulate a state and modify it according to the policies programmed into their methods. Hence, they have control over their state. One of the fundamental properties of object-oriented programming is encapsulation, which denotes that the state of an object is not directly accessible but it is subject to some sort of access control through methods. Encapsulation preserves state of objects from corruption and it is necessary because objects are passive by nature: it is not possible for an object to avoid method execution upon a correctly compiled method call. Then, it is not possible for an object to avoid undesired values for its state unless they provide a controlled access through it. For these reason objects do not have control over their behavior. Indeed, this is what agents do: interactions among agents do not occur through method calls but using a request-decide-(eventually reply) pattern. This means that agent request actions to be performed and command them. A passage from Wooldridge and Ciancarini [Wooldridge01] is illuminating on this issue:

The locus of control with respect to the decision about whether to execute an action is thus different in agent and object systems. In the object-oriented case, the decision lies with the object that invokes the
method. In the agent case, the decision lies with the agent that receives the request. This distinction between objects and agents has been nicely summarized in the following slogan: Objects do it for free; agents do it because they want to.

Moreover, the additional properties ascribed to agents – pro-activeness, reactivity, and social ability – are completely absent in the characterization of an object. Systems of interacting objects which do not expose these properties are still to be considered object-oriented systems. This is not the case of agent-based systems, since, according to the given definition, these are fundamental properties of the agent abstraction. Finally, objects can be considered as passive entities: if we consider an object oriented systems the number of instances performing an operation at any given time corresponds to the number of the active threads running (that is hopefully by far less than the number of objects). Within an agent-oriented system each agent has its own thread of control and is reasonably active during all its life-cycle: it senses the environment, performs the selected action, and reacts to events.

Agents are assumed to be continually active, and typically are engaged in an infinite loop of observing their environment, updating their internal state, and selecting and executing an action to perform. In contrast, objects are assumed to be quiescent for most of the time, becoming active only when another object requires their services by dint of method invocation [Wooldridge01].

Perhaps, the most similar abstraction coming from the object-oriented mindset is the concept of active object, proposed by Agha [Agha86]:

An active object is one that encompasses its own thread of control [. . . ]. Active objects are generally autonomous, meaning that they can exhibit some behavior without being operated upon by another object. Passive objects, on the other hand, can only undergo a state change when explicitly acted upon. [Booch93, p.91]

Thus, active objects are essentially agents that do not necessarily have to expose a flexible behavior.
2.3.2. Agent vs. Expert Systems

Expert systems were the AI technology of the 1980s [Expert83]. An expert system is a software system that is capable of solving problems or giving advices in some knowledge-rich domain [Jackson86]. It is possible to identify two main components within an expert system: the knowledge base which defines the information the system maintains about the domain and an engine which by reasoning on the knowledge base provides results to the user.

The major distinction between agents and expert systems is that expert systems are inherently disembodied: they do not have a model of the environment nor have to interact with it. Whereas agents strictly interact with the environment in order to perform their activities and get information from it, expert systems maintain all the information they need internally within the knowledge-base. Moreover, expert systems are normally stand-alone systems: they are not generally required to be capable of co-operating with other software components.

Despite these differences, some expert systems, (particularly those that perform real-time control tasks), look very much like agents. A good example is ARCHON [Jennings96b], which started life as a collection of expert systems, and ended up being viewed as a multi-agent system. ARCHON operates in the domain of industrial process control.

3. The Environment Abstraction

3.1. Characterization

If we recall the definition given by Wooldridge and Jennings [Wooldridge97] of software agent we notice that the environment is an essential component of the agent oriented mindset, because it actually is a requirement for agent situatedness:
An agent is an encapsulated computer system situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives [Wooldridge97].

Moreover, we can observe that agents are defined as reactive where reactivity identifies the ability to perceive the environment and respond in a timely fashion to the changes occurring in it. Hence, the vision that emerges characterizes the environment as a dynamic entity, which can influence the course of action of agents. Then, if we adopt the weak notion of agency it is necessary to model the environment as a primary abstraction in the agent-oriented mindset. Despite its importance, the community of researchers started to consider the environment as a primary abstraction only in the last years. As pointed out by Weyns et al. [Weyns04] the perception of environment has always been characterized by persistent confusion:

The confusion on what the environment comprises is mainly caused by mixing up concepts and infrastructure. Sometimes, researchers refer to the environment as the logical entity of a MAS in which the agents and other objects/resources are embedded. Sometimes, the notion of environment is used to refer to the software infrastructure on which the MAS is executed. Sometimes, environment even refers to the underlying hardware infrastructure on which the MAS runs.

In this section, I will briefly review the different characterizations proposed for the environment and I will present the latest results of the research in this field. For a complete survey and a detailed discussion on this issue Weyns et al. [Weyns04] is one of the most complete references.

### 3.2. Environment Models

As a consequence of the definition of agent, there is a generally acknowledged relationship between the agent and the environment:

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [Russel95a].
This model mimics the relation which binds a closed loop regulator (the agent) and the controlled physical system (the environment) that we may have in control theory. Russel and Norvig [Russel95a] discuss a number of key properties of environments that are now adopted by most researchers in the domain:

- **Accessible vs. inaccessible.** In an accessible environment the agent can obtain complete, accurate, up-to-date information about the environment's state. Most moderately complex environments (including, for example, the everyday physical world and the Internet) are inaccessible. The more accessible an environment is, the simpler it is to build agents to operate in it.

- **Deterministic vs. non-deterministic.** A deterministic environment is one in which any action has a single guaranteed effect – there is no uncertainty about the state that will result from performing an action. The physical world can be regarded as non-deterministic to all intents and purposes. Non-deterministic environments present great problems for the agent designer.

- **Episodic vs. non-episodic.** In an episodic environment, the performance of an agent is dependent on a number of discrete episodes, with no link between the performances of an agent in different scenarios. An example of an episodic environment is a mail sorting system [Russel95b]. Episodic environments are simpler from the agent developer's perspective, the agent can decide what action to perform based only on the current episode – it does not need reasoning about the interactions between this and future episodes.

- **Static vs. dynamic.** A static environment can be assumed to remain unchanged except by the actions results performed by the agent. A dynamic environment has other processes operating on it, and it changes in ways beyond the agent's control. The physical world is a highly dynamic environment.

- **Discrete vs. continuous.** An environment is discrete if there is a fixed, finite number of actions and perceptions in it. Russel and Norvig [Russel02] used the chess
game as an example of a discrete environment and taxi driving as an example of a continuous one.

The most complex classes of environments are those that are inaccessible, non-deterministic, dynamic, and continuous. The first three properties of this list are properties typically occurring in MASs.

Ferber [Ferber99] studied the characterization of the environment thoroughly. According to his view an environment can either be represented as a single monolithic system, i.e. a centralized environment, or as a set of cells assembled in a network, i.e. a distributed environment. In a centralized environment, all the agents have access to the same structure. In a distributed environment, each cell behaves like a centralized environment in miniature. Moreover, Ferber distinguishes between generalized and specialized models for environments: a generalized model is independent of the kind of actions that can be performed by agents; a specialized model is characterized by a well-defined set of actions. Central to Ferber's model of an environment is the way in which actions are modeled. The action model of Ferber distinguishes between influences and reactions to influences. Influences come from inside the agents and are attempts to modify the course of events in the world. Reactions, which result in state changes, are produced by the environment by combining influences of all agents, given the local state of the environment and the laws of the world.

Odell et al. [Odell02] made a substantial work on environment modeling for multi-agent systems. According to their vision:

*an environment provides the conditions under which an entity (agent or object) exists*

They distinguish between the physical environment and the communication environment. The former provides the laws, rules, constraints, and policies that govern and support the physical existence of agents and objects. The latter provides (1) the principles and processes that govern and support the exchange of ideas, knowledge and information, and (2) the functions and structures that are commonly employed to exchange
communication, such as roles, groups, and the interaction protocols between roles and groups. In the proposed characterization, a great emphasis is put on the environment as an active entity. Parunak [Parunak97] defines the environment as a tuple \((\text{state}, \text{process})\): the \textit{state} is a set of values that completely define the environment, including the agents and objects within it; the \textit{process} characterizes the environment as an active entity that can change its state, independently of the actions of the embedded agents. Its primary purpose is to implement dynamism in the environment.

### 3.3. Interaction Facilities

In the case of \textit{aggregations of agents} – that actually are multi-agent systems – the environment can act as context for interactions and provide some sort of infrastructure supporting them. Parunak et al. [Parunak03] provided a complete taxonomy of interactions in multi-agent systems: interactions politics range from direct message flow to indirect interactions. In the case of direct message flow the most used forms are peer to peer or master-slave conversations. In this scenario the environment can provide services for message transfer. Indirect interactions comprise \textit{black-board systems} [Black88], \textit{tuple-based interaction models} [Galernter92], and \textit{stigmergy} [Grassé59]. Whereas the blackboard model does not seem to provide a useful interaction for multi-agent systems, infrastructure supporting \textit{tuple-based interaction models} [Mamei03 and Schelfthout04] and \textit{stigmergy} [Brueckner00 and Bonabeu98] have been investigated with more interest. Finally, the environment can also act as a \textit{coordination} and \textit{organization layer} between agents [Omicini04]: in this case the concepts of \textit{organization}, \textit{group}, and \textit{role} become important abstractions as the agent concept.

The environment is then an essential component of the agent oriented mindset and becomes even more important when we consider aggregations of agents. Today many research challenges about the role and the properties of the environment are open [Weyns04] and their outcomes provide a fundamental contribution to the modeling of complex software systems.
4. Multi-Agent Systems

4.1. The Notion of Multi-Agent System

According to Jennings [Jennings01b] interactions are one of the most important elements of the agent-oriented approach. Social-ability, that is the ability of interact with peers in order to meet the proper design objectives, naturally prompts the agents to organize in social structures, thus forming Multi-Agent Systems (MAS). Multi-Agent Systems are not just the simple aggregation of software agents, but they identify a more sophisticated concept comprising the technical and structural issues of a community of entities as well as the social ones. Hence, multi-agent systems are constituted by:

- the community of interacting agents;
- the software infrastructure supporting the life cycle of agents;

As pointed out by Jennings et al., multi-agent systems open the way to new challenges, because they create new opportunities for complex problem solving:

However, the multi-agent case — where the system is designed and implemented as several interacting agents — is arguably more general and more interesting from a software engineering standpoint. Multi-agent systems are ideally suited to representing problems that have multiple problem solving methods, multiple perspectives and/or multiple problem solving entities. Such systems have the traditional advantages of distributed and concurrent problem solving, but have the additional advantage of sophisticated patterns of interactions [Jennings98a].

The same attention on the flexibility for problem solving has been noticed by Weyns and Holvoet:

A multi-agent system provides the software to solve a problem by structuring the system as a number of interacting autonomous entities embedded in an environment in order to achieve the functional and quality requirements of the system [Weyns06].
The richness of interactions that can take place within multi-agent systems makes them the perfect candidates to model complex systems and constitute the real distinctive value of using them:

*It is the flexibility and high-level nature of these interactions which distinguishes multi-agent systems from other forms of software and which provides the underlying power of the paradigm* [Jennings98a].

Hence, for all these reason it is interesting to investigate with major details on the properties of multi-agent systems.

### 4.2. Properties of Multi-Agent Systems

From a structural and architectural point of view a multi-agent system can be defined as a distributed computing system which enjoys the following properties:

- **Distributed control.**
- **Decentralized data.**
- **Asynchronous computation.**
- **Openness and dynamicity.**
- **Lack of global knowledge**\(^{14}\).

From an engineering standpoint, this kind of software infrastructure exposes the following features:

- **Speed and efficiency.** Agents can operate asynchronously and in parallel thanks to the distributed nature of the MAS.

---

\(^{14}\) This means that each agent has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint.
• Robustness and reliability. Distributed control and decentralized data eliminate a single point of failure.

• Scalability and flexibility. The dynamic nature of multi-agent systems allows introducing and removing agents with the same or new goals. Moreover, multi-agents systems can join together in order to evolve into wider organizations. Such features allow multi-agent systems to scale well.

• Incremental development and reuse. The dynamic nature and the inherent openness of multi-agent organizations allow the incremental development of the system. Moreover, since agents are autonomous components they can be used in different systems or join other organizations.

• Reduced costs. The multi-agent system is decomposable into the single agents, which represent the basic building blocks of the agent oriented approach. Hence, for this reason it is possible to define the MAS by selecting the single agents; such selection can be performed according to the available (eventually economic) resources.

Finally, an interesting property of multi-agent systems is computational emergence possibly originating from the wide number of interactions between autonomous agents. This is an issue of great scientific relevance since computational emergence is a property that is gaining interest in the study of complex software systems.

4.3. Historical Perspective

The research about systems composed of multiple agents was carried out under the field of DAI. Historically, DAI has been divided in two main research areas: Distributed Problem Solving (DPS) and MAS [Bond88, Kraus97]. DPS considers how the work of solving a particular problem can be divided among several agents having common goals and common preferences (cooperative agents belong to these systems). A description of DPS systems can be found in [Smith83, Durfee88, and Shehory98]. Conversely, research in MAS was concerned with the behavior of a collection of possibly pre-existing
autonomous agents aiming at solving a given problem; in the case of MAS commonality of goals is not a requirement.

Most of the early research on multi-agent systems was focused on cooperative multi-agent interaction, which turned out to be the application of planning techniques in a MAS environment. The activity of planning should consider additional constraints such as the fact that agents could have competing behaviors with the planning activity. Some notably outcomes of these research include Functionally Accurate Model (FA/C) [Lesser91], Partial Global Planning [Durfee88], and various studies on explicit team modeling [Levesque90, Jennings93, Grosz96, Jennings95, Tambe97].

The subsequent step in multi-agent systems research saw a radical change in the interaction politics: self-interestedness replaced commonality of goals and negotiation substituted cooperation. Negotiation is the natural form of interaction among self-interested agents. It has been seen as method for coordination and conflict resolution, but has also been used as a metaphor for communication of plan changes, task allocation, or centralized resolution of constraint violations. Hence, as happened for the notion of agent, it became an ill-formed concept. Jennings et al. [Jennings98a] proposed a characterization of negotiation suitable for real world scenarios and involving the following properties: the presence of some form of conflict that must be resolved in a decentralized manner; self-interested agents; bounded rationality; incomplete information. Some outcomes of the studies on negotiation include the PERSUADER system [Sycara90], the Contract Net Protocol [Jennings96], the ADEPT system [Sandholm95], and the Bazaar negotiation model [Zeng97].

The advances in networking and the wide diffusion of the web characterized the Internet as a driving force for MAS research and development. Coordination became harder due to the complex and global topology of the net and localization and interoperation gained the interest of the community of researchers. This is the era of the middle agents [Kuokka96] which tried to address the problem of finding agents in open environments.
At the same time, some advances in agent communication languages [Mayfield96, Smith96, FIPA, and Searle69] tried to address the difficulties of interoperation, but with no remarkably results. Another critical issue investigated in this period is effective allocation of limited resources to multiple agents. Economics-based mechanisms have been adopted in MASs to address problems of resource allocation (the central theme of economic research) [Mullen96, Sandholm93, and Huberman95]. Economics-based approaches and market mechanisms in particular, became increasingly attractive to MAS researchers both because of the ready availability of underlying formal models, but also because of their potential applicability in Internet-based commerce.

At present, MASs are widely studied in both academic and industry laboratories – but they are not yet widely available in real-world applications. However, many studies foresee an evolution of MAS technologies and the extension of their use to the large part of computing based applications [Zambonelli04]. Researchers [Omicini04] foresee a continuous diffusion of the agent technology and its consequent integration with other heterogeneous research areas – such as wireless sensor networks and utility computing – will play a key role in establishing MAS as the fundamental paradigms for the engineering of complex artificial systems [Zambonelli04].

5. Applications

Software agents and multi-agent systems are – at present time – a well know programming paradigm and a sufficiently developed research field for engineering real world applications. Despite this, the adoption of agent technology in mainstream software development still has to come, even though in the last two decades we have been noticing a growing interest in it and its incremental diffusion in a variety of application domains: from small systems for persons to open, complex, mission critical systems for industrial applications. Jennings et al. [Jennings98b] and Omicini and Poggi [Omicini04] provide a useful overview of the application agent technology from the
beginning to present time. This section briefly summarizes the application domains in which agent technology has been successfully applied.

Industry has always been an important test-bed for agent technology since it represents the field where the MAS techniques were first experimented, and where they first showed their huge potential. Today, MASs are used for a number of different industrial applications: in particular, they are employed in application scenarios like process control [Jennings94], system diagnostics [Albert03], manufacturing [Parunak87] and network management [Bieszczad98], whose distributed nature easily falls within the reach of MAS techniques.

Information management [Decker97] has been another important application field since the early days of agent technology. In particular, the Internet has been described as an ideal domain for MASs, given its distributed nature and the sheer volume of information available that make the use of agents of great interest for searching and filtering the information [Klusch01]. Internet has also pushed the use of MAS technologies in the fields of commerce and business process management. In fact, before the spreading of the Internet, commerce and business process management were almost entirely driven by human interactions; humans decided when to buy goods, how much they were willing to pay, and so on. Today, electronic commerce and automated business processes have increasingly assumed a pivotal role in many organizations because they offers opportunities to significantly improve the way in which the many entities involved in the business process interact. In this scenario, MASs have been shown both to be suitable for the modeling and the design of business process management systems [Matos01] and to be amenable to work as key components for the automation of some or all the steps of these processes [Jennings96a].

The distributed nature of traffic and transport processes, along with the strong independence among the entities involved in such processes, have made MAS a key tool for the engineering of effective, real-world applications for both traffic management and transport logistics [David05]. Different applications have been already realized; in
particular, one of them – OASIS [Lucas92] – can be considered as the proof that MAS are the ideal means for building open, complex, mission-critical systems. OASIS is in fact a sophisticated agent-based air-traffic control system based on the BDI agent model, which was used with success at the Sydney airport in Australia.

Other interesting MAS applications can be found in the health care domain [Ardissono06]. In this field, agent technology has already been initially adopted to deal with many different kinds of problems: patient scheduling and management, senior and community care, medical information access and management, and decision support [Ciampolini02]. These studies show that MAS are likely to be the right solution to build up medical decision support systems [Hudson02], and to improve the coordination between the different professionals involved in the health care processes [Lanzola02].

The area of computers & law is another hot field for MAS [Law04]. There, on the one hand, MAS technologies are becoming essential tools for the application and integration of traditional AI techniques (such as expert systems, automated deduction, deontic logic) within computational systems supporting techno-legal activities [Ciampolini04]. On the other hand, the complexity of agent-based systems proposes novel issues to the analysis of the legislative framework, such as the possible law-abidingness of software agents.
1. What is Agent Oriented Software Engineering?

Agent Oriented Software Engineering (AOSE) applies agent oriented technologies and methodologies to the analysis, the design, and the development of software systems. It uses the concepts of software agent and multi-agent systems to model, implement, and deploy highly complex and evolvable software infrastructures.

AOSE [Jennings01a] is a relatively new discipline if compared to agent oriented programming whose origins are to be found in the field of Distributed Artificial Intelligence (DAI). AOSE started to gain interest among the academic community at the end of the century [Wooldridge97] and established as research field in 2000 when the first AOSE Workshop was held at the 22nd International Conference on Software Engineering (ICSE 2000) in
Limerick, Ireland. Since 2002 the workshop is a featured event of the *International Conference on Autonomous Agent & Multi-Agent Systems (AAMAS)* – that is the most important conference on agent oriented technology – and constantly reflects the current trends and the future innovations in AOSE.

Despite their initial origin software agents and multi-agent systems are essentially *Computer Science* products rather than outcomes of *Artificial Intelligence (AI)*: they can incorporate AI techniques but mostly develop into the form of software applications instead of deductive models or logic frameworks. This is the reason why – after a reasonable period of assessment – research on agent oriented technology turned into the study of a systematic, disciplined, quantifiable approach to analysis, design, development, and deployment of software systems that is *Software Engineering* [IEEE90]. This approach uses software agents and multi-agent systems as fundamental abstractions; consequently we use the expression *Agent Oriented Software Engineering* to refer to such approach. AOSE comprises, then, methodologies, tools, processes supporting the software engineer in designing and quickly implementing multi-agent systems, which are the classic software products of this discipline. Since software agents are normally referred as intelligent entities or high level systems components it is possible to draw a quick comparison among AOSE and its closest ancestors: *Component Based Software Engineering (CBSE)* and Object Oriented System Development (OOSD).

As noticed on Figure 4, *Object Oriented System Development* uses the concept of *object* – along with the class defining its type – as the basic building block of software systems: objects along with their attributes and methods identify the elements through which the design of the domain is expressed and its entities are implemented. An object oriented system is then a net of interconnected objects by the means of *composition* and *inheritance*. 
Even if objects play a key role in OOSD the emphasis is concentrated on the domain which constitutes the starting point of the development process: the domain encloses all the elements of interest and the relationship among them; the correct identification of the domain greatly simplifies the design phase – in which objects and methods are normally identified with nouns and verbs meaningful in the domain – and improves the implementation stage. Component Based Software Engineering [Sizpersky02] provides a higher level abstraction: software components identify units of independent deployment and versioning, they are system elements offering a predefined service and able to communicate with other components. Software systems are made up by gluing different components that provide the services required in the system. The emphasis in this case is put on the services the components offer and their interfaces: services are the real value of components and interfaces describe how to use them. Agent Oriented Software Engineering makes a step further and provides a higher level of abstraction: software agents. Software agents are normally defined as autonomous software entities able to optimize the interaction with peers and with the environment in order to meet their goals. It can be noticed that the term software agent refers to something more complex than a software component and – at the same time – more powerful. Software systems are designed as communities of software agents which interact with each other in either a competitive or a collaborative manner. Interactions are essential within multi-agent systems: they are
the fundamental activity of software agents and are responsible of the overall system behavior.

Agent Oriented Software Engineering provides – if compared with OOSD and CBSE – a more expressive and flexible set of abstractions: since the emphasis is put on dynamically generated interactions, it supports software engineers in modeling and building system which are inherently designed to handle change. Today software systems are frequently characterized by changing conditions: system requirements, structure, and even goals; hence, agent oriented abstractions could be profitably applied in this context. In order to support this statement it is important to discuss a little more why software agents are an interesting technology for today’s software systems and how they contribute to build dynamic and complex systems.

2. The Nature of Complex Software Systems

Since from its first formalization, Agent Oriented Software Engineering aimed to provide agent oriented solutions for complex software systems. The first examples of this kind of systems are industrial strength software systems [Booch93]. The term industrial strength recalls the concept of something designed for high-volume and multi-user operation. These are scenarios requiring robustness and the presence of built-in safeguards against system failures. Then the expression industrial strength software is used to refer to any solid, sound program that has been thoroughly tested in live user environments for extensive periods, whether system software or application software. The term industrial strength is then the expression of a certain degree of robustness that software should have in order to handle complex environments as those that could be found in industry.

Grady Booch [Booch93], while characterizing industrial strength software, noticed that complexity seems to be an essential property of large software systems:
The distinguishing characteristic of industrial-strength software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all the subtleties of its design. Stated in blunt terms, the complexity of such systems exceeds the human intellectual capacity. Alas, this complexity we speak of seems to be an essential property of all large software systems. By essential we mean that we may master this complexity, but we can never make it go away.

Complexity was previously defined as an inherent property of all software systems\textsuperscript{15} by Brooks:

\begin{quote}
The complexity of software is an essential property, not an accidental one [Brooks\textsuperscript{87}].
\end{quote}

Since complexity is an essential property of software system it is important to understand where it comes from; Booch [Booch\textsuperscript{93}] identifies four elements which make software systems complex:

\begin{itemize}
\item The complexity of the problem domain. Some problems that software tries to address are complex by themselves, especially when competing and often contradictory requirements have to be satisfied. Moreover, since software products are mostly not to be used by their inventors, a considerable amount of complexity springs from the \textit{impedance mismatch} existing between system users and its developers: users and developers belong to different domains of expertise and it is often difficult for users to express their needs in a profitable way for developers. Finally, requirements changing during the software development process and which originated by the software project itself\textsuperscript{16} may imply additional complexity.
\end{itemize}

\textsuperscript{15} Even it there is no clear reference to large software systems the inherent complexity advocated by Brooks is characterizing complex software systems that are the subject of the article.

\textsuperscript{16} This kind of change in requirements is peculiar: it is not due to the fact that user needs or system features change, but it is an effect of prototyping. By using prototypes users can better comprehend the final result
• **The difficulty of managing the development process.** The large size of software systems is one of the biggest causes of troubles when managing the development process. Large software projects often involve software development teams which can be geographically dispersed. As the number of developers increases the more communication and coordination problems arise and maintaining a unity and integrity in design becomes a key challenge.

• **The flexibility possible through software.** Software offers the ultimate flexibility, so it is possible for a developer to express almost any kind of abstraction. This flexibility turns out to be an incredibly seductive property, however, because it also forces the developer to craft virtually all the primitive building blocks upon which these higher-level abstractions stand.

• **The problems of characterizing the behavior of discrete systems.** Within a large application, there may be hundreds or even thousands of variables as well as more than one thread of control. The entire collection of these variables, their current values, and the current address and calling stack of each process within the system constitute the present state of the application. Because we execute software on digital computers, we have a system with discrete states. Discrete systems by their very nature have a finite number of possible states; in large systems, there is a combinatorial explosion that makes this number very large. Each event external to a software system has the potential of placing that system in a new state, and furthermore, the mapping from state to state is not always deterministic. In the worst circumstances, an external event may corrupt the state of a system, because its designers failed to take into account certain interactions among events.

Whereas these issues can be discouraging, it is still possible to address the problem by first characterizing the properties of complex systems and then providing a useful and are able to better understand and articulate their needs; as a result they can ask for changes or additional features. At the same type prototyping helps developers to master the problem domain.
approach to handle them. Booch, summarizing the studies of Simon [Simon82], Rechun [Rechun92], Curtois [Curtois85] and Gall [Gall86], identifies the following five attributes of complex systems:

- Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached [Simon82].

- The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.

- Intra-component linkages are generally stronger than intercommoning linkages. This fact has the effect of separating the high-frequency dynamics of the components - involving the internal structure of the components - from the low-frequency dynamics - involving interaction among components [Simon82].

- Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements [Simon82].

- A complex system that works is invariably found to have evolved from a simple system that worked... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system [Gall86].

To be true, these are the properties that make complex systems manageable and distinguish organized complexity from disorganized (chaotic) complexity. According to Booch, only the first type of complexity can be managed with success in a software project. Therefore it is necessary to provide abstractions able to represent the multiple and different kinds of hierarchies of complex systems – that in the case of object oriented programming – are class structures and object structures:

Our experience is that the most successful complex software systems are those whose designs explicitly encompass a well-engineered class and object structure and whose structure embodies the five attributes of complex systems described in the previous section [Booch93].
What is worth noticing is that hierarchy, composition, categorization, and incremental development are fundamental properties for manageable complex systems. A decade later, Jennings et al. [Jennings01a] recognized these properties later to be still characterizing industrial strength software systems by paying more attention to the dynamics of interactions:

**Complexity frequently takes the form of a hierarchy.** That is, the system is composed of inter-related sub-systems, each of which is itself a hierarchy. The precise nature of the organizational relationships varies between sub-systems, although some generic forms (such as client-server, peer, and team) can be identified. Organizational relationships are not static: they can, and frequently do, vary over time.

According to the view proposed by Jennings more emphasis is put on the dynamic nature of the system, which can make evolve it along unpredictable paths:

*It is possible to distinguish between the interactions among sub-systems and the interactions within sub-systems. The latter are both more frequent (typically at least an order of magnitude more) and more predictable than the former. This gives rise to the view that complex systems are nearly decomposable. Thus, sub-systems can be treated almost as if they are independent of one another, but not quite since there are some interactions between them. Moreover, although many of these interactions can be predicted at design time, some cannot [Jennings01a].*

Whereas the general view of complex systems is the same, Jennings et al. do not stress the possibility of modeling contemporary complex systems with few different kinds of subsystems. This is indeed, an interesting issue: software systems are getting more and more interconnected each other, and openness is becoming more and more a desirable property of systems. Openness leads to heterogeneous systems, which are more hardly to be characterized by few different kinds of components. The advances in networking and computer technology have made possible the diffusion of software in every device and computing devices – and not only computers – have become a common feature of every day life. This implies that people are becoming not just a user of these systems but part of them thus forming a socio-technical ecosystem which, like a biological ecosystem, comprises a
dynamic community of interdependent and competing organisms (people, computing devices, and organizations) in a complex a changing environment [Northrop05].

Moreover, in the last years, evolution, emergent behavior, and adaptability have become fundamental properties for complex software systems [Bullock04] and the environment an essential element of their structure. Zambonelli and Parunak [Zambonelli03] identify four additional properties of complex software systems:

- **Situatedness.** Today’s computing systems have an explicit notion of the environment in which components are allocated and execute, they are affected in their execution by environmental characteristics, and their components often explicitly interact with that environment.

- **Openness.** Living in an environment, perceiving it, and being affected by it, intrinsically imply some sort of openness of a software system. The system can no longer be conceived as an isolated world, but must instead be considered as a permeable sub-system, whose boundaries permit reciprocal side-effects. Moreover, different software systems, independently designed and modeled, are likely to share the same environment and explicitly interact with each other.

- **Local control.** When software systems and components live and interact in an open context the concept of global execution control disappears. This trend is exacerbated by the fact that each independent system not only has its flow of control, but also may have components that are individually autonomous. As a result the system control is getting more and more decentralized.

- **Local interaction.** Local interaction – either geographically or logically – is enforced by the previous three properties. Autonomous components can interact with the environment in which they are immersed. In order to perform their activities
components need some form of context awareness; open software systems impose this context to be necessarily locally confined\textsuperscript{17}.

The overall picture that research and industry have today of complex systems is then the following: systems in which the structure, the interactions of components, the number entities, and the objectives are dynamic. Such dynamism along with the huge number of elements and the huge system size introduce new aspects: emergent behavior, evolution, and adaptability. These aspects are rather common in biological systems, which are normally conceived as complex systems due to their size and the richness of interaction among the huge number of their components. This is the reason why some biological models are becoming investigated in Computer Science.

While emergent behavior is a consequence, the ability to evolve and to adapt is something that can be instrumented in software systems and the next section will give some hints on how it can be possible to provide such feature through Agent Oriented Software Engineering.

3. The Weapons of AOSE

3.1. Fundamental Techniques to Tackle Complexity

The characterization of complex software systems has showed that it is possible to define a canonical view of these systems:

\textit{Complex systems are recursive and hierarchical structures. Such structures are organized according to the links connecting the components. Interactions change during time and take place among either}

\textsuperscript{17}In open and large software systems the context is dynamic as well as the components themselves. Hence, it is not possible to compute it statically; neither is possible to have complete view of it. For these reason, the information about the context need to be locally confined.
components of the same system or components located in different sub-systems. The frequency of interactions is higher if components belong to the same sub-system.

Given this view, software engineers have devised a number of fundamental techniques to manage these systems [Booch93]:

- **Decomposition.** Decomposition allows diving large problems into smaller and more manageable ones each of them can be handled with relative isolation.
- **Abstraction.** Abstraction allows defining simplified models of the system emphasizing some details while suppressing others.
- **Organization.** Organization allows defining the hierarchical infrastructures and the relations connecting the different problem-solving components. Organization is useful because allows selecting the desired level of detail while analyzing the system.

These techniques have been previously applied through an object oriented approach, but they are valuable in general and we will show how they can be naturally adopted by using an agent oriented mindset.

Jennings [Jennings01b] characterizes the agent-oriented mindset as follows:

The key abstraction models that define the agent-oriented mindset are agents, interactions, and organizations.

This suggests that adopting an agent-oriented approach to software engineering means decomposing the problem into multiple autonomous components that can act and interact in flexible ways to achieve their design objectives.

As pointed out in Section 2 of this chapter, there is no formal and universal definition of software agent, but the following characterization has been found useful by a wide number of researchers:
An agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives [Jennings01b].

When adopting an agent oriented approach it soon becomes clear that the wide range of features and elements of a complex system are better represented through a community of interacting agents, rather than a single agent. Such community is called multi-agent system and provides the environment in which they interact. Multi-agent systems are not only the result of agents' aggregation but normally provide agents with additional services supporting their activity. Given this, we can now see how these concepts can be powerful means for abstraction, decomposition, and organization.

3.2. Advantages of Agent-Oriented Abstractions

Abstractions play a key role in system modeling and become fundamental when systems are complex because they offer us a way harness their complexity. Abstractions provide a particular view of the system and establish a mapping between the entities of a model and the real components of the system. This mapping implicitly denotes a gap between the abstractions used to model the system and the system itself. When applied to software, abstractions are useful if they fully represent the problem and if they can be easily translated into a corresponding working implementation:

When designing software, the most powerful abstractions are those that minimize the semantic gap between the units of analysis that are intuitively used to conceptualize the problem and the constructs present in the solution paradigm [Jennings01b].

Hence, agent oriented abstractions become useful if they effectively represents all the characteristic features of complex systems and if it is possible to derive an implementation from them. Whereas the possibility of deriving an implementation depends on the tools supporting agent oriented programming – that normally are constituted by agent oriented programming frameworks – the effectiveness is something
specific to the abstraction itself. Then, we need to evaluate if agent oriented abstractions are “good lens” through which see a complex system.

According to the adopted canonical view, the elements that need to be characterized are sub-systems, sub-system components, interactions, and organizational (or hierarchical) relationships. Agent oriented abstractions can seamlessly model all these aspects:

- Sub-systems can be naturally modeled as agent organizations, which are multi-agent systems. They involve a number of constituent components interact according their role within the organizations.

- Sub-system components can be systems as well. In this case it is possible either to use a single agent to represent them or to model it with a community of agents; the decision strictly depends on the required level of granularity. Agents are flexible components and model either a system or a single component. If more detailed view of the system is required it is possible to represent it with an organization of agents: multi-agent systems can easily represent meta-organizations where communities of agents share the same infrastructure and the knowledge base.

- The interplay between the sub-systems and between their components is most naturally viewed in terms of high-level social interactions. While examining complex systems it is possible to find at any level of detail meaningful collections of objects that collaborate in order to achieve some higher-level view [Booch93]; such characterization is perfectly met by the interaction model proposed for software agents. Their activity is normally described in terms of cooperation to achieve a common objective and negotiation to resolve conflicts. These flexible ways of interacting, along with the possibility of competing for a give resource, allows modeling all the coordination and interaction strategies among components and systems.

- Other elements belonging to the agent oriented mindset, namely ontologies and interaction protocols, give a strong support in modeling dynamic interactions.
Thanks, to these properties agent oriented approach provides software engineers with powerful abstractions to represent and model complex systems. The characteristic flexible behavior exposed by software agents allow composing systems which are designed to embrace change either in the structure or in the interactions. Moreover, multi-agent systems seem to naturally support dynamic structure where the number of components is not defined a priori and change over time.

3.2.1. Addressing Decomposition

Complex systems can be intuitively decomposed by identifying the functions, the processes, and the actions that characterize it [Meyer97], then, it is rather natural to modularize the components of the system in terms of the objectives they achieve. Such decomposition, automatically leads to an organization whose components are identified by goal oriented software agents. Moreover, complex systems have multiple loci of control and agent based decomposition provides a system where individual components localize and encapsulate their own control. The inherent complexity of systems implies that it is impossible a priori to know about all potential links: interactions will occur at unpredictable times, for unpredictable reasons, and between unpredictable components. Agents are naturally equipped to engage dynamic interactions; hence, partitioning a complex system into socially able agents provides a natural supports for dynamic interactions. Agents interact through high-level agent communication languages, then, coupling becomes a knowledge level issue and decisions can be easily deferred to runtime without hard coding them. Ontologies, which provide a programmatic model of a domain of a discourse, can be really useful to support dynamic interactions.

We can briefly summarize the advantages of using agent-oriented decomposition:

- Decomposition by functions rather than objects and data.
- Natural support for multiple loci of control.
- Natural support for dynamic interactions.
Given this, it is worth considering an agent-based approach while decomposing complex systems. The advantages obtained by such approach are constituted by a more natural view of the system and by an increased manageability.

3.2.2. Supporting Hierarchy and Organization

Organizational structures are first-class entities in agent-systems – explicit representations are made of organizational relationships and structures. Moreover, agent-oriented systems have the concomitant computational mechanisms for flexibly forming and disbanding organizations [Jennings01b].

As stated by Jennings the concept of community and organizational structure are a fundamental element of the agent-oriented mindset. Moreover, agents are better characterized as socially able entities and they base their activity upon interaction with peers; this means that, as abstractions, are naturally designed to dynamic interactions. Agents offer a high-level abstraction upon which structuring a complex system, and hierarchy, as defined by Booch [Booch93], is an inherent property of multi-agent systems, which host a community of agents. The advantage of using an agent-oriented abstraction is mostly due to the fact that hierarchy is not statically defined but it is organized at run-time as a result of the activity of agents. Sen [Sen97] recognizes two basic contrasting patterns of interaction: cooperation and competition. We add negotiation to this list. These three different patterns of interactions allow the creation of extremely dynamic structures which can evolve during time according to the mutating goals of each agent and the global objective of the community. At the same time, agents can also organize in a hierarchy by imitating real life organizational structures. The same properties can be applied to systems where the components are represented by the single multi-agent systems: if a community of agents is designed and implemented to expose a shared global goal, then it can be easily modeled as a single entity interacting with the other peers and the same considerations made before for agents apply.

The agent oriented mindset provides abstractions that scale gracefully according to the number of the components and the size of the systems and they give a great flexibility in choosing the desired level of detail for each system component. Moreover, the dynamic
nature of multi-agent systems allows an incremental growth of the system which moves from an intermediate stable form to another; this is, according to Gall [Gall96], a fundamental requirement for the successful development of a complex system.

3.2.3. Supporting Evolution and Adaptability

A study committed by the UK Government’s Department of Trade and Industry [Bullock04] demonstrated that computational emergence is becoming a common feature of today software systems as a result of their heterogeneity and their high degree of interconnectivity. In general, we can observe that current trends in Software Engineering denote a growing interest in biological models, in particular self-organizing [Kephart03] and allopoietic - autopoietic systems [Gabriel06], as reference models for complex systems. Some biological systems expose properties – evolution and adaptability – that software engineers would include in their software systems in order to make them more manageable, or better, self-manageable. In particular, as discussed in Chapter 1, the self-* properties seem to be one of the most desired features for complex systems. Whereas the concept of self-* seems to be excessively demanding, it is possible to instrument complex systems with a subset of these properties. Autonomic systems are self-monitoring, self-healing, self-optimizing, self-configuring, and necessarily self-aware. These characteristics are the premises for system evolution and adaptability. Since evolution and adaptability are becoming requirements for software systems it is important to see if the agent oriented mindset provides some facilities in order to support them.

As noticed by Kephart and Chess [Kephart03], in order to become autonomic software systems need to be reactive and pro-active. Reactivity is the ability to cope with environmental changes in a timely fashion, while pro-activity identifies an anticipatory behavior [Hariri06] or, in other words, the ability to take the initiative:

...an autonomic system must be able to anticipate, to the extent that it can, its needs and behaviors and those of its context, and be able to manage itself proactively [Hariri06].
Reactivity and pro-activity are, according to Wooldridge and Jennings [Wooldridge95], two fundamental properties of software agents. These two characteristics, along with autonomy and social ability, identify what the authors define as intelligent agents, where the adjective intelligent refers to the ability of exposing a flexible behavior by means of the previously cited properties. Hence, the agent abstraction is a valuable starting point to model adaptable and evolvable components like an autonomic elements and considerable research has developed on this issue [Bigus02, DiMarzo06].

The flexible behavior of software agents is actually the most important property supporting system evolution and adaptability: thanks to the ability of interacting dynamically agents can change their course of action, plan new goals to achieve, and organize themselves differently. Such high degree of flexibility is partly due to the fact that agents interact at knowledge level; this means that not only we can defer the decision of the course of action at run-time, but also that we do not have to establish a statically, fixed, interaction protocol. By conferring such high degree of freedom to software agents we can obtain really flexible systems that naturally evolve and adapt to changes either in the environment or in the global objectives. Moreover, multi-agent systems are naturally conceived – and then implemented – as dynamic organizational structures; this means by adopting an agent oriented approach we can build systems where components can be easily added, migrated, or removed at run-time without downtimes. We can then change the systems while it is running by introducing new behaviors or changing the pre-existing ones; this facility has incredible value because it actually allows endowing a software system with evolution and adaptability. We can observe that whereas this seems an implementation issue, the major advantages derive from the abstraction: what really matters is the fact that, within the agent-oriented mindset, the abstraction of multi-agent system identifies a flexible infrastructure allowing dynamic interactions and discovery of services. Once we accepted this vision, the implementation of a real software infrastructure allowing this is just a coding task, and practical implementations of multi-agent systems demonstrate its feasibility.
The abstractions provided with the agent oriented mindset are then powerful enough to model adaptable and evolvable complex systems and they do it naturally. Moreover, the same properties allow multi-agent systems exposing an emergent behavior, which is another interesting aspect borrowed from biology and investigated today by computer scientists.

### 3.3. Observations

The agent oriented mindset provides powerful abstractions. These abstractions are naturally conceived to model today complex software systems. Despite its representative power, the agent oriented approach looses its effectiveness without specific methods, which drive engineers from the requirement to the complete system design and tools, which actually allow implementing this design. The alternative would be using an object-oriented technology that would inevitably loose the peculiarities – and the presented advantages – of the agent oriented mindset. In other words, agent-oriented methodologies and agent programming frameworks are needed to obtain the best results from the agent-oriented mindset. In the remaining of the chapter I will present a selection of the most important agent-oriented methodologies and a brief overview of the features of the most important agent programming framework. I will also introduce the FIPA Abstract Architecture which constitutes a useful reference model for multi-agent system development.

### 4. Agent-Oriented Methodologies

#### 4.1. What is a Methodology?

Agent-Oriented methodologies provide a set of mechanisms and models for developing agent-based systems. Most agent-oriented methodologies follow the approach of extending existing software engineering methodologies to include abstractions related to agents. Agent methodologies capture concepts like conversations, goals, believes, plans or autonomous behavior.
In this section I will show some of the most known and diffused agent-oriented methodologies. In particular I will describe: *Gaia*, *MaSE*, *SODA*, *Tropos*, *PASSI*, and *Prometheus*.

### 4.2. Gaia

*Gaia* [Wooldridge00] is a general methodology for agent-oriented analysis and design that supports both the micro-level (agent structure) and macro-level (agent society and organization structure) of agent development. The motivation behind *Gaia* is that existing methodologies fail to represent the autonomous and problem-solving nature of agents; they also fail to model agents’ ways of performing interactions and creating organizations. Using *Gaia*, software designers can systematically develop an implementation-ready design based on system requirements.

The first step in the *Gaia* analysis process is to find the roles in the system, and the second is to model interactions between the roles found. Roles consist of four attributes: *responsibilities*, *permissions*, *activities* and *protocols*. Responsibilities are of two types: *liveness properties* – the role has to add something good to the system – and *safety properties* – prevent and disallow that something bad happens to the system. Permissions represent what roles are allowed to do and which information they can access. Activities are tasks that a role performs without interacting with other roles. Protocols are the specific patterns of interaction, e.g. a seller role can support different auction protocols, e.g. “English auction”. *Gaia* has formal operators and templates for representing roles and their belonging attributes; it also has schemas that can be used for the representation of interactions.

In the *Gaia* design process, the first step is to map roles into agent types, and then to create the right number of agent instances of each type. The second step is to determine the services model needed to fulfill a role in one or several agents, and the final step to is create the acquaintance model for the representation of communication between the agents.
The main restrictions of *Gaia* are constituted by the inability to handle dynamic relationship among agents as well as dynamic capabilities. Hence, it is not well suited to model open and dynamic multi-agent systems, whereas it has been proven as a good approach for developing closed domain agent-systems. Moreover, organizations are not considered as primary abstractions and are elements implicitly modeled as consequence of role modeling. Extensions to the *Gaia* methodology have been proposed to cope with these lacks: Zambonelli, Jennings et al. [Zambonelli01] enhanced *Gaia* by introduced the concept of *organization* as a primary abstraction and provided support for open and dynamic domains such as the Internet; Gonzalez –Palacios and Luck [Palacios04] extended the work of Zambonelli et al. by introducing a framework of organizational patterns aimed to reduce development time and promotes reusability of design models.

### 4.3. *MaSE*

Wood and DeLoach [DeLoach00] suggest the *Multiagent Systems Engineering Methodology (MaSE)*. *MaSE* is similar to *Gaia* with respect to generality and the application domain supported, but in addition *MaSE* goes further regarding support for automatic code creation through the *MaSE* tool. The motivation behind *MaSE* is the current lack of proven methodology and industrial-strength toolkits for creating agent-based systems. Its goal is to lead the designer from the initial system specification to the implemented agent system. Domain restrictions of *MaSE* are similar to those of *Gaia*'s, but in addition it requires that agent-interactions are one-to-one and not multicast.

The *MaSE* methodology is divided into seven sections (phases) in a logical pipeline. *Capturing goals*, the first phase, transforms the initial system specification into a structured hierarchy of system goals. This is done by first identifying goals based on the initial system specification's requirements, and then ordering the goals according to importance in a structured and topically ordered hierarchy. *Applying Use Cases*, the second phase, creates use cases and sequence diagrams based on the initial system specification. Use cases present the logical interaction paths between various roles in and the system.
itself. Sequence diagrams are used to determine the minimum number of messages that have to be passed between roles in the system. The third phase is refining roles, it creates roles that are responsible for the goals defined in phase one. In general each goal is represented by one role, but a set of related goals may map to one role. Together with the roles a set of tasks is created, the tasks define how to solve goals related to the role. Tasks are defined as state diagrams. The fourth phase, creating agent classes, maps roles to agent classes in an agent class diagram. This diagram resembles object class diagrams, but the semantics of relationships is high-level conversation as opposed to the object class diagrams’ inheritance of structure. The fifth phase, constructing conversations, constructs a coordination protocol in the form of state diagrams that define the conversation state for interacting agents. In the sixth phase, assembling agent classes, the internal functionality of agent classes are created. Selected functionality is based on five different types of agent architectures: Belief-Desire-Intention (BDI), reactive, planning, knowledge based and user-defined architecture. The final phase, system design, create actual agent instances based on the agent classes, the final result is presented in a deployment diagram.

4.4. SODA

SODA (Societies in Open and Distributed Agent Spaces) [Omicini00], emphasizes the concept of agents inhabiting an environment which may be distributed, heterogeneous, dynamic, and unpredictable. In such an environment, agents live as a society. This methodology emphasizes more on inter-agent interactions rather than intra-agent communication. Since agents live within an environment, it is important to design them as components of a system. Agents should be thought of as “living dipped in societies” since their aggregate behavior cannot be understood outside their social structure. In the analysis phase, the SODA methodology produces three different models – the role model, the resource model and the interaction model. The role model presents the overall application goals broken down into tasks associated with roles and groups. The resource model outlines the available services. The interaction protocols are expressed as information required and provided by roles and the interaction rules drive the interaction within groups. The
SODA design phase is based on three strictly related models: the agent model – individual and social roles are mapped upon agent classes; the society model – groups are mapped onto societies of agents, which are designed and organized around coordination abstractions; the environment model – resources are mapped onto infrastructure classes, and associated to topological abstractions. In all of the previous models, the results of the SODA design phase are expressed in terms of agent classes, societies of agents, and infrastructure classes.

4.5. Thropos

Thropos [Giorgini04] is an agent-oriented methodology created by a group of authors from various universities in Canada, Italy, Belgium and Brazil. Tropos is based on the concepts of actor and goal and strongly focuses on early requirements. The development process in Tropos consists in five phases: Early Requirements, Late Requirements, Architectural Design, Detailed Design and Implementation.

The first phase identifies actors and goals represented by two different models. The actor diagram depicts involved roles and their relationships, called dependencies. These dependencies show how actors depend on each other to accomplish their goals, to execute their plans, and to supply their resources. The goal diagram shows the analysis of goals and plans regarding a specific actor in charge of achieving them. Goals and plans are analyzed based upon reasoning techniques such as means-end analysis, AND/OR decomposition, and contribution analysis. These models will be extended in the next phase, which models the systems within its environment. The goals can be decomposed into sub-goals.

The third phase is divided in three steps. In the first one, new actors, which are derived from the chosen architectural style, are included and described by an extended actor diagram. These actors fulfill non-functional requirements or support sub-goals identified in the previous phase. The second and third steps identify the capabilities, and group them to form agent types, respectively. The last step defines a set of agent types and
assigns each of them a set of capabilities. This assignment, which is not unique and depends on the designer, is captured in a table.

The Detailed Design phase deals with the detailed specification of the agents’ goals, belief and capabilities. Also communication among agents is specified in detail. This phase is usually strictly related to implementation choices since it is proposed within specific development platforms, and depends on the features of the adopted agent programming language. This step takes as input the specification resulting from the architectural design and generates a set of UML activity diagrams for representing capabilities and plans, and AUML [Odell00] sequence diagrams for characterizing agent interaction protocols. AUML is an extension of UML to accommodate the distinctive requirements of agent, which results from the cooperation established by FIPA and the OMG. This is achieved by introducing new classes of diagrams into UML such as interaction protocol diagrams and agent class diagrams.

Finally, the implementation phase follows the detailed design specification given in the previous phase. Tropos chooses a BDI platform for the implementation of agents, namely JACK Intelligent Agents [Coburn00], an agent-oriented development environment built on top of Java. The main language constructs provided by this platform (agents, capabilities, database relations, events and plans) have a direct correspondence with the notions used in the design phase. In addition, Tropos provides guidelines and heuristics for mapping Tropos concepts into BDI concepts and BDI concepts into JACK constructs.

4.6. PASSI

PASSI (Process for Agent Societies Specification and Implementation) [Chella04] is a step-by-step requirement-to-code methodology for designing and developing multi-agent societies integrating design models and concepts from both OO software engineering and artificial intelligence approaches using the UML notation. The models and phases of PASSI encompass anthropomorphic representation of system requirements, social viewpoint, solution architecture, code production and reuse, and deployment configuration
supporting mobility of agents. The design process with PASSI is supported by PTK (PASSI ToolKit), which is composed by an add-in for Rational Rose and a tool for reusing patterns of agents.

4.7. Prometheus

The Prometheus methodology [Padgham02] has been developed over several years in collaboration with Agent Oriented Software. The methodology has been taught at industry workshops and university courses. It has proven effective in assisting developers to design, document, and build agent systems. Prometheus differs from existing methodologies in that it is a detailed and complete (start to end) methodology for developing intelligent agents which has evolved out of industrial and pedagogical experience.

4.8. Other Methodologies

The review presented in this section is not exhaustive. There are other important agent-oriented methodologies which have been proposed and used by the community of researchers. In particular it is important to note: ADELFE [Bernon04], DESIRE [Brazier97], MESSAGE [Evans01], and MAS-CommonKADS [Iglesias98].

5. Agent Programming Frameworks

5.1. Definition of Framework

Designing and developing a software solution based on agent technology is a complex activity since it requires different skills and involves many tasks. First, it is important to identify the different kinds of agents which better represent the problem domain; then we need to carefully detail the roles played by the agents and describe the interactions among them. Once we have obtained a model of the system we need to implement it by using a set of libraries allowing us to define agents and create the multi-agent system. High level abstractions and automated tools integrated within professional IDEs can
help us developing multi-agent systems. Finally, after deploying the multi-agent system, we need to monitor its activity: in particular, graphical user interfaces can dramatically simplify the interaction with the MAS. All the tasks previously introduced are interrelated and concur in the development of a single project that is the multi-agent system.

A professional approach to the practice of multi-agent system development involves the adoption of tools which support the software engineer in all the previous tasks by enhancing his/her productivity and offering an integrated view of the entire process. This class of tools is normally referred with the term framework. According to Wikipedia\textsuperscript{18} a framework is:

\begin{quote}
\textit{[...] a defined support structure in which another software project can be organized and developed. […] a framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project.}
\end{quote}

In this section I will give a brief overview of the features of the most widely used agent programming frameworks, but before doing this, I will discuss about the Foundation of Intelligent Physical Agents (FIPA) [FIPA] which is an international organization promoting standards for the agent-technology. The value of FIPA resides in the fact that it provides a reference models for developers: these models are not standards but describe the technical features characterizing the abstraction of agent and multi-agent system from an implementation point of view. The FIPA reference models allow comparing the various tools and frameworks for the development of multi-agent systems available today.

\textsuperscript{18}http://www.wikipedia.org/
5.2. FIPA

The Foundation for Intelligent Physical Agents (FIPA) [FIPA] is an international association started in 1996 with the aim of providing reference models for the implementation of complex software systems by means of the agent technology. The most important objective of FIPA is to make possible, through the implementation of these models, a high degree of interoperability among the existing agent technologies. In order to promote the interoperability the FIPA recommendations – that are the documents through which FIPA promotes its reference models – mostly focus on the high level properties that software agents and multi-agent systems should have and on the general organization of the MAS architecture. The internal details of the implementation are beyond the scope of these recommendations. FIPA does not promote de-iure standards, but the recommendations are actually de-facto standards and are widely adopted community of developers.

The two most important documents released by FIPA are FIPA97 and FIPA2000 which describe the work done by the foundation. One of the most important outcomes of these two documents is the FIPA Abstract Architecture which describes the overall structure of an agent system.

![FIPA Reference Model for Agent-Based Systems](image)

The architecture is composed by five software blocks as described in Figure 5:
Applications. This layer comprises the agent oriented applications which interact with the remaining part of the architecture.

Abstract Architecture. It defines, at an abstract level, the elements that compose the agent system, by focusing on the interoperation among them.

Agent Communication Language. It defines the generic structure of the FIPA-ACL language, by specifying some among the most popular interaction protocols and knowledge representation languages.

Agent Management. It defines an infrastructure that can contain FIPA agents and allows their interoperation by specifying the fundamental components required for the agent management and communication.

Agent Message Transport. It defines the modalities used to transmit messages among agents by using the selected communication protocols.

FIPA provides a broader definition of software agent, if compared to those proposed by theorists: they are defined as computational entities embodying the communicative and autonomous properties of the application. The recommendation also requires that each software agent has to be described by an Agent Identifier (AID) whose is characterized by an immutable name identifying the agent, and a collection of key-value pairs representing additional properties.

Within the FIPA specification, software agents perform their activities within an agent platform (AP) which constitutes their run-time environment. The agent platform is a fundamental requirement for the agent system and constitutes a concrete implementation of the Abstract Architecture. The agent platform is defined by a collection of agent containers that, at a given instant of time, can only belong to a single platform. The agent platform is made up by the following building blocks:

- Software infrastructure supporting agents.
FIPA Agent Management Components. These components are responsible of the management of the entire software platform and of the agent hosted within it. They provide to the agents the basic services required to perform their activity: life-cycle management, localization, and communication.

Software agents.

Even though FIPA provide further details on the responsibilities of the management components it can be noticed the recommendation provides suggestions only about the general architecture of the system, without describing a specific process or selecting a specific technology. This is the strength and the weakness of FIPA because it is able to encompass a wide range of implementations but at the same time it does not actually allow the interoperation among different platforms.

5.3. Overview

In the last decade, the creation of software tools and environments supporting the development of agent oriented applications has been an active field of research. Featured and authoritative agent community web sites, such as AgentLink (www.agentlink.org), Multi-Agent Systems (www.multiagent.com), and Agent Portal (aose.ift.ulaval.ca), list a plethora of software tools, libraries, and toolkits to design, build, and deploy multi-agent systems. Among them just a few are still active and obtained a wide interest from the community of researchers or the industry field.

At now, much of the effort in supporting agent systems development is oriented to maintaining and extending with new features the most successful solutions. According to the previously given definition, not all of these projects can be considered frameworks. In this section we will briefly overlook those projects which, according to our opinion, are the most distinctive and important, then we will sketch a comparison with the work presented in this paper. Some important projects, like FIPA-OS [Poslad00], are not still maintained, I will focus my attention on those actually used and supported by the agent researchers community, in particular JADE, JACK, LEAP, and Agent Factory.
5.3.1. JADE

JADE [Bellifemine01] is one of the most known and used agent programming frameworks; the development of JADE has been started at CSELT (now TILab) and has been supported and extended (i.e. JADEX [Pokahr05]) by a wide community of researchers. The core of JADE is constituted by a FIPA compliant middleware hosting agents, a library of classes that developers can use or extend while creating agents, and a set of graphical tools for runtime monitoring. The agent model proposed by JADE is based on the concept of tasks constituting the activities performed by the agent. JADE agents are maintained in containers which can be easily managed and moved among different JVMs. The set of connected containers define a distributed JADE agent platform, which can be easily managed through the GUI delivered with the framework.

5.3.2. LEAP

LEAP [Berger01] is the synthesis of two earlier projects: JADE and ZEUS [Nwana99]. The JADE component delivers a FIPA-compliant agent platform constituting the target of LEAP agents, which can be easily developed with the tools shipped with the ZEUS component. Specifically, LEAP includes a re-factored version of the JADE code base that is compliant with J2ME and extends this code base to include the additional ZEUS functionality. Users can take advantages of the ZEUS development environment and deploy agents on a well established agent platform that is JADE.

5.3.3. JACK Intelligent Agents

JACK Intelligent Agents [Coburn00] is an industry strength product aiming to integrate the agent-oriented technology into large legacy software systems. JACK agents are designed as components that can interact with pre-existing object oriented systems. The main components of the framework are a set of syntactical additions to the Java language, along with a customized compiler, which simplify the development of JACK agents. Such extensions rely on a kernel of classes which constitute the run-time support to the generated code. Moreover, the plug-in architecture allows developers to extend the
framework with new features like support for additional agent models, different from the native BDI support, as well as new communication channel and language extensions.

5.3.4. Agent Factory

Agent Factory [Collier03] is a framework promoting a structured approach to the development and the deployment of agent oriented applications. It delivers extensive support for the creation of BDI agents and puts the emphasis on reusable agent designs which are the starting point to define new agents: through inheritance agent designs are extended with new features. The framework is organized into two core environments: the Agent Factory Development Environment and the Agent Factory Run-time Environment. The former delivers an integrated toolset giving support from design to deployment, while the latter provides facilities to deploy agent-oriented applications over a wide range of network-enabled Java compliant devices. The rich set of CASE tools integrated into the Agent Factory framework makes it an interesting solution for rapidly prototyping reusable agent components.

Along with the previously described frameworks there are many software projects which target specific applications or focus on a particular aspect of MASs: the Tracy Toolkit [Braun05] mainly addresses the support of mobile agents, while TuCSoN [Omicini98] allows defining MASs as a set of concurrent Prolog programs which interact and modify the environment constituted by artifacts (non agent, reactive software components).

6. Observations

Agent Oriented Software Engineering uses agent-oriented techniques to the modeling and the development of complex software systems. In this chapter I have clarified the motivations that justify the use of AOSE for the development of complex software systems. Agent Oriented Software Engineering uses the abstractions of agent, multi-agent system, interaction, and environment in order to provide a model of the systems that is manageable, flexible, able to evolve, and scalable.
In order to obtain this result the community of researchers has proposed a set of methodologies which are mostly concerned on the analysis and the design phase of the software development process. Whereas there are interesting commonalities between these methodologies – like the modeling of roles, groups, and organizations – the absence of a standard in this field makes their use and their diffusion harder, and, at present time, their use outside the academic community is scarce. The community of researchers is still actively working on the development and the refinement of agent-oriented methodologies as demonstrated by the various editions of the Workshop on Agent Oriented Software Engineering and by the activity of the AOSE Technical Forum Group which started in 2004. The last trends in this field integrate experiences and results from other fields of Software Engineering such as Model Driven Architecture and Development which are now gaining great interest in the community of SE researchers and practitioners.

Another interesting product of AOSE is the development of agent programming frameworks. In the last two decades a plethora of programming environment and software libraries for the development of multi-agent systems have been released by the community of researchers which often has been supported by the industry. Hence, a lot agent development tools have been designed and applied to specific industrial scenario (telecommunication, manufacturing, and process control). Actually, only a few of them can be considered programming frameworks; most of them provide a partial support, by focusing only on a restricted number of phases of software development product cycle which normally include the implementation phase. Among the few programming frameworks, JADE has become the de-facto standard for the development of multi-agent systems: initially developed by TILab\(^{19}\) along with the University of Parma, it is now known world-wide and the free availability of its code-base allowed other research group integrating new features or developing new solutions by starting from it. At present time JADE has a world-wide community of researchers working around it.

\(^{19}\) TILab stands for Telecom Italia Lab.
Whereas there is a complete lack of standards driving the research efforts in agent oriented methodologies, the practice and the research in multi-agent system development is somehow guided by a set of recommendations which allow the comparative evaluation of the different frameworks. These recommendations are promoted by the Foundation of Intelligent Physical Agents (FIPA) whose purpose is to propose a set of reference models for the development of agent and multi-agent systems and the interaction between different MAS implementations.

In the following chapters I will introduce the AgentService programming framework which constitutes the solution I am proposing for the development of complex and dynamic multi-agent systems. AgentService is the tool that we – me and my research group – have been using in the last three years for delivering agent oriented solutions. It is a product of AOSE and I will describe how the theory discussed in these last two chapters applies to the framework.
This section presents AgentService, the framework developed during my doctoral studies. The framework constitutes the practical contribution of this thesis to the practice of complex system development. After a general overview of the features of the framework in Chapter 4, I will explore in more details some advanced issues that make AgentService useful for complex system development: distributed programming, (Chapter 5) and ontologies (Chapter 6). Conclusions will summarize the results and conclude the dissertation.
Chapter 4 – AgentService Framework

This chapter is the result of a co-joint work with Alberto Grosso who designed and implemented with me the core of the AgentService framework.

1. Introduction

As previously noticed, a framework identifies a collection of tools, libraries, and other components which are organized together and provide an integrated solution for the development of a specific class of software applications:

[...] a defined support structure in which another software project can be organized and developed. [...] a framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project.

AgentService is an agent programming framework. Hence, it provides an integrated solution for developing software applications by using the agent technology and delivers as final product a software system structured in terms of agents and multi-agent systems. AgentService simplifies the development process by providing a layered approach: at the
top level visual tools and powerful abstractions help software engineers in designing their applications while at the bottom level the outcome of this design is translated and implemented upon a solid and robust software platform, which represent the core of the framework. The core of the framework constitutes the real value of AgentService and it will be carefully investigated in this chapter. In the next chapter I will show how such infrastructure allows an easy implementation of distributed mobile agent systems. Chapter 6 will conclude the overview of the framework by describing the support offered by the framework for designing and developing the most important aspect of an agent oriented applications: interactions.

1.1. Rationale and Goals

Before diving into the details it is worth discussing about the motivation which drove our research group to develop a new agent programming framework instead of relying on the pre-existing solutions. This discussion will also clarify the goals behind AgentService.

The main objective of the AgentService framework is to provide a flexible software infrastructure able to be customized to different scenarios. In the case of complex systems this is a fundamental requirement: these systems are normally subject to continuous evolution and modification due to the changing requirements of the users. Hence, it is important to use a system able to cope gracefully with these changes. AgentService is inherently flexible and modular: the entire structure of the framework is centered on the possibility to change all the components of the system, even the fundamental ones. The overall architecture of the framework is extremely modular and this architecture allows the seamless integration of new services. Flexibility has been obtained without renouncing to simplicity: end users are supported with designers and high level abstractions either in the design or in the development phase. These elements can automate many decisions and simplify the task, but it is always possible to switch to a more detailed and powerful interaction. The ability to provide two different options to perform a task is what a makes the framework able to adapt to different software development scenarios.
AgentService is almost unique in this sense. It has been designed with the engineering principles in mind before being conceived as a software environment for the development of agent oriented applications. Some other agent programming frameworks provide a lot of functionalities but do not exhibit the same degree of flexibility characterizing AgentService. Moreover, AgentService relies on a relatively new technology – that is the Common Language Infrastructure (CLI) – and fully exploits the new abstractions introduced by this technology. In particular, AgentService adopts some of the new programming abstractions introduced by the CLI, which allow the elegant implementation of the agent concepts that would have been otherwise artificious. The Common Language Infrastructure describes a language agnostic programming platform portable over different operating systems. The advantage of using the CLI as implementation technology is twofold: the portability over different platform architectures and different operating systems ensures a wide usability of the AgentService framework, while the possibility of developing with different programming languages our agent-oriented applications allows developers being more productive by selecting the programming language they prefer. At present time the Common Language Infrastructure supports more than twenty programming languages with even different programming paradigms (functional, imperative, and object-oriented) and many famous programming languages have been ported on this technology (Pascal, Visual Basic, C++, and Java). Obviously this is a secondary advantage, because the entire AgentService framework is developed by using the C# object model. Despite this, it can be noticed that since the underlying implementation technology allows an easy integration of different programming paradigms, the integration of different programming styles – useful to express in more effective manner – becomes easy and without additional effort.

All these issues make the AgentService framework an extremely open and flexible software environment, which is able to evolve with the changing requirements of the users.
1.2. Overview

AgentService provides software engineers with a library to develop agents, a software environment hosting multi-agent systems, and a set of tools supporting developers at design-time and during the deployment. The core features of the framework are its innovative agent model and the agent platform: the agent model allows a real multi-tasking activity for agents and is flexible enough to implement different agent architectures, while the agent platform provides a flexible and modular software environment able to evolve and adapt to different application contexts. A set of additional components provide a high level interface to these elements and simplify the development process.

The main features of the framework are the following:

- a flexible agent model through which different agent architectures can be implemented;
- a library which defines the core of the system and the basic services of the framework;
- a software environment that hosts multi-agent systems and controls their lifecycle;
- a set of programming language extensions simplifying the implementation of software agents;
- a collection of tools supporting users in designing, implementing multi-agent systems;
- complete support for ontology definition and development;
- automatic code generation for interaction protocols with ontology integration;
- a software infrastructure allowing agents to migrate among different instances of the AgentService platform;
a set of support programs through which users can maintain and monitor multi-agent systems.

Figure 6 gives the overall picture of the entire AgentService framework. We can identify three major components of the framework: the system core, the additional tools, and language support. As showed in the picture the AgentService framework relies on the Common Language Infrastructure which is the implementation technology of the framework. In this chapter I will investigate the system core and the technological solution adopted to implement the system infrastructure, the agent platform, and the agent model. In order to better understand the nature of these choices I will present a brief overview of the features of the Common Language Infrastructure, by focusing the attention to those innovative features that have been adopted in the design and the implementation of AgentService.

Figure 6. Architecture of the AgentService Framework
1.3. The Common Language Infrastructure

Common Language Infrastructure (CLI) is an ECMA\textsuperscript{20} [ECMA02] and ISO-IEC [ISO03] standard that defines a virtual execution environment. CLI is a component oriented programming platform where language-agnostic modules of code are executed in a secure fashion.

The Common Language Infrastructure has been designed to be the target for different programming languages. It offers a full-featured class library and a wide set of runtime services that guarantee proper code execution. Language interoperability is one of the most innovative features of CLI: modules written in different programming languages can be seamlessly and effortlessly integrated without building ad-hoc software connectors. Although language interoperability can also be obtained with other technologies (i.e.: the java virtual machine), such a possibility has not been exploited. This is not case of CLI for which it is possible to write applications in more than 20 programming languages.

At the heart of the CLI lies the Common Type System (CTS) which supports all the types and associated operations expected in modern programming languages. This unique set of type definitions is enforced across all languages targeting the CLI; this uniform view of primitives, enables cross-language interoperability. Source code is compiled into an intermediate format consisting of Common Intermediate Language (CIL) byte-code and metadata. In contrast to the COM and CORBA models where programmers were forced to explicitly generate component interfaces and they had to do that by using complex APIs and Interface Definition Languages, in CLI the metadata is automatically generated and persisted in a language-independent format. The final building block of the CLI is the Virtual Execution System (VES) implementing and enforcing the previously discussed CTS. Metadata provides a standard description of every component available in the run-time,

\textsuperscript{20} ECMA (European Computer Manufacturers Association) it is international industrial association promoting standards for information and communication systems. Further information is available at http://www.ecma-international.org/
and is thus a common interchange mechanism between application programs, system tools, and the run-time itself. This is made possible because every type defined and compiled in the CLI is based on this CTS; the execution engine can verify the type safety of every data element presented for layout in memory and every piece of code to be executed. For compiling and executing tasks, the source code is translated by front-end compilers into a mix of both Common Intermediate Language (CIL) and metadata. The generated intermediate code carries with it all the information necessary for it to be compiled into native code. Run-time byte-code interpretation is not directly supported by the CLI. The CIL is usually verified and compiled just-in-time (a JIT-compiler), method by method, by back-end compilers. Then applications are packaged and deployed in assemblies, which are collection of files containing types as well as resources. Both assemblies and their enclosed types are self-describing, and are deployed simply by placing them in a specified application directory.

The Common Language Infrastructure comes with two innovative concepts: Remoting and Application Domains. Remoting is a software infrastructure providing an explicit communication channel between objects which reside in different Application Domains either they are in the same process, or on different processes on the same machine, or on different process on different machines. Application Domains are a completely new concept for the scheduling of applications and play a key role within AgentService. Hence, they deserve a more detailed explanation.

Application Domains are the unit of execution inside the CLI. They also define the granularity of the security policies applied to code execution. The peculiarity of this new abstraction resides in the fact that it represents unit of isolation inside an operative system process. Isolation guarantees the following:

- The code running within application domain can be independently stopped. In this sense they are similar to threads, even though application domains have a separate address space.
• The code executing within an application domain cannot directly access instances hosted within a different application domain. Static instances are application domain wide, and in order to obtain a link between two instances in different application domains it is necessary to use explicit communication infrastructures such as Remoting.

• A fault within an application cannot affect the code running in others application domain of the same process. Thanks to the isolation provided with the application domain construct unhandled exceptions that originate inside an application domain walk back the call stack until they reach the method which executed the application domain and die without affecting the code running in other application domains.

• The configuration information is scoped by application domain. This means that an application domain controls the location from which code is loaded and the version of the code that is loaded. It also means that it is possible to have multiple versions of the same code running in different application domains.

With respect to a process, an application domain is lighter. Application domains are appropriate for scenarios that require isolation without the heavy cost associated with running an application within a process. A process runs exactly one application. In contrast the CLI allows multiple applications to be run in a single process by loading them into separate application domains. Additionally the CLI verifies that the user code in an application domain is type safe. Whereas a process can contain multiple application domains, the relation between an application domains and threads are more complex: threads are said to transversal to the application domains. This means that the same CLI thread can be used to execute code belonging to different application domains: whereas it is useful to think that an application domain contains its own threads, the actual implementation of the virtual machine does not enforce this constraint.

Another interesting innovation introduced by the CLI are attributes. Attributes are elements of declarative programming that can be used to attach additional information
to methods, classes, properties, fields, and parameters. Attributes are represented and
defined through CLI classes and by using the Reflection API can be inspected at runtime.
They are useful for different purposes and one of the most important is security: by using
attributes it is possible for example to automatically check if a given method has the
required permission in before its execution.

Different implementations of the CLI made this programming and executing
environment available on different operating systems and hardware platforms. A shared
source implementation of CLI is SSCLI (Shared Source CLI) formerly known as Rotor. The
Microsoft .NET framework (hereafter .NET) and Mono are professional implementations of the CLI specification and constitute two of the most important
technologies nowadays available for the development of the enterprise applications. In
particular, the .NET technology is the fundamental implementation technology of
Microsoft, while Mono represents the effort of the open source community to provide an
alternative and valuable solution to .NET framework.

2. Platform Architecture

The agent platform along with the agent model represent the core of the framework:
these two components identify the minimal subset required to implement a multi-agent
system with AgentService. In particular, the platform constitutes the software
environment which hosts the multi-agent system and provides it with all the required
services. In other words, it acts as virtual machine for agents: it controls their
instantiation, schedules their activities, gives them communication and localization
services, and maintains their state. The platform also constitutes the main interface to
the MAS since it gives access to all the services offered by the multi-agent system.

The platform is designed according to the reference implementation proposed by FIPA
(www.fipa.org) and by keeping modularity in mind. The adoption of such reference
model provides a well known and accepted model among the community of researchers;
the definition of a modular architecture allows obtaining a high degree of customization.
of the system. Modularity plays a key role in this case since the agent platform is one of the most important components of the framework and then is the one which mostly concurs in defining a flexible software infrastructure.

Figure 7 describes with more detail the internal organization of the AgentService platform. In this section I will describe the structure of all the three components: the system core, the modules, and the service components. These three software layers cooperate and provide the multi-agent system and end users with all the features expected by a powerful and flexible software environment.

### 2.1. System Core

The system core provides basic services to all the other software layers which compose the agent platform. It is responsible of setting-up a platform instance and controlling its activities. By the term platform instance we mean a single installation of the agent
platform on a machine. The system core coordinates the activity of modules and service components and exposes their services to end users.

The system core is responsible of managing the life-cycle of a platform instance and customizes the setup of the platform by choosing:

- the type of agent-factory used: the agent factory is an implementation of the factory pattern [Gamma94] and is responsible of creating the agent instances or resuming them from a persisted storage;

- the type of agent scheduler used: the agent scheduler is responsible of controlling the life cycle of the agent and the activities it performs. Agent schedulers control the number of threads and systems resources assigned to agents as well as the type of scheduling algorithm used;

- which modules are activated during platform start-up: we will see in the next paragraph that modules are software components through which the platform can be extended and customized;

- the type of the service components that will be activated in the platform: service components are specific software agents suggested by FIPA that should be present on every FIPA compliant implementation. FIPA service components are responsible of agent communication, localization, execution and provide a standard to simplify interoperation among different agent platform.

Moreover, a set of additional properties controlling other aspects of the platform, like remote access, can be added to the configuration settings. The set of properties is open-ended and new properties can be added and used by other software layers in a transparent manner. In example, modules could require additional configuration settings or specific installation preconditions; by using the system properties they can easily detect all what they need to work properly.

Figure 8 describes the life-cycle of every platform instance: the system core executes the state machine represented in the picture. By using a state machine as execution model,
we are able to track more efficiently bugs and detect illegal condition which could lead the platform instance to an inconsistent state. It is worth noticing three main states during the platform life cycle, which mainly concentrate the platform activity: *idle*, *ready*, and *running*. When the platform process starts the system core reads the configuration settings and takes the platform into the idle state. This state identifies a successful setup and the system core has been properly configured to serve all other components. The next step of the process moves the platform into the ready states after which all the pending installs are completed and modules activated. The running state activates all the service components and starts the activity of the multi-agent system hosted by the platform. We can notice that while the multi-agent system is running the control of the system is mainly delegated to modules and service components. In the next two paragraphs we will show how these two software layers exploit the system core to perform their activity.

![Platform Instance Life Cycle](image)

*Figure 8. Platform Instance Life Cycle*
2.2. Modules

The platform accomplishes many of its activities by delegating them to modules. The term module refers to a software component that provides a certain type of service within the context of the AgentService platform. Modules are the common way to add features to the agent platform or to customize it in order to meet the requirements of a specific scenario. Many of the platform tasks are accomplished by modules: the system core basically starts the engine that loads and configures modules; service components are configured to use the installed modules in a transparent manner; specific modules are required by the platform core to bootstrap an instance of the platform. Hence the concept of module is a fundamental part of the platform architecture. The system core provides the modules with a rich set of services: modules can inspect all configuration of the platform, register handlers for events, register new event sources and query for installed components. Such architecture allows modules either to fully exploit the services offered by the system or to extend it in a simple way.

Modules are classified into two different groups: core modules and additional modules. Core modules are required by the platform core in order to create a working agent platform since they perform critical activities. The following sub-systems are implemented through core modules:

- **Storage Management System.** It handles the installation of the assemblies where agent templates, behavior objects, and knowledge objects are defined. Every time a new agent type is deployed on the platform, all the dependent assemblies need to be placed in the storage. When an agent is instantiated all the necessary information for its creation and its execution have to be found in the storage.

- **Persistence System.** It maintains the state of the agents and restores it after a system crash. Agents can be instantiated either as persistent or as not persistent. Persistent agents are registered with the system that saves a copy of the agent state according to the policies defined for the knowledge objects composing the
state of the agent. After a system crash the Agent Management System (AMS) queries the persistence system and recreates the agent community by restoring all the agents whose state was persisted.

- **Messaging System.** It provides the agents with a communication channel based on message exchange. The messaging system is transparent to the agents that use the Message Transport System (MTS) to communicate with peers. The MTS routes all the messages to the messaging system which is in charge of delivering. Messaging modules can provide two different communication models: simple message exchange and conversations that are connected communication channels between two agents.

- **Logging System.** It provides the agents with logging facilities and allows different levels of logging. Even if the logging module is a core module it is not essential for the platform life-cycle and the activity of agents. Hence an instance of the platform can run even without this module.

The framework provides a standard implementation of these components relying only on the services of the Common Language Infrastructure: the storage uses the file system in order to store the assemblies; the persistence system stores the state of the agents by serializing the knowledge objects; the messaging system relies on Remoting. Nonetheless if specific requirements need to be met a custom implementation for these modules can be provided at installation time or afterwards.

Additional features can be provided by installing and running additional modules. By defining and integrating additional modules we can provide more complex services which rely on the platform infrastructure and the core modules. Additional modules are loaded after core modules and can be notified of events that occur during the platform life cycle: by registering to platform events modules can interact with the platform and perform their activity. The mobility infrastructure (see next chapter) has been implemented by using an additional module and the integration with the system has only
required a specific implementation of the agent factory. Such component is dynamically configured and created by the system core during the platform setup.

The plug-in architecture provided with modules is an efficient technique to obtain a flexible hosting environment that can seamlessly evolve according to the changing requirements of the multi-agent system: module installation, configuration, and execution are transparent to the agent activity. Agents or platform administrators can exploit the new features as they are installed into the system. As a proof of concept we can notice that during the development of the framework we used three different kinds of messaging modules: one which used as Microsoft Messaging Queue as storage for messages, the current one which provides a complete CLI compliant implementation, and another which allows inter-platform communication by using web services. The use of different messaging modules has been completely transparent to rest of the system.

### 2.3. Service Components

The platform constitutes the runtime environment for the agents and it has to provide services for make them work properly. FIPA defines a set of basic features the platform must have that are offered through dedicated service agents. AgentService follows the FIPA abstract specifications offering the following services:

- **Agent Management System (AMS).** It is the supervisor and the controller of the MAS. It provides facilities for controlling the life-cycle of the agents and it offers the white pages service to the community.

- **Directory Facilitator (DF).** It offers directory services (i.e. yellow pages service) to the agents of the platform and its external clients. The DF maintains a registry of all services offered by the agents and can be queried to retrieve all the agents exposing a specific capability. DFs deployed on different platforms can join together into a federation.
○ Message Transport System (MTS). It handles the messaging system. All the messages sent in the platform are routed by the MTS that also forwards messages to other instances of the platform. It makes agents interoperable.

In addition to these standard features the platform provides an ontology service in order to ensure that the agents ascribe the same meaning to the symbols used within the messages exchanged. The ontology service, as indicated by FIPA (www.fipa.org), is offered by a dedicated agent who manages all the ontology adopted by the agent community and takes trace of the capability of each single agent.

3. Agent Modeling and Deployment

The models and software artefacts used to define and execute software agents constitute the second fundamental component of the framework. These elements define the agent model adopted within AgentService. In this section I will point out the definition of software agent inspiring the model proposed within the framework, I will present the features of such model, and I will describe how the model is translated into a software artefact, which is managed by the platform. In this section I will also cover in detail the scheduling system used to implement the multi-behavioral activity of software agents. Finally I will discuss the agent deployment process and how the software agents are integrated with the MAS and perform their activities.

3.1. Agent Model

In order to introduce the agent model, we define the abstraction which better represent the concept of software agent within the framework: an agent is an autonomous software entity whose activity is constituted by a set of concurrent tasks and whose state is defined by a set of shared objects. Concurrent tasks are referred as behavior objects while the term knowledge object is used to identify each component of the agent state.
If we examine the previous definition we can notice that agents are represented as autonomous entities; this could suggest that software agents control their life-cycle. In this case this is partially true, since the agent platform which provides basic services to software agents implicitly controls the activity of agents while the AMS, as suggested by FIPA, explicitly manages the agents hosted in the platform. Hence, autonomy refers to the fact that software agents can be conceived, and then designed and implemented, as simple programs having their own execution stack; other software components (except the platform through the AMS) can not control the activity of agents. The autonomy model described here is common to almost all the others frameworks [Bellifemine00, Nwana99, and Poslad00] that want to be compliant with the FIPA reference model (www.fipa.org). Such model does not tell anything about the implementation of agent activities, which are only said to be concurrent, and gives more freedom to framework designers. I decided to keep separate the state of the agent from its activities in order to easily manage, persist, and move the knowledge acquired by it; for these reasons I introduced the concepts of behavior objects and knowledge objects:

- **Behavior objects** contain all the agent computational logic; they are used to model capabilities and services offered by the agents to the community. The overall activity of an agent is defined by the concurrent execution of behavior objects and it is responsibility of the agent runtime to schedule behavior instances in a concurrent fashion. The key concept behind behavior objects is that they have to be considered like little simple programs having their own entry point. Developers implement behavior objects by extending a special class of the framework, by overriding the method that constitutes the entry point, and by adding any other feature they need.

- **Knowledge objects** define the knowledge base of the agent. They are shared between behavior objects which, by modifying their data, can change the state of the agent. Knowledge objects are data structures defined like the Pascal record or the C struct: any CLI serializable type can be a field of a knowledge object. The agent
runtime ensures exclusive access to the knowledge objects and their fields and takes care about all the concurrency issues. The framework offers a reliable service for the agent life-cycle and knowledge objects are handled with different levels of persistence. The agent state can be saved upon each knowledge object change or on a specific user request. Persistence is transparent to programmers that just have to define the fields that compose the knowledge and specify the persistence policy.

The distinction between activities and data allows a clear decomposition of the agent definition providing the developer with a simple customization of the agent definition: new behaviour and knowledge types can be programmed, or they can be chosen from ready-to-use libraries.

The model described above (Figure 9) defines an agent either statically or dynamically, but two different views are provided and the framework makes a clear separation between the agent definition and its instance scheduled at runtime. The first one is
called agent template and it extends the AgentTemplate class, while the other one is the “real agent” and is an instance of the Agent class. The relationship between the two entities is similar to the relationship between the class type and its instances: each agent instance acts according to the relevant agent template. It is worth noting that agent instances do not strictly have the type of their template, but complete different classes are instantiated at runtime. By means of reflection, the agent template is inspected and the agent instance is created according to that model. The running instance of the agent has all the necessary stubs to exploit the platform services and to interoperates with other agents, while the agent template and the behaviour objects have only access to proxies of these services.

The separation between the agent definition and its running instance gives many benefits. Common implementations of the agent model use an abstract agent class specialized by inheritance in order to define new agent types. The types used at runtime are the same defined by the agent designers: such a model leads to high coupling and implicitly binds the definition of software agents to their runtime instances. The model proposed with AgentService completely isolates the definition of agents from the corresponding runtime instances: user implementing software agents are exposed only to APIs relevant to the agent definition while the framework internals are maintained into the Agent class. The new templates are used like prototypes for agent instances and the runtime environment uses them to create agent instances in a transparent manner. Hence, the runtime support can be easily changed without breaking the contract defined by the AgentTemplate class.

### 3.2. Runtime Execution

The agent model represents the essence of a software agent from a design and an implementation point of view. In order to have a complete picture of what software agents really are, it is important to investigate the runtime behavior. I will mostly concentrate the attention on the agent deployment process and the execution of its multi-behavioral activity. These two tasks heavily rely on the concepts defined with the
CLI that are assembly packaging and Application Domains: I will show how these features have been exploited in AgentService in order to deliver a robust and flexible agent runtime environment.

Agent instantiation and execution takes place after the corresponding template is registered with the platform. Template registration is automatically performed by the storage module when a new assembly is uploaded: by means of the Reflection API the assembly is inspected and all the types defining agent templates, behavior and knowledge objects are registered. The platform can be configured in order to accept only signed assemblies univocally identifying the types they contain. After registration templates become available to create new agent instances. The creation of an agent instance involves the retrieval of the corresponding template, the actual creation of the object instance, and its further customization according to the agent template. In particular, the agent instance performs the following operations:

- creates all the required knowledge objects and behavior objects declared in the template;
- connects the behavior instances with the selected knowledge objects;
- binds itself to the runtime that makes available to the behavior objects all the services offered by the platform;
- initializes the agent instance.

At the end of these operations, the agent instance is ready to run and the platform configures the scheduling engine, which starts the agent activity. This process is completely transparent to end users who, while defining new template, just have to compose them with behavior and knowledge instances and provide some initialization code. The overall process is delegated to the runtime environment and to the scheduling engine.
3.3. Agent Scheduling

Agents scheduling is a critical issue for every agent programming framework, since it dramatically influences the overall system performance. Agents scheduling mainly concerns the use of threads to execute the concurrent activities of agents. Threads are limited resources, especially in case of virtual execution environments (like Java and CLI) where often the abstraction of logical threads is used in place of physical threads. A common requirement for software agents is the ability to perform multiple tasks in the same time; hence, agent programming frameworks have necessarily to cope with the problem of defining and implementing a scheduling engine. There are mainly two scheduling models adopted as reference model by the community of agent researchers:

- **Agent-process model**: for each agent a new process is created and the multiple activities characterizing the agent have a proper execution thread; agents use IPC to interact. The creation of one process for each agent ensures isolation: each agent has a separate address space and resource sharing among processes is achieved by explicit cooperation. Processes cannot be easily managed from other processes and only the operative system scheduler has complete control over their execution;

- **Agent-thread model**: for each agent only a single thread, hosted into the process of the platform, is created and it is responsibility of programmers to simulate multi tasking. Interaction is fast and efficient because it does not cross the process boundaries. Threads allow an easy management of agents: by using the common API, agents can be started, stopped, resumed, and terminated. Isolation is not guaranteed since threads within a process share the same address space. Furthermore, threads cannot be given different execution permissions and they normally run with the same privileges of the owning process. Nonetheless, the common adopted solution for agent scheduling is the use of threads. A common extension to this model consists in assigning a variable (or fixed) number of threads to each agent in order to increase the degree of parallelism.
We can notice that the use of a particular scheduling model affects the performance of other features like interaction and basically communication. Communication is fundamental for software agents: the messaging infrastructure dramatically changes if object instances have to cross the boundary of a system process. The same considerations apply to the software infrastructure required to expose platform services. These are some of the reasons which shifted the community of researchers to prefer solutions based on the agent-thread model [Fonseca02]. Solutions based on the agent-process model are more appealing for systems based on single software agents or where the infrastructure defining the MAS is very simple and lightweight. In these scenarios agents can really take benefits from the real autonomy, which is assigned to each OS process.

The scheduling model proposed within AgentService tries to take the advantages of the two models by defining a flexible scheduling infrastructure which can be tuned according to different scenarios and resource requirements. In defining the scheduling model, Application Domains and the Remoting communication infrastructure, play an important role. As previously noticed Application Domains are a sort of light weight processes contained within CLI process and constitute the unit of execution for the CLI virtual machine (hereafter managed runtime). They can run with different security privileges from the owning process and have a separate address space. They also provide a unit of isolation for code execution and the only way to communicate with them is the use of Remoting which is the communication infrastructure provided by the CLI for remote method invocation.

3.3.1. AgentService Scheduling Architecture

The solution implemented can be considered as a variation of the agent-thread model; it gives the possibility to use a wide range of scheduling models that can be dynamically selected at platform start-up. The elements defining the scheduling engine are mainly constituted by the agent factory and the agent scheduler: the former is responsible of creating agent instances and the latter manages the execution of behaviour objects belonging to each agent. Both of the two elements can be configured at platform start-up
and the framework provides some ready-to-use agent factories and agent schedulers. By separating these functionalities from the system core the scheduling engine can implement both of the solutions previously presented. In particular, we can classify the different available solutions by examining the number of Application Domains and the number of thread used; it is interesting to analyze the effects of combining the different factories and schedulers provided by the framework. The system is open ended since third parties can implement their own schedulers and factories if the options provided by the framework do not meet the given requirements.

### 3.3.2. Agent Factories

The framework provides two different agent factories: `LightAgentFactory` and `AppDomainAgentFactory`. The former emulates the behaviour given by the agent-thread solution: in this case all the agent instances are created within the same Application Domain. The latter gives a smarter control over the mapping between the Application Domains used and the agents created; such mapping is controlled by a policy which selects, and eventually creates, the Application Domain to be used for the next agent instance. Software developers can define their own custom policies and instruct the platform to use them, but, in the end, the decisions taken with the policies can always be overridden by platform administrators who can define restrictions on the usage of system resources. By using the `AppDomainAgentFactory` and defining custom policies we can:

- emulate the behaviour of the `LightAgentFactory`: we just have to define a policy which use only one Application Domain;

- emulate the behaviour of the agent-process model with better performances: we simply create a new Application Domain for each agent. The communication between Application Domains within the same process is relatively fast if compared to inter-process communication;
• group a set of closely related agents within the same Application Domain in order to control them as a single unit: Application Domains can be unloaded and all the dependent resources released while the hosting process is running.

These three classes can meet the requirement of many different scenarios. We can observe that, even if Application Domains are extremely powerful and can be considered as lightweight processes, they have a cost for the managed runtime since they require additional data structures inside the virtual machine and need additional OS resources.

3.3.3. Behavior Schedulers

Whereas agent factories are responsible of setting up agent instances, agent schedulers control the execution of behavior objects within a single agent. In order to understand the features of the different options provided by the framework we have to spend some words on the different types of behavior objects which can be defined. Behavior objects can expose the thread-less property which means that they do not require a separate thread to be executed. Thread-less behavior objects are executed sequentially by using a round-robin strategy and can be processed with a single thread, while each non thread-less behavior object require a thread.

In order to make AgentService suitable for different application scenarios, the framework provides platform users with three different agent schedulers: DefaultBehaviourScheduler, ThreadPooledBehaviourScheduler, and CompositeBehaviourScheduler. The first one must be used for agents which have non thread-less behavior objects, this is the most expensive scheduler since requires the highest number of threads. The ThreadPooledBehaviourScheduler is a dedicated scheduler for agents for thread-less behaviour objects; it implements a custom thread pool which allows configuring the number of threads to use. The CompositeBehaviourScheduler is a composition of the previous schedulers and should be used when software agents have either thread-less or non thread-less behavior objects; it simply dispatches the management of behavior objects to the scheduler required by their type. We can observe that non-thread-less behavior simplifies the development but
requires more resources at runtime while thread-less behavior requires more attention in the development phase but can be managed more efficiently at runtime.

The combination of the agent factory and the agent scheduler results an extremely flexible scheduling engine which can be extended with more complex and powerful implementations of these components. Figure 10 shows the advantages and the

---

**Figure 10: Scheduling Engine**

The combination of the agent factory and the agent scheduler results an extremely flexible scheduling engine which can be extended with more complex and powerful implementations of these components. Figure 10 shows the advantages and the
drawbacks of the possible combinations that can be selected for a given multi-agent
system.

3.4. Discussion

It is now possible to present some considerations about the proposed agent model. In the
following I will briefly discuss the advantages deriving from the implementation choices
adopted and the underlying design. The advantages are mainly identified by the three
following features: flexibility, real multi-behavioral activity, and real autonomy.

3.4.1. Flexibility

Flexibility is, perhaps, the most evident property of the proposed agent model. In
defining the structure of the agent model, I have decided to maintain a general structure
rather than supporting specific agent architectures (BDI or Layered). For this reason, I
have devised a model general enough to express different agent architectures. At the
same time, the proposed model – based on the concept of concurrent tasks and shared
data – is sufficiently simple to allow a quick implementation of software agents.

Some programming frameworks, explicitly bind their model to a specific agent
architecture that normally is the Belief Desire Intention [Rao95] or a variation of it; I
think that this solution is restrictive since it forces the developer to use only a single
implementation strategy. Other programming frameworks – namely Agent Factory
[Collier03] and JACK [Coburn00] – are more flexible and provide support for targeting
the BDI architecture but do not force developers in implementing BDI agents. At present
time AgentService does not provide facilities for targeting the BDI architecture, even
though the implementation of this architecture on the proposed agent model is not a
hard task.

The simple structure of the agent model relies on the concepts of concurrent
programming based on threads with some additional features introduced to ensure a safe
access to the shared data. It is then possible to apply all the techniques of multi-threaded
programming by mapping behavior objects to threads, and knowledge objects to shared
data. If behavior objects do not have interact, they can be considered as standalone programs and the agent scheduler will run them concurrently in a complete transparent manner. Moreover, the AgentService API provide additional features for the management of behavior objects that allow the creation of complex concurrent applications: it provides advanced synchronization constructs and the possibility of create new behavior objects from within other behavior objects. Hence, even though the proposed structure is conceptually simple, it allows a variety of different strategies for implementing the agent internal behavior.

The possibility of customizing the agent scheduler introduces another degree of flexibility. In multi-agent systems with thousands of agents running on the same platform instance it is not feasible to use one thread for each agent behavior object because the CLI virtual execution environment does not allow it, and starts simulating logical threads thus giving the illusion of concurrency. This condition leads to a huge decrease in performance. By using custom schedulers, it is possible to introduce thread pooling and obtaining a more performing system. But it is possible to go beyond, and use thread-less behaviors when explicit concurrency is not required. Thread-less behavior objects are just a specialization of the common behavior object that do not require a separated thread to be executed.

In conclusion, the proposed agent model is characterized by an extremely flexible architecture: it is general enough to allow the implementation of different agent architectures and along with the features of the scheduler provides an incredibly scheduling engine under performance and resource usage constraints. Finally, from a design point of view, behavior objects and knowledge object constitute reusable components ready to be used to compose other agents.

3.4.2. Real Multibehavioral Activity

The AgentService scheduling engine allows a real multi-behavioral activity: while other agent programming frameworks impose some constraints on the design of the concurrent task, AgentService gives a complete freedom to the developer. The definition of
non thread-less behavior objects is as simple as defining a main program and the agent scheduler will take care of executing the different behavior objects in a concurrent fashion. Other agent programming framework – for example JADE – do not allow using one thread for each concurrent task and suggest that behavior objects should be designed in order to be executed quickly. Actually, this suggestion becomes a directive if the user wants to have the illusion of concurrency. Hence, in the case of complex and long algorithms they have to be implemented with more iterations of the same task or divided in subsequent tasks. Such modification is necessary for interleaving the execution of the long virtual task with the execution of the other tasks. This is a wrong and unnatural use of the concept of task which creates a gap between the design of the agent and its actual implementation.

AgentService does not require separating into smaller steps long tasks, but they are seamlessly scheduled along with the other tasks without the programmer intervention. However, the excessive use of non thread-less behavior will lead to the performance problems discussed in the previous point, and, for this reason, agents that have non thread-less behavior objects have to be carefully designed. The advantage of using AgentService, in this case, resides in the fact that lets you choose what to do.

### 3.4.3. Enforcing Autonomy

More than before, the design and the development of a multi-agent system is a Software Engineering task. Hence, it should be supported with effective tools which provide the right abstractions and, above all, guarantee that these abstractions are properly translated into code preserving their distinctive features. One of the fundamental features of the agent abstraction is autonomy: from a programming point of view autonomy does not only means having a proper thread of control but also implies that there is no code automatically executed as a consequence of an external method call. Put simply, this means that, while programming a multi-agent system, agent instances must not communicate through direct method calls, especially when it is given to user the possibility of defining any method he wants. Whereas this kind of interaction is
definitely efficient if compared to other techniques – like, for example, message exchange and conversations – it is potentially dangerous and implies the following risks:

- **Security and Loss of Autonomy.** As pointed out by Wooldridge and Ciancarini [Wooldridge01] by using method calls the control on method execution is maintained by the caller and not by the callee, which could even not know when the method is called. By allowing this, even in a controlled manner, we’re not guaranteeing the autonomy property of software agents, which could perform actions outside their thread of control, thus corrupting their internal state. It can be observed that this is minor implementation detail and that it only leads to corruption if there is a bad design of the internal behavior of the single software agent. My opinion on this issue is that it definitely opens a security hole and it is a potential weakness. The rationale behind this is that, since multi-agent systems are dynamic and open systems by definition, multiple and heterogeneous types of agents can concurrently run in the same process, even with different timings. Then, it is impossible for the software engineer when designing a specific type of agent to know all the types – and then the intentions – of the agents that will interact with the instances of the designed type. Even it could be possible from the administrative point of view to prevent the execution of malicious agents once recognized, a design which do not use direct method invocation is safe a priori.

- **Broken Design.** Agents are conceived, and should be designed, to interact at knowledge level and not through method calls. A design which allows using direct method invocation implies the risk of preferring to code an interaction through method calls just for the sake of efficiency. This design contributes to the definition of rigid and static interactions which undermine the supposed flexibility that should characterize intelligent software agent. If we remove the opportunity of using direct method calls we force designers to model interactions at a higher level, that is, the knowledge level.
For these two reasons, the agent model proposed with AgentService, avoids direct method calls and provides a structure in which the interactions among agents occur only by message exchange. The system then guarantees that all the messages are processed within the agent thread of control. By using this model is possible ensure the autonomy of agents that are controlled only by the platform infrastructure, which is in charge of managing their life cycle. It can be observed that the platform infrastructure actually has a complete control over software agents. This is not an issue anyways, because the tasks performed by the platform infrastructure are mainly concerned with maintaining alive the multi-agent systems or just execute commands issued by the users which can freely control their agents.

Moreover, by using one application domain for each agent we can obtain that kind of isolation which is normally given to the traditional OS processes: in this case software agents are real stand-alone autonomous applications that are only controlled by the platform infrastructure. The use of application domains separates the address space of software agents which cannot share any object unless they explicitly do it through the Remoting API. In this case, there must be an explicit will at design level allowing a more explicit communication: actually, the use of Remoting API requires the collaboration of two entities which are engaged in the communication. Hence, it is not possible to maliciously exploit it unless you open the door to it.

4. Language Support

Language support is not a fundamental feature of the AgentService framework and it is still under development. However it represents an example of the wide range of possibilities offered by the Common Language Infrastructure. Language support mainly concerns with the Agent Programming eXtensions (APX). These are a set of extensions to the C# language aimed to simplify the development of software agents for AgentService and to automatically enforce the application of the required programming patterns. In this section I will give a
brief overview of this feature by focusing on the integration of the language with the framework and its relation with the agent model.

The development of agents with the AgentService framework requires programmers to follow some programming patterns ensuring the production of safe code that takes advantage of the features of the adopted agent model. These programming patterns are mostly concerned with the definition of agents, knowledge and behaviour objects, and knowledge objects access. The wrong use of such patterns does not cause the platform instance to crash but only terminates the execution of the “ill” code. Writing the code required the framework can be sometime annoying and exposes to developers some uninteresting framework internals. This task can be partially automated: the information needed to implement the software pattern required by AgentService can be inferred from the context and developers just have to specify some additional properties.

In order to increase and simplify the implementation task we designed a set of extensions to the C# language which represent the key concepts of agent model adopted by AgentService. In particular new language constructs have been defined to represent agent templates, knowledge and behaviour types: these concepts have been modelled by adding the keywords agent, knowledge, and behaviour. Such extensions are translated by modified C# compiler into the programming patterns required by the framework. The compiler also performs additional checks in order to produce safe code: during compilation it verifies the proper use of the extensions and performs an additional semantic analysis in order to detect concurrency flaws when accessing knowledge instances. In particular the use keyword is introduced in order to control the scope of knowledge objects making them visible only within thread-safe code block.

AgentService proposes a development model where agent, behaviour, and knowledge instance are completely managed by the runtime environment. The extensions introduce into the language the concept of template that is a type that can be defined but cannot be directly managed (i.e. instantiated). Templates are used to define agent, knowledge, and behaviour types and expose just the properties required for their definition. The compiler
translates them into classes by adding all the boiler-plate code required by the framework and verifying the right implementation of patterns, thus reducing the number of possible bugs.

APX do not define a new language but just some extensions that integrate into the C# language: developers can take advantage of these features while they define agents, knowledge, and behaviour objects, without renouncing to the strength of C#. APX introduce some syntactic sugar and additional type checks that speed up and simplify the development of code targeting the AgentService framework; this is a technique already adopted by other compilers like the C# compiler itself for some high-level constructs like properties and the lock statement. The work performed by the APX compiler frees users of taking care about the details and the patterns imposed by the framework, thus simplifying the coding activity.

In order to completely support the development process, the framework provides a software package integrating the Agent Programming eXtensions into the Visual Studio .Net development environment. The package allows users to write code either with APX or with the C# language and exploits all the services offered to other languages like C#, VB.NET, and C++:

- project management;
- wizards and templates;
- source code control, syntax highlighting, and on-line parsing;
- package deployment.

These features allow programmers to organize all the elements of a software project, to detect many bugs during the implementation phase, and to give a complete support to code management and production. Thanks to the package, the APX compiler is integrated into the environment and is used side by side with the compilers available for the other installed languages.
5. Observations

In this chapter, I have presented the fundamental features of the AgentService programming framework. In particular I have described the internal structure of the platform and the agent model.

The agent platform represents the run-time environment of software agents and actually acts like a virtual machine for them by providing the basic services to agents and by taking care of their life cycle. The platform is one most important components of the AgentService framework since it is the software infrastructure that actually allows building multi-agent systems. Being a crucial component of a complex software system it has been designed with an extremely modular architecture that makes it a very flexible infrastructure. This is a fundamental requirement for today complex software systems which evolve quickly and then require evolvable technology. In the previous chapters I motivated the used of Agent Oriented Software Engineering for the development of complex software systems and I have showed how the inherent properties of software agents and multi-agent systems characterize a software system expressed in terms of agents and community of agents as an optimal solution for these scenarios. However, the power of such abstractions is not enough to lead to a successful implementation of the agent oriented design, but requires a good support from the implementation technology. Hence, the role of agent programming framework is crucial, and AgentService provide a good support to implement agent-oriented software systems: its modularity and inherent customizability allow adapting the framework to different scenarios, and changing even its core components if needed. Then, it delivers a software infrastructure as flexible as the agent oriented approach is.

The agent model represents the second pillar of the AgentService framework. It actually defines how agents have to be implemented and which properties they can exhibit by design. AgentService provides users with a rather general agent model based on the concept of concurrent tasks – called behavior objects – and shared data – namely knowledge objects. This model is simple and flexible: it allows a quick definition of
software agents as well as the implementation of a specific agent architecture. Moreover, AgentService provides two different views of a software agent: the design view and the run-time view. The design view is used by developers to define new types of agents which are called templates, while the run-time view is used by the platform and is mostly concerned with the active instances of agents. Developers define templates which are inspected at run-time and used to customize the agent instances. The strong separation of concerns obtained with AgentTemplate-Agent class pair, hides the internal details of the framework to the developer that just have to define the properties and the behavior of the agent. Finally, the proposed agent model allows a real autonomy for software agent, which is one of the most important properties of agency.

This chapter has also presented a brief overview of the Agent Programming eXtensions (APX) which constitute an additional tool aimed to simplify the development for AgentService. APX are a set of extensions to the C# programming language which introduce the key elements of the AgentService agent model as primary abstractions into the language. Such abstractions model the concept of agent, behavior object, and knowledge object. The extensions have been integrated into a professional development environment that is Visual Studio .Net, so that the software developer can use the extensions side by side with the other programming languages and increase its productivity.

In the next two chapters I will discuss about the advanced services which rely on this infrastructure and I will show how the implementation choices made for the core of the framework have simplified the introduction of additional and more complex features.
Chapter 5 – Distributed Programming with AgentService

This chapter is the result of a co-joint work with Simone Bellotti who designed and implemented with me the support for distributed computing in AgentService.

1. Introduction

Agent technology exhibits its true power when software agents interact and cooperate to pursue multiple goals whose some are common and others not. Hence, the notion of multi-agent system, better than the one of agent, is better suited to model complex and dynamic systems. Moreover, if we look at today’s software systems, it is rather common to deal with systems of systems. Thanks to the advances in networking and the advent of the Internet many preexisting applications have been connected each other and integrated in larger software systems. The resulting systems are naturally distributed and distributed programming [Tanenbaum06, Coulouris06] has become a necessary practice to support the development of contemporary applications. Distributed programming characterizes software development at any scale: Grid Computing [Foster01] is perhaps one
of the biggest examples, but even a simple software application on a desktop computer which connects to a web service, delegates to it a job, and uses the results of its computation adopts distributed programming techniques. Support for distributed system development is then a fundamental requirement for any programming framework and this is even more important in the case of agent programming frameworks. These are the natural means to implement multi-agent systems: by definition, multi-agent systems are *distributed computing systems* characterized by distributed control and asynchronous computation (see Chapter 2, Section 4). Moreover, evolvable software systems, which are the target of multi-agent systems development, are inherently distributed systems: they change their structure by aggregating new components and subsystems, and the interaction among their components change according to the new topology. Hence, support for distributed systems is a fundamental requirement for agent programming frameworks.

In this chapter I will describe the facilities introduced in the *AgentService* framework to support distributed programming. Such features are basically concerned providing a simple way to interact for agents belonging to different *AgentService*-based multi-agent systems and supporting the mobility of agents. Thanks to the modular architecture of *AgentService* these features have been implemented by developing two different modules that are the *Messaging Module* and the *Mobility Module* which seamlessly integrate into the framework. In this chapter I will describe the general architecture which allows distributed programming and explain in detail how distributed interaction and code mobility is performed by the two modules. Messaging and code mobility are two basic services that can be used to compose more complex services: in the last part of the chapter I will describe the load balancing infrastructure which uses code mobility to distribute the computational load of software agents among different federated *AgentService* platform. Before diving into the details of *AgentService* I will make some general considerations about applying distributed programming to software agents.
2. Agent-Based Distributed Computing

Software agents base their activity on interaction with peers. The standard interaction pattern – that is also the one proposed by FIPA – is message exchange: agents interact with each other by exchanging messages within a conversation. Moreover, multi-agent systems provide software agents with all the services they require to perform their activity. Among these services there is, according to the FIPA guidelines, the messaging service that provides message delivery for agents. By using the messaging service software agents send and receive messages just providing their credentials and the address of peers involved in the interaction. The system infrastructure is in charge of locating the software agents which take part of the interaction and providing access to their mailboxes. Since message delivery and agent localization are common services of FIPA compliant multi-agent systems, software engineers do not have to deal with these issues while designing multi-agent applications but they just have to query the Directory Facilitator to locate the agents and use the messaging system to make agents interact. The pattern is the same for either local or remote communication; hence, from a programming point of view distributed agent-based applications do not imply additional structural complexity if compared to local agent-based applications. This is an important advantage of using agent technology: other approaches require learning new techniques and using additional infrastructures in order to accomplish distributed computation. For example, with traditional object-oriented development the management of remote references requires the setup of additional infrastructures providing these services: in many cases is the application that is responsible of this setup.

Java technology relies on the RMI (Remote Method Invocation) infrastructure to build object oriented distributed systems: application developers have to setup RMI registries which are used to locate remote objects. RMI registries are stand-alone components which are accessed within the software application through the Java standard APIs. Once we have a reference to the registry we can require an interface to a remote object and interacting as if it were a local instance. Whereas the interaction with the object is easy and natural
the setup requires additional effort and is something that has to be done outside the application: we have to start a RMI registry and register the interfaces we need to use.

In the case of the .NET framework Remoting is the technology provided by the framework in order to simplify the development of distributed object oriented systems. As in the case of RMI, Remoting provides the infrastructure required for remote method invocation. In order to obtain a proxy to a remote object it is necessary to complete a two-step process: firstly instances that want to be accessible have to setup a remoting server in their process and register themselves with it; instances from other processes can connect to the remoting server and obtain a reference to a remote instance. After this initial setup the remote reference is managed like any other local instance. The .NET framework does not require setting up a registry outside the processes involved in the interaction, but – as happens for RMI – the use of Remoting requires a basic knowledge about what lies behind the proxy and how parameters are passed when a method is called. Even though these solutions simplify the development of distributed object oriented systems if compared with other technologies like CORBA [Henning99] and COM+ [Löwy01], they require additional knowledge and effort to be successfully applied.

Web Services are a cross language technology whose aim is to greatly simplify the development of distributed systems. Web services provide a standard communication infrastructure based on XML over HTTP and allow an easy integration of remote objects into applications. There are a lot of frameworks which allow a quick development of Web Services and the corresponding clients: within an application a reference to a web service is a like a reference to a remote object upon which it is possible to invoke methods. Again, whereas integrating a web service into an application is really simple, delivering a simple object as a web service requires more burden than using RMI or Remoting: web services are generally hosted into web servers and have to be deployed on them. Hence, they require an additional component which normally is not a part of the distributed applications.
The case of software agents is slightly different: multi-agent systems development is based on the use of agent platforms which constitute the runtime environment for software agents. In this scenario, software agents always require this runtime environment in order to interact with each other. Agent platforms implementing the FIPA guidelines provide agents with messaging and localization services as part of the runtime environment. These services are standard and basic features of multi-agent system development either they are distributed or not. Implementing a distributed multi-agent system is just a matter of instrumenting the infrastructure and not the application. Normally, this is a task performed by platform architects and not software engineers using the agent programming framework. The ability to deliver a message to software agents residing on a different platform is a feature available with many agent programming frameworks. Hence, it is possible to assume that the infrastructure is always enabled for agent-based distributed computing.

As previously said, the agent-paradigm implies no additional burden for distributed computing since it allows to apply the same programming model used for non distributed agent-based applications. It is the system infrastructure which actually makes possible distributed computing. In addition, agent technology suggests standards for interoperability among different agent platforms by means of a standard Agent Communication Language (ACL) \(^{21}\). It is then possible to aggregate preexisting multi-agent systems developed with different technologies by introducing interface agents: by communicating through the ACL these agents can establish a link between different implementations. Actually this opportunity is not fully exploited and the use of ACL to develop complex agent based applications is still a hard task. Nonetheless, the systems infrastructure, which has been designed to dynamically introduce new agents into the system in a seamless fashion, provides a solid base upon which implement the previous scenario.

\(^{21}\) More details about the Agent Communication Language can be found in the next chapter.
3. **AgentService Distributed Computing Support**

### 3.1. Overview

The support provided by AgentService for distributed computing mainly concerns creating a software infrastructure that allows software agents to easily interact regardless their location and to move among different sites. In order to supply these services AgentService exploits its modular architecture which allows an easy integration of additional features. Support for remote interaction has been implemented in the *Messaging Module* while support for agent mobility is a service provided by the *Mobility Module*. The two services are strictly connected: in order to orchestrate the mobility process the system requires the ability to exchange messages among AgentService platform instances. For this reason, I will first describe the features of the *Messaging Module* and then I will discuss the characteristics of the *Mobility Module*.

### 3.2. Messaging Service

The messaging service provides the basic features of message delivery for software agents. According to the FIPA guidelines, this task is accomplished by the MTS (*Message Transport System*) that should be implemented as a software agent: software agents delegate to the MTS the tasks of message delivery. Within the AgentService architecture message dispatching is demanded by the *Messaging Module* which also manages the mailboxes of agents and provides message persistence; the MTS just forwards the requests to the module and synchronizes the access to the module. By using the MTS as a general interface for software agents and demanding to the module the task of message delivery it is possible obtain a more flexible structure. It has been possible to change and extend the features of the messaging systems without modifying the infrastructure of the agent platform.
The services delivered by the each messaging module are the following:

- general agents mailbox management;
- message dispatching;
- connected message exchange among two agents (conversations);
- mailbox persistence management.

Different levels of quality of service have been offered by varying the implementation of the Messaging Module. The first implementation of the module provided just the basic service of mailboxes management without persistence. The module has been then changed and integrated with Microsoft Message Queuing by adding support for transactions and integrity. The current implementation of the service uses in-process mailbox storage and relies on web services to send and receive messages. Thanks to the use of Web Services the multi-agent platform can be easily integrated with any kind of application and not only with the AgentService platforms.

In this chapter I will only describe the implementation of the latest module, which is an extension of the original implementation, but before explaining the architecture of the system I will briefly describe the general infrastructure of the messaging system.

### 3.2.1. Messaging System Infrastructure

The components which make up the infrastructure of the messaging system are three:

- the messaging client;
- the Message Transport Systems (MTS);
- the Messaging Module.

These three elements are involved in every communication among agents. The messaging client is the interface that the runtime supplies to send, receive messages and handling conversations. This component uses the services of the MTS to access the Messaging Module that manages mailboxes. This process is the same either for local message dispatching or remote message delivery. In the case of remote message delivery the Messaging Module will
take care of sending the message to the remote platform by contacting the remote messaging module. This process is completely transparent to the software agents, which just discriminate between local and remote software agents with the AID containing the name of the hosting platform.

Figure 11 describes the process of setting up the messaging infrastructure: when the platform starts the module is loaded and attached to the platform runtime. The communication infrastructure is then attached to the agents when they are created or restored from the persistence storage. The platform core queries the Messaging Module and asks for a specific client if the module does not require a specific client the DefaultMessagingClient is supplied. Actually all the messaging modules provide the basic service of message exchange while the conversation abstraction is supplied through the DefaultMessagingClient. By delivering the conversation management to the default
messaging client it is possible to always have available, despite the kind of module installed, this service. Through the messaging client the agents can:

- send and receive messages;
- send messages to a group of agents;
- peek the messages in the queue and remove them;
- filter messages received;
- open conversations.

By using the messaging client all agents can interact with peers and perform their activity. What lies behind the client is completely transparent to software agents and basically refers to the module implementation.

### 3.2.2. Distributed Messaging Module

The implementation of the Messaging Module with support for remote message delivery relies on web services technology in order to dispatch messages on remote AgentService platforms. The architecture of the distributed messaging system is depicted in Figure 12.

![Figure 12. Messaging Infrastructure](image)

The connection among the different AgentService platforms is managed by the messaging module installed on each platform that maintain a mapping between the platform instances and the corresponding web service that has to be contacted. Whenever a new message has to be sent outside the platform the module discovers the appropriate web service and sends the message. The Messaging Module is connected through the Remoting
service with the local web service and is notified every time a new message is sent to the local web service.

The use of web services allows an easy integration of software agents with standard and legacy software applications. As previously discussed web services technology provide a very simple way to apply distributed computing and integrate preexisting application. In the case of the AgentService this technology contributes to create an open system as multi-agent systems should be.

3.2.3. Message Serialization and Delivery

The decision of sending messages through web services introduced the problem of expressing AgentMessage objects into a portable format and of providing a technique for transforming them into an XML document. Whereas the XML format is a plain text format the AgentMessage instances have been designed to be containers of any kind of instance. Hence, there has been the problem of providing the same flexibility of remote message delivering and a completely transparent service. According to the FIPA guidelines the AgentMessage class is composed by an Envelope and a Body: whereas the content of the envelope can be easily translated into an XML fragment the hard task is represented by converting the body that contains any kind of object. In order to provide a completely transparent service it has been necessary to devise a technique allowing the XML serialization of any agent message despite its content. This feature is what makes distributed programming with AgentService incredibly simple and without additional costs. Hence, it is worth noting how this has been addressed.

The conversion technique uses a mixture of XML serialization and binary serialization. Binary serialization is a technique that converts an object instance into a stream of bits. The resulting string of bits can be saved in a persistent storage or moved into another site. Unless programmers explicitly prevent it while defining a type, serialization is technique that can be applied to any object. Hence, being serializable is a sufficiently general property to be required for any object that needs to be added as content for a
message. This is a property required for any message content either it is dispatched locally or remotely. In order to allow remote delivery through web services we introduced a new kind of message body (namely `TransportBodyMessage`) which can be easily serialized into an XML format by converting into a binary string each item added as content and by keeping track of its type and all the information required for its restoration. An example of message serialized into XML is given in Figure 13.

```xml
<AgentMessage>
  <Envelope>...</Envelope>
  <Body fullTypeName="MyMessageBody"
    assemblyName="library.dll"
    pkToken="23.4f,24,5e,A2,8&07,C1">
    <Items>
      <Item fullTypeName="MyItemType3"
        assemblyName="library3.dll"
        pkToken="34,4f,34,5e,A2,8&07,C1"
        instance="[Serialized Data]/>
      <Item fullTypeName="MyItemType2"
        assemblyName="library2.dll"
        pkToken="55,49,5f,52,A0,8&07,C0"
        instance="[Serialized Data]"
      ...
      <Item fullTypeName="MyItemType1"
        assemblyName="library1.dll"
        pkToken="23,4f,24,5e,A2,8&07,C1"
        instance="[Serialized Data]"
      </Items>
  </Body>
</AgentMessage>
```

Figure 13. XML Format of the `AgentMessage` class.

This solution allows the maximum flexibility and relies on a well established technology, which is a good premise for openness and interoperability with other systems. Moreover, the use of binary serialization allows a simple verification of the message content: in order to be restored, each item is described with the additional information – namely the name of the type of the instance and public key token of the assembly containing the type definition – that guarantees its right reconstruction. This technique is also useful to track one of the common problems in distributed computing, which is the presence of different versions (and probably definitions) of the same element. Buy using the public key token we are guaranteed that elements are defined in the same way or we can easily identify the problem.
3.2.4. Observations

In this section I have presented one of the pillars of distributed programming with AgentService, which is the possibility of allowing agent interaction despite their position. As suggested by FIPA, AgentService uses a registry maintained by the Directory Facilitator which maps services to agents providing them. Agents are identified by a unique identifier containing information about their physical position. By querying the Directory Facilitator agents can easily discover other agents and interact with them and the distinction between local and remote interaction is just constituted by the Agent Identifier (AID). This architecture allows a simple development of distributed applications with AgentService and does not require additional effort respect to developing an entirely local multi-agent system. The technical aspects of distributed programming have been maintained inside the Messaging Module, which, despite its implementation, provides to developers always the same service: message delivery. Developers do not have to care about these aspects unless they need to design agent-based application interoperating with other systems at code level. For this particular scenario, AgentService allows an easy integration with other systems by relying on web services, which are a well established technology for distributed computing. By using web services the integration with a multi-agent system developed with AgentService is easy as calling a method on a local object.

Whereas the support for distributed programming does not require any additional effort, from the architectural point of view the systems requires that all the distributed platforms that want to communicate adopt the same Messaging Module. Even if this decision could be interpreted as a limitation, it basically denotes an agreement on a communication standard, which is a requirement for any interaction. Moreover, the modular architecture of the platform allows an easy integration of additional functionalities and it is possible to instrument an AgentService platform with the right module without recompilation or deployment. This flexibility is one of the features that characterize AgentService as a framework supporting evolvable systems.
3.3. Mobility in AgentService

Support for mobility of software agents is the second pillar of distributed programming with AgentService. I will briefly introduce the concepts of mobile code and mobile agent and I will describe in detail the architecture and the implementation issues of the mobility service in AgentService.

3.3.1. Code Mobility

Code mobility [Fuggetta98] is the ability of moving code through the node of a network. Whereas mobility seems to be just a new property added to the plain and old concept of executable code, a further investigation of what is considered code in this context is useful to understand the evolution and the history in this field. When we speak about code we normally refer to executable instructions that are executed by a processor and that perform some operation. We normally distinguish the concept of code from the concept of data that normally refer to structured information and are manipulated by code. Actually, even code is organized as structured information and the real difference resides in the processor which attributes to such data the meaning of instructions and executes them by mapping their content to a predefined set of operations. According to this view, the division line between the code and data is blurring and consequently when we speak about mobile code we can find a wide spectrum of techniques that differ in what is considered subject to migration. Mobile code techniques range from RPC [Birrel84], which can be considered the weakest notion of mobile code, to stronger ones. Actually, RPC does not involve mobile code, but relies on statically installed code on the RPC server side to be invoked by client applications. The later extensions of RPC towards object oriented systems, most notably CORBA [OMG01], do not have changed anything of this. These techniques are obviously closer to the data end of the mobile code spectrum and are very limiting: if the installed set of procedures on the server does not satisfy the client requirements additional effort and multiple RPC calls have to be done to simulate the desired behavior. In order to solve this problem a number of systems soon
appeared which were based on the concept of sending simple “programs” to the server which combine remote procedures as primitives [Falcone87, Stamos90, Stoyenko94]. These programs were evaluated by the server, invoking the indicated primitive procedures in that process. They constituted a form of mobile code, albeit in a rather simple language. A very simple example of this technique can be represented by SQL: database access libraries provide interfaces through which it is possible to issue a sequence of SQL statements that are executed on the database server. This technique is known as function shipping [Wang91] and it is a very well known practice Database Management Systems. A more conservative position has been indeed taken by the first traditional process migration systems where only process contexts are moved between nodes with all process code preinstalled at every node (like Locus [Thiel91], Mosix [Barak93], or Panda [Assenmacher95]). In these systems a process context is interpreted roughly like a complex “jump pointer” into the preinstalled code that is then executed. For this reason, they can be considered closer to the data end as the RPC systems. Solutions which are definitely closer to the code end of the mobile code spectrum are the following:

- The Abode Postscript Language [Adobe85]. The Postscript language is used to write programs describing the graphical appearance of a printed page. Such program is sent to a printer and executed there, printing the page as side effect. While Postscript programs were moved only once, prior to their execution (namely, from the computer to the printer), the Postscript language has, somewhat curiously, actually been used as a basis for fully mobile programs in the Transportable Postscript (TPS) system [Heimbigner95]. TPS programs can move during their execution, fully preserving their internal state. Presumably, the TPS design was initiated by the implementation of the Postscript interpreter as a stack machine, which considerably simplifies extracting and restoring its state during execution.

- The M0 system [DiMarzo95] is based on exchanging small snippets of code in a primitive language (actually similar to the Postscript language from above) called “messengers” between nodes. Each such messenger is interpreted as a separate
thread of control by a run-time environment which also provides inter-messenger communication and synchronization. Messengers can autonomously migrate to other nodes. Messengers are a very basic mechanism from which higher level distributed systems can be built, in particular a distributed operating system.

- **Intelligent routers** [Wolfson89, Voorhees91] were programs written in a rather limited procedural language containing a migration statement. Executing routers were shipped as text files containing code and state between interpreters on various machines, apparently allowing fully state-preserving and autonomous migration. The intended application domain was manufacturing automation. The primitive language and runtime system were to be replaced by more ambitious concepts, but further history is unclear.

- **Management by delegation** [Goldszmidt95] employs processes which can be extended at runtime by code received from remote nodes. Such code is compiled to native code and dynamically linked into the address space of the running process. This scheme allows maximally efficient mobile code execution, appropriate for the intended application domain of real-time control, in particular network node management. Obviously, code is moved only once in this scheme, and neither communication (trivial) nor security (impossible) is an issue here.

In this thesis we will identify the concept of mobile code with the *ability to move an already started computation to another execution context and continue the execution there*. The nature of the information moved along with the computation and the way in which it is continued are what make the spectrum of mobility techniques so wide. However, it is possible to distinguish two general approaches that classify all the techniques in those supporting *weak mobility* and those supporting *strong mobility* [Cabri00].

The concept of *weak mobility* denotes techniques based on partial resumption of code execution. This means that the target execution environment restores the computation according to a set of information constituting the state of code prior to its migration. The
state allows the system to restore the execution by cooperating with the developer which normally is offered a callback to handle the restoration process. This kind of techniques are considered weak because do not continue execution as if nothing happened, but they normally restart the execution by configuring it with the state moved along with the code. Weak mobility techniques are normally simpler to implement but they demand some responsibilities to the developers which have to implement callbacks executed by the system during the restoration process. Another option is to design the code and the state subject to migration in order to automatically handle restoration.

Strong mobility [Muhammad04] refers to that class of migration techniques that are completely transparent to the moved code and that guarantee execution to be restarted at the same point where it was interrupted. For this kind of techniques the granularity is at instruction level, this means that the migration process can be completely hidden within a pause-wake up interval that is normally performed by the CPU scheduler. Then, strong mobility techniques interrupt the execution at the current instruction, move the code to the target site, restore the execution context, and restart the computation from the next instruction. Strong mobility techniques require the ability to capture the call stack connected with the code to be moved and all the information needed to restore local variables for each stack frame. Software infrastructures supporting strong mobility are more difficult to implement because they require complete access to the runtime image of the executing code and the ability to inspect the runtime environment. Other difficulties are represented by the management of the resources used by the code: this is an issue common to all mobility techniques but it requires more attention in the case of strong mobility in which the migration is performed in a transparent manner. The problem can be easily stated as follows: resources that are location sensitive cannot be available – or simply not reachable – in the target site. In the case of strong mobility, where migration can be performed without noticing the code being moved, the effects can be disruptive: whereas in the case of weak mobility the code moved is participating in the restoration process and can fix up references to resources, in the case of strong mobility this could not happen and the infrastructure has to provide an automatic
technique for fixing up references. This is, in general, a hard problem to solve and this is the reason why even in the case of strong mobility infrastructures it is necessary to opportunistically design the code.

Software infrastructures supporting code mobility are generally complex because they involve additional issues mainly concerning security and localization. Security problems are originated by the fact that the code being executed can cross the borders of a security domain and enter into another one with different rules and permissions. Security problems are connected with the trust relationship between the executed code and the execution environment: it is important to prevent that either migrated code could harm the hosting environment or the execution host could inspect and modify migrated code in order to alter it with malicious purposes. Localization is another important issue especially in the case in which the migrated code needs to coordinate its execution with other entities. A portion of code migrating multiple times during its life cycle is subject to a continuous change of its address, where the term address refers to any kind of information required to localize it.

All these consideration depicts code mobility as a complex feature to be successfully implemented. Hence, it is worth asking ourselves if it is really needed. Debates about the usefulness of code mobility have taken place in the last decade especially in the case of mobile agents. The results of these debates are not definitive and today code mobility is not considered a hot topic of distributed computing in general even though maintains its relevance. The main advantage of code mobility is the reduction of exchanged data among different sites that is obtained by moving the computation near the data instead of moving the data. This strategy is a valid option when the amount of data to move is reasonably bigger than the code using the data. Moreover, mobility infrastructures turn to be very useful to maintain the computational load of distributed systems: the ability to move processes on a different node without restarting their execution can be a very important resource to exploit in Computational Grids. The issues concerning load balancing will be later explored in the particular case of the mobility infrastructure devised for AgentService.
3.3.2. Mobile Agents and Mobile Agent Systems

The concept of mobile code seems to be an interesting support to program dynamic intelligent agent systems. Software agents give a level of abstraction in which the concept of mobile code can be easily introduced. Agents are autonomous, pro-active, and socially able: the ability to move and to migrate between different nodes of the community enhances the previously cited features. A definition which sufficiently characterizes the essence of a mobile agent system has been proposed by Chess and Nwana [Chess95, Nwana96]:

_a mobile agent is a software entity which exists in a software environment. It inherits some of the characteristics of an agent. A mobile agent must contain all of the following models: an agent model, a life-cycle model, a computational model, a security model, a communication model and finally a navigation model._

A more practical characterization of the concept has been proposed by Peine [Peine02]:

_A mobile agent is a process with the ability to move itself during its own execution in all essential parts from one host machine to another within a heterogeneous network while preserving its internal state._

One of the first elements emerging from these definitions is the concept of location awareness: differently from traditional mobile code systems that provide the ability of moving executing code at system level, mobile software agents are aware of their location and can autonomously choose the next place to move to. Software agents know at any time during its execution on what machine they are currently running, and they may base their further actions on this knowledge. _Location awareness_ is in conflict with the common trend in distributed operating systems [Tanenbaum85] and distributed systems in general, which is _location transparency_ or the property of performing computations without having to take into account their location or knowing their distribution at all. In the case of mobile software agents, location awareness can be profitably exploited to pursue a goal. The common properties ascribed to software agents – namely autonomy, pro-activity, reactivity, and social ability – are enhanced by the ability of changing the
location of their computation, thus obtaining more flexible computing systems. Moreover, the last characterization of mobile agents, recalls the attention on what is moved: during migration only the essential part of a software agent have to be necessarily moved. The essential part constitutes all the elements that are required by the software agent to continue after migration. Another important element is division of the agent state into an internal and an external portion. The internal portion of the agent state is what necessarily has to be preserved and comprises all of its state which exists independent of any entity outside the agent, such that it could be changed without necessarily requiring a corresponding change in any outside object. The external state, conversely, comprises those parts of the state which do correspond to some outside entity. The migration strategies have to take into account both of the two portions: whereas the internal state can be represented by loop counters variables, the external state can be constituted by handles to open files or active connections to databases. The two portions lead to different kind of problems when devising a migration strategy that have to be solved. In general, we will see that practical implementations are mainly focused on the preservation of the internal state.

As discussed in Chapter 2, section 3, the environment is a fundamental element for software agents. Mobile agents require a suitable runtime environment that provides a transfer service allowing them to move from one node to another: this environment is built on top of a host system and is normally integrated into the run-time environment usually required for agent execution. In the case of mobile agents these environments have to implement additional features in order to promote portability of agents across different systems, and guarantee security and protection against malicious code. The possibility of having a sort of middleware managing the execution of software agents on a specific host system is definitely helpful for implementing mobile agent infrastructures. Multi-agent systems provide a virtual-machine like approach to code execution: this implies that the middleware we were previously talking about act like a real virtual execution environment. Many agent programming framework use the Java technology for implementation, in this case the virtual execution environment can coincide with the underlying virtual machine or, more commonly, constitute an additional layer over it.
Other frameworks can adopt different technologies and rely on interpretation of agent code; in this case the runtime environment is basically provided by the interpreter and the services it offers. Despite the specific technological solution adopted it is always possible to assume the presence of an abstraction layer between software agents and the host system. This layer manages the execution of software agents, applies security policies, and specifies the code format required for the execution of software agents. The presence of such layer simplifies the development of mobility infrastructures since it reduces the heterogeneity of the execution environments on different systems, provide means to enforce security with techniques like code verification and resource access control, and support portability.

This is also the solution provided by the AgentService framework, but before discussing the mobility infrastructure provided with AgentService I will briefly review the most notable agent systems supporting mobility.

### 3.3.3. Related Works

The first system that introduced mobility support for software agents was Telescript [Wayner94], which also coined the term mobile agent. Telescript was a commercial endeavour by General Magic, a company founded by various telecommunication service and hardware vendors in 1991. Initially maintained as secret with the aim of licensing it as a proprietary technology, Telescript development tools and documentation were released in public in 1996. Telescript consists of a language and virtual machine which runs the compiled code. The Telescript language was clearly influenced by C++, as far as basic programming and object orientation concepts are concerned, but adds certain safety and security features aimed at the secure execution of untrusted code, while at the same time removing certain unsafe features from C++. It has been an inspiring model for the Java technology especially for what regards memory management, inheritance of classes and interfaces, and dynamic class loading. In addition, Telescript included further safety and security concepts at language level. The Telescript run-time system offers an extraordinarily rich set of security mechanisms, adding further concepts to the above like
device-based (not agent-based) authentication, hierarchical agent reception authorization, and various library classes for security. All this can be used to build a highly flexible host security regime on. For what concerns mobility, Telescript support the strong migration of the agent internal state. The target site of a migration can be either explicitly named, or implicitly by naming another agent to meet. Besides migrating the current agent, there is also an analogous Telescript command (send) to make another, fresh agent start at a remote site. Telescript uses a proprietary, authenticated and encrypted transport protocol for agent migration. Telescript was not only the first but also one of the most comprehensive mobile systems. Despite this, its excessive complexity and the high requirements in terms of memory for the time, Telescript never established as standard technology for mobile agent systems. Nonetheless, many of the concepts introduced with this system appeared later in other mobile agent system implementations.

The Agent-Tcl system [Gray96] is a mobile agent system initially presented in 1994 and developed in the following years. Agent-Tcl was initially based on the Tcl scripting language; it was later extended to include other programming languages such as Scheme and Java and renamed to D'Agents, acronym for distributed agents. D'Agents provides a strong migration service and different communication techniques for software agents which are implemented as UNIX operating system processes. Within each process the execution is controlled by the specific language interpreter that calls through IPC a specific process called Agent Server whenever any mobile agent systems functionalities is required. As previously discussed in Chapter 4, section 3.2, this is an unusual solution because of the loss of performances of IPC. Despite this, the system designed with D'Agents fully delegates some fundamental tasks of a multi-agent system – such as agent scheduling and communication – to the underlying operating system, thus relying a well tested and solid implementation. D'Agents supports strong migration, moving the complete internal state (and no external state) of the agent. Additionally, there is a function to fork, that is, duplicate the current agent, and another to send an agent program to make another, fresh agent start at a remote site. Sites are addressed by their
DNS names. As in many systems supporting strong migration, D'Agents also utilizes the implied thread state capturing functionality in order to provide a full-state checkpointing function.

Mole [Baumann98] was probably the earliest mobile agent system based on the newly appeared Java platform in 1995. Mole supports weak migration. On issuing a migration, with the destination location as an argument, the mobility infrastructure will call the stop method of the agent, which the developer may override to perform application-dependent migration preparations. The agent is then subjected to standard Java object serialization (which completely captures the state of the agent), transferred to the destination location, and restarted. On restarting, the agent's execution is always resumed in its start method (this also where the agents begin initial execution). Mole pioneered the use of Java for mobile agent systems, and has its strong point in the flexible, powerful, and convenient support for remote communication and distributed programming. From the security point of view, Mole does not provide many features (authentication, for instance, seems to be completely lacking), and the whole design appears somewhat incomplete and inconsistent. Nevertheless, the Mole research explored many important issues in mobile agent system design.

The Aglets system [Lange98] developed at the IBM research laboratory in Tokyo since 1996 was, next to the Mole system, among the very first Java-based mobile agent systems. Aglets aims to provide a simple-to-understand and easy to use tool for experiments in mobile agent programming. The basic execution model of an aglet (as agents are called in this system) is based, as the name already hints, on that of the standard Java applet. The execution model of aglets is event-based; programming consists of defining handlers for certain types of events, to be invoked by the system whenever the corresponding event occurs. Aglets supports weak migration only, using standard Java object serialization like most Java-based implementations, and will always resume execution after migration in the run method of the agent. A migration invocation implicitly refers to the current agent in nearly all MAS; Aglets, however, also supports migrating another agent, an operation
which can be easily accommodated in the event-based execution model of Aglets (and less so in other, more linear execution models). In any case, the migration destination is identified by a URL, which seems to be simply a host name in most practical cases. Alternatively, a sequence of destinations may be assembled in a separate object similar to the itinerary object of Concordia (see next paragraph). Aglets has done much to spread the idea of mobile agent systems. It is generally viewed as the most well-known practical mobile agent system and has attracted a large user base, probably due to its timely appearance, the simplicity of its API, and the convenient graphical tools. IBM lost interest in the Aglets project later, and the system was transferred from a proprietary, closed-source project to an open source project which is still in active community development to date.

The Concordia mobile agent system was introduced by Mitsubishi Electric in 1997 [Wong97]. It is a typical Java-based system, running on top of a standard Java platform and supporting only Java for agent programming. Concordia particularly stresses reliability, security, and distributed cooperation, by offering more and higher-level communication concepts than most MAS, built on top of commonly used lower-level system functions for object persistence and transactional message queuing. Concordia supports weak migration only. Arguments to a migration invocation are the target site (which seems to be a DNS name), and the name of an arbitrary Java method in the agent code to be called as the re-entry point after migration. The migration arguments are assembled in a separate object called itinerary, which is also accessible outside the agent. The whole agent migration data is transferred using the queuing service, which will retry the transmission in case of network partitions until it succeeds, and a two-phase-commit protocol, writing persistent copies of the migrating agent on either end of the connection. This ensures an eventually successful and atomic migration, although at the cost of writing two persistent copies. There is no option for a light-weight migration without the persistence overhead. On the whole, Concordia provides an impressive functionality, in particular in the area of distributed application support as realized by the transactional queuing, multicast, and collaboration functions. The security design,
however, appears less convincing. No applications of Concordia have become known so
far, although the system is still being offered (though not commercially, as it seems) by
the vendor.

Voyager [Glass99] is a Java-based mobile agent system tightly integrated with CORBA,
which might as well be viewed as a CORBA ORB augmented by weak object mobility.
Voyager supports an elegant remote method invocation using local Java syntax (including
constructor methods and exceptions) by means of proxy objects hiding calls to the
standard Java remote method invocation API. Proxy classes are generated automatically
from the corresponding application object classes. Weak migration is supported, giving a
re-entry method as an argument as in Concordia, as well as a host name or (proxy) object
as the destination. Further features include multicast messages and checkpointing of
objects and agents. Security does not exceed standard Java mechanisms. Voyager has
been a rather successful commercial product since 1997; however, the mobile agent
aspect of Voyager appears to have been downplayed over the years and today plays only a
very minor role in a product which is marketed by the vendor Recursion Software
basically as a value-added CORBA ORB.

ARA (Agents for Remote Action) [Peine02] is a mobile agent system supporting strong
mobility of software agents. The main rationale design and implementation goal of ARA
is to incrementally add mobility to programming. That is, to integrate it as comfortably,
and not intrusively, as possible with existing programming concepts – algorithms,
languages, programs, and operating systems. ARA provides a system that basically adds
runtime support for code mobility to different host programming languages. In order to
implement this feature the ARA system has been equipped with modified interpreters of
Tcl, C++, and Java able to completely capture the execution context of an agent program
and migrate it to another machine. Hence, ARA does not define a specific programming
language for implementing agent but it can virtually instrument a generic agent program
with strong mobility, given an interpreter of its source or compiled code that is
compliant with the ARA requirements. Along with strong mobility and language
independence a particular attention was geared toward efficiency and for this reason ARA avoids remote coupling that is remote communication among agents. Finally, security is one of the core features of ARA, which applies either memory protection techniques or host security design through fine grained and flexible policies.

The JADE development board introduced the mobility support into the JADE agent programming framework since 2003. The mobility infrastructure supports weak migration and relies on Java object serialization to capture the state of the agent. JADE implements only intra-platform mobility: software agents can move across agent containers located on different host and belonging to the same platform. Software agents can either migrate to a different container or clone themselves there. The JADE framework provides a rich set of APIs to support mobility: agents can ask for migration (or cloning) by using a specific method of the Agent class along with a Location parameter specifying the destination or by sending a message to Agent Management System (AMS). The first method is similar to the one implemented in Concordia with the only difference that JADE does not allow arbitrary destinations but just those registered with the AMS. The second technique is more agent-based and constitutes an innovation: JADE provides a specific ontology\textsuperscript{22} for mobility that must to be used to require migration or cloning. Despite the particular use to migrate or clone agents, developers are provided with handlers in order to prepare the agent before migration or cloning (beforeMove() and beforeClone()) and resume it after the process is completed (afterMove() and afterClone()). This approach is similar to the one implemented in Aglets, and it is mostly used to manage the references to local resources that will not be valid anymore after leaving the original container. Moreover, software agents can query the AMS in order to discover the location of a specific agent.

\textsuperscript{22} The meaning of ontology and its use within multi-agent systems will be explained in the next chapter.
The Tracy Toolkit [Braun05] is an extendable Java-based framework for agent toolkits
designed to be the least common denominator of all agent toolkits. It is based on a
kernel, on top of the JVM, and additional services which are implemented with a plug-in
architecture. The Tracy Toolkit provides support for mobility as an optional service. The
mobility infrastructure is based on the Kalong mobility model and implements a weak
migration service. Kalong is a virtual machine for agent mobility providing the basic
commands to implement different migration strategies. Kalong is focused on optimizing
the bandwidth usage by adopting adaptive transmission of code and data, dynamic code
servers and mirrors, and code caches to prevent class transmission. Upon this
architecture designers can implement their own migration strategies.

3.3.4. AgentService Mobility Infrastructure

AgentService implements a FIPA compliant weak migration strategy. The introduction of
the mobility service has implied first the extension of the agent model as suggested by
FIPA, and then the implementation of the mobile agent system within the Mobility Module.

The extension of the agent model has concerned, as suggested by Nwana [Nwana96], the
introduction of a navigation model maintaining the information required for migration.
The navigation model extends the agent model with additional features implemented by
following the FIPA specification. This specification introduces in the common life cycle
of an agent the transit state and two actions to enter and leave that state (move and
execute). The state machine describing the update life cycle of the agent is represented in
Figure 14. The agent itself can require the move action while the platform, through the
AMS, is responsible of completing the migration by performing the execute operation. At
the same time software agents can be moved for administrative purposes by the AMS or
the platform administrators. The mobility service has been implemented as additional
service and it has been added to the framework later. This approach is the same adopted
by the Tracy Toolkit with implements the mobility service as an optional plug-in and
follows the philosophy suggested with ARA that incrementally adds the mobility feature
to a pre-existing system. The original implementation of the agent model already
exposed the move action but it was not implemented, hence, move requests from software agents were simply ignored. The extension of the agent model provided an implementation for the move, and then the execute operations and integrated seamlessly with the preexisting code-base.

As previously said, mobility in AgentService concerns the weak mobility concept. The mobile agent environment captures the state of the agent maintained in the knowledge objects and gets a snapshot of the running behavior objects. Being a weak migration technique, the execution context of each behavior object – namely the call stack and the stack frame – is not captured, but only the information of the type and the names of running instances are preserved. These data, along with the knowledge objects, constitute the internal state of the agent that is moved to the destination. On the target site, the agent instance is resumed: the framework automatically recreates the internal state of the agent by configuring the knowledge objects with the persisted data and by starting the previously running behavior objects connected to the proper knowledge
objects. Moreover, developers are provided with an additional method that is called by the framework during agent restoration and that can be used to customize the agent state reconstruction. This technique provides a high degree of flexibility since – as happens in Aglets, JADE, and Concordia – developers can participate into the restoration process and take advantage of local information to restore the state of the agent. At the same time, the automation level of the process is good enough to avoid the programmer’s intervention: other systems implementing weak mobility just start the activity of the agent from the beginning; the mobility infrastructure of AgentService reactivates all the activities that were previously running and configures them with the state saved before migration. This solution is a little bit closer than others to strong mobility implementations.

The mobile agent system is made up by the collaboration of the AMS and the Mobility Module: the former manages the life cycle of agents and mostly takes care of the bureaucratic issues of migration while the latter actually performs the transfer of all the necessary data between the source and the target platform. The migration process, which can be either activated by a software agent or for administrative issues, is depicted in Figure 15, and described as follows:

- **negotiate**: the AMS of the source target contacts the AMS of the target platform and asks if the agent can be moved;
- **stop and persist**: if the agent can be moved, the AMS stops its activity, persists its state, and puts it into the transit state (move action);
- **transfer**: the AMS instruct the mobility module to move the agent. The state of the agent all the knowledge objects and the state of execution of behavior instances is transferred to the target platform. This operation may require the transfer of the assemblies describing the types of the agent or used by it;
- **restore**: on the target platform the mobility module notifies the AMS that the transfer is completed. The AMS creates an instance of the same type of the agent
received sets its state and invokes the Resume method allowing the programmer to customize the re-activation of the agent;

- *execute*: the agent changes its state from *transit* into its original state, the AMS of the source platform is notified of the successful transfer and the agent is activated (*execute* action).

![Figure 15. Agent migration process](image)

The process previously described is implementation independent and hides the details of the transfer within the *Mobility Module*. *AgentService* defines the *IMobilityModule* interface, which represents the contract that modules have to implement in order to integrate a mobility service in the platform. As happens for the *Messaging Module*, the simplest deployment of a mobility infrastructure is constituted by the installation of the same mobility module on all the platforms that want to join the *virtual platform*. To be true, this restriction can be relaxed and it is possible to have different modules that necessarily have to share the migration protocol. By using this technique it is possible to leave untouched the *Agent Management System* that will always interact with modules through the *IMobilityModule* interface. Migration can be then implemented by using web services,
a normal ftp, or a custom channel. Mobility modules basically have to implement a Move method and expose an event that notifying the AMS the successful transfer. The Move operation accepts an AgentProfile, which basically maintains all the information to restore the agent execution on the target site, an AgentPersistent instance, which stores the persisted version of the agent, and the information about the target destination.

The presence of the AgentPersistent instance underlines the strict connection between the persistence and the mobility infrastructures within AgentService. It can be noticed that the steps involved in the transfer of an agent are really similar to the ones required to persist an agent in the local storage of the platform and resume its state when the platform starts again. Like the persistence service, the mobility service is characterized by the capability to extend the life cycle of an agent instance beyond the one of the operating system process containing it. In both cases the information about the state of execution need to be saved in order to restore it later. The mobility service just adds the capability to restore the state of execution on different site, thus implying the migration of the information required to resume the execution. For this reason, the integration of the mobility infrastructure in AgentService has been simple and seamless: most of the additional work performed by the mobility module involves the localization of the agent and not the maintenance of its state. The model adopted by AgentService to define an agent greatly simplifies the work of the mobility module. Thanks to the clear separation among the state and the activities the maintenance of the execution state is obtained by saving all the knowledge objects composing the knowledge base of the agent and the status of the behavior objects. When the agent is restored the information saved are loaded into the new instance and all the activities previously stopped are started. In particular, the mobility infrastructure has to deal with the transfer of assemblies if the repositories of the two platforms are not synchronized.

The current implementation of the Mobility Module relies on the FTP service to migrate agents between the AgentService platforms. As showed in Figure 16, each module sets up a dedicated FTP server to receive data and uses an FTP client to send the agent data to the
target platform. The FTP server is managed by the mobility module which is notified whenever a file is transferred and interacts with the server in order to receive all the data required for agent restoration. The architecture is able to handle multiple migration processes and provides a basic security based on authentication and authorization that can be customized for each platform. For each platform belonging to the virtual platform the FTP server creates a folder in which contains a specific folder for each agent moved. Each client is allowed to browse and modify only the folder of the FTP file system corresponding to their platform.

The implementation uses a modified version of the FTP protocol that comprises the following states:

- **Idle**: This is the initial state of the server and its normal state when no transfers are active.

- **WaitingForProfile**: When an incoming connection has been accepted and authenticated the server is waiting the AgentProfile instance containing information about the assemblies needed to execute the software agent.

- **RetrieveAssembliesList**: Upon the reception of the profile the target mobility module checks the presence of the required assemblies and creates a list of the missing assemblies by relying on the services provided by the storage module.

![Mobility Infrastructure](image)

*Figure 16. Mobility Infrastructure*
Figure 17. FTP Server automaton

- **WaitingForAssemblies**: If there is at least one missing assembly the FTP server goes into this state and waits for the required assemblies to be transferred by the client. On the client site of the migration process the mobility module will obtain the missing assemblies from the local storage module.

- **WaitingForAgent**: This step of the process involves the transfer of the AgentPersistent instance.

- **TransferComplete**: After the successful transfer of the AgentPersistent instance the server notifies the mobility module that all the data are ready to restore the software agent. The module picks up the persisted version of the agent along with its configuration data (basically the configuration file) and the connection with the client is closed.
If errors occur during the migration process, the server notifies the error, closes the connection, and puts itself in the *Idle* state. In this case the migration has to be restarted by the AMS of the source platform that is notified of the error.

The design of a mobility infrastructure cannot ignore some basic security issues concerned to the execution of code. The *AgentService* platform is a host for the agent instances which are designed and developed by third party; hence it must guarantee a reliable service and prevent the execution of malicious code. This problem becomes more critical when platform instances are allowed to move agents. The primary security level is provided by the *Storage Module* which can be configured to accept only signed assemblies. Digital signature on assemblies allows the platform to identify the code it executes. Assembly identification is a prerequisite to build a more sophisticated security framework which allows administrators to decide under which security profile execute agent instances. An additional module which takes care of user management can be installed in the platform. By using this module platform a user with the related code execution permission is associated to each agent instance. Hence a second level of security can be added by customizing the execution of migrated agents in the target platform with a specific execution profile: platform administrators can set up complex policies and restrict the permissions granted to agents transferred from other platform, or agents of a specific type. The installation of such module is optional and then they are available only if the module is present. This is a flexible solution since installations which do not require particular attentions to security issues will not comprehend this module but administrators will still be able to decide to accept mobility and eventually assign a default user profile to mobile agents. The customization level, available with the user management module installation, is the same provided by the operating system. It can be noticed that user management and then security management is something that seamlessly integrates with the existing framework: each agent is executed within the context of a specific Application Domain, and Application Domain can be configured to be executed with the permission of a user different from the one that owns the process. The default behavior inherits the user profiles and then the permission set from the
owning process. Finally, the entire process that allows mobility of agents takes place if and only if the platforms are allowed to transfer agents; otherwise it stops during the negotiation phase. The AMS of the source platform will look in the platform configuration to determine if it is allowed exporting agents, while the AMS of the target platform will check if it is allowed to import agents. This information is contained in the platform profile and administrators can customize the platform behavior by modifying the profile in the configuration file.

3.3.5. Observations

The infrastructure provided for the mobility of agents in AgentService is very flexible and provides the basic service of weak migration. The AgentService framework provides a general infrastructure of the mobility architecture and a default implementation of it. By abstracting the channel and the protocol that is used to actually move the code, the framework leaves to developers the freedom to implement a more sophisticated support for mobility. As opposed to other programming frameworks, which are not designed with modularity in mind, the introduction of the mobility infrastructure did not require important changes to the preexisting code-base and its design fosters the principle of seamless integration. New implementations and protocols can be adopted by providing a module that adheres to the contract defined by the IMobilityModule interface, integrated into the system without changing the behavior of the Agent Management Systems, and without taking care of the security infrastructure that is implemented as an additional layer.

Along with the messaging system the mobility infrastructure constitute a solid foundation upon which build more complex services, some of them will be described in the next section.
4. Building Complex Services: The Case of Load Balancing

4.1. Overview

In this section I will briefly present one example of a service relying on the support provided by AgentService to build distributed systems. This is the Load Balancing Service that applies to a collection of AgentService platform instances which federate together in order to constitute a virtual platform where agents can freely move. The Load Balancing Service takes care of distributing the computational load of executing software agents among the platform instances. In order to do so it exploits the ability of moving software agents among AgentService platforms provided by the mobility infrastructure. The Load Balancing Service adopts an on-line strategy; that is, it operates when software agents are already running. A different approach is adopting a preemptive strategy; that is, dispatching the creation of a software agent to a selected platform instance according to the global overhead of the virtual platform. By using an on-line approach it is possible to integrate the solution with the preexisting system without changing the code-base. Thanks to the modular architecture of AgentService, the Load Balancing Service is implemented as an additional module and uses the services offered by the AMS to move agents. Once the modules are installed and the collection of corresponding platforms is configured as a virtual platform, the maintenance of the overall computational load of the systems is delegated to the Load Balancing Service, which operates in parallel with the preexisting system and in a completely transparent manner.

In the remainder of section of I will introduce the concept of virtual platform that lies behind the implementation of load balancing. Then, I will describe the concept of balancing policies that are the means through which different balancing strategies can be implemented.
4.2. System Architecture: Virtual Platforms

The virtual platform defines the boundaries in which the mobility of agents and then the Load Balancing Service applies. A virtual platform is a federation of platforms that join together to create a virtual environment in which localization of software agents is guaranteed and migration is allowed. Whereas the concept of virtual platform is something not explicitly required for the mobility infrastructure, it becomes fundamental in the case of load balancing in which platform have to be configured in order to accept incoming software agents and freely execute them. Actually, one of the requirements for load balancing is that each platform joining the federation provides the same service for executing agents; otherwise it would not be possible to move agent instances among them. This is necessary because load balancing applies as a consequence of the system load and not as a response of an agent request; hence, it is not a decision of software agents to move to another platform. At first sight this could seem a constraint that limits the agent autonomy. Actually the concept of virtual platform refers to a single AgentService platform that is scattered on multiple networked hosts; hence, the act of moving among these hosts is just a technical aspect.

AgentService virtual platforms are created by the cooperative work of Load Balancing Modules and based on a master-slave architecture. The federation of platform constituting the virtual platform is managed by a one of the platform that is identified as the master platform. The other platforms are defined as slave platforms. In order to become a component of the virtual platform platforms have to send a join message to the master platform, which is a fundamental requirement for the creation of the federation. On the master platform, the Load Balancing Module is configured to run in master mode and configuration file defining the static structure of virtual platforms is shared among all the modules and it is updated according to the topographical changes of the virtual platform. On slave platforms, the Load Balancing Module runs in slave module and notifies the master module whenever a software agent is created or terminates its activity. This information is used by the master module for applying load balancing, which can eventually identify a software agent to move. If this happens, the master module instructs the platform
involved in the migration process by talking with their Load Balancing Modules. On the source platform the local balancing module will send a message to in order to instruct the AMS to start the migration process.

The decision about which agents to move and the determination of their destination is encapsulated within the Load Balancing Policies that will be explored in the next paragraph.

### 4.3. Load Balancing Policies

Load Balancing Policies are the component of the architecture encapsulating the decisional logic of the balancing system. In order to be open and flexible the system abstracts the concept of balancing policy and implements a plug-in architecture in which policies are dynamically loaded at runtime and have to adhere to the contract defined by the ILBP interface. This solution gives the maximum freedom to developers which can implement their own custom policies and integrate them in the system without efforts.

```java
interface ILBP
{
    PolicyDescription Description;
    void ConsumeEvent(PlatformEvent event);
    void GetNextAgentToMove(out AID aid, out PlatformDescription target);
    void AddProfile(PlatformProfile profile);
    void RemoveProfile(PlatformProfile profile);
    void UpdateProfile(PlatformProfile profile);
}
```

Figure 18. Interface for load balancing policies

Figure 18 describes the interface ILBP. By looking at the methods exposed through the interface it is possible to characterize the role of policies within the system. The principal purpose of policies is determining which agents have to be moved; this task is accomplished by the GetNextAgentToMove method. The method returns the AID of the selected agent and specifies its destination. In order select the right agent the policy
needs to know what happens in the system; in particular it has to be notified of interesting events happened in the virtual platform and to maintain updated its topology. The master module forwards all the interesting events to the policy by calling the `ConsumeEvent` and forwards any topological change by calling the `Add/Remove/UpdateProfile` methods.

The system provides two simple policies ready to be used. The first policy distributes the number of software agents among the federated platform equally without considering the specific type of agent. This policy provides a very simple balancing method that applies to a set of homogeneous platforms running almost the same types of software agents. The second policy minimizes the remote interactions by moving software agents that frequently exchange messages into the same platform. This policy can be useful when time constraints are applied to the execution of software agents.

### 4.4. Observations

Thanks to the modular architecture of the platform the load balancing service has been implemented very easily. The possibility of relying on preexisting – like mobility and remote communication – has made the implementation of the load balancing module lighter and more focused on balancing than other secondary aspects. Preliminary tests of the architecture revealed that the most onerous operation during the mobility process is the transfer of assemblies required for agent activities. This is a minor drawback since in common balancing scenarios all the nodes are likely to run the same agent types; hence there is no need for huge assembly transfers.

### 5. Conclusions

Distributed programming with `AgentService` is principally based on remote interaction between software agents and agent migration among `AgentService` platforms. These services constitute the basic bricks that can be used to compose more complex systems and services as demonstrated by the implementation of the `Load Balancing Service`. The
main goals behind the design and the implementation of remote interaction and agent mobility are a seamless integration with the preexisting system and a high degree of flexibility given to third parties in substituting the default implementations of these components with custom solutions.

The seamless integration allows developers to use the newly integrated services without additional efforts. This is particularly true for the messaging system: during the period covering my doctoral studies three different implementations of this service have been provided. When the system has been enhanced with the ability to interact with remote agents its use has been immediate and did not required additional effort from the programmer’s point of view. The current implementation of the messaging system relies on Web Services and the use of such standard technology allows an easy integration of AgentService with others applications either they are .NET based or not. The integration of applications based on different paradigms and technologies is one of the most important issues in open and distributed systems, which are the target of the AgentService framework. The support provided by AgentService for integration develops on both sides: from the agent point of view a uniform approach to the local and remote interaction is given, this is based on the use of messaging system; from the infrastructure point of view a high degree of freedom is given to software engineers that can supply different implementations of the messaging system in order to make interact software agents with other technologies. Even though this solution is not a panacea for all the problems in distributed computing, it provides a strong help for overcoming some of them.

Agent mobility is the other pillar of distributed computing with AgentService. In a certain way code mobility is an alternative to remote interaction: instead of making entities interact remotely they are concentrated into the same place by moving one of the two into the system process of the other. Defining and implementing an infrastructure supporting code mobility is a more complex task than devising an architecture for remote interaction. Moreover, in the case of mobility support for software agents, additional issues have to be considered. The solution proposed with the AgentService framework is based on a weak migration and provides some additional features for state
restoration and agent resumption. This service has been implemented by exploiting the lessons learnt from the implementation of the persistence subsystem on the interruption of agent execution and its further resumption. The agent model based on the knowledge and behavior object has simplified the implementation first of the persistence subsystem and then of the mobility infrastructure. The solution provided with AgentService allows a simple customization of the migration process that is abstracted away into the implementation of the IMobilityModule interface by a specific mobility module. Again, even in the case of mobility the implementation of this service is completely transparent to its use that remains unchanged.

Mobility can be useful to implement additional services but introduces new problems. The first is constituted by the restoration of the state on a different place in which the context of execution is not the same. As many architectures supporting weak mobility do, the AgentService framework provides means to customize agent restoration and give a chance to developers for solving this kind of problems. Another important aspect concerning mobility is agent localization. Agent localization consists in being able to identify agent location univocally. This information is contained in the Agent IDentifier (AID) which qualifies an agent with the instance name and the platform name. When an agent migrates to another platform the AID changes and the community needs to be notified of this change. The solution implemented in AgentService uses resolvers. A resolver is a specific agent that can be contacted for delivery problems. The AID maintains a list of resolvers can be contacted to resolve the new address of software agent and deliver to it messages. This is the technique suggested by the FIPA guidelines for agent mobility and constitutes a partial solution of the problem that becomes even more frustrating when agents change their location frequently. This behavior can be the side effect of the activity of the load balancing service that is applying a policy which is not adapt to the conditions of the system. Finally, interoperation in the case of mobility is very hard task to implement: AgentService does not support the migration of software agents on different platforms. This is a common drawback of all the mobility infrastructure reviewed in this chapter and does not have a solution in general: the FIPA guidelines do not provide any
standard for a portable representation of software agents neither defines a detailed agent architecture. Moreover, different implementation technologies constitute a barrier that is very hard to remove. Whereas it is possible to agree on standard protocol and language for interoperation, it is almost impossible to migrate an agent instance to a different runtime system, especially in the case of strong migration. Hence, solutions for this issue are by far less general than in the case of remote interaction and not deeply investigated.

Interoperation among different systems and technologies is one of the most challenging tasks in distributed computing. The architecture of the AgentService framework has been designed with modularity and flexibility in mind in order to allow incremental development of features. This property, which mostly relies in hiding the implementation details within modules and using them to add new services, allows an easy integration of different technologies that are used within the multi-agent system as services. Some experiments on interoperation have been carried out during the last period of my doctoral studies. In particular I investigated the possibility of integrating AgentService with Alchemi [Luther05] that is a .NET based infrastructure for Enterprise Grid Computing. The use of the same technology has greatly simplified interoperation. The outcome of this study has been a prototypal architecture, devised with the researchers at GRIDS Lab at the University of Melbourne, in which AgentService platforms can join the Grid and provide computational power to execute grid jobs as any other node of the Grid. The integration has been performed by designing a specific module which takes care of communicating with the manager of the Grid and demands to software specific software agents the jobs that have to be executed. The prototype proved to be an interesting starting point to work on and demonstrated that flexibility of the AgentService framework.
This chapter is the result of a co-joint work with Andrea Passadore who has been mainly engaged in implementing the ontology object model in AgentService and the support for interaction protocols.

1. **Introduction**

Even though there is huge class of applications domains in which the single software agent is enough to model the software solution – softbots, personal assistants, and e-mail filtering clients – the power of the agent abstraction is completely exploited when agents use their *social ability* that is, when they organize themselves in a community and interact each other in order to pursue their goal. Given this, agents have to use effective methods to interact and to exploit at best their membership to the community of agents. It can be observed that the scenario is challenging: multi-agent systems are open and dynamic systems where agents can join and leave at any given instant of time. Moreover, multi-
agent systems are normally composed by heterogeneous agents that could pursue the same goals with different strategies and knowledge base. The problem is actually twofold: it is necessary to establish a common way of representing the knowledge base and it is necessary to find a formal way to describe interactions. In addition, any kind of solution devised for the problem should be not limiting and flexible enough to represent a wide range of possibilities. Formal representations can then provide a solution to the problem: they supply abstract models, which are general enough to characterize a wide variety of domains. Ontologies are formal representations of a general knowledge base, and protocols are formal representations of interactions.

In this chapter I will describe the support provided by the AgentService framework for designing, and managing ontologies and interaction protocols. I will start by describing what ontologies are and how they can be used in Computer Science, and consequently with AgentService. I will then introduce the notion of protocol, discuss the value of using protocols for formalizing interactions, and describe the support provided by AgentService for modeling, developing, and using interaction protocols with software agents.

2. Ontologies

2.1. Definition

The Webster’s Third New International Dictionary [Gove02] provides the following definition of ontology:

1. A science or study of being; specifically, a branch of metaphysics relating to the nature and relations of being; a particular system according to which problems of the nature of being are investigated; first philosophy.

2. A theory concerning the kinds of entities and specifically the kinds of abstract entities that are to be admitted to a language system.
While the first definition normally applies to the philosophical tradition, the second one is more likely to be used in Artificial Intelligence and Knowledge Representation. Perhaps, the most famous definition of ontology in Computer Science is the one given by Tom Gruber, researcher associate at the Stanford Knowledge Systems Lab and innovator in technologies extending the human experience:

An ontology is an explicit specification of a conceptualization.

The term is borrowed from philosophy, where an ontology is a systematic account of Existence. For knowledge-based systems, what “exists” is exactly that which can be represented. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. This set of objects, and the describable relationships among them, are reflected in the representational vocabulary with which a knowledge-based program represents knowledge. Thus, we can describe the ontology of a program by defining a set of representational terms. In such an ontology, definitions associate the names of entities in the universe of discourse (e.g., classes, relations, functions, or other objects) with human-readable text describing what the names are meant to denote, and formal axioms that constrain the interpretation and well-formed use of these terms [Gruber93].

Despite some critique to this view [Guarino95], the definition given by Gruber captures the essential meaning of ontology in Computer Science, that is, by quoting from Wikipedia:

[an ontology is] ... a data model that represents a set of concepts within a domain and the relationships between those concepts. It is used to reason about the objects within that domain.

Ontologies are used in Artificial Intelligence, in the Semantic Web, in Software Engineering and in Information Architecture as a form of knowledge representation about the world or some part of it. They generally describe:

---

23 http://en.wikipedia.org/wiki/Ontology_(computer_science)
• **Individuals.** They represent the basic or 'ground level' objects. These can be either concrete or abstract objects. Individuals do not properly belong to ontologies, since these ones provide means for classifying, and do not consider directly, them.

• **Classes.** They are sets, collections, or types of objects. The main objective of classes is classification of instances (or individuals). Classes can be organized hierarchically through inheritance. Classes which belong to the same inheritance level and share the direct ancestor, actually create a partition of the ancestor.

• **Attributes.** They are properties, features, characteristics, or parameters that objects can have and share. Attributes can be primitive or complex and multi-value and are a fundamental requirement for ontologies. If classes do not exhibit attributes we would likely to talk about taxonomy rather than ontology.

• **Relationships.** They represent connections that relate one object to another. Typically relationships are expressed with attributes whose value corresponds to another object in the domain. This is the common technique which relates one object to another.

Relationships are the means to ascribe semantics to the domain that is modeled by an ontology. Two important relationships which contribute to organize the body of knowledge belonging to a given ontology are inheritance and meronymy (i.e. part-of, or composition). The use of these two relationships allows the definition of complex and accurate ontologies.
Figure 19 provides a graphical representation of an ontology: rounded rectangles represent classes, which contains attributes; links between classes identify inheritance relationships while between an attribute and another object are commonly the expression of meronymy. The ontology models the domain of marriage which is constituted by the concepts of Person, Man, Woman, Husband, and Wife. Person is identified by four simple attributes Name, Surname, Age and Sex. The concept of Person is then specialized in Man and Woman which constitute a partition of the concept Person according to the value of the Sex attribute. Man and Woman are further specialized in the concepts of Husband and Wife. These two concepts exhibit a pair of complex attributes which relate each other and express the relation of marriage.

2.2. Languages for Ontologies

Ontologies can be expressed more formally by using formal languages [Pérez02]. They can be classified according to the syntax and the structure. Such languages are normally expressed through an XML syntax, otherwise are based on predicate logic or descriptive logic. The community of researcher has proposed a plethora of languages for encoding ontologies. In the following I will briefly review the most widely used:
OKBC (Open Knowledge Base Connectivity) [Chaudhri98] is an application programming interface for accessing knowledge representation systems, and was developed to enable the construction of reusable knowledge base tools. OKBC has been promoted by DARPA with the aim of providing an integration protocol of different technologies.

KIF (Knowledge Information Exchange) [DARPA06 and Genesereth92] is a computer-oriented language for the interchange of knowledge among disparate programs. It has declarative semantics (i.e. the meaning of expressions in the representation can be understood without appeal to an interpreter for manipulating those expressions); it is logically comprehensive (i.e. it provides for the expression of arbitrary sentences in the first-order predicate calculus); it provides constructs for the representation of knowledge about the representation of knowledge, the representation of non-monotonic reasoning rules, and the definition of objects, functions, and relations.

DAML+OIL [Horrocks02]. It is a Web Ontology Language, resulting from a merge between DAML-ONT [McGuinness02] developed as part of the US DARPA Agent Markup Language (DAML) [Hendler00] programme and OIL (Ontology Inference Layer) [Fensel01] developed by a group of mostly European researchers. The language provides an extension to RDF\(^{24}\) and RDF-S\(^{25}\) and takes an object oriented approach enabling the standardised representation of classes, properties and inheritance relationships. It also allows restriction, unions and intersections. This makes the language more expressive and more accessible to automated processes than XML or RDF on which it is built.

---

\(^{24}\) RDF (Resource Description Framework)

\(^{25}\) RDF-S (Resource Description Framework Schema)
• **OWL (Ontology Web Language)** [Smith04] is an ontology mark-up language for publishing and sharing ontologies on the Internet. OWL is an extension of the RDF vocabulary and it is an evolution of other ontology languages (DAML and OIL). Along with RDF and other components it constitutes the basis for the Semantic Web.

OWL is composed by three languages whose expressiveness is increasing:

• **OWL Lite**: supports those users primarily needing a classification hierarchy and simple constraints. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1. It should be simpler to provide tool support for OWL Lite than its more expressive relatives, and OWL Lite provides a quick migration path for thesauri and other taxonomies.

• **OWL DL**: supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computed) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions. OWL DL is so named due to its correspondence with description logic, a field of research that has studied the logics that form the formal foundation of OWL.

• **OWL Full**: is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support complete reasoning for every feature of OWL Full.

Each of these sublanguages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded.

It can be noticed that among the different languages OWL is taking a prominent role, even because it is seen as a major technology for the future implementation of a Semantic
Web. It is playing an important role in an increasing number and range of applications, and is the focus of research into tools, reasoning techniques, formal foundations, and language extensions.

Finally, it can be observed that there are other ontology languages which have been proposed and gained interest in the community of researchers. Among them it is worth noting: OCML (Operational Conceptual Modeling Language), LOOM, CycL, F-Logic (Frame-Logic), KM (Knowledge Machine programming language), and XOL (eXentisble Out of bound Language).

2.3. **Ontology Development**

As noticed by Gruber [Gruber95] the development of ontologies should be driven by the following principles:

- **Clarity and Objectivity**: the ontology should provide the meaning of defined terms by providing objective definitions and also natural language documentation.

- **Completeness**: a definition expressed in terms of necessary and sufficient conditions is preferred over a partial definition (defined only through necessary or sufficient conditions).

- **Coherence**: inferences have to be consistent with the definitions.

- **Maximum monotonic extendibility**: new general or specialized terms should be included in the ontology in such a way that is does not require the revision of existing definitions.

- **Minimal ontological commitments**: to make as few claims as possible about the world being modelled, giving the parties committed to the ontology freedom to specialize and instantiate the ontology as required.

Moreover, Arpirez et al. [Arpirez98] underline the importance of standardization of names whenever is possible and Borgo et al. [Borgo96] stressed the importance of the concept of Ontological Distinction, which means that different classes belonging to the same ontology should be disjoint and refer to different elements or aspects of the domain.
These suggestions are a useful guide for developing an ontology, because they concentrate in few rules some well known practices that avoid defining incoherent models. When it is necessary to design ontologies for wide domains characterized by complex relationships and a huge number of entities, the activity of modelling the ontology becomes an engineering process. Then, a more structured and well organized approach is needed to obtain a successful result: it is necessary to adopt a systematic approach or methodology to define ontologies.

It can be observed that the need of methodologies or a systematic approach is real, because there are application scenarios in which ontologies could be useful to maintain and collect the related body of knowledge and which present the properties previously cited. The knowledge areas of biology and law are just two examples. Hence, in order to address this issue various methodologies have been proposed, the most important ones are those proposed by Uschold and King [Uschold96], Grüninger and Fox [Grüninger95], and Methontology [Pérez98].

2.4. Applications

Ontologies are formal representations of a domain of discourse, they capture the essential elements, the relations among them, and provide a corresponding abstract representation on which it is possible to reason about. Their applicability goes beyond the field of Computer Science and they can be virtually used to model any kind of domain in various contexts. This is a brief list of the different fields in which ontologies have been applied in order obtain a formal representation of the body of knowledge that resides in the corresponding domains:

- **Mission Critical Tasks.** The Advanced Concepts and Technology Division GE31 has started the project COSMOS (Coalition Secure Management and Operations System). The aim of COSMOS is providing the capability for multiple disparate Command and Control applications to share vital battlefield information within a coalition environment. It uses ontologies to improve information exchange with the aim of reducing confusion on the battlefield and avoiding ‘Friendly Fire’ incidents.
Law and Public Administration. The European project for Standardized Transparent Representations in order to Extend Legal Accessibility (Estrella, IST-027665) aims to develop and validate an open, standards-based platform allowing public administrations to develop and deploy comprehensive legal knowledge management solutions, without becoming dependent on proprietary products of particular vendors. Ontologies have been used to model and maintain a complete body of knowledge of the legal concepts that is represented by the LKIF-Core Ontology.

Antropometry. World Engineering Anthropometry Resource (WEAR) is an international collaborative effort to create a world wide resource of anthropometric data for a wide variety of engineering applications. Ontology have been applied to model the world wide resource of antropometric data and to make them reusable for the different engineering applications.

Genomics, Biology, and Medicine. The Bio-Informatic Research Network (BIRN) has started – by creating the Ontology Task Force (OTF) – many ontology related projects whose aim is to extensively use ontologies and eventually define new ones in order to maintain a reusable, computer manageable, body of knowledge for medicine, genomics, and biology. Some of the outcomes of the work of the OTF are Gene Ontology (GO), Neuronames, Systematized Nomenclature of Medicine Ontology (SNOMED), Mammalian Phenotype Ontology (MPO), and FuGo.

26 http://www.estrellaproject.org/lkif-core/
27 http://ovrt.nist.gov/projects/wear/
28 http://www.nbirn.net/research/ontology/index.shtm
29 http://www.nbirn.net/research/ontology/ontology_neuronames.shtm
30 http://www.nbirn.net/research/ontology/snomed.shtm
Cultural Heritage. CNOSO\textsuperscript{33} is a project funded by MIUR – Ministero dell’Istruzione, dell’Università e della Ricerca – Italy. The aim of the project is to define a technological platform for web learning focused on the Italian Cultural Heritage. It uses ontologies in order to model and maintain the body of knowledge concerning Cultural Heritage.

Historical Data Mining. Armadillo\textsuperscript{34} is a project that examines new ways of extracting (‘mining’) relevant information from unconnected electronic sources. Armadillo uses ontologies in the humanities to combine historical repositories for researcher investigation. It is an attempt to answer the question of how to locate and interpret information contained in distributive online research datasets effectively, using criteria acceptable to the Arts and Humanities community.

These are just few examples on the possible application of ontologies. I have deliberately avoided presenting the use of ontologies in Computer Science based application fields and this will be the topic of the next chapter when I will discuss the use of ontologies within multi-agent systems and the support provided by AgentService.

3. Ontologies and Multi-Agent Systems

3.1. The Need for a Meta-Ontology

Within a multi-agent system ontologies are used to establish a semantic agreement between agents. Ontologies can be profitably exploited to convey a meaningful

\[\text{http://www.nbirn.net/research/ontology/mammalian_ontology.shtm}\]

\[\text{http://obi.sourceforge.net/}\]

\[\text{http://dif.tno.it/cnosso/}\]

\[\text{http://www.hrionline.ac.uk/armadillo/}\]
interaction if and only if given an ontology there is an automatic process which allows software agents to learn its content and its structure. Hence, there must be a formal specification able to represent all the different kinds of ontologies that is always comprehensible by software agents. In other words, it is necessary to establish a meta-ontology, or better a foundational ontology related to the multi-agent system domain. A foundational ontology is a generic ontology describing generic concepts, which have no specific domain but are used to describe concepts within specific ontologies. A foundational ontology for multi-agent systems describes the abstractions that can be used to specify any kind of ontology within the MAS. This is the only ontology that software agents need to know in order to interoperate with other agents; by using the foundational ontology they can learn all the other ontologies\textsuperscript{35}.

3.2. FIPA Recommendations for Ontologies

3.2.1. Meta Ontology for Agent Interaction

According to Wooldridge et al. [Wooldridge95], agents interact at knowledge level; hence, as previously noted, it is necessary to have a standard specification for knowledge representation. The Foundation of Intelligent Physical Agents suggested the use of OKBC to describe the foundational ontology for multi-agent systems [FIPA86-01].

OKBC [Chaudhri98] adopts an object oriented abstraction for knowledge representation and provides the following elements to describe an ontology: constant, frame, slot, facet, class, and individual. These elements are used to define the meta-ontology that need to be shared by all the agents and whose concepts are illustrated in Figure 20.

\textsuperscript{35} At least they can learn the content, the structure, and the relationships within the entities of the domain. It is not possible to automatically guarantee that agents will learn the meaning.
These elements describe the basic bricks through which each ontology has to be modeled. Such representation allows software agent dynamically investigating the content of a new ontology and discovering the relations among the entities within it.

- **Predicate.** Predicates state a fact about the state of the world, which can be true or false.

- **Term.** Terms are entities existing in the world which agent talk and reason about. Terms can further distinguished into the following elements:
  - **Concept.** Concepts are complex entities that have attributes. Within a conversation among agents they cannot exist as stand-alone entities but should be used in conjunction with predicates or as properties of other concepts.
  - **Action** An action is specialization of a concept and represents an action that can be executed by an agent.
  - **Primitive.** They are simple entities for example primitive types like integers, strings....
  - **Aggregate.** Aggregates represent group of entities.
• **Query.** Also known as *Identifying Referential Expression (IRE)*, a query is an expression whose aim is to identify those entities which satisfy the given predicate.

• **Variable.** They are expressions used to identify elements not known a priori.

Agents then articulate their interactions by using the elements defined in this ontology which are further specialized in the specific ontologies. This means for examples that a query will be a *specific* query, which a *specific* type of predicate and will make sense in a *specific* domain of discourse. We can observe that, since agent interactions take place at knowledge level, *only actions, queries, and predicates* can be sent as content of an agent message. The delivery of a concept does not imply anything because if it is part of the ontology it is already known, if it is not part it cannot be understood. Moreover, a concept does not tell anything additional about the state of the world, whereas predicates, actions, and queries correspondingly identify facts, gent decisions, and requests.

### 3.2.2. Agent Communication Language

FIPA recommends the use of the *FIPA-ACL* [FIPA61-02] communication language in order to express the content of agent messages. The rationale behind this is the fact that *FIPA-ACL* is based on the SL language family [FIPA08-02]. These languages are explicitly designed for knowledge representation and can be easily used to reason about the content they convey. The family of SL languages is constituted by three languages, which exhibit an increasing expressiveness: SL0, SL1, and SL2. SL languages define the syntax and the semantics of FIPA-ACL messages.

### 3.2.3. Ontology Agent

FIPA recommends the use of an *Ontology Agent* in order to provide support to other agents, which investigate on the ontologies available in the multi-agent system. The activity of the *Ontology Agent* and the services it supplies constitute the *Ontology Service.*
The Ontology Agent is in charge of:

- discovering public ontologies and making them available to the agents;
- maintaining the ontologies used and stored in the system;
- translating expressions between different expressions and languages;
- answering to queries about the relations among terms or among ontologies;
- detecting shared ontologies with the aim of simplifying the communication.

FIPA provides an ontology to communicate with the Ontology Agent whose service is registered with the Directory Facilitator. The Ontology Agent can perform only a subset of the previous tasks, and FIPA indicates different scenarios in which the tasks which are asked to the Ontology Agent are increasingly difficult. These scenarios constitute a roadmap for the progressive enhancement of the Ontology Agent.

### 3.3. State of the Art

At present time, there is restricted number of agent programming frameworks that comprise ontology support – that is design, development, and management – in the set of features they deliver to the software engineer. In the following I will provide a brief overview of these frameworks with a particular attention on the facilities they offer for ontology management.

- **Cougar.** Cougar [Helsinger04] is a DARPA research project which applies multi-agent system technology for developing logistic applications within the military context. Cougar features a Logical Domain Model supporting the development of application data ontologies (defined domains) along with the necessary automation for translating concepts between these domains and the Message System. A domain is composed by classes and objects which define together a language. It is possible to translate an external ontology into a domain. Objects are represented through assets which have peculiar features defined in their prototypes.
• LS/TS is a programming suite for developing autonomous agents. It provides a model for representing and managing knowledge base based on OWL. This model is used to exchange messages among software agents.

• SPYSE (Smart Python Simulation Environment). SPYSE [Spyse06] is a multi-agent programming framework that uses different standards – FIPA, Web Services, and RDF/OWL – which constitute the basis of the ontologies used by the agents for message exchange.

• Magenta [Magenta06]. This is a multi-agent system technology which targets the enterprise business process. The framework provides facilities for ontology design. Ontologies are independent from the application code base and this allows developers to control the behavior of the multi-agent system just by modifying the ontologies. In this case ontology support is used to elaborate the business process rather than for mere message exchange.

• JADE [Caire02]. JADE is a FIPA-compliant agent middleware. It provides a reference implementation of the FIPA specification for what concerns the recommendation on ontology support. Even in this case ontologies are mostly used to formalize the content of message exchanged among agents.

It can be observed that all the programming frameworks, except Magenta, use ontology to provide a formal model for the message exchanged among agents. This is interesting outcome because it abstracts the interaction which drives the application and provides a uniform view regardless the specific kind of application is running or being developed.

Among the different solution we have decided to take as a reference model for the implementation JADE with the aim of make possible a future interoperation through shared ontologies.
4. Ontology Support in AgentService

AgentService provides a complete support to ontology design, implementation, and management. These three functionalities are collectively referred as the ontology service. The main objective in introducing ontology support is related to exploiting at the interaction level the advantages coming from a tool for knowledge representation, that are a formal specification of the interactions and the knowledge body of a particular domain.

The ontology service is based on the following components:

- Classes representing the object model defining all the elements required to represent ontologies (classes, concepts, instances, attributes, constraints, validation, etc).
- Tools that can be used to automatically generate the specific classes for a given ontology by starting from its visual or textual representation.
- An Ontology Agent (OA) which maintains the knowledge about all the ontologies registered in the hosting agent platform and about the agent which are able to communicate by using the concepts defined into a given ontology.
- FIPA SL0 [FIPA08-02] ACL message support.

As we can notice, from the previous list the framework does not directly provide any facility to visually design software ontologies. We decided to rely on a very well known and established tool that is Protégé [Gennari02] a software projects maintained by the KSI lab the Stanford University. Protégé, when equipped with Jambalaya [Storey01], provides all the required features to quickly design a given ontology. In the following I will illustrate the object model designed to support ontology definitions in AgentService with a particular attention on the implementation issues and the related advantages. Then I will discuss the support of SL0, the role of the Ontology Agent, and I will describe the ontology development process.
4.1. The Object Model

The design and the implementation of the object model defining the ontology API reflects the specifications outlined in the corresponding FIPA standards [FIPA86-01] and has been inspired by the type system designed in JADE [Caire02] for providing ontology support. The object model defined within AgentService defines a meta-ontology which contains all the concepts and the elements which are required to compose user defined ontologies. The structure and the elements defined perfectly mimic the meta-model previously discussed and illustrated in Figure 20. Then the object model will be characterized by the following elements: Term, Primitive, Aggregate, Variable, Query, Concept, Action, and Predicate.

4.1.1. Classes and Interfaces vs. Schemas

The object model provides two different views and implementations of the elements: the meta-ontology: classes and interfaces, or schemas. This double view conveys the same information but serves to different purposes. It is also present when dealing with the specializations of the previous elements into a specific ontology.

The implementation based on classes and interfaces leads to the definition of new types: for each element a specific type, whose attributes are defined according to the properties of the element, is created. The relationships of inheritance and composition/meronymy are enforced through the corresponding relationships that are defined in the object oriented model. This means that two elements which are connected through an inheritance relationship are translated into two types whose one is the base type of the other. This implementation is useful for the user who can manage the elements of the ontology by exploiting the advantages of strong typing. Figure 21 shows the equivalent type model obtained from using this implementation to represent the meta-ontology.
Conversely, the implementation based on schemas leads to the creation of new objects that are always of the same type for different specializations of the same element of the meta-model. The type used to represent the element or its specialization is one of the types showed in Figure 22 in a perfect correspondence with model depicted in Figure 20. This kind of implementation implies the following: given an ontology OntX which describes a domain DomX all the concepts defined in the ontology OntX will be represented through different instances of ConceptSchema class, all the actions will be modeled through different instances of ActionSchema, and so on. Schemas are late-bound or indirect representations of the elements. Whereas the previously described implementation do not create instances but types, this implementation maintains fixed the type model and increments the instances.
The *Schema* view and implementation is more powerful than the use of strongly typed classes because it allows the developer to programmatically manage types that he does not have to know a priori. For this reason, this implementation is the one internally used by the *AgentService* framework to maintain a global knowledge of the ontologies. It should be now clear why the previous approach is more efficient: whereas the first implementation allows the user to deals directly with the types, the second one actually allows him to deal with description of these types. As a final remark on this issue it can be observed that while the first approach implies the hard coding of the ontology within the algorithm defining the agent, the second approach allows a dynamic use and inspection of ontologies not known a priori. Given this second case, it is possible from an *OntologyDescriptor* extract all the information related to a given ontology and even to obtain the specific types for each element of the ontology.
4.1.2. The Ontology Class and its Services

The Ontology class represents the starting point from the analysis of ontologies. It represents the base class for all the classes which provide the information of a specific ontology. The Ontology class represents the information repository of a given ontology, and it provides the following:

- general information about the ontology (name and other qualifying properties);
- vocabulary of meaningful terms related to the ontology;
- the list of base ontologies if any;
- the mappings between schemas and actual classes and interfaces.

This class is defined as a singleton and this ensures that the information on a given ontology maintained by only one instance in a given application domain, that is at least for a single agent.

Moreover, the Ontology class delivers advanced services which allow a dynamic management and use of ontologies:

- Validation: at any given time it is possible to ask the Ontology class for validating an instance. The ontology class will check that for the given instance all the constraints are satisfied and the state is not corrupt.

- Queries: the Ontology class can be queried in order to obtain additional information about its properties.

For all these reasons, the Ontology class along with the OntologyDescriptor class, provide a professional support for the management of ontologies, which allow implementing a real dynamic and evolvable behavior for software agents.

4.2. Language Support

Along with the object model for representing and managing ontologies it has been introduced a language support for describing the content of a message. AgentService
provides a full implementation of the SL0 specification. Such language is the minimum subset of functionalities required for an agent communication language.

The adoption of ontologies within a conversation implies the use of an agent communication language for interaction and message exchange, that is, in the case of AgentService SL0. SL0 is used to describe the content of messages which are enriched in meaning with an ontology. However, the use of SL0 imposes some limitations to the content of a message that will be opportunely equipped with an ACLMessageBody. The ACLMessageBody is a container of message data and conveys only ontological content, which can be only represented by the following elements:

- **actions**;
- **predicates**;
- **queries or IREs**;

Predicates and queries can be composed in more complex expressions by exploiting the following operators of the SL0 specification:

- **Action**: it contains an action and the AID of the agent which executed it;
- **Done**: it contains a reference to a specific action and the result of that action represented by a boolean value, which provides information about its successful execution.
- **Result**: it contains the reference to an action and the result produced by that action.
- **TrueProposition**: identifies an atomic formula having value true.
- **FalseProposition**: identifies an atomic formula having value false.
- **Equals (‘=’)**: it does not belong to SL0 but it is part of SL2. This operator has been implemented for its obviously evident utility. It can be considered like a predicate which asserts that the element on the left is equal to the element of the right.
Finally, according to the common implementations of the SL0 specification, the content of an ACL message should be represented in a textual form by observing the syntax of the language. Since AgentService has always allowed the transport of instances through binary serialization, even in this case it is possible to transport objects as message content, but the only objects that can be used are the object oriented representations of the allowed elements (predicates, queries, and actions).

### 4.3. Ontology Agent

In order to be compliant with the specification provided by FIPA we have introduced the Ontology Agent which is responsible of maintaining the catalog of all the ontologies registered with the system and of providing useful information to software agents. The entire list of tasks that should be performed by the ontology agent is the following:

- ontology discovery and publishing;
- ontology maintenance;
- ontology mapping and translation;
- shared ontology discovery.

The ontology agent provided with the framework implements only the two features of the previous list which are also the most important. We think that the ontology mapping service is a very difficult task to implement and requires some sort of inductive knowledge in order to detect similarities among different knowledge representations.

In order to be available to the community of agents the Ontology Agent registers its service to the DF. Since we have implemented a reduced set of task of the ontology agent we decided to embed these functionalities directly into the Directory Facilitator, by adding a specific behavior performing these tasks. The community of agents asks to the DF which agent provides the ontology service and the DF returns its own agent identifier. Hence, the implementation of this feature is completely transparent to the community of agents which only expect to obtain the address of the Ontology Agent in order to query it.
4.4. **Ontology Development Process**

Figure 23 describes the entire process that by starting from the *Protégé* [Gennari02] editor generates the assembly containing the type definitions for the ontology which are required by the *AgentService* framework.

![Ontology Code Generation](image)

Projects designed in *Protégé* can be exported into the XML format and a tool provided with *AgentService* (*Protégé2AgentService*) automatically generates the corresponding object model, writes the source code and compiles it into an assembly which can be easily deployed into the agent platform or used by the protocol designer. After the compilation process takes place MAS engineers are provided with the entire object model describing the ontology they designed with *Protégé*. The assembly can be included into a generic
software project and used as a library, directly deployed into the agent platform, or used as a basis for creating interaction protocols.

5. Interaction Protocols

An agent using a given ontology is sure that its interlocutor, sharing the same ontology, will be able to understand the entities described by their schemas, and then the messages containing these entities. Software ontology helps agents to communicate since it establishes a domain and a dictionary containing a set of well defined terms. The use of software ontology clarifies the content of messages and sometimes establishes a sequence among them, but does not give a strong support for modeling and structuring complex interactions among agents. Since software agents fund their activities on the interaction with peers, the possibility of having facilities for this task is a great value for AgentService. Here is where Pericles comes to play. Pericles is a component of AgentService and provides it with a tool allowing the definition and the implementation of an interaction protocol from a visual model. An interaction protocol is a dialog among agents, where each agent plays a role. Each role is defined by a lifeline constituted by AUML elements. The roles implementing the protocol are then compiled into an assembly ready to use within agent behavior objects.

In this section we explain the Pericles contribute to the interaction protocol design and source code generation. Firstly, we list the interaction protocol models, examining their features and the state of the art of interaction protocol design tools.

5.1. Interaction Protocol Representations

Interaction for agents is a fundamental activity and structured interaction becomes an important issue when designing a multi-agent system where a considerable number of entities communicate among them. Different abstractions have been proposed to provide a formal model for agents’ interactions each of them by stressing a particular aspect as concurrency management, transition among states, the evolution of the knowledge of
peers. The abstractions which obtained more acceptance within the researchers’ community are the following:

- **Finite State Machines (FSM):** an intuitive method to represent protocols, for which there exists many design software, algorithms of analysis and model checking algorithms;

- **Petri nets:** Petri nets are a very popular abstraction to represent protocols. They introduce synchronization and meeting points, to manage concurrent and distributed processes. As with FSM, there exist various tools to design, analyze and validate Petri nets;

- **Temporal Logic:** it introduces a formalism to represent past, present and future events. An example is TRIO [Ghezzi90]: a first order temporal logic;

- **Formal Rules:** similarly to the temporal logic, they introduce a formalism to represent strictly the rules that manage a protocol;

- **Hybrid Methods:** to compensate the limits of a single method, it is possible to integrate it with another one. For example FSM with formal rules, or FSM with explicit textual information about sender and recipient of a message;

- **AUML:** it is an extension of UML, aimed to agent interactions, representing a specialization of UML sequence diagrams.

The aim of *Pericles* is to provide a set of tools which simplifies the development of multi-agent systems; hence, we decided to rely on a solution easy to use, intuitive but, at the same time, powerful enough to express agent interactions. FSM are intuitive, but don’t manage messages; Petri nets offer a powerful method to represent state transactions and concurrent events, but they don’t have the explicit notion of an agent executing an action; moreover, the same protocol, represented by Petri nets, results more complex than, for example, the corresponding AUML diagram. Temporal logic and formal rules provide an efficient way to express strict protocol specifications, but they don’t offer a graphical notation to easily model interactions among agents. AUML is especially
designed to represent agent interactions and is easy to comprehend. On the other hand, if the protocol is wide, AUML, as the other methods, may become inappropriate; moreover, AUML is not a standard: there exist a lot of versions of this UML extension, all similar but not identical. Nevertheless, AUML represents a good method to model agent conversations; moreover, since it is an extension of UML, agent developers can capitalize their experience with UML.

5.2. Related Works

The design of agents and agent interactions is involved in several MAS frameworks. Considering only those projects that generate source code, we introduce briefly:

- **JADE** extensions [Dinkloh03] that generate code from a FSM integrated with additional information strings. They are written in Java and it is an Eclipse plug-in. The tool generates code for JADE using the native FSM management of the well-known platform;

- **PAUL** [Ehrler04]: it is another plug-in for Eclipse for the Opal Multi-Agent platform [Purvis02]. Although Opal supports Petri nets, PAUL represents its interaction protocols in a simplified AUML (only the alternative operator is implemented). Specifications for message contents are hosted in textual constraints applied to diagrams;

- **VIPER** [Collier04]: it is a design tool for Agent Factory [Collier03]. Protocols are represented by an old version of AUML using AND, OR, and XOR operators in place of combined fragments suggested by UML 2.0. VIPER generates a code skeleton; the user must fill the blank parts of the skeleton, writing or modifying rules, in accordance with belief-commitment architecture of Agent Factory;

- **SmartAgent** [Griss02]: it extends the JADE management of FSM introducing hierarchical state machines, represented by UML state chart diagrams in Visio 2002. SmartAgent also improves the event dispatcher of JADE, adding several new
events (associated to timeouts, incoming messages and message exceptions), which are linked to the states of the FSM;

- **Mulan** [Cabac03]: Mulan is a framework reference architecture for MAS, implemented in Petri nets and execute in Renew [Kummer01]. An AUML diagram is firstly designed, then, this interaction protocol is converted in a high level Petri net (named reference net), from which it is generated the skeleton code. Presently loops are not well represented and the AUML version is the old one, with AND, OR and XOR operators.

- **Agentis** [Kinny98]: Agentis allows the design of agent interaction protocols represented by the Z language [Spivey92], which offers formalism based on the set theory and first order logic. Z specifications define the state space of a system and the possible operations that transform the current state to another.

It can be observed that there is no tool that integrates ontologies and interaction protocols and often a representation model of interaction protocol is not sufficient to clearly represent the conversation, considering the source code generation. Nearly all are plug-ins or use existing design tools, exploiting widely used graphic environments. FSM and AUML diagrams appear to be the most used representation model, probably due to their intuitive and practical approach to the modeling of state transactions (FSM) and interactions among agents (AUML).

### 5.3. Design

AUML [Odell00] is widely adopted within the community of agent researchers and the absence of a well established standard gave birth to a lot of dialects such as UAML [FIPA25-01], EAUML [Wei01], and AUMLe [Koning01]. FIPA is now considering an AUML standard proposal [Huget02], but in the meantime every one has his version of AUML. The AUML version of Pericles is inspired to the unofficial FIPA AUML and considers specification of UML 2.0 standard proposal [OMG04].
In order to represent AUML diagrams we decided to rely on a professional visual modeling environment: the Microsoft Visio editor. Visio provides great facilities to model and represent almost all kinds of diagrams and its environment can be easily extended with plug-ins and new stencils.

Stencils are a set of shapes with a specific behavior and custom properties that can be used in a diagram. In order to provide support for protocol design we created a new stencil containing the shapes representing all the components required to define a protocol. By following the UML 2.0 specification the Pericles AUML is composed by:

- **the protocol frame**: it is a box containing all the interaction protocols;
- **the lifeline**: it represents an agent role that evolves in the time, sending and receiving messages, through split paths and loops;
- **messages**: they are connections between two lifelines;
- **combined fragments**: they represent various types of paths as loops, optional paths (namely an if then without else), alternative paths (as a switch case), and break paths (interruption of the protocol);
- **nodes**: they are activation points of the protocol used to link messages to lifelines and to attach a combined fragment.

![Figure 24. Role Representation and Lifelines](image-url)
The visual model proposed by the Pericles AUML editor looks like the other AUML editors in which we integrated additional features to allow automatic code generation. The definition of a new protocol implies the creation of two roles which are defined by the lifelines. The lifelines are made up by the shapes available in the stencil. Each shape exhibits additional properties which can be easily customized through windows forms integrated in the Visio editor. Developers can choose to rely on software ontology to describe the communication or on simple messages. If an ontology is selected, messages are intended as ontological, therefore their content consists in a $SL0$ operator hosting an object associated to an ontology schema. If the user doesn’t suggest a software ontology, he must create message attributes, entering their name and their type.

Concretely, in order to instance a new interaction protocol, the user must enter the protocol name and, if a software ontology already exists, he must select the assembly containing the ontological classes. The new protocol is represented by a rectangle that will contain the interacting roles.

To create a role and paste it into the protocol box (Figure 24 shows, for example, the buyer and the seller roles), it must enter the role name and the maximum and minimum cardinalities allowing the representation of peer to peer protocols (where each role can be played by only one agent) and client-server protocols (where only one agent plays the server role and more agents play the client role). It is also possible to enter statements for additional error management (as wrong message receiving) or exception management (as a runtime error). The role evolves through its lifeline, along which the nodes are connected. Each node is linked to message connectives or combined fragments.

Combined fragments are difficult to manage in connection with the source code generation (our goal is to avoid any user’s code writing) because guard conditions in AUML are generic labels which contain simple text. Pericles provides operators to manage an optional path (covered only if the guard condition is true) and alternative paths (as a switch case, only a path is covered if its guard is true, otherwise, if it exists, a default case is covered, namely an else block). In order to manage the guard condition, we distinguish
two cases: a guard involving a previously sent or received message or a guard involving a decision of an agent playing a role. In the first case, the interaction protocol developer must select a previous message, point a property of the message, select an operator (as $=,$ $\geq,$ $\leq,$ is, etc.), and finally suggest the right member of the condition. If the condition is too complicated, the designer offers the possibility to delay the decision at runtime, to the agent which will implement the role. The decision is automatically communicated to the interlocutor role. Graphically, an alternative (or optional) path is represented through a box; to obtain a series of alternative paths, it only must add a box under the previous one and enter the guard condition.

A box differently painted represents a loop block, which repeats the execution of internal lifelines until the guard condition is true. It is a role that decides the guard condition and it is the protocol designer who decides which role will set the guard. The

Figure 25. Fragment of AUML Diagram with Nested Boxes
loop end is automatically communicated to the passive role that exits from the loop. To finish in advance the protocol execution, a protocol developer must use the break block, which, as any other combined fragment, can be nested (see, in Figure 25, an example of nested combined fragments).

5.4. Code Generation

Automatic code generation is one of the most important features of the Pericles AUML editor; the suite is able to translate the visual model of the protocol into form that can be directly used while programming software agents. This feature is easily accessed within the AUML editor through an additional toolbar provided by the implemented plug-in.

The translation process firstly creates the object model representing the roles within the AgentService framework along with all the required classes, generates the source code of these classes and then compiles them into an assembly. Users can also decide to generate only the source code without generating the assembly. This two step process is very useful in order to allow users to further customize and enhance the protocol: this is particularly true in case of complex and big protocols, where not all the aspects can be easily managed through the visual editor. On the other hand the automatic generation of the assembly is the most commonly used option. In both the two cases the generated code will contain (see Figure 26):

- a class representing each role along with the definition of the callback interface used by this class. This class representing the role derives from the StateMachine class and overrides its only virtual method Execute. This method will contain the statements representing the role;

- a set of classes representing the custom message content which has been defined through the visual editor;
a class describing the protocol which holds the protocol name, the number, and the type of participants etc.

Pericles distinguishes between peer to peer protocols and client-server protocols, depending on the maximum cardinality of each role. A peer to peer role has an execute method that contains sequential statements, while a server role relies on a more complex execution structure since it has to maintain a different state according to the client it is serving. In case of client-server protocols the server role executes a state for each connected client, using a round-robin scheduling algorithm. We associate the concept of state to the node of our AUML diagrams: in a state, an agent sends or receives messages and checks the guard condition of a combined fragment. The server role remains at the client disposal until a timeout expires (the timeout can be set during the design phase). Pericles also manages the sending of system messages among roles, in order to
communicate to the interlocutor the end of a loop or the user-defined condition guard. The role which decides the end of a loop or a guard condition value, sends automatically a system message (hidden to agents which play roles) that informs the other role about its decision. A system message is also sent in case of exception occurred during the execution or in case of wrong message reception, informing the interlocutor that the conversation will be terminated in advance.

As we saw, the guard condition value, the loop end, and the message contents can be decided by agents through the entry points: methods that return objects sent by the role as a message contents or evaluated as guard condition. Figure 26 shows the two entry points of a role, related to the interface methods. Pericles generates a class for each role, representing an interface containing the entry points of the role. The statements of these methods must be written by agent developers, because they represent the business code. Moreover, an agent can obtain the received message contents, using the exit points, which are property members of the class representing the role. The generated assembly contains also classes that implement the message contents, both ontological and native (non-ontological). If the messages are ontological, there are classes for every SL0 operator. Each class has properties to host the effective message contents (instances of the classes associated to ontological schemas). For native messages, every class has the properties set by the user during the design phase. At last, an agent that wants to play a role through a behavior, must implement the entry point interface, defining each method with its business code; then it must instantiate the class representing the role and run the `Execute` method.

6. Observations

At present time two agent oriented projects have been using the features presented in this chapter: the first – namely MATT – concerns the implementation of a groupware meeting scheduler based on agent technology [Grosso05], while the second project – namely DFTalk – is still on-going and is related to the development of an interaction
protocol (which relies on ontologies) with the Directory Facilitator of the AgentService platform.

The two projects are rather different: whereas the groupware meeting scheduler has been implemented by developers who were familiar with the underlying architecture of the AgentService platform, DFTalk has been proposed as a didactical project and is being developed by students who neither know the C# language. Moreover, the size and the degree of complexity of the two projects are different as well: MATT is by far more complex and required more time than DFTalk. This allowed us having a good, even though really small, test bed, for analyzing the proficiency of software developers and comparing it with previous experiences.

In the case of DFTalk the students who did not know the C# language, generated ontologies and interaction protocols automatically and without having to code some fragments of the application. We noticed that the main obstacle encountered in the DFTalk project by the student has been becoming familiar with conceptualization of the entities of the discourse domain and the abstraction of a conversation. This obstacle has been soon removed as soon as the students got a little experience, and the design of ontologies and protocols has consequently becomes easy and intuitive. The conjunct use of ontologies and FIPA-ACL, more precisely the SL0 subset, for defining the content of messages exchanged between agents and the DF, greatly simplified the development of the protocol because allowed the students to concentrate on the real complexity of the problem that was the structure of the conversation and not the algorithms required to code or decode messages.

In the case of MATT, the software developers who knew either the C# language or the API of AgentService, found anyways useful and extremely powerful the high level approach provided through the design tools for ontologies and interaction protocols. The use of these tools reduced implementation time. The possibility to modify the code generated by the protocol designer has been sometimes exploited in order to implement specific features that went beyond the flexibility of the design tool.
As a final remark, we noticed that, usually, the structure of the execution flow related to the interaction among agents is not that complex and can be automatically generated by protocol designer. One of the limitations of the actual design is its difficulty in handling large interaction diagrams: this weakness is not due to the inability of automatically generating code, but to the fact it is difficult to visually manage large diagrams. We are working on introducing the possibility of having different levels of the zoom of the interaction diagram and investigating on the opportunity of splitting the interaction in sub-protocols which would be linked by behavior and knowledge objects, through entry and exit points.

The introduction of ontology support along with the interaction protocols has been driven by the desire of simplifying the work of developers and of laying the foundations for a clear representation of knowledge bases, messages and interactions among agents. The tight integration between the ontology support and the interaction protocol designer allows a simple design and a quick implementation of well structured, formally described, and meaningful agent interactions. The feedback of users rewards our efforts and gives us some suggestion to enhance the actual implementation of these components. For example, the strict use the SL0 language has demonstrated to be useful but limiting: SL0 allows managing actions and action responses, assertions, and simple queries, permitting agents to express meaningful messages but not complex ones. For this reason, we are thinking to provide a full implementation of SL1 and SL2 along with an interactive Visio addon that would allow composing complex messages during the design phase, coherently with our approach of easing developer’s work.
This thesis presented AgentService, which has been the outcome of my doctoral studies. AgentService is a software framework for agent-oriented systems development. In particular, AgentService addresses that class of software systems which are evolvable. By the term evolvable I mean software systems that change during their life-cycle either by integrating new components or by modifying the interconnections between the existing ones. These systems evolve their structure in order to adapt to the changing environment in which they operate. Such environment is constituted by the users, the stakeholders, and by the software and technology infrastructure into which these systems are integrated. The concept of change and evolution in software has now become of primary importance and evolvable systems play an important role in the software industry. These systems are commonly the result of aggregations of preexisting systems which get interconnected and rely on each other in order to perform their activities. In other words, they are dependable. In the first chapter of this dissertation I gave a brief overview of the current trends in software engineering and showed how the community of software engineers is more and more focusing on this property of software systems. In particular, the concept of software ecosystem and the growing interest of biologically inspired software models and architectures are a proof of this trend. Some notable examples of
these models are autonomic systems and, in general, self-* systems. These approaches have strong connections with the theory of software agents, which have been considered as an interesting approach to look at in many other contexts. Hence, the study of Agent Oriented Software Engineering (AOSE) as a mean for modeling and implementing dynamic and complex systems still deserves attention. This is also demonstrated by the various editions of the AAMAS conference, which is considered one of the most important conferences in software design and development and it is the top conference for researchers in software agents. This dissertation has then provided an overview of the theory of software agents and a detailed analysis of the AgentService framework that is the outcome of my studies and supports the engineering of complex systems through the paradigm of software agents.

As pointed out in the second chapter of this dissertation, software agents are powerful abstractions since they are high-level system components characterized by autonomy, reactivity, pro-activity, and social ability. These properties define the agents’ flexible behavior, which makes possible – through the abstraction of multi-agent systems – the creation of complex and evolvable software systems. Multi-agent systems are the primary means of AOSE for developing complex and large-scale software systems. AgentService directly supports the implementation of multi-agent systems by providing the software artifacts that constitute the basic bricks for multi-agent system development: a software model for implementing agents and an extendable run-time environment for them, which is the agent platform. These are common components of any toolkit or framework for agent-oriented software development. The innovation introduced with the AgentService framework resides in its architectural design, which is based on flexibility and extensibility. These features directly characterize either the agent model or the agent platform. On the other hand, AgentService is also a product of Software Engineering and puts a strong emphasis on the concepts of software variability and design for change [Gamma94] that are fundamental for extensible software systems.
Whereas most agent programming frameworks and toolkits limit the development of software agents to a single agent architecture, AgentService provides a flexible agent model able to support different agent architectures. Moreover, it supplies an efficient run-time support that can be customized and extended even in its core functionalities. In Chapter 4 I presented in detail the scheduling system of the framework which is a clear example of this design decision. The scheduling engine is a core component of the framework directly influencing the performance of the entire multi-agent system and AgentService provides a great degree of flexibility in executing software agents. Such flexibility has been obtained by making the engine work with simple abstractions, which are those that constitute the agent model: behavior objects and knowledge objects. Despite all the complex models developers can design with knowledge and behavior objects, the scheduling engine will always be able to execute software agents since it manages agent execution by dealing with the fundamental bricks of its model. This is the same politic used in providing the persistence service and in designing the mobility infrastructure. By using simple and general abstractions the advantage is twofold: developers deal with intuitive models and can build – if they need – complex ones; the framework can easily provide a strong run-time support for all the services required during the life cycle of agents. Such services can be more easily customized and extended in a complete transparent manner and this is a fundamental property when designing frameworks which have to provide support for evolvable systems.

The same philosophy lies behind the implementation of the agent platform, which is the second pillar of multi-agent system development. From an implementation and architectural point of view the agent platform is probably the most important component of a multi-agent system, since it constitutes the middleware required for the execution of software agents. The agent platform provides the run-time environment for software agents and actually behaves as a virtual machine for them. Hence, the possibility of extending this software layer in order to adapt the system to new constraints is fundamental for providing a good support for evolvable systems. Extensibility in this case means the ability to integrate new services, providing different
implementations of existing ones, or making available new technologies. The AgentService allows such degree of extensibility and obtains this flexibility by relying on an extremely modular and customizable architecture through which it is possible to compose the multi-agent system by selecting the best technological solutions for each service offered by the agent platform. The modular design is also responsible of allowing an easy interoperation with other technologies, thus making AgentService-based MASs open systems. Open architectures are able to change more easily and with less effort; this is a fundamental property when supporting evolvable systems in which change is a common condition rather than a rare event. Modularity and openness have been the profitably used for designing and implementing the support for distributed programming in AgentService. As discussed in Chapter 5, the framework has been enhanced with the possibility of managing distributed systems – modeled as a network AgentService platforms but also other applications – in a completely transparent manner and with almost no impact on the way in which software agents are designed and implemented. This is an advantage originating from the overall architecture of the framework and also a proof that its design is flexible enough for incremental development and painless extensibility.

Around this flexible core, the AgentService framework offers additional components which simplify the development of multi-agent systems. As noticed in Chapter 2 software agents base their activities on interaction with peers. Interactions are the means through which a task is accomplished within a multi-agent system. Hence, a profitable communication and the ability of being understood are fundamental requirements for interaction. Multi-agent systems are open and dynamic by nature; these features make the design and the implementation of interactions a challenging task. In order to provide a flexible solution for modeling interactions the theory of agents suggests the introduction of an additional level of abstraction which establishes the way in which software agents should manage knowledge and information: ontologies. Ontologies constitute an organic and detailed representation of the essential elements within a given knowledge domain. Rather than being a fixed vocabulary of terms,
ontologies can be extended, can inherit concepts from other ontologies, and can be composed in order to shape complex knowledge domain. By devising ontology-based interactions, communities of heterogeneous software agents can interoperate easily and without a pre-established set of idioms (or at least with a very limited one): one of the dreams of the community of agent researchers is developing software agents with the ability of integrating new knowledge, understanding its essence and the implications deriving from it, and influence their course of action according to their new mental state. Even though this is still really far from happening, AgentService provides a strong support for software ontologies, thus allowing the design and the implementation of software agents that expose a more flexible behavior. This contributes to narrow the gap between the current implementation of multi-agent systems and the scenario envisioned by researchers. As discussed in Chapter 6, AgentService provides a complete support for ontologies with facilities for design and implementation and run-time services which make ontologies available for software agents. Developers can visually define ontologies, create interaction protocols, and the framework automatically translates these models into ready to use code-libraries. In a complete assonance with the other components of AgentService these automation tools provide a high degree of customization: developers that want to modify the outcome of their visual model can always switch to the underlying object model and directly modify the automatically generated code. Whereas it is possible to statically design software agents that embed the capability of managing a specific ontology, the frameworks also provides the possibility for agents of dynamically inspecting ontologies and using them in their course of action. This is the basis for making the dream of learning happen, even though still a lot of work resides in the ability of developers in programming agents that are capable of attributing a meaning – and then making a profitable use of them – to ontologies dynamically learnt at run-time.

The support provided for ontologies is probably the most important component among the additional services offered by the framework, which are mainly focused in providing a simple but powerful interface for programming multi-agent systems with AgentService. As demonstrated thorough this dissertation and summed up in these conclusions, the
main goal of the framework is to provide by means of simple and intuitive abstractions a strong support for evolvable multi-agent system development. These abstractions rely on a flexible and modular design which is strength of AgentService and its key factor for adapting to the changing requirements that characterize the development of complex systems.

As expressed in the subject of this dissertation, AgentService proposes an agent-based approach for modeling and implementing evolvable software systems. At the time of writing, the theory of agents is almost twenty years old and cannot be considered a new paradigm for software development. What makes the work exposed in this dissertation still of interest is the fact that a software engineering approach to the development of agent-based systems is relatively new. Moreover, the concept of software agent – an autonomous and flexible entity that can be easily integrated in non agent-based systems – is now obtaining a growing attention in other fields of software engineering and computer science. The idea of software agent and its essential characterization is becoming more familiar than before and multi-agent systems are now investigated as a possible solution in many application scenarios. For all these reasons, the creation of frameworks and toolkits for experimenting and developing agent-based systems plays an important role in making available this technology, which it is always been confined in the community of agent researchers. AgentService it is not “just another agent programming framework”, but by developing it with my colleagues, I wanted to provide a software infrastructure through which easily implement software agents and exploit all their essential features along with the other technologies and programming models that characterize contemporary software development. It is my personal opinion that the challenges which reside in architecting the software systems for tomorrow cannot be faced by using a single approach or a unique technology. The variety of scenarios in which the practical applications of software operate and the wide range of problems they have to face can only be managed by flexible software systems which are able to integrate different technologies and paradigms. The agent oriented technology is just one of the technologies available today and AgentService provides developers with a good starting
point for making a profitable use of this technology by means of simple abstractions for implementing agents and multi-agent systems and integrating these systems with other applications.

During my doctoral studies I mostly concentrated my efforts in the development of the core features of the AgentService framework and the additional features presented in this work, by applying the best solutions and the outcomes of research in the theory of agents. At present time, AgentService does not provide any facility covering the analysis phase of software life-cycle. In particular does not natively support any agent-oriented methodology among those presented in Chapter 3, neither defines its own methodology. In planning the activity of my doctoral studies I preferred to concentrate my work on defining and implementing the basic bricks of AgentService. This decision led to having a working and usable implementation of the agent programming framework which allowed us implementing the first agent-based systems with AgentService [Grosso05] and then testing and evaluating the property of the framework. Future directions of the research concerned with the framework will necessarily see, along with a continuous enhancement of its features, the support for the most known agent oriented methodologies and, if required, the development of a specific methodology for developing multi-agent systems with AgentService.
Bibliography


Chapter 7 – Conclusions


Chapter 7 – Conclusions


Chapter 7 – Conclusions


[Gamma94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley, Reading, Massachusetts, 1994


Chapter 7 – CONCLUSIONS


Chapter 7 – Conclusions


Chapter 7 – Conclusions


Chapter 7 – Conclusions


Chapter 7 – Conclusions


Chapter 7 – Conclusions


Chapter 7 – Conclusions


List of Figures
List of Figures

Figure 1. Highlights in academic attention in software engineering [Shaw96] .................... 16
Figure 2. Structure of an autonomic element ........................................................................ 33
Figure 3. Layered Agent Architectures.................................................................................... 54
Figure 4. Comparison among the OOSD, CBSE, and AOSE Software Engineering approaches .................................................................................................................. 73
Figure 5. FIPA Reference Model for Agent-Based Systems .................................................... 96
Figure 6. Architecture of the AgentService Framework .......................................................... 108
Figure 7. Platform Instance Internal Structure ...................................................................... 113
Figure 8. Platform Instance Life Cycle ..................................................................................... 115
Figure 9. Agent Instance at Runtime ....................................................................................... 121
Figure 10. Scheduling Engine .................................................................................................. 128
Figure 11. Setting up of the messaging infrastructure .............................................................. 145
Figure 12. Messaging Infrastructure ......................................................................................... 146
Figure 13. XML Format of the AgentMessage class ................................................................. 148
Figure 14. Agent life cycle ....................................................................................................... 164
Figure 15. Agent migration process .......................................................................................... 166
Figure 16. Mobility Infrastructure ........................................................................................... 168
Figure 17. FTP Server automaton ............................................................................................ 169
Figure 18. Interface for load balancing policies ...................................................................... 174
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Example Ontology: Marriage</td>
<td>183</td>
</tr>
<tr>
<td>20</td>
<td>Meta-Ontology for Multi-Agent System</td>
<td>191</td>
</tr>
<tr>
<td>21</td>
<td>Class &amp; Interface Implementation</td>
<td>197</td>
</tr>
<tr>
<td>22</td>
<td>Schemas Implementation</td>
<td>198</td>
</tr>
<tr>
<td>23</td>
<td>Ontology Code Generation</td>
<td>202</td>
</tr>
<tr>
<td>24</td>
<td>Role Representation and Lifelines</td>
<td>207</td>
</tr>
<tr>
<td>25</td>
<td>Fragment of AUML Diagram with Nexted Boxes</td>
<td>209</td>
</tr>
<tr>
<td>26</td>
<td>Classes Generated by the Designer</td>
<td>211</td>
</tr>
</tbody>
</table>
As normally happens for any path that reached its end, what remains, after all the celebrations are over, it is not, if any, the entitlement that eventually you get, but all those precious gifts you collected along the way, which determined the rest of the voyage. This thesis is a tribute to all those people which contributed to enrich my perception of things, to broaden my vision, and to improve my sensibility, especially for what regards software design and development. Some of them are known and have been always near me, others not and I just met them reading books or papers during my studies. Both of them deserve a remembrance in this work, which would has been definitely different, if I did not met them. Finally, among the virtual authors of this work I would like to include all those people that trusted me and let me free of doing what I thought to be better. These are my family first, my friends, my supervisors, and colleagues.

Christian Vecchiola