Architecture-Based Design of Multi-Agent Systems

Danny Weyns

Architecture-Based Design of Multi-Agent Systems

Foreword by Len Bass

Springer

Danny Weyns
Katholieke Universiteit Leuven
Dept. Computer Science
Celesijnenlaan 200a
3001 Leuven
Belgium
danny.weyns@cs.kuleuven.be

ISBN 978-3-642-01063-7 e-ISBN 978-3-642-01064-4
DOI 10.1007/978-3-642-01064-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2010920463

ACM Computing Classification (1998): I.2.11, D.2.11, C.3, J.7

Cover design: KuenkelLopka GmbH, Heidelberg

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Tessa, Eva, and Marina

Foreword

One of the most important things an architect can do is reflection. That is, examine systems, organizations, people and ask "What alternatives were considered and why was that particular decision made?" Thinking about the response gives an architect insight into the motivations and decision processes that others have used and this, in turn, should help the architect make better decisions in the future. A pre-requisite for doing this type of reflection is that the decisions and alternatives are made explicit. One venue that gives an architect an opportunity to do this type of reflection is during an architectural evaluation. Another venue is from a book such as this. This book lays out the design process used in building a collection of multi-agent systems.

In addition to providing a case study of a design process and the rationale for the design decisions, the topic of the book also is of great interest. Systems of the future will increasingly have the characteristics of the autonomous systems described here: they are simultaneously becoming more interconnected and more autonomous. Think of your smart phone that is mostly connected but can operate many functions while it is disconnected. These systems of the future will also increasingly operate without central control. Again, the telephone system and how cellular communication is managed provides a good example of this phenomenon.

Problems of connectivity raise issues of a node discovering that it is disconnected, other nodes discovering that a particular node is disconnected, how the node operates while it is disconnected, and reconnecting the node. The case study provides solutions to this problem in the context of autonomous vehicles within a limited geographic area. The essence of the solution provided—define the concept of a neighborhood for a node and treat neighboring nodes in a different fashion from other nodes—seems like it is more general than the particular application in the case study but that is still to be determined.

Communication and protocols seem to be basic to providing solutions to relatively autonomous nodes, and the structure of the middleware for such systems is of interest independently from the particular application area. The middleware needs to provide not only communication structure and neighborhood definition but also security and authentication services. The case study describes a middleware structure, and a portion of the reflection process of the architect is to ask not only about the rationale for the specific decisions made but also how well these decisions will generalize to other situations that the architect can envision. Designing

systems is one aspect of the work of an architect but as stated in the introduction "Developing multi-agent systems software is 95% software engineering and 5% multi-agent systems theory." All of the portions of the software engineering life cycle must operate efficiently in order for systems to be effectively constructed. This means that the important requirements must be identified, a design generated, the design documented and evaluated, and the system constructed from the design. Each of these topics is treated in the book.

- The important requirements are typically the quality attribute requirements. Eliciting these requirements requires a different mindset from the normal requirements process whether through a formal process, through user stories, or through some other technique. Quality attribute requirements tend to be the requirements that are taken for granted by the user until they have not been met.
- Documentation is important for helping the designer think through difficult design issues and for communicating the design to others. From the perspective of a reflecting architect, the documentation provided in this book provides some understanding of the division of functionality and design rationale.
- Evaluation of a design is an important step for verifying nothing has been missed by the architect. An evaluation is an application of the multiple eyes principle—get an outside, knowledgeable perspective to look at the design. As was pointed out, this process takes time. Partially this is because it takes time to educate the outside eyes and partially because evaluation requirements look at the design with a variety of different concerns. In the ATAM process, these concerns are expressed as scenarios but, in general, looking at a complicated design in sufficient detail to determine potential problems will take time. This time could be done as one activity when the outside eyes have to travel, as in the ATAM, or the time could be spread over a collection of shorter activities when the outside eyes are generally available.

In summary, this book is interesting both for its expressed topic—the design of multi-agent systems—and as a case study where a reader can read, and reflect on, the rationale for the approach taken in building such a system.

Pittsburgh, Pennsylvania, USA Len Bass
October 23, 2009

Acknowledgements

This book is based on 8 years of applied research conducted by a team of researchers at DistriNet Labs in collaboration with multiple industrial partners. First and foremost, I must acknowledge the contributions of Tom Holvoet and Kurt Schelfthout. Their insights and stimulating discussions have significantly contributed to the development of architecture-based design of multi-agent systems. Kurt invented the ObjectPlaces middleware described in Chapter 5. The following people must also be acknowledged for their invaluable contributions to the approach: Alexander Hellboogh, Nelis Boucké, and Elke Steegmans. The advanced coordination mechanisms described in Chapter 6 are the result of an enjoyable collaborative effort with Nelis over multiple years. I thank my former colleagues Koen Mertens and Tom De Wolf for their contributions. I also thank the Master students that have supported the development of various aspects of the approach described in this book, in particular Robin Custers, Olivier Glorieux, Bart Demarsin, Wannes Schols, Els Helsen, and Koen Deschacht. I am grateful to Jan Wielemans, Tom Lefever, Walter De Feyter, Rudy Vanhoutte, Wim Van Betsbrugge, Jan Peirsman, Raf Sempels, and Jan Vercammen for the collaboration in the EMC2 project. I wish to acknowledge the international colleagues I worked with during the past years for the stimulating collaborations. I am particularly grateful to Van Parunak and Fabien Michel for the enjoyable joint efforts. I would like to thank the researchers and architects of the Software Engineering Institute for the inspiring discussions on the design and evaluation of decentralized architectures. Several anonymous reviewers commented on earlier versions of this manuscript and suggested many improvements. Ralf Gerstner provided useful advice toward getting this book published. Thank you.

Danny Weyns

Contents

Acronyms

ADD	Attribute-Driven Design
ADL	Architecture Description Language
AGV	Automatic Guided Vehicle
APL	Agent Programming Language
ATAM®	Architecture Tradeoff Analysis Method®
AUML	Agent Unified Modeling Language
BDI	Belief, Desire, Intention
C&C	Component and Connector
cfp	call for proposals
CNET	Contract NET
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DynCNET	Dynamic Contract NET
E'nsor®	Egemin navigation system on robot
E'pia®	Egemin platform for integrated automation
ERP	Enterprise Resource Planning
FiTA	Field-based Transport Assignment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
ISO	International Standard Organization
JEE	Java Enterprise Edition
LAN	Local Area Network
LIME	Linda In a Mobile Environment
Mbps	Megabits per second
QAW	Quality Attribute Workshop
RMI	Remote Method Invocation
SOAP	Simple Object Access Protocol
TB	Transport Base
UML	Unified Modeling Language
XML	Extensible Markup Language

Chapter 1
Introduction

A well-known claim for multi-agent systems is that they are especially suited to develop software systems that are decentralized, can deal flexibly with dynamic conditions, and are open to system components that come and go. While we endorse this claim, developing real-world multi-agent systems taught us that achieving these goals is a complex engineering problem. Our experience with real-world multi-agent systems development can be captured succinctly in the following statement:

> Developing multi-agent systems software is 95% software engineering and 5% multi-agent systems theory.

In this book, we present *architecture-based design of multi-agent systems*, an architecture-centric approach for developing real-world multi-agent systems. The approach integrates multi-agent system concepts with state-of-the-art principles and methods from mainstream software architecture and middleware. The practical applicability of the approach is demonstrated for an industry-strength application in the domain of automated transportation systems.

The objective of the book is twofold. On the one hand, we provide a guide to software engineers for the architectural design of real-world multi-agent systems. On the other hand, we give a detailed description of how we have used this guide for developing a complex multi-agent system in an industrial setting.

We start by introducing two fields of software engineering that are central in architecture-based design of multi-agent systems: software architecture and middleware. Next, we explain how and why our perspective on engineering multi-agent systems differs from existing agent-oriented methodologies. Then, we introduce the automated transportation system where we use a case study to demonstrate the applicability of architecture-based design of multi-agent systems. The introduction concludes with an overview of the book.

1.1 Software Architecture and Middleware

Two fields of software engineering are central in this book: software architecture and middleware. In this section, we introduce both fields and illustrate their importance with respect to the design of real-world multi-agent systems.

D. Weyns, *Architecture-Based Design of Multi-Agent Systems*,
DOI 10.1007/978-3-642-01064-4_1, © Springer-Verlag Berlin Heidelberg 2010

1.1.1 Software Architecture

Since the mid-1990s, software architecture has been the subject of increasing interest in software engineering research and practice. Software architecture is a corner stone for ensuring that systems achieve their quality and functional goals and ultimately serve an organization's business and mission goals. Software architecture provides the required level of abstraction and generality to deal with the increasing challenges of adaptation required in distributed software applications [91]. Bass, Clements, and Kazman define software architecture as "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [21]. Software elements provide the functionality of the system, while the required quality attributes (performance, adaptability, usability, modifiability, etc.) are primarily achieved through the structures of the software architecture. Software architecture constitutes a model for how a system is structured and works. This model is transferable to other systems with similar requirements and can promote large-scale reuse of design. Besides technical value, software architecture is also considered as a key vehicle for communication among stakeholders. Software architecture provides a basis for creating mutual understanding and consensus about the software system [46].

During architectural design, architects apply proven architectural approaches to transfer the system requirements into appropriate software structures. Architectural patterns offer well-established solutions to architectural problems. An architectural pattern expresses a fundamental structural organization schema for a software system which exhibits known quality attributes. For example, layers is a well-known pattern that structures a software system into an appropriate number of layers and places them on top of each other. The services of each layer implement a strategy for combining the services of the layer below in a meaningful way. Layers enhance maintainability, extensibility, and reusability of the system. However, applying the layer pattern can be expensive on system resources affecting performance.

A multi-agent system is in essence a system that is structured as a set of autonomous agents that are able to flexibly adapt their behavior to changing operating conditions. Durfee and Lesser define a multi-agent system as "a loosely coupled network of problem solvers (agents) that interact to solve problems that are beyond the individual capabilities or knowledge of each problem solver " [52]. Characteristics of multi-agent systems are as follows: (1) each agent has incomplete information or capabilities for solving the problem and, thus, has a limited viewpoint; (2) there is no system global control; (3) data is distributed; and (4) computation is asynchronous. Multi-agent systems are characterized by specific intra-agent and inter-agent structures. At the level of individual agents, many different architectures have been developed, ranging from simple reactive agents to complex reasoning agents. At the system level, the multi-agent system can be structured as an organization of selfish agents that play different roles in the organization pursuing their own interests. Other multi-agent systems consist of cooperative agents that aim to achieve a common goal. Agents can interact in different ways: via a high-level communication

language and specific interaction protocols or via manipulating marks in a shared coordination medium. Since specific multi-agent system structures imbue the software systems with certain qualities, while making certain tradeoffs, a primary focus of multi-agent system engineering is on the software architecture of the system. Multi-agent systems are known for quality attributes such as adaptability, openness, robustness, and scalability, making multi-agent systems particularly interesting to deal with the demanding challenges of complex distributed software applications.

In Chap. 3, we explain how design expertise in multi-agent systems can be captured as architectural patterns. We present a number of architectural patterns for a family of multi-agent systems. These patterns embody a set of architectural best practices derived from the experiences with developing various multi-agent system applications. In Chap. 4, we explain how architectural patterns play a key role in transferring stakeholder requirements into appropriate software structures. The primary structures of a multi-agent system are critical for the achievement of the system's quality attributes. Chapter 7 elaborates on the evaluation of multi-agent system architectures. Architectural evaluation allows determining the tradeoffs and risks of architectural decisions with respect to satisfying important quality attribute requirements.

1.1.2 Middleware

Middleware is the software layer that lies between the operating system and the application components. Middleware provides high-level abstractions to support the coordination of distributed software components. With networks becoming increasingly pervasive, middleware appears as a major building block for the development of complex distributed software systems [77]. Since multi-agent systems are particularly useful for problem domains characterized by highly dynamic operating conditions and inherent distribution of resources, it is clear that any multi-agent system application should deal with the distribution concern.

Domain-specific middleware for multi-agent systems typically consider agent communication as the prior means for agent coordination. A communication infrastructure usually provides a management system that enables agents to register and locate one another and a message transport system. Coordination infrastructures offer an alternative for direct message exchange allowing agents to interact indirectly via a shared medium. Two different examples of coordination infrastructures are an electronic institution that acts as a governor for interaction and digital pheromones that agents use to mark dynamic paths to areas of interest similar as social ants. Since interactions necessary for coordination often take place in a concurrent and distributed environment that is unreliable, middleware is a crucial aspect in software development of multi-agent systems. Concerns such as security, persistency, and transactional behavior are typically supported by domain-independent middleware services. Since these concerns often crosscut the system, vertical integration with domain-specific middleware is an important aspect of the design of any real-world multi-agent system.

In Chap. 5, we elaborate on the role of middleware for supporting the development of distributed multi-agent systems. We take a closer look at the multiple layers of middleware in distributed software systems in general, and we zoom in on middleware for multi-agent systems. We explain in detail the middleware used in the case study, called ObjectPlaces. ObjectPlaces proposes two new programming abstractions, *view* and *coordination role*, to support the development of mobile multi-agent system applications.

The first abstraction, a view, is an automatically up-to-date collection of data objects that are copies or representations of data objects available on a set of nodes in the network. The middleware automates gathering the data objects from a set of nodes and maintains the view in the face of dynamically changing availability of the data objects. The second abstraction, a coordination role, encapsulates the behavior of a component of the application engaging in a protocol. The middleware automates the setup and maintenance of an interaction session between a number of participating components in the mobile network, in the face of a frequently changing number of participants.

The ObjectPlaces middleware encapsulates the tedious management tasks associated with distribution in mobile multi-agent systems. The middleware has significantly reduced the complexity of tackling distributed coordination problems in the case study. In Chaps. 5 and 6 we show how the middleware has simplified the development of the application components in the automated transportation system for collision avoidance and task assignment, respectively.

1.2 Agent-Oriented Methodologies

Since the early 1990s the idea that multi-agent systems are a radically new way of engineering software has dominated research and practice in agent-oriented software engineering. Wooldridge et al. [177] state that

> There is a fundamental mismatch between the concepts used by object-oriented developers and other mainstream software engineering paradigms, and the agent-oriented view. [...] Existing software development techniques are unsuitable to realize the potential of agents as a software engineering paradigm.

Zambonelli and Omicini [182] state that

> Agent-based computing can be considered as a new general-purpose paradigm for software development, which tends to radically influence the way a software system is conceived and developed.

This vision has led to the development of numerous multi-agent system methodologies. Some of the methodologies focus on particular phases of the software development process, e.g., Gaia [177, 181]. Others cover the full software development life cycle, e.g., Tropos [64]. Some of the proposed methodologies adopt mechanisms and practices from mainstream software engineering. Prometheus [119] is inspired by object-oriented mechanisms. MaSE [174] uses practices of the Unified Process. Adelfe [25] uses constructs of the Unified Modeling Language. However, nearly all

methodologies take an independent position, barely embedded in mainstream software engineering practice. Studying literature reveals that very limited results have been obtained in the application of these methodologies to real-world problems. A notable exception is the DACS methodology (Designing Agent-based Control Systems), introduced by Bussmann et al. [38], that was applied in the design of a multi-agent system for manufacturing control at DaimlerChrysler.

Our perspective on engineering multi-agent systems starts from the viewpoint that multi-agent system engineering fits well within mainstream software engineering. This vision is based on the observation that multi-agent systems are essentially a way to structure a software system, making software architecture of paramount importance in engineering multi-agent systems.

By putting software architecture and middleware at the heart of the engineering process, architecture-based design of multi-agent systems places multi-agent systems in a larger context of software engineering. This perspective provides at the same time insights and opportunities for both multi-agent system and mainstream software engineers and researchers. Considering multi-agent systems from a software architecture perspective does not delude existing results of agent-oriented software engineering. On the contrary, agent-oriented software engineering has developed a wide body of valuable concepts and techniques for engineering agent behavior, adaptation, advanced interactions, organizations, learning, etc. This domain-specific know-how is required to support architects and developers of multi-agent systems. The architecture-based approach for developing multi-agent systems presented in this book integrates such domain-specific concepts and techniques with mainstream software engineering methods and practices.

1.3 Case Study

Throughout this book, we use an automated transportation system as a case study. The description of the architectural design and development of this application demonstrate the practical applicability of architecture-based design of multi-agent systems. The case study was developed between 2004 and 2007 by a team of engineers and developers of Egemin, a leading company that provides full life cycle support for automated transportation systems [53], and researchers of DistriNet Labs. This section introduces the application and motivates the use of a multi-agent system architecture.

An automated transportation system consists of a number of automatic guided vehicles (AGVs) that need to work together to transport loads in an industrial environment. Transports are generated by client systems, typically an enterprise resource planning (ERP) system. The main functionalities that an AGV transportation system has to fulfill are assigning incoming transport tasks to an appropriate AGV, routing the AGVs through the warehouse efficiently while avoiding collisions and deadlocks, and recharging the AGVs' batteries.

An AGV transportation system has to deal with dynamic and changing operating conditions. The stream of transports that enter the system is typically irregular and

unpredictable, AGVs can leave and re-enter the system for maintenance, production machines may have variable waiting times, etc. All kinds of disturbances can occur, supply of goods can be delayed, certain areas in the warehouse may temporarily be closed for maintenance services, loads can block paths, AGVs can fail, etc. Despite these challenging operating conditions, the system is expected to operate efficiently and robustly.

Traditionally, the AGV systems deployed by Egemin are directly controlled by a central server. The server plans the schedule for the system as a whole, dispatches commands to the AGVs, and continually polls their status. This results in reliable and predicable solutions. The central point of control also enables easier diagnosis of errors. However, a shift in user requirements challenges the centralized architecture. Customers increasingly request for *self-managing systems*, i.e., systems that are able to adapt their behavior with changing circumstances autonomously. Self-management with respect to system dynamics translates to two specific quality requirements: flexibility and openness. Flexibility refers to the system's ability to deal with dynamic operating conditions. Openness refers to the system's ability to deal with AGVs leaving and entering the system.

To deal with these new quality requirements, a radically new architecture was conceived based on situated multi-agent systems. Applying a situated multi-agent system opens perspectives to improve flexibility and openness of the system: the AGVs can adapt themselves to the current situation in their vicinity, order assignment is dynamic, the system can deal autonomously with AGVs leaving and re-entering the system, etc. However, introducing a decentralized architecture may have a number of implications, in particular decentralized decision making may have an impact on the overall efficiency of the system such as throughput and bandwidth usage. These are critical issues that have to be considered during the design and development of the multi-agent system.

The software system was implemented on a prototype setup with real AGVs and tested in larger, industrially used simulations. The design and implementation of the AGV control system needed 8+ man-years of effort. The delivered code base for the control software consists of about 100K lines of C# code. This system interfaces with a lower level AGV steering system that for its real-time properties is written in C.

1.4 Overview of the Book

In Chap. 2, we give a general overview of architecture-based design of multi-agent systems. We situate architectural design in a software development life cycle, and we zoom in on the different steps in the approach. These steps include requirements elicitation, architectural design, architecture documentation, and architecture evaluation. For each step, we give some background and we introduce the different techniques and methods that are used. The chapter concludes with a brief explanation of how software architecture serves as a basis for downstream design and implementation of the system.

Chapter 3 shows how architectural patterns provide the means to capture expertise with multi-agent system design. We introduce a set of architectural patterns for situated multi-agent systems, the family of multi-agent systems we have applied in the case study. The set of architectural patterns provides an asset base that architects can use in the design of a family of multi-agent systems.

In Chap. 4, we elaborate on architectural design of multi-agent systems and the documentation of software architecture. In architecture-based design of multi-agent systems, we use attribute-driven design (ADD) [173] as a design method. ADD is concerned with the high-level decomposition of a software system which is critical for satisfying the system's quality requirements. ADD yields a set of architectural views. To document the views we use the Views and Beyond [45] method. We apply the methods to the design and documentation of the case study. The case study makes clear how the various patterns for situated multi-agent systems were applied during architectural design.

In Chap. 5, we zoom in on middleware for distributed multi-agent systems. Middleware supports application developers with the design and implementation of coordination solutions in a distributed setting. We explain in detail a concrete middleware that was developed for the case study and we illustrate how this middleware supported a complex coordination problem in a mobile setting.

One particularly complex coordination problem in distributed multi-agent systems is task assignment. Chapter 6 is dedicated to this problem. We zoom in on two approaches for adaptive task assignment that are characteristic for two classical families of coordination mechanisms for task assignment: a protocol-based approach and a field-based approach. We explain the design and validation of both approaches in the case study, and we make a tradeoff analysis.

Chapter 7 elaborates on the evaluation of a multi-agent system architecture. In architecture-based design of multi-agent systems, we use the Architecture Tradeoff Analysis Method (ATAM) [46][1] for the evaluation of software architecture. ATAM is a structured method to examine whether a software architecture is suitable for the system for which it was designed. ATAM uncovers architectural tradeoffs and risks in the design. We explain in detail the ATAM evaluation for the case study and reflect on the experiences with using ATAM for the evaluation of a multi-agent system architecture.

In Chap. 8, we discuss related approaches that explicitly connect software architecture with multi-agent systems. We also examine related work on middleware for mobile systems. Additionally, we give a brief overview of related work on the control of AGV transportation systems.

In Chap. 9, we reflect on architecture-based design of multi-agent systems and its application to the case study, and we report lessons learned from applying the approach in practice. We conclude with an outline of challenges for future research on engineering multi-agent system derived from our experiences.

---

[1] Architecture Tradeoff Analysis Method® and ATAM® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.