Agent-Oriented Software Engineering

Kuldar Taveter
The University of Melbourne, Australia

Leon Sterling
The University of Melbourne, Australia

INTRODUCTION

Over the past decade, the target environment for software development has complexified dramatically. Software systems must now operate robustly in a dynamic, global, networked environment comprised of distributed diverse technologies, where frequent change is inevitable. There is increasing demand for flexibility and ease of use. Multi-agent systems (Wooldridge, 2002) are a potential successor to object-oriented systems, better able to address the new demands on software. In multi-agent systems, heterogeneous autonomous entities (i.e., *agents*) interact to achieve system goals. In addition to being a technological building block, an agent, also known as an *actor*, is an important modeling abstraction that can be used at different stages of software engineering. The authors while teaching agent-related subjects and interacting with industry have observed that the agent serves as a powerful anthropomorphic notion readily understood by novices. It is easy to explain to even a non-technical person that one or more software agents are going to perform a set of tasks on your behalf.

We define *software engineering* as a discipline applied by teams to produce high-quality, large-scale, cost-effective software that satisfies the users' needs and can be maintained over time. Methods and processes are emerging to place software development on a parallel with other engineering endeavors. Software engineering courses give increasing focus to teaching students how to analyze software designs, emphasizing imbuing software with quality attributes such as performance, correctness, scalability, and security.

Agent-oriented software engineering (AOSE) (Ciancarini & Wooldridge, 2001) has become an active research area. Agent-oriented methodologies, such as Tropos (Bresciani, Perini, Giorgini, Giunchiglia, & Mylopoulos, 2004), ROADMAP (Juan & Sterling, 2003), and RAP/AOR (Taveter & Wagner, 2005), use the notion of agent throughout the software lifecycle from analyzing the problem domain to maintaining the functional software system. An agent-oriented approach can be useful even when the resulting system neither consists of nor includes software agents. Some other proposed AOSE methodologies are Gaia (Wooldridge, Jennings, & Kinny, 2000), MESSAGE (Garijo, Gomez-Sanz, & Massonet, 2005), TAO (Silva & Lucena, 2004), and Prometheus (Padgham & Winikoff, 2004). Although none of the AOSE methodologies are yet widely accepted, AOSE is a promising area. The recent book by Henderson-Sellers & Giorgini (2005) contains a good overview of currently available agent-oriented methodologies.

AOSE approaches loosely fall into one of two categories. One approach adds agent extensions to an existing object-oriented notation. The prototypical example is Agent UML (Odell, Van Dyke, & Bauer, 2001). The alternate approach builds a custom software methodology around agent concepts such as roles. Gaia (Wooldridge et al., 2000) was the pioneering example.

In this article, we address the new paradigm of AOSE for developing both agent-based and traditional software systems.

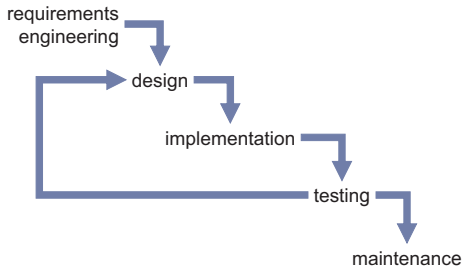
BACKGROUND

Software engineering deals with sociotechnical systems. Sommerville (2004) defines a *sociotechnical system* as one that includes hardware and software, has defined operational processes, and offers an interface, implemented in software, to human users. Software engineering addresses developing software components of sociotechnical systems. Software engineering is therefore critical for the successful development of complex, computer-based, sociotechnical systems because a software engineer should have a broad awareness of how that software interacts with other hardware and software systems and its intended use, not only the software itself (Sommerville, 2004).

A conventional *software engineering process* represented in Figure 1 contains the stages of requirements engineering, design, implementation, testing, and maintenance.

Requirements engineering—understanding, and specifying the user's needs—is followed by *software design* consisting of both high-level architectural design of the system and detailed design of the software components. Implementation follows, increasingly in the form of code generation using different Computer-aided software engineering (CASE) tools. The code must then be tested to uncover and correct as many errors as possible before delivery to the customer. Pressman (2001) provides an overview of many potentially

Figure 1. A software engineering process



useful testing strategies and methods. Finally, there is ongoing maintenance.

Object-oriented software engineering (OOSE), which numerous AOSE methodologies build on, adopts these conventional steps. Object-oriented approaches characterize the problem as a set of objects with specific attributes and behaviors (Pressman, 2001). Objects are categorized into classes and subclasses. Object-oriented analysis identifies classes and objects relevant to the problem domain; design provides architecture, interface, and component-level detail; and implementation (using an object-oriented language) transforms design into code.

To facilitate OOSE, the Unified Modelling Language (UML) (Object Management Group (OMG) 2003a, 2003b) for software analysis and design has become widely used in the industry over the past decade. A widely used OOSE process is the *Rational Unified Process* (RUP) (Kruchten, 1999) derived from UML and the associated *Unified Software Development Process* proposed by Jacobson, Booch, and Rumbaugh (1999). The core software engineering disciplines of RUP are domain modeling, requirements engineering, design, implementation, testing, and deployment.

In parallel with RUP, agile methodologies, such as *Extreme Programming* (Beck, 1999), have emerged. They emphasize lightweight processes such as test-case-based development and rapid prototyping. They de-emphasize detailed modeling on which they blame the heavy weight and inflexibility of traditional methodologies. In contrast, the *Model-Driven Architecture* (MDA, http://www.omg.org/mda) approach of the OMG identifies modeling as the key to state-of-the-art software engineering. In the MDA, computation-independent domain models are transformed into platform-independent design models that are then turned into platform-specific models and implementations.

NEW SOFTWARE ENGINEERING PARADIGM

Dynamic adaptive systems in open environments create new challenges for software engineering. For such systems, current software engineering techniques cannot guarantee quality attributes such as correctness to hold after deployment. Correctness is traditionally assured by testing the system before release, against documented requirements. Such assurance is lost if system behavior changes due to continuous adaptation or environment change. For example, a sociotechnical business system may change its structure (architecture) to mirror changes in the human organization.

Similarly, system performance, reliability, security, usability, and maintainability can be compromised due to adaptation or environmental changes. Without explicit representation of system requirements and constant validation at run time, there is no guarantee that the system functions correctly.

To address the challenges described, a sociotechnical system should be analyzed and designed in terms of agents (actors), roles, and goals at the stage of requirements engineering, as well as at design, implementation, testing, and maintenance stages of the software lifecycle. From the start of a software engineering process, a distinction should be introduced between active and passive entities, that is, between agents and (nonagentive) objects of the real world. We define an agent as an autonomous entity situated in an environment capable of both perceiving the environment and acting on it. The agent metaphor subsumes *artificial* (software and robotic), *natural* (human and animal) as well as *social/institutional* agents (groups and organizations). According to Wagner (2003), the agent metaphor lies beyond UML where actors are only considered as users of the system's services in *use cases*, but otherwise remain external to the sociotechnical system model.

We define a *role* as a coherent set of functional responsibilities specifying what the agent playing the role is expected to do in the organization within some specialized context or domain of endeavor: with respect to both other agents and the organization itself. Roles may be subject to constraints.

Roles go hand in hand with goals. For example, in a business domain, a human or institutional agent acting in the role of "customer" has a goal of accomplishing something. To achieve its goal, the agent uses some service provided by another agent. An agent's autonomy means that the service provider performs the service requested if it is able to do so but the service provider also has an option to refuse the service request. Even though the agent requesting the service may not explicitly communicate its goals to the service provider agent, the latter always "internalizes" the whole or part of the customer's goal in attempting to provide the service. For example, given a customer wanting to rent a car, the goal of a car rental company is to provide the customer