

# Agent-Oriented Software Engineering

Nicholas R. Jennings

Dept. Electronic Engineering, Queen Mary & Westfield College,  
University of London, London E1 4NS, UK.  
n.r.jennings@qmw.ac.uk

## 1 Introduction

Increasingly many computer systems are being viewed in terms of autonomous agents. Agents are being espoused as a new theoretical model of computation that more closely reflects current computing reality than Turing Machines. Agents are being advocated as the next generation model for engineering complex distributed systems. Agents are also being used as an overarching framework for bringing together the component AI sub-disciplines that are necessary to design and build intelligent entities. Despite this intense interest, however, a number of fundamental questions about the nature and the use of agents remain unanswered. In particular:

- what is the essence of agent-based computing?
- what makes agents an appealing and powerful conceptual model?
- what are the drawbacks of adopting an agent-oriented approach?
- what are the wider implications for AI of agent-based computing?

These questions can be tackled from many different perspectives; ranging from the philosophical to the pragmatic. This paper proceeds from the standpoint of using agent-based software to solve complex, real-world problems. Building high quality software for complex real-world applications is difficult. Indeed, it has been argued that such developments are one of the most complex construction tasks humans undertake (both in terms of the number and the flexibility of the constituent components and in the complex way in which they are interconnected). Moreover, this statement is true no matter what models and techniques are applied: it is a consequence of the “essential complexity of software” [2]. Such complexity manifests itself in the fact that software has a large number of parts that have many interactions [8]. Given this state of affairs, the role of software engineering is to provide models and techniques that make it easier to handle this complexity. To this end, a wide range of software engineering paradigms have been devised (e.g. object-orientation [1] [7], component-ware [9], design patterns [4] and software architectures [3]). Each successive development either claims to make the engineering process easier or to extend the complexity of applications that can feasibly be built. Although evidence is emerging to support these claims, researchers continue to strive for more efficient and powerful techniques, especially as solutions for ever more demanding applications are sought.

In this article, it is argued that although current methods are a step in the right direction, when it comes to developing complex, distributed systems they fall short in one of three main ways: (i) the basic building blocks are too fine grained; (ii) the interactions are too rigidly defined; or (iii) they possess insufficient mechanisms for dealing with organisational structure. Furthermore, it will be argued that: *agent-oriented approaches can significantly enhance our ability to model, design and build complex (distributed) software systems.*

## 2 The Essence of Agent-Based Computing

The first step in arguing for an agent-oriented approach to software engineering is to precisely identify and define the key concepts of agent-oriented computing. Here the key definitional problem relates to the term “*agent*”. At present, there is much debate, and little consensus, about exactly what constitutes agenthood. However, an increasing number of researchers find the following characterisation useful [10]:

*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*

There are a number of points about this definition that require further explanation. Agents are: (i) clearly identifiable problem solving entities with well-defined boundaries and interfaces; (ii) situated (embedded) in a particular environment—they receive inputs related to the state of their environment through sensors and they act on the environment through effectors; (iii) designed to fulfill a specific purpose—they have particular objectives (goals) to achieve; (iv) autonomous—they have control both over their internal state and over their own behaviour; (v) capable of exhibiting flexible problem solving behaviour in pursuit of their design objectives—they need to be both reactive (able to respond in a timely fashion to changes that occur in their environment) and proactive (able to opportunistically adopt new goals) [11].

When adopting an agent-oriented view of the world, it soon becomes apparent that most problems require or involve multiple agents: to represent the decentralised nature of the problem, the multiple loci of control, the multiple perspectives, or the competing interests. Moreover, the agents will need to interact with one another: either to achieve their individual objectives or to manage the dependencies that ensue from being situated in a common environment. These interactions can vary from simple information interchanges, to requests for particular actions to be performed and on to cooperation, coordination and negotiation in order to arrange inter-dependent activities. Whatever the nature of the social process, however, there are two points that qualitatively differentiate agent interactions from those that occur in other software engineering paradigms. Firstly, agent-oriented interactions occur through a high-level (declarative) agent communication language. Consequently, interactions are conducted at the knowledge-level [6]: in terms of which goals should be followed, at what time, and by whom (cf. method invocation or function calls that operate at a purely syntactic level). Secondly, as agents are flexible problem solvers, operating in an environment over which they have only partial control and observability, interactions need to be handled in a similarly flexible manner. Thus, agents need the computational apparatus to make context-dependent decisions about the nature and scope of their interactions and to initiate (and respond to) interactions that were not foreseen at design time.

In most cases, agents act to achieve objectives either on behalf of individuals/companies or as part of some wider problem solving initiative. Thus, when agents interact there is typically some underpinning organisational context. This context defines the nature of the relationship between the agents. For example, they may be peers working together in a team or one may be the boss of the others. In any case, this context influences an agent's behaviour. Thus it is important to explicitly represent the relationship. In many cases, relationships are subject to ongoing change: social interaction means existing relationships evolve and new relations are created. The temporal extent of relationships can also vary significantly: from just long enough to deliver a particular service once, to a permanent bond. To cope with this variety and dynamic, agent researchers have: devised protocols that enable organisational groupings to be formed and disbanded, specified mechanisms to ensure groupings act together in a coherent fashion, and developed structures to characterise the macro behaviour of collectives [5] [11].

### 3 Agent-Oriented Software Engineering

The most compelling argument that could be made for adopting an agent-oriented approach to software development would be to have a range of quantitative data that showed, on a standard set of software metrics, the superiority of the agent-based approach over a range of other techniques. However such data simply does not exist. Hence arguments must be qualitative in nature.

The structure of the argument that will be used here is as follows. On one hand, there are a number of well-known techniques for tackling complexity in software. Also the nature of complex software systems is (reasonably) well understood. On the other hand, the key characteristics of the agent-based paradigm have been elucidated. Thus an argument can be made by examining the degree of match between these two perspectives.

Before this argument can be made, however, the techniques for tackling complexity in software need to be introduced. Booch identifies three such tools [1]:

- *Decomposition*: The most basic technique for tackling any large problem is to divide it into smaller, more manageable chunks each of which can then be dealt with in relative isolation.
- *Abstraction*: The process of defining a simplified model of the system that emphasises some of the details or properties, while suppressing others.
- *Organisation*<sup>1</sup>: The process of identifying and managing interrelationships between various problem solving components.

Next, the characteristics of complex systems need to be enumerated [8]:

- Complexity frequently takes the form of a hierarchy. That is, a system that is composed of inter-related sub-systems, each of which is in turn hierarchic in structure, until the lowest level of elementary sub-system is reached. The precise nature of these organisational relationships varies between sub-systems, however some generic forms (such as client-server, peer, team, etc.) can be identified. These relationships are not static: they often vary over time.
- The choice of which components in the system are primitive is relatively arbitrary and is defined by the observer's aims and objectives.
- Hierarchic systems evolve more quickly than non-hierarchic ones of comparable size. In other words, complex systems will evolve from simple systems more rapidly if there are *stable intermediate forms*, than if there are not.
- 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

---

<sup>1</sup> Booch actually uses the term "hierarchy" for this final point [1]. However, the more neutral term "organisation" is used here.

rise to the view that complex systems are *nearly decomposable*: 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.

With these two characterisations in place, the form of the argument can be expressed: (i) show agent-oriented decompositions are an effective way of partitioning the problem space of a complex system; (ii) show that the key abstractions of the agent-oriented mindset are a natural means of modelling complex systems; and (iii) show the agent-oriented philosophy for dealing with organisational relationships is appropriate for complex systems.

### 3.1 Merits of Agent-Oriented Decomposition

Complex systems consist of a number of related sub-systems organised in a hierarchical fashion. At any given level, sub-systems work together to achieve the functionality of their parent system. Moreover, within a sub-system, the constituent components work together to deliver the overall functionality. Thus, the same basic model of interacting components, working together to achieve particular objectives occurs throughout the system.

Given this fact, it is entirely natural to modularise the components in terms of the objectives they achieve<sup>2</sup>. In other words, each component can be thought of as achieving one or more objectives. A second important observation is that complex systems have multiple loci of control: “real systems have no top” [7] pg 47. Applying this philosophy to objective-achieving decompositions means that the individual components should localise and encapsulate their own control. Thus, entities should have their own thread of control (i.e. they should be active) and they should have control over their own choices and actions (i.e. they should be autonomous).

For the active and autonomous components to fulfil both their individual and collective objectives, they need to interact with one another (recall complex systems are only nearly decomposable). However the system’s inherent complexity means it is impossible to know *a priori* about all potential links: interactions will occur at unpredictable times, for unpredictable reasons, between unpredictable components. For this reason, it is futile to try and predict or analyse all the possibilities at design-time. It is more realistic to endow the components with the ability to make decisions about the nature and scope of their interactions at run-time. From this, it follows that components need the ability to initiate (and respond to) interactions in a flexible manner.

The policy of deferring to run-time decisions about component interactions facilitates the engineering of complex systems in two ways. Firstly, problems associated with the coupling of components are significantly reduced (by dealing with them in a flexible and declarative manner). Components are specifically designed to deal with unanticipated requests and they can spontaneously generate requests for assistance if they find themselves in difficulty. Moreover because these interactions are enacted through a high-level agent communication language, coupling becomes a knowledge-level issue. This, in turn, removes syntactic concerns from the types of errors caused by unexpected interactions. Secondly, the problem of managing control relationships between the software components (a task that bedevils traditional functional decompositions) is significantly reduced. All agents are continuously active and any coordination or synchronisation that is required is handled bottom-up through inter-agent interaction. Thus, the ordering of the system’s top-level goals is no longer something that has to be rigidly prescribed at design time. Rather, it becomes something that is handled in a context-sensitive manner at run-time.

From this discussion, it is apparent that a natural way to modularise a complex system is in terms of multiple, interacting, autonomous components that have particular objectives to achieve. In short, agent-oriented decompositions aid the process of developing complex systems.

### 3.2 Appropriateness of Agent-Oriented Abstractions

A significant part of the design process is finding the right models for viewing the problem. In general, there will be multiple candidates and the difficult task is picking the most appropriate one. When designing software, the most powerful abstractions are those that minimise the semantic gap between the units of analysis that are intuitively used to conceptualise the problem and the constructs present in the solution paradigm. In the case of complex systems, the problem to be characterised consists of sub-systems, sub-system components, interactions and organisational relationships. Taking each in turn:

- Sub-systems naturally correspond to agent organisations. They involve a number of constituent components that act and interact according to their role within the larger enterprise.
- The appropriateness of viewing sub-system components as agents has been made above.
- The interplay between the sub-systems and between their constituent components is most naturally viewed in terms of high-level social interactions: “at any given level of abstraction, we find meaningful collections of entities that collaborate

<sup>2</sup> Indeed the view that decompositions based upon functions/actions/processes are more intuitive and easier to produce than those based upon data/objects is even acknowledged within the object-oriented community [7] pg 44.

to achieve some higher level view” [1] pg 34. This view accords precisely with the knowledge-level treatment of interaction afforded by the agent-oriented approach. Agent systems are invariably described in terms of “cooperating to achieve common objectives”, “coordinating their actions” or “negotiating to resolve conflicts”.

- Complex systems involve changing webs of relationships between their various components. They also require collections of components to be treated as a single conceptual unit when viewed from a different level of abstraction. Here again the agent-oriented mindset provides suitable abstractions. A rich set of structures are typically available for explicitly representing and managing organisational relationships. Interaction protocols exist for forming new groupings and disbanding unwanted ones. Finally, structures are available for modelling collectives. The latter point is especially useful in relation to representing sub-systems since they are nothing more than a team of components working to achieve a collective goal.

### 3.3 Need for Flexible Management of Changing Organisational Structures

Organisational constructs are first-class entities in agent systems. Thus explicit representations are made of organisational relationships and structures. Moreover, agent-based systems have the concomitant computational mechanisms for flexibly forming, maintaining and disbanding organisations. This representational power enables agent-oriented systems to exploit two facets of the nature of complex systems. Firstly, the notion of a primitive component can be varied according to the needs of the observer. Thus at one level, entire sub-systems can be viewed as a singleton, alternatively teams or collections of agents can be viewed as primitive components, and so on until the system eventually bottoms out. Secondly, such structures provide a variety of stable intermediate forms, that, as already indicated, are essential for rapid development of complex systems. Their availability means that individual agents or organisational groupings can be developed in relative isolation and then added into the system in an incremental manner. This, in turn, ensures there is a smooth growth in functionality.

## 4 Conclusions and Future Work

This paper has sought to justify the claim that agent-based computing has the potential to provide a powerful suite of metaphors, concepts and techniques for conceptualising, designing and implementing complex (distributed) systems. However if this potential is to be fulfilled and agent-based systems are to reach the mainstream of software engineering, then the following limitations in the current state of the art need to be overcome: a systematic methodology that enables developers to clearly analyse and design their applications as multi-agent systems needs to be devised; there needs to be an increase in the number and sophistication of industrial-strength tools for building multi-agent systems; and more flexible and scalable techniques need to be devised for enabling heterogeneous agents to inter-operate in open environments;

## References

- [1] G. Booch (1994) “Object-oriented analysis and design with applications” Addison Wesley.
- [2] F. P. Brooks (1995) “The mythical man-month” Addison Wesley.
- [3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stahl (1998) “A System of Patterns” Wiley.
- [4] E. Gamma, R. Helm, R. Johnson and J. Vlissides (1995) “Design Patterns” Addison Wesley.
- [5] N. R. Jennings and M. Wooldridge (eds.) (1998) “Agent technology: foundations, applications and markets” Springer Verlag.
- [6] A. Newell, (1982) “The Knowledge Level” *Artificial Intelligence* **18** 87-127.
- [7] B. Meyer (1988) “Object-oriented software construction” Prentice Hall.
- [8] H. A. Simon (1996) “The sciences of the artificial” MIT Press.
- [9] C. Szyperski (1998) “Component Software” Addison Wesley.
- [10] M. Wooldridge (1997) “Agent-based software engineering” *IEE Proc Software Engineering* **144** 26-37.
- [11] M. Wooldridge and N. R. Jennings (1995) “Intelligent agents: theory and practice” *The Knowledge Engineering Review* **10** (2) 115-152.
- [12] M. J. Wooldridge and N. R. Jennings (1998) “Pitfalls of Agent-Oriented Development” *Proc 2nd Int. Conf. on Autonomous Agents (Agents-98)*, Minneapolis, USA, 385-391.