Pitfalls of Agent-Oriented Development

Michael Wooldridge and **Nicholas R. Jennings**

Department of Electronic Engineering
Queen Mary & Westfield College
University of London, London E1 4NS
United Kingdom

{M.J.Wooldridge, N.R.Jennings}@qmw.ac.uk

Abstract

While the theoretical and experimental foundations of agent-based systems are becoming increasingly well understood, comparatively little effort has been devoted to understanding the pragmatics of (multi-)agent systems development — the everyday reality of carrying out an agent-based development project. As a result, agent system developers are needlessly repeating the same mistakes, with the result that, at best, resources are wasted — at worst, projects fail. This paper identifies the main pitfalls that await the agent system developer, and where possible, makes tentative recommendations for how these pitfalls can be avoided or rectified.

1 Introduction

It is now more than two decades since Frederick Brooks wrote *The Mythical Man-Month* — arguably the best-known and most influential work on software engineering and software project management yet published. In a series of memorable essays, Brooks highlighted some of the most common mistakes made in the software development process. Despite the immense amount of effort devoted to understanding and improving it, the software development process today is no easier *in essence* than it was in 1975, when *The Mythical Man-Month* was first published. The most significant improvements in software engineering have come about through the introduction of powerful abstractions with which to manage the inherent complexity of software — object-oriented programming is the most obvious example. Elsewhere, we, (along with many others), have argued that the notion of an *agent* as a self-contained problem solving system capable of autonomous, reactive, pro-active, social behaviour is such an abstraction tool [23, 24], and that agent–based computing is a promising approach to developing a range of complex, typically distributed, computer systems.

Agent-based solutions have already been developed for many different application domains, and field-tested agent systems are steadily increasing in number. In addition, a range of theoretical and experimental results attest to the fact that the scientific foundations of agent-based systems are becoming increasingly well understood. But despite these significant advances in the *science* of agent systems, comparatively little effort has been devoted to understanding how to *engineer* them. In short, our aim in this paper is to begin to rectify this omission. We identify what we perceive to be the main *pitfalls* that await the agent system development project. If agent technology is to achieve its potential, then these *pragmatic* aspects of agent system development must be studied and understood — just as they have been for object-oriented programming. There is a very real danger that if no attempt is made to do this, then agent technology will fail to live up to the claims currently being made of it. The result will be a backlash similar to that experienced against expert systems, logic programming, and all the other good ideas that, it was promised, would fundamentally change computing. Our goal is not to suggest that agent technology is in any way a bad thing, or that it is more prone to problems than other software technologies, but rather to recognise that agent systems have their own specific problems and pitfalls. By recognising these pitfalls, we cannot guarantee success, but we can at least avoid some of the most obvious sources of failure.

Note that this is not a *scientific* paper, in the sense that it does not present any theorems or easily falsifiable experimental results. However, the paper *does* make claims — specifically, claims about bad practice in agent system development. While the justification for these claims seems anecdotal, it is based on a decade of experience in developing multi-agent systems, ranging from industrial control systems [14] to Internet-based information retrieval and management systems [8].

The structure of the paper is loosely based on [22]. We identify seven categories of problem areas: political, management, conceptual, analysis and design, micro (agent) level, macro (society) level, and implementation issues. For each of these categories, we list the key pitfalls that may be found. For each pitfall, we discuss the nature of the pitfall, how it may be avoided, and, where possible, what steps may be taken to recover from it. We stress that we have focussed on pitfalls which seem specific (or at least very common) to agent-based development projects; we ignore issues that are common to software development in general [22].

Before proceeding, it is worth commenting on what *sorts* of agent system we are discussing in this paper. Our interest (and experience) is primarily in the area of *multi-agent systems*, i.e., in systems composed of multiple interacting agents, where each agent is a coarse-grained computational system in its own right. We use the more neutral term *agent-based* system to refer to one in which the key abstraction used, either in conceptualisation, design, or implementation, is that of an agent. A paradigm example of the type of project for which this paper seems appropriate is ARCHON, a multi-agent system in the field of industrial power systems management [14]. Other pertinent examples include the agent-based development projects undertaken at the Australian AI Institute (AAII) [12], and the University of Michigan Digital Library (UMDL) [6].

2 Political Pitfalls

2.1 You oversell agents

There are a number of good reasons for supposing that agent technology will enhance the ability of software engineers to construct complex, distributed applications: other considerations aside, agents are a powerful and natural metaphor for conceptualising, designing and implementing many systems. But agents are not a magical problem solving paradigm: tasks that are beyond the scope of automation using non-agent techniques will not necessarily be made tractable simply by adopting an agent-based approach. Problems that have troubled software engineers for decades are still difficult with agent systems. Indeed, there is no evidence that any system developed using agent technology could not have been built just as easily using non-agent techniques. In short, agents *may* make it easier to solve certain classes of problems, (and there are good arguments for supposing that this is the case), but they do not make the impossible possible. The reason for this is that the atomic problem solving components within agent-based systems still have to be able to perform the necessary domain tasks, and their implementation can only use the (limited) techniques that are currently available. Naturally, extra leverage can be obtained by applying multiple problem solving methods and by carefully managing the interactions between the components; but ultimately, these components still need to be written.

The other aspect of overselling is to equate agents with intelligent problem solving. Those unfamiliar with the achievements (and failures) of Artificial Intelligence (AI) often believe that agents are capable of human-like reasoning and acting. Obviously, this is not the case: such a level of competence is well beyond the state of the art in AI. Thus agents may sometimes exhibit smart problem solving behaviour, but it is still very much limited by the current state of the art in machine intelligence.

There are a number of very good reasons for not overselling agents [17], not the least of which is that artificial intelligence as a field has arguably suffered a great deal from over optimistic (some would say absurd) claims about its potential. Most recently, perhaps, the expert systems experience vividly illustrates the perils of overselling a promising technology.

See also: pitfall 4.1

2.2 Getting religious or dogmatic about agents

Although agents have been used in a wide range of applications [15], they are not a universal solution. There are many applications for which conventional software development paradigms (such as object-oriented programming) are far more appropriate. Indeed, given the relative immaturity of agent technology and the small number of deployed agent applications, there should be clear advantages to an agent based solution before such an approach is even contemplated. Given a problem for which an agent and a non-agent approach appear likely to produce equal quality solutions, the non-agent approach should generally be preferred, since it will be better understood by the software engineers involved in the system development, and as a consequence is likely to be more manageable and predictable. Unfortunately, this point is missed by many agent research and development groups, who have a somewhat blinkered attitude to other development paradigms. Such groups exhibit the single technique syndrome — if the only tool you possess is a hammer, then everything looks like a nail. To summarise, there is a danger of believing that agents are the right solution to *every* problem. As a consequence, agent solutions are often developed for quite inappropriate problems.

The other form of dogma associated with agents relates to their definition. Most agent developers have their own opinion on exactly what constitutes an agent — and no two developers appear to share exactly the same opinion (see [10] for a collection of agent definitions). Thus having made a valid case for an agent-based approach, people tend to shoe-horn their solution to fit with their definition, even when some facets of the definition are clearly gratuitous in the particular context. For example, those who feel that mobility is an essential characteristic of agenthood invariably propose mobile agent solutions even when static agents represent a more obvious and natural approach.

See also: pitfall 4.2

3 Management Pitfalls

3.1 You don't know why you want agents

This is a common problem for any new technology that has been hyped as much as agents. Managers read forecasts such as "agents will generate US$2.6 billion in revenue by the year 2000" [17], and, not surprisingly, they want to jump on the bandwagon. This phenomena is exacerbated for agents because they are such an intuitively simple concept. It is easy think of a whole army of useful agents — if only I had an agent to book my flights, read my email and automatically generate responses, and so on. However in many cases, managers that propose an agent project do not actually have a clear idea about what "having agents" will buy them. That is, they have no clear vision of how agents can be used to enhance their existing products, or how they can enable them to generate new product lines. As a consequence, agent projects are often initiated with no clear goals in mind (other that to "have" agents). With no goals, there are also no criteria for assessing the success or otherwise of the initiative, and no way of telling whether the project is going well or badly. The net result is that catastrophic project failures can occur seemingly out of the blue. The lesson is simply to really understand your reasons for attempting an agent development project, and what you expect to gain from it.

See also: pitfall 3.2

3.2 You don't know what your agents are good for

This is related to pitfall 3.1, and concerns a general lack of clarity of purpose for the use of agent technology and a lack of understanding about its degree of applicability. Having once developed some agent technology or some specific agents, there is a tendency to search for an application in which they can be used. Invariably, the process of seeking to find an application for a technology leads to mismatches and dissatisfaction, either because the full potential of what an agent could add to the application is not achieved (because the agents have the wrong functionality or emphasis), or else because only a subset of the agent's capabilities get exploited as that is all the application requires. The lesson is simple: be sure you understand how and where your new technology may be most usefully applied. Do not attempt to apply it to arbitrary problems, and resist the temptation to apply it to every problem you come across.

See also: pitfalls 3.1 and 3.3

3.3 You want to build generic solutions to one-off problems

This is a pitfall to which many software projects fall victim, but it seems to be especially prevalent in the agent community. Typi-

cally it manifests itself in the devising of an architecture or testbed that supposedly enables a whole range of potential types of agent to be built, when what is really required is a bespoke design to tackle a single application. In such situations, a custom built solution will be easier to develop and far more likely to satisfy the requirements of the application. As anybody with experience of object-oriented development knows, re-use is difficult to attain unless development is undertaken for a close knit range of problems with similar characteristics [22]. Moreover, general solutions are more difficult and more costly to develop and often need extensive tailoring to work in different applications. Yet agent developers continually speak about generic architectures that can be used and re-used for a seemingly infinite range of applications. Such claims are often unsubstantiated and based on flimsy evidence; they are reminiscent of the early days of AI when, researchers claimed to have developed general purpose problem solvers.

See also: pitfall 7.4

3.4 You confuse prototypes with systems

Having found an application for which an agent solution appears to be well suited, and having planned the solution at an appropriate level of generality, it is comparatively easy to develop a prototype system consisting of a few interacting agents doing some semi-useful task. However, this is a world away from having a solution that is sufficiently robust and reliable to be used in practice. While such claims can be levelled at any problem solving paradigm which lends itself to rapid prototyping, the gap is especially large for agent based systems because of the general characteristics of the software being developed. Agent systems, by their very nature, tend to involve: (i) concurrent and distributed problem solving; (ii) flexible and sophisticated interfaces between the problem solving components; and (iii) complex individual components whose behaviour is context dependent. Each of these characteristics in isolation makes it more difficult to bridge the gap between a prototype and a full strength software solution, but when they are all present the gap can become a chasm.

See also: pitfall 7.6

4 Conceptual Pitfalls

4.1 You believe that agents are a silver bullet

The holy grail of software engineering is a "silver bullet": a technique that will provide an order of magnitude improvement in software development [5]. Many technologies have been promoted as the silver bullet: automatic programming, expert systems, graphical programming, and formal methods are some examples. Agent technology is a newly emerged, and as yet essentially untested software paradigm: it is only a matter of time before someone claims agents are a silver bullet. This would be a dangerous fallacy. As we pointed out above, there are good arguments in favour of the view that agent technology will lead to improvements in the development of complex distributed software systems [23, 15]. But, as yet, these arguments are largely untested in practice. There is certainly no scientific evidence to support the claim that agents offer any advance in software development — the evidence to date is purely anecdotal. Even if agents *do* lead to a real improvement in software development practice, it would be naive to suppose that the advance would represent an order of magnitude improvement.

We argue that the most important developments in software engineering have presented the developer with yet more powerful *abstractions* with which to understand and manage complexity. Procedural abstraction, structured programming, abstract data types, and objects are all examples of the progressively more powerful programming abstractions developed over the past three decades, which have enabled developers to attack successively more complex programming tasks. For us, agents are just such an abstraction. They appear to provide a powerful way of conceptualising, designing, and implementing a particularly complex class of software systems. We expect that, with time, agent technology will be proven to have benefits for the software developer. But at the time of writing, it is naive and misleading to imply that such benefits are a matter of fact.

See also: pitfall 2.1

4.2 You confuse buzzwords with concepts

One of the reasons why agent technology is currently so popular is that the idea of an agent is extremely intuitive. This is on the one hand a good thing — the fact that the concept of an agent cuts across so many different disciplines is testament to its wide applicability. But unfortunately, it also encourages developers to believe that they understand concepts when in fact they do not. A good example of this is the belief-desire-intention (BDI) model of agency, as embodied in the work of Georgeff and colleagues [11]. The BDI model is interesting to the agent developer because it is underpinned by a respectable theory of (human) agency, (primarily developed by Michael Bratman [4]), it has an elegant logical semantics [20], and perhaps most importantly, it has been proved in extremely demanding applications — such as real-time fault-diagnosis on the space shuttle [12]. Unfortunately, the label "BDI" has now been applied to so many different types of agent (many of which are simply *not* BDI systems), that the phrase has lost much of its meaning. One often finds phrases like BDI repeated as if they were mantras: "our system is a BDI system", the implication being that being a BDI system is like being a computer with 64MB memory: a quantifiable property, with measurable associated benefits. This is clearly misleading.

See also: pitfall 2.2

4.3 You forget you are developing *software*

At the time of writing, the development of any agent system — however trivial — is essentially a process of experimentation. There are no tried and trusted techniques available to assist the developer. Unfortunately, because the process is experimental, it encourages the developer to forget that they are actually developing *software*. Project plans tend to be pre-occupied with investigating agent architectures, developing cooperation protocols, and improving coordination and coherence of multi-agent activity. Mundane software engineering processes — requirements analysis, specification, design, verification, and testing — become forgotten. The result of this neglect is a foregone conclusion: the project flounders, not because of agent-specific problems, but because basic software engineering good practice was ignored. The abandonment of the software process is often justified with reference to the fact that software engineering for agent systems is, as yet, a research area. While it is true that development techniques for agent systems are in their infancy, it is nevertheless also true that almost *any* principled software development technique is better than none. Thus in the absence of agent-oriented development techniques, object-oriented techniques may be used to great effect. They may not be ideal, but they are certainly better then nothing.

See also: pitfall 4.1

4.4 You forget you are developing *distributed* software

Distributed systems have long been recognised as one of the most complex classes of computer system to design and implement. A great deal of research effort has been devoted to understanding this complexity, and to developing formalisms and tools that enable a developer to manage it [2]. Despite this research effort, the problems inherent in developing distributed systems can in no way be regarded as solved. Multi-agent systems tend, by their very nature, to be distributed — the idea of a centralised multi-agent system is an oxymoron. So, in building a multi-agent system, it is vital not to ignore the lessons learned from the distributed systems community — the problems of distribution do not go away, just because a system is agent-based. A multi-agent system will if anything be *more* complex than a typical distributed system. The multi-agent system developer must therefore recognize and plan for problems such as synchronization, mutual exclusion for shared resources, deadlock, and livelock.

See also: pitfall 7.2

5 Analysis and Design Pitfalls

5.1 You don't exploit related technology

When developing any agent system, the percentage of the design that is agent-specific (e.g., doing cooperation or negotiation, or learning a user's profile) is comparatively small. This conforms to the *raisin bread* view of system development, attributed to Winston [7], in which the parts of the system which can be considered agent-based conform to the small percentage of raisins and the more standard technology needed to build the majority of the system conforms to the significantly larger amount of bread. Given these relative percentages, it is important that conventional technologies and techniques are exploited wherever possible. Such exploitation speeds up the development process, avoids re-inventing the wheel, and enables sufficient time to be devoted to the value-added agent component. This point may seem obvious, but many agent projects fail to take it on board and, as a result, suffer in their development. While the exact set and degree of those technologies which are related varies between applications, many agent projects could benefit from exploiting available technology from the following fields: distributed computing platforms (such as CORBA [19]) to handle low-level inter-operation of heterogeneous distributed components; database systems to handle large information processing requirements; and expert systems to handle reasoning and problem solving tasks.

See also: pitfall 7.4

5.2 Your design doesn't exploit concurrency

There are, in general, many different ways of cutting up any particular problem. Decomposition can be made along functional, organisational, physical, or resource related lines. In terms of developing agent-based systems, no single approach is universally best. However, not all decompositions yield equally good solutions. System design is thus a crucial determinant of the project success — a poor design leads to poor exploitation of the agent metaphor which, in turn, leads to an unsuccessful project.

One of the most obvious features of a poor multi-agent design is that the amount of concurrent problem solving is comparatively small or even in extreme cases non-existent. Typically in poorly designed systems, one agent does some processing, produces some results, and then enters into an idle state. The results are passed onto another (previously inactive) agent which then processes them, produces more results, and returns to inactivity, and so on. This is an unsatisfactory design because there is only ever a single thread of control: concurrency, one of the most important potential advantages of multi-agent solutions, is not exploited. Concurrency allows the system to simultaneously handle multiple objectives and perspectives, to respond and react to the environment at many different levels, and to allow multiple complementary problem solving methods to cooperatively inter-work. Given this, one of the aims of the analysis and design phases is to produce a system which ensures a reasonable and appropriate amount of concurrent problem solving activity.

See also: pitfalls 7.2 and 7.6

6 Micro (Agent) Level Pitfalls

6.1 You decide you want your own agent architecture

Agent architectures are essentially designs for building agents [24]. Many agent architectures have been proposed over the years, to deal with many different types of problem domain; a good example of such an architecture is the Procedural Reasoning System (PRS) [11]. There is a great temptation, when first attempting an agent project, to imagine that no existing agent architecture meets the specific requirements of your problem, and that it is necessary to design one from first principles. Contributing to this temptation are several factors. The "not designed here" mindset, which only trusts products developed in-house, is one factor. The desire to generate intellectual property — either for profit or academic glory — is another. But designing an agent architecture from scratch in this way is usually a mistake, for several reasons. First, in order to develop a new architecture that is both reliable and that offers sufficient power to be usable takes years of effort — not person years, but *years*. This is time that could otherwise have been devoted to gaining experience with, and, ultimately, proving the technology. Second, unless the design process is carried out in tandem with a major research effort, it is unlikely that the architecture you end up with will be sufficiently novel to generate either interest or revenue. Our recommendation is therefore to study the various architectures described in the literature [24], and either license one or else implement an "off the shelf" design. This approach will not bring you architecture-related intellectual property or revenue, but it *will* get you developing applications quickly. (It is also worth observing that for many applications, a formal agent architecture is not actually required: it is quite sufficient to implement individual agents in the language best suited to the application.)

See also: pitfall 7.4

6.2 You think your architecture is generic

If you *do* develop your own architecture, then resist the temptation to believe that it is generic. Many agent-based architectures have been developed, that deal with both the *micro* (agent) and *macro* (society) levels of agent systems. Typically, these architectures are developed by building a solution for a particular problem, and then generalising. There is a temptation, having developed a successful agent solution, to imagine that the architecture and techniques developed for one problem domain can be directly applied to another. But this is a fallacy: it inevitably leads one to attempting to apply an architecture to a problem for which it is patently unsuited. If you have developed an architecture that has successfully been applied to some particular problem, try to understand *why* that particular architecture succeeded with that particular problem. Only attempt to apply the architecture to problems with similar characteristics.

See also: pitfall 3.3

6.3 Your agents use too much AI

When one builds an agent application, there is an understandable temptation to focus exclusively on the agent specific aspects of the application. After all, these are seen as the justification for the project in the first place. If ones does this, then the result is often an agent framework that is too overburdened with experimental AI techniques (first principles planners, theorem provers, reason maintenance systems, ...) to be usable. This problem is fuelled by a kind of "feature envy", where one reads about agents that have the ability to learn (or plan, or communicate in natural language, ...), and imagines that such features are essential in one's own agent system. In general, a more successful strategy is to build agents with a minimum of AI techniques; as success is obtained with such systems, they can be progressively evolved into richer systems. This is what Etzioni calls the "useful first" strategy [7].

See also: pitfall 5.1

6.4 Your agents have no intelligence

While at one extreme, we find developers obsessed with developing agent systems that employ only the most sophisticated and complex AI techniques available (and as a consequence fail to provide a sufficiently robust basis for the system), at the other, we find so-called agents that do nothing to justify the use of the term. For example, it is becoming increasingly common to find straightforward distributed systems referred to as multi-agent systems. Another very different, but equally common example, is the practice of referring to WWW pages that have any behind the scenes processing as "agents". Such practices are unhelpful, for the following reasons. First, they will lead to the term "agent" losing any meaning it has. Second, they raise expectations of software recipients, who will only be disappointed when they ultimately receive a very conventional piece of software. Finally, they lead to cynicism on the part of software developers (who come to believe that the term "agent" is simply another meaningless management buzzword).

See also: pitfalls 2.1, 4.1, 4.2, 7.4

7 Macro (Society) Level Pitfalls

7.1 You see agents everywhere

When one learns about multi-agent systems for the first time, there is a tendency to view *everything* as an agent. This is perceived to be in some way conceptually clean — after all, an object-oriented language is considered "pure" if everything in the language is an object — isn't the situation the same for multi-agent systems? If one adopts this viewpoint, then one ends up with agents for everything — including agents for addition and subtraction. In the enormously influential ACTOR paradigm of concurrent computation [1], this is pretty much what happens. In order to do some computation, actors (which are very similar to agents) must be spawned to do the various components of the computation. These actors in turn spawn more actors to do successively smaller parts of the computation, and so on. Eventually, the computations required are so small that they are carried out by "built in" actors (cf. native methods). But by the time a computation of even moderate size has bottomed out in this way, a great many actors will have been created, and a great deal of communication overhead will have been incurred. For example, in the classic "factorial" example, computing $n!$ requires the generation of n actors. It is not difficult to see that naively viewing *everything* as an agent in this way will be extremely inefficient: the overheads of managing agents and inter-agent communication will rapidly outweigh the benefits of an agent-based solution.

In general, agents should be *coarse grained*, in that each should embody significant, coherent computational functionality. While it is sometimes useful to view agents as being composed of further agents, one should be very careful how one applies this idea, as it can lead to enormous — and pointless — computational overheads.

See also: pitfall 7.2

7.2 You have too many agents

It is well-known that a number of systems interacting with one-another using simple rules can generate behaviour that appears to be considerably more complex than the sum of the components would indicate [21]. Therein lies one of the great strengths — and weaknesses — of multi-agent systems. The strength is that this *emergent functionality* can be exploited by the multi-agent system builder, to provide simple, robust cooperative behaviour. The weakness is that emergent functionality is akin to chaos [16]. In short, the dynamics of multi-agent systems are complex, and can be chaotic. It is often difficult to predict and explain the behaviour of even a small number of agents; with larger numbers of agents, attempting to predict and explain the behaviour of a system is futile. Often, the only way to find out what is likely to happen is to run the system — repeatedly. If a system contains many agents (many is often interpreted as greater than 10), then the dynamics can become too complex to manage effectively.

There are several techniques that one can use to try to manage a system in which there are many agents. First, one can place it under central control, perhaps by having a coordinator agent. Unfortunately, this is often impossible, and usually undesirable. Another way of keeping control is to severely restrict the way in which agents can interact with one-another. This can be done in several ways. First, one can ensure that there are few channels of communication between agents. The theoretical maximum number of communication channels in a system containing n agents is $\frac{n(n-1)}{2}$, in which case every agent can talk to every other agent. The minimum number of communication channels is $n-1$, in which case every agent can talk to just one other. (The idea of minimising the number of communication links between modules in a software system is, of course, not new — but there is often an assumption that, because agents are a new *type* of software, the old rules do not apply.) Another way in which a designer can try to keep a handle on multi-agent dynamics is by restricting the *way* in which agents interact. Thus *very* simple cooperation protocols are preferable over richer ones, with "one-shot" protocols (such as requesting and replying) being both adequate and desirable for many applications.

See also: pitfall 7.1

7.3 You have too few agents

While some designers imagine a separate agent for every possible task, others appear not to recognise the value of a multi-agent approach at all. They create a multi-agent system that completely fails to exploit the power offered by the agent paradigm, and develop a solution with a very small number of agents doing all the work. Such solutions tend to fail the standard software engineering test of *coherence*, which requires that a software module should have a single, coherent function. The result is rather as if one were to write an object-oriented program by bundling all the functionality into a single class. It can be done, but it is not pretty. In addition, such solutions tend not to exploit concurrency.

See also: pitfall 5.2

7.4 You spend all your time implementing infrastructure

One of the greatest obstacles in the way of the wider use of agent technology is that there are no widely-used software platforms for developing multi-agent systems. Such platforms would provide all the basic infrastructure (for message handling, tracing and monitoring, run-time management, and so on) required to create a multi-agent system. As a result, almost every multi-agent system project that we have come across has had a significant portion of its budget devoted to implementing this infrastructure from scratch. During this implementation stage, valuable time (and hence money) is often spent implementing libraries and software tools that, in the end, do little more than exchange KQML-like messages ([18]) across a network. By the time these libraries and tools have been implemented, there is frequently little time, energy, or enthusiasm left to work either on the agents themselves or on the cooperative/social aspects of the system.

A related issue is that infrastructure is often implemented by developers with a background in artificial intelligence, rather than networks or distributed systems. As a result, the infrastructure is often naive with respect to communications, and is too unreliable or badly designed to be of any real value. The system one ends up with is then simply a poorly designed distributed system.

See also: pitfall 5.1

7.5 Your system is anarchic

A common misconception is that agent based systems can be developed simply by throwing together a number of agents in a melting pot; that the system requires no real structuring and all the agents are peers. While this may be true in certain cases, it should not be viewed as the *only* way of developing agent societies. Many agent systems require considerably more system-level engineering than this. For large scale systems, or for systems in which the society is supposed to act with some commonality of purpose, this is particularly true. In such cases, a means of structuring the society is needed to reduce the system's complexity, to increase the system's efficiency, and to more accurately model the problem being tackled. The precise nature of this structuring is clearly dependent on the problem at hand, but common options include [3]: close-knit teams of agents working together to achieve a common goal; abstraction hierarchies modelling the problem from different perspectives; and intermediaries acting as a single point of contact for a number of agents.

See also: pitfall 4.4

7.6 You confuse simulated with real parallelism

Almost every multi-agent system starts life as a prototype, with all agents running on a single computer. The agents are often implemented as UNIX processes, lightweight processes in C, or JAVA threads. But crucially, the system starts life with *simulated* distribution: the agents are not *really* distributed across a network. The advantages of starting a multi-agent project by simulating distribution are obvious — apart from any other considerations, not many institutions can provide a dedicated network of expensive servers for a demonstrator project. However, there is a tendency to assume that results obtained with simulated distribution will immediately scale up to *real* distribution. This is a very dangerous fallacy: distributed systems are an *order of magnitude* more difficult to design, implement, test, debug, and manage. There are innumerable practical problems in building distributed systems, from the mundane (how does one start up a number of agents running on different machines, perhaps in many different physical locations?) to the research level (how can one coordinate the actions of the agents, ensuring that deadlock and livelock do not occur?)

Another manifestation of this problem involves assuming that a development methodology which worked for simulated distribution will also work for a truly distributed system. Again, the problem of developing a truly distributed system is an an order of magnitude more than that of developing a centralised one: the development methodology cannot be assumed to scale up.

Perhaps the heart of the problem is that with simulated distribution, there is the possibility of centralised control — a fact that is exploited in many experimental testbeds, by providing a single trace facility and "control panel". In truly distributed systems, such centralised control is not possible unless one foregoes the advantages that distribution brings.

See also: pitfall 4.4

8 Implementation Pitfalls

8.1 The *tabula rasa*

When building systems using an emerging new technology, there is often an assumption that it is necessary to start from a "blank slate": every component of the system must be designed and built from scratch. Often, however, the most important components of a software system will be *legacy*: functionally essential, but technologically obsolete software components, which cannot readily be rebuilt. Such systems are often mission critical. When proposing a new software solution, it is essential to work *with* such components, since they can in general neither be ignored nor replaced. Such systems can be incorporated into an agent system by *wrapping* them with an *agent layer* [13]. The basic idea is to enable legacy components to communicate and cooperate with agents by proving them with a software layer that realises an agent-level application program interface (API). In this way, the functionality of the legacy software can be extended by enabling it to work with other newly developed software components (agents).

See also: pitfall 5.1

8.2 You ignore *de facto* standards

In a field as new as agent systems, there are few established standards that a developer can make use of when building the agent-specific components of an application. This is particularly true of the communication and cooperation components. Although there *are* initiatives underway to establish such standards [9], at the time of writing these efforts are still at a preliminary stage. As a consequence, developers often believe they have no choice but to design and build all agent-specific components from scratch, with the result that agents developed by different organisations are unable to inter-operate in any way. However, despite the lack of internationally accepted standards, there *are* a number of *de facto* standards in the area, which may usefully be employed in many cases. The most obvious example is KQML [18], an agent communication language (ACL) that has been employed in many agent development projects.

See also: pitfalls 5.1, 7.4, 8.1

9 Conclusions

There are good arguments in support of the claim that agent technology will prove to be a valuable tool for building complex dis-

tributed systems. But as yet, these arguments are unsupported by much substantial evidence: agent technology is essentially immature and untested. With no body of experience to guide them, agent system developers tend to find themselves falling into the same traps. In this paper, we have described what we perceive to be the most common and most serious of these pitfalls. We thereby hope to have initiated a debate on the pragmatic, engineering aspects of agent-based systems. In future, we intend to consolidate this work by investigating development methodologies for agent-based systems. Such methodologies will provide a systematic framework that can be used to address the pragmatic concerns of software engineers charged with the development of agent-based systems.

Acknowledgement

The authors would like to thank Simon Lewis, for pointing them at [22], and hence providing a model within which this paper could be framed.

References

[1] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press: Cambridge, MA, 1986.

[2] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[3] A. H. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers: San Mateo, CA, 1988.

[4] M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press: Cambridge, MA, 1987.

[5] F. P. Brooks. No silver bullet. In H.-J. Kugler, editor, *Proceedings of the IFIP Tenth World Computer Conference*, pages 1069–1076. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1986.

[6] E. H. Durfee, D. L. Kiskis, and W. P. Birmingham. The agent architecture of the university of michigan digital library. *IEE Transactions on Software Engineering*, 144(1):61–71, February 1997.

[7] O. Etzioni. Moving up the information food chain: Deploying softbots on the world-wide web. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, OR, 1996.

[8] I. Ferguson and M. Wooldridge. Paying their way: Commercial digital libraries for the twenty-first century. *dLib Magazine: The Journal of Digital Library Research*, June 1997.

[9] The Foundation for Intelligent Physical Agents. See `http://drogo.cselt.stet.it/fipa/`.

[10] S. Franklin and A. Graesser. Is it an agent, or just a program? In J. P. Müller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III (LNAI Volume 1193)*, pages 21–36. Springer-Verlag: Berlin, Germany, 1997.

[11] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.

[12] M. P. Georgeff and A. S. Rao. A profile of the Australian AI Institute. *IEEE Expert*, 11(6):89–92, December 1996.

[13] N. R. Jennings, J. Corera, I. Laresgoiti, E. H. Mamdani, F. Perriolat, P. Skarek, and L. Z. Varga. Using ARCHON to develop real-world DAI applications for electricity transportation management and particle accelerator control. *IEEE Expert*, dec 1996.

[14] N. R. Jennings, J. M. Corera, and I. Laresgoiti. Developing industrial multi-agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 423–430, San Francisco, CA, June 1995.

[15] N. R. Jennings and M. Wooldridge. Applying agent technology. In N. R. Jennings and M. Wooldridge, editors, *Agent-based computing: Markets and Applications*. Springer-Verlag: Berlin, Germany, 1998.

[16] J. O. Kephart, T. Hogg, and B. A. Huberman. Dynamics of computational ecosystems: Implications for DAI. In L. Gasser and M. Huhns, editors, *Distributed Artificial Intelligence Volume II*, pages 79–96. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1989.

[17] Ovum Ltd. Intelligent agents: The next revolution in software, 1994.

[18] J. Mayfield, Y. Labrou, and T. Finin. Evaluating KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI Volume 1037)*, pages 347–360. Springer-Verlag: Berlin, Germany, 1996.

[19] The Object Management Group (OMG). `http://www.omg.org/`.

[20] A. S. Rao and M. P. Georgeff. Asymmetry thesis and side-effect problems in linear time and branching time intention logics. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 498–504, Sydney, Australia, 1991.

[21] L. Steels. Cooperation between distributed agents through self organization. In Y. Demazeau and J.-P. Müller, editors, *Decentralized AI — Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89)*, pages 175–196. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[22] B. F. Webster. *Pitfalls of Object-Oriented Development*. M&T Books (New York), 1995.

[23] M. Wooldridge. Agent-based software engineering. *IEE Transactions on Software Engineering*, 144(1):26–37, February 1997.

[24] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.