A New DAI-Based Software Paradigm

Michael N. Huhns
Department of Electrical and Computer Engineering
University of South Carolina
Columbia, South Carolina 29208
U. S. A.

Abstract

This panel presentation will describe a new approach to the production of robust software. We first motivate the approach by explaining why the two major goals of software engineering—correct software and reusable software—are not being addressed by the current state of software practice. We then describe a methodology based on active, cooperative, and persistent software components, i.e., agents, and show how the methodology enables robust and reusable software to be produced.

1 Introduction

The field of software engineering has exhibited almost glacial progress over the last twenty years, and appears to be in a malaise. Programmers still produce approximately the same number of lines of tested and debugged code per day as they did in 1975, in spite of several "silver bullets," such as structured programming, declarative specifications, object-oriented programming, formal methods, and visual languages.

This should not be surprising, for three reasons:

1. Software systems are the most complicated artifacts people have ever attempted to construct
2. Software systems are (supposedly) guaranteed to work correctly only when all errors have been detected and removed, which is infeasible in light of the above complexity
3. The effect of an error is unrelated to its size, i.e., a single misplaced character out of millions can render a system useless or, worse, harmful.

Software engineering attempts to deal with these problems by considering both the process of producing software and the software that is produced. The major goal for the software is that it be correct, and the major goal for the process is that it be conducted efficiently. One fundamental approach to meeting these goals is to exploit modularity and reuse of code. The expectations are that small modules are easier to test, debug, and verify, and therefore more likely to be correct, that small modules will be more likely to be reused, and that reusing debugged modules is more efficient than coding them afresh. This is the major result of the series of programming paradigms that have evolved from the machine language of the 1950's:

1. Imperative paradigm: procedure-oriented programming
2. Declarative paradigm: functional and logic programming
3. Interactive paradigm: object-based and distributed programming.

However, software has not kept pace with the increased rate of performance for processors, communication infrastructure, and the computing industry in general [Lewis 1996]. Whereas processor performance has been increasing at a 48% annual rate and network capacity at a 78% annual rate, software productivity has been growing at a 4.6% annual rate and the power of programming languages and tools has been growing at an 11% annual rate. CASE tools, meant to formalize and promote reuse, have not been widely adopted [Iivari 1996]. In addition to these sluggish rates for software, there is a legacy of approximately 50 billion lines of Cobol, representing roughly 80% of all software written since 1960. It is unlikely that we can replace it anytime soon, even though maintaining it is a $3 billion annual expense.

The current "hot" computing paradigm is based on Java, and the ability it provides for users to download the specific functionality they want at the moment they request that functionality. This is leading to the rise of a software-component industry, which will produce and then distribute on demand the components that have the users' unique desired functionality [Yourdon 1996]. However, because of this uniqueness, how can component providers be confident that their components will behave properly? This is a problem that can be solved by agent-based components that actively cooperate with other components to realize the user's goals.

1.1 A New Software Paradigm

Therefore, it is time to consider a completely different approach to software systems. We propose one based on the (intentionally provocative) recognition that

- errors will always be a part of complex systems
- error-free code can at times be a disadvantage
- where systems interact with the complexities of the physical world, there is a concomitant power that can be exploited.

We suggest an open architecture consisting of multiple, redundant, agent-based modules interacting via a verified kernel. The appropriate analogy is that of a large, robust, natural system. We motivate our approach by means of the following examples.

2 Requirements for New Applications

There are a new class of applications evolving thanks to ongoing advances in computer systems. The applications have characteristics that lead naturally to an agent-based approach to their development [Woelk et al. 1995]:

- They solve a specific business problem by providing a user with seamless interaction with remote information, application, and human resources.
- The identities of the resources to be used are mostly unknown at the time the application is developed.
- The pattern of interaction (workflow) among the resources is a critical part of the application, but the pattern might be unknown at the time the application is developed and might vary over time.

The development of these new applications requires improved programming languages and improved system services. These are not alternatives to such capabilities as OMG CORBA and Microsoft DCOM, but rather advanced features implemented at a higher level of abstraction and useful across multiple heterogeneous distributed computing environments.

Because each application executes as a set of geographically distributed parts, a distributed active-object architecture is required. It is likely that the objects taking part in the application were developed in various languages and execute on various hardware platforms. A simple, powerful paradigm is needed for communications among these heterogeneous objects. Due to the distributed nature of the application, an object might not always be available when it is needed. For example, an object executing on a PDA might be out of physical communications with the rest of the application.

Because the identities of the resources are not known when the application is developed, there must be an infrastructure to enable the discovery of pertinent objects. Once an object has been discovered, the infrastructure must facilitate the establishment of constructive communication between the new object and existing objects in the application.

Because the pattern of interaction among the objects is a critical part of the application and may vary over time, it is important that this pattern (workflow) be explicitly represented and available to both the application and the user. When an object has been discovered to be relevant to an application, the language for interaction and the pattern of interaction with the object must be determined. This interaction becomes part of the larger set of object interactions that make up the application. The objects collaborate with each other to carry out the application task.

Fundamentally, most business software modules are intended to be models of some real object within the business. A problem is that these modules are passive, unlike most of the objects they represent.

3 Multiagent-Based Development

Multiagent-based interoperation is a new paradigm distinguished by the features that requests are specified in terms of "what" and not "how," agents can take an active role, monitoring conditions in their environment and reacting accordingly, and agents may be seen as holding beliefs about the world.

We have initiated development of a new paradigm—a cooperative paradigm—based on interacting agents, active objects, and active wrappers of legacy components. The resultant methodology and language, interaction-oriented programming, represent a fundamental extension of the earlier paradigms, with greater expressive power, different conceptual foundations, such as beliefs held by components, and new modeling techniques.

References

[Iivari 1996] Juhani Iivari, "Why Are CASE Tools Not Used?" *Communications of the ACM*, Vol. 39, No. 10, October 1996, pp. 94-103.

[Lewis 1996] Ted Lewis, "The Next $10,000_2$ Years: Part II," *IEEE Computer*, May 1996, pp. 78-86.

[Woelk et al. 1995] Darrell Woelk, Michael Huhns, and Christine Tomlinson, "Uncovering the Next Generation of Active Objects," *Object Magazine*, July-August 1995, pp. 33-40.

[Yourdon 1996] Edward Yourdon, "Java, the Web, and Software Development," *IEEE Computer*, August 1996, pp. 25-30.