Software Services: Scientific Challenge or Industrial Hype?

José Luiz Fiadeiro

Department of Computer Science, University of Leicester,
University Road, Leicester LE1 7RH, UK
jose@fiadeiro.org

Abstract. Web-services keep making headlines, not only in technical journals but also in the wider media like The Economist. Is this just a sales plot of the fragile software industry targeted to the companies and organisations that want to operate in the new economy as enabled by the internet and wireless communication? Or is there a new paradigm as far as software development is concerned? Should we, scientists, regard this as a challenge? Or dismiss it as hype? In this paper, we analyse these questions in the context of the notions of complexity that arise in software development and the methods and techniques that can be offered to address them.

1 Introduction

> ***component*** *(n): a constituent part*
> ***complex*** *(a): composed of two or more parts*
> ***architecture*** *(n):*
> ***1*** *: formation or construction as, or as if, the result of conscious act;*
> ***2*** *: a unifying or coherent form or structure*

Hardly anybody working in computer science or software engineering can claim to be immune to the hype that surrounds "web services". However, in spite (or because…) of all the frenzy, it is not clear whether there is any room for a real scientific discussion. After all, it is the big companies that have been driving most of the activity in this area. This is why many people in academia and research are asking if this isn't just a sales plot of the software industry targeted to the companies and organisations that want to operate in the internet… Is there really a new paradigm as far as software development is concerned? Should we, scientists, regard this as a challenge? Or dismiss it as mere industrial hype?

One of arguments made in favour of a new discipline, and a line that one could envisage pursuing in a scientific debate, opposes "components" to "services". However, the term "component" is being used more and more frequently in software engineering, at the expense of conveying less and less meaning. People are also drawing analogies with the use of these concepts in arts, science, and engineering without a clear sense of purpose. This is raising more confusion and less confidence in the usage of methods and tools being advertised for component-based development or software architecture design.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 1–13, 2005.

Nevertheless, bringing the term component into the arena is important because it points a finger to one of the crucial dimensions of (software) engineering: *complexity*. Decomposition of a problem into sub-problems is the best (only?) way that humans have found to tackle complexity. Every discipline of decomposition leads to, or is intrinsically based on, a notion of component and composition: *ça va de soi*! The way we decompose a problem, or the discipline that we follow in the decomposition, further leads to an architecture, or architectural style, that identifies the way the problem is structured in terms of sub-problems and the mechanisms through which they relate to one another.

This remark can hardly be classified as a deep insight but the fact is that the differences that we can witness in the use of the terms "software component" and "software architecture" can be attributed to the simple fact they address different notions of complexity that arise in the process of engineering software systems. Hence, can we frame the debate on services in the context of complexity? Are services the components of a new architectural approach? If so, for which notion of complexity?

In our opinion, any answer to this question requires an analysis of the forces that have made software development practice evolve over the past 50 years or so, as well as the recognition of some of the fundamental milestones of this evolution. The purpose of this paper is, precisely, to guide the reader through a journey in the history of software engineering, hoping that, at the end, a clear case for a new paradigm will have emerged. In this process, we will make use of the following figure borrowed from [24]:

2 In the Beginning…

Fig. 1 makes clear the fact that, in the early days, software development took place "in-the-head" of a person (a term that we prefer to "any-which-way"). That person had a problem that, typically, consisted of some complex computation needed to obtain some important result (e.g. the next best move during a game of chess) that the person would consume upon termination. To solve that problem, the person would develop a program to run on a particular machine.

2.1 From Programming "in-the-head" to "in-the-small"

Programming took place "in-the-head" in the sense that it did not concern anybody else except the programmer: only the results of the execution were needed, the program being just a means to that end. The program thus built would reflect very closely the architecture of the machine available to run it. The programmer would often have to resort to all sorts of "tricks" to get around the limitations of memory and speed. This is why it seems unjust to qualify this activity as programming "any-which-way" as it often required a deep knowledge of the target machine. In any case, programming was a one-off activity best performed by virtuosi in absolute control of the execution infrastructure and with the final result of the execution as the primary goal of the activity.

This changed when, instead of the result of the execution, the programmer had the solution (as embodied in the program itself) as a business goal. For instance, instead

1960 ± 5 Programming-any-which-way	*1970 ± 5 Programming-in-the-small*	*1980 ± 5 Programming-in-the-large*	*1990 ± 5 Programming-in-the-world*
Mnemonics, precise use of prose	Simple input-output specifications	Systems with complex specifications	Distributed systems with open-ended, evolving specs
Emphasis on small programs	Emphasis on algorithms	Emphasis on system structure, management	Emphasis on subsystem interactions
Representing structure, symbolic information	Data structures and types	Long-lived databases	Data & computation independently created, come and go
Elementary understanding of control flow	Programs execute once and terminate	Program systems execute continually	Suites of independent processes cooperate

Fig. 1

of a chess fanatic developing a program for his pocket calculator to compute the next move on a given configuration, we are now talking of a scenario in which a chess-playing program is developed to be sold to clients who will run it themselves on their machines for their own purposes. A crucial landmark is thus reached: programs, instead of the results of their executions become the goods. In other words, software becomes a product.

In order to make commercial sense, it is essential to develop programs that can be run in different machines. Programming becomes an activity that cannot be purely conducted "in-the-head" as it needs to take into account that the resulting software is to be commercialised. The separation between program and code executable on a particular computer is supported by machine-independent programming languages and compilers. In fact, this separation consists of an abstraction step in which the program written by the programmer is seen as a higher-level abstraction of the code that runs on the machine.

A crucial aspect of this abstraction process is the ability to work with data structures that do not necessarily mirror the organisation of the memory of the machine in which the code will run. This process can be taken even further by allowing the data structures to reflect the organisation of the solution to the problem, even if they are not available in the target programming language. Specification languages support the definition of such data structures and the high-level programs that use them.

Program development methodologies [6,] further address the problem of developing "real" programs from such high-level descriptions. They help the software developer arrive to a solution to the original problem regardless of the fact that the resulting program is for self-consumption of for sale. For instance, in the 70s, so-called *structured programming* provided abstractions for controlling execution that introduced a totally new discipline into software development by separating control flow from the text of programs. Before, control flow was largely defined in terms of GOTO state-

ments that transfer execution to a label in the program text. Structured programming provides constructs such as "if-then-else" and "while-do" for creating a variety of control execution patterns that can be understood independently of the order in which the program text is written.

2.2 Program Architectures

Through methods supporting programming in-the-small we are led to notions of program component and architecture that allow us to tackle the complexity of controlling the flow of execution. In Fig. 2, we present the architecture of a run length encoder[1] in JSP [15], one of the methods that introduced structured programming. In JSP, program development follows a top-down approach in the sense that blocks are identified and put together according to given control structures (sequential composition, iteration, etc). Each block is then developed in the same way, independently of the other blocks. The criteria for decomposition derive from the structure of the data manipulated by the program.

The advantage of this architectural representation is that it decomposes control flow according to the structure of the input data into well-identified components. Each of these can be individually programmed and put together using the primitives of the specific programming language that is chosen for a particular implementation. Different methodologies lead to different architectures, of course.

In what concerns the software industry, it is clear that programming methodologies have a significant impact in the delivery time and cost of the final product. By allowing the programmer to work at higher levels of abstraction, results from the theory of algorithms and complexity can be used for controlling performance in space and time, which is an important factor of quality. When seconded by mathematical semantics, a method and associated language can even assist the proof of the correctness of the product with respect to a high-level specification of its functionality as given, for instance, through input/output specifications. This further adds to the quality of the final product.

It is not our purpose in this paper to promote any specific such language and method, especially because the debate on what consists good support for program construction is not closed and new methods/languages keep being proposed. Nevertheless, we would like to mention *artificial intelligence* as a methodology that provides abstractions for programming that derive from the way humans solve problems, and *object-oriented programming* as a discipline based on the packaging of data and functionality together into units called objects.

Finally, we would like to point out that we have been discussing abstractions for handling the complexity of solving a given problem in terms of a computer program. In this activity, the complexity lies more in the nature of the problem that needs to be understood and the process of coding it than in the resulting solution (application). For instance, programs that play chess, unlike some of their mechanical ancestors, are

[1] A run length encoder is a program that takes as input a stream of bytes and outputs a stream of pairs consisting of a byte along with a count of the byte's consecutive occurrences in the input stream.

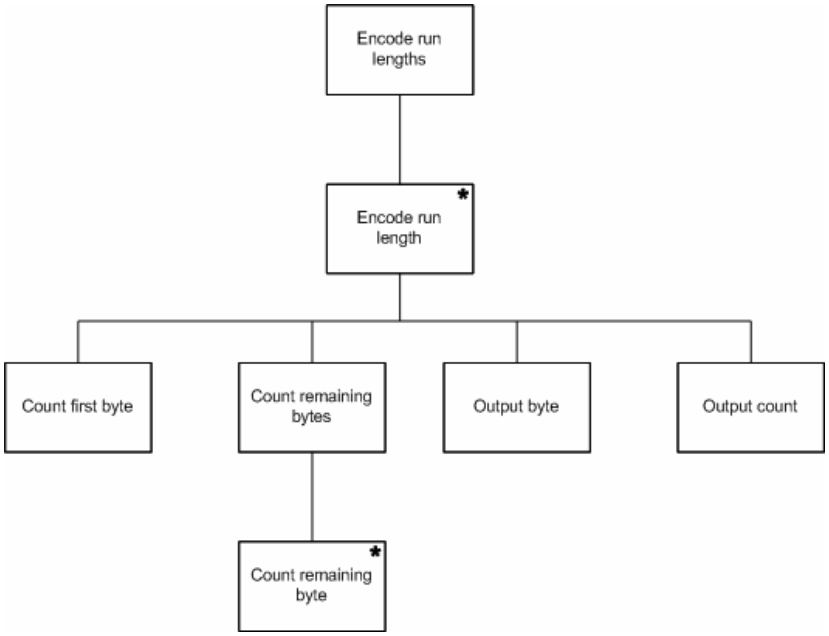

Fig. 2

not known for the complexity of their structure in terms of the modules/parts that they are assembled from. They give headaches to coders, not to project managers and their maintenance groups. Indeed, chess playing is seen more as a testing ground for artificial intelligence than software engineering.

3 The "Software Crisis"

In September 1994, an article in the Scientific American alerted to the "Software's chronicle crisis": software was recognised to be manufactured as in a "cottage industry" and not according to the "industrial standards of mass production and reliability. The article identifies the underlying problem as being one of "complexity":

> *The challenge of complexity is not only large but also growing. […]. To keep up with such demand, programmers will have to change the way that they work. "You can't build skyscrapers using carpenters," Curtis quips.*

3.1 Programming in-the-Large

The problem identified in this article was known for many years when it was published, certainly since the famous 1968 NATO conference in Garmisch-Partenkirschen. What is significant about this article is the fact that it appeared in the Scientific American, a publication that reaches an audience much wider than computer scientists and software engineers. The reason it deserved being published in

such a journal was that the general public had just been hit by one of the most famous software-related failures: the luggage delivery system that kept the brand new Denver airport shut for months at a huge expense. In other words, it is not that the problem had suddenly started to give headaches to software developers but that it became clear that it was hurting the economy, i.e. reaching into people's pockets.

Indeed, as the scope and role of software in business grew, so did the size of programs: software applications were (and still are) demanded to perform more and more tasks in the business domain and, as a consequence, they grew very quickly into millions of lines of code. Sheer size compromised quality: delivery times started to suffer and so did performance and correctness due to the fact that applications became unmanageable for the lone programmer. The analogy with building skyscrapers using carpenters is a very powerful one. Engineering principles were quickly identified to be required to face the complexity of the product and the term programming "in-the-large" was coined to reflect the fact that software development needed another activity to be supported: one that could break the task into manageable pieces [5].

> *We distinguish the activity of writing large programs from that of writing small ones. By large programs we mean systems consisting of many small programs (modules), possibly written by different people.[…]*
>
> *We argue that structuring a large collection of modules to form a "system" is an essentially distinct and different intellectual activity from that of constructing the individual modules. That is, we distinguish programming-in-the-large from programming-in-the-small.*

This is where a second important landmark in the history of Software Engineering is normally placed. Please note that these are just milestones: the columns in Fig. 1 should not be taken as disjoint periods in this history. In fact, one should ignore the reference to the decades (60s, 70s, 80s and 90s) as they are neither accurate nor identifiers of periods in the history of Software Engineering. Indeed, there is still a role for programming-in-the-small, as recognised above, and for programming-in-the-head, e.g. for software embedded in some critical systems.

3.2 Module Interconnection Languages

It should be clear that programming in-the-large addresses an altogether different notion of complexity, one that occurs at "design" or "compile" time. We are not so much concerned with the flow of execution of a computation but with "workflow" in a development process.

A different kind of decomposition is, therefore, at stake: one that addresses the global structure of a software application in terms of what its modules and resources are and how they fit together in the system. The resulting components (modules) are interconnected not to ensure that the computation progresses towards the required output, but that, in the final system, all modules are provided with the resources they need (e.g. the parsing module of a compiler is connected to the symbol table). In other words, it is the flow of resources among modules that is of concern. Therefore, one tends to use primitives such as *export/provide/originate* and *import/require/use* when designing individual modules.

The conclusions of Parnas' landmark paper [20] are even clearer in distinguishing program complexity/architecture from the complexity that is associated with programming "in-the-large":

> *We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules.*

That is to say, we cannot hope and should not attempt to address the complexity of software systems as products with the mechanisms that were developed for structuring complex computations. That is why so-called *module interconnection languages* (MILs) were developed for programming in-the-large [22].

In the architectures that are described in such languages, dependencies between components concern access to and usage of resources, not control flow. Whereas program architectures make it much simpler to understand and prove the correctness of the code with respect to input/output specifications, module interconnection architectures are essential for project management, namely for testing and maintenance support: they enforce system integrity and inter-modular compatibility; they support incremental modification as modules can be independently compiled and linked, and thus full recompilation of a modified system is not needed; and they enforce version control as different versions (implementations) of a module can be identified and used in the construction of a system.

We should also make clear that problems of "size" do not just arise during software design. Even if specification languages can factor down the size of code by at least one order of magnitude, they do not factor out complexity altogether. For instance, in algebraic specifications, which consist of sets of sentences (axioms) in a given logic [17], the need to structure these sets in manageable pieces was recognised as early as 1977 in a famous article by Burstall and Goguen whose title is, precisely, "putting theories together to make specifications" [4]. This concern for the complexity of specifications signalled the advent of category theory [8] as a mathematical toolbox offering techniques for structuring logical theories into what became known as the "theory of institutions" [14]. Modularisation principles were extensively explored in this setting that contributed to the maturation of software development as an engineering discipline [13,25].

Other problems of "size" are also reflected in Fig. 1, e.g. in the data that some software applications are required to manipulate, which led to the development of database technologies. The same applies to control structures in the sense that termination ceased to be a correctness factor and properties of on-going execution like responsiveness started to emerge in applications such as operating and air traffic control systems.

3.3 The Case for Object-Oriented and Component-Based Development

The article in the Scientific American goes one step further and offers possible ways out of the crisis:

> *Musket makers did not get more productive until Eli Whitney figured out how to manufacture interchangeable parts that could be assembled by any skilled workman. In like manner, software parts can, if properly standardized, be reused at many different scales. […]*
>
> *In April [1994], NIST announced that it was creating an Advanced Technology Program to help engender a market for component-based software.*

Indeed, the publication of this article is also marked by the advent of object-oriented (OO) and component-based development. In the context of the "small"/"large" divide, OO makes important contributions:

- State encapsulation provides a criterion for modularising code: software is organised in classes that group together in methods all the operations that are allowed on a given piece of the system state.
- Programming-in-the-small is used within a class to define its methods.
- Clientship is used for interconnecting objects: an object can be declared to be a client of another object, and methods of the client can invoke the execution of methods of the server as part of their code.
- Inheritance is used for classifying and organising classes in hierarchies that facilitate reuse.

Object-orientation cannot be taken primarily as a means of "programming-in-the-large". In fact, one can argue that, in spite of grouping functionalities in classes, OO development could do with additional mechanisms for managing huge collections of classes… Organising classes in inheritance hierarchies is a step in that direction but many would argue that it does not take software development deep enough into an engineering practice.

Still, one has to recognise that OO software development has brought a significant improvement to the management of the complexity of software development. When one considers alternative mechanisms for modularising imperative programming such as those introduced over Pascal to produce Modula, it is clear that OO is much richer in "methodological" contents in the sense that classes as software modules and interconnection via clientship, even if providing only for a rather fine grain of decomposition, organise systems according to structures that can be recognised in the problem domain.

4 The Crisis 10 Years Later

In spite of the recognised progress towards the management of the complexity of constructing large applications, an article published in May 2003 alerted to the fact that software was still under a "crisis":

Computing has certainly got faster, smarter and cheaper, but it has also become much more complex. Ever since the orderly days of the mainframe, which allowed tight control of IT, computer systems have become ever more distributed, more heterogeneous and harder to manage. […]

In the late 1990s, the internet and the emergence of e-commerce "broke IT's back". Integrating incompatible systems, in particular, has become a big headache. A measure of this increasing complexity is the rapid growth in the IT services industry. […]

Computing is becoming a utility and software a service. This will profoundly change the economics of the IT industry. […]

For software truly to become a service, something else has to happen: there has to be a wide deployment of web services. […]

Applications will no longer be a big chunk of software that runs on a computer but a combination of web services.

4.1 Programming in-the-World

One has to recognise that a different notion of complexity is involved here. The article is very explicit in saying that the problem now is not one of size – "large chunks of software" – but that the complexity lies in the fact that systems are ever more distributed and heterogeneous, and that software development requires the integration and combination of possibly "incompatible" systems. Societal and economical implications of this notion of complexity are not any smaller. The fact that this article appeared not in the Scientific American but in a wider circulation publication – The Economist – shows that the debate now concerns a much more general public.

In our opinion, one can realise that this crisis is of a different nature in the fact that the discussion is no longer around the complexity of building a large application that one needs to deliver, in time and budget, to a client, but of managing an open-ended structure of autonomous components, possibly distributed and highly heterogeneous. This means developing software components that are autonomous and can be interconnected with other components, software or otherwise, and managing the interconnections themselves as new components may be required to join in and other to be removed.

In software engineering, the term *software architecture* [3,10,21,24] has recently been reserved to this different kind of complexity. Components are treated as independent entities that may interact with each other along well-defined lines of communication called architectural connectors [19]. By focusing on particular kinds of components and connectors, one can identify different architectural styles. One can even compare the architectures induced by different styles on the same system and discuss system properties without having to analyse the code.

In a sense, we are going back to the kind of architecture provided by structured programming but at a higher level of abstraction, one which often involves abstractions not directly provided by the underlying programming language: pipes, filters, event broadcast, client-server protocols, etc. In other words, it is not so much the flow of control that we want to structure but the flow of interactions.

It is interesting to note that, from a mathematical point of view, category theory [8], as the mathematics of "structure", still plays a fundamental role in formalising these architectural principles and techniques [9]. However, instead of structuring large specifications, as discussed in Sect 3.2, we are now interested in run-time configurations of complex systems [12] and in the properties that emerge from the interactions within them [7].

Notice that, in this context, object-orientation can be clearly identified with an architectural style among many others. In this respect, the article in The Economist challenges us to identify an architectural style that can address the complexity of the new generation of systems that is emerging from the internet, mobile communication, and other such "global computers"[2] in what one could label programming "in-the-world" [24]. In this context, one can indeed debate the merits of an object-oriented style and discuss the role and status of services.

4.2 The Case for Services

Object-oriented techniques offer little support for the kind of decomposition, organisation and architectural style required for programming in-the-world. Interactions in OO are based on *identities* [16], in the sense that, through clientship, objects interact by invoking the methods of specific objects (instances) to get something specific done: to use another object's services, an object needs to have the server's identity to send it a message or call the required service. This implies that any unanticipated change on the collaborations that an object maintains with other objects needs to be performed at the level of the code that implements that object and, possibly, of the objects with which the new collaborations are established [23].

This in why some people claim that OO brought GOTOs back into fashion. One cannot but recognise that, indeed, clientship through feature calling and method invocation works for interactions in the same way as GOTOs worked for control flow. Indeed, one often has the feeling that HyperText and URLs in web-based IT applications tend to become the web designers' version of spaghetti code.

Hence, in our opinion, the challenges raised in The Economist show that a different paradigm is required to address what is clearly a different form of software complexity. This is, precisely, the paradigm that started to emerge in the guise of *web services* [1].

Web services have been often characterised as "self-contained, modular applications that can be described, published, located, and invoked over a network, generally the web" [27]. Building applications under this new paradigm is a dynamic process that consists in locating services that provide the basic functionalities that are required, and "orchestrating" them, i.e. establishing collaborations between them, at run-time, so that the desired global properties of the application can emerge from their joint behaviour, just in time.

---

[2] "A global computer is a programmable computational infrastructure distributed at worldwide scale and available globally. It provides uniform services with variable guarantees for communication, cooperation and mobility, modalities and disciplines for resource usage, security policies and mechanisms, and more." [26].

"Integration" is another keyword in this process, often found married to "orchestration" or "marshalling": application building in service-oriented architectures is based on the composition of services that have to be discovered and "marshalled" dynamically at run-time. Therefore, one of the characteristics of the service-oriented paradigm is, precisely, the ability that it requires for interconnections to be established and revised dynamically, in run-time, without having to suspend execution, i.e. without interruption of "service". This is what is usually called "late" or "just-in-time" integration (as opposed to compile or design time integration).

What marks the difference between this aspect of the complexity of software from the one addressed by programming-in-the-large is, precisely, the fact that software is not being treated as a *product* but as a *service*, as the article of The Economist makes clear. It is interesting to note that this shift from object/product to service-oriented interactions mirrors what has been happening already in the economy: more and more, business relationships are established in terms of acquisition of services (e.g. 1000 Watts of lighting for your office) instead of products (10 lamps of 100 Watts each for the office). That is, software engineering is just following the path being set for the economy in general and, thus, shifting somewhat away from the more traditional "industrial" technologies oriented to the production of goods to exhibit the problems that are characteristic of more "social" domains.

5 Concluding Remarks

We hope that the previous sections made clear that, in our opinion, the case for a new service-oriented paradigm rests, essentially, in the recognition that there is more than one dimension of complexity in the engineering of software intensive systems.

Indeed, even in our everyday life, we use the term "complex" in a variety of ways. Many times, we apply it to entities or situations that are "complicated" in the sense that they offer great difficulty in understanding, solving, or explaining. There is nothing necessarily wrong or faulty in them; they are just the unavoidable result of a necessary combination of parts or factors. For instance, the human body is a complex entity; none of its organs operates autonomously if separated from the whole but we know which vital functions each provides as part of the body, and can understand how the functioning of each of them depends on the rest to the extent that we can replace them by others of the same type.

In other circumstances, complexity derives more from the number and "open" nature of interactions that involve "autonomic" parts. Social systems are inherently complex in the sense that it is very difficult to predict what properties can emerge from the behaviours of the parts and their interactions. Regulations and regulators can be superposed to influence the way the parts interact or induce properties that one would like to see emerge, but complete control is hard to achieve.

Knowledge of the physiological structure of a part does not necessarily help in understanding how it can contribute to the functioning of a social system. For instance, the social behaviour of human beings is essentially independent of their physiology. One does not go to a psychiatrist because of a toothache (even if the toothache is driving us mad), or to the dentist complaining with stress. A car will usually have a driver's manual explaining how it can be used for social purposes (i.e. driving) and a

technical manual that a mechanic can use for fixing a faulty part: one does not consult the technical manual to find out where to switch the headlights and the mechanic does not need the driver's manual to replace a bulb. A mechanic does not need a driver's licence to repair a car, and knowing that one should stop at the red light is not something that derives from the structure of the car.

Having said this, we should point out something equally obvious: that they can very well be related. We all know that a speech impediment is likely to influence one's ability to socialise and that, without brakes, a car cannot be stopped at a red light.

In our opinion, this distinction applies to software as well. Programming in-the-large is concerned with the physiological complexity of software systems. Programming in-the-world with their social complexity. As such, the methods and techniques that best apply to one do not necessarily serve the other in the best possible way. We see component-based development as addressing in-the-large issues, as highlighted in the article of the Scientific American. We see services as addressing social complexity, as the article of The Economist clearly suggests.

Service-oriented architectures are still very much in their infancy, and still too much bound to the internet, in the guise of web services, or other specific global computers like the grid, in what are known as grid services. Service-based computing and software development is being uptaken by the IT industry in an ad-hoc and undisciplined way, raising the spectrum of a society and economy dependent on applications that have the ability to "talk" to each other but without "understanding" what they are talking about.

This scenario suggests very clearly that a scientific challenge is there to provide the mathematical, methodological and technological foundations that are required for the new paradigm to be used effectively and responsibly in the development of the generation of systems that will operate the Information Society of tomorrow. Such is the challenge that a consortium of European universities, research institutes and companies chose to address under the IST-FET-GC2 integrated project SENSORIA (Software Engineering for Service-Oriented Overlay Computers). SENSORIA is addressing the social complexity involved in service-oriented development precisely through some of the technologies that characterise programming "in-the-world": coordination languages and models [2,11], distributed reconfigurable systems [18], and software architectures [10,21,24].

References

1. G. Alonso, F. Casati, H. Kuno, V. Machiraju (2004) *Web Services*. Springer, Berlin Heidelberg New York
2. F. Arbab (1998) What do you mean, coordination? In: *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI),* March 1998
3. L. Bass, P. Clements, R. Kasman (1998) *Software Architecture in Practice*. Addison-Wesley, Reading MA
4. R. Burstall, J. Goguen (1977) Putting theories together to make specifications. In: R. Reddy (ed) *Proc. Fifth International Joint Conference on Artificial Intelligence,* August 1977, Cambridge, MA, pp 1045–1058
5. F. DeRemer, H. Kron (1976) Programming-in-the-Large versus programming-in-the-small. *IEEE Transactions on Software Engineering* SE-2(2): 321–327

6. E. Dijkstra (1976) *A Discipline of Programming*. Prentice Hall, London
7. J. L. Fiadeiro (1996) On the emergence of properties in component-based systems. In: M. Wirsing, M. Nivat (eds) *Algebraic Methodology and Software Technology. LNCS, vol 1101*. Springer, Berlin Heidelberg New York, pp 421–443
8. J. L. Fiadeiro (2004) *Categories for Software Engineering*. Springer, Berlin Heidelberg New York
9. J. L. Fiadeiro, A. Lopes (1997) Semantics of architectural connectors. In: M. Bidoit, M. Dauchet (eds) *Theory and Practice of Software Development*. *LNCS, vol 1214*. Springer, Berlin Heidelberg New York, pp 505–519
10. D. Garlan, D. Perry (1994) Software architecture: practice, potential, and pitfalls. In: *Proc. 16th International Conference on Software Engineering*. IEEE Computer Society Press, Silver Spring MD, pp 363–364
11. D. Gelernter, N. Carriero (1992) Coordination languages and their significance. *Communications ACM* 35(2):97–107
12. J. Goguen (1973) Categorical foundations for general systems theory. In: F. Pichler, R. Trappl (eds) *Advances in Cybernetics and Systems Research*. Transcripta Books, New York, pp 121–130
13. J. Goguen (1986) Reusing and interconnecting software components. *IEEE Computer* 19(2):16–28
14. J. Goguen, R. Burstall (1992) Institutions: abstract model theory for specification and programming. *Journal ACM* 39(1):95–146
15. M. Jackson (1975) *Principles of Program Design*. Academic Press, New York
16. W. Kent (1993) Participants and performers: a basis for classifying object models. In: *Proc. OOPSLA 1993 Workshop on Specification of Behavioral Semantics in Object-Oriented Information Modeling*
17. J. Loeckx, H.-D. Ehrich, M. Wolf (1996) *Specification of Abstract Data Types*. Wiley, New York
18. J. Magee, J. Kramer, M. Sloman (1989) Constructing distributed systems in Conic. *IEEE TOSE* 15(6):663–675
19. N. Mehta, N. Medvidovic, S. Phadke (2000) Towards a taxonomy of software connectors. In: *Proc. 22nd International Conference on Software Engineering*. IEEE Computer Society Press, Silver Spring MD, pp 178–187
20. D. Parnas (1972) On the criteria for decomposing systems into modules. In: *Communications of the ACM* 15(12):1053–1058
21. D. Perry, A. Wolf (1992) Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes* 17(4):40–52
22. R. Prieto-Diaz, J. Neighbors (1986) Module interconnection languages. *Journal of Systems and Software* 6(4):307-334
23. M. Shaw (1996) Procedure calls are the assembly language of software interconnection: connectors deserve first-class status. In: D. A. Lamb (ed) *Studies of Software Design*. *LNCS, vol 1078*. Springer, Berlin Heidelberg New York, pp 17–32
24. M. Shaw (1996) Three patterns that help explain the development of software engineering (position paper), *Dagstuhl Workshop on Software Architecture*
25. Y. Srinivas, R. Jüllig (1995) Specware™: formal support for composing software. In: B. Möller (ed) *Mathematics of Program Construction*. *LNCS, vol 947*. Springer, Berlin Heidelberg New York, pp 399–422
26. FET-GC2 Workprogramme text. www.cordis.lu/ist/fet/gc.htm#what
27. www.ibm.com/developerworks/web/library