An Outlook on Software Engineering for Modern Distributed Systems

Carlo Ghezzi and Gian Pietro Picco

Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci, 32, I-20133 Milano, Italy
Phone: +39-02-23993519, Fax: +39-02-23993411
E-mail: {ghezzi,picco}@elet.polimi.it

1 Introduction

The evolution of software engineering has been constant over the past thirty years. Some major technological discontinuities, however, can be identified in this progress, which caused a more radical rethinking of the previous established approaches. This, in turn, generated research for new methods, techniques and tools to properly deal with the new challenges. This paper tries to identify some of these major evolutionary steps from a historical viewpoint, with the goal of understanding if common treats can be found to characterize them.

We argue that major discontinuities indeed have an underlying common driving factor; namely, the goal of making software development increasingly more decentralized, flexible, and evolvable. Decentralization manifests itself in different forms: from the architecture viewpoint (from monolithic to fully distributed systems) to the process and business models.

After a brief outline of this evolution, we discuss the nature of the major discontinuity that we are currently facing, which is pushing decentralization, distribution and dynamic evolution to their extreme, and we try to identify the challenges to software engineering research that this is posing.

2 Architecting Software: Evolution and State of the Art

To better understand the situation that software engineering researchers need to face for the next years, it is instructive to look back at the evolution that occurred in the last decades.

In the beginning, software development methods typically assumed a stable environment (and hence stable requirements) and a fully centralized system architecture. These are the years of the waterfall lifecycle, advocated as a way to impose a much needed disciplined development style over the previous prevalent "code and fix" approach, which was recognized to be responsible for the lack of industrial quality of software [5].

From this rather static scenario, several evolutionary steps brought increasing degrees of dynamicity into software development. For instance, a first major shift was the recognition that requirements cannot be frozen before design and

implementation, since the environment typically changes, and the requirements with it. To cope with this problem, research addressed both methods and tools to support change. From the process viewpoint, this led to flexible and incremental models, like the spiral model and prototyping-based processes. Regarding methods, this lad to fundamental principles like design for change, encapsulation and hiding of design decisions, clear distinction between interface and implementation, and between implementation and specification [10]. Also, this led to modular and then object-oriented programming languages. Notably, however, the mechanisms devised to deal with software change where mostly static. A change would require modifying software, recompiling, and re-running the modified application. In a sense, dynamicity was introduced at the process level, but changes were still dealt statically at the product level.

Another significant step took place in application architectures, driven by changes in the underlying available physical architecture. From monolithic systems, we moved to distributed, client-server applications with a multi-tier structure. This made it possible to accommodate changes due to the growth of the system, such as adding new clients, or splitting a server into a–say–two tier structure. Efforts were made to make the shift easy, by allowing interaction between clients and servers to look like as if they would coexist on the same physical machine. This was possible through mechanisms like remote procedure call and middleware that would support them. This solution supported a clear separation of the issues of component allocation on physical nodes, which affected such properties as system performance, and conceptual structure of the application, which could be designed without bothering about physical distribution.

Another big leap was the need for decentralizing the responsibility to provide some needed functionality to pre-existing components or subsystems, and the ability of integrating in a distributed architecture such components or existing (legacy) subsystems. This was driven by two equally important issues, which both aim at decentralizing functionality and saving development costs. On the one side, the increasing availability of components-off-the-shelf to be used to build new applications, instead of starting developments from scratch. On the other, the need for reusing previously developed subsystems into new systems. This led to the need for architectures that would support interoperability of different types of distributed components, via suitable middleware [7].

Certainly, this summary of the history of software engineering is reductive. Still, it evidences how the evolution of software development has been largely driven by the need to accommodate increasing degrees of dynamicity, decentralization, and decoupling. Future scenarios show that this tendency is further exacerbated. Components are not only decoupled, but more and more autonomous in nature. Moreover, this is becoming true irrelevant of their size. Thus for instance, macrocomponents like clients for peer-to-peer file sharing are designed in a way that is largely independent of other components. At the same time, mobile code technology [8] enables microcomponents of the size of a language module (e.g., Java classes) to get relocated into a totally different execution context.

Dynamicity is also increasingly permeating software development [6]. For in-

stance, technological factors like the surge of wireless communications and mobile computing, together with factors like the structure of businesses run over the Internet, are defining application scenarios that are extremely dynamic. In the latter case, software components need to be deployed in environments that may change from time to time, due to changes in the business structure, or to adapt to the preferences of the user. In the former case, dynamicity must be dealt at an even finer time scale, since mobility defines a fluid system configuration where interactions among components may become only transient [16].

In these scenarios, assumptions about the existence of a single point of control, persistency, or authorization are often inefficient, impractical, or simply not applicable. More and more, the developer loses global control over the components belonging to the system, since they are autonomously and independently created and managed. Components can disappear either in an announced or an unannounced manner. Hence, system design must increasingly strive towards applications that are constituted by highly decoupled, autonomous, self-reconfiguring components. To some extent, the focus is no longer on component *integration*, rather it is shifted towards component *federation*.

It is interesting to note how some of the principles enunciated in the past for conventional systems are still valid, albeit interpreted in the new scenarios. For instance, Brooks' advices of "buy versus build", and "incremental development—grow, don't build, software" [2] acquire new meanings when transposed to the current environment. These statements have usually been associated with the need for reusing software components whenever possible, and with a software development process that favors evolution of the product across a number of small increments. Nowadays, "buy vs. build" is becoming even more crucial. For instance, often only a portion of a distributed system is totally under control of the designer; the rest is constituted by pre-existing components and services that is convenient—or mandatory—to exploit. The current interest for Web services, once filtered from hype, is symptomatic of this trend. Similarly, incremental development is now a necessity more than a choice, with the added complexity induced by the fact that more and more the growth of the system cannot be handled statically, rather it needs to be managed during the runtime.

In the next section we briefly survey some of what we believe are the most challenging (and intellectually stimulating) issues that arise in the environment defined by modern distributed systems, in an effort to help shaping a software engineering research agenda.

3 Research Challenges

The scenario depicted so far raises a number of fundamental questions, affecting software engineering practices.

It was argued that applications are increasingly built out of highly decoupled components. Nevertheless, components need to interact to become a true federation, i.e., to achieve the required behavior. An immediate question is then about the kind of interface components should provide to make them interact.

Traditionally, components define a list of features they export to and import from other components. A type system is typically used to ensure the correct use of such features, and more recently also to govern the lookup for a given service and the setup of the binding towards it, e.g., in Jini [12]. Nevertheless, the syntactic matching enabled by the type system is often not enough. Instead, one would like to match component services based on their semantics. Hence the increasing importance of languages for knowledge representation, and opportunities for synergies between the fields. Moreover, the services offered by a component are typically not known in advance. In this context, reflection techniques become a fundamental asset for allowing applications to discover dynamically the characteristics of a given component, as well as for allowing a given component to reconfigure itself dynamically. Finally, in such a fluid environment, one would have guarantees about the behavior of a given component. In this context, methods like "design by contract" [13] could gain even more importance than in conventional environments, if properly adapted.

The immediately next problem is how to make the components actually interact. In the traditional case, a monolythic system is built out of a set of components, and interaction is achieved by statically binding a service request from one component to a matching service supply from another component. This requires components to agree on some global naming scheme, which is then used to establish the binding. Static binding allows type checking to be also done statically, and thus type safe programs can be easily obtained. Object-oriented languages introduce more flexibility in this scheme, by introducing a form of constrained dynamic binding, still retaining the safety of static type checking.

Distributed computing platforms typically rely heavily on establishing a binding between the client and the supplier of some service. This is the case of mainstream middleware like RPC or distributed object technology. Desirable properties, e.g., location transparency, are usually achieved by exploiting an additional step of indirection when establishing the binding, e.g., by using a lookup service. Nevertheless, the question is whether such a tightly coupled model of interaction is still suitable in a fluid environment where components tend to be highly decoupled and their overall configuration is frequently changing.

By and large, two approaches have emerged to date. On the one hand, there are systems where interaction still occurs through a binding to another component, but dynamicity and reconfiguration is taken under account by allowing the targeted component to be changed transparently. This approach has its roots in earlier work on object migration (e.g., Emerald [1]). In this approach, binding occurs in two steps: first, a service discovery step matches the request for a service to the available offers. Second, a choice is made, and the binding is set. For these steps, we can envisage different levels of complexity and sophistication. For example, one may think that among the available services, a choice is made based on some offered quality of service (and maybe on price). One may even think of a negotiation that goes on to establish this. And even one may think of federating a number of subservices to match a service request.

The other approach completely decouples components by replacing the bind-

ing towards another component with a binding to some other external entity, that is assumed to be global to every component. This is the case of event-based systems [4, 3], where components react to the emission of events through the event dispatcher, and of systems [11, 14] inspired by Linda [9] where components communicate by exchanging information through a shared tuple space. In both cases, the problem of enabling component interaction in a highly dynamic, federated environment is regarded as a coordination problem. Coordination models and languages decouple sharply the internal behavior of components from the interactions they need to carry out. Typically, the latter is represented explicitly by using some kind of abstraction, like Linda tuple spaces. One reason why coordination approaches resonate with the problems we are concerned with in this paper is that, by modeling explicitly the space where component interactions take place, coordination models naturally represent the computational context for components. The notion of context is fundamental in dealing with the high dynamicity set by the scenarios we defined here. If components are to be decoupled from one another and yet able to interact with the rest of the world, they need to have some way to define what "rest of the world" means to them. Moreover, this is likely to change according to the application domain at hand, and/or different portion of the context need to be treated in a different way for different applications. We believe that one of the major research challenges for software engineering researchers is the definition of abstractions that are able to properly capture the essence of the notion of context and its inherent dynamicity, and allow application component to customize and access their contextual view (see for instance [15]).

Clearly, the tension between the two solutions, i.e., explicit binding among components or use of coordination media, is ultimately resolved only in the application context, where the choice of either style is tied to a specific functionality at hand. Nevertheless, the challenge is to devise programming abstractions, as well as methods for designing and reasoning about applications that naturally support and integrate both styles.

The radical changes due to decentralization and continuous evolution not only characterizes software products (i.e., applications), but also the business models and the software processes that drive software developments [6]. For several classes of mainstream applications, we are moving from a situation where application development is mostly done from scratch, and under control of a single authority who dictates the requirements, supervises design, development, and often even deployment, to a situation where components are integrated, maybe even dynamically, to form new applications. Components are made availbale through the network by independent authorities. Thus the network is evolving from an information bazaar to a service bazaar. The bazaar metaphor indicates that computational resources are made available in a largely unstructured showcase, with no centralized control authority, where averybody can have some form of access. In this new setting, systems are built by federating services available in the bazaar, rather than building everything from scratch, and by publishing local information to the global world. What kinds of business models are possi-

ble for this setting? How do these affect the process models we need to follow to guide developments?

4 Conclusions

In the limited space of this contribution we attempted at providing an overview of the state of the art of software engineering by evidencing how old trends, like those towards dynamicity, decentralization, and decoupling, are now exacerbated by modern distributed computing. Moreover, we touched on a few key research challenges that we believe are fundamental for dealing with this kind of systems from a software engineering standpoint.

References

1. A.P. Black et al. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, 1987.
2. F.P. Brooks. No silver bullet: essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.
3. A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, August 2001.
4. G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering*, 27(9):827–850, September 2001.
5. G. Cugola and C. Ghezzi. Software Processes: A Retrospective and a Path to the Future. *Software Process Improvement and Practice*, 4(3):101–124, 1999.
6. M.A. Cusumano and D.B. Yoffie. Software Development on Internet Time. *IEEE Computer*, 32(10):61–68, October 1999.
7. W. Emmerich. *Engineering Distributed Objects*. John Wiley, 2000.
8. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
9. D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, January 1985.
10. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.
11. JavaSpaces. The JavaSpaces Specification web page. `http://www.sun.com/jini/specs/js-spec.html`.
12. Jini Web page. `http://www.sun.com/jini`.
13. B. Meyer. Design by Contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*. Prentice-Hall, 1992.
14. A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In F. Golshani, P. Dasgupta, and W. Zhao, editors, *Proc. of the 21st Int. Conf. on Distributed Computing Systems*, pages 524–533, May 2001.
15. G.P. Picco, A.L. Murphy, and G.-C. Roman. On Global Virtual Data Structures. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, pages 11–29. CRC Press, August 2002. To appear.
16. G-C. Roman, G.P. Picco, and A.L. Murphy. Software Engineering for Mobility: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 241–258. ACM Press, 2000.