

# Issues in Reengineering the Architecture of Component-Based Software

J.M. Favre<sup>+</sup>   H. Cervantes<sup>+</sup>   R. Sanlaville\*   F. Duclos\*   J. Estublier<sup>+</sup>

<sup>+</sup>Laboratoire LSR-IMAG  
220, Rue de la chimie  
Domaine Universitaire, BP53X  
38041, Grenoble Cedex 9, France

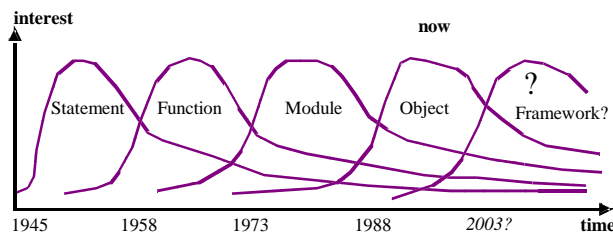
\*Dassault Systèmes  
9, quai Marcel Dassault  
92150 Suresnes  
France

## Abstract

"Architecture", then "component", became buzzwords in the last decade. The precise meanings of these terms have been evolving over time, and vary among different research communities. Traditionally the reengineering community has focused on recovering the architecture of *unstructured* or *modular* software. Recently, significant amount of work has been dedicated to the integration of the reengineering and *object-oriented* worlds. In this paper we claim that the next step could be the integration of reengineering and *component-based* development. Our collaboration with Dassault Systèmes, the world leader in the CAD/CAM market, shows that time has come to investigate this issue.

## 1. Evolution of forward engineering techniques

Structuring large software products has always been difficult and this will remain an open issue for a long time. Which is the good paradigm to organize huge amounts of software entities? Looking backwards in the history of computer science reveals that there is no definitive answer to this question. According to Raccoon's wave model [1], various paradigms have been successively judged as the "good one".



The stream of interest in structuring paradigms [1]

The stream depicted above represent the trends in *forward engineering*. However, there are at least two problems when considering this evolution in the context of large software products: (1) it takes times for large teams to learn (and really understand) a new paradigm, and (2) the paradigm used to structure a long-living software product will invariably become out of date since better techniques will appear. As a result, large amount of "old-fashioned" (yet valuable) code accumulates in large software companies. To deal with this problem, a wide range of concepts and techniques has been studied: *reverse engineering*, *restructuring*, *reengineering*, etc. In the context of this paper, RE will make reference to these techniques and to the corresponding research community.

## 2. Evolution of RE techniques

Indeed, we feel that the wave model could also be used to interpret and forecast the evolution of the RE field. This evolution is linked with the evolution of forward engineering.

- **RE and old technologies.** Traditionally the RE community focuses on *legacy software* based on *old technologies*, providing for instance modernization techniques. These techniques typically aim to pull software from an "old" wave to the current one. Since RE techniques came late in the history of computer science, it is not surprising that most of the work is dedicated to recover the assets of software created during the first waves.
- **RE and current technologies.** More recently it has been shown that RE techniques are also useful in the context of *modern software* based on the *current technology*. For instance, projects like Famoose and Spool investigate the role of RE techniques in the context of object-oriented (OO) software. Integrating smoothly forward engineering and reverse engineering leads to the concept

of round-trip engineering. Work on refactoring also shows that RE techniques can help in the continuous evolution of software.

- **RE and emerging technologies.** We believe that RE techniques can make sense in the context of the so-called *emerging technologies* (e.g. component technology). Even though these techniques are still in their infancy, pioneers already used them to implement successful software. First, *RE techniques may provide a migration path towards these emerging technologies*, easing their diffusion. Second, *RE techniques may be used to ease the maturation and evolution of newborn software products*. This last point is especially important since emerging technologies are immature by essence: they are defined incrementally and are complex to understand. In the absence of established "best practices", it is unlikely that pioneers will avoid all potential pitfalls. When large companies play the role of pioneers with a successful technology, large amount of code naturally accumulate.

In this paper, we claim that the emerging component-based technology leads to new RE issues.

### 3. Did you say "architecture" ?

We believe that the *architecture recovery* topic cannot be discussed without clearly stating, for a given technique, the paradigm used to structure the source and the target of the transformation. In other words, when presenting a technique, we have first to answer two questions:

- (1) How the software has been structured initially?
- (2) What kind of structure are we looking for?

The answer to these questions can be expressed in terms of the wave model. For instance, a technique aiming to recover ADTs in a legacy Fortran software is a transformation from the "function" wave to the "object" wave; improving the modularity of a modular system is a transformation within the "module" wave.

In fact, we feel that using the term "architecture" without further precision could be very misleading since the meaning of this term has evolved over time and became a buzzword. Similarly, the term "component" is also getting an increasing importance in software engineering, in part because there is no clear definition about what is exactly a component. Ultimately, what does "architecture" and "component" are supposed to refer to in practice? Which paradigm will constitute the next wave? *The answers to these questions not only evolve over time, they also differ from one research community to the other.* Ideally, we believe that a workshop on the recovery and modelling of software architecture should gather together people from at least the following communities (fortunately there is no clear separation, but these community tend to concentrate around a given set of workshops and conferences):

- **OO community.** Object-orientation has received a considerable attention in the last decades. While the OO paradigm represents the current wave, it is now recognized that structuring large OO software is an open issue requiring concepts beyond the notion of objects and classes. In this domain, the architecture of the systems is often expressed in terms of *patterns* and *frameworks*. As said before, several projects such as FAMOOSE are dedicated to study the intersection between RE and OO.
- **ADL community.** Thanks to foundational contribution by Garlan and Shaw, the notion of software architecture has been clearly identified in the 90's. This seminal work has led to a significant amount of academic research. In particular many *Architecture Definition Languages* (ADL) have been proposed in the literature (e.g. Acme, Wright, Rapide, Unicon, C2, etc.) [14]. The main contribution of this trend is to clearly identify the concepts of *architecture*, *component*, *port*, *connector*, *constraint*, *behaviour*, and *styles*, etc. While this approach provides an excellent conceptual framework, until now ADLs have had almost no impact on the software practice. Most companies do not feel that the formal description of the software architecture as a separate artefact can be cost effective. Moreover, the very strong emphasis put by ADLs on explicit connectors and behaviour specification is not recognized as a priority in industry, except maybe in some specific domains such as real-time systems.
- **RE community.** Recovering the structure of large software products has been for a long time one of the goals of the RE community, and this long before the work led on software architecture. For instance the CIA project at AT&T took its root in the 80's. While considerable progress has been achieved, the overall principle remains roughly the same. Relationships between source code entities are extracted and represented as a graph. Then various analyses and transformations are performed to identify significant architectural entities (see for example Rigi, PBS, Dali and Bauhaus). Interestingly the target paradigm has evolved according to the wave model presented above: the first techniques were aimed at identifying modules (the "module" wave in [1]); currently many techniques look for objects and ADTs (the "object" wave). To avoid misleading

interpretations, emphasis should be put on the fact that while the terms "architecture recovery" and "component recovery" are often used in the RE community, the range of concepts covered by these terms is much more restrictive than in the other communities. For instance, currently most RE environments recover "only" some aspects of the *structural part* of the architecture, and do not attempt to extract the behaviour specification as described by ADLs. Similarly, these environments only consider simple connectors (typically procedure calls), though some work has been led to localize other elementary connectors such as remote procedure call, process creation, etc.

- **CBSE community.** While architecture is central to ADLs, components are central to component-based software engineering (CBSE). Producing software from the assembly of existing components has been a goal for a long time. However, though promising this idea was not put into practice at a large scale until the proposition, in the last few years, of industrial-strength *component models* (e.g. Microsoft' (D)COM, Sun' JavaBeans and Enterprise Java Bean (EJB), OMG' Corba Component Model (CCM), Dassault Systèmes' Object Modeler (OM), OSGi, etc.). While these component models differ in their scope, they all share a common set of features as shown in [21]. Some indicators suggest that component-based development could be the next wave in the stream of structuring paradigms (e.g. there is an increasing number of techniques, standards, books, and conferences on this topic). The recent announcement from Microsoft to center future versions of its hegemonic operating system on the .NET component framework also goes in this direction. In particular, C# is sometimes announced as "the first component programming language", the successor of object-oriented programming. Besides commercial announcement effects, it is now highly probable that component-based technology will stay and will have a strong impact on software practice.

All approaches presented above deal in some way with the concept of software architecture, in the broad sense of the term. We feel that these different facets are significant in a discussion on architecture recovery. The evolution of the OO paradigm could not be ignored because this trend is widely supported by the software industry; the concepts of pattern and framework already have an important impact on OO software products. The work done in ADLs is interesting since it brings a neat conceptual framework, though no significant amount of software has been developed using these concepts. CBSE is interesting both at the conceptual and technological levels. Large component-based software products already exist. Finally, we believe that the extensive experience acquired by the RE community in supporting the evolution of large software could be transposed in the context of emerging technologies. In the remainder of this paper, intersection between CBSE and RE is discussed. Then our experience in this domain is shortly described.

#### 4. Component-based technology

There are many trends in CBSE, including basic research aiming to formalize the concept of components, research on suitable development process and dealing organizational issues, etc. Here, we focus on *component technology* for its direct impact on software industry.

As shown in the previous section, in the last few years various *component models* have been proposed (e.g. COM, JavaBeans, EJB, CCM, .NET). Each component model defines its own set of concepts. This may include for instance the concept of component, interface, implementation, event source, event sink, receptacle, etc.

Current component technology is built on top of existing technology. A big problem is that while new concepts are introduced, no specific language is provided. Conceptual entities must be implemented by hand in a conventional programming language, using code patterns, naming conventions, etc. For instance JavaBeans and EJB are actually pieces of java code. Actually, current component technology is difficult to grasp because there is no clear distinction between concepts and their implementations. Learning is often achieved through the reading of obscure specification documents or huge books giving programmers a long list of recipes to implement and connect components. Once implemented, components and component-based applications could reveal difficult to understand. A simple concept may be implemented by means of many artefacts spread in the source code. There are sometimes many realization choices to improve performance or other non-functional properties. Not all combinations are meaningful, but since most concepts are implemented by hand, nothing prevents the creation of "anti-patterns". Sometimes programmers have to write code interacting with generated code they don't really understand. In such a context novice programmers will invariably produce poor code. However, experience has shown that after a period, skilled programmers learn how to circumvent most issues. However when errors occur, they are difficult to find and repair in the absence of tools working at the conceptual level. What is more, the component model itself evolves over time: new concepts and

implementation techniques are introduced; others become obsolete but are not removed to ensure backward compatibility.

All these issues result in *software that is difficult to understand and evolve, because (1) often only implementation artefacts are available, (2) the concepts used to structure the software are not still and continue to evolve with time.*

## **5. RE and component-based software**

The description presented above must not discourage the reader interested in CBSE. Component-technology is an emerging but *successful* technology. It brings a degree of flexibility that could not be achieved by any other traditional technology. Some major companies have already developed large successful component-based software products. Many others are in the process of adopting this technology.

The problems mentioned above should be view on the contrary as challenging issues for the RE community. As discussed in section 2, we envision two main ways in which RE techniques can help in the context of an emerging technology.

### **5.1. Supporting the maturation and evolution of CB software.**

We consider here the case where a significant amount of CB software is already available. Corresponding RE transformations are internal to the CB wave. For instance, reverse engineering starts from the implementation of a CB software product and tries to recover CB concepts. The feasibility and complexity of all reverse-engineering transformations greatly depend on (1) the component model considered and (2) the information to be extracted.

Some facts are trivial to extract when introspection facilities are provided by the component infrastructure. Other facts are much more complex to extract and require RE techniques. Our experience in that domain suggests that recovering information about the component internal structure is usually feasible. Recovering connections and therefore the overall topology could be either a trivial task, if connections are externalized (e.g. CCM), or a complex task if connections are buried deeply into the code (e.g. COM, JavaBeans, EJB). Due to polymorphism and other late binding mechanisms, this cannot be achieved by source code analysis only. Often the architecture is not known until loading time. What is more, the architecture may evolve dynamically during execution. In any case, it is clear that to be useful RE techniques for CB software must be based both on static and dynamic information. Static information must be extracted information from a wide range of sources, including source code, but also configuration files, deployment descriptors, etc. The good news is that getting runtime information at the component level could be easy when the component runtime support provide facilities for monitoring component activity.

Once the problem of information extraction is solved, almost all techniques available in the RE could be reviewed and adapted to the context of CB software. Restructuring components (e.g. splitting a component) will become soon an issue since today there is no clear notion of what is a "good" component. CB metrics have to be defined and validated, etc.

Another interesting point it is that it is not only necessary to deal with the evolution of the software but also with the evolution of the "language" (the component model) used to write it. Over a period of time a large CB software may contain pieces of code written with different versions of the component model. Supporting the localization of obsolete constructions and their replacement by newer ones could be very important, especially when major releases imply incompatible behavior.

### **5.2. Supporting the migration to CB technology and integration with traditional ones.**

The transformations described above only make sense for companies that already have CB software. Others possible RE transformations start from traditional software and produce component-based entities. In this case, much of the work done on traditional software could probably be reused after some adaptation, since only the target of transformation change. For instance, a large body of work have been led in recent years to discover/recover "components" [6]. The term "component" here is used in the broad sense of the term, but these units of functionality could be wrapped into components as defined by component models. The good news is that some component technology like CCM provide facilities to wrap legacy code. For instance a CCM component could be decomposed in many different "segments" and "executors" than can be implemented in legacy languages. Actually, it is not clear if the component identified in this way will constitute "good" component from a component-based point of view. In any case, extra effort will be need to exploit the full potential of component technology. On the first hand, component model provide richer communication protocols, for instance event-based communication. On the other hand, component infrastructures like those provide by EJB and CCM provide services such as transaction management, persistence, etc.

## 6. Experience and work in progress

In this paper we claim that the integration between RE and CBSE is an emerging issue. This claim is supported by our experience in this domain. To be more precise we designed RE techniques to support the evolution of a large component-based software and thus explore the issues described in section 5.1. In fact, we discovered the importance of this topic in the context of a collaboration between an academic institution, the LSR laboratory, and one of the largest software companies in Europe, Dassault Systèmes (DS).

Dassault Systèmes is the world leader on CAD/CAM with more than 19 000 clients and 180 000 seats. Its main software product, CATIA, alone, has about 5 millions lines of code. What makes CATIA a very interesting case study is, on the one hand, because in DS alone, 1000 engineers are developing that software with a commercial release every 4 months, and on the other hand because many CATIA customers around the world are also developing large amount of code for extending and adapting CATIA to their needs.

DS is one of the pioneers in component-based technology. In the mid 90's DS started to develop a proprietary component model, namely the OM component model. Since then this component model has been successfully used for years in the development of CATIA. The current version of this software is made of more than 50000 C++ classes and 8000 OM components!

The development and evolution of such large component-based software naturally raises a number of issues as described in section 4. In particular what is missing is a clear picture of the software at a conceptual level. It thus appeared that a reverse engineering approach could be useful to extract information from available artifacts. Before trying to develop reverse engineering tools, the first step was to give a rigorous definition of the component model. We spent a lot of time studying the OM component model and deciding, for each feature provided, whether it should appear at the conceptual level or not. The OM component model was formalized as a meta model [15] expressed in UML and OCL, following the approach we took in [19] to describe the meta model of JavaBeans. From the meta-model, we derived a platform based on a common repository. Different tools have been implemented including exploration tools and analysis tools [15][16].

Actually, structuring software in terms of OM components represents only one view of application architecture. To organize the huge set of software entities DS uses many other kinds of architectures [16] including for instance the "physical architecture" expressed in terms of "modules" and "frameworks", and the "packaging architecture" expressed in terms of "products" and "configurations". Our goal is to provide a complete architectural environment [17][20][22] targeted to DS' needs.

Since many different architectures and concepts are used within this company, developing a specific tool for each architectural view is difficult. After an evaluation of existing tools such as Rigi [18], we choose to define a Generic Software Exploration Environment called  $G^{SEE}$ . The genericity of this environment enables to explore arbitrary software structures.  $G^{SEE}$  itself is based on a component approach: new source components (extractors) and visualization components can be connected interactively [24]. We are currently extending  $G^{SEE}$  to support the exploration of multiple versions of an arbitrary software structure [25]. In particular, this will enable to study the evolution of the CATIA software. We are also working on the definition of a component model, called Beanome, with a specific application domain in mind: the construction of CBSE environments, including forward and reverse engineering tools.

## 7. Conclusion

The terms architecture and components are useful as rallying terms, but their exact meanings evolve over time and depend on the field of research considered. Bringing people from different communities is obviously a good way to build a common understanding, but to avoid misunderstanding each discussion should start with a description of what does these terms means. Our experience, both in academic and industrial contexts, shows us that they are plenty of views on software architecture, so one should not be dogmatic about this topic.

While the reengineering community has mainly focused on traditional technologies, the extensive use of emerging technologies also raises a number of interesting issues. We believe that studying the intersection between reengineering and component-based software engineering is a challenging but promising direction. On the first hand many reengineering techniques could be adapted to the reengineering of component-based software. This is the topic of this paper. On the second hand, the flexibility of the component-based approach could be used to build component-based reengineering tools. But this is another story.

## 8. References

- [1] L.B.S. Raccoon, "Fifty Years of Progress in Software Engineering", Software Engineering Notes, Vol.22, No 1, ACM SigSoft, Jan. 1997
- [2] RIGI, <http://www.rigi.csc.uvic.ca/>
- [3] PBS, <http://www.turing.toronto.edu/>
- [4] FAMOOSE, <http://www.iam.unibe.ch/~famoos/>
- [5] SPOOL, <http://www.iro.umontreal.ca/labs/gelo/spool/>
- [6] R. Koschke, "Atomic Architectural Component Recovery for Understanding and Evolution", Phd. Thesis, University of Stuttgart, 2000
- [7] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice", Addison Wesley, 1998
- [8] OMG, CORBA Components: Joint Revised Submission. Object Management Group, 1999.
- [9] Sun Microsystems. Java Beans 1.01 Specification, 1997 , <http://java.sun.com/beans>
- [10] D. Box, "Essential COM", ISBN 0201634465, Addison-Wesley, Jan. 1998
- [11] OSGI Service Gateway Specification, Release 1.0, May 2000. <http://www.OSGI.org>
- [12] Pfister, C., Szyperski, C., "Why Objects Are Not Enough", Proc. of the 1st International Component Users Conference (CUC'96), Munich, Germany, July 1996
- [13] F. Bachman et al, "Volume II: Technical Concepts of Component-Based Software Engineering", Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon University, May 2000
- [14] N. Medvidovic, R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages". IEEE Transaction on Software Engineering, Vol. 26, No. 1, January 2000.
- [15] J.M. Favre, F. Duclos, J. Estublier, R. Sanlaville, J.J. Auffret, "Reverse Engineering a Large Component-based Software Product", CSMR'2001
- [16] R. Sanlaville, J.M. Favre, Y. Ledru, "Helping Various Stakeholders to Understand a Very Large Software Product", European Conference on Component-Based Software Engineering (ECBSE'2001), September 2001
- [17] H. Cervantes, "Dependency Analysis and Slicing in the context of a large (component-based) software", Rapport de DEA, in french, University Grenoble I, june 2000
- [18] S.T. NGuyen, J.M. Favre, Y. Ledru, J. Estublier, "Software Exploration Environments", ICSEA'2000, in french, dec. 2000
- [19] V. Marangozova, "Linking software architecture and source code", Rapport de DEA, in french, University Grenoble I, 1999
- [20] Y. Ledru, R. Sanlaville, J. Estublier, "Defining an Architecture Description Language for Dassault Systèmes", International Software Architecture Workshop (ISAW4), June 2000
- [21] J. Estublier, J.M. Favre, "Component Models and Component Technology", Chapter in *Builiding Reliable Component-Based Systems*, I. Crnkovic, M. Larsson editors, Archtech House publishers, to appear, 2002.
- [22] R. Sanlaville, "An Architectural Environment for Dassault Systèmes", Ph.D. Thesis, in french, University of Grenoble, in preparation, dec. 2001.
- [23] J. Estublier, J.M. Favre, R. Sanlaville, "An Industrial Experience with Dassault Systèmes Component Model", Chapter in *Builiding Reliable Component-Based Systems*, I. Crnkovic, M. Larsson editors, Archtech House publishers, to appear, 2002.
- [24] J.M. Favre, "G<sup>SEE</sup>: a Generic Software Exploration Environment", International Workshop on Program Comprehension (IWPC'2001), <http://www-adele.imag.fr/~jmfavre/GSEE>
- [25] S. Jamal, "A Framework to Analyze Software Evolution", Rapport de DEA, in french, University of Grenoble, Juin 2001