

Continuous Discovery of Software Architecture in a Large Evolving Company

Jean-Marie Favre Remy Sanlaville

*Adele Team, Laboratoire LSR-IMAG
University of Grenoble, France
<http://www-adele.imag.fr/> (~jmfavre | ~sanlavil)*

Abstract

There is a wide agreement that software architecture is a useful notion. Unfortunately there is no agreement on what does this term means exactly. Despite of that, the recovery of the architecture is certainly necessary since (1) source code does not provide the good level of abstraction and (2) software architecture is almost never described explicitly in large software companies. In this paper we claim that : (A) evolving software companies using evolving technologies continuously need to discover what is architecture about, and this in order to control the development of their software products ; (B) the notion of software architecture is actually a moving target. Supporting the discovery/recovery of software architecture therefore implies the use of very flexible tools. These claims are based on our experience on discovering/recovering the software architecture of CATIA, a multi million LOC software product line developed over the last decades by Dassault Systèmes, one of the major software editors in Europe.

1. Introduction

In the last decades, a large amount of work has been dedicated to academic research on software architecture. One the major achievements in this domain is the formalisation of architectural concepts and the provision of Architecture Description Languages (ADLs). ADLs make it possible to describe software at a high level of abstraction providing a logical view on software.

Until now, however ADLs have not had significant impact on existing software products. One important issue is how to maintain the link between the architectural description and the actual implementation in a cost effective way. Moreover, the abstractions that could be described using ADLs are not necessarily the ones that are needed by the different stakeholders in large software companies.

This does not mean that the notion of software architecture is useless in the context of software industry! On the contrary, everybody agree that views on software architecture are required but this notion is still quite fuzzy.

Significant advances in architecture recovery have been achieved in the context of legacy software based on traditional technologies. Currently most research in reverse engineering focuses on procedural code. In this case the software architecture is often expressed in terms of "systems" and "subsystems" that are abstractions based on call graphs and data flow information.

This logical view is often complemented by a physical view dealing with entities such as source or object files, files hierarchies and dependency relationships between these entities. In the case of software based on traditional techniques, the term architecture recovery is well suited: the kind of information to be recovered is more or less well understood; the goal is to get a set of pictures on some structures that have been constructed more or less intentionally using well understood techniques.

Studying the architecture of legacy software might give the false impression that the notion of software architecture is well understood. This is not the case. The techniques used to develop modern software are in constant evolution and our view on software architecture should also evolve accordingly. What about for instance the architecture of a distributed software based on a component-based approach and using extensively WWW techniques? What if this software is developed concurrently by large teams spread in different companies? Will the traditional views on software architecture fit the needs of these companies? Are the abstractions provided by ADLs or current reverse engineering tools the good ones to control the evolution of such systems?

Actually we believe that large software companies first need to discover what software architecture is all about in their context; then, and only then, they will be able to recover useful architectural views.

Staying on the cutting-edge of technology leads to a continuous improvement of the techniques used. Managing such a software requires a continuous discovery of new architectural abstractions. This at least what suggests our experience in the context of Dassault Systèmes, a company that has evolved a huge software product over the last decades.

2. Continuous evolution in a large company

In the 70's, the French company Dassault developed an in-house CAD/CAM software called CATIA. The success of this product led to the creation of Dassault Système (DS). The scope of this software have considerably evolved: starting in avionics, it has addressed a ever widening range of application domains. Currently CATIA supports the design and manufacture of automobiles, plants, ships, consumer goods, etc. Continuous improvements have been made to take advantages of the evolution of available technologies. While first versions required powerful computers to run specific single-user applications, CATIA now supports the complete processes of client companies including the collaborative work of geographically distributed teams working on a wide range of hardware and operating systems.

All these changes have had a considerable impact on the way the software has been developed. The first version was developed internally by a very small team. This has changed. CATIA is now developed by more than 1000 software engineers working concurrently in the DS company, but also, and this is a very important aspect, in collaboration with many other partners. This model allows CATIA to bring effective solutions in many specific applications domains: for each domain, one or more partners include their know-how and expertise by extending and adapting CATIA. Today third parties delivering "plugins" has become a common business model, but the approach of DS is much more powerful since it enables the construction of specific applications by partners. For instance, Boeing alone is said to have developed more lines for CATIA adaptation and extension than DS for CATIA itself.

Indeed, along the years, CATIA slowly evolves from a monolithic product used internally to a very sophisticated product line used all around the world (CATIA has more than 180 000 seats today) and developed concurrently by a complex net of partnerships crossing the boundaries of companies.

3. Continuous discovery of useful structuring techniques

The evolution of CATIA along the decades has led to a considerable evolution of the techniques used for its development.

While the first versions were in Fortran, in the early 90's C++ appeared as the language of choice for new developments. Most parts of the software were rewritten using this language. Early experiments revealed however that C++ alone did not fulfil DS' requirements of extensibility and scalability (CATIA V5 is made of more than 50 000 C++ classes; a full recompilation takes a week end, etc).

To cope with these problems it was decided to take a component-based approach. A in-house component model, called OM, was designed and developed internally. This component model inspired by Microsoft' COM and Corba includes many innovative features specific to DS. Just as other component technologies such as JavaBeans, COM, CCM, EJB, and the like, the OM component model is defined as a layer on top of an existing programming language (C++ in this case).

In practice, this means that to avoid any change in the programming language, the concepts of the model are defined in terms of programming patterns, naming conventions, files with specific formats, and so on. Just like the others industrial component technologies, the OM has evolved over time (and it still evolves, some other layers being added on top of it). During this evolution, some implementation techniques revealed to lead to performance issues when applied at large; some concepts revealed to be too complex to be used, some others were added to cope with unexpected problems.

In fact, currently the evolution of a component models in industry could be compared with the evolution of a programming language, excepted that :

- the concept introduced in component models are still very unclear;
- there is a serious lack of abstractions and software engineers deal only with the implementation of the concepts, not with the concepts;
- large amounts of software based on the component model are developed while this model is evolving itself. In fact, this is just like if, twenty years ago, you were developing an huge object-oriented software in assembler, trying at the same time to define what are the object oriented concepts as well as a set of corresponding implementation techniques. In such case the software has to be developed using a moving set of sometimes unclear rules.

Anyway, the experience of DS shows that this kind of evolution in a moving context is not only necessary, but it can also be successful if great care is taken. The discovery of what are the good techniques to structure a large software is a slow but continuous process.

In fact this kind of evolution is not restricted to the logical notion of components as described for example in ADLs. Any large company must deal with all aspects of the production of the software, all at the same time.

At this scale every aspect of the development could soon or later require a specific attention, for conceptual or practical reasons. For instance, number of DLLs produced in DS ever increases (thousands of DLLs by now); this is also true for the number of files managed by the configuration manager (millions of files), the number of products in DS' portfolio, the number of developers collaborating, the number of test cases, and so on. These increases unveil new issues and reveal the importance of dealing with new structuring mechanisms. Discovering new techniques to structure the software from different points of view is a slow but continuous process.

4. Continuous discovery of useful architectural abstractions

After a fruitful collaboration in the domain of configuration management, the ADELE team started in the late 90's a new collaboration with DS on the software architecture theme. The goal was to study how recent advances in software architecture research could enhance the development of DS' software. Rapidly we came with the following conclusions:

- Academic research results, in particular those provided by ADLs, did not correspond to the preoccupations of DS and these techniques seemed anyway inapplicable at this scale.
 - There was no point in describing the architecture of the software if this description were not strongly linked with the code, and since the code is one of the strongest assets of the company it should drive the dance. In other words, abstractions are considered useful as long as they enable to communicate, but they should never constrain unnecessarily people that collectively know how to solve problems when they occur.
 - While most software engineers deal mostly with the code they are responsible for, many persons in the company had their own vision on software architecture and all these visions seemed to be very pertinent according to their respective concerns.
 - All these views on software architecture seemed to bring their own concepts, even if a lot of them were overlapping in some way.
- It was very difficult to grasp the essence of the architecture at DS because the comprehension was very often hindered by a lot of technical details and historical events linked either with the evolution of the company (section 2) or the evolution of the techniques used (section 3).
 - Finally it was clear that any enhancement in the development has to be based on the extensive experience of DS in building very large software products; while some concepts were ill-defined, they had been introduced to cope with actual problems.

Instead of providing new concepts that will not fit in DS' development process, we decided to discover what could be the useful architectural abstractions, starting with the study of the OM component model.

From a very concrete point of view, we spend various months in trying to separate potential concepts from their implementations. This was done by studying large amount of technical documentation. Many interviews were also necessary to learn about non documented improvements, to recover rationales, etc. At this point we also found very useful to extract information from the software itself (as described in section 6) to see how the concepts were actually used in practice. This brings out a lot of unexpected facts: some notions had almost never been used though they took a very large place in the documentation; other techniques were used in unexpected ways, some others were used only in some part of the software, etc.

This study resulted in a much better comprehension of the OM and in a formalisation of the concepts in the form of a meta-model described with UML and OCL constraints (roughly speaking a meta model corresponds to what is called schema in the reverse engineering community).

In parallel with this formalisation process, we discovered that the OM component model represented only a very specific view on the architecture. Further investigations unveiled the existence of different kinds of architectures that could be classified as following (a comparison with [2] [3] and can be found in [6]):

- **Logical architecture.** The goal here is to split the functionality of the software in logical units. The OM component model introduces different types of entities (e.g. "components", "base implementations", "extensions", etc). All these entities being linked by different types of relations (e.g. "inheritance", "implements", etc). On top of the OM different other models had been later added, including the "feature modeller" based on notions such as "features", "specifications", or "results". At this point it is very important to stress that all these concepts are abstractions at the architectural level. Their implementation could be very complex. For instance some components gather together many C++ classes

linked by quite complex relationships.

- **Physical architecture.** For many different reasons the software has also to be split in physical units. From an implementation point of view the physical entities includes typically DLLs, but at the architectural level physical types of entities include for instance "modules" and "frameworks" connected by different types of dependencies.
- **Collaborative architecture.** To support simultaneous modifications by thousands of developers DS spent many years in designing and implementing a suitable configuration process. This had a very strong impact on how the software product is produced and structured resulting in an architecture based on entity types like "workspaces", "groups", "repositories" or "teams".
- **Packaging architecture.** DS portfolio of applications become so large that it also has to be structured. This architecture, also referred as the "licensing architecture", defines architectural entities such as "medias", "products", "solutions" or "configurations", these entities being linked by relations like "authorize".

It is very important to stress that these different architectures are linked by complex relations even if they are more or less orthogonal. For instance a "component" can gather together classes coming from different "frameworks" produced by different "teams", etc.

We found meta models very useful to document all this (meta) information about architecture. We believe nevertheless that we don't have covered yet the whole spectrum of techniques used in DS. In fact, since structuring techniques evolve over time (see section 3), discovering what are the useful architectural abstractions should also be considered as a slow but continuous process.

5. Continuous discovery of useful architectural view points

Discovering the right set of concepts is one thing, but it is far from enough in practice. As shown above this process may end with many types of architectural entities and relations. There is absolutely no point on showing a big-picture trying to convey all types of information at the same time. On the contrary, people interested in the architecture of the software only require a subset of the information accordingly to their jobs and to their goals. In the ANSI/IEEE 1741 standard on software architecture [8], this concept is referred as a "view point" and should be linked to a "concern" of a particular "stakeholder".

At DS, stakeholders include for instance "product managers", "release managers" and "packaging managers".

In our approach, a view point can be described by a subset of the meta model described in the previous section, indicating which types of entities and relations are involved in that particular view point. Many information can be crossed, but not all combinations are really pertinent.

Discovering what are the useful view points is far from easy, and obviously this require the help of the stakeholders themselves. Often most of them know perfectly how to do their jobs, but they do not have necessarily a clear idea of what view points are required. In particular while some view points focus on the activity of one stakeholder, the ones that cross these boundaries can potentially greatly improve the communication between them. Since the useful view points largely depend on the development process and the structure of the product, the discovery of useful architectural view points is a slow but continuous process.

6. Continuous discovery and recovery of architectural views

A view point is just like a query. This concept should not be confused with a "view", that is the result of the query on a particular set of instances, in this case on a particular piece of software.

To produce an architectural view, information have first to be extracted from the software; then, a suitable representation have to be chosen to visualize the result. In this sense, this is a typical reverse engineering process. However, while a lot of experience has been gained in generating traditional views such as call graphs, this is not the case for architectural view points like those described above : the views to be produced involve new types of entities that are often specific to the company. Moreover, building extractors is not necessarily an easy task. While some information might be relatively easy to retrieve from different kind of files or repositories (like the ones used by the configuration manager), other information require the code to be parsed in order to detected the use of implementation patterns.

Anyway, to show the benefits of using architectural views in the context of DS, we focus our attention on the OM component model, and we developed a specific reverse engineering tool called the OMVT. More than 20 different view points were defined, mostly centred around the notion of component.

Thanks to this tool software engineers discovered for the first time how a component could look like. Remember that they work at the code level and that a single component is often a complex aggregation of C++ classes developed by different teams. Many software engineers were surprised when they saw the shape and size of the components they had collectively built, in particular because of the use of "inheritance" relationships (at the level of component, not at

the level of C++ classes). The feedback on the OMVT was very positive. Interviews with different stakeholders uncovered interest for many different extensions corresponding to complementary view points. This in turn raises the issue of the cost of implementing new view points. It became clear that given the number of types of entities in the meta model, the number of potentially interesting view points would be really important.

Since new architectural concepts remained probably to discover, a very flexible approach had to be taken. So we started to develop G^{GSEE}, a Generic Software Exploration Environment [7] based on the use of meta models. This environment can generate new views on demand and acts as a prototyping tool to discover new architectural view points. If a view point proves useful, a specific tool like the OMVT can then be implemented to help software engineers recover architectural when they need it. GSEE aims to support agile exploration of the architecture while specific like tools like OMVT aim to support routine inspections. We believe that both kind of tools are needed to support the discovery and recovery of software architecture [7].

7. Conclusion

While the observations made in this paper come from our experience in the context of Dassault Systèmes, they may also apply to other large companies. The notion of architecture appears to be a moving target.

Large companies evolve; their software evolve; the ways to structure them evolve and will continue to evolve. As a result each company has continuously to make evolve its catalogue of architectural abstractions and view points.

In this paper a general process to build such catalogue is presented: the set of architectural concepts are first identified and described as a meta model, then the set of view points are established as subsets of this meta model and finally tools to extract views are build. \$Actually, this is an iterative process. To evaluate the usefulness of an abstraction, one has to generate a particular view and discuss with the concerned stakeholders.

Discovering what the term "architecture" means in a particular company is an exploratory process that should be supported by very flexible tools.

We expect software architecture to remain a research theme for a long time. A lot has to be learn from industry since building very large software shows the actual importance of issues that are often neglected in academy.

8. References

- [1] R. Koschke, "Atomic Architectural Component Recovery for Understanding and Evolution", Phd. Thesis, University of Stuttgart, 2000
- [2] P. Kruchten, "The 4+1 View Model of Architecture", IEEE Software, 12(6), 42-50, 1995
- [3] C. Homeister, R. Nord, P. Soni, "Applied Software Architecture", Addison Wesley, 2000
- [4] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice", Addison Wesley, 1998
- [5] 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.
- [6] R. Sanlaville, "An Architectural Environment for Dassault Systèmes", Ph.D. Thesis, in French, Univ. of Grenoble, dec. 2002.
- [7] J.M. Favre, "A New Approach to Software Exploration: Backpacking with GSEE ", CSMR'2002
- [8] IEEE, "Recommended Practice for Architectural Description of Software-Intensive Systems", IEEE Std 1741-2000.
- [9] O'Brien, C. Stoermer, C. Verhoef, "Software Architecture Reconstruction: Practice Needs and Current Approaches", Technical Report CMU/SEI-2002-TR-024, 200