Evolution of a Software Component – Experiences with a Network Editor Component

Jyrki Akkanen
Nokia Research Center
P.O.Box 407, FIN-00045
NOKIA GROUP, Finland
jyrki.akkanen@nokia.com
+358 7180 36649 / +358 7180 36229

Attila J. Kiss
Nokia Research Center
Hungary, 1092, Budapest,
Köztelek utca 6, C-528
attila.kiss@nokia.com
+36 20 9849 324 / +36 20 963 8405

Jukka K. Nurminen
Nokia Research Center
P.O.Box 407, FIN-00045
NOKIA GROUP, Finland
jukka.k.nurminen@nokia.com
+358 7180 36442 / +358 7180 36229

ABSTRACT

Even though the benefits of component-based software development are widely accepted, they are easily overestimated. To provide a firmer basis for the general discussion we describe our real life experiences with a software component. Having a lifetime of a whole decade the component has evolved from a class library to an independent component. In this paper we focus on the major evolution steps, their rational, and their outcomes, hoping that this gives some relevant insight to the issues that are important for software component evolution and maintenance. Surprisingly often the lessons learned have little to do with the hot topics of software technology that are being marketed. We discuss the risks attached to component selection, the usage of a shared platform for a product family, and the strengths and weaknesses of application frameworks and components. We also comment practical issues in designing and implementing major architectural changes.

1. INTRODUCTION

The component-based software development has been a fashionable buzzword in the software community. The advocates of the new component technologies often cite a long list of benefits in development time, code reuse, ease of maintenance and other areas. To provide a firmer basis for this discussion we, in this paper, describe our experiences with using a software component in real life.

The component was initially developed around ten years ago and has gone through major evolution steps during its lifetime. We will be focusing on these steps, their rational, and their outcomes. By presenting the reality of the software component evolution and sharing our experiences we hope that we are able to provide some relevant insight to the issues that are important for software component evolution and maintenance. Surprisingly often the lessons learned have little to do with the hot topics of software technology that are being marketed.

(Voas, 1998) gives an overview of the maintenance challenges raised by component-based development suggesting the need to rethink our software maintenance strategies. (Lehman and Ramil, 2000) analyse laws of software evolution in the context of component-based software engineering. (Crnkovic and Larsson, 2000) is a practical case study somewhat similar to our paper about the use and evolution of a component-based system. The domain of that paper is industrial process control where the requirements especially for reliability are extremely high. (Favre et al., 2001) discusses some issues how component-based techniques affect software comprehension and evolution. In particular, it focuses on using a meta-model to help in software understanding. (Pighin, 2001) discusses a methodology to create a component catalog for reuse and maintenance purposes. (Wu et al., 2000) proposes a technique based on static analysis for the maintenance of component-based software.

The rest of the paper is structured in the following way. In section 2 we briefly describe the network editor component. In section 3 we cover the major evolution steps in its history. In section 4 we focus on some key issues of the evolution, on the lessons learned, and on what might be practical issues to consider. In section 5 we summarize our main conclusions.

2. THE NETWORK EDITOR

The component we are focusing on is an in-house developed network editor, which has been used in several commercial products and internal applications within Nokia. The initial goal was to create a re-usable class library, called Interactive Graph Editor (IGE), that could be used in applications requiring network visualization and interactive network editing. The library was written in C++ and targeted mainly for the PC/Windows environment although it was meant to be platform independent.

The initial, experimental versions of IGE (Toivonen, 1990) were built 1990 on Glockenspiel Common View, one of the first commercial C++ libraries for developing applications with graphical user interface in MS-Windows environment. However, for the subsequent releases of the framework for actual applications, we soon adopted another commercial library, C++/Views from CNS. Both application frameworks followed the model-view-controller paradigm copied from Smalltalk, hiding the operating system of the computer behind its own layer of functionality. The programmer did therefore not need to know how to program with Microsoft Windows but it was enough to know the C++/Views library. The added benefit of this was that the programs were (mostly) platform independent and could be ported with reasonable effort to other operating systems like OS/2 and OSF/Motif.

In 1997, the component and related framework won 3rd prize at Object World OO awards, in the category of "Best application utilizing reusable components leveraged from or in use in other projects."

Figure 1 shows some of the applications that have been developed using the network editor component. The boxes with the thick border represent commercial applications; the rest of the applications have been for Nokia internal use. The arrows represent code that was reused in subsequent framework or application.

The applications cover a range of areas: circuit design (NASSE), software engineering (KISS, TDE, Mermaid), telecom network planning (NPS/10, FixNet, Assi), service creation for intelligent networks (NSE), IP network modeling (Calipran), and wireless router network management (RMS). Some of the applications were developed from scratch using the network editor component and its related framework as a basis (NASSE, KISS, NPS/10, NSE, Calipran, and RMS). Other applications took an existing application as a starting point and modified that to reach the goal (Mermaid, TDE, FixNet, and Assi).

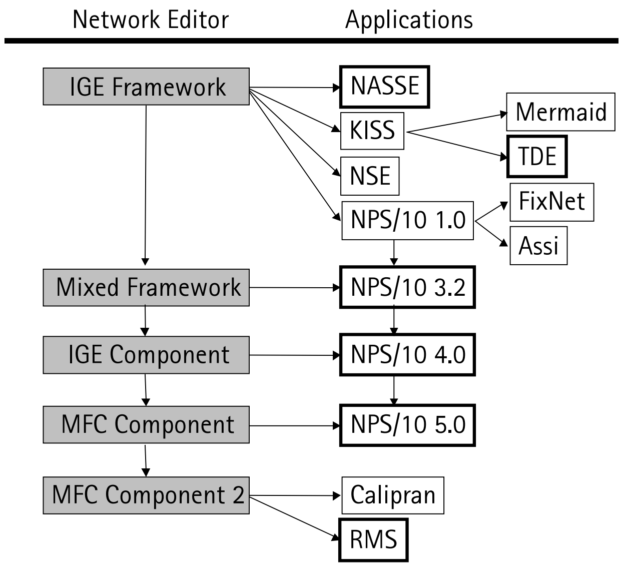
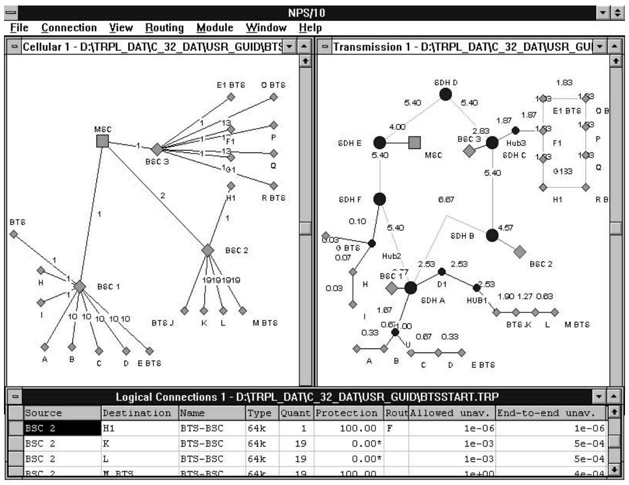


Figure 1 Applications using the network editor

Figure 1 also shows the major generations of the network editor component and associated application architecture (gray boxes on the left). In the following we will follow this development which occurred mostly along the NPS/10 application (Akkanen and Nurminen, 2001; Nokia Networks, 1999). Being one of the first applications using the network editor component its development started in 1992 and it has grown from a small drafting tool to full-scale detailed transmission network planning tool. Since 1996 it has been available as a commercial product. New functionality is still actively developed to the tool to keep up with the new demands and technologies in network planning. Figure 2 shows an example screenshot of the NPS/10 application.


NPS/10
File Connection View Routing Module Window Help
Cellular 1 - D:\TRPL_DAT\C_32_DAT\USR_GUID\BTS
Transmission 1 - D:\TRPL_DAT\C_32_DAT\USR_GUI
Logical Connections 1 - D:\TRPL_DAT\C_32_DAT\USR_GUID\BTSSTART.TRP

Source	Destination	Name	Type	Quant	Protection	Rout	Allowed unav.	End-to-end unav.
BSC 2	H1	BTS-BSC	64k	1	100.00	F	1e-06	1e-06
BSC 2	K	BTS-BSC	64k	19	0.00*		1e-03	5e-04
BSC 2	L	BTS-BSC	64k	19	0.00*		1e-03	5e-04
BSC 2	M BTS	BTS-BSC	64k	19	100.00		1e+00	4e-04

Figure 2 Sample screen of NPS/10 network planning tool

3. MAJOR DEVELOPMENT STEPS

Table I summarizes the major technical steps in the history of the network editor component. Every major modification in the network editor has also been a major modification in the software architecture.

Table I . Key development steps

Date	Network Editor	Features	Applications
12/92	IGE Framework	Windows 3.1, OSF/Motif Borland C++, C++/Views	Nasse, KISS, NSE, NPS/10 1.0
6/96	Mixed Framework v1	Major architectural renewal C++ templates introduced	NPS/10 3.2
8/97	Mixed Framework v2	Windows NT 4.0 Microsoft Visual C++	NPS/10 3.5
10/98	IGE Component	Major architectural renewal STL introduced	NPS/10 4.0
12/99	MFC Component v1	MFC, Stingray Objective Views	NPS/10 5.0
12/00	MFC Component v2	Strict encapsulation and interface changes	RMS, Calipran

3.1 IGE FRAMEWORK

In the early years of the network editor the idea of components in software had not yet fully matured and the best one could hope for was a re-usable, object-oriented application framework. In our case the IGE framework provided support for the network presentation in a data model and multiple graphical views to the data. That was appropriate and adequate for multiple applications requiring graphical presentation of networks.

We took full advantage from the cross-platform capabilities of the framework. While most applications were indeed developed in MS-Windows, there were a couple of OSF/Motif applications and even one application, which was ported to both platforms.

3.2 MIXED FRAMEWORK

Eventually, however, we noticed that the IGE framework could no longer support all the new application requirements. The data model of NPS/10 application had grown and was now more a stack of related networks rather than a single network. In addition, the user-interface design was rather rigid and did not completely satisfy our requirements. The framework, which once was a benefit, began to be a burden and restrict our work rather than help it. Another driving force for renewing the architecture was the fact that Microsoft was actively updating the GUI features in its Windows operating system and our library was gradually dropping out of the pace. We began to foresee that some day we might need to change the GUI library.

This triggered a long sequence of major software architecture improvements in the NPS/10 tool. Our first target was to employ a clean data model and to use the IGE framework only for the network views. This was not easy because IGE, being designed as a complete framework, persistently resisted our attempts to use only a part of it. After a lot of struggle we were able to create a satisfactory mixed framework, where the influence of the old framework was relatively limited.

At this point we started to introduce some more modern C++ language as well e.g. template-based collections. We knew that STL existed, but had no implementation that our compiler would have supported.

3.3 MIXED FRAMEWORK V2

Despite all the similarities in the operating systems, the need to move from the 16-bit Windows 3.1 to 32-bit NT 4.0 was not a small step. The essential difference was not at the operating system level, but the fact that it required us to update all our basic libraries (C++/Views and IGE) to more recent versions. Even though newer versions typically try to be compatible with the old, there were lots of small differences and improvements, which gave us a lot of work. Furthermore, we also had to change the compiler to Microsoft Visual C++ that gave us some extra struggle. The compilers did not very smoothly accept all the newer C++ features we were using, and what was acceptable varied from vendor to vendor.

At this point we noticed that all our active development was in the same platform, and thus we decided to give up the goal of targeting multiple platforms to take advantage of the special features of the Microsoft Windows operating system.

3.4 IGE COMPONENT

The mixed framework served us for a while even though we all the time knew it was far from ideal. We were still rather dependent on the IGE platform and, due to some short cuts, had managed to mess up our data model. We saw that even though the previous architecture fix was a major operation, it still was only a first step in a long run. At this time we considered starting to use components (component-based software had been available a couple of years already) and made some long-term plans and defined goals for the further architecture work.

This time we had matured enough to make the final step with the network editor: to encapsulate it as a component. From the point of the old IGE framework this required a little abuse: we had to create tiny "fake applications" inside the network view component to fool the framework to think that it still was a complete application and not just a single network view. But there it was: a network view behind a nice, well-defined interface.

3.5 MFC COMPONENT

We were now rather satisfied with our architecture, but more and more annoyed with our terribly obsolete GUI library. We selected a step to the mainstream: picked MFC as the basic GUI library and took Stingray Objective Views as the graphical view component. After the basic work for investigating how to replace our component with the Stingray component was done, converting the whole application (which had grown already to a size of over 400 kLoc) went rather quickly. The most demanding task was to convert all dialogs.

3.6 MFC COMPONENT V2

After using MFC and Stingray for a while we, once again, noticed that there was room for some improvements. Now the major problem was that the new component was not very strictly encapsulated. We had tried to avoid large risks in the adoption of the Stingray component by following the "sample" patterns rather than trying out something of our own. The side effect of this was that our application became dependent on the internal structure of the component. Once we had gathered enough experience and knowledge about our new libraries, the dependencies were eliminated and the component interface changed.

4. KEY ISSUES AND LESSONS LEARNED

4.1 VENDOR INSTABILITY

The selection of the C++/Views as the base GUI library was based on an evaluation of available tools on the market. At that time the offering was quite limited and the subsequent market leaders, like Microsoft MFC, became available only a few years after the selection had been made. From our point the market situation changed unfavorably, and support and maintenance of C++/Views became a problem. The ownership of C++/Views moved from its original developers in CNS to Liant and later to Intersolv, which eventually decided to close down the support of the product. Recently, Stanton Consultancy Limited in UK has taken up the support and recreated the product.

The lessons from this are twofold. First, selecting a mainstream product with a wide user group helps to guarantee a steady flow of bug fixes, new features, and other maintenance. Secondly, since this is not always possible and market situations are changing, it is useful to try to limit dependencies on 3rd party libraries in the code. This implies extra effort: intermediate layers between actual application and underlying library and/or proprietary software architectures. In the long run this can pay off when moving the code to a new platform.

4.2 IMPORTANCE OF ACCESS TO SOURCE CODE

As the support for the underlying libraries in our case was quite poor we ended up fixing many bugs in the libraries by ourselves as well as creating support for missing features, e.g. new controls, copy/paste, printing support. Access to the source code of C++/Views was essential for this.

Access to source code is important not only to fix the code but also to debug any problems. With source code available it is possible to follow with a debugger the program execution not only in the application code but also on the lower layer code. In our case the source code was also the most reliable documentation that was available.

4.3 SHARING THE CODE BETWEEN THE DIFFERENT APPLICATIONS

The obvious benefit of using a component by multiple applications is that the same code is shared and reused resulting into a product family concept. With our experience this is especially beneficial at the early stage of the development since it allows fast creation of a first working prototype which can be incrementally developed further.

Contrary to a genuine product family we ended up into a situation where each application development project used its own version of the network editor component. This situation arose mainly because there was not enough dedicated developers to handle the component maintenance. Instead, each application project created small fixes and enhancements to the component. Occasionally the changes were collected and merged to the master component, and when the schedules of the application projects permitted they took into use the resulting newer versions of the network editor. However, this resulted into divergence especially as the number of people in the application projects increased.

Probably the divergence could have been avoided or limited by having a stronger development group handling the component development and maintenance. In any case the cost of updating a framework and taking the newer framework version into use is not negligible.

4.4 FRAMEWORKS VS. COMPONENTS

As the framework forces the application to be structured to follow a certain set of patterns the risk of major architectural failures is decreased. This allows the developers to focus on the features needed by the application, get the application faster into use, and, at the same time, to learn to understand better what are the key architectural requirements.

In our case, in the longer run, the use of a rigid framework became a burden. Finally it was not possible to implement new features without changing the architecture of the framework. Even with the most successful and widely used frameworks, users tend to run into problems when they start using the framework against the original design.

Our lesson is again twofold. Developers need well-tested frameworks to quickly create applications and to avoid major architectural risks. On the other hand, they need independent components for better reuse and adaptability to changing situations.

4.5 EXECUTING MAJOR ARCHITECTURAL CHANGES

We made several major architectural improvement steps during our long software project and can notice that, to our satisfaction, all of them were successful. In our case they key to success was careful planning and limiting the scope of the actual changes to minimum.

The architectural changes were carefully designed. We always started by identifying the major troubles in our current architecture and sorting out nice solutions to practical problems. We tended to take care that we all the time have a long-term goal in which we are going so that our improvements were individual steps towards the same direction. However, after getting a clear understanding about what we want we always had to return back to reality: to see what is actually feasible concerning the schedule of our next release. In most cases the bottom line was that we could fix the most damaging troubles, but not much more. (This forced us e.g. to drop the ideas about using explicit component technologies like COM). In this sense even the major architectural changes can more be seen as steps in continuous refactoring process rather than as complete rewriting of the application.

The rest of the preparation work consisted of coding some key pieces, making migration plans, and writing directions for developers who finally executed the architecture update. Because of the careful preparation the actual architecture renewal did not shift our schedule more than a couple of months, which was tolerable, taking in account that two months is easily lost in bug-hunting of a messy application.

Designing architectural changes is not fulltime work and can't be isolated from application development. In our case the lead developer was considering the architectural issues in parallel besides his other tasks. Participating in application development provided important insight about what kind of new user requirements typically come up, what are the hard tasks for the software developers, and how these issues can be helped by proper architectural changes.

One important lesson for us, which we confronted every time we improved our architecture, was that, however well the architecture has been designed, something always goes wrong. After the major troubles are out of the way the next troubles become visible. And sometimes the solutions are not so good after all, and show their nature the day the requirements for the next release are available. This taught us not to try to be too perfect: a working application, even with shortcomings, is better than no application. Knowing the basic refactoring techniques (Fowler, 2000) allowed us to make minor fixes in architecture on the fly.

Another lesson was not to try to achieve too much when maintaining an existing application. The customers want new features and better usability and are not very willing to invest in architectural issues: investing more than, say 5 – 10 % of the costs in architecture is not possible. The tight budget usually implies that development has to concentrate on the most important issues where it is expected to get investment back very soon.

4.6 INTERFACES CAN BE AFFECTED BY COMPONENT CHANGES

In theory a well-defined interface hides the details of a component in such a way that changing the component implementation should not affect its users. Our experience suggests that this is not completely true when a bigger change in the component implementation is needed.

When moving from IGE and C++/Views to the Stingray/MFC library we tried to hide the component change from the application by providing the same interface to the applications in both cases. Unfortunately the underlying platform with new graphical features required expanding the component functionality. The interface to the component had to be changed, which further propagated changes in the application.

Furthermore, a new component implementation is always unknown to the developers, and thus finding the correct patterns and interfaces to join it in the application is far from trivial. Our experience was that relying on the component vendor's suggestions and examples does not necessarily lead to a good result.

The lesson is that the independent development of component and embedding application is not always possible. Internal changes in the component can propagate through the interface to the application.

5. CONCLUSIONS

In this paper we have discussed the evolution of a software component and how it has gradually developed from an application framework to a proper component. The evolution has been a result of new application requirements causing changes to the software structure and consequently to the used components. This kind of process can be slow: introducing major architectural changes can take a few years to do in parallel with the normal application development.

Our main experiences are:

- Component selection is always a risk because the strengths of different technologies and vendors are constantly changing. The risk can be reduced by mainstream vendor selection, proper architectural solutions, and by having access to the source code of the component.

- Using the same platform for a product family is beneficial especially at the early development stage. Later the applications tend to branch off to different directions.

- Both frameworks and components have their pros and cons. A mixture of both would be ideal.

- A practical way to implement architectural changes is with small steps, aiming to a bigger goal in the horizon, and by learning from previous steps and from new application requirements.

- There is no perfect architecture. Accepting this and using it in project planning is important.

- Components are useful even if they do not completely fulfil their promise. Component changes frequently cause interface changes.

Our experience has shown that independent software components very naturally grow up from architectural needs, but cannot be used effectively without a supporting application framework. We have as well seen that practical needs are the best guides in design work, concerning software architectures, frameworks and components. We are convinced that, while the components are gradually losing their novelty and becoming everyday tools, there is still a lot to learn how architectures and frameworks can support their efficient use.

6. ACKNOWLEDGEMENTS

We are grateful to Nokia Networks for their support and funding.

REFERENCES

Akkanen, J., Nurminen, J. K, 2001. Case-study of evolution of routing algorithms in a network planning tool, Journal of Systems and Software (58), 181-198.

Crnkovic, I., Larsson, M., A case study: demands on component-based development, in Proceedings of the 2000 International Conference on Software Engineering

Favre, J.-M., Duclos, F., Estublier, J., Sanlaville, R., Auffret, J.-J., Reverse engineering a large component-based software product, in Fifth European Conference on Software Maintenance and Reengineering, 2001.

Fowler, M., 2000. Refactoring: improving the design of existing code, Addison-Wesley.

Lehman M.M., Ramil J.F., 2000. Software evolution in the age of component-based software engineering, IEEE Proceedings on Software 147(6), 249-255.

Nokia Networks, 1999. NPS/10, Computer software,

Pighin, M., A new methodology for component reuse and maintenance, in Fifth European Conference on Software Maintenance and Reengineering, 2001.

Toivonen, H. T. T. 1990. An Interactive Graph Editor: Network Managing and Graph Layout. Report-C-1990, Department of Computer Science, University of Helsinki.

Voas, J., 1998. Maintaining component-based systems, IEEE Software 15 (4), 22-27.

Wu, Y., Pan, D., Chen, M-H, Techniques of maintaining evolving component-based software, in International Conference on Software Maintenance, 2000