Journal of Computer Science 9 (10): 1267-1273, 2013
ISSN: 1549-3636
© 2013 Science Publications
doi:10.3844/jcssp.2013.1267.1273 Published Online 9 (10) 2013 (http://www.thescipub.com/jcs.toc)

A TOOL ANALYSIS IN ARCHITECTURAL RECONSTRUCTION

[1]Jess Nielsen and [2]Michael Lykke

[1]Department of Research and Development, Trapeze Group Europe A/S,
Søren Frichs Vej 38K, DK-8230 Åbyhøj, Denmark
[2]Bang and Olufsen A/S, Peter Bangs Vej 15, Struer, DK-7600, Denmark

Received 2013-07-18, Revised 2013-08-03; Accepted 2013-08-21

ABSTRACT

An understanding of a system’s software architecture is central to successful system modifications. In the fortunate cases, the architecture is well understood as the original software architect and lead developers are responsible for maintenance. However, often systems must be modified based upon incomplete architectural information due to staff changes and incomplete or outdated documentation. In this case, software architecture reconstruction is vital in order to re-establish overview and understanding of the software architecture. In this study, we report on a tool analysis where the goal is to clarify the correctness of a number of tools which offer such architectural reconstruction.

Keywords: Architectural Reconstruction, Reconstruction Tools

1. INTRODUCTION

This document outlines an analysis which facilitates how a number of both commercial and non-commercial tools document or reconstruct a software architecture which can be used as architectural documentation.

The primary reason for this is that the maintenance phase is well known to be costly for a successful software product. A key aspect of maintaining, enhancing and extending a software system is the ability of developers to overview, understand and analyse the software architecture of the system. However, more often than not, the software architecture is largely undocumented and only vaguely understood. A major problem with documenting software architecture is the uncertainty in predicting the exact nature of documentation necessary in the future. Often customer feature requests or new technological platforms demand documentation of a certain type and aspects which are not obvious at the onset of system development. Thus, there will always be a demand for architectural reconstruction, i.e., the ability to create a (partial) software architecture based upon the artefacts which define an existing software system: typically the source code base, old documentation and maybe interviews with the initial architects, developers and maintainers. This is a labour intensive and therefore costly process and architects should have all the necessary means at their disposal to ensure a cost-effective reconstruction process. Research efforts have thus been invested in defining tools and platforms to aid in the reconstruction process which has lead to a large number of tools. A comprehensive overview can be found in (Pollet *et al*., 2007).

2. PRELIMINARY

OpenSpeak is an open source voice-over IP application based on Speex and wxWidgets, aimed at casual gamers who like to chat while playing a game. The application runs on both a Windows and a Linux platform. The key metrics for the system can be found in **Table 1**.

The article will base its analysis on a reconstruction of an open source project OpenSpeak with the use of the principles described in (Deursen *et al*., 2004) and by observing the system's runtime behaviour.

Corresponding Author: Jess Nielsen, Department of Research and Development, Trapeze Group Europe A/S, Søren Frichs Vej 38K, DK-8230 Åbyhøj, Denmark

Table 1. Overview of openspeak

Programming Language	C++
Lines of Code	183.536
Available from	http://openspeak-project.org/
IDE	Visual Studio 2005 +
Components	Client, server

The basis for this reconstruction is to generate information which can be used to produce (or reproduce) the architectural documentation for the system.

3. TOOL ANALYSIS

The goal of the analysis is to clarify how comprehensively the tools document the software architecture in three essential viewpoints as outlined in (Bass *et al*., 2003): Module view is a static decomposition in packages, classes, modules, Component-connector view (CandC view) is a dynamic decomposition in objects, processes and communication paths. Allocation view is a physical decomposition in deployment units, computing nodes.

Below we outline the choice of tools used when trying to construct these viewpoints.

3.1. Lattix

Lattix LDM is a commercial application, produced by Lattix, Ltd. (http://www.lattix.com). The application is primarily used to analyse the static structure of the source code. The producer highlights the following key features:

- Clarifying the relationship between directories, source files, header files and idl files
- Analysing the relationship between the contents of the C/C++ source files and explores the dependencies at member level

The source code is typically structured by dividing it into directories. These directories might each represent a component of the system. An analysis of the #include-directives files will outline the dependencies among the source files. It will also outline the dependencies among the components through the source files that are placed in different directories.

The tool has identified three directories as candidates for components (**Fig. 1**), each representing a part of the system. The three components are named “client”, “lib” and “server”. The degree of dependency among the components is shown by stating a number for each relation. The number tells how many times the component has been referred to, while a missing number indicates that no relations or references exist.

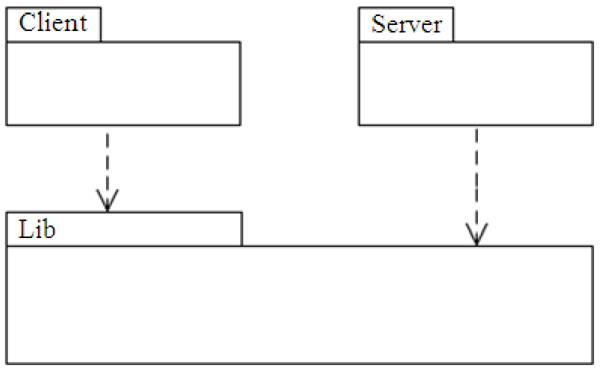

Fig. 1. The component structure identified by Lattix

It is possible to add constraints to the components to augment the visualization of the layers. The constraints tell whether the relation is legal or not. Colours indicate whether constraints are placed manually and whether they have been violated.

Constraints to clarify whether it is a strict layered model without circular references have been added. These constraints state that none of them have been violated and all communication should be done through the communication layer, denoted library.

The tool addresses especially the module viewpoint. It identifies object diagrams and classes based on e.g., the classes and data member references and it identifies packages and nodes based on the directory structures (**Table 2**).

Although the structure discovered by Lattix was the actual high-level component structure, it did not recognise the low level and complex structures (**Fig. 2**). A manual inspection revealed a complex reactor pattern (Schmidt *et al*., 2000) as depicted above.

3.2. Code Visual to Flowchart

Code Visual to Flowchart is developed by FateSoft/Share-IT. The application has the capability to analyse source code and construct a flowchart diagram based on the flows in the source code. The manufacturer highlights the following key features:

- Reverse engineering a program with code analyser; create programming flow charts from code
- Generating Visio, Word, Excel, Power Point, PNG and BMP flow charts document from code

In relation to these key features, it can be used to examine the flows in the source code. A certain sequence of e.g., invocations or a large and complex function can be visualised to help the understanding of the source code.

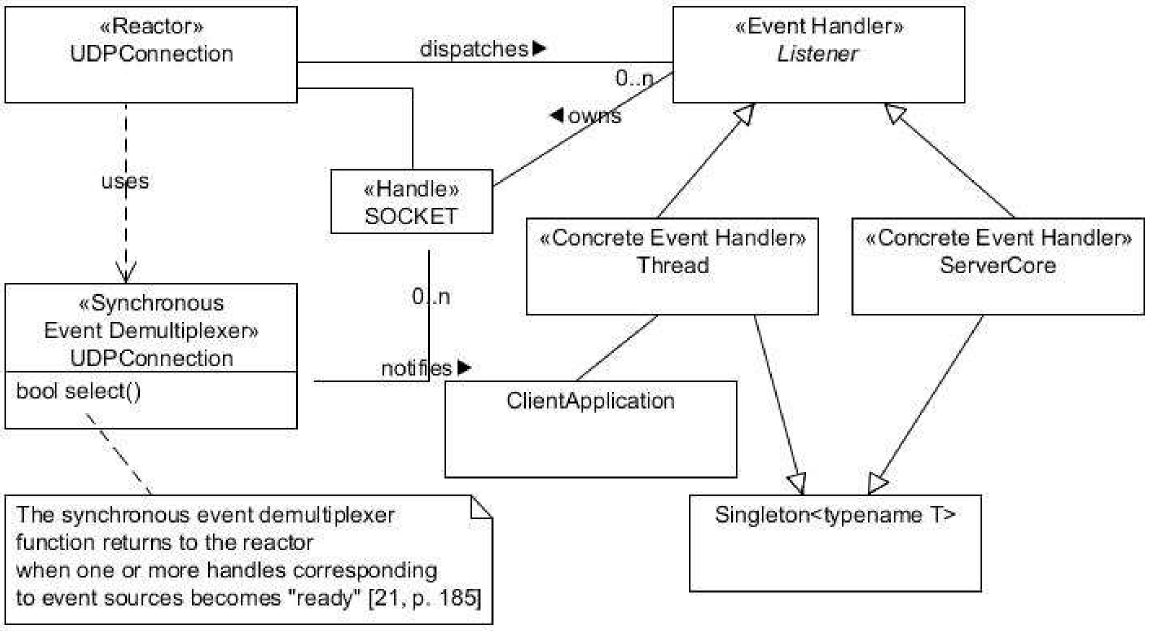

Fig. 2. The reactor pattern identified by manual inspection

Table 2. The symbolism in Lattix mapped to a proper UML notation and an appropriate view

Symbolism	Description	UML	View
Directories	Identifies the components	Packages and nodes	Module view Allocation view
Source files, #includes	Identify the relations	Objects and classes	Module view
Source files, functions, class and data members	Identify the relations	Objects and classes	Module view
Source files, grouping Allocation view	Group source files into components	Packages and nodes	Module view
Constraints	Identifies both legal and illegal communication about components.		Module view

Table 3. The symbolism in Code Visual mapped to a proper UML notation and an appropriate view

Symbolism	Description	UML	View
State charts	Describes the flow using the syntax of state charts.	Objects and stereotypes	candC view

The tool can be used to illustrate a central flow inside methods and to navigate through the code by clicking at functions. It illustrates the flow within the functions. However, the tool is not very suitable for reconstruction of software architectures because it is very low-level, while it might be useful if a sequence diagram is needed.

The usage of the tool addresses the dynamic component connector view by using a static analysis (**Table 3**). The reason is generally that the tool is not able to neither collect data from running systems nor analyse such data. The views are just illustrating a dynamic point of view-scenarios can be walked through afterwards.

3.3. Visual Studio 2008

Visual Studio 2008 Developer Edition (VS 2008) produced by Microsoft, also offers a set of analysis tools for both static and dynamic analysis. For static analysis, VS 2008 supports the following features for unmanaged C/C++ code:

Table 4. The symbolism mapped to a proper UML notation and an appropriate view

Symbolism	VS 2008	UML	View
Class, function and data member	Class view	Classes	Module view
Class, function	Call tree view	Classes and objects	C and C view
	Modules view	Packages	Module view
	Caller/Callee view	Objects	C and C view
	Functions view	Objects	C and C view

- Creation of a class diagram; however it only includes inheritance relations in the diagram, not the association and composite relations

In dynamic analysis, it is possible to perform both code instrumentation and sampling. We only used code instrumentation, but the following views are available for both:

- Call tree view where it is possible to see the call stack of the system
- Module view shows a list of the modules (executable files) used by the system
- The Caller/Callee view shows all the function calls and by whom they are called
- Function view lists all the functions called during system execution
- Memory allocation view is targeted for managed code and shows how much memory each function and all its descendent has allocated
- Object lifetime view only works with managed code and shows the total instances of each type and the amount of memory they consume
- Process view shows the processes that are executed during system execution

For the dynamic analysis, we tried to use the Caller/Callee view as the primary tool, as this is a sequence diagram containing only a single execution. Surprisingly we could not use it, mainly because the amount of information and the details was overwhelming. Secondly, the list of called functions by caller cannot be ordered by time of execution which makes it difficult to find the sequence of method calls.

Even though VS 2008 is an advanced tool, it only includes inheritance relations when creating class diagrams, as stated earlier. It is, however, possible to create relatively simple mapping rules for finding associations and composite relations see symbolism in (**Table 4**), but these rules still have to be located manually.

The class diagram from VS 2008 identifies simple composite relations. A list of all the names of the member variables is displayed along with their types when clicking on a class. However, the associations are more difficult to identify because they are identified by classes which are either created in a function or as part of the parameter list.

3.4. MOOSE

MOOSE is an open source project started at the Software Composition Group in 1975. It is a language-independent tool developed for reversing and re-engineering legacy software systems (Nierstrasz *et al*., 2005). It is a framework for software development based on formal models and code generation principles (MOOSE, 2008).

One of the nice-to-have features of MOOSE is that it allows developers to write their own parser in order to translate the code into the FAMIX meta-model supported by the MOOSE core. The meta-model is an object-oriented representation of entities representing the software artefacts of the target system (Nierstrasz *et al*., 2005). This means that all information transformed into MOOSE has the same internal representation, which other tools can use to extend the functionality of MOOSE.

MOOSE offers a lot of different views. Some of the views use colours and shapes to express the essence of the view. The following is a short presentation of some of the views offered:

- Blueprint complexity: shows the internals of a class, by dividing the class into five layers, which are described in (VIZ, 2008)
- Method distribution map: visualises the number of methods each class has
- Method invocation: This view is similar to sequence diagrams in UML
- Class diagram
- System complexity: The system complexity visualises a class hierarchy where each rectangle resembles a class

The parser is one of the more powerful features of MOOSE because it captures a lot of information about the source code.

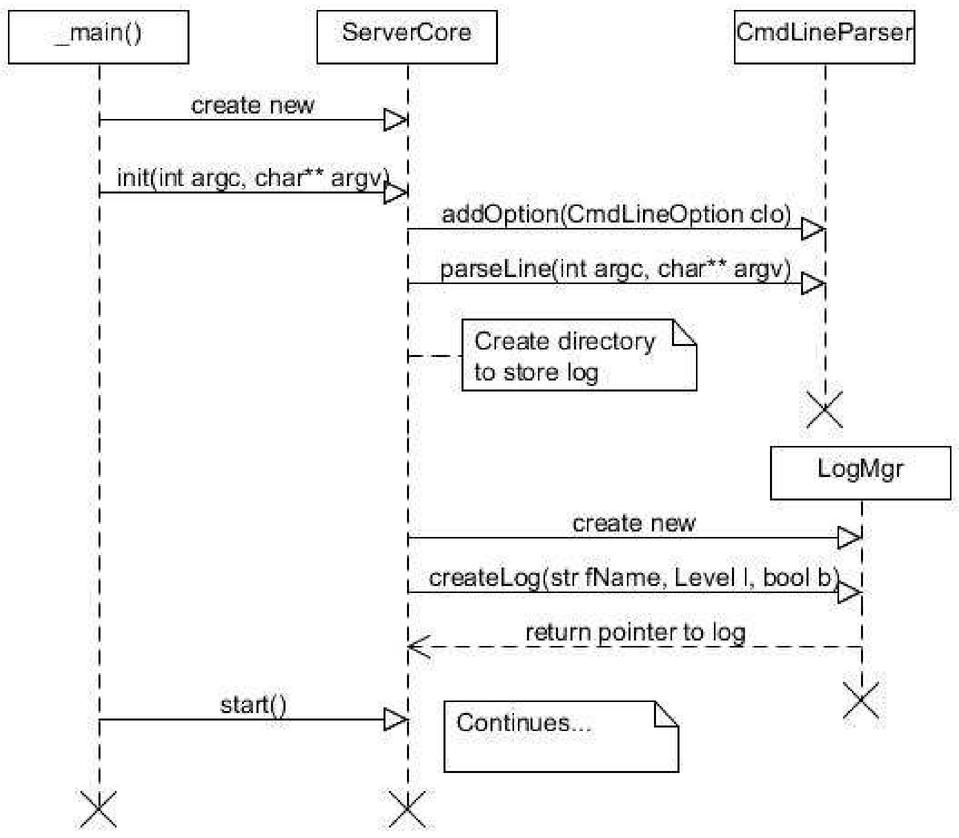
_main()
ServerCore
CmdLineParser
create new
init(int argc, char** argv)
addOption(CmdLineOption clo)
parseLine(int argc, char** argv)
Create directory to store log
LogMgr
create new
createLog(str fName, Level l, bool b)
return pointer to log
start()
Continues...

Fig. 3. Sequence diagram illustrating the start process of the server

Table 5. The symbolism mapped to a proper UML nota-tion and an appropriate view

Symbolism	MOOSE	UML	View
Class, function and data member	Blueprint complexity	Classes and objects	Module view C and C view
Class, function	Method distribution map	Classes	Module view
Class, function	Method invocation	Classes and objects	C and C view
Class, function and data member	Class diagram	Classes	Module view
Classes	System complexity	Classes	Module view

For instance, MOOSE creates different kinds of lists which contain information of all classes, variables, method invocation, namespaces, outgoing/ingoing accesses from a class Unfortunately, one of the weaknesses of MOOSE is that it does not support any search features. Finding specific variables to see which classes use it, takes time.

Among all the views provided by MOOSE (**Table 5**), the most used view from MOOSE is the Method invocation view. It gives a good overview and visualization of the collaboration among classes. This is useful when creating class diagrams which support a scenario. The set of classes used in a scenario is typical only a subset of all the classes constituting the system. It is difficult to find the first class in this subset. This is because it may require domain knowledge, debugging or heuristics. However, from that point, it is easy to find the others by following the arrows in the Method invocation view.

The Method invocation view can also be used when reconstructing sequence diagrams to highlight scenarios and collaborations. It is also difficult to find the starting point using this view, but apart from that, the overview provided by this view helps to identify collaborated objects.

The time it takes to make sequence diagrams (**Fig. 3**) using the Method invocation view is roughly the same time as without using it. It only helps to find collaborated objects, as it does not allow you to see the structure of the code, as Code Visual does, which is necessary in order to build the sequence diagram. However, it is faster when using it for Class diagrams as you do not have to look inside the code to see collaborated classes. The list of outgoing accesses from a class is useful for finding the associations.

4. EVALUATION

Next, we discuss and evaluate the experience of the reconstruction process from two perspectives. First we will outline our experience with the tools. Secondly, we will evaluate the value of the tools and discuss how the results can be verified.

4.1. Experience with the Tools

The experience of the tools used to reconstruct the software architecture will now be outlined. The tools are categorised into two categories: static analysis and dynamic analysis. The static analysis is based on the application artefacts while the dynamic analysis is used to capture data from the running system. Whether the tools support either static or dynamic analysis or even both are briefly summarised in the following taxonomy (**Table 6**).

It was not possible for one tool alone to generate diagrams for all the viewpoints outlined in (Bass *et al*., 2003) as outlined below (**Table 7**). The tools are mainly targeted for the module viewpoint.

Table 6. The tools categorised into analysis types

Tool	Static Analysis	Dynamic Analysis
Lattix	X	
VS 2008	X	X
Code Visual	X	
MOOSE	X	X

Table 7. The tools categorised by the viewpoints, they address

	Tool			
Viewpoint	Lattix	MOOSE	VS 2008	Code Visual
Module view	X	X	X	
C and C view		X	X	X
Allocation view	X			

However, some of the tools do have the ability to collect data for both the C&C view and allocation view, even though the data for the C&C view is displayed in an unstructured manner that is difficult to use.

4.2. Verification and Usability of the Result

It is difficult to verify the output generated by the tools alone and this is why we have chosen to map the symbolism to a common set of views. For tools that cover the entire set of views, we have a complete reconstruction of the architecture and for tools that signal the same structures we also have a reasonable correct structure.

However, not all tools are able to offer the same level of abstraction hence some tools offer a conceptual output while others offer a detailed sequence diagram. This makes it difficult to compare the output, but all tools are able to collect and generate data that can be used for the module view.

Illustratively, this means that the tools Lattix, MOOSE and VS 2008 can be helpful to construct both class and package diagrams for the module view. Lattix and VS 2008 identify packages and all of the tools identify common classes. The tools agree on the collected information about the simple structures such as classes and their simple relations. This facilitates that generation of the simple structures is reasonably correct, but still they do not necessarily reflect upon the correct use of complex structures.

The result is reliable as long as it just depends on simple structures and relationships, but it is insufficient when it comes to complex patterns. The final result was conclusively not as useable as expected, but the tools were able to generate a high-level description of the system which might be helpful when archiving an overview.

5. CONCLUSION

How much of an architectural description needs to be reconstructed depends on the task at hand and can range from a single diagram, if the company regularly updates its documentation, to everything included in an architectural description. Reconstructing the architectural description is not a process that is done overnight if everything is required. A lot of data needs to be extracted from the artefacts of the system and further analysed, visualised and then analysed again.

The reconstruction process uses two general techniques to collect the required data: A static analysis which collects its data from the artefacts belonging to the application and a dynamic analysis which collects its information from a running system.

In general, it is very difficult for tools to collect data from an existing application. This makes it even harder to find a tool which is suited for your special needs. There are some commercial products on the market, but very few of them are capable of collecting all the data you might need for the reconstruction process as you have to combine several tools. The disadvantage with multiple tools is often missing collaboration between them. When selecting the tools to be used it is important to be aware of prices (especially for commercial products) because the economical aspects are definitely not cheap and the initial costs could therefore increase unexpected.

Future work is to further investigate the features of the tools to make a more automatic process of the sequence diagram generation and pattern recognition through both simple and complex structures.

6. ACKNOWLEDGEMENT

Thanks to everyone who helped with this article. Especially thanks to Trapeze Group Europe A/S for the economical contributions.

7. REFERENCES

Bass, L., P. Clements and R. Kazman, 2003. Software Architecture in Practice. 2nd Edn., Addison-Wesley Professional, Boston, ISBN-10: 0321154959, pp: 560.

Deursen, A.V., C. Hofmeister, R. Koschke, L. Moonen and C. Riva, 2004. Symphony: View-driven software architecture reconstruction. Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture, Jun. 12-15, IEEE Xplore Press, pp. 122-132. DOI: 10.1109/WICSA.2004.1310696

MOOSE, 2008. Model-Based Object-Oriented Software. Generation Environment. Universität Kaiserslautern.

Nierstrasz, O., S. Ducase and T. Girba, 2005. The Story of Moose: An agile reengineering environment abstract Oscar Nierstrasz. Pennsylvania State University.

Pollet, D., S. Ducasse, L. Poyet, I. Alloui and S. Cimpan *et al*., 2007. Towards a process-oriented software architecture reconstruction taxonomy. Proceedings of the 11th European Conference on Software Maintenance and Reengineering, Mar. 21-23, IEEE Xplore Press, Amsterdam, pp: 137-148. DOI: 10.1109/CSMR.2007.50

Schmidt, D., M. Stal, H. Rohnert and F. Buschmann 2000. Pattern-Oriented Software Architecture. 1st Edn., John Wiley and Sons, England, ISBN-10: 0470065303, pp: 636.

VIZ, 2008. Polymetric Views. University of Berne.