Design Suite: Towards an Open Scientific Investigation Environment for Software Architecture Recovery

Roberto Almeida Bittencourt[1,2], Jemerson Figueiredo Damásio[1], Gustavo Jansen de Souza Santos[1], Adauto Trigueiro de Almeida Filho[1], José Martins da Nóbrega Filho[1], Jorge César Abrantes de Figueiredo[1], Dalton Dario Serey Guerrero[1]

[1]Formal Methods Group – Federal University of Campina Grande (UFCG)
Caixa Postal 10.106 – 58.109-970 – Campina Grande – PB – Brazil

[2]Computer Engineering Program – State University of Feira de Santana (UEFS)
Av. Transnordestina, s/n – 44.036-900 – Feira de Santana – BA – Brazil

roberto@uefs.br, {abrantes,dalton}@dsc.ufcg.edu.br

Abstract. *Empirical research in architecture recovery suffers from a lack of open scientific environments for experimentation. Moreover, used tools generally suffer from interoperability issues, either being too language-specific or requiring the use of a variety of data exchange formats. This paper describes Design Suite, a toolset that targets these issues and is used to extract low-level designs from source code, abstract them into architecture structural views and visualize the latter in different software visualization layouts. Designs and architecture views are represented as directed, typed, attributed multi-graphs which are exchanged between tools through the XML-based GXL standard. A proof of concept is presented as an initial evaluation of the suite.*

Resumo. *Pesquisas experimentais sobre recuperação arquitetural sofrem com a ausência de ambientes de experimentação científicos abertos. Além disso, as ferramentas atuais possuem baixa interoperabilidade. Este artigo apresenta a Design Suite, uma suíte de ferramentas que tenta atacar estas questões e que permite extrair designs de código-fonte, abstraí-los em visões arquiteturais estruturais e visualizá-las em diferentes leiautes. Designs e visões arquiteturais são representados por multigrafos dirigidos, tipados e com atributos, os quais podem ser trocados entre as ferramentas através de GXL, um padrão baseado em XML. Uma prova de conceito é apresentada como avaliação preliminar da suíte.*

1. Introduction

Architecture recovery is an issue that has received considerable attention from the software engineering community since the early nineties. Among other uses, it generally improves software comprehension, helps in documenting legacy systems, is a starting point for reengineering processes and helps to identify components for reuse [Kazman et al. 2001].

A large body of research on architecture recovery has been built for around fifteen years. A recent work surveys 146 papers which are either considered influential or propose a specific approach to the architecture recovery problem [Pollet et al. 2007]. With so much work dealing with different inputs, outputs, techniques and processes and

aiming at different goals, a variety of tools has been built to support architecture recovery. The trouble with these tools is that they either are too language-specific (e.g.: focused on Cobol, Java or C) or use different formats for their inputs and outputs. Furthermore, most research tools are proprietary, what makes it difficult to take advantage of existing approaches in order to build better tools or even reproduce the existing approaches in experiments. Last but not least, most available tools do not take full advantage of the latest advances in extraction, abstraction and visualization techniques, basic steps in the architecture recovery processes. For instance, some advanced extraction tools may be coupled to old visualization tools, what makes them less appealing to the research community.

This work is a first step in an effort towards an open scientific experimentation environment for software architecture recovery. Our toolset *Design Suite* does so in the following ways. First, it uses an XML-based standard named GXL to provide data interoperability. Second, it is built on top of a popular open source framework for graph representation and analysis [O'Madadhain et al. 2005]. Third, whenever available, it makes use of open source tools for each of the steps in the architecture recovery process. Finally, it presents a modular architecture, separating the concerns of extraction, modeling, abstraction and visualization in different software components. The toolset presently recovers only structural architecture module views [Clements et al. 2002].

2. Architecture Recovery Process

Architecture recovery usually follows a three-step process: extraction of low-level facts from software artifacts, abstraction of these facts into high-level components and connectors and their presentation in easily understandable architecture views. Each step is detailed below, where we discuss issues specifically related to structural module view recovery.

2.1. Extraction

Through static analysis one can extract low-level facts either from source code or from intermediate code (e.g.: Java bytecodes). These facts are composed of entities from source code and relations that capture dependencies between these entities. A myriad of output formats are generally used to represent low-level facts: DBMS tables, graph adjacency matrices and a sort of different entity/relation tuples files, what brings data interoperability problems. To deal with these issues, an XML-based standard named GXL has been developed. GXL is the result of a cooperative effort of an international group of researchers from software reengineering and graph transformation communities [Holt et al. 2006]. It represents a design abstraction as a typed, attributed, directed multigraph.

2.2. Abstraction

Abstraction tools combine low-level entities and relations into more abstract components and connectors [Armstrong and Trudeau 1998]. Typical operations from structural abstraction tools are *lifting*, *filtering*, *clustering* and *reclustering*.

In order to explain these operations, it is important to define an architectural model. Our architectural model is expressed as a hierarchy of designs lifted from initial low-level facts (level 0 design model) to more abstract existing entities (e.g.: classes or

files in a level 1 design model), which are then clustered into an architecture module view (modules in a level 2 or higher design model).

Lifting abstracts low-level entities (e.g.: fields, methods or functions) into their containers (e.g.: classes or files), generating a new design model with only container entities and their relations (either simple relations or derived relations from contained entities). *Filtering* removes entities from a design model which are considered unimportant for other purposes (e.g.: lifting or clustering). *Clustering* aggregates entities into cohesive sets, which form higher-level components. It can be done through quasi-manual, semi-automatic or quasi-automatic techniques. *Reclustering* allows changing an existing module view through specific move, join and split operations made by an expert or an algorithm in order to improve or correct the module view. It is typically used to allow an expert giving the final word on the software architecture.

2.3. Visualization

Visualization is essential to architecture understanding. Structural architecture module views can be graphically described through a variety of visualizations layouts [Diehl 2007]. Visualization results must be clear and concise, and are usually displayed as views routinely adopted for architecture documentation, such as graph, UML or matrix views. Hierarchical visualization is desired, since module views may contain decomposition styles. Furthermore, visualization is also used as a means of architecture view interaction and modification. Typical viewing layouts are graphs (forces, radial), hierarchies, aggregations, tree maps, trees, matrices and many others [Diehl 2007].

3. Design Suite

In our *Design Suite* toolset, each step from section 2 is respectively implemented as a set of techniques in the component tools *Design Wizard*, *Design Abstractor* and *Design Viewer*. Presently, we explicitly decided to recover only structural information.

Experimentation activities with the suite are supported by: *i)* command-line tools for extraction and abstraction, *ii)* scripting and logging capabilities in the abstraction tool and *iii)* command shell and graphical user interaction in the visualization tool.

3.1. Design Wizard

Table 1. Entity and relation fact set types

Source Entity	Relation
package	package contains class
	package contains interface
interface	interface contains field
	interface contains method
	interface extends interface
class	class contains field
	class contains method
	class contains class
	class extends class
	class implements interface
field	field is-a class
method	method receives class
	method returns class
	method accesses field
	method calls method
	method throws class
	method catches class

Design Wizard extracts a low-level design model from the bytecodes of a Java object-oriented system and outputs data in GXL format. It is used as our prototype tool, but it can be replaced by other GXL extractors for other languages. Extracted entities and relations are described in table 1. Entities and relations are typed and attributed.

3.2. Design Abstractor

In *Design Abstractor*, design entities are represented as graph vertices and design relations as graph edges. Our architectural model is then represented as a set of hierarchical graphs. In this way it is possible to represent each design level as a graph and connect a higher level to a lower level through special collapsed vertices and edges. These special vertices and edges belong to the higher level graph but are respectively composed of lower level vertices and edges.

Design Abstractor stores graphs in main memory. Each graph is a typed, attributed, directed, hierarchical multigraph. Abstractions are output into GXL files for tool interoperability. Besides, an API is also available so that other tools (e.g.: visualization tools) could manipulate in-memory design abstractions.

Up to the writing of this paper, implemented abstraction operations in *Design Abstractor* were: container *lifting* (e.g.: class, package and other hierarchical containers); *filtering* (by attribute or by regular expression over attributes); *clustering* by manual, semi-automatic (through regular expressions) and fully automatic (k-means, edge betweenness, modularization quality, design structure matrix and agglomerative) techniques; and *reclustering* (moves between clusters, cluster joins and cluster splits).

3.3. Design Viewer

Design Viewer currently presents four visualization layouts: *flat graph* views, *flat circular graph* views, *aggregate graph* views and *design structure matrix* views. In a *flat graph* view, force-based layout organizes graph vertices for a clearer view. In a *flat circular graph* view, the user can interact, choosing which node will be the central node of the graph. In an *aggregate graph* view, clusters are enhanced with shadows involving a set of nodes that belong together. Finally, in a *design structure matrix* view, entities are shown in rows and columns of a square matrix and matrix cells show the strength with which a row entity depends on a column entity.

Design Viewer allows for flexible addition of viewing layouts, due to the use of an MVC pattern that separates views from the design model graphs. Layouts either from visualization frameworks or in-house developed are used in the tool.

3.4. Design Suite Architecture

The architecture of *Design Suite* follows a modular decomposition to facilitate reuse and system evolution. The toolset is composed of four components: *Design Model*, *Design Wizard*, *Design Abstractor* and *Design Viewer*. The last three components were explained above. Besides these, *Design model* is responsible for encapsulating the set of hierarchical design graphs that semantically represent a system abstraction. This encapsulated model is accessed through an API that is used by *Design Abstractor*.

Figure 1 shows a component-and-connector view of *Design Suite* that mimics the architecture recovery process steps. Figure 2 shows a module view from Design Suite that show dependencies between tools and, inside them, between modules.

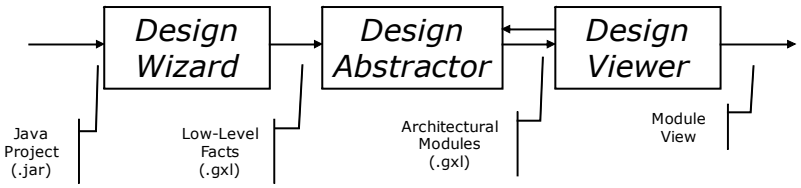

Figure 1. *Design Suite* component-and-connector view

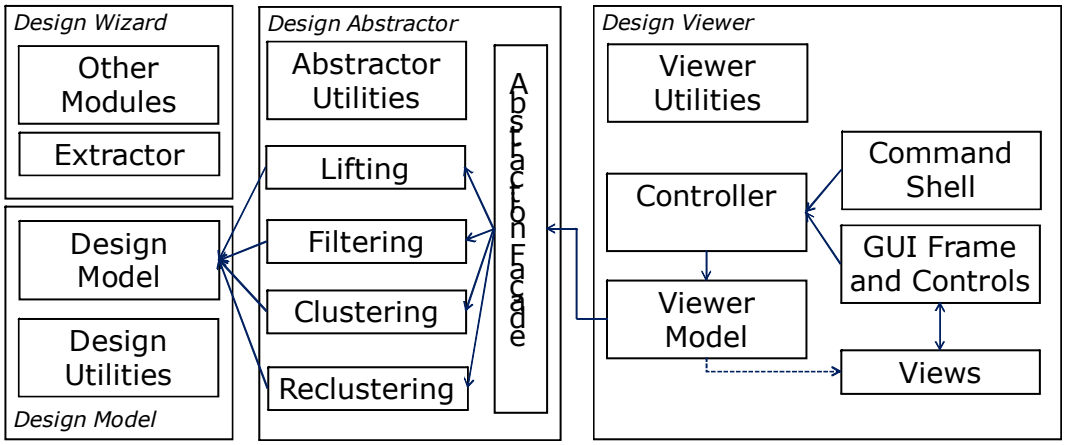

Figure 2. *Design Suite* module view

4. Proof of Concept

In order to have an initial qualitative evaluation of *Design Suite*, we applied the toolset to recover the architecture of the *Design Viewer* together with its dependencies (*Design Model*, *Design Abstractor* and *DSMViewer*). The whole analyzed system comprises 181 classes and 18492 lines of code. Its jar file size, without libraries, is of 346 KB.

In the following, we describe the steps we took in order to recover a module view from the system:

1. First, we extracted the low-level design from *Design Wizard*, which resulted in a GXL file of size 1953 KB with 2549 entities and 6897 relations;
2. Then, we filtered out packages because we did not want to take them into account in our module view recovery;
3. In the following, we aggregated inner classes elements inside them, so that these classes could be treated the same way as methods and fields in outer classes;
4. Then, we lifted the design to class level, resulting in a GXL file of size 150 KB with 181 entities and 464 relations;
5. With the class level design, we ran the *Design Viewer* tool, opened this file and chose a flat graph view;
6. Inside the viewer tool, we clustered the class level design with a design structure matrix algorithm, which generated an initial module view;
7. We opened an aggregate graph view and a design structure matrix view from previous module view and interacted with the system through reclustering steps until we reached the final architecture module view with 15 modules. The final state of this module view is shown in figure 3;

8. Finally, we saved the module view hierarchical graph as a GXL file which represents the recovered system architecture module view.

The steps above took approximately two hours, which seems reasonable for a small system where the architect knows well the architecture. The automated steps (extraction, filtering, clustering and viewing) did not take more than 5 seconds each. Most of the time was devoted to manual reclustering operations. The final output file represents the recovered architecture module view and can be used as the reference architecture for other purposes (e.g.: documentation, evolution, reuse or compliance).

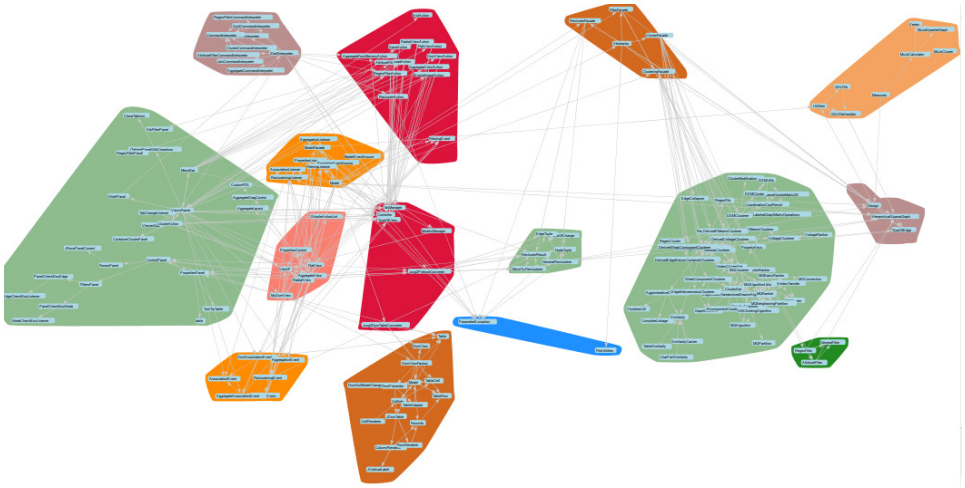

Figure 3. *Design Viewer* aggregate module view, after manual reclustering

5. Related Work

Design extractors extract information from software artifacts such as source code, intermediate bytecodes or UML design diagrams and export their results in a variety of data formats. Holt et al. (2006) describe some of these tools and propose a standard for data exchange between software reengineering tools.

Software architecture abstraction techniques operate on extracted designs and may be classified into quasi-manual, semi-automatic and quasi-automatic. A recent survey describes prolific work in this area [Pollet et al. 2007].

Research on information and software visualization provides insights for layout visualization ideas. Diehl (2007) surveys software visualization techniques and classifies them into methods that describe software structure, behavior and evolution.

AT&T has provided some of the first tools to extract designs from code. They used the framework CIA/Ciao to model concepts in programming languages and extract facts from source code, first in C [Chen et al. 1990] and later in C++ and Java. They also produced the GraphViz tool to visualize graphs [Gansner et al. 2000]. These tools were used as input and output to various other design abstraction tools.

Rigi is a system that extracts facts from source code into graphs that can be queried and edited by a graph editor [Müller et al. 1993]. Quasi-manual abstraction helps to filter, group and navigate through software entities. Visualization was improved by SHriMP, which provides a hierarchical fish-eye view [Storey and Müller 1995].

University of Waterloo has produced a pipeline of tools for architecture recovery named SWAG Kit [SWAG 2009]. Source code can be extracted with tools like *cppx* and *javex*, abstraction is done through Tarski relational algebra and scripts with the *grok* tool and software landscapes can be viewed with *lsedit*.

Bauhaus is probably the most comprehensive research toolset for reverse engineering, with tools ranging from dead code finding, clone detection, architecture recovery and compliance, feature detection and metrics [Raza et al. 2006]. Facts are extracted from source code and dynamic traces into low- and high-level representations. Abstraction is done through a semi-automated interaction framework that combines 12 automatic approaches and user guidance. Visualization is done through the Gravis tool.

Regarding availability, both Rigi and SWAG Kit are open source, while AT&T tools and Bauhaus are available by request for academic purposes.

As for exchange standards, all four toolsets have their own data formats (RSF for Rigi, TA for SWAG Kit, IML and RFG for Bauhaus and proprietary databases and DOT for AT&T tools), while we use the GXL standard for reengineering. Of course, this issue can be solved with converters to the GXL standard. Furthermore, most tools use two or more different formats that differ between low- and high-level representations while our toolset uses GXL for both levels.

Finally, all four related toolsets either do not take full advantage of the latest advances in visualization layouts and techniques or were produced with obsolete graphical rendering technologies, while ours uses recent 2D viewing frameworks.

Previous work with our toolset was done on design extraction and checking [Monteiro et al. 2009] and abstraction evaluation [Bittencourt and Guerrero 2009].

6. Conclusions

This work presented *Design Suite*, a toolset for software architecture recovery. This toolset is being constructed with two goals in mind: improving data interoperability with other tools and providing an open environment for scientific investigation of software architecture recovery. Up to the writing of this paper, three prototype tools had been developed: *Design Wizard*, which extracts low-level designs from bytecodes of Java systems, *Design Abstractor*, which provides facilities for design lifting, filtering, clustering and reclustering, and *Design Viewer*, which presents recovered module views in four visualization layouts and allows interacting with and modifying design models. A proof of concept was shown as a preliminary qualitative evaluation of the toolset.

Preliminary scalability analysis shows that *Design Wizard* can handle very large systems (with more than 10000 classes). *Design Abstractor* also handles them well, but some of its clustering algorithms (e.g.: modularization quality) do not scale for large systems (with more than 1000 classes). And *Design Viewer* is able to show graph views of large systems (up to 10000 classes). Scalability is an issue for further evaluation.

Future work will focus on improving viewing facilities and user interaction of abstracting features through *Design Viewer*. Moreover, *Design Abstractor* shall be improved with other abstraction algorithms in order to allow other comparative studies of abstraction techniques. Finally, tools for architecture mapping of lower level components to a reference architecture and architecture compliance checking between module views and implementation are being pursued.

Of course an effort like the one proposed here can only be successful with a cooperative work of the software maintenance research community. Thus, we would like to invite researchers to join us in this effort, which, in the long run, might bring us to a higher level in empirical research on software architecture recovery.

References

Bittencourt, R. A. and Guerrero, D. D. S. (2009). Comparison of Graph Clustering Algorithms for Recovering Software Architecture Module Views. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*.

Chen, Y., Nishimoto, M. and Ramamoorthy, C. (1990). The C Information Abstraction System. IEEE Transactions on Software Engineering, 16(3):325-334.

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J. and Little, R. (2002). *Documenting Software Architectures: Views and Beyond*. Pearson Education.

Diehl. S. (2007). *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Verlag.

Gansner, E. R. and North, S. C. (2000). An Open Graph Visualization System and Its Applications to Software Engineering. *Software: Practice and Experience*. 30(11):1203-1233.

Holt, R. C., Schürr, A., Sim, S. E. and Winter, A. (2006). GXL: A Graph-Based Standard Exchange Format for Reengineering. *Science of Computer Programming*, 60(2):149-170.

Kazman, R., O'Brien, L. and Verhoef, C. (2001). *Architecture Reconstruction Guidelines*. Technical Report CMU-SEI-2001-TR-026.

Monteiro, J. A. B., Guerrero, D. D. S. and Figueiredo, J. C. A. (2009). Design Tests: An Approach to Programmatically Verify the Design of your Code. In *Proceedings of the 31st International Conference on Software Engineering*.

Müller, H. A., Orgun, M. A., Tilley, S. R. and Uhl, J. S. (1993). A Reverse Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204.

O'Madadhain, J., Fisher, D., Smyth, P., White, S. and Boey, Y. B. (2005). *The JUNG (Java Universal Network/Graph) Framework*. Technical Report UCI-ICS-03-17.

Pollet, D., Ducasse, S., Poyet, L., Alloui, I., Cimpan, S. and Verjus, H. (2007). Towards A Process-Oriented Software Architecture Reconstruction Taxonomy. In *Proceedings of the 11th Conference on Software Maintenance and Reengineering*.

Raza, A., Vogel, G. and Plödereder, E. (2006). Bauhaus: A Tool Suite for Program Analysis and Reverse Engineering. In *Reliable Software Technologies, Ada-Europe 2006*, LNCS 4006, pages 71-82.

Storey, M. D. and Muller, H. A. (1995). Manipulating and Documenting Software Structures Using SHriMP Views. In *Proceedings of the ICSM*.

SWAG. (2009). SWAG Tools. http://www.swag.uwaterloo.ca/tools.htm. Accessed in May, 2009.