Architecture Recovery as first step in System Appreciation

Sandipkumar Patel
sandipkumar_patel@infosys.com

Yogesh Dandawate
yogesh_dandawate@infosys.com

John Kuriakose
john_kuriakose@infosys.com

SET Labs, Infosys Technologies
Pune, India

Introduction and Motivation

A large percentage of software projects within the practical context of software engineering activity deal with the evolution of systems. This implies that previous engineering activity has produced an executable representation of the business requirement's and now the software has to be changed due to changed implementation technologies including improvements to language, tools and runtime platforms or changes in the business context that require enhancements to be built into the software system. In either case we observe that the initial architecture representation is rendered obsolete.

There exists a mental model of the 'to be' or what we call the *prescribed architecture* that is initially captured either informally or even using available tools. However as the software evolves over its lifecycle the real architecture emerges. This architecture is in the source code and is rarely studied and measured for compliance with the prescribed model. This is what we call the *as-is architecture* and typically is represented only within source code. Significant effort and some amount of expertise in program comprehension are required for a developer or architect to understand this actual architecture.

In maintenance projects the first important phase of activity is well known as the system appreciation phase and one of the key deliverables from this phase is documentation that describes what the system does in terms of its functional features and how it achieves those feature in terms of its the architecture and the design. We focus on the specific tasks of recovering the as-is architecture from the source code.

Previous work in software architecture [1] has provided sufficient basis for the architecture views and styles of representation. Parnas has demonstrated that that the decomposed modular views of a software system is important in managing it design and evolution. [2,3]. We therefore intend to recover this modular view that represents the decomposition of software system, into its major building blocks. These can be organized to provide the big picture view of the system in terms of responsibilities and interactions. This view is best understood by styles such as the layered style, the ring style [1] or the box-and-line style.

Pervious work in this area [4] has demonstrated beneficial results in architecture recovery by observing the dependencies that exist among code elements. We start with a similar premise but adopt a different approach.

Approach

Our work is based on the premise that syntactical dependencies between code elements are always preserved in source code and form the basis of higher order relationships between packages and modules. Thus we set out to recover the modular view based on these syntactic dependencies. The key concept in our approach is the *dependency usage graph* that we extract from source code where code elements form nodes and the dependency relationships from edges between them. The edges are directional and each node therefore has a set of incoming and outgoing edges representing the elements that use a particle node (I am used by) and the code elements that are used by this node (I depend on) respectively. Our edges are classified based on clear taxonomy that we develop for this purpose. This taxonomy defines the strength of the relationship by attaching a numeric weight for each category within the taxonomy. This assignment is based on our subjective interpretations of design strengths based on syntactic features of the java language.

If $W_{(D)}$ represents weight for a dependency.

$W_{(SuperClassAccess)} > W_{(SuperInterfaceAccess)} > W_{(Proceduraldefinition)} > W_{(ConstructorInvocation)} > W_{(StaticMethodInvocation)} > W_{(InstanceMethodInvocation)} > W_{(InterfaceMethodInvocation)} > W(_{TypeUseAccess\backslash)} > W_{(ObjectInstantiationAccess).}$

Once the usage graph is available our algorithm first identifies language libraries and third-party libraries by identifying packages for which source code has not been supplied. This also represents a very simple mechanism for the user to determine his sphere of interest in large code bases – the specific subset of packages that he is interested in. Next our algorithm captures the inbound and outbound edges at each node and clusters the packages by detecting the current Top package with least inbound edges. Specific differentiation is achieved by removing self (dependencies on self) and dependency cycles. We address dependency cycles by systematically eliminating the weakest dependency relationship between the code elements in a cycle. Our algorithm assigns a layer index to each package that determines whether a package is above, below or a peer to other packages.

The model is finally visualized in text form using our simple variation of the ring view espoused in [1]. We call our visualization as *the brick layout of software modules*. The layout is different from the layering style in that is does not mandate strict dependencies to a single layer below. A module may freely depend on any of its peers and all modules below it at any depth below this module. Other than the visualization of packages in this brick layout we address two other concerns.

- The reduction of cognitive complexity by aggregating many packages into *domain friendly language* – we use a simple regular expression based mapping to organize packages into architecture modules. The names of these architecture modules can be such that even non-technical stakeholders get a grip of their responsibility and function. For example: javax.swing.* = SWING User Interface
  represents the mapping that all packages with "javax.swing" will be now shown in our results as SWING User Interface.
- The second contribution we make is with regard to our model in the form of the *dependency usage graph*. While the visualization captures the relationships at a high level of packages (with respect to Java) our model is strong on knowledge fidelity.

We are able to support interactive exploration of the elements shown in our visualization from package → To Type → to Method level to examine the exact nature of relationships that contribute to a packages position in the brick layout. We do this with the intention of supporting tool based comprehensions of a given code bases interactively.

Results and Future Work

We have evaluated our approach by running it on two open source software systems viz. JUnit – www.junit.org and JEdit – www.jedit.org and on a proprietary source of business application for the banking domain called SETL Bank. The results are promising and are attached in the appendix for the open source systems.
Specifically our approach is fairly lightweight and has reduced cognitive overload for non-technical stakeholders. It also supports our ongoing work related to tool based interactive program comprehension.
We are currently enhancing our work in two directions.

- Annotate the dependency usage graph with information about statement (location in source code) and data variable. To achieve this we need to analyze the source code over and above the compiled byte code.
- Design a simple language to capture architecture constraints that are derive the prescribed or 'to-be' model of the software system. This is crucial to measure architectural compliance as software evolves over time. This language will capture the "is allowed" (allowed-to-depend-on) and what is not allowed in terms of syntactic dependencies. It allows us to capture and apply partial architectural constraints on a software system

References

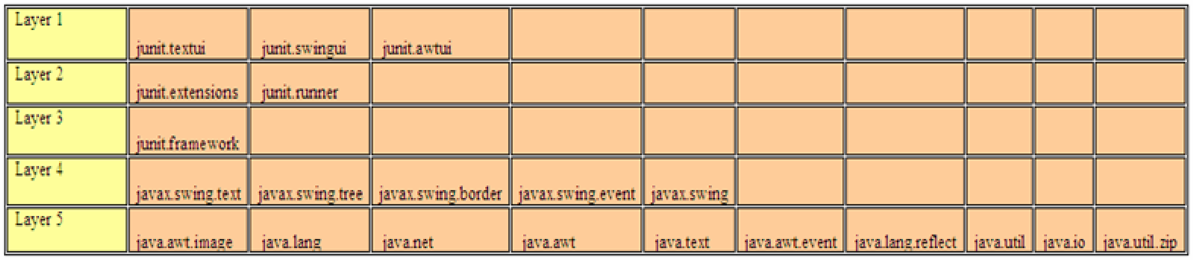
[1] Clements Paul…[et al.], *Documenting Software Architectures: Views and Beyond, chapter 2 pages 53-101,* Addison Wesley (ISBN 0-201-70372), 2002

[2] Parnas D.L., *The secret history of information hiding*. In Software Pioneers: Contributions to Software Engineering. Springer, 2002.

[3] Parnas, D. On the Criteria for Decomposing Systems into Modules. In Communications of the ACM, vol. 15, no. 12, pp. 1053–1058, 1972

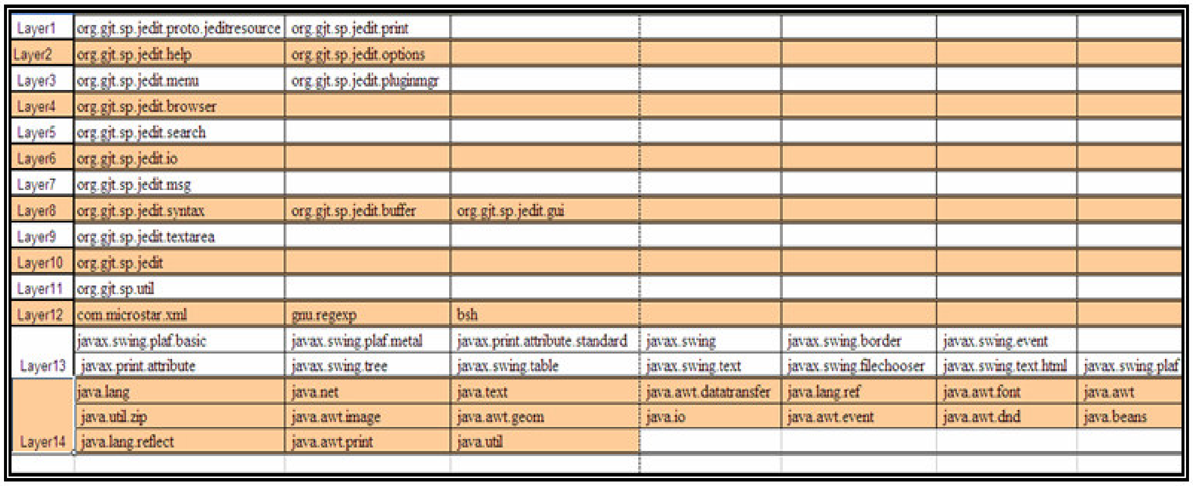
[4] Sangal, N., Jordan, E., Sinha, V., and Jackson, D. 2005, *Using dependency models to manage software architecture*. In companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05), San Diego, CA, USA, October 16 - 20, 2005, ACM Press, New York, NY, 164-165.

Appendix 1: Recovered Architecture View from JUnit



Layer 1	junit.textui	junit.swingui	junit.awtui							
Layer 2	junit.extensions	junit.runner								
Layer 3	junit.framework									
Layer 4	javax.swing.text	javax.swing.tree	javax.swing.border	javax.swing.event	javax.swing					
Layer 5	java.awt.image	java.lang	java.net	java.awt	java.text	java.awt.event	java.lang.reflect	java.util	java.io	java.util.zip

Appendix 2: Recovered Architecture View from JEdit



Layer1	org.gjt.sp.jedit.proto.jeditresource	org.gjt.sp.jedit.print					
Layer2	org.gjt.sp.jedit.help	org.gjt.sp.jedit.options					
Layer3	org.gjt.sp.jedit.menu	org.gjt.sp.jedit.pluginmgr					
Layer4	org.gjt.sp.jedit.browser						
Layer5	org.gjt.sp.jedit.search						
Layer6	org.gjt.sp.jedit.io						
Layer7	org.gjt.sp.jedit.msg						
Layer8	org.gjt.sp.jedit.syntax	org.gjt.sp.jedit.buffer	org.gjt.sp.jedit.gui				
Layer9	org.gjt.sp.jedit.textarea						
Layer10	org.gjt.sp.jedit						
Layer11	org.gjt.sp.util						
Layer12	com.microstar.xml	gnu.regexp	bsh				
	javax.swing.plaf.basic	javax.swing.plaf.metal	javax.print.attribute.standard	javax.swing	javax.swing.border	javax.swing.event	
Layer13	javax.print.attribute	javax.swing.tree	javax.swing.table	javax.swing.text	javax.swing.filechooser	javax.swing.text.html	javax.swing.plaf
	java.lang	java.net	java.text	java.awt.datatransfer	java.lang.ref	java.awt.font	java.awt
	java.util.zip	java.awt.image	java.awt.geom	java.io	java.awt.event	java.awt.dnd	java.beans
Layer14	java.lang.reflect	java.awt.print	java.util				