Architecture Reconstruction from Code for Business Applications - A Practical Approach

Santonu Sarkar
Accenture Technology Labs, Bangalore, India
santonu.sarkar@accenture.com

Vikrant Kaulgud
Accenture Technology Labs, Bangalore, India
vikrant.kaulgud@accenture.com

ABSTRACT

During application development or maintenance, the application (or a family of applications) under consideration often co-exists with several existing applications. This obviously requires the team members to be aware of the underlying architecture of these applications. To comprehend a complex code base it is necessarily to create multiple views (such as functional, technical, deployment etc.) of these applications, at multiple levels of abstraction. In absence of any formal documentation, and due to unavailability of any automated tool, the practitioner ends up creating a partially complete, ambiguous design from code and thereafter continuously struggle to keep it up-to-date. Most of the state-of-art tools reconstruct a low-level design from a code base. For a typical business application having a large code base, extracted low-level design is extremely complex to comprehend due to overwhelmingly large number of fine grained entities and their relationships. In this paper, we describe a semi-automated, iterative approach, called the "Design Discovery Method" (DDM), to model the hierarchical functional architecture of a family of applications at three levels of abstraction. We have experimented DDM on a few real life systems with multiple applications written in Java and received encouraging feedback from the practitioners on the overall approach.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.10 [**Software Engineering**]: Design—*Representation*

General Terms

Design

Keywords

Technical Design, COTS, Ontology, Architecture Style

1. INTRODUCTION

In an application development project, the system under construction, often co-exists with several existing applications. This obviously requires the designers to be aware of the underlying architecture of existing applications. In a maintenance scenario, the team inherits a large code base of a family of applications, and struggles thereafter, to comprehend the code base by manually reconstructing various views of the software. Comprehension of the underlying architecture of business applications at different levels of abstraction is paramount for their development and maintenance. An appropriately reconstructed architecture (or design) of a family of applications can significantly help the design, development and maintenance team to accomplish their tasks. An important question here is that what sort of design should be reconstructed? Most of the state-of-art tools reconstruct a low-level design from a code base. This serves the purpose of visualization of the code very well, but the question remains whether code visualization is sufficient for comprehension. For a typical business application having a large code base, extracted low-level design is extremely complex to comprehend due to overwhelmingly large number of fine grained entities and their relationships. Through our interactions with the practitioners, we found the following issues with current tools:

1. The low level design of a system does not help the practitioners to understand the functional model of system, without which, the reconstructed low-level design by the tool remains a poor comprehension aid.
2. Current tools do not provide the "top level view" of the application environment- specifically other existing applications with which a particular application co-exists and collaborates. Without this, practitioners never understand the overall context and find it difficult to gauge the complexity of a feature enhancement, or a change request.

To address these concerns, we have implemented an iterative mechanism, called the "Design Discovery Method" (DDM), to model the hierarchical functional architecture of a family of applications and link it with the codebase. The iterative mechanism combines automated discovery of the design from the code and the functional expert's viewpoint of the system. This project is currently in progress. In this paper we provide an outline of the approach.

The paper has been organized as follows. In the next section we provide a short survey of the existing works in design recovery and provide motivation behind our approach. In

Section 3 we briefly discuss the approach we have adopted, and the response we have obtained from our experiments. Finally we conclude the paper by discussing the issues related to the current approach and future plans.

2. BACKGROUND AND RELATED WORK

A representation of an application or a family of applications, that models the structural organization of these applications in terms of constituent components, their interactions among each other and with the environment is known as the software architecture [1]. Software architecture in itself, is complex and multidimensional, and it is expressed as different views [3]. An architecture consists of one or more architecture styles such as pipe and filter, and layered architecture to name a few [3]. The architecture also documents various quality attributes of the application, namely performance, security, reliability and so on. A software architecture of a family of applications, containing the above mentioned aspects, has always been deemed necessary for comprehension, analysis and implementation. In reality, however, the practitioners often find it difficult to represent the architecture containing all these aspects. Furthermore, as applications continuously evolve, the underlying code drifts away from the original architecture. This calls for an automated architecture recovery mechanism [6]. Discovery of all the architectural aspects from the code (different views, styles, quality attributes and so on) is extremely challenging; the underlying programming language is not capable of capturing the architectural characteristics. The architecture reconstruction, therefore, can't be fully automated; it has to be iterative, and interactive [5]. Ducasse et al. [4] have given a comprehensive survey of various architecture recovery approaches. Ducasse et al. have classified the approaches into three categories, namely *bottom-up, top-down*, and *hybrid* approach. A *bottom up* approach extracts low level implementation information from the source code and provides a visualization mechanism to view the extracted information. A typical *top-down* approach such as the Reflexion mechanism[7] allows the designer to define the high level architectural model in terms of modules and their interactions. Then the designer describes how this model is mapped to the source code. The discovery tool then uses the mapping information to link code elements with the high level model. The *hybrid* approach is a combination of these two. For instance, the approach proposed by Sartipi [11] uses the low-level information extracted from the code as well as the high level design elements to reconstruct the architecture. Our earlier paper [9] also uses a hybrid approach to discover layered architecture style from the source code. The approach proposed by Sangal et al. [8] also extracts layered architecture style only based on structural characteristics of the code.

3. PROPOSED APPROACH

We model the architecture of a family of applications as the hierarchy modules, with three levels of abstractions. The idea has been explained in Figure 1. We explain the model in the following subsection.

3.1 Model

The topmost level of abstraction in this figure captures the application itself, or a family of applications, and the

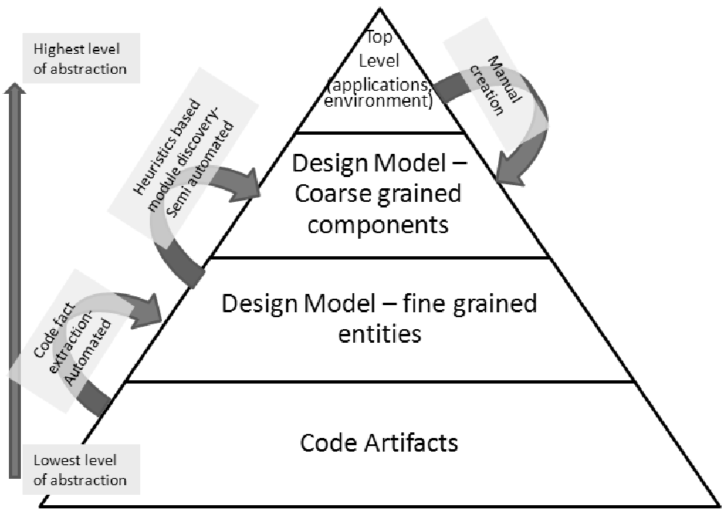
Highest level of abstraction
Lowest level of abstraction
Top Level (applications environment)
Manual creation
Heuristics based module discovery- Semi automated
Design Model – Coarse grained components
Design Model – fine grained entities
Code fact extraction- Automated
Code Artifacts

Figure 1: Functional Architecture- Hierarchical Model

environment. Interactions of these applications among each other and with the environment are captured in the model. Each application, if necessary, is then hierarchically decomposed as a collection of coarse grained functional components. Thereafter, each component is further broken down in terms of packages, fine grained implementation classes and their interactions as illustrated in Figure 1. Finally, each implementation class and package are linked to the codebase. Such a hierarchical representation can manage the underlying complexity of the system through abstraction. Thus, this approach helps practitioners to easily understand the application; starting from its environment, and navigating down to its implementation details.

To represent the hierarchical representation, we have used Acme [12] as the architecture description language. The language supports hierarchical decomposition. Furthermore, the language allows us to define new component types which are extremely useful to capture the notion of a business application, external systems, database and so on. With the help of component types such business application, databases, external systems we can model the family of applications, their interactions and the environment. One may refer to our earlier publication [10] for a detailed description of the type system.

3.2 Process

Our approach falls under the hybrid mechanism as described in [4]. DDM uses a combination of a manual top-down and an automated bottom up approach to construct the architecture, at three levels of abstraction, shown in Figure 1. The process has been shown in Figure 2.

Bottom-up. In the bottom up approach, the tool uses available fact extractors to extract code facts from the source code. It creates the implementation level view in terms of implementation classes and packages. Next, in the "Discover Functional Modules" step in Figure 2, the tool constructs a set of coarser grained functional components at the next higher level of abstraction. The heuristics utilizes the direc-

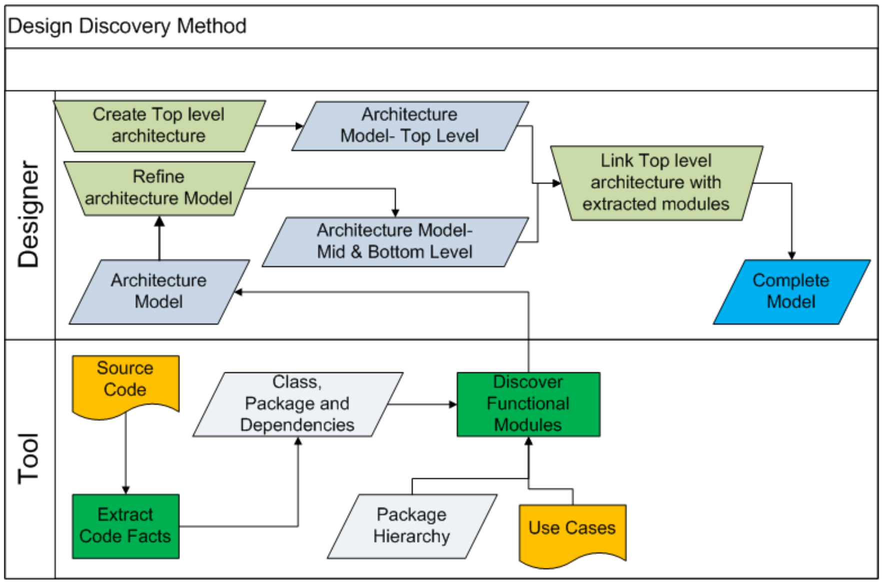
Design Discovery Method
Designer
Tool
Create Top level architecture
Architecture Model- Top Level
Refine architecture Model
Link Top level architecture with extracted modules
Architecture Model- Mid & Bottom Level
Architecture Model
Complete Model
Source Code
Class, Package and Dependencies
Discover Functional Modules
Extract Code Facts
Package Hierarchy
Use Cases

Figure 2: Architecture Extraction Process

tory structure of the codebase, since developers often uses directory structure to organize the code elements related to a functional component. Currently the tool decides the component name from the name of a directory residing at a certain depth. The depth is decided by analyzing the directory tree and the maximum fanout.

Top-down. In the top-down approach, the designer manually creates the architecture at the highest level of abstraction; here the human expert defines the topmost view, in terms of a family of applications in Acme. At this level the designer typically defines the applications under consideration, external systems and third party applications using our own architecture family [10]. The DDM tool then assists the practitioners to link this architecture model with the extracted mid-level components as shown in Figure 2. We have also defined a lightweight process to keep the reconstructed model in sync with the system (or the family of systems) as it evolves. In the following section we illustrate this process in more details with the help of an example.

4. DISCUSSION

In this section we illustrate the approach using a familiar example of Java "Petstore". A Petstore is a simple on-line application that allows user to log in to a portal and buy pets. The example illustrates various aspects of a web-based distributed application. For our purpose we have chosen the Petstore version 2.0[2]. The top level and mid level architecture of Petstore has been shown in Figure 3 and Figure 4 respectively. As discussed in the beginning of Section 3, the top level architecture in Figure 3 is created manually. Here the architect defines the applications such as the petstore, and other external systems with which it interacts such as the document search module (SearchEngine), map and geo-location identifier (GeoMap), RSS feeds (RSSFeeder) and credit card payment module (Paypal). Next, by analyzing the petstore code base, DDM finds out the set of classes,

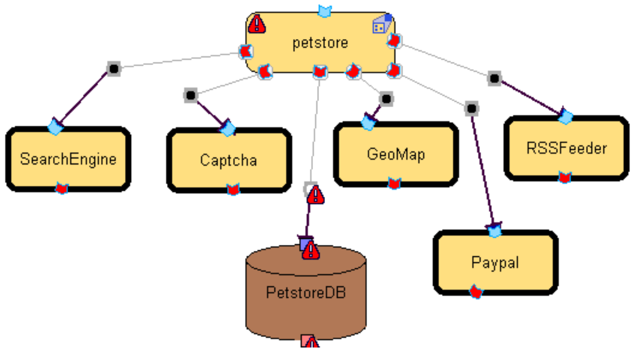

Figure 3: Petstore Architecture- Top Level

their interactions and packages. This forms the design at the lowest level of abstraction. The classes and packages are then grouped together to form higher level components, as shown in Figure 4. This forms the design at the next higher level abstraction. The tool decides the name of the component from the name of package hierarchy. For Petstore, the selected names are: `controller`, `search`, `model`, `util`, `captcha`, `mapviewer` and `proxy`. These components are then manually linked to the petstore application in Figure 3 so that they become subcomponents of "petstore". To link these components, we utilize the notion of *representation* in Acme. Note that a *representation* of a component in Acme models hierarchical decomposition of the component into more fine grained (sub)components. After the linking is complete, we now get the architecture of petstore system at three levels of hierarchy.

We have experimented DDM on a few other real life systems with multiple applications written in Java. Specifically, we wanted to understand the usefulness of the model and the effort required for complete reconstruction. The initial feedback is encouraging. Even for a large codebase, the bulk

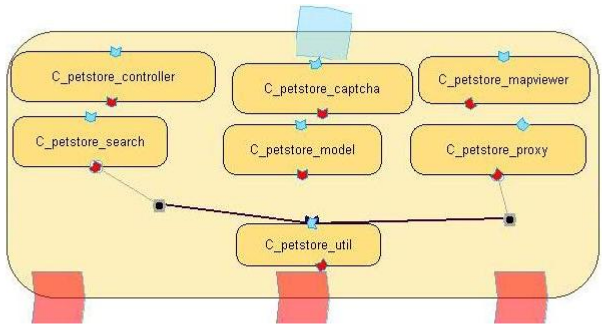

Figure 4: Petstore Architecture- Next Level

of the effort to construct the lower level views, are created automatically (with manual correction) by the bottom-up approach. The top-down views are not effort intensive at all even for large family of applications. This is due to fact that at the highest level of abstraction, the number of extremely coarse grained components are not that many and can be easily created by hand. We believe that the complexity of maintenance of the model can be managed through our process. Project teams felt that the architecture at multiple levels of abstraction helps in integrating the functional knowledge of business applications with design reconstruction techniques to create a unified, hierarchical model. Most importantly, we found the model to be an effective communication mechanism between the business analysts (who are functional experts) and the designers. We hope that such a representation will eventually lead to a better comprehension for application development or maintenance work.

5. CONCLUSION

In this paper we describe an approach to extract the functional architecture at multiple levels of hierarchy from code. The architecture consists of three levels of hierarchy, modeled using Acme architecture description language. In this paper we illustrate this approach using Petstore as the example. Representing the architecture at multiple levels of hierarchy is an effective mechanism to comprehend the system. Our interaction with the practitioners have validated this hypothesis. The notion of integrating the top level functional view with the low level detailed design view has its own challenges. The current heuristics to construct the mid level components is relatively naive. This can certainly be improved to get a more accurate set of components.

Extraction of information from source code can only provide the structural information of the application. Unfortunately this information hardly provides any clue to various other architectural information such as the architecture style used, the technical components, non-functional attributes and other stuff. Recovering those aspects from source code only is not always possible. User-assisted automatic discovery of these architectural characteristics will be an interesting research area to explore.

As the next step, we intend to improve the heuristics to recover higher level components. We also intend to analyze other artifacts such as SQL table schema, configuration files, use case documents and discover their relationships with the source code. While interacting with the practitioners, we realized that keeping the extracted model up-to-date with the latest development will be an important factor. We intend to take up some of the above challenges as the future research direction.

6. REFERENCES

[1] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

[2] M. Basler, S. Brydon, D. Nourie, and I. Singh. Introducing the Java Pet Store 2.0 Application. 2007.

[3] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architecture*. Addison Wesley, September 2002.

[4] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[5] J. Grundy and J. Hosking. High-Level Static and Dynamic Visualisation of Software Architectures. In *Symp. of Visual Languages*, pages 5–12, 2000.

[6] N. Medvidovic and V. Jakobac. Using Software Evolution to Focus Architectural Recovery. In *Automated Software Engineering*, volume 13, pages 225–256, 2006.

[7] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, 27:364–380, 2001.

[8] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[9] S. Sarkar, G. Maskeri, and S. Ramachandran. Discovery of Architectural Layers and Measurement of Layering Violations in Source Code. *Journal of Systems and Software*, 82(11):1891–1905, 2009.

[10] S. Sarkar and A. Panayappan. Formal Architecture Modeling of Business Application− Software Maintenance Case Study. In *IEEE Tencon, Region 10.* IEEE, 2008.

[11] K. Sartipi. *Software Architecture Recovery based-on Pattern Matching.* PhD thesis, School of Computer Science, University of Waterloo, 2003.

[12] B. Schmerl and D. Garlan. AcmeStudio: Supporting Style-Centered Architecture Development. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.