Tool Support for *Just-in-Time* Architecture Reconstruction and Evaluation: An Experience Report

Ian Gorton
Empirical Software Engineering Group, National ICT Australia
School of Computer Science and Engineering, University of New South Wales
Ian.Gorton@nicta.com.au

Liming Zhu
Empirical Software Engineering Group, National ICT Australia
School of Computer Science and Engineering, University of New South Wales
Liming.Zhu@nicta.com.au

ABSTRACT
The need for software architecture evaluation has drawn considerable attention in recent years. In practice, this is a challenging exercise for two main reasons. First, in deployed projects, software architecture documentation is often not readily available, and may not be a correct representation of the as built architecture. Second, large software systems have numerous potential views of the various architecturally significant structures in the system. In this paper we assess the capabilities of software reverse engineering and architecture reconstruction tools to support just-in-time architecture reconstruction. If an application's architecture can be reconstructed efficiently, this could promote more effective architecture reviews and evaluations. We describe our experiences in leveraging multiple reconstruction tools and how these guided the choice of design artifacts to construct. We discovered that the tools complemented each other in identifying reconstruction scope, critical architectural elements, potential design irregularities and creating useful architectural views for different evaluation tasks. With the help of these tools, the reconstruction and evaluation effort was significantly streamlined and productive. Finally, we also report some potential improvements these tools could make.

Categories and Subject Descriptors
D.2 [**Software**]: Software Engineering; D.2.10 [**Software**]: Software Engineering – *Design*; D.2.11 [**Software**]: Software Engineering – *Software Architectures*; D.2.2 [**Software**]: Software Engineering – *Design Tools and Techniques*

General Terms
Documentation, Design, Theory

Keywords
Software Architecture, Architecture Evaluation, Reverse Engineering, Reverse Architecting, Automated Software Engineering, CASE Tools

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

1. INTRODUCTION
Software architecture (SA) evaluation has emerged as an important software quality assurance technique [7]. Architecture evaluation has been used as a way to prevent architecture erosion, prepare for re-engineering and ensure implementations are in alignment with the intended application architecture.

To perform an effective architecture evaluation, a suitable set of architecture documentation is required [11]. Ideally, the architecture documentation needs to be concise but effective, flexible, interactive and dynamic. However in many deployed applications, having high quality architecture documentation available and synchronized with the application implementation is difficult and costly [23]. This is especially true for large applications with many potential views for different subsystems or aspects of the system.

Software reverse engineering technologies can help with just-in-time reconstruction of the architecture documentation when a review is scheduled. Generating current architecture documentation precisely when it is required for review ensures the evaluation team has an accurate representation of the application architecture. In such circumstances, architecture reconstruction becomes an integrated activity for performing architecture evaluations. However, reverse engineering remains a human-intensive activity that requires knowledge from software engineers and domain experts in addition to automated tool guidance. [25].

In this paper, we exercise five different tools to conduct an architecture reconstruction and evaluation on a major subsystem of a commercial application. Our aim was to assess if architecture reconstruction and evaluation can be streamlined with appropriate tools support. This would make just-in-time reconstruction a potentially attractive strategy in the many projects that do not maintain current architecture documentation, and promote more frequent reviews due to the reduced overhead of generating the architecture documentation. The overall process followed was based on Quality Attribute Driven Software Architecture Reconstruction (QADSAR) [29] which integrates the spirit of scenario-based architecture evaluation [7].

We begin by discussing background work on architecture evaluation, reverse engineering and architecture reconstruction tools. We then explain our selection of tools along with the introduction of the software we evaluated. We report our experience in section five by demonstrating the reconstruction and evaluation process. We then present our findings in section six.

2. RELATED WORK

2.1 Software Architecture Evaluation and Modifiability

It has been shown that SA constrains the achievement of various quality attributes (such as performance, maintainability and usability) in a software intensive system [7]. One of the most important quality attributes is modifiability. Architecting applications in the face of uncertain requirements and creating flexible, modifiable architectures has drawn substantial attention from both industry [16] and academia [9]. Models for predicting modifiability at the architectural level have been proposed and validated in industrial settings [24]. However, these methods have not focused on reconstructing an architecture from an existing code base and evaluating its modifiability.

2.2 Software Architecture Reconstruction

Software architecture reconstruction has its root in traditional software reverse engineering. Software reverse engineering has focused on program understanding and visualizing code structure and behavior (e.g. call graphs, data flow diagrams) at different abstraction levels [25].

Architecture reconstruction requires more than program comprehension and code level analysis. This is because software architecture affects quality attributes, and these are mostly not directly expressed in an application's source code. Architecture evaluations have the goal of evaluating important quality attributes. Quality attribute driven architecture evaluation complements various architecture reconstruction methods [15, 19, 20, 30, 32] to address this issue [29].

Consequently, architecture reconstruction requires the reconstructed model to incorporate architecturally significant concepts. These include architectural level components and connectors including COTS technologies, design patterns, component distribution information and architectural decisions as first class elements [10]. These typically cannot be directly extracted from source code without input from software engineers/architects. This additional information must be constructed and expressed during the reconstruction process [20] by producing models at different abstraction levels. Importantly, the mapping between levels must be retained and available during the architecture evaluation [25]. However, the precise nature of the architectural level concepts that need to be constructed depend on the goals of the evaluation being undertaken - they can not in the general case be pre-determined.

2.3 Reconstruction Tools and Design Metrics

There is a range of existing tools that can contribute to an architecture reconstruction exercise. Substantial effort has also been made in developing reconstruction tool frameworks to facilitate tool interoperability [28, 31]. Comparison studies have been performed to evaluate reverse engineering tools for program understanding [8] and architectural recovery [6]. These comparisons address the usability of the tools, the common views the tools can generate and the accuracy of their extraction and abstraction.

In contrast, little has been reported on how these tools support a selective architectural reconstruction and evaluation process. Architecture reconstruction is inherently an exploratory activity [25, 29]. An architect must determine the architecturally significant abstractions and choose suitable views to depict their relationships.

Many of the tools reviewed in the literature have been superseded or are only research prototypes. Therefore, in order to undertake the reconstruction project, we selected state of the arts tools from industry and the open source and research communities. As far as we could ascertain, there is no single *silver bullet* tool that can solve all the needs of an architecture reconstruction effort. It was hence necessary to use several tools in a complementary fashion, exploiting the strengths of each to provide architecture insights, metrics and views.

3. The PROJECT

A NICTA industrial collaborator was interested in assessing a core Java subsystem of one of their commercial products as a candidate for porting to a new component architecture. Consequently, they were interested in the compatibility of their existing architecture with the target component architecture, and how much effort would be required to modify the code.

The code base is an integral part of a series of financial products that are written in Java. The system generates interesting financial data which can be accessed by subscribers to the service. The financial data produced by the application needs to be generated in an extremely flexible manner so that it can be easily tailored to each subscriber's needs. The content delivery mechanism exploits XML formats and XSLT transformations to render tailored views. In addition to the main business logic packages, there are several utility packages for handling view generation, XML serialization and subscriber request handling.

The code base was first developed in 1999. It has since been used in multiple projects, and has been refactored on several occasions to progressively improve the design quality and functionality. The code is well documented using Javadoc. It has 50k lines of code with 296 classes/interfaces in 27 packages.

4. SELECTION OF TOOLS

Five tools were selected in this study. These were:

1. Understand for Java (UFJ) [4]
2. JDepend [3]
3. Structural Analysis for Java (SA4J) [1]
4. ARMIN (Architecture Reconstruction and MINing)
5. Enterprise Architect (EA) [2]

The following is a summary of the each tool's capabilities for architecture reconstruction and evaluation.

4.1 Understand for Java (UFJ)

One of the first steps in an architecture reconstruction is to extract information from the source code. Several source code extraction tools exist for Java, such as UFJ and jfx [18]. The tools vary in their level of functionality and output formats. In terms of output,

some tools generate Rigi Standard Format (RSF) or extensions of RSF, while others generate proprietary formats.

UFJ is a high quality commercial tool for reverse engineering Java code. We selected UFJ for the following reasons:

1) UFJ provides more than simple source information extraction. It also generates several comprehensive cross-reference data dictionaries supported by both web-based and desktop-based code navigation environments. These navigation tools are useful in helping the reconstruction team explore the code and understand the relationships between architecturally significant elements.

2) UFJ also provides object-oriented code metrics and other standard metrics, including LCOM (Percent Lack of Cohesion) DIT (Max Inheritance Tree) and CBO (Count of Coupled Classes). These metrics help give an overall measure of the code quality.

3) UFJ has an integrated static call graph viewer that depicts call dependencies between classes.

4) The output format of the source information is a proprietary text format. The format is well structured and amenable to Perl script manipulation into RSF.

4.2 JDepend

UFJ provided standard class level OO metrics, which were useful but not necessarily architecturally significant. Producing package level metrics with UFJ is possible but required additional effort. We selected JDepend because it is an open source tool which provides design metrics beyond traditional OO metrics at the package level. Among the many metrics supported, the following proved to be particularly useful for architecture reconstruction and evaluation:

Afferent Couplings (Ca) of Package A: The number of other packages that depend upon classes within package A. This is an indicator of package A's responsibility.

Efferent Couplings (Ce) of Package A: The number of other packages that the classes in package A depend upon. This is an indicator of package A's independence.

Abstractness (A) of Package A: The ratio of the number of abstract classes and interfaces in package A to the total number of classes and interfaces in package A.

Instability (I) of Package A: I = Ce/(Ce+Ca). This is an indicator of package A's relative *resilience to change.*

Distance from the Main Sequence (D): the perpendicular distance of a package from the idealized line A+I = 1. This is an indicator of the package's balance between abstractness and stability. In an ideal situation, if the package is almost completely abstract (A->1), it should be very stable (I -> 0). On the other hand, if the package is almost completely concrete (A->0), its instability (I->1) could be justified.

The use of these metrics in the reconstruction project is demonstrated in the section 5.

4.3 Structural Analysis For Java (SA4J)

SA4J is IBM's latest project for analysing Java dependencies. It analyses structural dependencies between java packages and measures the stability of both individual objects (classes, interfaces and packages) and the overall system. It helps to identify potential design problems as anti-patterns. SA4J reconstructs a series of architecturally significant diagrams automatically, for example, package dependency diagrams and class/interface dependencies diagrams.

SA4J calculates the overall stability of the system by first determining the potential average transitive impact AI (Average Impact) of each component on the rest of the system. The overall AI is then the average impact of each component.

Transitive impact analysis is different from the standard coupling measurement and appears to be more useful in architecture evaluation. This is because standard coupling metrics only capture the impact of changes to adjacent architectural elements. However, whether the impacts of a change have the potential to be contained or propagated can not be seen directly.

Based on the transitive change impacts, SA4J also provides the following design metrics and anti-patterns:

Global Butterfly: If the component is changed, it may affect many other components.

Global Breakable: The component is often affected if anything in the system is changed.

Global Hub: The component is both a global butterfly and a global breakable.

Skeleton: This layered view of the system is constructed by putting objects (class/interface/package) that do not depend on anything at the bottom of the visualization. The objects that are dependent on the lowest layer appear in the above layer, and so on. In this view, a stable system should have a normal pyramid shape. An unstable system may look like an upside down pyramid shape. The skeleton can be cross-referenced with other artifacts, like packages or inheritance trees.

We selected SA4J because it provides a unique set of useful design diagrams and metrics which are not available in any other tools.

4.4 ARMIN

ARMIN (Architecture Reconstruction and MINing) is a tool developed by the Software Engineering Institute and Robert Bosch Corporation. Use of ARMIN in architecture reconstruction has been reported in [26, 27] It is a highly configurable modeling and visualization tool, recognizing inputs in RSF, based on an element and relationship schema.

After data is imported, a scripting engine provides capabilities for further elements/relationships manipulation, such as grouping, sorting, collapsing and so on. The resulting views can be visualized in an aggregator window.

We selected ARMIN for the following reasons:

1) ARMIN can import any source information in RSF format, the de facto standard for source information extraction.

2) ARMIN is highly configurable in terms of manipulating elements and relationships. Its scripting engine can be programmed to produce any desired relationships and visualize them. Using this capability, ARMIN can produce unique architectural views which other tools are not capable of.

3) During data model abstraction, the mapping between levels can be retained in two ways:

- The logic of the mapping is reflected in the scripts which incorporate knowledge from software engineers and domain experts.
- The mapping can also be inspected in the visualized graph using a *drill down* feature.

4) ARMIN can incorporate architectural specific elements and concepts into the model, and is not bound by the information in the source code. The tool does not prevent the introduction of any new elements or relationships that can be potentially helpful in reconstruction and evaluation. These capabilities are demonstrated in section 5.

4.5 Enterprise Architect

Enterprise Architect (EA) is a widely used commercial UML modelling tool. It has useful features for capturing certain architectural views in UML notation. However, UML is somewhat limited when it comes to expressing architecturally significant information [14].

EA's current reverse engineering capabilities can only reverse engineer UML semantics such as class diagrams and associations [22]. Class diagrams are reverse engineered only within the context of their containing package. This means that constructing diagrams across multiple packages requires extra effort. Nevertheless, EA provides UML notation support and the capability to manually configure UML-based architectural views.

5. THE RECONSTRUCTION AND EVALUATION PROCESS

Initially, the reconstruction and evaluation were first performed by the authors based only on the source code and existing documentation. Then the findings were confirmed and clarified with the client. We expect this sequence of activities to be representative of many architecture reconstruction projects where the code base has been acquired, or in which the original architect may no longer be available.

We have loosely followed the QADSAR approach in conducting the architecture reconstruction by applying the approach iteratively. An iterative approach was more suitable given the exploratory nature of the project. QADSAR has the following five steps:

1) Scope Identification

2) Source Model Extraction

3) Source Model Abstraction and Visualization

4) Element and Property Instantiation

5) Quality Attribute Evaluation

1) Scope Identification:

The quality attributes of interest in the project were modifiability and reusability (which is sometimes considered as a special case of modifiability). To this end, metrics produced by JDepend and SA4J were used to identify the architectural subsystems and relationships which impacted on modifiability. As an example, an extract of the JDepend output is shown in Table 1:

Table 1. Top 5 and bottom 5 JDepend outputs (sorted by D column)

Package Name	Ca	Ce	A	I	D
au.com.xfinancial.database	14	4	0	0.2	0.78
au.com.xfinancial.product	5	7	1	0.6	0.51
au.com.xfinancial.sql.expr	5	5	0	0.5	0.5
au.com.xfinancial.company	6	7	0	0.5	0.46
au.com.xfinancial.sql	8	4	0	0.3	0.44
……	…	…	…	…	…
au.com.xfinancial.product.impl	2	10	0	0.8	0.06
au.com.xfinancial.tools.company	2	20	0	0.9	0.06
au.com.xfinancial.tools.report	1	19	0	1	0.05
au.com.xfinancial.structure	1	23	0	1	0.04
au.com.xfinancial.content	1	23	0	1	0.01

Ca: Afferent Coupling **Ce: Efferent Coupling**

A: Abstractness **I: Instability**

D: Distance from Main Sequence

These metrics show that certain packages (e.g. the tools related packages) are deemed unstable. However, as they contain concrete implementation classes, the instability may be justified. We discussed most of the high instability packages with the development team, and they confirmed that most were not intended for reuse. Consequently, these were not deemed problematic from a modifiability perspective.

The D metric indicates possible problems balancing instability and abstractness (potential reuse). For example, the *product* package is relatively unstable, but is highly abstracted, which means it may be heavily reused in the future.

In order to explore this measure, we constructed a set of views around this package and presented them to the development team. The package turned out to be undergoing extensive migration from a previous design. In order to accommodate compatibilities with existing packages, the *product* package had to maintain a high level of dependencies with other packages. When the migration is complete in near future, the package will be refactored into several components to improve its modifiability and reusability. Hence, deeper analysis showed that the *product* package is in its present state problematic from a modifiability and reusability perspective.

Another useful output for scoping is from the SA4J report, shown below in Table 2. The top global breakables, global butterflies and global hubs are presented along with other useful architectural level design metrics.

***"The overall stability of the system is 91%**. Highly stable systems are typically above 90%."*

Table 2. SA4J output excerpts

Global Breakable

Object	Number of Times Affected	Percent of Times Affected
Content.ViewPortfolio	146	50%
Content.ViewPortfolioProfile	146	50%
Content.ReportResult	139	48%
…	…	…

Global Butterfly

Object	Affected objects	Affected Percent of the System
XMLUtils	126	43%
Debuggable	125	43%
XMLSerializable	114	39%
…	…	…

Global Hub

Object	Number of Times Affected	Percent of Times Affected	Affected objects	Affected Percent of the System
HomeFactory	65	22%	40	13%
LicenceHome	65	22%	40	13%
HomeManufacturable	65	22%	40	13%
…	…	…	…	…

In summary, the design metrics and anti-patterns produced by both JDepend and SA4J led us to focus our attention on certain components in the architecture. However, these tools could not help us with further reconstruction and deeper analysis of these components with the customizable capabilities we need. Consequently, it was necessary to utilize more flexible tools like ARMIN to reconstruct the areas of interest.

2) Source Model Extraction:

Source model extraction was performed by using UFJ and Perl scripts. The elements/relationships we extracted and relationships with UFJ files are presented in Table 3.

The SEI has developed a collection of extensible scripts for converting UFJ formats to RSF, which can be input in to the ARMIN tool. In the original scripts, the *depends_on* relationship is used to represent all dependency relationships including imports, uses, cast, and so one. This seemed too coarse grain for our needs, so we modified the scripts to discern individual dependency relationships for use in separate views.

SA4J does not require a separate source model extraction step. It can take compiled class files in archives (.jar, .war, .zip and .ear) directly.

Table 3. Relationships between RSF information and UFJ files

Relationships/elements	UFJ files
defines_fn, class, function	.cmx, .dct
defines, file, class	.cmx
contains, file, function	.dct, .pux
has_member, class, member_variable	.dic
defines_var, function, local_variable	.obx
defines_global, file, global_variable	.obx
calls, function, function	.pux
depends_on, file, file	.tyx
defines_class, package, class	.dct

3) Source model abstraction and visualization:

Both EA and SA4J can depict package dependency diagrams (Figure 1) and class/interface dependency diagrams automatically. SA4J in particular has an excellent GUI and intelligent graphic layout.

This visualization depicts useful information on package dependencies. However, the package diagram is inflexible in two ways:

1) All packages are on the same level. Dependencies between package and sub-packages are not presented, (e.g. the *tools* package has several sub-packages which are depended on by other packages). By navigating and drilling down through the packages, these dependencies can be visualized, but the resulting views are less useful in a lower level context. Ideally, we would like to have had customized depictions that present dependencies across different abstraction levels in a single diagram.

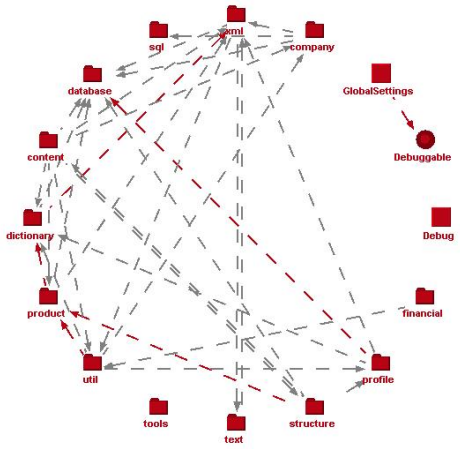

Figure 1. SA4J package dependency diagram

2) The precise nature of the dependencies can not be analyzed. A weight indicator represents the *heaviness* of the dependency. This

provides some insights in to the dependency, but it takes effort to manually navigate through diagrams to perform a deeper analysis.

It is clear that SA4J is limited in terms of customizing dependency diagrams to show grouping, and removing hand-picked elements that are not wanted by the architect in the view. Hence SA4J outputs become cluttered for visualizations with a large number of elements in certain diagrams. Because of this limitation, ARMIN was used to reconstruct customized views that contained only the packages and components that we were interested in from a modifiability perspective. Figure 2 shows ARMIN's dependency depiction after importing the RSF data produced from the raw UFJ information.

Here, different levels of packages are presented in the same diagram and dependencies are aggregated. For any aggregated relationships between two packages, we can drill down to see the detailed dependency listed along with the diagram, as shown in Figure 3.

As indicated in [25], one of the most important features of reverse engineering tools is to maintain the mappings between abstraction levels and to make them explicitly available to the user. For SA4J, although the mapping information is not lost, it is implicit in the program itself and can not be easily viewed and utilized.

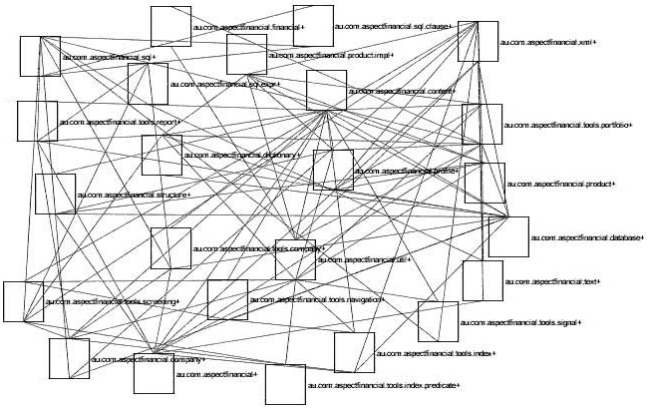

Figure 2. ARMIN package dependency diagram

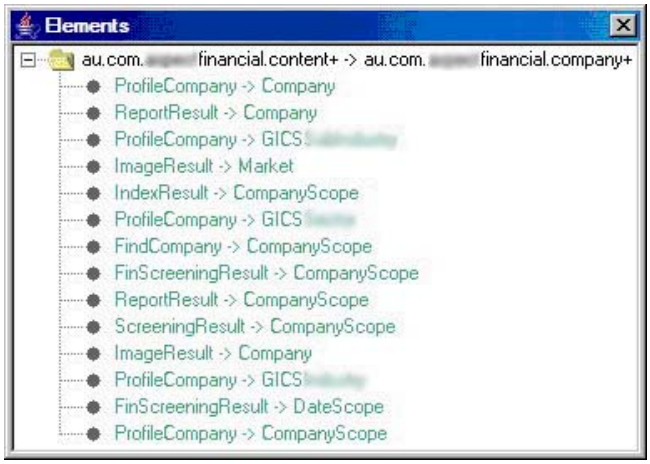

Figure 3. Drill down into an aggregated dependency

Next, we constructed views based upon one of the global breakable components (*ViewPortfolioProfile*). Initially, we used SA4J (Figure 4), as this view could be constructed by the tool with no additional effort on our behalf.

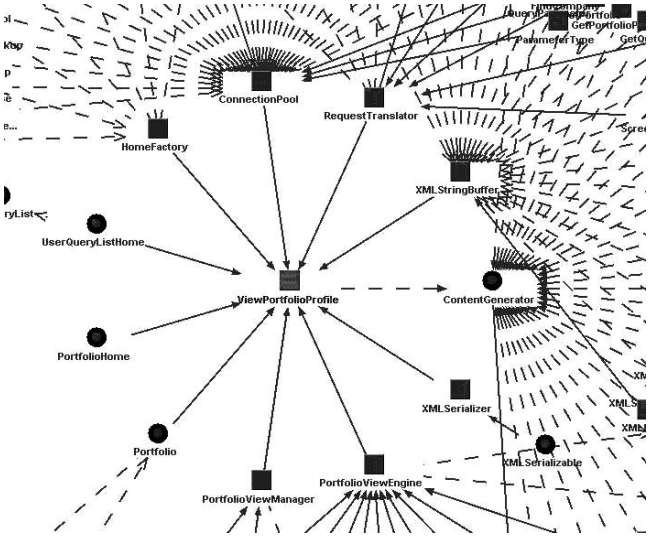

Figure 4. Dependency diagram for ViewPortfolioProfile (two degrees of separation of concern)

The SA4J diagram gives a clear view of the dependencies on closely related classes and interfaces. In Figure 4, the *two degrees of separation* option for *ViewPortfolioProfile* is enabled. This shows both the direct and indirect class and interface dependencies. Again however, the inability to customize this view and omit classes and interfaces that are not of interest architecturally diminishes its usefulness.

Another view for *ViewPortfolioProfile* can be easily generated using the call graph feature in UFJ (see Figure 5).

Figure 5. UFJ Call Graph for *ViewPortfolioProfile*

While these views are useful in code understanding, they are limited in expressing architecturally significant information. For example, in addition to the class dependencies and call graphs, we would like to see this class in the context of its enclosing package, along with dependencies on other packages. This would require the construction of an architectural view that spans different level of abstractions and includes manually selected design elements. Unfortunately, SA4J can not handle this kind of customization.

It is possible however to construct this view using ARMIN. By executing a script that selects only the classes and packages related to *ViewPortfolioProfile*. we are able to collapse the packages in the view, while still including dependent classes and interfaces in the same visualization (see Figure 6).

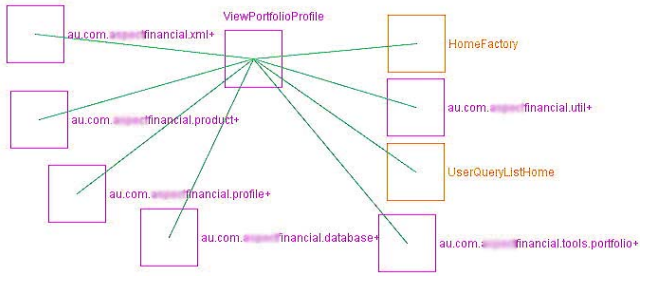

Figure 6. Class dependencies with packages

Again, the detailed relationships between design elements can be examined by drilling down on the dependency relationship.

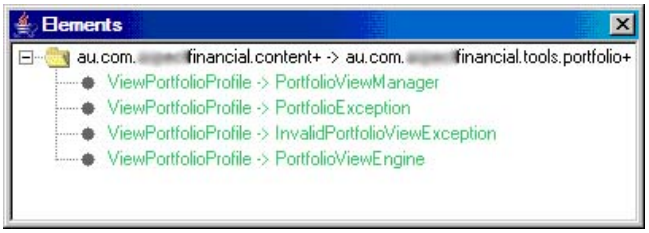

Figure 7. Drill down into an aggregated dependency

We also found the skeleton diagram from SAJ4 to produce interesting architectural information. Figure 8 shows a skeleton diagram that depicts the dependencies of the the *util* package.

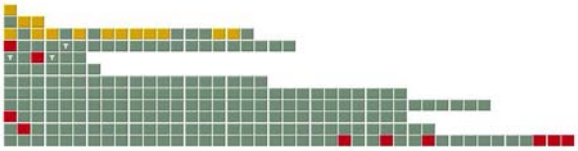

Figure 8. Skeleton diagram cross-referencing *util* package

The grey squares represent the classes and interfaces in the whole system. The red (black)[1] squares represent the classes/interfaces in the *util* package. The orange (light grey) squares represent the classes/interfaces that depend on the *util* (red) package.

Typically, more domain specific packages reside in the upper layers in the skeleton. As can be seen, two red (black) blocks (classes in the *util* package) do in fact reside in the upper layers. Closer inspection deteremined that these two classes are indeed domain specific. Discussion with the development team revealed that they reside in the same utility package for purely historic reasons. In future, they may be moved to other more domain specific packages.

Hence, by utilizing this view, we identified some interesting design irregularities without actually looking at the code. Traditionally, architecture reconstruction is based on recovering common architecture views [11]. In this case, novel visualizations beyond common architecture views have brought new insights to the understanding and evaluation of the system.

4) Element and Property Instantiation

The initial views we generated were closely related to Java language elements such as packages, classes, inter-object calls, and so on. This is mainly because these are the easiest for a tool to produce, since they are simply visualizations of the source code structure.

One of the important and difficult activities in architecture reconstruction and evaluation is to discover more abstract elements and properties in the architecture – ones which are not directly expressed in the source code of the application. Typical examples would be patterns utilized in the design or logically

[1] Colors in braces are for black and white prints of this document.

related components spanning across multiple implementation packages.

During the reconstruction, we noticed some design patterns, which had been *unconsciously* incorporated in to the application by the development team. The successful identification of these patterns can expedite architecture evaluation by leveraging reusable pattern information on possible positive and negative consequences [33]. The use of the patterns was subsequently confirmed in discussions with the development team.

We then used ARMIN to construct views that explicitly incorporated the pattern information into the architecture model. The identified patterns are Abstract Factory [13], Intercepting Filter [5] and View Helper [5].

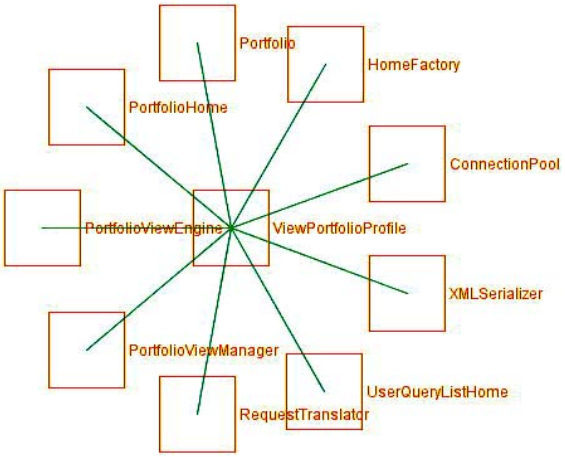

Figure 9. Class diagram for ViewPortfolioProfile

To achieve this, we had to introduce new architectural elements that do not explicitly exist in the extracted source information. ARMIN's extensible scripting facilities made this possible. The ARMIN script we used is shown below:

```
# Constructing design patterns
$factory={"PortfolioHome", "UserQueryListHome", "HomeFactory"};
collapse ($factory,/name="AbstractFactory",/graph="pattern");
$viewhelper = {"PortfolioViewEngine", "PortfolioViewManager"};
collapse ($viewhelper,/name="ViewHelper",/graph="pattern");
rename("Intercepting Filter:RequestTranslator","RequestTranslator");
$pattern.show()
```

The original view is presented in Figure 9. The new view with pattern information is shown in Figure 10.

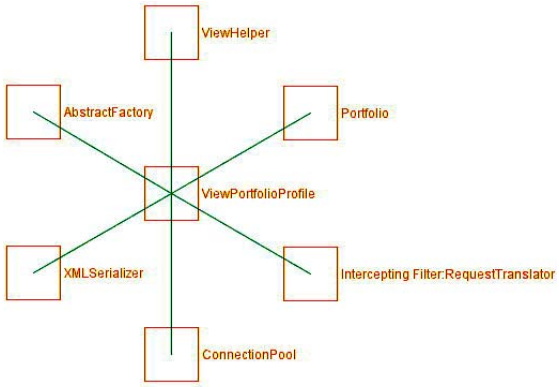

Figure 10. A diagram representing three design patterns

5) Quality Attribute Evaluation

By applying QADSAR approach iteratively and interactively, we conducted modifiability evaluations and constructed useful architectural views as the reconstruction progressed. In the process, we also collected specific modifiability and reusability scenarios from the client. These scenarios concern modifiability of particular packages and possible business requirement changes. Some of these scenarios inspired our preliminary findings. These new scenarios have been subsequently evaluated using the existing diagrams and as well as some newly generated ones.

6. SPECIFIC FINDINGS

Conducting this architecture reconstruction and evaluation has given us considerable insights in to various facets of the available tools and how they tackle the essential problems that must be dealt with during a reconstruction effort. The following discusses these insights.

1) *Abstraction and visualization tools should provide high levels of flexibility, while at the same time routinely-supporting commonly used architectural views.*

At one extreme, SA4J provides an excellent set of useful visualizations for assessing modifiability and reusability. However, these are not easy to customize. At the other extreme, ARMIN provides extremely flexible abstraction and visualization models, but does not support out of the box visualizations. ARMIN's scripting language is powerful enough to produce virtually any required views, but it takes additional effort to create the scripts. Although this may not be a problem for research purposes, it is likely to be an inhibiting factor for wider industrial use. A suggestion would be to pre-package useful scripts or even replace certain scripts with custom GUIs for generating common architectural views.

2) *Maintaining mapping information between abstraction levels is important.*

This has been considered one of the most important features of any reverse engineering tools [25]. Mapping between abstraction levels incorporates knowledge and design rationales that should be able to be inspected at any time. SA4J maintains these mappings implicitly. ARMIN, on the other hand, makes these mappings explicit and accessible to the user.

3) *Architectural level design metrics should be used to guide both reconstruction and evaluation.*

Architecture reconstruction and evaluation should be iterative and interactive. Scoping should go hand in hand with the inspection of high level design metrics and identification of anti-patterns. SA4J and JDepend provide an excellent range of metrics to guide this exploration. Many reverse engineering tools focus on increasing understanding of the system based on standard views of the software architecture. Without additional metrics, these views only provide limited visual cues on where to look for potential problems.

4) *UML is not sufficient for architecture reconstruction and evaluation.*

UML-based reconstruction is still immature [22]. But it has the potential to capture much architecturally significant information. We found EA's UML views useful initially, but the other tools provided much better capabilities for reconstruction. With profile extensions and dedicated tools, we envision that architecture reconstruction could eventually be conducted using solely UML notations.

5) *The tool environment needs XML-based self describing data.*

RSF has become the de facto standard for representing source information. It however needs a separate file to describe its data - this is known as a schema file in ARMIN. Even though the schema file can be reused, the linkage between a data file and a schema file and also the meaning of the schema file are not easy to maintain in a reconstruction environment. XML could provide a mechanism for self-describing RSF relationships. In addition, an XML-based graph exchange format called GXL has been proposed [17]. Only ARMIN of the tools in our review supports GXL. This lack of data exchange capability between tools has resulted in the additional effort of writing and extending ad hoc scripts to transform one tool's output to another tool's input. For example, ready-to-use views from SA4J can not be exported in such a format and be further customized in other tools.

6) *More work needs to be done on capturing dynamic information.*

Although research has been done in fusing dynamic views generated from profile tools with static views, they have not been successfully integrated into popular toolsets [19]. This remains a challenging and potentially highly productive area of research.

7) *Information representing middleware characteristics should be integrated into the reverse architecting tools.*

With more and more enterprise level software utilizing COTS middleware technologies and application frameworks, architecture views should capture middleware dependencies [12]. Making middleware/framework specific components explicit in a reconstruction environment gives a complete and rich view of an architecture. Pioneering work has been done to address this problem [21]. However, it has not been integrated into flexible environments like ARMIN to benefit from the other features in a unified fashion.

7. CONCLUSION AND FUTURE WORK

This project has successfully utilized a number of reverse engineering and architecture reconstruction tools to evaluate the modifiability of commercial product subsystem. By judiciously exploiting the metrics generation and visualization capabilities of these tools, we were able to construct numerous architectural views and identify problematic components from a modifiability perspective.

Our ultimate aim is assess technology support for *just-in-time* reconstruction. While we do not feel we are in a situation at this stage to make a definitive call, the findings from this project are encouraging, namely:

- Many tools automatically produce useful architectural metrics and views. These help guide the reconstruction scope by highlighting potentially problematic areas that require deeper analysis and richer architectural views.
- The scripting capabilities of ARMIN incur one-time setup costs, but once established, can be used in subsequent reconstructions with no additional effort

In total, after tool familiarization, we estimate this reconstruction effort took approximately 3 person days effort. This is reasonably efficient, and could no doubt be reduced on subsequent iterations. Of course, the code base we used was only of moderate size, homogenous in terms of programming language and was fundamentally well designed.

We therefore intend to further investigate the feasibility of this approach on larger, more complex code bases. This will give greater insights in to the scalability of the available tools to reconstruct complex software architectures in an efficient and effective manner.

8. ACKNOWLEDGMENTS

The authors are employed by National ICT Australia, which is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

9. REFERENCES

[1] alphaWorks: Structural Analysis for Java (Updated 31/03/2004, fix pack 1), 2004.
[2] Enterprise Architect (V4.1).
[3] JDepend (V2.7), 2004.
[4] STI: Understand for Java (V1.4), 2004.
[5] Alur, D., Crupi, J. and Malks, D. *Core J2EE patterns : best practices and design strategies*. Prentice Hall PTR, Upper Saddle River, NJ, 2003.
[6] Armstrong, M.N. and Trudeau, C., Evaluating architectural extractors. in *Fifth Working Conference on Reverse Engineering (WCRE)*, (1998), 30-39.
[7] Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice*. Addison-Wesley, 2003.
[8] Bellay, B. and Gall, H., A comparison of four reverse engineering tools. in *Fourth Working Conference on Reverse Engineering (WCRE)*, (1997), 2-11.
[9] Bengtsson, P., Lassing, N., Bosch, J. and Vliet, H.v. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, *69(1-2)*. 129-147.
[10] Bosch, J., Software architecture: the next step. in *First European Workshop on Software Architecture (EWSA)*, (2004), Springer.
[11] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. *Documenting Software Architectures : Views and Beyond*. Addison-Wesley, 2003.
[12] Emmerich, W., Software engineering and middleware: a roadmap. in *22nd International Conference on on Software*

Engineering (ICSE), Future of Software Engineering Track, (2000).

[13] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995.

[14] Garlan, D. and Kompanek, A., Reconciling the Needs of Architectural Description with Object-Modeling Notations. in *Unified Modelling Language (UML)*, (2000).

[15] Girard, J.-F. and Koschke, R., Finding components in a hierarchy of modules: a step towards architectural understanding. in *International Conference on Software Maintenance (ICSM)*, (1997), 58-65.

[16] Gorton, I. and Haack, J., Architecting in the Face of Uncertainty: An Experience Report. in *26th International Conference on Software Engineering (ICSE)*, (Edinburgh, United Kingdom, 2004), IEEE, 543-551.

[17] Holt, R.C., Winter, A. and Schurr, A., GXL: toward a standard exchange format. in *Seventh Working Conference on Reverse Engineering (WCRE)*, (2000), 162-171.

[18] Kaastra, M.D. and Kapser, C.J. Toward a semantically complete Java fact extractor, Department of Computer Science, University of Waterloo, 2003.

[19] Kazman, R. and Carriere, S.J., View Extraction and View Fusion in Architectural Understanding. in *Fifth International Conference on Software Reuse (ICSR)*, (1998).

[20] Kazman, R., O'Brien, L. and Verhoef, C. Architecture Reconstruction Guidelines, Third Edition, Software Engineering Institute, Carnegie Mellon University, 2003.

[21] Knodel, J. and Pinzger, M., Improving fact extraction of framework-based software systems. in *10th Working Conference on Reverse Engineering (WCRE)*, (2003), 186-195.

[22] Kollmann, R., Selonen, P., Stroulia, E., Systa, T. and Zundorf, A., A study on the current state of the art in tool-supported UML-based static reverse engineering. in *Ninth Working Conference on Reverse Engineering (WCRE)*, (2002), 22-32.

[23] Kruchten, P., Hilliard, R., Kazman, R., Kozaczynski, W., Obbink, H. and Ran, A. The SARA report (Software Architecture Review and Assessment), 2002.

[24] Lassing, N., Bengtsson, P., Bosch, J. and Vliet, H.V. Experience with ALMA: Architecture-Level Modifiability Analysis. *Journal of Systems and Software, 61* (1). 47-57.

[25] Müller, H.A., Jahnke, J.H., Smith, D.B., Storey, M.-A.D., Tilley, S.R. and Wong, K., Reverse Engineering: a Roadmap. in *22nd International Conference on on Software Engineering (ICSE), Future of Software Engineering Track*, (2000).

[26] O'Brien, L. Architecture Reconstruction to Support a Product Line Effort: Case Study, Software Engineering Institute, Carnegie Mellon University, 2001.

[27] O'Brien, L. and Tamarree, V. Architecture Reconstruction of J2EE Applications: Generating Views from the Module Viewtype, Software Engineering Institute, Carnegie Mellon University, 2003.

[28] Stoermer, C., O'Brien, L. and Verhoef, C., Architectural Views through Collapsing Strategies. in *12th IEEE International Workshop on Program Comprehension (IWPC)*, (2004), 100-110.

[29] Stoermer, C., O'Brien, L. and Verhoef, C., Moving towards quality attribute driven software architecture reconstruction. in *10th Working Conference on Reverse Engineering (WCRE)*, (2003), 46-56.

[30] Stoermer, C., O'Brien, L. and Verhoef, C., Practice patterns for architecture reconstruction. in *Ninth Working Conference on Reverse Engineering (WCRE)*, (2002), 151-160.

[31] van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L. and Riva, C., Symphony: view-driven software architecture reconstruction. in *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA)*, (2004), 122-132.

[32] Yan, H., Garlan, D., Schmerl, B., Aldrich, J. and Kazman, R., DiscoTect: a system for discovering architectures from running systems. in *26th International Conference on Software Engineering (ICSE)*, (2004), 470-479.

[33] Zhu, L., Ali Babar, M. and Jeffery, R., Mining Patterns to Support Software Architecture Evaluation. in *4th Working IEEE /IFIP Conference on Software Architecture (WICSA)*, (2004), IEEE.