External Dependencies-driven Architecture Discovery and Analysis of Implemented Systems

DHARMALINGAM GANESAN, MIKAEL LINDVALL
Fraunhofer Center for Experimental Software Engineering (CESE), College Park, Maryland, USA

MONICA RON
Honeywell, Greenbelt, Maryland, USA

A method for architecture discovery and analysis of implemented systems (AIS) is disclosed. The premise of the method is that architecture decisions are inspired and influenced by the external entities that the software system makes use of. Examples of such external entities are COTS components, frameworks, and ultimately even the programming language itself and its libraries. Traces of these architecture decisions can thus be found in the implemented software and is manifested in the way software systems use such external entities. While this fact is often ignored in contemporary reverse engineering methods, the AIS method actively leverages and makes use of the dependencies to external entities as a starting point for the architecture discovery. The AIS method is demonstrated using the NASA's Space Network Access System (SNAS). The results show that, with abundant evidence, the method offers reusable and repeatable guidelines for discovering the architecture and locating potential risks (e.g. low testability, decreased performance) that are hidden deep in the implementation. The analysis is conducted by using external dependencies to identify, classify and review a minimal set of key source code files. Given the benefits of analyzing external dependencies as a way to discover architectures, it is argued that external dependencies deserve to be treated as first-class citizens during reverse engineering. The current structure of a knowledge base of external entities and analysis questions with strategies for getting answers is also discussed.

Categories and Subject Descriptors: D 2.11 **[Software Engineering]**: Software Architectures; D.2.7 **[Software Engineering]**: Distribution, Maintenance, and Enhancement - *Documentation; restructuring, reverse engineering, and reengineering*; D.3.2 **[Programming Languages]**: Language Classifications—C, C++, Java

General Terms: Documentation, Software Architecture, Reverse Engineering, Quality

Additional Key Words and Phrases: Concerns, Performance, Testability, External Entities

1. INTRODUCTION

The software architecture of a system to be built can be analyzed using state-of-the-art architecture analysis methods [Krutchen et al. 2006, Shaw and Clements 2006]. These analysis methods can be used to identify potential software-related risks such as low maintainability and low performance. Once the risks have been identified, they can be mitigated, for example, by choosing alternative architectural styles early in the software lifecycle [Clements et al. 1995]. However, there are a large number of software systems in use that were developed either before the invention of these state-of-the-art architecture analysis methods or lacked an explicit software architecture design phase in the first place. Even for systems that were developed using existing architecture analysis methods, most of us will agree that the implemented architecture will most likely deviate from the specified architecture after a couple of iterations, making the documented architecture out of date and difficult to use [Krikhaar 1999, Murphy et al. 2001, Lindvall et al. 2010]. Thus, for many existing software systems, there are no trustable and up-to-date artifacts but the source code. In situations when the original developers are gone and the documentation is either outdated or non-existent there is a practical need to reverse

engineer the architecture from the implementation so that we can identify potential risks and offer practical strategies to mitigate them.

Two authors of this paper work for CESE, which is an organization that analyzes customers' existing software systems, for example in order to detect risks. The customers of CESE want an "independent eye" to look into their implemented software systems, evaluate the implemented architecture, identify high risk areas, and propose practical suggestions for improvements and risk mitigations. Naturally, the customers expect the analysis results to be delivered "as soon as possible" so that they can effectively make use of the findings in their decision making process, incorporate improvements into their products and processes, remove issues, and meet their goal to produce a high quality software product on time. With this pressure to deliver critical and accurate architecture insights regarding previously unfamiliar software systems in relatively short time, CESE is always seeking ways to improve and make their analysis methods more efficient. The AIS method, introduced in this article, is the result of more than 10 years of such analysis and accompanying improvements.

One of the more fundamental insights that we have gained through our work is the importance of being able to zoom in on and reason about individual software concerns and how they are implemented in the source code. Modern (as well as many not so modern) software systems are no-doubt large and inherently complex. Apart from offering features, software systems manage multi-dimensional concerns, such as OS variants, configuration parameters and settings, database interactions, remote connections, inter-process communication, dynamic reconfiguration and updates, licensing, security, error handling, internalization, etc. In the software systems we have analyzed, any given source file[1] typically addresses more than one concern and each concern is typically distributed across more than one source file. For example, a source file may contain code that executes a database SQL query as well as code that writes each database interaction, as well as other events, to a log file. Thus, the code in this file addresses several concerns (i.e. database management and logging). In addition, several source files may contain code that is involved in various kinds of database interactions. Thus, code that address the database interaction concern is spread across several source files.

We can conceptually imagine every source code file as being one point in a multi-dimensional space, where each dimension refers to a concern. Most readers will agree that it is beyond our capability to comprehend and visualize shapes in more than two or three dimensions. Hence, we need to build abstractions of the software under study that emphasize only the concerns we are interested in and suppress (for a moment) everything else. Since the software under study is typically represented by source code only, we need to create these abstractions using entities found in that source code. Thus, we can say that we need to identify selected implementation concepts in order to recognize the implementation of architecture concepts such as layers, styles, components, and connectors that are typically used to express the high level software architecture and which are often "hidden" in the multi-dimensional space of concerns in the source code.

Existing reverse engineering tools typically build abstractions using the directory structure (or similar hierarchies that organizes the source code) and code relations (e.g. calls, data accesses, and includes). Code level relations are lifted by composing them with

[1] We use the term "file" or "source file" to denote any file that contains computer instructions that can be compiled.

the hierarchy structure. This lifted relation is basically a two-dimensional hierarchical view of the system. Fraunhofer's SAVE tool is an example of a tool that works this way and its usefulness has been demonstrated in many different projects [Lindvall et al. 2010]. However, given that the implementation usually handles so many concerns, it is not surprising that the projection from multi-dimensions to a two dimension hierarchical graph looks like "spaghetti". In several endeavors, we realized that hierarchical dependency graphs alone do not convey the true architectural story of an unfamiliar system, simply because there are too many concerns in the reverse engineered model. Informally, it was not easy to see one or more "Lasagna" hidden inside the "Spaghetti". Using tools that produce two-dimensional hierarchical views, we had to work around the problem by manually creating views based on certain implementation entities of the software under study, which was sometimes very time consuming. In addition, we could not pin-point testability and performance risks by only analyzing the dependency graphs because some of the necessary information was missing.

We have also observed that analysts (ourselves included) applying the SAVE tool tend to ignore dependencies to external entities such as programming language libraries, COTS, Frameworks etc. The reason is that, in general, around 40-50% of function calls (and other dependencies such as include/import) are to external entities and these dependencies can easily affect the performance of the tool because the more dependencies the tool has to process, the longer the processing time. In addition, the more the dependencies the more they clutter the diagrams. It is not only users of the SAVE tool that ignore external dependencies, but also users of other reverse engineering methods and clustering tools. The reason is typically to minimize the size of the dependency model of large systems or to avoid the influence of external entities in the search for "good" clusters. The justification for doing so is the common misconception that external dependencies are not "interesting" and less important than internal dependencies between software components that are part of the software under study. However, ignoring such external dependencies comes at a significant cost. Once the external dependencies have been removed from the extracted dependency model, the analyst cannot easily tell, for example, whether the system is distributed (based on multiple processes in one or more machines) or stand-alone, because the dependencies to the underlying libraries for Inter Process Communication (IPC) were removed in the extracted code relations. As a consequence, the analysts do not really get good architectural insights by analyzing the remaining dependencies in the dependency model. This is because several concerns (e.g. GUI, Persistence, and Security) are implemented using the support of external entities, which were often removed in the extracted code relations.

Thus, the premise of the AIS method is that dependencies to the very same external entities that were used to build the system can also be used to efficiently discover its implemented architecture focusing on individual concerns as necessary. This will lead to the discovery of potential risks hidden in the implementation, and can be achieved by reviewing a limited number of files.

This article will demonstrate a) how we can efficiently discover architectures by using external dependencies coupled with knowledge about the semantics of such external dependencies stored in a knowledge base, and b) how we can use external dependencies to slice the implementation for individual concerns in order to gain deep concern-specific architectural insights and potential risks.

The proposed method is based on the fact that much of the critical information about an existing software system are stored in source files, and thus an analyst has to review

such files in order to understand critical parts. For a small system, it is not a problem analyzing each and every file of the system. However, for larger systems there are typically too many source files (10,000 files is not unusual) for the analyst to review. Thus, we need a way to identify the most important parts of the source code for review. We have discovered that external dependencies can help us identify the parts of the source code that are most important – for the task at hand. Often the external dependency is based on a file name as well as a function name of the external entity. We have classified many of the commonly occurring external files and functions in such a way that we can select a perspective or category and can trace back to the files and functions in the software under analysis that use them. Thus, by reviewing only those specific parts of the source code, we can understand how a specific concern is handled by the system. This technique also allows us to reason about other parts of the system that have similar dependencies and we can draw conclusions about large portions of the source code without having to review it all.

The acquired knowledge and insights of analyzing software systems are packaged into a knowledge base, which on the one hand is used to analyze new systems more efficiently. On the other hand, it is used to improve our understanding of various real-world solutions to architectural challenges (e.g. how to architect database aspects for a huge volume of transactions). These arrays of solutions are also discussed with our customers as alternative solutions to their architectural problems, if any.

Using the AIS method on the relatively large (~600 KLOC) NASA Space Network Access System (SNAS) system, the independent analyst, discovered several architectural insights by reviewing less than 4% of the 1578 source files. Examples of architectural insights are a) that the implemented architecture of the SNAS is based on a distributed client-server architectural style, b) that the distributed subsystems exchange data by sending and receiving objects using the transfer object design pattern [Alur et al. 2003], c) that each subsystem of the SNAS has a dedicated layer for handling the persistence concern, and d) that the GUI subsystem is based on an event-driven architecture.

In addition, with the help of external dependencies, several architecturally relevant performance related constructs were discovered including the usage of a) a database connection pool design pattern in order to overcome the performance overhead of frequently creating and deleting database connections [Apache DBCP], b) the reactor design pattern in order to reduce the overhead of frequently creating and deleting threads for each client connection in a client-server architectural style [Schmidt 1995]. Some testability problems due to a weak separation of GUI concepts with core logic were also discovered as well as some performance risks due to threading models. The analysis, detected problems, potential risks as well as concrete solutions and risk mitigation strategies were reported to the SNAS team and are discussed in this article.

Software Engineering Contributions: We hope that this paper makes the following novel software engineering contributions:

- A practically inspired and validated method to discover software architectures from implementations using external dependencies.
- An architecture-centric framework for evaluating quality of implemented systems without reviewing each individual file.
- Several concrete real-world code snippets to demonstrate the true meaning of software architecture concepts such as abstraction, separation of concerns, and design for testability and performance.

2. The AIS Method

The goals of the AIS method are to support the analyst in a) discovering the software architecture from the implementation, and b) evaluating the quality of the implementation using the discovered architecture.

The goal is achieved by exploiting the dependencies on external entities for the following reasons. First, frameworks (e.g. CORBA, EJB, and Hibernate) define the reference architecture(s). Second, software connectors (e.g. middleware, sockets, queues, and shared memory) and concerns (e.g. GUI, Persistence, Security, and licensing) are usually built using external entities. Thus, the analyst could get a good overview of the potential architecture concepts implemented in the system. Third, the analyst can classify the code and assign responsibilities to the system under analysis by tracing back to the internal entities that use external entities. Fourth, even if no documentation of the software under study is available, external entities are often well-documented on vendors' websites, and there are usually several example programs, user manuals, discussion forums, blogs, etc. devoted to external entities facilitating understanding the external entity, which can be used to understand the software under study. The analyst can use such freely available resources to better understand the purposes of external entities with little or no domain knowledge of the system under analysis. Furthermore, this knowledge becomes a reusable asset for the analyst because he/she can reuse the knowledge in the architecture analysis of other systems. Because of these novel benefits, the AIS method treats external entities as first class elements for architectural analyses, instead of simply excluding them as often the case in many reverse engineering approaches.

Now, we will explain how to use the dependencies to external entities for a) discovering the architecture concepts such as Layers, Styles, Design Patterns, Components, Connectors and Interfaces, and b) discovering the concerns in the implementation which can be used to slice the system and offer concern-specific architectural insights.

2.1. Working in Four Dimensions

The basic model of software systems our approach is built upon is based on the following observations: Most systems are based on a collection of components (a.k.a. modules). Those components are typically represented by source code stored in files organized in folders. Components that form part of the same executable can use simple function calls, whereas components in different executables must use other mechanisms such as shared memory, shared files, remote procedure calls, and socket communication. Some commonly occurring components are the GUI and the Database Management System (DBMS). Often one or more code libraries and code frameworks are used, for example to bridge the gap between the DBMS and the application.

Having this basic model of software systems in mind, one of the analyst's first goals is to understand what these components are, how they are organized in terms of files and folders, how they are related and share source code, and how they communicate with each other. In our approach, we use implementation concepts to discover facts related to such components. The approach uses the *directory structure* because it often helps in identifying components of the system. For example, the directory structure can reveal how components are organized as files and folders.

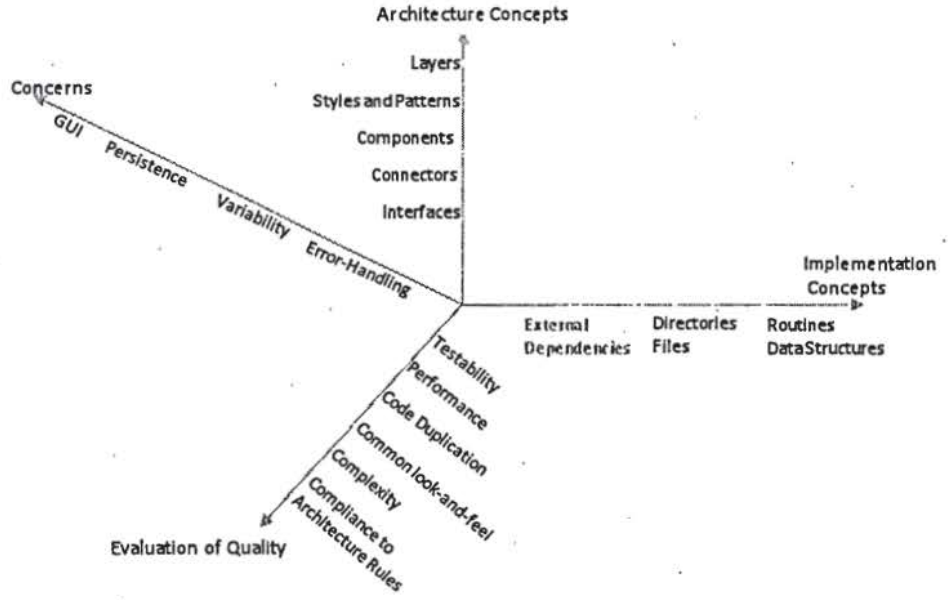

Figure 1 Conceptual dimensions of the AIS Method. Concerns are identified using dependencies on external entities. The software architecture of an existing system is incrementally discovered by the slicing the system using one concern at a time. The listed items on each axis are samples, and not a complete list.

Once the components and executables are understood, the next step is often to understand how they communicate using software connectors (e.g. using sockets, pipes, or queues), which can be accomplished using external dependencies. For instance, the use of the header file `socket.h` suggests that components of the system communicate using socket channels. Using this external dependency, an analyst can trace back and locate all files that are involved in socket communication, review them, and determine how the sockets are used. In several cases, we have also discovered layers related to communication just by tracing dependencies to `socket.h`. More specifically, by analyzing the extracted code relations the analyst quickly found that almost all usages of the functions declared in `socket.h` were only using intermediate wrapper functions that build upon primitive socket functions and offer a higher-level abstraction (i.e. hiding the details of a specific connector) to the rest of the system. In addition, dependencies to `socket.h` were used to discover the components that are involved in the socket communication, resulting in the discovery of a high-level component-connector view, as described by [Shaw and Garlan 1996].

Once the component-connector view has been established, the next step is often to understand other important concerns such as how the software under study handles OS Variants or how it handles Persistence. To this end, the AIS method proposes to slice the implementation based on concerns using dependencies to external entities. For example, if the system uses both the `CreateThread` C function for Windows and the corresponding C function `pthread_create` for UNIX, then these external dependencies indicate that the implementation manages several OS variants. Using our approach, the analyst follows these external dependencies and identifies the source files that use these functions, and by reviewing the code determine how the OS variants are architected in

the implementation. This might lead the analyst to discover an OS Abstraction Layer (OSAL) that, for instance, offers an abstract interface and alternative implementations for different types of OS.

In our experiences with several commercial systems, we have found that external dependencies help in producing a valuable list of concerns that are implemented in the system, which the analyst can use to slice the system. This slicing by concerns allows the architecture to be discovered in an incremental fashion focusing on one concern at a time. Due to this concern-based slicing of the implementation, in many cases, the amount of source code the analyst has to review is significantly reduced. Furthermore, the analyst can now zoom-in to the details where devils usually hide and can reveal quality issues such as testability and performance risks. Typically, a vast majority of the implemented system's source code can be covered if we analyze concerns including a) GUI, b) Persistence, c) Variability of the OS, and d) Error or Exception Handling.

Once a basic understanding of the system has been established, the analyst can proceed with the second goal, which is to evaluate implementation quality. Typical evaluation perspectives that we repeatedly follow, because our customers found them informative and useful for decision making are testability, performance, common look-and-feel, code duplication, complexity analysis, and compliance to architectural rules, see the "Evaluation of Quality" Axis in Figure 1. It is worth noting that on the one hand, the analyst can use the discovered architecture to illustrate potential issues (e.g. Performance risks due to threading models for events notification). On the other hand, the architecture discovery activity can be influenced by the actual need to construct a special slice in order to demonstrate a specific issue. The central idea is to evaluate quality using the discovered architecture by focusing on one concern at a time.

One evaluation perspective is *testability*, which is evaluated by focusing on one concern at a time. For example, testability can be evaluated with respect to the persistence concern. That is, to answer the question: can the system's core logic be tested without the database being up and running? On the one hand, the analyst can use the discovered architecture to show that it is impossible to test the system without the database. On the other hand, the architecture discovery activity is also influenced by the need to evaluate testability, meaning that the analyst should slice the system in such a way that he can show evidence to the members of the project team why the system is not possible to run and test without the database. Similarly, the analyst can evaluate testability with respect to other aspects, for example the GUI. That is, to answer the question: can the system's core logic be tested without the GUI?

Another evaluation perspective is *performance*, which is also evaluated by focusing on one concern at a time. For example, the analyst can take the persistence concern and evaluate the performance of the database due to the style the implemented architecture uses for database connections. Similarly, the analyst can focus on the GUI concern and evaluate how the event listeners and dispatchers might impact the GUI performance. In the AIS method, performance evaluation is conducted at an architectural-level, meaning that the analyst focuses on high-level principles that influence the whole system. For example, the threading model used by the implementation in order to read incoming data from a socket and dispatch data to data processors can be considered architecturally significant because if the same thread is used to read from the socket and synchronously dispatched to data processors, then there is a risk that low performing data processors might affect the rest of the system.

A third evaluation perspective is *common look-and-feel*. The purpose of the common look-and-feel is to evaluate how different parts of the system implement the same concern. For example, the analyst can focus on the persistence concern, which was located above, and evaluate whether or not all modules use the database in the same way, unless there is a need for differences. Another example is if the system is based on a publisher-subscribe architectural style, then the analyst can analyze whether or not all publishers send messages in the same way and all subscribers receive messages in the same way, unless there is a need for differences. Good common look-and-feel is an aesthetic property that helps programmers and new-comers to easily understand different parts of the system.

The purpose of the *code duplication or clone analysis* is to understand how the architecture abstracts commonality and manages variability. To achieve this, the analyst interprets the collected clone data within a context of a concern using the discovered architecture. For example, the analyst offers insights on code clones due to the persistence concern.

The purpose of the *complexity analysis* is to understand and evaluate how complexity is managed for each concern. For example, the analyst uses the discovered component-connector view to analyze the complexity of transferring data from one end of the communication channel to the other.

The purpose of evaluating *compliance to architectural rules* is to determine whether or not the specified architecture is consistent with the actual (i.e. the implementation) built architecture. If the existing documentation specifies that the interaction to a hardware port should be only via the specified software interfaces of the hardware abstraction layer, the goal of the verification is to check whether there are deviations to this specification. We observed that by analyzing one concern at a time, verification of architectural rules becomes focused and detected deviations are clearly explainable to the development team [Ganesan et al. 2009]. In addition, the analyst was able to discover undocumented architectural rules, for example, from the discovered architecture showing how the COTS for logging is used by the implementation the analyst was able to identify some code elements that by-pass the logging wrapper.

2.2. Choosing the Subspaces of the Four Dimensional Space

Although the AIS method has four conceptual dimensions for architecture discovery and evaluation of quality, the analyst can choose certain subspaces of interest based on his goals. For example, if the goal is to evaluate testability then the analyst can work on the subspaces that include testability. That is, the analyst can focus on discovering architectural information and create matching diagrams that can help in reasoning about testability with respect to several concerns (e.g. GUI, Persistence). Similarly, if the goal is to discover and document how the implementation handles the persistence concern, the analyst can work within those subspaces that include persistence, and need not traverse the "Evaluation of Quality" dimension. Basically, it is up to the analyst to decide which subspace is of interest. The method does not enforce the order in which the analyst should proceed in choosing the subspaces. Thus, the method offers flexibility to the analyst, meaning that the analyst can incrementally cover the conceptual space of our method based on goals and available effort.

Now, we introduce the knowledge base that facilitates architecture discovery and analyses in a repeatable and reusable way.

2.3. The Knowledge Base for Architecture Discovery and Analysis

We realized that even though the dependencies to external entities are very useful to discover software architectures from an implementation, there were additional challenges that analysts typically faced including a) the obvious need to remember significant function names and header files of programming language libraries, COTS, and Frameworks, b) how to actually use the dependencies to external entities in constructing the architectural story and pin-pointing potential risks in the implementation. In order to address these challenges, we developed a knowledge base that supports the analysts in architecture analysis endeavors.

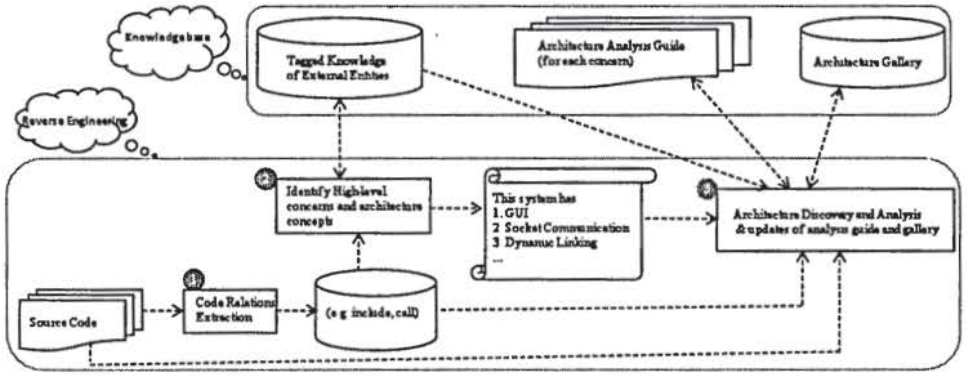

Figure 2 The major Steps of the AIS method, showing the integration of the knowledge-base with reverse engineering. Arrows denote data flow. Code relations (also called dependency models) are extracted automatically and stored as binary relations [Chen 1990].

2.3.1. Tagged Knowledge of External Entities

Given the source code, our approach discovers potential architecture concepts and concerns that are implemented in the system using its external dependencies. To achieve this, we created a classification and tagged the external entities that were used by the systems we analyzed. For example, Hibernate is a framework for accessing databases, and `org.hibernate.Query` is used for querying databases [Hibernate]. Thus `org.hibernate.Query` is tagged as being an indicator of code that queries databases. When our approach detects that the software under study makes use of (is dependent on) `org.hibernate.Query` then it concludes that the software under study has a database concern. Similarly, other concerns could be listed based on the knowledge of external entities. For a person new to the system under analysis, this list is a good introduction to potential concerns and architecture concepts that are implemented. Figure 4 shows a snippet of the knowledge base drawn automatically by the Perfuse tool using a XML representation of the knowledge base. We have also stored the tagged knowledge in a relational model, allowing the analyst to formulate queries such as: "List all files that depend on the Hibernate framework". We use the RPA language to formulate queries on extracted code relations [Feijs et al. 1998].

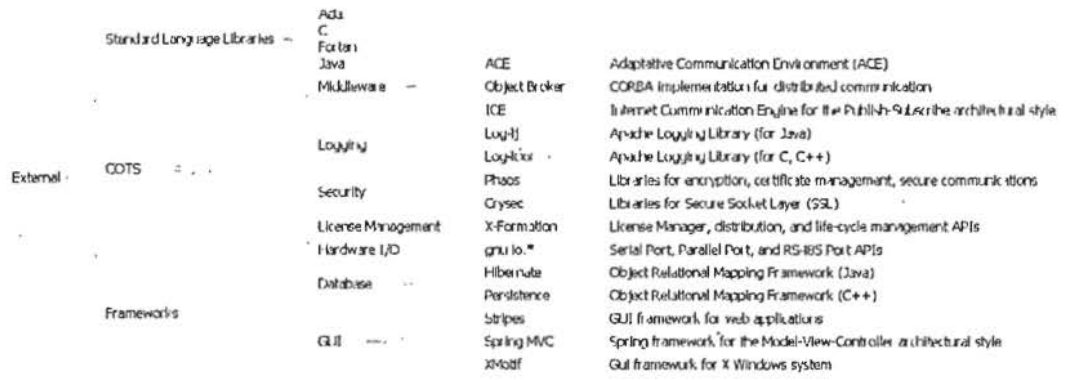

Figure 3 A snippet of the knowledge base showing some COTS and Frameworks. The knowledge base was tagged manually (e.g. Security is a concern if the system uses Phaos or Crysec COTS). Scripts could use the dependencies to external entities and print high-level facts of the system (e.g. The System uses the ACE Middleware and the Persistence framework for database interactions).

We also refined the high-level knowledge base shown in Figure 3 to the next more detailed level allowing the analyst to drill down to the functions and features of COTS and Frameworks that are actually used. On the one hand, this knowledge helps the analyst to learn relevant portions of new and unfamiliar technologies. On the other hand, it helps the analyst to obtain concrete architecture insights based on the implementation. For example, Figure 4 shows the next level decomposition of knowledge related to the ACE middleware [Schmidt 95]. Using the refined knowledge, scripts can detect where the connection to a remote peer is established (see `SOCK_Connector.h` in Figure 4).

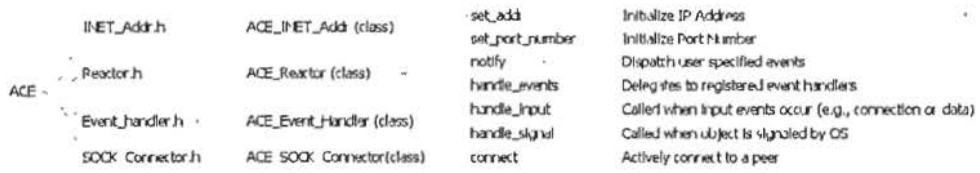

Figure 4 A snippet of the tagged knowledge base for the ACE middleware COTS. Scripts could use this knowledge and identify, for example, code elements that are involved in connecting to a remote peer by using the `connect` method of the `ACE_SOCK_Connector` class.

Figure 5 shows excerpts of the tagged knowledge base of the C language library on UNIX, respectively. It is clear that architecturally relevant concepts, such as Communication, Concurrency, Dynamic Linking, and Error Handling are already supported by the language libraries. Hence, we can take advantage of them for architecture discovery. Similarly, a knowledge base is also constructed for different OS types, such as Windows, Rtems, and Vxworks. Based on this knowledge, scripts can determine the various OS platforms that are supported by the system-under-analysis. Furthermore, the analyst can slice the system in order to analyze how the system handles OS variants.

			mqueue.h	mq_open	Open the queue connection
				mq_receive	Receive data from the queue
				mq_send	Send data to the queue
				mq_close	Close the queue connection
			shm.h	shmat	Attach the shared memory
		Communication		shmdt	Detach shared memory segment
				shmget	Get shared memory segment
			socket.h	accept	Accept a new connection on a socket
				connect	Connect a socket
				listen	Listen for socket connections
				recvfrom	Receive a message from a socket
				sendto	Send a message on a socket
				shm_open	Open a shared memory object
			mman.h	mlock	Lock a range of process address space
				munlock	Unlock a range of process address space
OS	Unix			mmap	Map pages of memory to a file or shared memory object
				mprotect	Remove a shared memory object
		Concurrency	pthread.h	pthread_cancel	Cancel execution of a thread
				pthread_create	Create a new thread
			semaphore.h	sem_init	Initialize a semaphore
				dlopen	Dynamically gain access to an executable object file
		Dynamic Linking	dlfcn.h	dlsym	Obtain the address of a symbol from an object file
				dlclose	Close a object file
				ENETDOWN	Network is down
			errno.h	ETIMEDOUT	Connection timed out
				ENETRESET	Connection aborted by network
		Error Handling		alarm	Generates a alarm signal
		...	signal.h	raise	Raise a signal

Figure 5 A snippet of the knowledge base for the standard C libraries (under UNIX). Scripts could, for example, highlight that the system under analysis may have a dynamic reconfiguration capability if it uses `dlopen`, `dlsym`, and `dlclose` functions.

2.3.2. Architecture Analysis Guide

The architecture analysis guide is another building block of the knowledge base. For each concern, we prepared a list of questions that the analyst may want to follow in order to discover architectural insights efficiently. In addition, it contains step-by-step advice for getting answers to them. For example, without reviewing say a few hundred files, how can the analyst conclude that these files are equivalent or of the same category? The analysis guide offers advice on what types of evidences the analyst has to collect in order to claim that all these files are equivalent. For example, the evidence the analyst has to collect in order to conclude that all files are equivalent database interaction files are that a) all files depend on the external libraries related to SQL for preparing and embedding SQL queries, b) all files depend on the external SQL exception files, c) all files depend on the data beans (i.e. classes with getter and setters methods of data attributes) either for inserting data to the database or to store the results of SQL queries into beans, and d) all files are stateless, that is, classes do not contain member variables or attributes. We codified such strategies for each concern and stored them in the knowledge base so that analysts can share knowledge with other analysts and reuse them for analysis of other systems.

2.3.3. Architecture Gallery

Because we want to store architecture knowledge and experience gained in analyses of implemented systems in a systematic way, we include an architectural gallery in our

knowledge base. An architectural gallery contains architectures of systems that were analyzed in the past. In many cases, code snippets were also stored to provide a precise description of the discovered architecture. For example, the gallery contains descriptions of how one project designed a software bus for inter-subsystem communication. This architectural gallery is a source of knowledge of best and failed practices, which the analyst can use during discussions with customers for recommending alternative architectural solutions.

2.3.4. Updating the Knowledge base

The knowledge base of external entities is updated if the system under analysis uses new COTS, frameworks, and programming language libraries or parts thereof that were not stored in the knowledge base. Our scripts compare the existing entries of the knowledge base with external entities used by the system under analysis, and reports missing entities that need to be added to the knowledge base. The analyst reviews the missing entities before adding them to the knowledge base. It is crucial to update the knowledge base in order to facilitate the analysis of future systems. In addition, the knowledge base is also updated if the analyst detects that the system under analysis follows a different strategy for the implementation of a specific concern. In this way, knowledge is incrementally added to the reverse engineering environment, and the knowledge base becomes a novel collection of best and failed practices of commercial systems, thus acting as an experience factory [Basili et al. 92].

2.4. Tools used in the AIS method

In order to efficiently perform architectural analysis of commercial systems using knowledge bases, analysts require several tools. In order to deploy the AIS method, analysts use several tools including the tools (or scripts) listed in Table 1.

Table 1 Tools used in the AIS method

Tool	Purposes
The Understand tool	This commercial tool helps in extracting code-level dependency models from the source code. It also identifies dead code elements.
The RPA tool	This tool supports in efficiently querying the extracted dependency models. It offers several relational algebraic operators enabling the analyst to easily query and filter the necessary information from the large dependency models [Feijs et al. 1998, Krikhaar 1999].
The SAVE tool	This tool supports in visualizing the extracted dependency models [Lindvall et al. 2010]. It provides user interfaces to import dependency data collected from external parsers (see http://www.fc-md.umd.edu/save/).
The Prefuse tool	This tool supports a variety of visualization, layout, and animation features. It helps in visualizing the directed and undirected graphs, tree views, and tree maps (see http://prefuse.org/)
Text Similarity tool	This tool finds a list of files similar (in terms of text) to the given file. Used here for detecting code clones or duplications.

3. The AIS method in Action

In this section, the applicability of the method is demonstrated using examples from NASA's Space Network Access System (SNAS) system, developed by Honeywell. The SNAS is intended to be a customer interface to the Tracking and Data Relay Satellite System (TDRSS) and is used for planning, scheduling and real-time service monitoring and control of the Space Network (SN). The SNAS is implemented in Java and in SQL. Excluding the test code present in the test folder, the SNAS has 650KLOC Java code and 30KLOC SQL code, including blank lines and comments. All code is handwritten and thus there is no generated code.

3.1. The reasons why the SNAS was chosen for validation of the method

We have applied the AIS method onto several commercial systems, implemented in several languages including Ada, FORTRAN, C/C++, and Java. In many cases, there were limited or no possibility to validate the findings of the method because the people (often contractors) who built the system were not accessible. In the SNAS case, fortunately we have access to several members of the team who are familiar with different parts of the system. It should be noted that we never had a meeting or technical discussion during our analysis of the SNAS source code. All the findings explained here were performed completely independent of the SNAS team. Once the analysis was completed, the analysis results were presented and feedback was collected from the SNAS team. Thus there were no influences whatsoever from the SNAS team on the results described here, unless explicitly stated.

In this section, we will describe how the analyst applied the proposed method to the SNAS system. We will first describe the directory or package dependency diagram before demonstrating how the external dependencies can offer novel help in architecture analysis.

3.1.1. Directory or Package Dependency Diagrams

The source code dependencies between subsystems, i.e., the top folders on the disk, of the SNAS are shown in Figure 6. While this figure offers a useful overview of how the source code is organized on the disk, and which folder uses other folders, there are some limitations if one analyzes architectures only from this dependency diagram alone. For example, we also need to get answers to the following questions: a) Do the subsystems communicate at run-time using intermediate connectors? b) Do the subsystems run in the same machine or is the system distributed? c) If there is a database, what is the interaction style or pattern used by different subsystems? These questions are not straightforward to answer by looking at the dependency diagram alone.

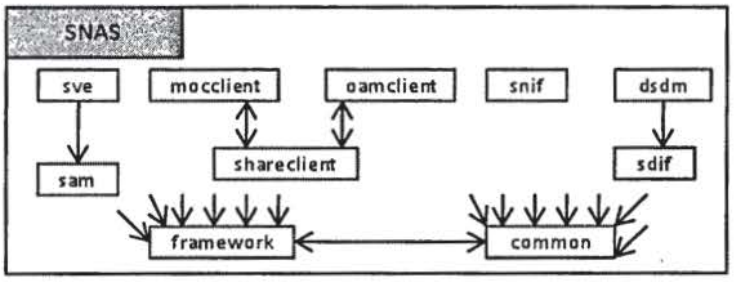

Figure 6 Dependencies of the SNAS, boxes represent folders and arrows are code relations (e.g. import, inherit, and call). The `common` and `framework` folders are used by all folders. sve to sam dependency is due to dead code. dsdm to sdif dependency is due to the sharing of a string utility class.

We will now show how an analyst can answer such questions by using external dependencies and information stored in the knowledge base.

3.1.2. Some Facts about the SNAS using its External Dependencies

The analyst starts by producing a high-level summary using a collection of scripts that use the extracted source code relations of the SNAS and the knowledge base as inputs, see Figure 7. One of the advantages of this generated summary is that it shows the list of concerns (e.g. GUI, Database, Configuration, Security, and Logging) built inside the SNAS. It also shows some potential architectural connectors (e.g. Sockets) implemented in the SNAS.

A Snippet of summary produced using the knowledge of External Dependencies

<<SNAS> has
- GUI because it uses `java.awt` and `javax.swing` packages
- Database because it uses the `java.sql` package
- Secured Socket Layer (SSL) because it uses the COTS `crysec.ssl` package
- UDP sockets because it uses the `java.net.DatagramSocket` class
- TCP sockets because it uses the `java.net.ServerSocket` and `java.net.Socket` classes
- Non-blocking Socket channels because it uses `java.nio.ServerSocketChannel`
- Configuration files because it uses `java.util.Properites` (`load`, `getProperty`) methods
- OS interaction because it uses `java.lang.Runtime` (`exec`) method
- Logging because it uses the `org.apache.log4j` package

Figure 7 An Excerpt of the high-level summary produced using the knowledge of external dependencies used by the SNAS.

3.2. Discovering Architectural Styles and Communication Patterns using External Dependencies of SNAS

The analyst understands from the high-level summary that the SNAS uses sockets. Therefore, it is reasonable to assume that the subsystems of the SNAS might be communicating using sockets as runtime connectors. Thus, the analyst's natural next step is to identify server-side and client-side sockets.

Table 2 Analysis Questions for Discovering Server-side sockets, Client-side sockets, and Connection ports.

- Discovering Server-side sockets: Which subsystems create instances of `java.net.ServerSocket`, `java.nio.ServerSocketChannel`, and `crysec.ssl.SSLServerSocket`?
- Discovering Client-side sockets: Which subsystems create instances of `java.net.Socket`, `crysec.ssl.SSLSocket`, `DatagramChannel.socket()`, and `SocketChannel.connect(...)`?
- Discovering Socket Wrappers: Also check whether there are wrapper classes to external socket libraries, because socket instances could be indirectly created by creating instances of wrappers.
- Discovering Ports: Use dependencies to the `java.util.Properties` class and locate configuration files. Experience tells us that IP addresses and port numbers are often specified in configuration files.

3.2.1. Discovering Server Subsystems using External Dependencies

Based on the strategies stored in the knowledge base (see Table 2), the analyst queried the extracted code relations of the SNAS in order to identify all subsystems that create instances of the `java.net.ServerSocket` class. The results showed that the `dsdm` and `sdif` subsystems create one instance of the `ServerSocket` class, see Figure 8 (a) and (b). Since the SNAS also uses Java's non-blocking Input/Output class `java.nio.ServerSocketChannel`, the analyst also queried the code relations for dependencies on this class. The query detected that the `ServerSocketAcceptor` within the `framework` folder create a socket instance using `java.nio.ServerSocketChannel`. After a quick glance at the `ServerSocketAcceptor` class it became clear to the analyst that this is a wrapper class for creating server side socket instances. Thus, the analyst queried the code relations in order to find all subsystems that create instances of this wrapper class. The analyst found that `snif` creates two instances of this server socket wrapper class and that `sam` creates four instances; see the unfilled circles of Figure 8 (c), (d). However, the `mocclient` has a different strategy to create server socket instances. The analyst found that the `mocclient` has a base class that uses the wrapper class of the `framework` to create a server socket instance. In addition, there are six children of the base class that indirectly create their server-side socket ports using calls to the `super` method of their parent class, see Figure 8 (e).

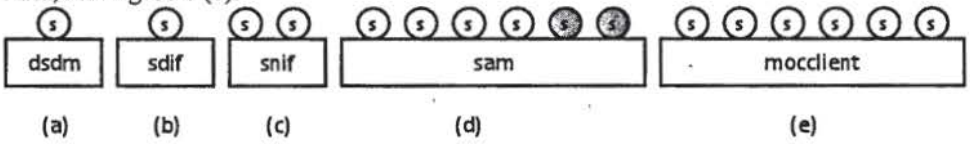

Figure 8 Discovered Server-side sockets using the architecture discovery guide of the knowledge base. Filled circles denote instances of `crysec.ssl.SSLServerSocket`. Unfilled circles are the instances of `java.net.ServerSocket`. All the server socket instances are created in different files of each subsystem.

Since there were also dependencies from the SNAS to the `crysec.ssl.SSLServerSocket`, which is a COTS component, the analyst also queried the dependency model in order to find all subsystems that create instances of this class. It turned out that the `sam` subsystem is the only subsystem that creates and uses two secured server side instances of the `SSLServerSocket`, see the filled circles of Figure 8 (d).

3.2.2. Discovering Client Subsystems using External Dependencies

The analyst repeated the above process and discovered the client side socket instances, using external dependencies to Java's client-side socket class `java.net.Socket` and Crysec's `SSLSocket` class.

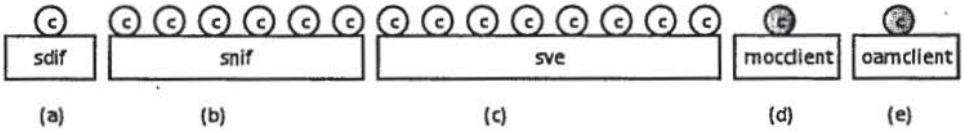

Figure 9 Discovered Client-Side sockets. Filled circles denote instances of `crysec.ssl.SSLSocket`. Unfilled circles are the instances of `java.net.Socket`. All the client socket instances are created in different files of each subsystem.

3.2.3. Connecting Server and Client Side Ports using External Dependencies

In order to connect client-side and server-side ports, the analyst used Java's Properties file used for configuring each subsystem. The analyst located the right set of property files using dependencies to the `java.util.Properties` class. These property or configuration files contain the IP address of each subsystem together with the actual port values. From this information, the analyst was able to map the server side ports to the client side ports. The names of the files involved in socket communication contain a good prefix (e.g. sam2sveConnector.java), offering additional valuable data to connect the ports. The external dependencies to Java's Datagram socket class, which contains methods for implementing the UDP protocol, showed the analyst that the UDP protocol is used between the `sve` and `snif` subsystems (see Figure 10).

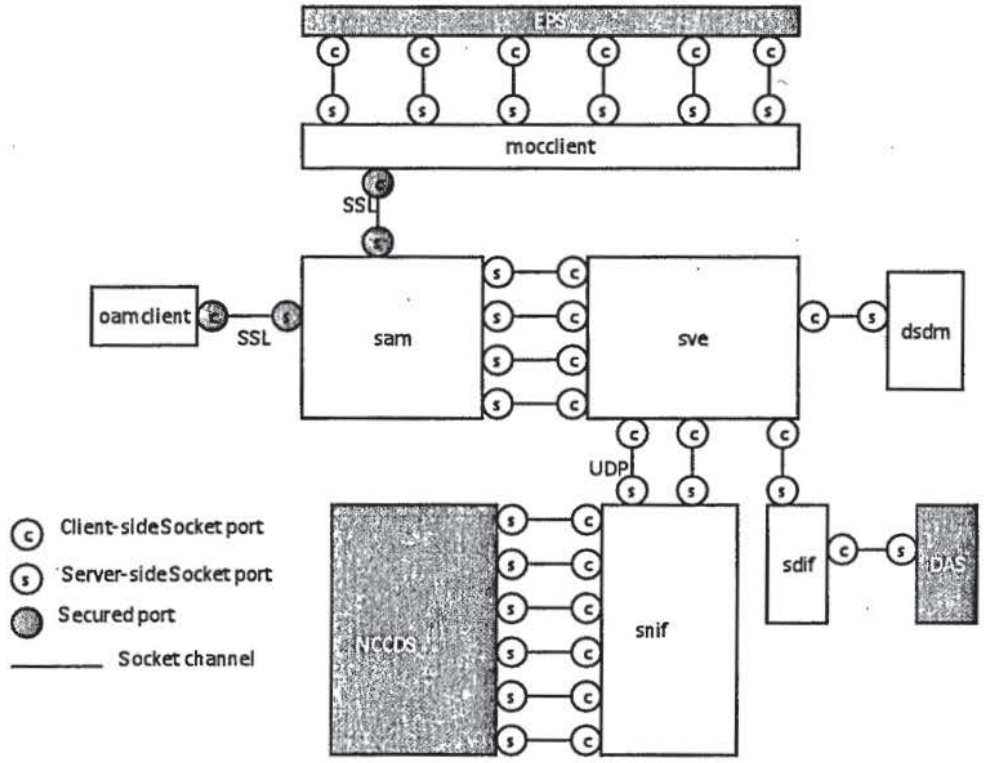

Figure 10 Discovered Run-time Architecture (Blue boxes are back-end systems that interact with the SNAS). Objects are sent between subsystems using the Transfer Object Design Pattern [Alur 2003].

3.2.4. Discovery of Transfer Object Design Pattern for Transporting Data over Sockets using External Dependencies

The extracted code relations showed that the files that are involved in socket communication use the `java.io.ObjectStream.writeObject` and `java.io.ObjectStream.readObject` methods. The analyst reviewed those files and found that the subsystems of the SNAS use the `writeObject` and the `readObject` method for sending and receiving serialized objects over the socket, as required by the Transfer Object Design Pattern [Alur 2003]. The central idea of this pattern is to transfer objects across communication channels, instead of making remote procedure calls to

overcome the inherent network performance overhead of RPC [Waldo et al. 1994]. In addition, the extracted code relations had shown the analyst that all those serialized objects that are sent over sockets are located in the `common` folder explaining why all subsystems depend on the `common` folder. Had the analyst excluded the external dependencies to Java's `writeObject` and `readObject` methods, it would not have been straightforward to discover this design pattern hidden in the implementation.

The analyst noticed that there are 384 files (or classes) inside the `common` folder. Without reviewing all those files the analyst concluded that all of them are equivalent bean classes (i.e. data containers) because a) all the classes of the `common` directly or indirectly (i.e. using inheritance) implement the `Serializable` interface, a vital condition for transferring objects on sockets, b) all methods of all `common` classes are simple setters and getters, i.e. they have the prefix `get`, `set`, and `toString` (in some cases), c) in addition, the collected code metrics showed that almost all methods of the `common` classes are one-line getters or setters, d) and there were no logging which matches the fact that in general, bean classes do not typically log their activities. A few classes do use logging, but the majority of them do not use logging, and e) there were no outgoing dependencies from `common` to other subsystems, except that some the classes of `common` use utilities of the `framework`. Using these gathered evidences, the analyst inferred that all classes of the `common` folder are equivalent bean classes, which are used for just transferring data among distributed subsystems. This capability to generalize a collection of classes and summarize their role in one sentence is so crucial in architecture discovery because now the analyst knows that these 384 files are beans that are used for transmitting data across communication channels between subsystems.

For this paper, it was not possible to analyze the back-end systems (colored boxes in Figure 10) without discussing with the SNAS team because the analyst did not have access to the source code. The SNAS team told the analyst after the analysis was completed that the reason for having 6 client-side socket ports at the `snif` subsystem is due to its counterpart back-end: the NCCDS system, which was developed many years before the SNAS was developed. Similarly, a new requirement drove them to introduce 6 server-side socket ports at the `mocclient` subsystem, in order to allow the EPS system to communicate with the NCCDS system.

Finally, the SSL is used for the connection between `mocclient`, `oamclient` and the `sam` because both clients are deployed in an open network and the connection must be secure. These are the kinds of design rationale we will not be able to discover from the source code alone, and definitely need to talk to the people (if available). There is a limit for Reverse Engineering.

3.3. Analyzing the Discovered Run-time Architecture

The analyst then discovered the run-time software architecture of the SNAS using the following questions: 1) What are the performance influencing architectural decisions from the communication perspective? 2) How complex is the implementation from the communication perspective? 3) Is there a common look-and-feel from the communication perspective? 4) Are the files involved in interactions with communication channels cloned from each other? and 5) Can the subsystems of the system be tested independently?

3.3.1. Performance and Communication: The analyst has just concluded that the distributed subsystems of the SNAS communicate using the Transfer Object Design

pattern by sending and receiving serialized objects over the sockets. SUN's book mentions that Remote Procedure Calls (RPC) using the Java's Remote Method Invocation (RMI) can be slow due to communication overhead [Alur et al. 2003], despite the fact that RMI is simple and fairly easy to understand and program. [Alur et al. 2003] also argue that by using the Transfer Object Design pattern, performance can be improved. However, according to the SNAS team the introduction of Transfer Object Design pattern did not solve all performance problems, because if objects are sent over sockets, then there is an issue of managing the waiting time in the sockets before the receiver picks-up the objects for processing. To avoid potential delays and degraded performance, the SNAS team introduced additional ports to different subsystems and transferred different types of objects through different ports. For example, the four socket connections between the `sam` and `sve` are used for exchanging four different types of objects, see Figure 10. By doing so, the SNAS team attempted to reduce the waiting time of objects on sockets.

Traditionally, socket programming uses one thread per client connection. However, frequently creating and destroying threads due to short-lived sessions would incur performance overhead. Also, valuable CPU time can be wasted just because of context switching due to threads. In order to overcome these performance issues, Java 1.4 introduced a new architecture concept called non-blocking socket communication channels for client-server communication. The analyst found that the files that are involved in socket communication use the `java.nio.channels.SelectionKey` and the `java.nio.channels.Selector` classes. These two classes are the core for implementing the reactor design pattern in Java (see [Schmidt 1995] and [Naccarato 2002] for details). In this pattern, the event demultiplexer waits for events that indicate when a socket is ready for a read or write operation. The demultiplexer passes this event to the appropriate handler, which is responsible for performing the actual read or write. Based on these collected evidences, the analyst hypothesized that the SNAS inter-subsystem communication architecture is inspired by performance goals.

3.3.2. Complexity and Communication: In order to reason about complexity from the communication perspective, the analyst reviewed the files involved in socket communication. Because the Transfer Object Design pattern is used for transferring data, some of the files that read objects from the socket channels contain a lengthy sequence of `if/then/else` statements for deciding the data type of the incoming objects in order to delegate them to methods responsible for processing each particular object type. Thus, some of the complexity can be attributed to the Transfer Object Design pattern.

3.3.3. Common look-and-feel and Communication: The analyst detected some common look-and-feel issues due to the by-pass of the socket wrapper defined in the shared `framework` folder. In particular, both the `dsdm` and `sdif` subsystems create instances of server sockets by directly using the `java.net.ServerSocket` class. Similarly, the `mocclient`, `oamclient`, and `sdif` subsystems create instances of client sockets by directly using the `java.net.Socket` class instead of using the wrapper. Thus, the look-and-feel from a communication point of view is different among the subsystems. The reasons for differences in look-and-feel will be discussed together with other architectural violations at the end of this paper.

3.3.4. Code Cloning and Communication: The analyst used the similarity tool to compare all files of the SNAS and produced a similarity table that contains pairs of

potential file clones. Since the list of files involved in socket communication was located during the discovery of the SNAS runtime architecture, the similarity table was sliced with respect to those files only. This gave the analyst remarkable insights into code cloning from a very specific perspective, i.e., the communication perspective.

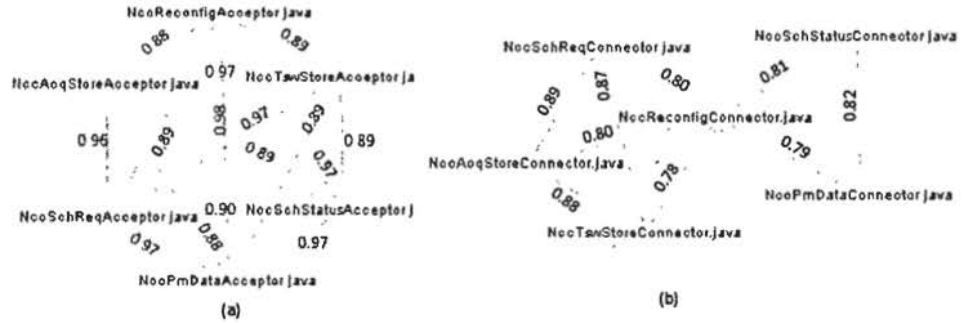

Figure 11 (a). Similarity among the 6 files of the `mocclient` involved in the server-side socket communication. (b). Similarity among the 6 files of `snif` involved in the client-side socket communication. Similarity value toward 1 means high code duplication between the file pairs.

The analyst concluded that all six files (see Figure 11 (a)) which accept client connections and act as server-side socket ports of the `mocclient` are very similar to each other. It is interesting to note that even though there is a base class for each of those six classes in `mocclient`, there is a lot of code duplication between these six files. In addition, the six files that act as client-side socket ports of the `snif` are very similar despite the fact that they have the same base class. This means that shared behavior is not properly abstracted yet, see Figure 11 (b). The two server side socket files of the `sam` subsystem are also cloned. Similarly, the files that are involved in the client-side socket communication of the `oamclient` and `mocclient` are very similar. There are many methods in these two files which are exact copies of each and could be moved to the `shareclient`, which is a shared infrastructure for both client types.

3.3.5. Testing and Communication: Because of the distributed client-server architectural style, clients can be tested with fake servers, and vice-versa without changing any source code. However, some changes (e.g. IP address and ports) are needed in the configuration file. In fact, the SNAS team has also developed simulators for back-end systems so that the SNAS can be tested without the real back-end systems being up and running. These simulators can send data over the socket to subsystems of the SNAS. Classes involved in the socket communication will read the incoming object types as in the real scenario. More details on testability issues due to Databases and GUI are discussed below.

3.4. Discovery of the Database Interaction Architecture using External Dependencies

Our approach for analysis of database concerns is based on the following observations of several commercial systems: Many systems implement their need for persistence by using a RDBMS that is based on the SQL language, which is typically external to the software under study. Thus the software under study needs to connect to and disconnect from the database, communicate with and transfer data to and from the database, as well as manage errors during interaction with the database. It is also desirable if the software

under study is not directly dependent on the database so that the software under study can be tested without the database and so that the database can be replaced if necessary. Many systems implement DAOs (Data Access Objects) layer which contains classes that are responsible for interacting with databases for storing and retrieving data from the database. On the one hand, DAOs collect the results of database queries and convert them into data beans which are basically data containers with getters and setters. On the other hand, if we want to store a bean into a database table, the bean object is passed as an argument to the methods of the responsible DAO [Alur et al. 2003].

Based on this model, we derive the following questions: 1) Is there a DAO (Data Access Object) layer that abstracts the physical database? 2) What is the general strategy for managing database connections? 3) Can the system be tested without the database being up and running? 4) Are database errors abstracted and propagated upwards in such a way that higher-level layers are not aware of databases? 5) Is there a common look-and-feel in the way database tables are accessed by different subsystems? and 6) What are the different DBMSs the system supports?

In order to answer these questions, the analyst started by slicing the system with respect to dependencies on database tables. The analyst used a parser that identifies files that use the database tables based on regular expressions involving key SQL statements, for example, "select", "insert", "update", and "delete". The extracted dependency relation from the SNAS source code files to database tables is shown in Figure 12. Once the analyst had determined that such dependencies existed, the conclusion was that the system must be using a database in a direct way, instead of using indirect database dependencies that can be created using java.persistence. Such indirect database dependencies can make use of a database without using any of the SQL keywords listed above, which the analyst confirmed was not the case for the SNAS. The analyst then made the observation that there is a good common look-and-feel in the way the files that are using database tables are organized on the disk because there is a db folder per subsystem, each containing the classes that interact with database tables using SQL statements. The analyst's other observation was that `snif`, `dsdm`, `sve`, and `sdif` depend on a database and thus interact with it in some way, but `oamclient`, `mocclient`, and `sam` do not depend on a database.

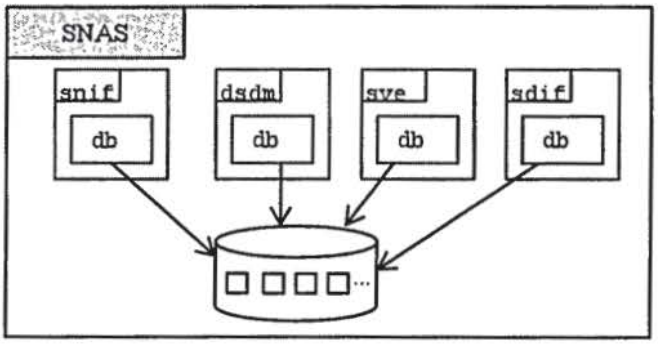

Figure 12 Sliced View of the SNAS showing dependencies on database tables. Arrows denote SQL queries from Java files of the db folders.

Here, the answers to above questions are presented using the knowledge of external dependencies.

3.4.1. Discovering the Database Interaction Architecture of the snif subsystem using External Dependencies

The analyst selected the `snif` subsystem, which is one of the four subsystems that depend on a database, and proceeded to analyze the `snif` subsystem's database interaction style. The knowledge base knows that the methods of the `java.sql.PreparedStatement` class can be used to prepare and execute database queries in Java. Using that knowledge, the analyst queried the extracted code relations and found that all classes of the `snif` that prepares database queries are organized in one folder/package: `snif.db`, see Figure 13 (a). In addition, the analyst observed that all Java files that use SQL statements such as `select`, `insert`, `update`, and `delete` are only present in the `snif.db` folder, thus confirming that all direct database interactions are limited to the db folder.

The analyst then proceeded to analyze how the execution of SQL queries is managed. The analyst used the knowledge that in order to execute SQL queries from Java, a `java.sql.Connection` object is needed. The analyst then found, by analyzing the extracted code relations for dependencies to `java.sql.Connection`, that each class in `snif.db` contains a method called `setDbConn` which takes the `Connection` object as a parameter, see Figure 13 (b).

The analyst then concluded that all classes of the `snif.db` folder can be safely categorized as DAOs because of the following evidences: a) all classes in `snif.db` depend on the `java.sql` package, b) all classes in `snif.db` use classes of `common`, which contains data beans as shown earlier, and these data beans are either used to convert SQL results into objects or to insert data into database tables as explained above, c) there are no out-going dependencies from `snif.db` to other folders of `snif`, and d) each class in `snif.db` gets a database connection object from outside through the `setDbConn` method. Based on these collected evidences and without reviewing all classes in `snif.db`, the analyst inferred that the `snif` subsystem has clear separation of database table concepts from other concepts.

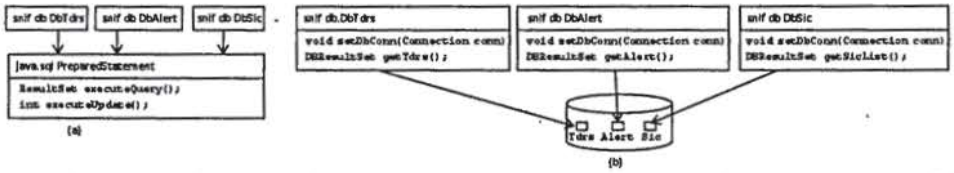

Figure 13 (a) Classes interacting with the `java.sql.PreparedStatement` class used for preparing SQL queries. (b): The DAO layer containing all classes that are using the database tables via SQL queries. `DBResultSet` (the return type) has an array of objects where each object corresponds to one row of the queried table.

3.4.2. Clones due to the Java Language and Exception Handling in the DAO Layer of the snif Subsystem

The analyst now knows that there is a dedicated DAO layer consisting of the classes in `snif.db`, which interact with database tables, see Figure 13. The analyst also knows, thanks to the extracted dependency relations, that the classes of the DAO layer are independent of each other.

The analyst proceeded to run the similarity tool on the files in the DAO layer, which reported occurrences of clones. The analyst analyzed some of the reported clones to gain

insights into the underlying reason behind cloning. The analysis showed that the `catch` and `finally` blocks in each file of the DAO layer are identical. In the `catch` block, the error code stored in the `SQLException` is processed and converted into a SQL independent error code. The `catch` block contains code that is used to roll back database transactions that did not complete properly. In the `finally` block, all classes call the close method of the `java.sql.PreparedStatement` object and commits successful database transactions.

In our opinion, the developers are not to blame for these clones in the DAO layer. Rather, this is an inherent limitation of the Java language and its way of supporting database programming because it leads to the creation of boiler-plate code that is identical across all DAO classes except for only a few parameters that differ. The boiler-plate code the `catch` block includes, for example, code to a) manage the database connection, b) create an instance of `PreparedStatement`, c) handle SQL exceptions, and rollback of transactions, and d) close the `PreparedStatement` object. It is not straightforward to abstract the `catch` block into a modular unit. Modern frameworks (e.g. Hibernate and javax.persistence) were invented exactly to solve these code redundancy problems in database interactions, making a solid business case with ample evidence to migrate to modern frameworks in the future.

3.4.3. Is the snif Subsystem testable without a running database?

In order to evaluate testability from the database point of view, the analyst first had to understand how the DAOs (i.e. classes in the `snif.db` package shown in Figure 13) are used within the `snif` subsystem. More specifically, the analyst must understand whether or not it is possible to avoid interactions with the database. To this end, the analyst checked whether or not the DAOs that interact with the database are instantiated by other classes of the `snif` in a hard-wired way. To achieve this, the analyst extracted all incoming dependencies to the DAO layer and found that no other class is using the DAO layer except the `SnifDatabaseManager` class, which creates instances of all classes of the DAO layer, see Figure 14 (b). The analyst also found that the `SnifDatabaseManager` is instantiated only by the `ServiceManager` class, which gets all necessary database parameters such as login, password, and url, from the configuration file. Based on these findings, the analyst concluded that the `SnifDatabaseManager` class is the gateway for interaction with the DAO layer, see Figure 14 (a). By studying the extracted dependency relations, the analyst also noted that the `SnifDatabaseManager` class uses the `DBConnectionPool` class in the `framework`. This discovery is fully explained in the next section. The analyst reviewed the `SnifDatabaseManager` class and found that it creates a pool of database connections with the capacity of eight connections, see Figure 14 (c). The review also showed that the `SnifDatabaseManager` distributes the eight database connections among the several classes of the DAO layer, based on the knowledge of frequency of access to different database tables, through calling the `setDbConn` method DAOs, see Figure 14 (d). These findings led the analyst to conclude that the `snif` subsystem's database interaction architecture is driven by performance goals because it creates several database connections and distributes them among the classes of the DAO layer. This example also shows the power of slicing the extracted code relations using concerns; otherwise we cannot easily see the beauty of hidden lasagnas in spaghetti of complex dependency relations or graphs.

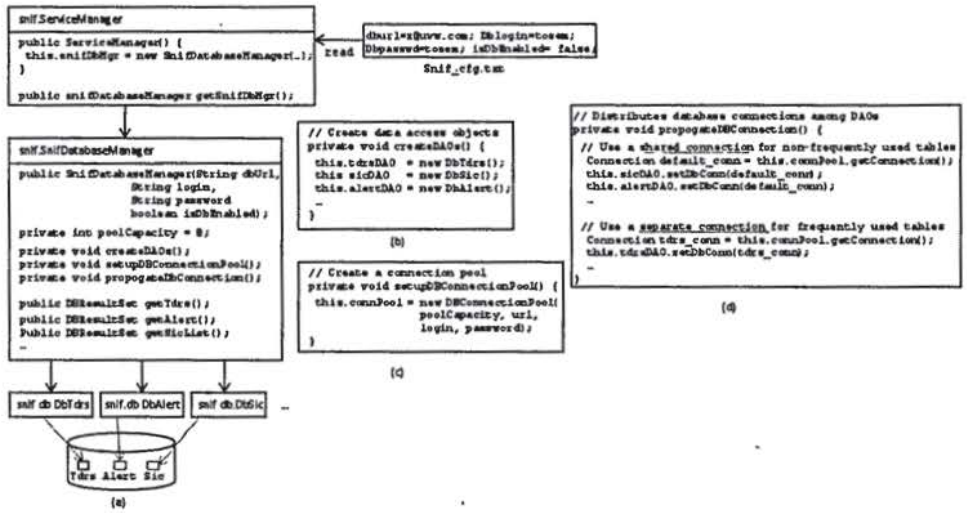

Figure 14 (a): All accesses to DAOs (i.e. snif.db.*) are only using the database manager class, indicating that database concepts are separated from other concepts. Database connections are shared among DAOs. For example, Tdrs has its own database connection (the red dotted arrow), Alert and Sic tables share a database connection (the green arrows). Note that the set of public methods (e.g. getTdrs, getAlert, getSic) are the union of public methods offered by all classes of the DAO layer (shown earlier in Figure 13(b)). (b). The database manager is creating instances of all classes of the DAO layer. (c). A connection pool with a capacity of 8 connections is created using the DBConnectionPool class, and (d). Database connections are distributed among DAOs based on the frequency of access to different tables.

During the review of the `SnifDatabaseManager` class, the analyst also found that its constructor has a boolean flag called `isDbEnabled`. If the flag is false, then the public methods of the `SnifDatabaseManager` call the "dummy" methods of the classes of the DAO layer. The analyst randomly picked one of the dummy methods of one of the DAOs and found that it populates dummy data for testing purposes. The extracted code relations also showed that each class of the DAO layer contains methods with the name "dummy" in it, which confirmed the analyst's hypothesis that this construct existed to facilitate testing. For example, if the higher-level layers call the `getSicList` method of the `SnifDatabaseManager` then either the real `getSicList` defined in `DbSic` or the dummy `getDummySicList` method will be called, see Figure 15 (b). Note that the users of the snif database manager do not have to change the source code for unit testing because it is enough the change the configuration file (snif_cfg.txt) and set the `isDbEnabled` parameter to false.

Although the `snif` subystem has the capability for testing without running the database, the analyst concluded that there is a mix of testing concerns with the real behavior: the `isDbEnabled` flag is used as a control to switch been real and dummy DAO methods at run-time. Our recommendation is to separate the testing concern using dependency injection concepts proposed in [Spring] or Google's [Guice] frameworks as follows. The core idea is to let each class within the DAO layer implement an interface of the services it offers. In addition, corresponding to each real DAO class, there is a separate dummy DAO class with the same interface but with an implementation that populates fake or dummy data. Instead of creating instances of DAO classes in a hard way, as is currently the case, the database manager will create instances of DAO interfaces. These interface can be bound either to real DAO instances or to dummy

instances for testing. This can be done with the help of configuration concepts used in the Spring or the Guice frameworks. Thus, using this design one could separate testing concerns from the real code, which would increase testability and readability significantly.

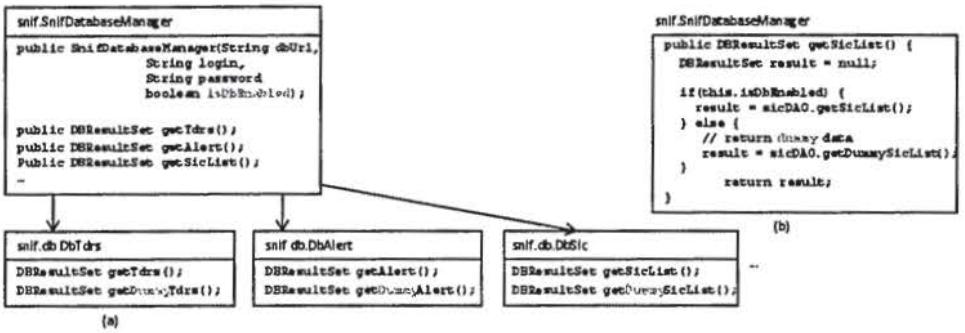

Figure 15 Testability is built into the design. (a): if the database manager is instantiated with `isDbEnabled` as false, then dummy methods of the DAOs will be called - similar to the method shown in (b).

3.4.4. Discovery of the Database Connection Pool Design Pattern using External Dependencies

As mentioned above, the analyst noted that the `SnifDatabaseManager` class uses the `DBConnectionPool` class in the `framework`. The analyst's next step was to understand the details of how the connection to the database was managed and discovered that the `DBConnectionPool` class within the `framework` uses the `DriverManager` class, see Figure 16 (a).

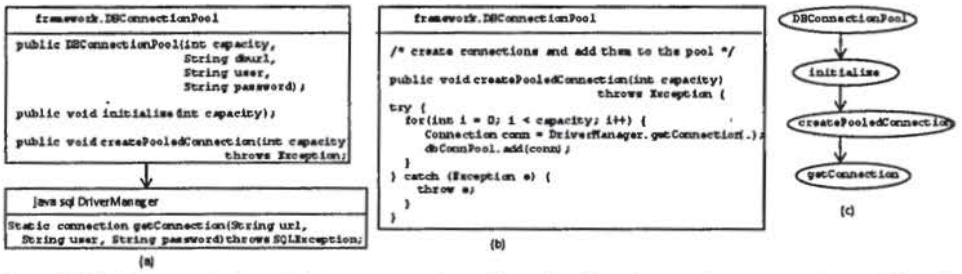

Figure 16 (a): Discovered slice of database connections, (b): code snippet for creating a connection pool (see the for-loop), and (c) The call graph showing how a database connection is created. The `initialize` method is shown in Figure 17.

The analyst then reviewed the `CreatePooledConnection` method because it calls the `DriverManager's getConnection` method, which drew the analyst's attention because the knowledge base says that the `getConnection` method of the `java.sql.DriverManager` is used to create connections to the database. The review showed that a pool of database connections is created in a for-loop of the `createPooledConnection` method, see Figure 16 (b). The for-loop calls the `java.sql.DriverManager.getConnection` method, which returns a database connection object that will be stored in the connection pool. This discovered pattern is called the Database Connection Pool design pattern: The core idea is that a set of

connection objects are created up-front, as demonstrated here, and when a component needs to access a database table they can take one already created connection object from the pool, and return it to the pool after using it. Thus saving the time it takes to create and destroy they connection [Apache DBCP]. This discovered slice of the SNAS indicates that the database interaction architecture is driven by performance goals because experience reminds us that frequently creating and destroying connections to a database can affect performance of the system. This concrete example highlights the value of the knowledge base of external entities helping us in easily finding the file and the methods that implement the design pattern for database connections.

3.4.5. An Error Handling Issue in the Connection Pool Design Pattern Implementation

During the process of analyzing the database connection management strategy, the analyst also noted that the `java.sql.DriverManager.getConnection` method throws a `SQLException`, see Figure 16 (a). In order to learn more about the error management strategy utilized by the SNAS in the context of database connections, the analyst proceeded by analyzing how this exception is handled. The analyst noticed that the `createPooledConnection` method, which calls the `getConnection` method, catches the exception, see Figure 16 (b & c), and throws it to the caller method, namely the `initialize` method, see Figure 17. The `initialize` method logs the exception but fails to propagate problems (e.g. database connection failures) to higher-level layers. The analyst also noted that the return type is void, which prevents the method from communicating any result to the caller. The analyst concluded that the developers applied an elegant connection pool design pattern to achieve high performance for database connection that were provided as a reusable asset for all subsystems to use, but did not pay sufficient attention to error handling strategy.

```
public void initialize(int capacity) {
      try {
          createPooledConnection(capacity);
      }
      catch (Exception e) {
          LogService.logException(…);
      }
  }
```

Figure 17 The `initialize` method does not throw the exception upward, instead it just logs the exception. Thus, higher-level layers have no idea in case something goes wrong during the creation of a database connection.

3.5. Discovering the Database Interaction Architecture of the `sdif` Subsystem using External Dependencies

Similar to the analysis of the `snif` subsystem database interaction architecture, the analyst used the dependencies to the `java.sql.PreparedStatement` class and discovered that the only class that prepares SQL statements is `sdif.db.DbInteractor`. The extracted call graph of this class showed that almost all of its methods use methods of `PreparedStatement` in order to prepare and execute SQL queries. In addition, the analyst found that the only outgoing dependency from the `DbInteractor` class is to the `common` folder, which contains data beans as explained earlier. The only exception is dependencies to logging methods. Based on these evidences the analyst concluded that

the DbInteractor is the only class of its DAO layer and that it is responsible for interacting with several database tables, in contrast to a collection of several DAO classes in other subsystems.

The analyst also noticed that there were no dependencies from the DbInteractor class to the database connection creation method getConnection of the java.sql.DriverManager class. This led the analyst to investigate further how the sdif subsystem creates database connections. The extracted dependency relations showed that the only class that depends on the DriverManager class is the sdif.db.DbConnectionManager class. The analyst reviewed this class and found that it uses the Singleton design pattern [Gamma et al. 1995] and creates only one instance of the database connection, see Figure 18 (b), as opposed to dividing the database traffic among several database connections using a Database Connection Pool as was the case for snif . In addition, the analyst queried the extracted code relations and found that the database connection manager class gets all parameters (e.g. database url, login) from a configuration file, see Figure 18 (a).

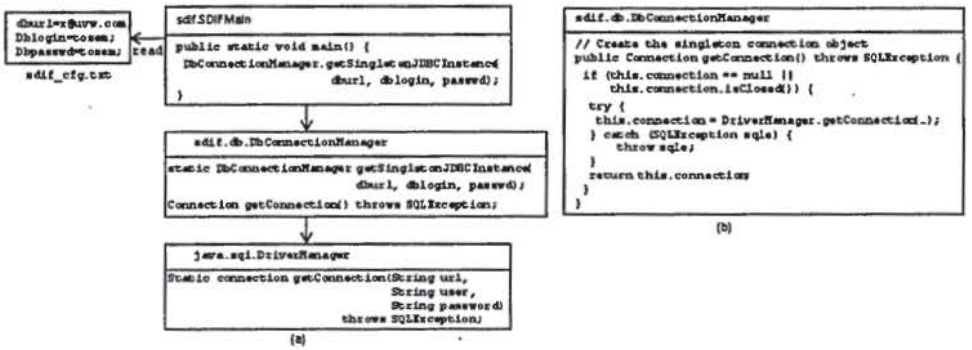

Figure 18 Discovered Singleton Design Pattern for database connections in the sdif subsystem.

The extracted call graph of the methods of the DbInteractor class showed that all its public methods use the methods of the DbConnectionManager class in order to get an instance of a database connection. The analyst reviewed some of the methods of DbInteractor and concluded that they all follow a general pattern: First, in order to get an instance of the DbConnectionManager, all methods of the DbInteractor call the static getSingletonJDBCInstance method. Second, using that instance of the database manager, all methods of the DbInteractor call the getConnection method of the database connection manager. Third, all methods of the DbInteractor run SQL queries and return results to their callers. These three steps are summarized in Figure 19 (b). Note that there is an architectural mismatch due to the way the DAOs of the sdif and snif subsystems create a database connection: the DAOs of sdif are responsible for getting an instance of the database connection, whereas the DAOs of snif are assigned an instance by the data manager. Thus, these two subsystems have different common look-and-feel with respect to the database connection concern.

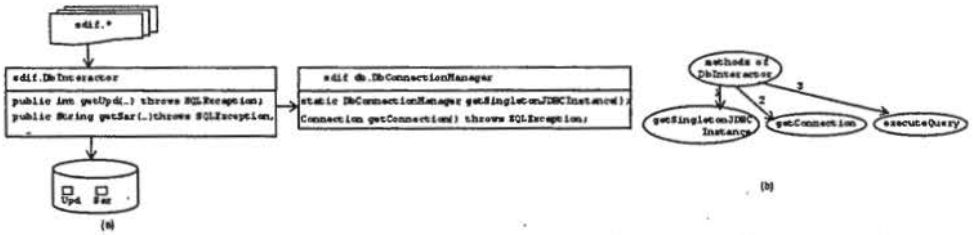

Figure 19 The `DbInteractor` class offers public methods that queries database tables and return the results to the callers. (b): First, all public methods of the `DbInteractor` call the static `getSingletonJDBCInstance` method in order to get an instance of the database connection manager. Second, they call the `getConnection` method to get a database connection. Third, they run SQL queries by calling the `executeQuery` method.

3.5.1. Database Error Abstraction Issues in the sdif Subsystem

The analyst has concluded that all accesses of database tables are only using the methods of the `sdif.db.DbInteractor` class, see Figure 19(a). The analyst also noted that all the public methods of `DbInteractor` throw `SQLException`, see the methods declarations in Figure 19(a). As a consequence, the knowledge of the database concepts had leaked into the higher-level layer because it has to handle `SQLExceptions` being thrown by the methods of `DbInteractor`. Thus, in contrast to the other subsystems, the DAO layer in `sdif` (i.e. the `DbInteractor` class) fails to abstract the `SQLException` into an error object type that is free of database concepts. This example shows that the developers implemented the DAO layer but did not give sufficient attention in abstracting the error raised by the lower-level layer.

3.5.2. Testability Issues in the sdif Subsystem

The analyst then proceeded to analyze whether it is possible to test the `sdif` subsystem without the database. Having known that the `sdif.db.DbInteractor` is the only class that interacts with the database, the analyst queried the extracted call relation and found that all instances of the `DbInteractor` class is created within constructors, e.g. see Figure 20 (a), of higher-level layers.

```
sdif.ActiveSchedule
public ActiveSchedule() {
    this.dbInteractor = new DbInteractor();
}
```

(a)

```
sdif.ActiveSchedule
public int processUpd() {
    int upd = this.dbInteractor.getUpd();
    ...
}
```

(b)

Figure 20 Because constructors cannot be overriden, the methods of the `ActiveSchedule` class are not testable without a database due to the hard-wired dependency to the `DbInteractor` class, which queries database tables (also see Figure 19 (a)).

The analyst recalled the fact that constructors cannot be overridden. As a consequence, none of the public methods of those classes that use methods of the `DbInteractor` can be tested unless the database is running. Figure 20 (b) shows an example method that cannot be tested without the database because it uses an instance of the `DbInteractor` in a hard-wired way for calling the `getUpd` method, which accesses database tables, also see Figure 19 (a). The analyst found that there is no way to stop the control flow from reaching the physical database and the analyst found that there are 20

classes which unfortunately create instances of the `DbInteractor` within their constructors similar to the pattern shown in Figure 20. Hence, the `sdif` subsystem, in contrast to the other subsystems, is not testable without a running database. The code could be refactored to allow for testing without database [Flower 1999]. However, experience reminds us that managers are generally nervous about investing in refactoring because it does not add value to the product from the end-user's point of view. However, in our opinion, managers are open to refactorings if the proposed solution will make testing easier, as offered in this concrete case. Thus, this analysis helped to make a business case for refactoring to improve the testing capability. [Jacobson 1992] says, "To make the design minimally affected by the DBMS, as few parts of our system as possible should know about the DBMS's interface." Yes, this analysis has shown that the subsystems of the SNAS satisfy this quote in general. There are a few cases where the database exception knowledge is mixed with business logic as shown above.

The other subsystems (sve and dsdm) have a similar database interaction architecutre. Thus, we will not discuss them here. The analyst has not yet reviewed the stored procedures and Entity-Relationship models. Hence, the analyst cannot answer how the SNAS handles variants in DBMS (e.g. Oracle or MySQL). This example shows that the AIS method offers flexibility because the analyst can decide whether or not to address each question mentioned in the analysis guide, based on the available effort and the needs.

Now, the analyst proceeds to the analysis of GUI architecture using external dependencies of the SNAS.

3.6. Discovery of the GUI Architecture using External Dependencies

The model we base the analysis of the GUI concern on is based on the following observations of several commercial systems: In the interaction with users, the GUI: prompts the user to enter data, validates and accepts user data. The data is processed and stored locally and/or sent elsewhere for processing and/or storing. If the data is processed elsewhere, then the processed data most likely needs to be communicated back to the GUI. Often data from other data sources are communicated to the GUI. The GUI displays such data as well as error messages to the user. To achieve this, the GUI often has a supporting data model that holds both user data and processed data. If the GUI is part of a client that communicates with a server, then there needs to be a strategy for how to communicate the data between the client and the server (or between peers). Of course, there are other important GUI-related concerns such as the layout of GUI panels, fields, buttons, undo/redo support, etc. In this analysis, we do not address such concerns because this analysis focuses on how the GUI is architected.

Based on this model, we derive the following questions for discovering and analyzing the architecture of SNAS GUIs, which is based on the previous discoveries of clients and servers etc: 1) How is data that was entered using GUI panels communicated to servers? 2) How is data communicated from servers to GUI panels? 3) How do GUI panels manage their data model? 4) What is the general threading model for GUI panels? 5) How does the user interact with the GUI? 6) How do GUI panels validate user input data? 7) Can the system be tested without the GUI being up and running? 8) Are there clones among the GUI related files? 9) Where and how is data processed?

We believe the above questions are architecturally significant because they are related to global principles that are of interest to all GUI panels. SNAS has 273 panels, thus it would be a time consuming task for the analyst to discover the architecture and answer

the above questions. Fortunately, the analyst can use the knowledge of external entities to detect the basic set of files that spans the GUI architectural subspace of the whole architectural space. The analyst first ran the summary generator, which reported that the SNAS implements GUI concepts because it uses classes in the `javax.swing` and `java.awt` packages. We built a knowledge base for the Swing and AWT packages in order to support the discovery of GUI architectures of implemented systems, and answer the above questions.

GUI	Event Generation	Java.util.EventObject	All event state objects shall be derived from this class
	Event Listener	Java.awt.event.ActionListener	The listener interface for receiving action events
		Java.awt.event.WindowListener	The listener interface for receiving window events
		Javax.swing.event.EventListenerList	A class that holds a list of EventListeners
	Event Management	Java.awt.EventQueue	Extract events from queue and dispatches based on event types
	Input Validation	Javax.swing.InputVerifier	Used to validate input data
	Menu Bar	Javax.swing.JMenuBar	Used for creating a menu bar and add menus
	Password	Javax.swing.JPasswordField	Provides specialized fields for password entry
	Progress Bar	Javax.swing.JProgressBar	Used for displaying the progress of some work
	Task Handling	Javax.swing.SwingUtilities	Used for running task on the Event Dispatching Thread
		Javax.swing.SwingWorker	Used for lengthy GUI-interacting tasks in a dedicated thread
	Window	Javax.swing.JFrame	A Frame is a window with a title and a border
		Javax.swing.JPanel	A generic lightweight container of GUI objects

Figure 21 A snippet of the knowledge base for Java GUI libraries

By analyzing the dependencies, the analyst then discovered that that the `mocclient` and the `oamclient` subsystems depend on Java's Swing and AWT package, Figure 22 (a). The analyst also noted that the `JPanel`, which is a lightweight container of GUI objects that is used to hold other GUI objects (e.g. Button, Text Fields), is used in both subsystems within the GUI package, Figure 22 (b). The analyst has already determined that these two subsystems can be executed separately, which implies the SNAS has two GUI interfaces. The analyst also discovers that both subsystems have a menu bar each, see Figure 22 (c).

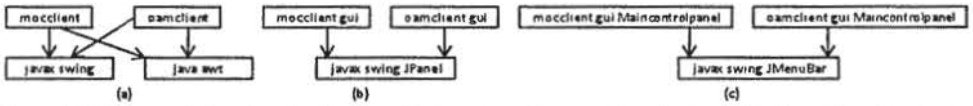

Figure 22 Discovered view showing that mocclient and oamclient are the subsystems that have dependencies to swing and awt and therefore the analyst concluded that they are the only ones that deal with GUI concepts. (b). There is a sub-package called GUI in both subsystems that depends on JPanel. (c). the main classes that construct the menu bar of the SNAS.

This paper will only discuss the analysis of the GUI architecture of the `mocclient` subsystem because the `oamclient` is similar to the `mocclient`.

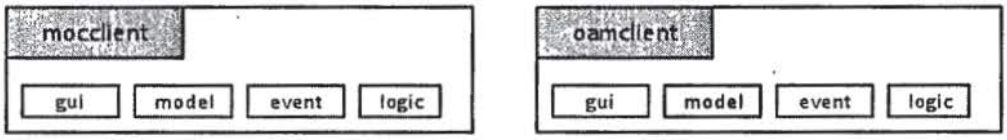

Figure 23 Package containment view for `mocclient` and `oamclient` showing that they both have four sub packages.

3.6.1. Locating the Main Menu using External Dependencies

Because the SNAS uses the Swing's `JMenuBar` class, the analyst decided to locate the class that implements the menu bar. By querying the extracted dependency relations the main panel class `gui.MainControlPanel` was discovered, see Figure 22 (c). The analyst reviewed this file and found that this is the main GUI panel for the `mocclient`. The analyst discovered that the `gui.MainControlPanel` creates several menus and delegates all menu events to the class `model.MainControlModel`. The dependency relation also showed that the `MainControlModel` class depends on almost all model classes in the model directory and all gui panel classes in the gui directory. This triggered the analyst to review the `model.MainControlModel` class after which the analyst concluded that this class is indeed the main controller of almost all gui panels and models as the name suggests. That is, all public methods of the main model class create and start instances of models and panels upon being invoked by the main panel class, see Figure 24 (a). The analyst reviewed the extracted code relations and concluded that almost all panels implement the `ActionListener` interface of the Java AWT. In addition, almost all panels take a model class as the argument to their constructors, see Figure 24 (b), such general patterns are good for architecture discovery because they indicate there is an underlying architectural skeleton where all panels and models could be plugged-in.

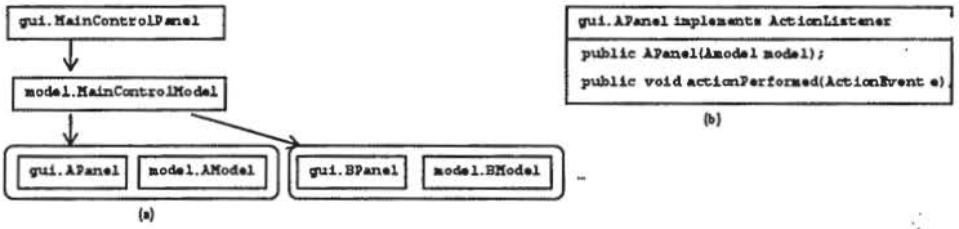

Figure 24 (a). The high level structure of GUI, (b). All panels implement the `java.awt.ActionListener` interface.

3.6.2. How is data communicated from servers to GUI panels?

The analyst discovered above that the `mocclient` connects to the `sam` server using an instance of `SSLSocket`, see Figure 10. The analyst queried the extracted dependency relations and located the gateway class `DataManager`, which is the only way for data to come in and go out of the `mocclient` subsystem, see Figure 25 (a).

Figure 25 (a). The remote server connection is established by the `ServerConn` class. `DataManager` creates an instance of the `ServerConn`. All data comes in and goes out of the `mocclient` only via the `DataManager` – it is the gateway, (b). The `DataManager` has a static method (`sendObject`) for sending data objects to the remote server. All processors call the corresponding set method to initialize their instance for call-backs from `DataManager`, when it reads response from the socket, it delegates to processors based on the response object type as shown in (c).

The analyst reviewed the DataManager and found that when responses come in from the `sam` server, the data manager delegates to appropriate data processors based on the incoming response data type. The analyst reviewed the extracted dependency model and found that there are many data processor classes that depend on the data manager and vice-versa. The analyst also found that all data processors call the `set` methods of the data manager in order to pass their object ids, which will be used by the data manager to call-back methods of the data processors, see Figure 25 (b) and (c).

The analyst queried the dependency model in order to locate the classes that create instances of the data processors and the data manager. The query revealed that the class `model.DataBroker` creates instances of all data processors and the data manager. The `MainControlModel` create an instance of the data broker, see Figure 26 (a). The analyst determined that the `MainControlModel` gets references to the data processors in order to send data to the `sam` server and this is done using the `DataBroker` instance.

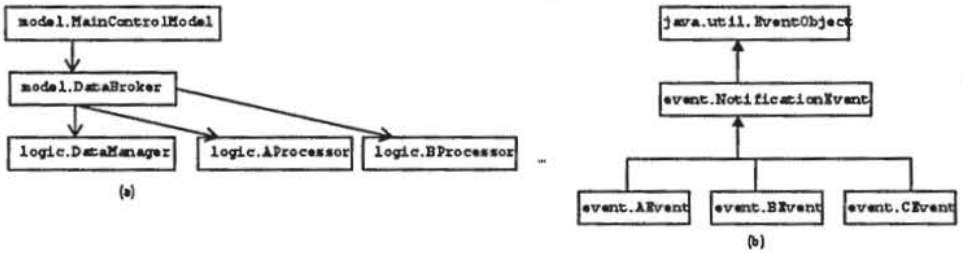

Figure 26 (a). An Arrow denotes a call to a constructor (i.e. an object creation). DataBroker creates an instance of DataManager and also several processors. MainControlModel creates an instance of the DataBroker. Using the DataBroker instance, the MainControlMo del can get references to processors in order to send data to the remote server. (b). Arrows denote the inheritance relation. The event driven architecture is followed to notify events to GUI panels based on the response from the remote server.

Because the `mocclient` uses the `java.util.EventObject` class, the analyst hypothesized that the GUI architecture is influenced by an event driven architecture. The extracted inheritance structure further supported that claim because one of the base classes of the SNAS inherits from the `EventObject`, see Figure 26 (b). The analyst queried the extracted dependency model to locate the classes that use the `NotificationEvent` class. The query showed that the `DataManager` class calls the constructors of children of the `NotificationEvent` class. The analyst also determined that if the `DataManager` receives, for example, a `PasswordExpired` object from the `sam` server, then it creates a `PasswordExpiredEvent` instance and fires this event. All panels that are registered for this `PasswordExpiredEvent` will then get notified. Similarly, panels and processors can also create events, and other registered panels and data processors will get notified.

Above we discussed how the analyst discovered answers to the analysis questions such as a) how panels are controlled, b) how panels get data from the server, c) how panels get notified when data comes from the server. Based on these answers, the analyst concluded that he discovered the architectural skeleton where all 750 files of clients could be plugged-in. We often do not need to understand and review all 750 files of clients, however, we do need to locate and understand the file that controls all panels, the file that sends and receives data from remote servers, and the file that handles event

processing. This analysis showed that knowledge of external dependencies can help in locating those files that spans or controls the GUI sub-space.

Some design decisions invite bugs – the SNAS Maintainer's view: The `DataBroker` is one place where objects are being reused and data is getting mixed up. For example, `ReportProcessor` has a single instance in `DataBroker`, which causes problems if the user requests a second report before the first one is finished. Other processors are used by multiple windows, so the user could be doing what appears to him/her to be two entirely separate operations, but because `DataBroker` only has one instance of the processor, the operations sometimes interfere with each other. To allow the user to perform multiple operations at once, the whole `DataBroker` class should likely be removed—or, at least, the singleton pattern should replaced by multiple instances so a new processor is returned instead of reusing the existing singleton one. Also, the fact that almost all of the `DataManager's` fields are 'static' can cause issues when the user logs out and logs back in very quickly (using Logout/Login on Main Menu, instead of Exit and restarting from scratch)—the re-initialization of the values may occur before the cleanup of the old values, and this can cause a number of problems. This is because, even though a new `DataManager` instance is created if the user Logout/Login, the static variables are retained because they are initialized only once by the Java Virtual Machine (JVM), in addition the cleanup process and the login process run in different threads.

3.6.3. GUI Architecture and Performance Analysis using External Dependencies

This subsection demonstrates how the knowledge of external entities can help in locating architectural decisions that have potential performance risks.

3.6.3.1. Some Performance Problems in Event Notification Architectural Style

In order to keep the GUI responsive, the threading model and the event dispatchers need to be carefully designed and the analyst wanted to analyze how that part was constructed. The analyst used the fact that the knowledge base knows that the `javax.swing.event.EventListenerList` class is typically used to store the list of event listeners. Using that knowledge, the analyst discovered the method that calls the event listeners, see Figure 27. The analyst determined that there is a problem with this solution: if any one of the event listeners has a slow `eventStarted` method, it will affect other listeners too because all event listeners `eventStarted` methods are called in the same thread synchronously. If a new event listener is introduced into the system and its `eventStarted` method is slow, then the entire system has the risk of slowing down. The analyst noted that Java 5 has a new flexible threading model to exactly solve this synchronous event dispatching problem. The class `java.util.Executor` allows listeners to be executed asynchronously, using the concept of thread pools, so that slow listeners do not affect other event listeners. The SNAS team revealed that they are facing this issue in the current version and therefore the proposed solution, which takes advantage of the services of Java 5's `Executor`, is being considered for the upcoming release.

```
// This method is used to fire NotificationEvents to all listeners

public static void fireNotificationEventStarted(NotificationEvent evt) {
   Object[] listeners = SnasFrame.getListener.getListenerList();
   for (int i = 0; i < listeners.length; i+=2) {
      if(listeners[i] == NotificationEventListener.class) {
         listeners[i].eventStarted(evt);
      }
   }
}
```

(a)

```
SnasFrame

javax.swing.event.EventListenerList getListener(){
  return listeners;
}
```

(b)

Figure 27 (a) Performance risk if some listeners have a slow implementation of the `eventStarted` method they will affect other listeners. (b). This performance issue was detected using dependencies to the `javax.swing.event.EventListenerList` class.

3.6.3.2. Some Performance Problems Due to Thread Models and Socket Timeout

The analyst discovered above that the `DataManager` class is responsible for reading responses from the socket connected to the remote server, and delegating these messages to appropriate processing classes, see Figure 25 (a). The analyst tried to understand the threading strategy used for reading the data from the socket and dispatching it to the appropriate processors. To achieve this, the analyst reviewed the `run` method within the `DataManager` that reads data from the socket. The analyst revealed that the method uses the same thread for reading data from the socket and also synchronously dispatching it to the data processors. The analyst then concluded that due to this synchronous threading model, slow performing methods of data processor classes may hurt the entire system since data cannot be read from the socket until control returns from the data processors to the `DataManager`.

We discussed this potential issue with the SNAS team, and they acknowledged this problem and even mentioned that the socket timeout happens before processing all data in the socket due to synchronous method calls. We are discussing the possibility of either introducing a thread pool design pattern using the `java.util.Executor` class to resolve this performance problem, or to introduce additional queues so that the data from the socket can be just transferred to different queues, and thus socket timeout can be avoided. The SNAS team is evaluating these solutions for the next release.

3.6.4. GUI Architecture and Testability Analysis using External Dependencies

The analyst discovered earlier that every panel class has an associated data model, see Figure 24. When the analyst reviewed some of the panel classes he observed that each of them has a small `main` method which populates a dummy data model. The analyst queried the extracted dependency model and found that nearly all panel classes contain a `main` method. Based on these findings, the analyst concluded that each panel can be tested without running the whole system.

The analyst already determined that the SNAS uses the `JPanel` class, which allows users to input data into various fields, and therefore wonders: Does the SNAS use the input verifier capability built-in the Java Swing architecture? The analyst therefore queried the extracted code relations and found that the SNAS uses the `javax.swing.InputVerifier`, and overrides the call-back `verify` method as demanded by the Swing architecture. The analyst concluded, however, that the logic behind the `verify` method is not that trivial, which means it has to tested well, see Figure 28 (a). Unfortunately, the `verify` method is not easily testable without running the GUI and filling the input into the fields of panels. The analyst thus concluded that the risk is that the `verify` method is not tested in-depth as required (e.g. using JUnit) because it assumes that the data is provided by a GUI panel. Most readers will agree that

regular expressions can be error-prone, and unit test programs are needed to test them. If we refactor the `verify` method, as in Figure 28 (b), thereby separating the GUI concept from the validation of the IP address, then the method `isValidIpAddress` can be easily tested using the JUnit test framework, for example, see Figure 28 (c). This analysis, with the help of the knowledge base on Java's Swing libraries, also detected other panels that verify the user input, such as range constraints, numeric constraints, and alpha-numeric constraints.

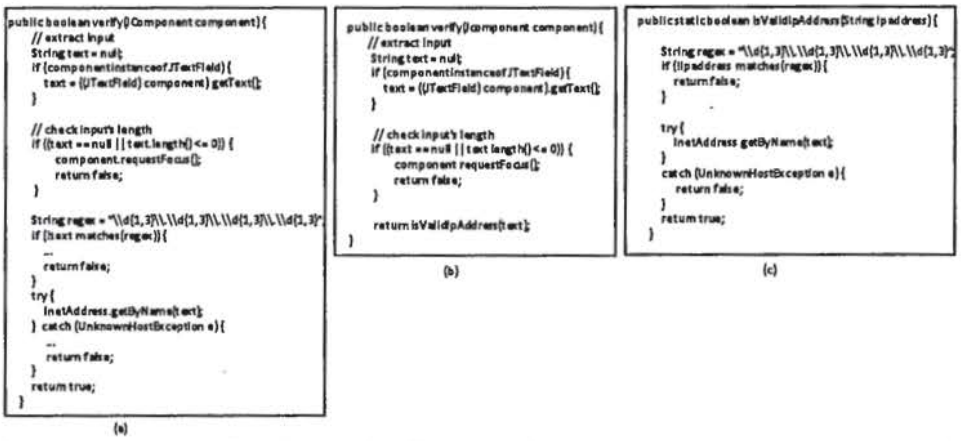

Figure 28 (a). A testability issue because the validation logic is mixed with GUI concepts. (b). A better version of the `verify` method we proposed to the SNAS team, that separates the gui from validation logic for improved testability. (c). The Validation of a IP address can now be tested without the gui being up and running!

To sum up, even if one tries to construct a JUnit test suite, the source code has to be "open" for testing; in the sense GUI concepts ought to be separated from logic. The principle of abstraction and separation of concerns is one of the foundational pillars of software engineering [Parnas 1985, Tarr 1999], but perhaps developers (also code reviewers) either overlook this fact or there is a lack of concrete examples to really understand the concrete meaning behind this principle in order to apply in practice. That's why this paper uses code snippets to demonstrate fundamental software engineering principles. Only because of the knowledge base of Java's GUI classes, it was possible for us to easily discover code elements that affect testability.

3.6.5. GUI Architecture and Clones

Experience tells us that clones often exist in the GUI portion of the system. Thus, the analyst used the similarity tool to automatically analyze all files of the `mocclient` and `oamclient` subsystem, and discovered that nearly every file that is present in the `mocclient` is also present in the `oamclient`, with very small differences in content.

Discussions with the SNAS team revealed that the `mocclient` and `oamclient` are variants for two different groups of users. The analysis pointed out there is a still a large number of GUI related files that could be moved to the `shareclient` folder in order to minimize duplication. The similarity tool also reported clones between GUI panels within these subsystems. The problem is that in order to develop a GUI panel, there are basic boiler-plate code elements (e.g. defining the GUI panel structure, fields, buttons, etc) that every program must implement, which naturally results in code cloning. It is not easy to

solve this cloning problem. However, other projects that we have analyzed have positive experiences with generating Java Swing GUI code using the Jigloo Editor. This was recommended to the team so that, at least in the future, new panels can be generated using such code generators thereby avoiding copy of bugs due to copy-and-paste of code. Now the analyst proceeds to analyzing the OS interaction strategy using the external dependencies of the SNAS.

3.7. An Analysis of the OS Concern using External Dependencies

The model on which we base the analysis of the OS concern is based on the following observations: a) any software system needs an Operating System to run and some need to run on more than one operating system, and b) there should be some strategy (good or bad) to manage OS variants. From this model, we derive the following questions: 1) What are the different OS types the system supports? 2) How does the system abstract underlying OS and when does binding to a particular OS take place? and 3) How are OS concerns separated from other concerns?

For the Java programming language, the snippet of the knowledge base, shown in Figure 29, helps in discovering architectural insights from the OS perspective.

OS Concerns	File Separator	java.lang.System.getProperty(file.separator)	Returns '/' for Unix, and '\' for Windows
	Path Separator	java.lang.System.getProperty(path separator)	Returns ':' for Unix, and ';' for Windows
	OS Architecture	java.lang System getProperty(os.arch)	Returns the architecture of the machine
	OS Command	java.lang.Runtime.exec	Executa the given OS command
	OS Name	java.lang System.getProperty(os name)	The name of the OS that runs the Java program
	OS Version	java.lang System.getProperty(os.version)	Returns the version of the OS

Figure 29 A Snippet of the Knowledge base for analyses of OS concerns.

The analyst queried the extracted dependency model of the SNAS and discovered the files that use the `exec` method of the `java.lang.Runtime` class that helps for interacting with the OS (see Figure 29). The query showed that all OS commands are executed only through the `PidService` class (see Figure 30 (a)) defined in the `framework` folder. The analyst reviewed this class and found that the OS type and the OS command to run must be passed as arguments to the methods of `PidService` (see method parameters of Figure 30 (a)). The analyst extracted all dependencies to the methods of the `PidService`. The extracted dependency diagram showed that the higher-level layers ((see Figure 30 (b)) must pass the OS command and the OS type to run the methods of the `PidService`. This clearly implies that OS concerns penetrated and mixed with other concerns, and the architecture did not offer a separate OS abstraction interface that hides the actual OS type.

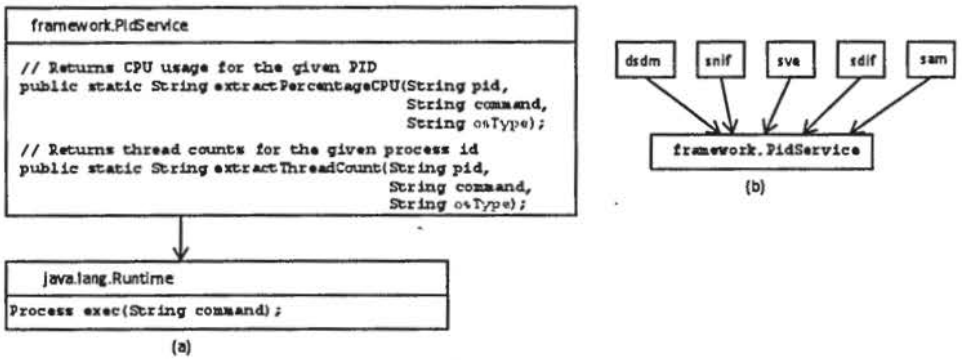

Figure 30 (a). All OS commands are executed using the `PidService` class defined in the framework. (b). However, the higher-level layers are also dealing with OS concerns such as the OS type and the OS command to run (see the parameters of methods of `PidService`). This problem can be solved if `PidService` uses Java's APIs to discover automatically the current OS type and then decide which OS command to run.

The recommended solution to this OS variation management problem is to let the `PidService` use Java's APIs (e.g. `System.getProperty(os.name)`) and automatically discover the OS type and then decide what OS commands to run. By doing so, the higher-level layers will become agnostic to OS variations. Thus, the complexity due to managing OS variants can be controlled in a cleaner and consistent way across all subsystems, resulting in a good common look-and-feel. This example reminds us that having a system implemented in Java does not necessarily imply the system is ready-to-run in all OS platforms; it has to be architected to manage OS variants. The devil is in the details, by slicing the implementation using external dependencies we were able to discover novel insights and deep architectural problems, and furthermore offer constructive solutions where possible.

The analyst did not spend effort on identifying the different types of OS supported by the SNAS because when he reviewed some of the configuration files used for configuring IP address, ports, etc, his eyes also saw a configuration variable called `osType`, with commented lines such as "set `osType`=Unix or set `osType`=Windows". Thus the analyst admitted that it was some luck that pointed him to the OS types used in the SNAS. Otherwise, he had a strategy of searching of "/" or "\" used file names, for example, to identify OS types. It was commented in the configuration files that for testing purposes one can choose the OS type by modifying configuration parameters.

3.8 Summary

3.8.1. Summary on Wrapper Violations and Heritage of the SNAS

During this analysis, the analyst discovered many potential rules such as: All usages of the logging COTS (Apache Log4J) should be done using the logging service wrapper defined in the `framework` folder, and: All usages of threads should be done through a wrapper class defined in the framework. However, the analyst noticed that the `sdif` subsystem violated many of the potential rules. In fact, the `sdif` has its own wrapper class offering logging service. The analyst also discovered during the analysis of the database concern that the `sdif` has a different style for implementing database connections and the DAO layer. Basically, the `sdif` lives in its own world. Discussions with the SNAS team revealed that the `sdif` was taken from the predecessor of the SNAS called SWSI. The SNAS team has done a lot of refactoring to integrate the `sdif` with the

SNAS, but time was not spent on cleaning-up the architectural differences with other subsystems. We noted earlier some subsystems by-pass wrappers to sockets, defined in the `framework` folder, and directly use Java socket libraries. Discussions with the SNAS team revealed that wrappers were not there in the first-place, and therefore it is not practical to expect that all subsystems use the wrappers. These are some "classical" examples of individual parts looking good but if we take one step up and see the whole system, there are architectural deviations in common look-and-feel, partly because of organizational factors and migrating or merging existing systems.

3.8.2. Can Two Requirements influence the Architecture and Size of the System so much?

This analysis shows that the SNAS contains many performance-oriented architectural design decisions. For example, a) it uses the transfer object design pattern to overcome communication overhead due to remote procedure calls, b) it uses multiple ports to reduce the waiting time of objects in socket channels, c) it uses a reactor design pattern to facilitate non-blocking I/O of socket channels, and d) it uses a database connection pool to reduce the overhead of frequently creating and destroying connections. Naturally, we were more than curious and asked the SNAS team what is the real need for this elaborated architectural design. They pointed to two requirements (see Figure 31) out of the many hundreds. We can notice that both requirements deal with timing aspects.

```
3.4.1.6 The SNAS shall not exceed two seconds from the receipt of a message to the transmission of that message.
3.4.1.7 The SNAS shall not exceed five seconds from the receipt of a request to retrieve data from the SNAS database to
the transmission of the result.
```

Figure 31 The two requirements (snippet) behind the elaborated performance driven architecture.

The number of transfer objects (or bean) files is 384 out of 1578 Java files. The beans contribute 80KLOC (including comments and spaces) out of 650KLOC. We mentioned in the GUI analysis that the SNAS uses event-driven architectural style by creating event objects for each type of incoming object from the remote server. For example, if the incoming object is `LoginFailed` type then the corresponding `LoginFailedEvent` object is created and notified to registered panels to display the login failure message. There are around 60 event files, contributing 10KLOC. We also noted earlier that the GUI architecture has data models for each panel. The data models contribute 40KLOC out of 650KLOC. Unfortunately, there is not much difference between data models and transfer objects, except the latter implements the `Serializable` interface to transfer objects across the network. They both contain data attributes with `get` and `set` methods. Also note that GUI data are present inside Java's internal model of Swing. In addition, database tables contain the same data present in data models and transfer objects. Basically, data are redundantly present in different formats. This analysis offered insights on the influence of the transfer object design, mainly chosen for overcoming the inherent limitations of remote procedure call, on the overall size of the system. We are discussing with the requirements team in order to understand the rationale behind the timing values and how stringent they must be followed. Also, we are discussing whether or not anyone measured the running system's response time to satisfy those timing constraints. The major lesson from this study is that requirements analysts (also architects) must be careful in specifying and analyzing timing constraints; otherwise there is a danger of over-engineering with an "elaborated" architecture and a lot of source code to develop, test, maintain, and evolve.

3.8.2. Summary of the Number of Files Reviewed

The analyst took notes on the number of files he has reviewed during the analysis of the SNAS. The analyst mentioned that he has good experience with the RPA query language [Feijs et al. 1998, Krikhaar 1999, and Ganesan et al. 2009] used for automatically finding files with certain characteristics. For example, using RPA it is straightforward to detect all files that directly or indirectly (i.e. through inheritance) implement the `java.awt.event.ActionListener` interface. If the analyst is not familiar with RPA-like query languages, then he has to use some other source code search tool and may have to open many more files. In addition, the analyst agreed that there is a learning effect that fortunately reduced the number of files to be reviewed. For example, when the analyst opened a file to understand how data from the socket is read and delegated to data processors, his eyes also saw other concerns such as the error-handling strategy used for automatically reconnecting to the remote server if the connection is lost for some reasons. That helped the analyst because he learned more than what he was originally intending to do with that file. Table 3 shows the number of files reviewed for each goal. Although this is honestly collected data, the analyst agreed he might have forgotten to count some files, but the paper has provided abundant evidence that the dependencies on external entities can be of great value in finding the right entry points into the system.

Table 3 Summary of files reviewed for each goal

Goals	**# of Files Manually Reviewed**
Discovery of Server-side Socket Ports	10 out of 1578 Java files
Discovery of Client-side Socket Ports	12 out of 1578 Java files
Discovery of Port Connections	5 out of 25 configuration files
Discovery of Data Beans	5 out of 384 Java bean files
Discovery of the DAO layer	10 out of 112 Java files dealing with database interaction
Discovery of GUI Architecture	20 out of 723 Java files in the mocclient, oamclient, shareclient folders
Discovery of OS Variability	4 out of 1578 Java file
Total	**61 out of 1578 Java files (i.e. 4% of Java files) were reviewed**

3.8.3. Discussion

Can the analysis be done in *any* order? Yes, we believe that the method of following external dependencies into the application can be done in any order and for any purpose. For example, the analyst could have applied the AIS method first for discovering the architecture of the GUI concern and then for the Persistence concern. However, the analyst found it useful to first discover the component-connector view before slicing the implementation based on concerns. For example, if the analyst did not know the fact that the SNAS follows a distributed architecture using sockets as connectors, then the analyst may not have known easily the fact that the GUI parts send and receive data to a remote server.

Are there threats to the validity of the architecture discovered using the AIS method? There may be some threats to the classification of roles played by a huge collection of files based on collected evidences. As we noted earlier, for example, the analyst without reviewing 384 Java files claimed that all of them are data beans because the criteria of a class being considered as a bean is based on characteristics collected from his experience (e.g. almost all methods are getters and setters, no calls to logging, implements `Serializable` interface, etc). There may be some files within the 384 files

which are doing more than what a data bean is suppose to do. A common threat in any reverse engineering method is the correctness and completeness of the code relations extracted by parsers.

Is the AIS method flexible? The method is flexible because the analyst can decide, based on his goals, which subspace of the four dimensional space is appropriate to work (recall Figure 1). The analyst can select a concern of his interest; refine it to the necessary depth. The method neither enforces the order in which the analyst has to select a concern nor the level of depth in refining and analyzing the selected concern. The external dependencies can still support the analyst even if he wants to go further deep and analyze, for example, how all GUI panels implement windows close action, which are also often implemented using the APIs of the underlying external entities. Almost all programming language libraries support common concerns and architecture connectors. Therefore, we are more than confident to claim that external dependencies play a novel role in reverse engineering.

Is the AIS method repeatable on other systems and/or other languages? Although the case study is in Java, the method has evolved from analyses of several systems implemented in the C/C++, ADA, and FORTRAN languages [Stratton et al. 2007, Ganesan et al. 2009, Lindvall et al. 2010]. Yes, the analysis questions and the reasoning process are certainly repeatable on other systems. Experiences with formal query languages and basic knowledge of programming language libraries are important in order for other analysts to repeat and produce the same results discussed in the case study. The knowledge base of external entities can help analysts who are not familiar with the classes and methods of programming language libraries. In general, analysts are like a detective, the method is trying to codify, reuse, and share experiences so that they can analyze commercial strength systems efficiently.

4. Comparison to Existing work

Here, we will highlight some key differences between our method and the existing work related to architecture analysis, knowledge-based reverse engineering, pattern-based architecture discovery, clustering, software clones, exception handling, and testability.

Architecture-Analysis in the early phase of the lifecycle: The SAAM method uses "what-if" scenarios to evaluate the proposed architecture in a workshop with the project team [Kazman 1994, Clements 1995]. Their method was designed to be applied in the early phases before the implementation starts and thus architectural risks can be identified early. We believe our architecture analysis method complements the SAAM because it helps in discovering the implemented architecture which could be used in conjunction with their method for analyzing evolving systems. [Rozanski and Woods 2005] use a catalog of concerns to construct and analyze views of the architecture. We also use such a catalog but to slice the implementation and discover the architecture for each concern. [Martin 2005] discusses architectural principles for bypassing the GUI and database for testing. [Binder 1994] discusses design principles for testability. [Wirfs-brock 2006] discusses best practices for managing exceptions, including abstraction of exceptions raised by lower-level layers. We use these principles and best practices for evaluations of implemented systems. We share the vision of storing best (or problematic) practices with Booch's handbook of software architectures [Booch].

Knowledge-based Reverse Engineering: The PROUST tool takes as input a text description of a program and uses its knowledge base for detecting errors novice programmers make and help them correct their mistakes [Soloway and Johnson 1985].

The PAT tool generates a high-level specification (i.e. an algorithm) of a given program using its rule-based inference engine [Harandi and Ning 1990]. In contrast to our approach, these approaches are small scale as they focus on understanding and generating specifications at the sub-routine level, while our approach focuses on understanding large systems.

We share the LaSSIE's high-level goal of solving the "invisibility" problem inherent in software systems [Devanbu 1991]. LaSSIE helps in understanding of how and where features are implemented using its domain ontology. In our opinion, LaSSIE does not focus on discovering testability and performance risks as discussed in our method. We are exploring ways to enrich our knowledge base with domain concepts for facilitating a domain-oriented architectural reasoning. The MIDAS approach uses a knowledge base for automatic reengineering of database programs from the network model to the relational model [Chiang 1995]. Our method supports discovering and analyzing the database interaction architecture and its testability. The MORPH process uses a knowledge base for migrating text-based user interfaces into GUIs [Moore and Rugaber 1997]. Our method supports discovering and analyzing the GUI architecture of systems that already have a GUI. [Michail 2002] uses GUI messages and function names of GUI frameworks used by the system under analysis for browsing and searching the source code in order to overcome limitations of general text search tools. MicroScope is a tool-suite for maintenance activities. Our future work can benefit from its rule-based inference engine [Ambras and O'Day 1988]. MicroScope can also benefit from our knowledge-base that helps in discovery of software architectures.

Pattern-based Architecture Discovery: [Dong et al. 2007] review methods and research tools for recognition of design patterns from the source code. We have shown where several patterns were implemented simply by using the programming language libraries, which are often excluded in many research methods. It would be interesting to investigate how pattern discovery methods can benefit from a knowledge base of external dependencies. [Harris et al. 1995] constructs a library of architecture concepts recognizers in the source code. A challenge, in our experience, is that it is difficult to codify the way different systems implement the same architecture concepts. In some cases, the same architecture concept might have been implemented in different styles by different subsystems, as is the case for the database abstraction layer of the SNAS, for example. Thus, it is difficult to automatically discover architecture concepts.

Clustering: [Maqbool and Babri 2007] provided a long discussion on clustering methods and how they could shed some light on the software structure. One of the challenges is that, especially in GUI parts, function calls occur indirectly using event-driven concepts and implicit invocations, and thus the call graph is often broken into disconnected graphs. Also, many systems contain intermediate connectors for communication. Another challenge is that the architecture concepts (e.g. Interfaces, Connectors, and Components) are invisible in the output of clustering, and is not easy to do a detailed analysis, because all concerns are still part of the clustered model. In general, clustering methods do not give names to subsystems or summarize in a few sentences the role played (e.g. DAO layer, OSAL layer) by them. After all, it is the name and the brief summary that helps in understanding the architectural roles played by a huge collection of files. It would be fruitful to investigate how the existing clustering methods behave if they are combined with a knowledge base of external dependencies.

Software Clones: [Koschke 2007] discusses several clone detection methods in detail. Our focus is on analyzing the detected clones by concentrating on one concern at a

time and interpreting them using the discovered software architecture so that we can offer constructive advices where possible. For example, we have discussed clones in GUI panels and across files in database abstraction layers. We offered concrete advices on how to migrate to new technologies in order to overcome inherent cloning problems due to the Java language.

Exception Handling: The Jex tool was used for analyzing the flow of exceptions [Robillard and Murphy 1999]. We analyzed exceptions using dependencies to external entities and selecting a concern of interest. We interpreted the flow of exceptions from an architecture point of view. For example, we have shown cases where the database is abstracted but database errors had leaked into the higher-level layers. Also, using the knowledge of dependencies to external entities, we have shown how we can find how the system handles specific exception types such as the socket timeout exception or host not available exception. Thus, we believe Jex can also benefit from a knowledge base. [Shah et al. 2010] reported that, in their survey, novices make mistakes in exception handling. Of course, the truth is in the source code, experts also make mistakes because exception handling is often not given much attention during the architecture design.

Assessment of Testability: We share the spirit of understanding "What is it that makes code hard to test" as [Bruntink and Deursen 2004] formulates this important question. In contrast to their testability assessment model, our method covers testability in the presence of a GUI or a database. [Feathers 2004] offers a piece of "clean" code (e.g. good method/variable names, comments) that was not easy to test because of a hard-binding to a remote stock server, which cannot be replaced by a dummy server for testing purposes. We collect such anti-testing patterns into our knowledge base and analyze implemented systems for the existence. [Ganesan et al. 2010] provide insights on "What types of architectural decisions make unit testing easier/harder" in a product line context.

Closing Remarks

Yes, external entities offer novel insights during reverse engineering. This paper has offered abundant evidence that by leveraging the semantics of external entities, we can efficiently discover the software architecture hidden in the implementation. Most architectural problems are hidden deep in the source code. As shown in this paper, external entities help us to efficiently locate the details where devils hide. The paper also disclosed a knowledge base for reverse engineering. Construction of a knowledge base is an investment. We have shown how one can incrementally build a knowledge base over time using external entities used by systems under analysis. If your organization is regularly conducting architectural analysis of several implemented systems, you could reap the benefits of your investment in a knowledge base. Our future prospects include a) improving the usability of the tool-chains so that analysts can easily add their knowledge of analyzing commercial systems, and b) building "intelligent" analysis environments to further improve the productivity of our analysts.

REFERENCES

ALUR, D., CRUPI, J., AND MALKS, D. 2003. Core J2EE Patterns. Sun Microsystems press.

AMBRAS, J. AND O'DAY, V. 1988. MicroScope: A Knowledge-Based Programming Environment. IEEE Software, 5(3), 50-58.

Apache DBCP. Open Source Database Connection Pool, http://commons.apache.org/dbcp/

BASILI, V., CALDIERA, G., McGARRY, F., PAJERSKI, R., PAGE, G., AND WALIGORA, S. 1992. The Software Engineering Laboratory: An Operational Software Experience Factory. Proceedings of ICSE, 370-381.

BINDER, R.1994. Design for testability in object-oriented systems. Communication of the ACM, 37(9), 87–101.
BRUNTINK, M. AND DEURSEN, A.V. 2004. Predicting Class Testability using Object-Oriented Metrics. Proceedings of the Source Code Analysis and Manipulation Conference, 136-145.
BOOCH, G. http://www.handbookofsoftwarearchitecture.com.
CHEN, Y.-F., NISHIMOTO, M. Y., AND RAMAMOORTHY, C. 1990. The C information abstraction system. IEEE Transaction on Software Engineering, 16, 3, 325–334.
CHIANG, H.L.R. 1995. A knowledge-based system for performing reverse engineering of relational databases. Decision Support Systems, 13, 295-312.
CLEMENTS, P., BASS, L., KAZMAN, R., AND ABOWD, G. 1995. Predicting Software Quality by Architectural-Level Evaluation. Proceedings of the Conference on Software Quality. 485-497.
DEVANBU, P., BRACHMAN, R.J., SELFRIDGE, P.G., BALLARD, B.W. 1991. LaSSIE: a Knowledge-based Software Information System. Communication of the ACM, 34(5), 34-49.
DONG, J., ZHAO, Y., AND PENG, T. 2007. Architecture and Design Pattern Discovery Techniques – A Review. International Conference on Software Engineering Research and Practice, 621-627.
FEATHERS, M. 2004. Before Clarity. IEEE Software, 21(6), 86-88.
FEIJS, L., KRIKHAAR, R., AND OMMERING, R. 1998. A Relational Approach to Support Software Architecture Analysis. Software Practice and Experience, 28(4), 371-400.
FLOWER, M., BECK, M., BRANT, J., OPDYKE, W., AND ROBERTS, D. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.
GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. Design Patterns—Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley.
GANESAN, D., LINDVALL, M., McCOMAS, D., AND BARTHOLOMEW., M. 2009. Verifying architectural design rules of the flight software product line. Proceedings of the Software Product Line Conference, 161-170.
GANESAN, D., LINDVALL, M., McCOMAS, D., BARTHOLOMEW, M., SLEGEL, S, AND MEDINA, B. 2010. Architecture-based Unit testing of the flight software product line. Proceedings of the Software Product Line Conference.
Google's Guice Framework, http://code.google.com/p/google-guice/
HARANDI, M.H. AND NING, J.Q. 1990. Knowledge-Based Program Analysis. IEEE Software, 7(1), 74-81.
HARRIS, D.R., REUBENSTEIN, H.B., AND YEH, A.S. 1995. Reverse Engineering to the Architectural Level, Proceedings of ICSE. 186-195.
Hibernate Framework, http://www.hibernate.org/
JACOBSON, I. 1992. Object Oriented Software Engineering, Addison-Wesley.
KAZMAN, R., BASS, L., ABOWD, G., AND WEBB, M. 1994. SAAM: A Method for Analyzing the Properties of Software Architectures, Proceedings of ICSE, 81-90.
KOSCHKE, R. 2007. Survey of Research on Software Clones. Proceedings of Dagstuhl Seminar 06301.
KRIKHAAR, R. 1999. Software Architecture Reconstruction, PhD Thesis, University of Amsterdam.
KRUCHTEN, P., OBBINK, H., AND STAFFORD, J. 2006. The Past, Present, and Future of Software Architecture. IEEE Software, 23(2), 22-30.
LINDVALL, M. 2010. Connecting research and practice: an experience report on research infusion with software architecture visualization and evaluation. NASA's Journal on Innovations in Systems and Software Engineering.
MAQBOOL, O. AND BABRI, H. 2007. Hierarchical Clustering for Software Architecture Recovery. IEEE Trans. Software Eng., 33(11), 759–778.
MARTIN, R.C. 2005. The Test Bus Imperative: Architectures that support automated acceptance testing. IEEE Software, 22(4), 65-67.
MICHAIL, A. 2002. Browsing and searching source code of applications written using a GUI framework. Proceedings of ICSE. 327–337.
MOORE, M. AND RUGABER, S.1997. Using Knowledge Representation to Understand Interactive Systems. Proceedings of the International Workshop on Program Comprehension, 60-67.
MURPHY, G.C., NOTKIN, D., AND SULLIVAN, K. 2001. Software Reflexion Models: Bridging the Gap between Design and Implementation. IEEE Transactions on Software Engineering, 27(4), 364-380.
NACCARATO, G. 2002. Introducing Nonblocking Sockets. O'Reily Publications.
PARNAS, D.L., CLEMENTS, P., AND WEISS, D. 1985. The Modular Structure of Complex Systems. IEEE Trans. Software Eng., 11(3), 259–266.
ROBILLARD, P.M. AND MURPHY, C.G. 1999. Analyzing Exception Flow in Java Programs. Proceedings of ESEC/FSE, 322–337.
ROZANSKI, N. AND WOODS, E. 2005. Software Systems Architecture. Addison-Wesley.
SCHMIDT, D. 1995. Using Design Patterns to Develop Reusable Object-Oriented Communication Software. CACM, 38(10), 65-74.

SHAH, B. H., GOERG, C., AND HARROLD, M.J. 2010. Understanding Exception Handling: Viewpoints of Novices and Experts. IEEE Transaction on Software Engineering, 36(2), 150-161.
SHAW, M. AND CLEMENTS, P. 2006. The Golden Age of Software Architecture. IEEE Software, 23(2), 31-39.
SHAW, M. AND GARLAN, D. 1996. Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall.
SOLOWAY, E. AND JOHNSON, W.L. 1985. PROUST: Knowledge-Based Program Understanding. IEEE Transactions on Software Engineering, 11(3), 267-275.
Spring Framework, http://www.springsource.org/
STRATTON, W., SIBOL, D., LINDVALL, M., AND COSTA, P. 2007. The SAVE Tool and Process Applied to Ground Software Development at JHU/APL: An Experience Report on Technology Infusion. SEW, 187-193.
TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, S. M. 1999. N degrees of separation: Multidimensional separation of concerns. Proceedings of ICSE, 107–119.
WALDO, J., WYANT, G., WOLLRATH, A., AND KENDALL, S. 1994. A note on Distributed Computing. Sun Microsystems, TR-94-29.
WIRFS-BROCK, R.J. 2006. Toward Exception-Handling Best Practices and Patterns. IEEE Software, 23(5), 11-13.