A Hierarchy of Dynamic Software Views: from object-interactions to feature-interactions

Maher Salah and Spiros Mancoridis
Department of Computer Science
College of Engineering
Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
msalah@cs.drexel.edu, Spiros.Mancoridis@drexel.edu

Abstract

This paper presents a hierarchy of dynamic views that is constructed using tools that analyze program execution traces. At the highest-level of abstraction are the feature-interaction and implementation views, which track the inter-feature dependencies as well as the classes that implement these features. At the middle-level is the class-interaction view, which is an abstract view of the object-interactions. The object-interaction view is the base view for all the views, and captures the low-level runtime interactions between objects. Two case studies are used to demonstrate the effectiveness of our work.

1 Motivation

An important, but difficult, problem in software comprehension is the identification of program features, and the traceability of these features to program source code. This problem has been studied extensively [6, 24, 25], with an emphasis on techniques for mapping program features to source code. In this paper, we complement these efforts by providing a set of views that not only associates features to code but also establishes feature-interactions automatically.

Over two thirds of software maintenance efforts are spent on non-corrective maintenance activities, mainly perfective and adaptive maintenance [18, 23]. Perfective maintenance is performed on software to meet new users requirements. Adaptive maintenance is performed in response to modifications in the environment. Perfective or adaptive maintenance activities typically involve the study and analysis of specific portions of the system to fix bugs, add new features, or modify existing features.

To perform a maintenance activity, a developer's initial task is to study and analyze the source code and its documentation. For example, the task may be to modify the 'print' feature of a word processing application. The developer studies the source code to locate the portions that are related to the 'print' feature. For many software systems, this task is difficult and time consuming, since the implementation of a feature may involve multiple classes or modules. More than half of a typical developer's effort is spent on reading and analyzing the source code to understand the system's logic and behavior [16, 18].

A developer can take an alternative approach to this maintenance task by instrumenting the application's code, and exercising the subject feature using a profiler, and then analyzing the execution traces to determine which portions of the code were exercised by the feature. An *execution trace* is a sequence of runtime events (*e.g.*, object creation/deletion, method invocation, thread creation/termination) that describes the dynamic behavior of a running program. With adequate tool support, this approach is more effective, because it reduces the complexity of the task by allowing the developer to locate the code of interest quickly. This approach is suitable for many practical settings, since change requests are usually written in plain langauge with explicit references to identifiable program features. A developer can start from the change request, then execute the application in a profiling mode, and finally exercise the desired features to locate the portions of the source code, instead of starting with the code and trying to map it to features manually.

Dynamic analysis is a powerful technique for identifying the portions of the code that need to be changed. Specifically, the developer can execute the system and mark execution traces of interest. This paper describes a set of software views and tools that support this approach to software understanding. The approach is centered around the concept of marked execution traces, which developers use to define program features. In our technique, a *feature* is defined as

a functionality, or a usage scenario of a program, that can be represented as a sequence of runtime events. Features are specified by the developer in terms of marked-traces. A *marked-trace* is established manually during the execution of the program by specifying the start and the end of the trace using a trace-marker utility that is part of the profiler.

The set of views provides various levels of detail that the developer can explore. The tools support the navigation among the different views as well as the expanding of the nodes and edges of the graphs of the views to allow the developer to view specific detailed information that is encoded in the nodes and edges. Depending on the view, a node can be a feature, a thread, a class, an object, or a method. Similarly, an edge encodes the relationship type, and the objects and classes that are participating in the relationship.

The organization of the rest of the paper is as follows: Section 2 describes related research and the architecture of the tools used, Section 3 describes the dynamic views, and Section 4 describe the case study.

2 Background

This work is related to two research areas. Namely dynamic analysis and program feature analysis. The work also depends on our earlier research pertaining to the design of a software comprehension framework.

2.1 Dynamic analysis

Dynamic analysis is used to study the behavior of software systems. There are three methods for collecting runtime data. The first method is source code instrumentation. Bruegge *et al.* designed the BEE++ system [3] as a framework for monitoring (*e.g.,* function calls, variable modifications) systems written in C/C++. In this system, runtime event generation is achieved by instrumenting the program source code. Another system that uses source code instrumentation is SCED [10]. This system uses runtime data to create models of object-oriented programs, which are visualized as state diagrams or state charts. SCED only collects data from stand-alone applications, while BEE++ can also collect data from distributed applications.

The second method for collecting runtime data involves the instrumentation of complied code. This method is widely used to instrument Java bytecode [11]. The third runtime data collection method is based on debugging and profiling. In this method, code instrumentation is not required. Modern development frameworks provide interfaces to facilitate the collection of runtime data. Examples of such interfaces include JVMDI and JVMPI for Java [21, 22], CLR Profiling Services for Microsoft .NET [14], and COM+ instrumentation services for COM+ applications [13]. Debuggers have been used to emulate profiling interfaces by automatically inserting breakpoints and manipulating the stack frame of the executing program. For example, Drexel University's GDBProfiler [5] uses the GNU debugger interface to profile C programs.

2.2 Program feature analysis

The objective of feature analysis is to correlate program features with implementation artifacts found in the source code. In this context, a feature usually refers to a usage scenario of the program [6] or a test case [25].

Eisenbarth *et al.* use dynamic and static analysis to associate features to components [6]. Dynamic profiling is used to identify the subprograms that are exercised when a specific feature is executed. Concept analysis is then used to analyze the relationships between features and subprograms. Concept analysis results are combined with static analysis to refine the classification of subprograms as well as to establish the dependencies between subprograms with respect to a given set of features.

Wong *et al.* used program execution slices to identity the portions of the code that implement a given feature or a set of features [25].

Wilde and Scully developed a technique for locating program features by analyzing the execution traces of test cases [24]. The technique uses two sets of test cases, one set that executes the feature, and a second set that does not. A comparison of the execution traces of each set is used to identify the subprograms that implement a given feature.

Chen and Rajlich [4] have developed a technique for identifying program features from an abstract system dependencies graph (ASDG). The ASDG is derived from the source code, and it describes source code entities (*e.g.*, procedures, types, variables) and the relationships between them. The identification of features is performed manually by traversing the graph.

Our work, contributes to the state-of-the-art by creating the feature-interaction view, which identifies the dependencies between features and the creation of tools that automate the creation of feature-interaction views.

2.3 Software comprehension environment

Next is a brief introduction to the software comprehension environment used to create the set of software views described in this paper. Further details about the environment are described elsewhere [19, 20]. Figure 1 illustrates the architecture of the software comprehension environment. The main subsystems are: data gathering, repository, and analysis/visualization subsystems.

The **data gathering subsystem** defines the interfaces and data formats of the data collection tools (*i.e.,* static analyzers and dynamic profilers). In our experiments, we used

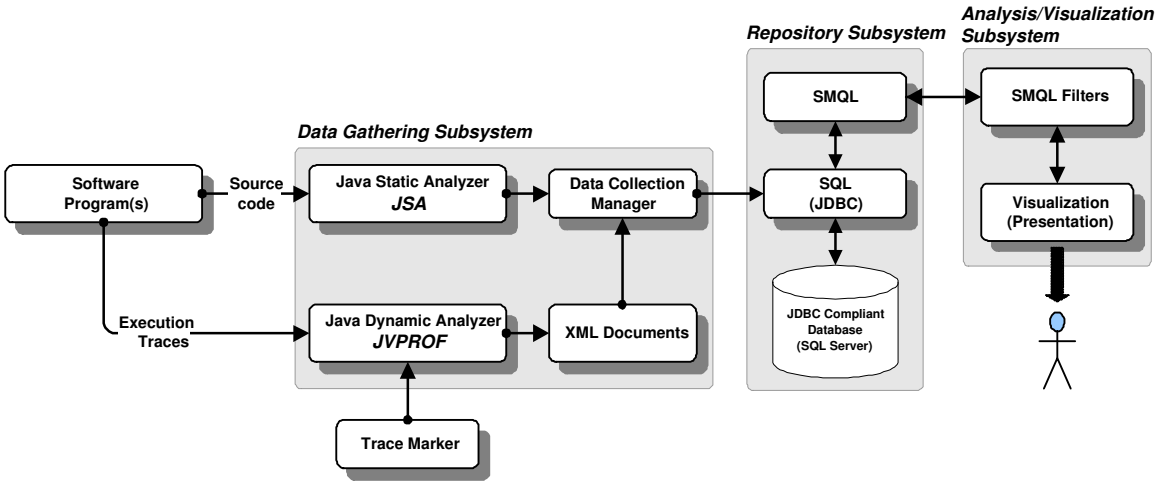


Figure 1. Architecture of the software comprehension environment

both static and dynamic analyzers. The static analyzer is JSA, which is a Java extractor implemented using the BCEL [7] package. The dynamic analyzer is JVPROF, which is a profiler implemented using the JVMDI and JVMPI interfaces [21, 22]. We have enhanced JVPROF to allow users to mark runtime execution traces by providing a wizard that enables users to mark the beginning and the end of a trace. The data collection manager is the common facility that all data extractors use to store the collected data into the repository. In the case of the JSA, it connects directly to the data collection manager, while in the case of JVPROF, it stores data as an XML document, which is later exported to the repository.

The **repository subsystem** defines the data and metadata models, as well as the data manipulation and query language. The data repository stores the program entities, relationships, and runtime events that are gathered from the subject system. The repository is manipulated using standard SQL and is queried using either SQL or our own SMQL query langauge [19]. The repository uses any JDBC-compliant database.

SMQL (Software Modelling Query Language) simplifies the data retrieval and analysis of program data to create software views. Even though the repository can be queried using SQL, designing queries for comprehending software systems using SQL is cumbersome. Many of the queries that are of interest to an engineer, for example queries that involve the transitive closure of a relation, are not supported directly by SQL. SMQL is a set-based language that facilitates the definition of queries about entities, relations, and events by translating the SMQL code into SQL query statements. SMQL provides a built-in `closure` function as well as binary operators such as union, intersection, and join. SMQL is similar to `grok` [9] for manipulating binary relational algebra expressions. Unlike `grok`, SMQL can be extended to support additional operations that are implemented in Java.

The **analysis and visualization subsystem** is responsible for the creation and visualization of software views. Analysis is performed using SMQL analyzers, which are SMQL extensions that are implemented in Java. Software views are visualized either as a tree or as a graph. The graph visualization is performed using JGraph [2], graph layout is performed by dot [8], and the clustering feature uses Bunch [12]. Software views are implemented as SMQL extensions written in Java.

3 Dynamic views

This section describes a hierarchy of views that captures the dynamic behavior of programs from execution traces. There are different levels of abstraction in the hierarchy. The lowest level represents the runtime events that are generated during the execution of the marked-traces. The second level of abstraction includes runtime objects and the relationships between them. This level is the basis for higher-level views. The third level includes classes and the runtime relationships between them. The highest level of the hierarchy represents program features and feature-interactions. In this context, a program feature refers to an externally visible functionality of the program, and is identified as a marked-trace. The set of the views are described below and are outlined in Figure 2:

1. **Object-interaction view**. This view is constructed from the execution traces of the program. It serves as the basis for higher level views such as the class-interaction and feature-interaction views.

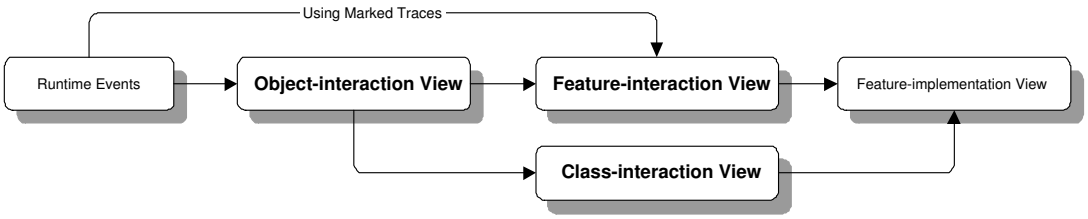


Figure 2. Hierarchy of Views

2. **Class-interaction view**. This view is an abstraction of the object-interaction view, where sets of objects are represented by their corresponding classes. However, the interactions between the classes in this view represent the dynamic relationships derived from the object-interaction view.

3. **Feature-interaction view**. This view illustrates the interactions between program features. Features are defined by users in terms of marked-traces. A marked-trace is established manually during the execution of the program by specifying the start and the end of the trace. Feature interactions are derived from the object-interaction view automatically.

4. **Feature implementation view**. This view is a mapping between program features and the classes that implement these features.

The creation of the aforementioned views requires identifying various sets of runtime objects (objects are identified by their unique runtime references). For features, these sets are defined as follows:

$$\begin{aligned} \mathbf{L}\text{ocal}(\mathcal{F}_k) &= \text{Set of objects created and used by feature } \mathcal{F}_k \text{ only} \\ \mathbf{I}\text{mport}(\mathcal{F}_k) &= \text{Set of objects used by feature } \mathcal{F}_k \text{ and created by feature } \mathcal{F}_j, j \neq k \\ \mathbf{E}\text{xport}(\mathcal{F}_k) &= \text{Set of objects created by feature } \mathcal{F}_k \text{ and used by feature } \mathcal{F}_j, j \neq k \end{aligned}$$

These sets allow us to define the following two relations, which are used to describe the views in the next subsections:

$$\begin{aligned} depends(\mathcal{F}_k, \mathcal{F}_j) &= \mathbf{I}(\mathcal{F}_k) \cap \mathbf{E}(\mathcal{F}_j)\ ,\ j \neq k \\ shares(\mathcal{F}_k, \mathcal{F}_j) &= \mathbf{I}(\mathcal{F}_k) \cap \mathbf{I}(\mathcal{F}_j)\ ,\ j \neq k \end{aligned}$$

In the $depends(\mathcal{F}_k, \mathcal{F}_j)$ relation, feature $\mathcal{F}_k$ used objects that were created during the execution of feature $\mathcal{F}_j$. While in the $shares(\mathcal{F}_k, \mathcal{F}_j)$ relation, both $\mathcal{F}_k$ and $\mathcal{F}_j$ features used the same set of objects that were not created by $\mathcal{F}_k$ nor $\mathcal{F}_j$ features.

3.1 Object-interaction view

This view shows the creation and method invocation relationships between objects. Field accesses of primitive data types (*e.g.*, `int`, `char`) are not included in the analysis. This view provides information about the collaboration between objects based on the *depends* and *shares* relations described above. This view is very detailed and not suitable for program comprehension. However, it serves as the base view for deriving higher level views such as those described in the following subsections. The object-interaction view can be summarized by clustering the object-interaction graph. An example of this view is shown in Figure 10.

The object-interaction view highlights threads that execute concurrently and share objects. The view isolates objects that can be examined for potential concurrency issues such as race conditions or deadlock. As shown in Figure 3, the edges in the view encode information about the shared objects, and the user can view these objects by double-clicking on specific edges.

The object-interaction view also highlights the communication endpoints between the modules of a distributed systems. Further details have been described in our previously published work [19].

3.2 Class-interaction view

The class-interaction view is an abstract view of the object-interaction view, where objects are grouped by their class type. Figure 4 shows the class-interaction view of the KWIC (keyword in context) algorithm. There are two types of relationships in the view: *creates* (represented as an arrow with dotted-tail) and *uses* (represented as a normal arrow) relationships derived from the *depends* and *shares* relations of object-interactions view. The *creates* relationship between two classes C_1 and C_2 implies that an instance of C_1 created and possibly used an object instance of type C_2. The *uses* relationship between C_1 and C_2 implies that an instance of C_1 used (*e.g.,* invoked a method) an instance of C_2 that was not created by C_1. This view is essentially a hybrid static/dynamic view as the entities are static (*e.g.,* classes) and the relationships are dynamic (*e.g.,* invokes and

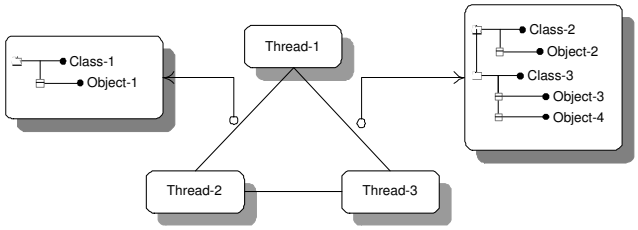


Figure 3. Inter-thread Interaction

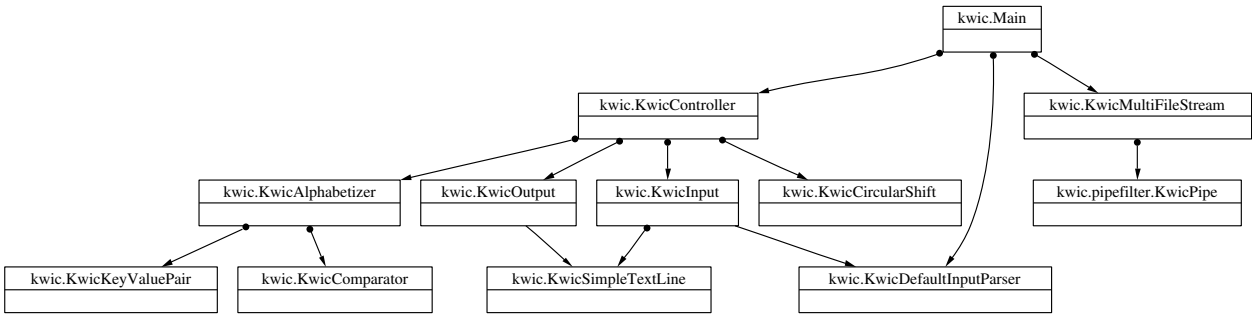


Figure 4. Class-interaction view of KWIC program

creates). The content of this view is similar to the inter-class call matrix and histogram of instances grid developed by De Pauw *et al.* [17]. This view is far less detailed than the object-interaction view and can be used by developers for program comprehension.

3.3 Feature-interaction view

The feature-interaction view captures relationships between features. This view is derived from the object-interaction view by grouping objects based on where a given object is created or used. Features in our analysis are identified in terms of marked-traces, which the user specifies during the execution of a program using a utility to mark the start or the end of each execution trace.

Figure 5 shows a features-to-classes grid. The filled rectangles in the diagram are objects that are created during the execution of the respective feature; unfilled rectangles are objects that are used during the execution of the feature but created by another feature. The content of Figure 5 is described as follows:

Feature	$\mathcal{F}_1$	uses objects	$\{O_1, O_2, O_3\}$,
Feature	$\mathcal{F}_2$	uses objects	$\{O_2, O_4, O_5\}$,
Feature	$\mathcal{F}_3$	uses objects	$\{O_1, O_4, O_6\}$,
Feature	$\mathcal{F}_4$	uses objects	$\{O_4, O_7\}$,
Class	C_1	instantiates objects	$\{O_1, O_2\}$,
Class	C_2	instantiates objects	$\{O_3, O_4, O_5, O_6\}$,
Class	C_3	instantiates object	$\{O_7\}$

O_1 is created during the execution of $\mathcal{F}_1$ and is used by $\mathcal{F}_3$. O_2 is created during the execution of $\mathcal{F}_1$ and is used by $\mathcal{F}_1$. O_4 is created during the execution of $\mathcal{F}_2$ and is used by $\mathcal{F}_3$ and $\mathcal{F}_4$. The feature-interaction diagram, in Figure 6(a), is constructed as follows:

1. Identify the **I**mport($\mathcal{F}$) and **E**xport($\mathcal{F}$) sets for each feature:

$$\begin{aligned}
\mathbf{E}(\mathcal{F}_1) &= \{O_1, O_2\} \\
\mathbf{I}(\mathcal{F}_1) &= \phi \\
\mathbf{E}(\mathcal{F}_2) &= \{O_4\} \\
\mathbf{I}(\mathcal{F}_2) &= \{O_2\} \\
\mathbf{E}(\mathcal{F}_3) &= \phi \\
\mathbf{I}(\mathcal{F}_3) &= \{O_1, O_4\} \\
\mathbf{E}(\mathcal{F}_4) &= \phi \\
\mathbf{I}(\mathcal{F}_4) &= \{O_4\}
\end{aligned}$$

2. Identify the *depends* and *shares* relations between each pair of features using the **E** and **I** sets (empty sets are excluding):

$$\begin{aligned}
depends(\mathcal{F}_2, \mathcal{F}_1) &= \mathbf{E}(\mathcal{F}_1) \cap \mathbf{I}(\mathcal{F}_2) = \{O_2\} \\
depends(\mathcal{F}_3, \mathcal{F}_1) &= \mathbf{E}(\mathcal{F}_1) \cap \mathbf{I}(\mathcal{F}_3) = \{O_1\} \\
depends(\mathcal{F}_3, \mathcal{F}_2) &= \mathbf{E}(\mathcal{F}_2) \cap \mathbf{I}(\mathcal{F}_3) = \{O_4\} \\
depends(\mathcal{F}_4, \mathcal{F}_2) &= \mathbf{E}(\mathcal{F}_2) \cap \mathbf{I}(\mathcal{F}_4) = \{O_4\} \\
shares(\mathcal{F}_3, \mathcal{F}_4) &= \mathbf{I}(\mathcal{F}_3) \cap \mathbf{I}(\mathcal{F}_4) = \{O_4\}
\end{aligned}$$

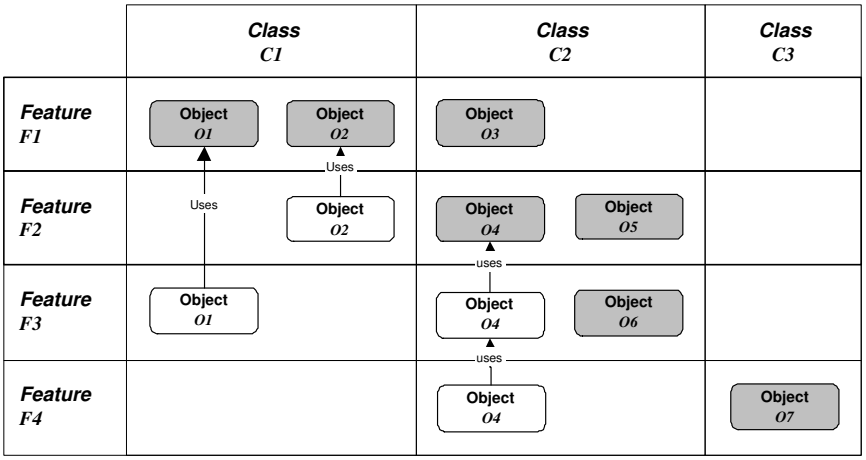
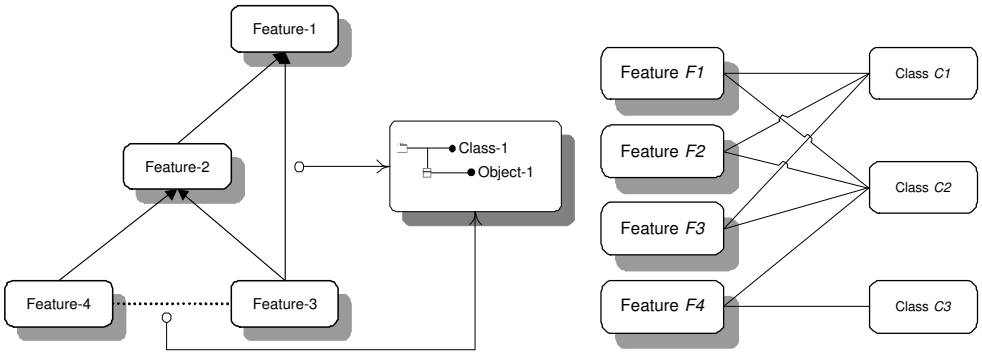


Figure 5. Feature-interaction grid



(a) Feature-interaction graph

(b) Feature-implementation graph

Figure 6. Feature-interaction and implementation views

3. Draw an edge between $\mathcal{F}_j$ and $\mathcal{F}_k$ if the $depends(\mathcal{F}_j, \mathcal{F}_k)$ set is not empty. The *depends* edge is represented as a solid-arrow, where the arrow head points to the feature that created the object(s) participating in the *depends* relationship.

4. Draw an edge between $\mathcal{F}_j$ and $\mathcal{F}_k$ if the $shares(\mathcal{F}_j, \mathcal{F}_k)$ set is not empty. The *shares* edge is represented as a dotted-line.

The data combined from the class-interaction and the feature-interaction views is used to produce the **feature-implementation view** (Figure 6(b)), which identifies the classes that are used to implement a given feature. This view provides a simple mapping between features and implementation-level classes. This view is not unique to our work, several research efforts [4, 6, 24, 25] have identified this kind of mapping. The view is represented as a graph, where the nodes represent features and classes, and edges represent instantiation relations between features and classes.

4 Case study

Table 1 outlines the systems that were analyzed as part of our case study. The counts in the table exclude common packages such as `java.*` and `javax.*`. The two case studies are Jext [1], a programmer's text editor, and Jetty [15], a web server. Jext is a feature-rich application, whose externally visible features corresponds to a set of marked-traces. Jetty is a software system whose features are not clearly visible to the user. The marked-traces for Jetty capture how a developer interacts with a client application (web browser) to locate portions of the server code.

```
// Events from JEXT Editor

EventSet JextEvents
    {
    caption = "Event (JEXT) " ;
    type = {"method-entry",
            "method-exit",
            "endpoint",
            "thread-start",
            "thread-end",
            "module-load",
            "process-start",
            "object-create",
            "process-end"} ;
    include (project) = {"jext"} ;
    include (trace) = {"*"} ;  // all traces
    exclude (trace) = {"startup", "shutdown"};
    exclude (name) = {"finalize"} ;
    exclude (container) = {"java.*",
            "javax.*", "gnu.*"} ;
    }
```

```
// Get java.net.* events (Network events)
EventSet NetEvents
    {
    caption = "Net Event (JEXT) " ;
    type = {"method-entry",
            "method-exit",
            "endpoint",
           "thread-start",
            "thread-end",
            "module-load",
            "object-create"
            } ;
    include (project) = {"jext"} ;
    include (trace) = {"*"} ;
    exclude (trace) = {"startup", "shutdown"};
    exclude (name) = {"finalize"} ;
    include (container) = {"java.net.*"} ;
    }

// Compute the union of JextEvents & NetEvents
AllEvents = JextEvents + NetEvents ;

// Apply the event-analyzer
results = AnalyzeEvents (AllEvents) ;
```

Figure 7. Jext: SMQL code

Project	Classes	Methods	Objects	Events
jext	538	2,893	55,624	334,852
jetty	189	2,088	940	36,367

Table 1. Systems analyzed

4.1 The Jext text editor

Jext is a programmer's text editor, written in Java, that supports many languages (*e.g.,* C/C++, Java, XSLT, TEX). In our study, we marked traces to identify a subset of the Jext features. The scenarios used to mark the selected traces are: (a) opening three different documents (a Java source file, an HTML file from the web, and a Java source file from bookmarks), (b) performing edit activities (copy, cut, paste) on the Java documents, (c) searching for a string in the Java documents, (d) searching and replacing, (e) adding bookmarks, and (f) e-mailing the opened Java documents. Table 2 shows the number of objects used, the number of objects created, and the number of events generated during the execution for each of the marked traces.

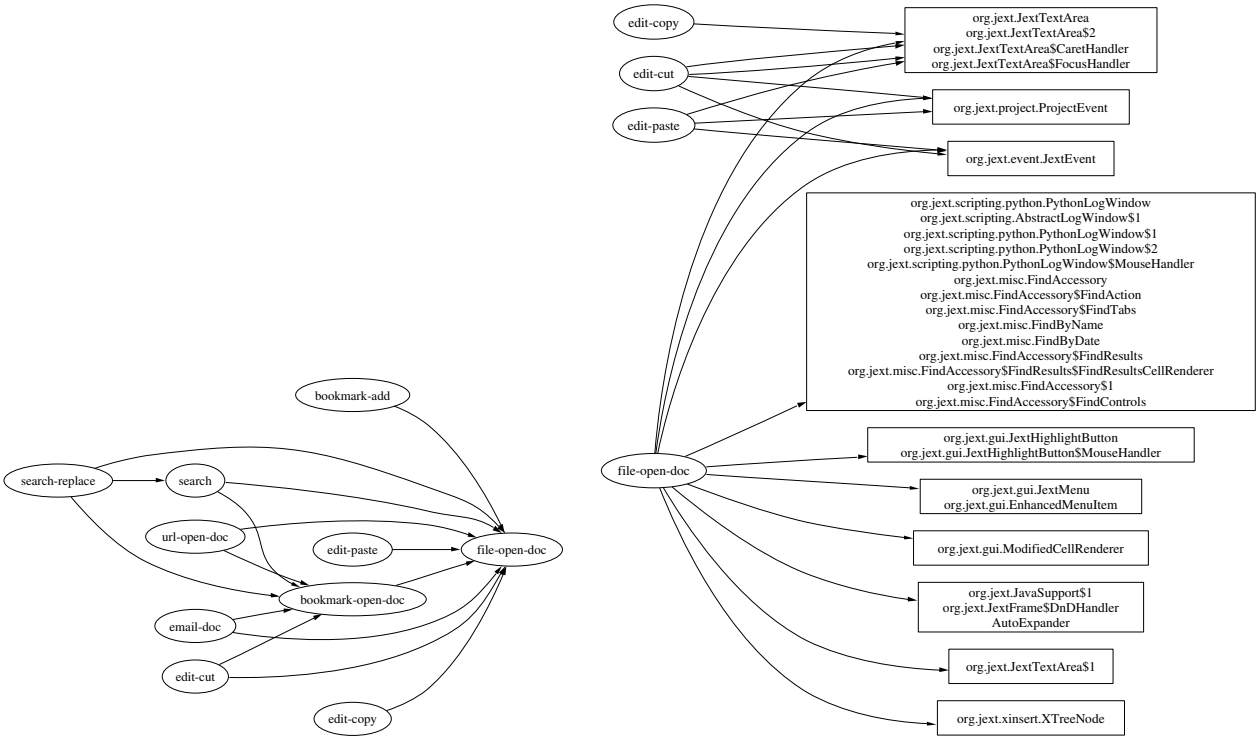
The SMQL code used to derive the software views from the runtime events is listed in Figure 7. First, we define two sets of events (**EventSet**): `JextEvents` and `NetEvents`. In the `JextEvents` set, we include events that were created during the execution of the marked-traces, but we exclude events created from the standard packages such as `java.*` and `javax.*`. We also exclude events that were created during startup and shutdown of the application.

Feature	Obj. used	Obj. created	Events
bookmark-add	68	16	1,866
bookmark-open-doc	4,321	2421	31,872
edit-copy	33	4	4,719
edit-cut	50	6	1,973
edit-paste	132	104	5,545
email-doc	2,691	43	10,889
file-open-doc	12,470	9,552	113,452
search	47	16	9,013
search-replace	3,171	89	121,433
url-open-doc	2,457	2,414	28,254

Table 2. Jext: Objects and events per feature

The second event set is `NetEvents`, here we are only interested in `java.net.*` events to identify data pertaining to remote method invocations of a distributed application. The `finalize()` method is excluded to prevent the mis-identification of interactions, since this method is invoked by the garbage collector and the JVM does not guarantee which thread will invoke this method. The union of the two event sets is computed using the "+" operator.

Finally, we invoke the event analyzer (**EventAnalyzer**) to create the dynamic views from the input event set. The returned value `results` is a set of sets, where each element (set) represents a **RelationSet** that defines a view or a part of a view.



(a) Feature-interaction view

(b) Feature-implementation view

Figure 8. Jext: Feature-interaction and implementation views

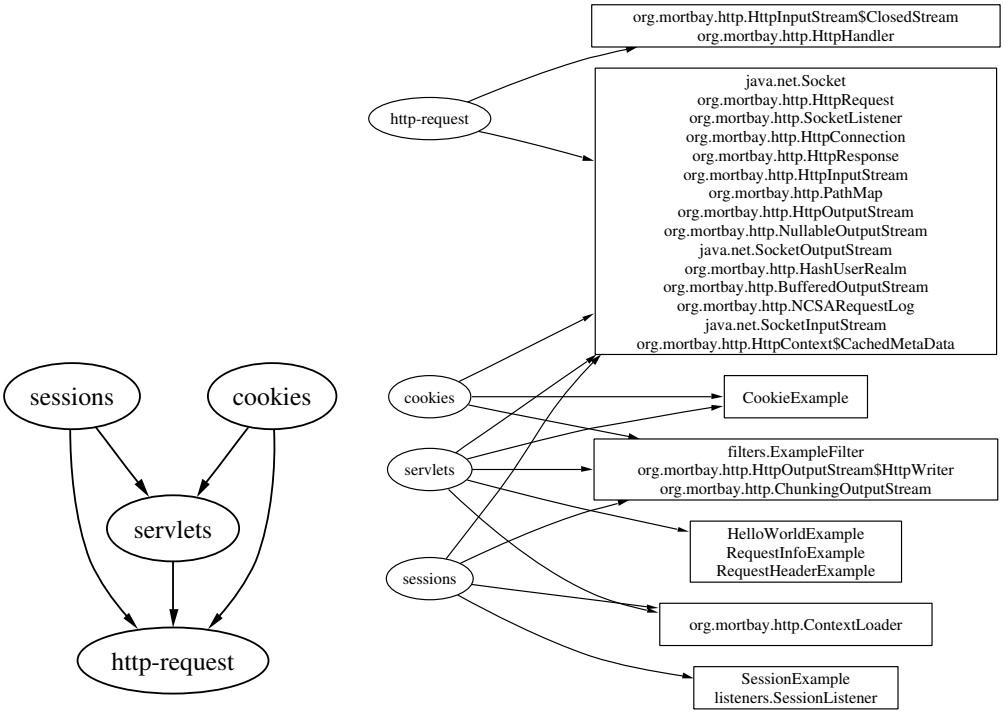
Figure 8 shows the feature-interaction view and feature-implementation view for Jext. In Figure 8(a), arrows represent *interaction* relationships between features. The dominant features are `file-open-doc` and `bookmark-open-doc`, where a new document (`JextTextArea`) is created and initialized from the source files. It is expected that other features such as editing, searching and e-mailing will reference the open documents of a text editor. We also observe that the `search-replace` feature uses objects that were created by the `search` feature. This is also expected for a search/replace feature, since the `search` function is needed before the `replace` function is applied. The view provides a developer with useful information to reason about the impact of changes without the need to read the source code. For example modifying the replace part of `search/replace` feature may not have an impact on the `search` feature, while modifying `search` will have direct impact on the `search/replace` feature. Similarly, modifying the `file-open-doc` will have a direct impact on all the other features.

Figure 8(b) shows a partial mapping of features to classes. The arrows point to the classes that were exercised during the execution of a given feature.

4.2 The Jetty web server

The SMQL code for the Jetty [15] web server is similar to that for the Jext example. In our case study, we marked four traces. The scenarios used were designed to exercise features such as: servlet support, session management and cookies management. The server traces were marked while interacting with servlet examples (part of Jetty distribution package) via a web browser. The feature-interaction view is shown in Figure 9(a), and the feature implementation is shown in 9(b). The dominant feature is `http-request`, which is responsible for accepting and processing http requests. Note that the `sessions` and `cookies` features are independent of each other, but both depend on the `servlets` features.

Figure 10 shows a clustered object-interaction view. In the diagram, the cluster labels are the names of the dominant component of the clusters. To keep the diagram simple, we only expanded one cluster, which is *CL-5*. The other clusters are represented as filled octagons. This view is con-



(a) Feature-interaction view

(b) Feature-implementation view

Figure 9. Jetty: Feature-interaction and implementation views

structed by clustering the object-interaction using Bunch [12], then the objects within each cluster are grouped based on their type (or class) to simplify the diagram (as in *CL-5* cluster). Edges represent either *uses* or *creates* relationships, and nodes represent classes (rectangles) and clusters (octagons).

Using our tools, a user can navigate between different views as well as expand nodes and edges in the diagram to view the encoded information in the nodes and edges. For example, a user can double-click on a feature to view the object-interaction and class-interaction views of a given feature, or a user can double-click on an edge to reveal the objects and classes involved in a given relationship.

5 Conclusions and future work

In this paper we describe a hierarchy of dynamic views, which includes feature-interaction, feature-implementation, class-interaction, and object-interaction views. We describe the environment used to construct these views. Through two case studies, we demonstrate the ability of the environment to collect dynamic data, analyze the data, and visualize it as a set of views. We believe that the software views and the tools described are helpful for maintenance tasks that require a detailed understanding of specific parts of a software system.

Our work contributes to the state-of-the-art by creating a set of views with various levels of abstraction and the design of tools to automate the creation of these views.

Future efforts will focus on two areas. First, we are planning to conduct a more extensive study to evaluate the practical effectiveness of the views and the tools. Second, we would like to improve the performance of the profiler and view generator.

References

[1] *Jext: Source code editor*. http://www.jext.org/.

[2] G. Alder. *Design and Implementation of the JGraph Swing Component*, 2003. http://www.jgraph.com.

[3] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analysis. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSLA93)*, Washington, USA, September 1993.

[4] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International*

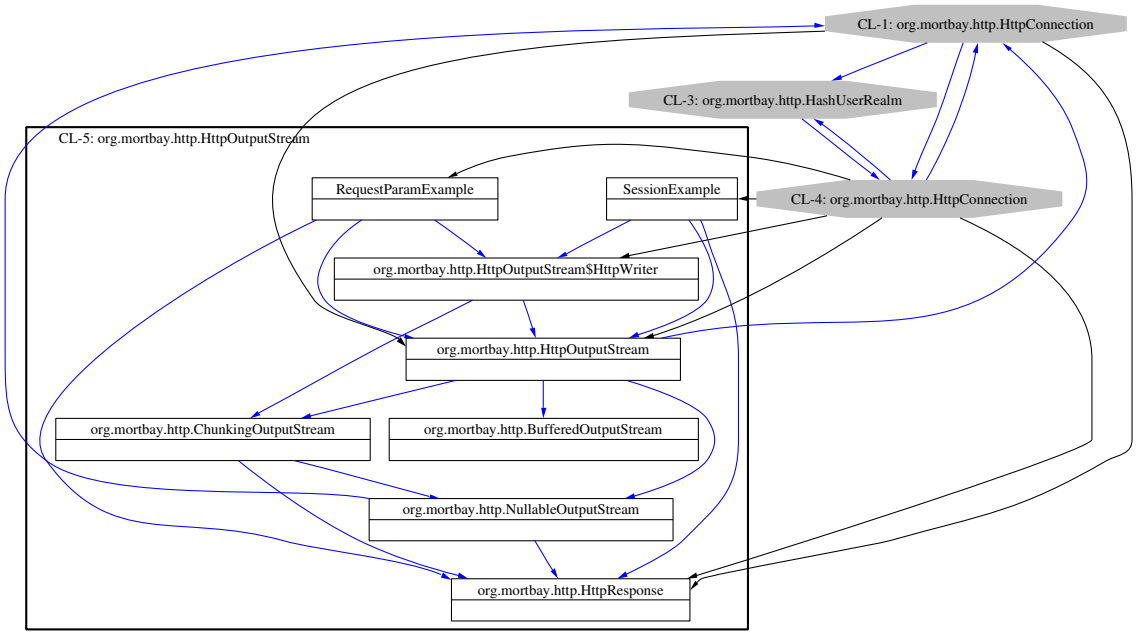


Figure 10. Jetty: Clustered object-interaction view

Workshop on Program Comprehension, Limerick, Ireland, June 2000.

[5] C. Dahn and J. Penrose. GDBProfiler for GNU C/C++. http://serg.mcs.drexel.edu/gdbprofiler.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *International Conference on Software Maintenance*, Florence, Italy, November 2001. IEEE.

[7] Free Software Foundation. *Byte Code Engineering Library (BCEL)*. http://jakarta.apache.org/bcel.

[8] E. Gansner, E. Koutsofios, and S. C. North. *Drawing graphs with dot*. AT&T Bell Laboratories, Murray Hill, NJ, February 2002.

[9] R. C. Holt. Binary relational algebra applied to software architecture. Technical Report CSRI-345, University of Toronto, March 1996.

[10] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. SCED: A tool for dynamic modelling of object systems. Technical Report A-1996-4, Department of Computer and Information Sciences, University of Tampere, Finland, 1996.

[11] H. Lee and B. Zorn. *BIT: Bytecode Instrumenting Tool*. http://www.cs.colorado.edu/ hanlee/BIT.

[12] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *International Conference on Software Maintenance*, pages 50–59. IEEE Computer Society Press, 1999.

[13] Microsoft Corporation. *COM+ SDK Documentation: COM+ Instrumentation*, 1999.

[14] Microsoft Corporation. *.NET Framework: Runtime profiling*, 2001.

[15] Mort Bay Consulting. *Jetty Web Server and Servlet Container*. http://jetty.mortbay.org.

[16] M. R. Olsem. Reengineering technology report. Technical Report Volume 1, Software Technology Support Center (STSC), October 1995.

[17] W. D. Pauw, D. Kimelman, and J. M. Vlissides. Modelling object-oriented program execution. In *8th European Conference on Object-Oriented Programming (ECOOP)*, Bologna, Italy, July 1994.

[18] T. M. Pigoski. *Practical Software Maintenance: Best Practices Managing Your Software Investment*. John Wiley & Sons, 1997.

[19] M. Salah and S. Mancoridis. Toward an environment for comprehending distributed systems. In *Proceedings of Tenth Working Conference on Reverse Engineering (WCRE)*, Victoria, Canada, November 2003. IEEE.

[20] T. Souder, S. Mancoridis, and M. Salah. Form: A framework for creating views of program executions. In *International Conference on Software Maintenance*, Florence, Italy, November 2001.

[21] Sun Microsystems, Inc. *Java Platform Debugger Architecture*, 1999.

[22] Sun Microsystems, Inc. *Java Virtual Machine Profiler Interface (JVMPI)*, 1999.

[23] B. Swanson and C. M. Beath. Departmentalization in software development and maintenance. *Communication of the ACM*, 33(6), June 1990.

[24] N. Wilde and M. Scully. Software reconnaissance: Mapping program features to code. *journal of Software Maintenance: Research and Practice*, 7(1), January 1995.

[25] E. Wong, S. Gokhale, J. Horgan, and K. Trivedi. Locating program features using execution slices. In *Proceedings of Application Specific Software Engineering and Technology (ASSET 99)*, Dallas, TX, March 1999.