# Playing Detective: Reconstructing Software Architecture from Available Evidence

**Rick Kazman, S. Jeromy Carrière**
**Software Engineering Institute, Carnegie Mellon University**
**Pittsburgh, PA 15213**
**{kazman, sjc}@sei.cmu.edu**

**Abstract:** It is important to be able to reason architecturally about a software system. However, architectural documentation frequently does not exist and even when it does exist, it is often out of sync with the implemented system. In addition, it is rare that software development begins with a clean slate; systems are almost always constrained by existing legacy code. As a consquence, we need to be able to extract information from existing system implementations and reason architecturally about this information. This paper presents *Dali*, an open, lightweight workbench that aids an analyst in extracting, manipulating, and interpreting architectural information. By assisting in the reconstruction of architectures from extracted information, Dali helps an analyst redocument architectures and discover the relationship between as-implemented and as-designed architectures.

## 1 Software Architecture as Shared Hallucination

The formal study of software architecture has been a significant addition to the software engineering repertoire in the 1990s. It has promised much to designers and developers: help with the high level design of complex systems; early analysis of high level designs, particularly with respect to their satisfaction of quality attributes such as modifiability, security, and performance; higher level reuse such as the reuse of designs; and enhanced stakeholder communication ([1], [6]). These these benefits seem enticing. However, much of the promise of software architecture has as yet gone unfulfilled. Why is this?

Some of the problems simply stem from the fact that architectures are seldom properly documented:

- Many systems have no documented architecture at all (*all* systems have an architecture, but frequently it is not explicitly known or recorded by the developers, but rather evolves in an ad hoc fashion).
- Architectures are represented in such a way that the relationship between the architectural representation and the actual system, particularly its source code, is unclear.
- In systems that *do* have properly documented architectures, the architectural representations are frequently out of sync with the actual system, due to maintenance of the system without a similar effort to maintain the architectural representation.

We see these problems on a regular basis at the Software Engineering Institute when we do architectural evaluations. There is little completely new development. Development is typically constrained by compatibility with, or use of, legacy systems. And it is rare that such systems have an accurately documented architecture. Because of these issues, we have a serious problem in assessing architectural conformance, and if we cannot assess architectural conformance, what good is having an architecture? How do we know that what was designed is what was built? If we can not confidently establish this relationship, much of the value of having an architecture is lost.

In addition, when a system enters the maintenance portion of its life-cycle, it may sustain modifications that alter its architecture. Hence, a second problem arises: how do we know that maintenance operations are not eroding the architecture, breaking down abstractions, bridging layers, compromising information hiding, and so forth?

All of these are manifestations of two underlying causes. The first is that a system does not have "an architecture". It has many: its run-time relationships, its data flows, its control flows, its code structure, and so on. The second, more serious, cause is that the architecture that is represented in a system's documentation may not coincide with *any* of these views. "The architecture" is frequently some abstracted run-time view of the system. For example, even though a system is described as "layered" the location and boundaries of the layers are not obvious from an examination of *any* of the architectural views.

Quite simply, there is no accepted way of enforcing a "layer" in a system's implementation—there is no explicit "layer" construct in any modern programming language. Attempts to enforce layering are typically made through means such as naming conventions (e.g. all functions defined by the X windows library, Xlib, begin with "X"), code ownership (the graph layout layer is owned by a single development group, and only they can change it), and design conventions (the graph layout layer cannot directly call the window system, but must instead call a virtual toolkit layer). For example, the "operating system" layer seen in many architectural diagrams is only a layer by virtue of code ownership: few designers or developers have access to its source. So, designers must treat it as a sealed layer. This is the best case. In many cases we have access to *all* of the source code in our architecture. The architecture as a whole does not exist in any artifact that we actually implement.

So, is software architecture a mass hallucination that we, as developers, gladly and glibly ascribe to? If not, how do we know what the architecture of a system is? How do we validate this? How do we measure the conformance of an "as-implemented" architecture to its "as-designed" architecture? If we can't answer these questions then our use of software architecture amounts to little more than a vague vision and blind faith in the abilities of the original designers and their successors.

In this paper we present a prototype system, called *Dali*, for helping a user reason about an implemented architecture. Dali is an interactive system that aids the user in *interpreting* architectural information that has been automatically extracted. The system does not attempt to do it all, which is to say that it does not attempt to automatically "find" the architecture for the user. This approach

has been tried before and has failed before. Rather, Dali *supports the user* in defining architectural patterns and in matching those patterns to extracted information.

There are three techniques used in reconstructing an architecture using Dali:

1. Architectural extraction, that captures the as-implemented architecture from source artifacts such as code and makefiles. We can augment this static information with output from analysis tools that capture a system's dynamic behavior (such as profilers or test coverage tools).
2. User defined architectural patterns, that collectively link the as-implemented architecture to the as-designed architecture. The as-designed architecture consists of the kinds of abstractions used in architectural representations (subsystems, high level components, repositories, layers, conceptually-related functionality, and so forth). The architecture patterns explicitly link these abstractions to the information extracted in step 1.
3. Visualization of the resulting architecture—the extracted information, as organized by the patterns—for validation by the user.

Each of these techniques on its own is insufficient to address the problem of architectural extraction: the architectural extraction and visualization techniques used here are not in themselves new. The solution to "finding" an architecture from extracted artifacts derives from the synergy of the parts and from our model of user interaction through pattern matching and interpretation. This is the main contribution of our work.

In this paper we describe how Dali is used, and the kinds of insights that one can get from using it. We exemplify its use through assessments of two systems: VANISH [11], a system for prototyping visualizations, and UCMEdit, a system for creating and editing Buhr-style use case maps [4].

In summary, our goals with Dali are to:

- support architectural analysis, which implies the need to
- redocument architectures.

## 2 Hello Dali: An Extraction/Analysis Workbench

Because there is a great deal of variance in languages, architectural styles, implementation conventions and so forth, we believe that no single collection of tools will suffice for all architectural extraction and analysis. Thus, in creating support for extraction and analysis we have created an *open*, *lightweight* "workbench": a environment that provides an infrastructure for opportunistic integration of a wide variety of tools and techniques. New elements must be easy to integrate into

the workbench (openness), and such integration should not unnecessarily impact other elements of the workbench (dependencies are lightweight).

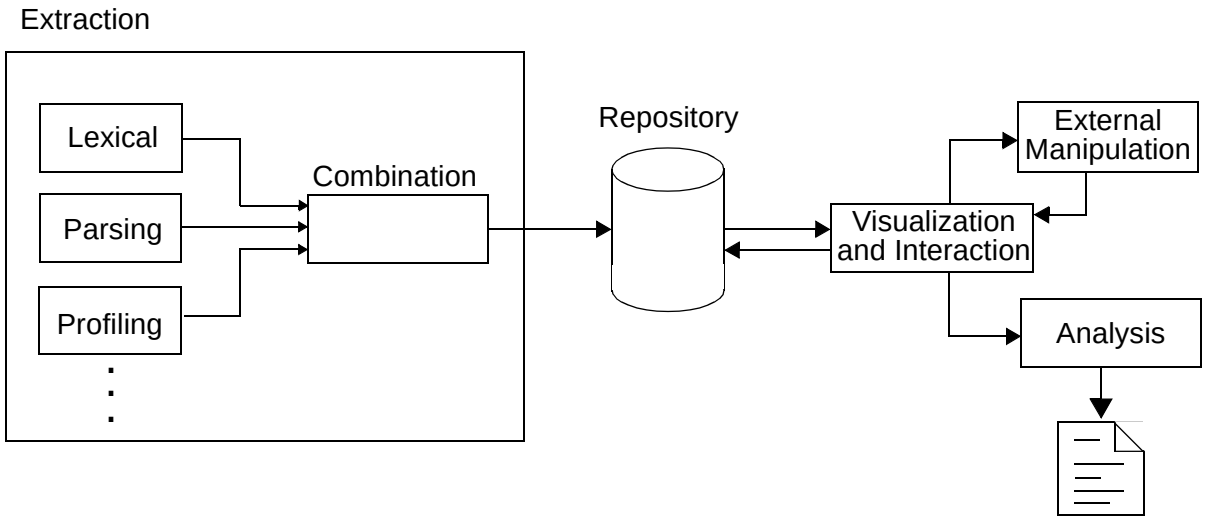


**Figure 1: The Dali workbench**

The following sections describe the Dali workbench, as illustrated in Figure 1. The workbench is first discussed in general terms, leaving elements of the workbench that may vary between applications (such as particular extraction techniques, analysis tools, or manipulation techniques) unspecified. Section 2.5 then discusses a particular population of the workbench.

## 2.1 Concrete Model Extraction: Gathering Clues

A necessary first step in supporting the analysis and evaluation of a software architecture is the extraction of a *concrete model*, a representation of the implemented system. Such a representation comprises a collection of *elements* (e.g. functions, files, variables, objects, etc.), a collection of *relations* between the elements (e.g. “function calls function”, “file contains function”), and a set of attributes of these elements and relations (e.g. “function calls function *N* times”, “object A has type B”). A concrete model may reflect several views of a system, such as: its static structure, its dynamic (run-time) nature or its build-time structure.

There are many techniques and tools for static source model extraction, largely divisible into two classes: those based on parsing and those based on lexical techniques. Lexical techniques are usually more versatile and lightweight than parse-based techniques, but typically achieve lower accuracy. Regardless, it is important to appreciate that *no one tool* will successfully extract a complete source model: first, tools are designed to extract particular source elements, rather than comprehensive models and second, tools never accurately extract source elements. Murphy *et al* demonstrate this point convincingly with a comparison of static call graph extractors in [15].

It might appear to be acceptable that extractors are imperfect when considering systems from the architectural perspective: why should one missed function call disturb the high level model? However, this is a dangerous assumption as it will not be a single function call that is missed, but more

likely a whole class of related elements or relations. This deficiency will likely affect the architectural model. So, what are we to do? We propose that *composition* of multiple extraction techniques will alleviate these problems by providing a concrete model of higher accuracy than any individual technique.

In the simplest case, extraction techniques will be directed toward different views—disjoint sets of elements and relations (e.g. one technique for extraction of function calls and another for extraction of variable access). Composition is then simply a matter of constructing the union of the concrete models; however, this does not address the potential deficiencies of any individual technique. Composition of techniques that are not intended to generate disjoint models requires several issues to be addressed. First and foremost is that of *conflicts* between models: the situation when one extractor identifies a particular element or relation, and another extractor does not. The simplest, though potentially incorrect, solution is to generate a union in this case as well, resulting in a concrete model that incorporates false positives from each contributing model. An alternate solution—on analogy with software fault tolerance—is to combine multiple overlapping models with "voting": an element or relation is included in the composite model if it appears in a majority of the contributing models, or weighted voting, where the votes of some models weigh heavier than others with respect to specific extracted artifacts.

This model of composition of extraction techniques contributes to the open, lightweight nature of Dali. The method for integration of a new extraction technique is typically trivial and at worst requires an analysis of the characteristics of the new extractor. Because extractors are ignorant of each other, integration of a new technique does not imply that the existing extractors need to reanalyse the source corpus. For these reasons, a single all-encompassing extractor is not necessary; instead, individual lightweight extractors are incorporated into the Dali workbench opportunistically.

## 2.2 A Repository Hybrid

Once a concrete model is extracted it must be stored. We considered two options for this storage: use of a database that manages all access to the model and use of a interchange format that decentralizes model access. An interchange format provides more flexibility in a multi-tool environment, but involves significant up-front effort for definition. As the intention of the workbench approach is to apply tools opportunistically, the use of a centralized database complicates model maintenance: each tool must have an interface to the database and be responsible for updating the database with any modifications that it makes. The interchange format, then, appears to be the solution. On the other hand, a database provides handy functionality with respect to querying, multi-user support, and history mechanisms.

With Dali, we have adopted a hybrid approach in which we use an SQL database for primary model storage, but application-specific file formats for interchange between tools. Tools may thus either access the database directly, via programmatic interfaces (a less open solution), or depend on provision of data files in appropriate formats. This scheme allows us to enjoy the advantages of having a repository, but alleviates the burden of model maintenance by all participating tools.

Once a tool has manipulated the model, the repository is updated. This approach is discussed further in Section 2.4.3.

## 2.3 Derived Relationships

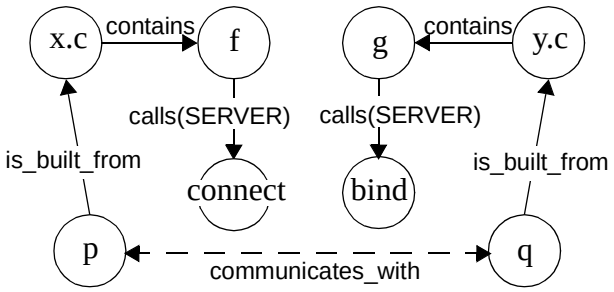


**Figure 2: A derived relationship**

Extraction of information is the foundation of architecture reconstruction. But it is also important to *augment* the collection of relations stored by the repository. Figure 2 illustrates an example from socket-based IPC. The circles represent extracted elements from the concrete model: functions (f, g, connect and bind), files (x.c and y.c) and processes (p and q). Solid arrows represent relations, labelled with the type of relation and, in the case of calls, tagged with attributes (SERVER). The dashed line, labelled communicates_with, depicts a *derived* relationship. Process q acts a server because it is built from a file containing a function that calls bind. p is a client because it is built from a file containing a function that calls connect. We know that p and q are a client/server pair because the arguments to connect and bind are identical. Thus, p communicates with q and *vice versa*. This relationship could have instead been defined asymmetrically, for example, as "p connects_to q".

Derived relationships of this type provide a mechanism for abstraction over the extracted artifacts, thus creating new views of the architecture. For this reason, flexible relationship derivation is an import feature in Dali. This functionality is facilitated by the (off-the-shelf) SQL database that provides Dali's central repository [21].

## 2.4 Model Manipulation: Organizing the Evidence

Perhaps the most important component of the Dali workbench is its interaction element: the central component that is used to directly manipulate the model and guide analyses and automatic manipulation. This component is best realized by a flexible tool that effectively balances generality with domain applicability. For example, a generic graphics package would provide a great deal of generality, but little or no functionality specific to the domain of software. A tool such as DISCOVER [20], on the other hand, provides significant domain functionality but little flexibility. We have found Rigi [24] to be a satisfactory compromise between these competing concerns: Rigi provides generality via a control language based on TCL, called RCL (Rigi Command Language). RCL also provides functionality specific to the tasks of manipulation of software models, satisfying the domain applicability concern.

While Rigi in fact comprises both an extraction component (a parser) and a user interface component, and while both of these are applicable within the Dali workbench, we are currently only using Rigi's user interface. The Rigi user interface (rigiedit), at its most basic level, provides graph editing functionality: layout, annotation, subgraph collapsing and so forth. At a higher level, Rigi pro-

vides semi-automatic facilities for subsystem identification based graph-theoretic properties such as interconnection strength. In experimenting with Rigi's more advanced functionality, we did not find that the automatically identified subsystems corresponded with the architectural components we wished to identify. Our alternative to using automatic architectural discovery techniques is to allow direct manipulation of the model augmented by external manipulation and analysis tools.

### 2.4.1 Direct Manipulation

Notwithstanding any foreseeable advances in automatic manipulation or pattern matching of software models, it is necessary to provide a direct manipulation interface for users. Facilities such as those mentioned above (layout, annotation, subgraph manipulation) enable a user to impose their *interpretation* of the architecture on the model being manipulated.

Designers have standard ways in which they visually organize the components of their architecture that correspond to their understanding of some property of the system (e.g. top-down control flow or bottom-up data flow). So, it is helpful to be able to visually reorganize the currently visible set of components arbitrarily to reflect this understanding.

### 2.4.2 External Manipulation and Analysis

Dali supports the ability to opportunistically apply tools to the current architectural model. This is achieved through a three step process:

1. exporting the model (or some subset thereof) from Rigi,
2. applying an appropriate tool to manipulate or analyze the model, and
3. (optionally) importing of the result.

The simple example of this process exists within Rigi by default: Rigi can export a snapshot of its current model in a graph representation language, execute an external graph layout algorithm, and then import the result. We have extended this approach to incorporate tools that structurally alter the model. We do this by implementing, in RCL, "glue code" that synchronizes Rigi's model with any changes to the model from the application of external tools.

The central technique for external model manipulation within Dali is based on queries over the SQL database that stores the model. Although the SQL query language does not provide sufficient expressive power to describe constructs such as the transitive closure of a relation, we have found that it is sufficient to allow model manipulation based on structural relationships as well as attributes of elements and relations. This is because our model of manipulation is iterative, depending on the user to define higher order relations.

These query-based manipulations are the basis for the architectural patterns that collectively relate an as-built architecture to an as-designed architecture. In addition, they provide the foundation for one mechanism of architectural *pattern-matching*. These applications of query-based manipulation will be discussed in Section 3.

One goal of reasoning architecturally about a system is to support the ability to analyze an architecture for properties such as availability, performance, security, schedulability and so forth. The literature provides many techniques for such analyses and many tools have been developed to perform them (e.g. [13], [19]). Dali provides a mechanism by which analyses can be performed on the architectural model currently being explored, using the export-process-import model described above. Integration of a new technique for which an analysis tool exists is simply a matter of implementing appropriate translators to produce a format acceptable to the tool and to interpret its results (if appropriate). We have done this with IAPR [12], a tool that determines the presence of patterns in a software architecture, and with RMTool [14], a tool that measures architectural conformance. Each of these generates output that is viewed by other external tools.

### 2.4.3 Repository Synchronization

The concrete model within Rigi is a local "working copy" of the model stored within the SQL database. It is important to recognize the potential difficulties resulting from lack of synchronization between this local working version and the central repository. The most significant example is the query-based manipulation mechanism outline above, as a query over the SQL database may manipulate the model in ways that conflict with what is currently being displayed by Rigi.

Repository synchronization is achieved by additional RCL code that updates the SQL database with modifications made to the Rigi model (those made by external tools or by direct user interaction).

## 2.5 Tools of the Trade: Populating the Dali Workbench

The Dali workbench, as described, specifies only one constraint, the use of a database for central model storage. The other elements of the system: extraction techniques, methods for combination of extracted data, the visualization system and its interaction, and particular analysis or manipulation tools, are left unspecified. This is in theory. In practice, we have of course specified tools, as illustrated in the preceding discussion of Rigi for visualization and interaction. We currently populate the rest of workbench as follows:

- Lightweight Source Model Extraction (LSME) [16], Imagix [8], make, and Perl [23] for extraction of source model information for C and C++,
- gprof for extraction of dynamic (profile) information,
- PostgreSQL (based on POSTGRES [21]) for model storage,
- IAPR (Interactive Architecture Pattern Recognition) [12], RMTool [14], and Perl for analysis and manipulation.

As Dali has an open, lightweight architecture, replacement of any of these components or inclusion of new components is intended to be, and has proven to be, straightforward.

# 3 Playing Detective: Dali on the Streets

This section describes the application of Dali to the architectural reconstruction of two systems, both implemented in C++: VANISH [11], a 50 KLOC system for prototyping visualizations, and UCMedit, a 15 KLOC system for creating and editing Buhr-style use case maps [4]. We liken this process to detective work: we gather evidence, pose hypotheses that organize the evidence and view and interpret the resulting organization. We iterate through this process until we are satisfied with the results. What does in mean to be satisfied with an architectural representation? We discuss this in Section 4.

## 3.1 Extraction

Extraction of static source models for both VANISH and UCMEdit was performed using Lightweight Source Model Extraction (LSME) [16]. To apply LSME, one provides a set of patterns specified as regular expressions and a set of actions written in the Icon programming language [7]. Actions are executed when patterns match elements of a source corpus. To perform extraction of elements from C++ using LSME, a complex set of patterns and actions were developed.

The elements and relations that were extracted are shown in Table 1.

| **Relation** | **"From" Element** | | **"To" Element** | |
|---|---|---|---|---|
| | **Element Type** | **Element Name** | **Element Type** | **Element Name** |
| calls | function | tCaller | function | tCallee |
| contains | file | tContainer | function | tContainee |
| defines | file | tFile | class | tClass |
| has_subclass | class | tSuperclass | class | tSubclass |
| has_friend | class | tClass | class | tFriend |
| defines_fn | class | tDefined_by | function | tDefines |
| has_member | class | tClass | member variable | tMember |
| defines_var | function | tDefiner | local variable | tVariable |
| has_instance | class | tClass | variable | tVariable |
| defines_global | file | tDefiner | global variable | tVariable |

**Table 1: Elements and relations extracted from VANISH and UCMEdit**

It is important to note that variable accesses are not included in Table 1; that is, there are no relations "function reads variable" or "function assigns variable". LSME was not designed with these relations in mind. A second extraction technique, based on the Imagix [8] C++ parser, was incorporated to accomplish the extraction of this disjoint set of relations. The Imagix C++ parser (a component of the Imagix 4D program understanding tool) generates rich and detailed output, including

variable accesses, in any easy-to-process format. A simple Perl script post-processes this output, generating the relations of interest. Additional relations, of the form “file depends on file”, are extracted by processing the output from running the GNU make utility on the application’s makefile.

Once the concrete model of interest was extracted, functions thought to be “uninteresting” were filtered out; among these are built-in functions such as `return` and standard C library functions such as `scanf` and `printf`. Next, an SQL database was populated with the extracted relations. Two additional database tables, called `relations` and `components`, were defined for convenience: the former identifies all defined relation types and the latter identifies all defined components. The `components` table has an additional field (called `type`) that stores the component’s type (file, function, etc.).

## 3.2 There’s an Architecture In There?

The process of manipulating the concrete model to derive the as-built architecture of the system is an iterative, interactive and interpretive one. It requires the interaction of not just any person, but of a person familiar with the system. This interaction consists of alternating pattern definition and pattern recognition activities.

The end result will be a representation of information extracted from the as-built architecture, as organized by the analyst/detective. Clearly, it is possible to group source elements into architectures in a huge variety of ways. This is why the task must be done by someone familiar with the system’s design. Recall that we are interested in architectural conformance, which means that there exists an as-designed architecture, even if it is only in the architect’s mind.


**Figure 3: A raw concrete model: white noise**

But how do you get there from here? How do you get from a mountain of evidence to a concise, accurate representation of the architecture? We will illustrate the process of by walking through a

typical set of pattern applications in Dali that move an analyst from the raw data that is a concrete model to a (hopefully) simple elegant software architecture.

Consider Figure 3, which shows the raw extracted concrete model, containing 830 nodes and 2507 relations, of UCMEdit (the corresponding image for VANISH would not be recognizably different, with 2844 nodes and 7387 relations). This is the starting point of the architecture reconstruction process.

The following sections will first describe the use of *application-independent* patterns to transform the models of UCMEdit and VANISH, followed by a discussion of the use of patterns leveraging architectural information common to both applications. The examples will conclude with the application of direct manipulations and patterns specific to each system.

## 3.3 Application-Independent Patterns

The first step toward reconstructing a system's architecture is to apply several simple *low-level*, application-independent patterns to augment and simplify the raw concrete model with some derived information. Patterns are specified as sets of SQL queries; a Perl wrapper acts as the "glue" for importation of the results into Rigi. The first pattern set is used to identify the types of the source elements, as shown in Figure 4.[1] A pattern set comprises a series of patterns, where each pattern is made up of an SQL query and a Perl expression. The former selects a set of elements from the concrete model and the latter post-processes the query, extracting and manipulating important fields. For example, consider the second group of patterns in Figure 4, that specifies the element type "class". This group comprises three patterns used to identify sets of elements that are classes. These sets may or may not overlap with each other.

The first of the class patterns selects all "definers" from the defines_fn relation. That is, any element that is the definer in a "defines function" relation is a class. The second pattern selects all components that "have instances" and the third selects all components that are either a superclass or a subclass. The Perl expressions associated with each of these patterns simply generate output of

1. It should be noted that this step would not necessarily be required for all extracted concrete models. Some extracted concrete models would already contain typing information.

the form: “null <component> Class”, identifying <component> as a Class (the first field will be discussed further below). The output is processed by Rigi and used to update its internal model.

```
# File type.
SELECT tName
    FROM components
    WHERE tName LIKE '%.h'
        OR tName LIKE '%.cc';

print "null $fields[0] File\";

# Class type.
SELECT DISTINCT d1.tDefined_by
    FROM defines_fn d1;

print "null $fields[0] Class\n";

SELECT DISTINCT i1.tClass
    FROM has_instance i1;

print "null $fields[0] Class\n";

SELECT DISTINCT c1.tName
    FROM components c1, has_subclass s1
    WHERE c1.tName=s1.tSuperclass
    OR c1.tName=s1.tSubclass;

print "null $fields[0] Class\n";

# Member variable.
SELECT DISTINCT h1.tMember
    FROM has_member h1;

print "null $fields[0] MemberVarible\n";

# Local variable.
SELECT DISTINCT d1.tVariable
    FROM defines_var d1;

print "null $fields[0] LocalVariable\n";

# Global variable.
SELECT DISTINCT d1.tVariable
    FROM defines_global d1;

print "null $fields[0] GlobalVariable\n";
```

**Figure 4: Patterns for element types**

Any elements that have not been assigned a type by this pattern set will retain Rigi’s default type of “Function”.

At this point, we have a database synchronization problem, as discussed in Section 2.4.3: the type information that we have just derived is stored only in the Rigi model, not in the SQL database. To synchronize the two models, a Rigi RCL script is executed to export the derived types back into the database.

Thus far the concrete model as displayed in Rigi will be indistinguishable from Figure 3; we have not yet made any structural modifications to the model, we have only identified the types: Function, File, Class, LocalVariable and GlobalVariable. The next collection of low-level patterns applied to the model group together functions with any variables that they declare locally.

Figure 5 shows the pattern set for function aggregation. This pattern set has the effect of incorporating a function and all of the local variables that it defines into a new composite component. This new component has the original function's name appended with a '+'. The Perl expression

```
print ”$fields[0]+ $fields[0] Function\n”;
```

specifies the aggregation. In this expression, $fields[0]+ identifies the name of the new composite, $fields[0] is the name of the original function and Function is the type of the new composite.

```
# Function grouping.

SELECT tName
    FROM components
    WHERE tType='Function';

print ”$fields[0]+ $fields[0] Function\n”;

SELECT d1.tDefiner, d1.tVariable
    FROM defines_var d1;

print ”$fields[0]+ $fields[1] Function\n”;
```

**Figure 5: Patterns for function aggregation**

After application of this pattern set, the models for UCMEdit and VANISH *still* appear as inscrutable webs of nodes and arcs. However, they are simpler than the concrete model of Figure 3, prior to the application of the function aggregation patterns. The UCMEdit model now shows 710 nodes and 2321 relations and the VANISH model shows 2282 nodes and 6586 relations.

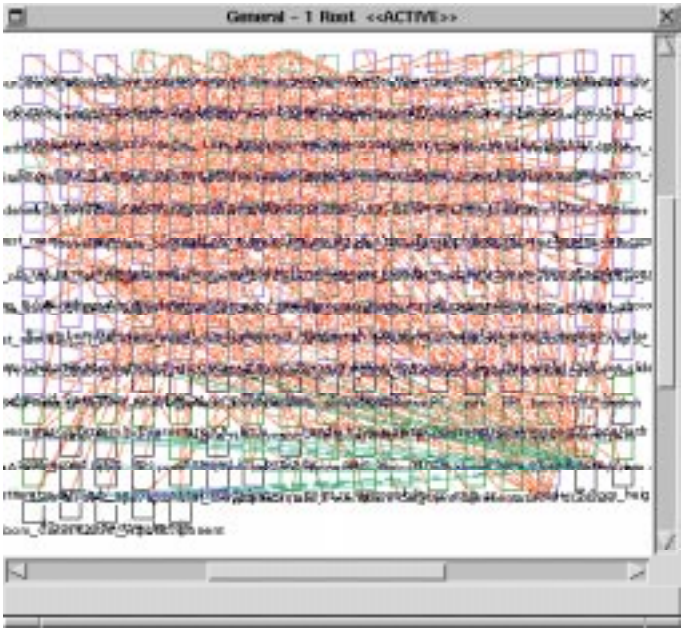

**Figure 6: The UCMEdit concrete model after collapsing classes**

The next low-level pattern set applied is similar in nature to that for collapsing functions, but generates a much more significant visual effect. This pattern set collapses together classes, their member variables and their member functions. The resulting (top-level) concrete model for UCMEdit is shown in Figure 6; it contains 233 nodes and 518 arcs. The VANISH (top-level) concrete model now contains 798 nodes and 2359 arcs. The dramatic simplification of the models is one that should be expected after application of these patterns to object-oriented systems. By virtue of the fact that there are still elements that are not related to any class in the concrete model we

have already exposed either a deficiency in the extractors applied, or ways in which these systems deviate from pure object-oriented designs. In fact, each of these cases has occurred.

There are false positives generated by the LSME extraction patterns in the form of apparent calls to global functions that are actually calls to member functions, and there are several functions that are indeed global functions, belonging to no class defined in the system. Of course, some global functions, in the form of system calls or windowing system primitives, are necessary. How these "left-over" cases are separated from the rest of the architecture is discussed below.

The models for UCMEdit and VANISH are now collections of files, classes, left-over functions and global variables. Local variables have been incorporated into the functions in which they are defined and member functions and member variables have been incorporated into their associated classes. At this point we can compose global variables and functions into the files in which they are defined, in the much the same manner as functions and classes were composed. The resulting models, shown in Figure 7, contain three separate groups of elements: files, classes and the remaining left-over functions.

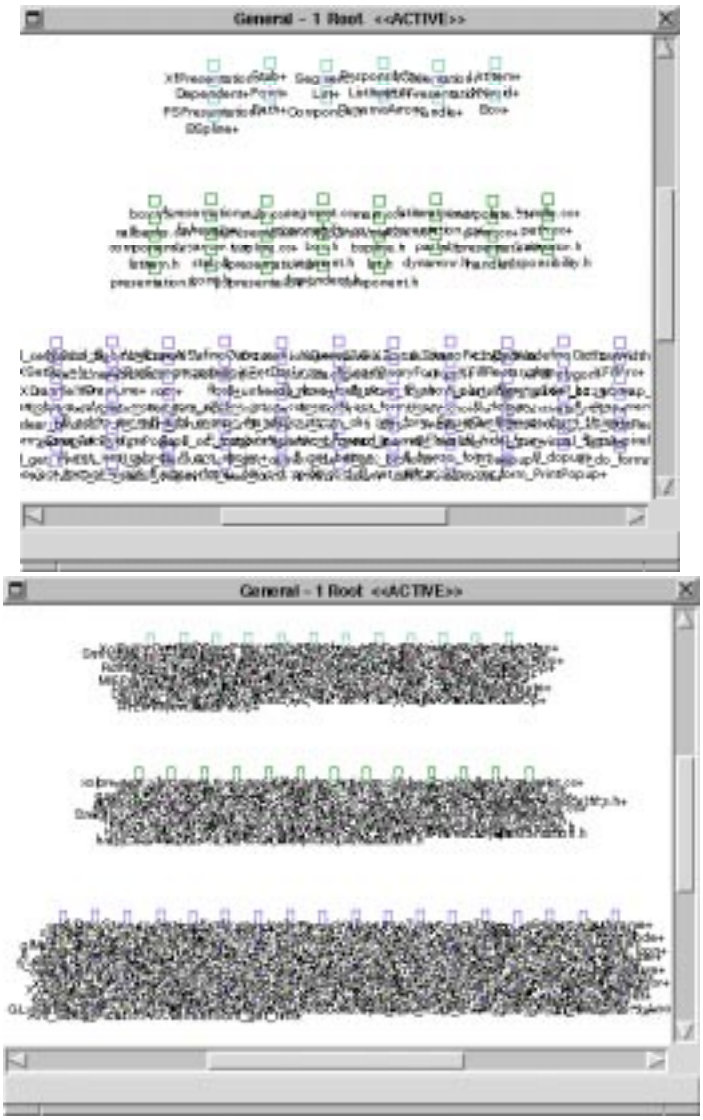


**Figure 7: The UCMEdit and VANISH models showing (from top to bottom) classes, files and "left-over" functions (arcs are hidden)**

## 3.4 Common Application Patterns

Until now, each of the pattern sets applied has been application-independent, but specific to the extraction techniques and to the domain of C++ software. The next sets of patterns to be applied use expert knowledge of the architectures of UCMedit and VANISH. At this point the reconstruction process diverges from a rote analysis, where we apply off-the-shelf patterns, into opportunistic pattern recognition and definition, leveraging the kinds of information that a designer or experienced system programmer could be expected to know about a specific system's architecture.

The first application-specific knowledge that we apply to our example systems is as follows:

- they are both interactive, graphical applications;
- they both attempt to encapsulate access to the underlying windowing and graphics subsystem within a layer; and
- the functions comprising the graphics libraries used (Xlib, XForms and Mesa) have characteristic naming conventions.

These observations lead to the pattern set shown in Figure 8, intended to identify the graphics subsystem, those external functions providing rendering and interaction functionality to the application. (The patterns shown are for UCMEdit; the VANISH patterns are only slightly different, due

```
# 1: Identify calls from graphics access layer.
DROP TABLE tmp;
SELECT * INTO TABLE tmp
    FROM components;
DELETE FROM tmp
    WHERE tmp.tName=defines_fn.tDefines;
SELECT t1.tName
    FROM tmp t1, calls c1, defines_fn d1,
        has_subclass s1, has_subclass s2
    WHERE t1.tName=c1.tCallee
    AND c1.tCaller=d1.tDefines
    AND d1.tDefined_by=s1.tSubclass
    AND s1.tSuperclass='Presentation';

print "Graphics $fields[0]+ null\n";

# 2: Identify calls to Mesa functions.
SELECT tName
    FROM components
    WHERE tType='Function'
    AND tName LIKE 'gl%';

print "Graphics $fields[0]+ null\n";
```

```
# 3: Identify calls to XForms functions.
SELECT tName
    FROM components
    WHERE tType='Function'
    AND tName LIKE 'fl_%';

print "Graphics $fields[0]+ null\n";

# 4: Identify calls to Xlib functions.
DROP TABLE tmp;
SELECT * INTO TABLE tmp
    FROM components;
DELETE FROM tmp
    WHERE tmp.tName=defines_fn.tDefines;
SELECT c1.tName
    FROM tmp c1
    WHERE tType='Function'
    AND tName LIKE 'X%';

print "Graphics $fields[0]+ null\n";
```

**Figure 8: Patterns for UCMEdit graphics subsystem**

to a more elaborate encapsulation of the graphics layer.) Consider the first pattern. This pattern first constructs a new table from the components table by filtering out all functions that are members of classes (those that appear as the tDefines field in a tuple of the defines_fn relation). Then the pattern selects from this new table all functions that are called by functions defined by subclasses of the Presentation class. Note that this pattern references subclasses of the Presentation class. In doing so, it implicitly identifies the layer that the original designers created to encapsulate accesses to the graphics subsystem. This information will be leveraged further below. The second, third and fourth patterns in this pattern set identify functions defined by the Mesa, XForms and Xlib libraries, respectively, by specifying patterns over the function names.

These patterns collectively identify an architectural component, called Graphics. This component does not exist in the extracted information, but it does exist in the as-designed architecture. This is an example of linking the as-built and as-designed architectures through a cumulative series of

pattern applications. The results of the application of this pattern set to the UCMEdit model are shown in Figure 9.

Note that the names of the elements to be aggregated into the Graphics component include the '+' that was appended by the patterns in Figure 5. This technique thus refers to previously constructed composite elements without the patterns explicitly querying the database for the composites. An alternative approach for synchronizing the database with the interaction component is to populate the database with relations reflecting the compositions and include these new relations in the pattern queries. We chose to avoid this alternative for efficiency considerations, but it is supported equally well by Dali.

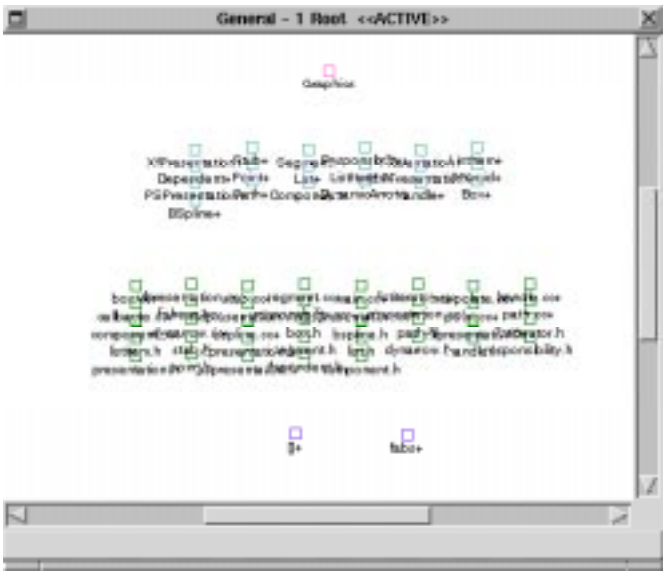


**Figure 9: UCMEdit model showing the Graphics subsystem, classes, files and remaining functions (arcs are hidden).**

Examining Figure 9, we see that there are only two left-over functions remaining: fabs and []; the latter is obviously an extraction error while the former is a math library function that should have been filtered out along with standard C library and built-in functions. Regardless, neither of these functions are of interest, and can thus be pruned from the model. Identifying the graphics subsystem for the VANISH model has reduced the number of left-over functions by more than half, from 259 to 120. These were similarly identified as either extraction errors or uninteresting low-level functions and pruned from the model.

Rather than identifying uninteresting functions as a side-effect of creation of the graphics subsystem, a more direct approach could have been applied. Such an approach would simply have selected all functions not contained within a source file of the system or defined by a class of the system and removed them from the model.

It is important to realize that the determination of which functions are "interesting" or "uninteresting" is an arbitrary one. An analyst interested in a different aspect of the system, such as how its subsystems depend on platform- or operating system-specific libraries, would not have pruned these functions from the concrete model. These functions would more likely be aggregated into a

layer to analyze how they are used by the rest of the application. As we are interested in constructing an architectural representation of the application-specific part of the system we remove these types of functions from the model.

A second common application pattern set takes advantage of knowledge of the relationship between classes and files in the example applications, thus bridging two architectural views. First, a source (.cc) will contain functions for at most one class and second, a header (.h) file will contain a definition for at most one class. This makes it possible to define a unique containment relationship: a class can include the header file in which it is defined and the source file which contain its functions. The pattern set that generates these aggregations is shown in Figure 10. We see one additional feature of pattern specifications in this example: the last field in the Perl expression associated with the first pattern ($fields[0]++) specifies a renaming of the component being incorporated into an aggregate. In this pattern, we are incorporating classes (named with trailing '+'s due to the class-collapsing patterns of Section 3.3) into new composite components. The names of the new composites are <class>+ and the original class composites are renamed <class>++. The results are shown in Figure 11.

```
SELECT DISTINCT tDefined_by
    FROM defines_fn;

print "$fields[0]+ $fields[0]+ Class $fields[0]++\n";

SELECT DISTINCT d1.tDefined_by, c1.tContainer
    FROM defines_fn d1, contains c1
    WHERE c1.tContainee=d1.tDefines;

print "$fields[0]+ $fields[1]+ Class\n";

SELECT d1.tClass, d1.tFile
    FROM defines d1;

print "$fields[0]+ $fields[1] Class\n";
```

**Figure 10: Patterns for class/file containment**


**Figure 11: The UCMEdit and VANISH models after application of common patterns.**

## 3.5 Application-Specific Patterns

The patterns applied in Section 3.3 were completely application independent, while those applied in Section 3.4 were applicable to both systems because of commonalities in their construction. At

this point, the analyses of UCMEdit and VANISH diverge as we apply patterns specific to each application.

### 3.5.1 UCMEdit

UCMEdit was constructed as a prototype, intended to demonstrate the advantages of computer-based editing of use case maps. The high level architectural design of the application was not considered at the start of development; thus, identification of architectural components from the concrete model must be guided by an understanding of the structure of the application as it stands at the completion of development. Our understanding of the application will be imposed on the model via direct manipulation, as follows.

First, we know (and can tell by observation of the model) that callbacks.cc is central to the structure of the application, containing all of the system's event handlers and the bulk of the user interface implementation. Second, we can observe the obvious relationships between the two remaining files and the classes to which they are connected: interpolate.cc is associated exclusively with BSpline and fisheye.cc is used only by Box and Component. Third, we may now re-apply our knowledge of the structure of the system's graphics-encapsulation, or *presentation* layer: it is embodied in the Presentation class and its subclasses. Fourth, we can make the observation that the List, ListItem and ListIterator classes are functionally related to one another and are used by almost all of the other classes.

We realize the above observations in Dali by:

- identifying the callbacks.cc file with an architectural component, Interaction
- incorporating interpolate.cc into the BSpline component (we'll ignore the observation about fisheye.cc for now)
- aggregating the Presentation class and its subclasses into a Presentation component
- aggregating the List, ListItem and ListIterator classes into a List component and hiding it, treating it as a "utility layer"

The results of these changes to the model are shown in Figure 12. At this point, it is necessary to carefully consider how we may further simplify this model. Automatic clustering based on graph-theoretic properties, such as interconnection strength, does not provide any insight. Another option is to attempt to build layers based on the organization generated by the graph layout algorithm, as shown in Figure 12. However, this approach results in little functional consistency within the layers. Instead, we chose to cluster classes based on the domain of use case maps. Further discussion addressing how to choose appropriate architectural components appears in Section 4.

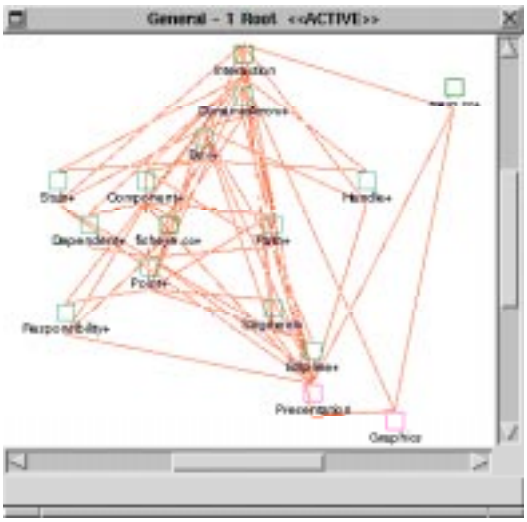


**Figure 12: UCMEdit model after application-specific direct manipulations**

After considering concepts from use case maps, we identified two broad categories of elements: those related to components and those related to paths, these being the two primary constructs comprising a use case map. DynamicArrow, Path, Point, Responsibility, Segment, Stub and BSpline are related to paths and Box, Component, Dependent, Handle and fisheye.cc are related to components. Figure 13 shows the effect of clustering these elements into two architectural components: Path and Component. In probing the connections between elements we find that they still comprise a large number of interrelationships. While in itself this is not necessarily harmful, it suggests that UCMEdit's architecture demonstrates a lack of functional consistency within the elements and their connections.

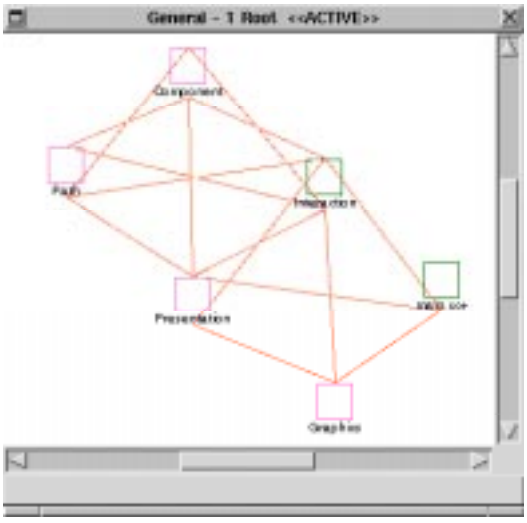


**Figure 13: UCMEdit model after clustering based on application-domain concepts**

Unfortunately, there are not any significant improvements we can make to the UCMEdit model. The system was not well designed—the mapping from functionality to software structure is complex. This makes the abstraction of functionally coherent high level components within UCMEdit's architecture impossible. However, we can take advantage of what we have learned, with further analysis, to suggest improvements to the UCMEdit design.

### 3.5.2 VANISH

With VANISH we have a different situation: VANISH was developed following an explicitly documented architectural design [11]. VANISH is a system for prototyping visualizations and as such must easily accommodate incorporation of new visualization domains as well as integration of new

presentation toolkits. The Arch metamodel of interactive software [22] is intended to provide exactly these benefits by specifying five architectural components:

- **Functional Core** - the system's core functionality or purpose;
- **Functional Core Adapter** - a mediator between the dialogue and functional core by providing a unified, generic view of the functional core to the dialogue;
- **Dialogue** - a programmable mediator between domain specific and presentation specific functions;
- **Logical Interaction** - a virtual interaction toolkit layer that mediates between the dialogue and the presentation; and
- **Presentation** - the toolkits that implement the physical interaction between the user and the application.

These components are arranged in a strictly layered fashion, as shown in Figure 14. The Arch metamodel in fact loosely defines an *architectural style*: a collection of component types and a set of constraints on their relationships. VANISH's architecture is a particular instantiation of this style. Thus, we should be able to identify these architectural components within the implemented VANISH system and, taking an optimistic attitude, verify that the as-built architecture conforms to the as-designed architecture.

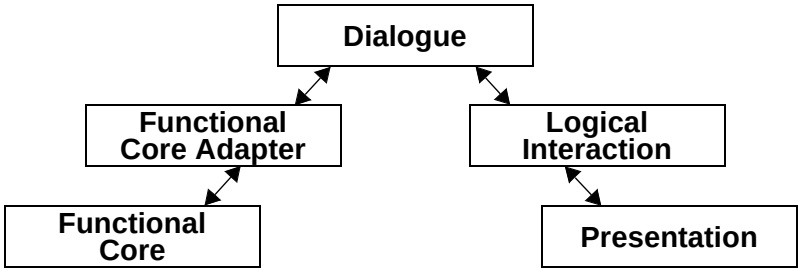


**Figure 14: The Arch metamodel of interactive software**

The first step towards uncovering VANISH's architecture is consideration of the files remaining in the model, as shown on the right of Figure 11. Because we know that VANISH has a single-process architecture, and we know the name of the executable file, we can determine which of the remaining files are "interesting" by applying the depends_on relation extracted from the system's makefile. Source files that do not contribute to the construction of the executable can be removed from the model. Examination of these unused files identifies them as either "dead code", as elements of earlier versions of the application or as tools used to test particular aspects of the application's functionality. This pruning is an example of how one architectural view can be used

**Figure 15: VANISH model after removal of unused files**

to constrain another. Figure 15 shows the VANISH model after the removal of unused files. The two remaining files are interpolate.cc and vanish-xforms.cc; the former contains global functions used exclusively by the BSpline class and the latter is the main initialization and event handling component of the application. Direct manipulation is used to include interpolate.cc in the BSpline aggregate.

Note that this technique would provide additional insight when manipulating a multi-process system: candidate processes could be identified from analysis of the depends_on relation and used as top-level architectural components. Thus, a run-time “process view” could be developed.

Now we are ready to apply the pattern set that identifies the top-level architectural components in VANISH; it appears in Figure 16 and defines architectural components as follows:

```
SELECT tSubclass
    FROM has_subclass
    WHERE tSuperclass=’Presentation’;

print ”Logical_Interaction $fields[0]+ null\n”;

SELECT tName
    FROM components
    WHERE tName=’Presentation’
    OR tName=’BSpline’
    OR tName=’Colour’;

print ”Logical_Interaction $fields[0]+ null\n”;

SELECT s1.tSubclass
    FROM has_subclass s1
    WHERE s1.tSuperclass ~ ’Presentation’
    AND s1.tSuperclass !~ ‘^Presentation$’;

print ”Presentation $fields[0]+ null\n”;

SELECT tName
    FROM components
    WHERE tName=’BaseNode’
    OR tName=’BaseAttribute’;

print ”Functional_Core_Adapter $fields[0]+ null\n”;

SELECT tSubclass
    FROM has_subclass
    WHERE tSuperclass=’BaseAttribute’;

print ”Functional_Core_Adapter $fields[0]+ null\n”;

SELECT tSubclass
    FROM has_subclass
    WHERE tSuperclass=’BaseNode’;

print ”Functional_Core $fields[0]+ null\n”;

SELECT tSubclass
    FROM has_subclass
    WHERE tSuperclass=’PrimitiveOp’;

print ”Dialogue $fields[0]+ null\n”;

SELECT tName
    FROM components
    WHERE tName=’vanish-xforms.cc’
    OR tName=’PrimitiveOp’
    OR tName=’Mapping’
    OR tName=’MappingEditor’
    OR tName=’MappingLibrary’
    OR tName=’Attributes’
    OR tName=’Application’
    OR tName=’Renderer’
    OR tName=’InputValue’
    OR tName=’Point’
    OR tName=’VEC’
    OR tName=’MAT’
    OR ((tName ~ ’Dbg$’ OR tName ~ ’Event$’)
        AND tType=’Class’);

print ”Dialogue $fields[0]+ null\n”;

SELECT tName
    FROM components
    WHERE (tName ~ ’^List’
    OR tName~’Map$’ OR tName=’Socket’)
    AND tType=’Class’;

print ”Utilities $fields[0]+ null\n”;
```

**Figure 16: Patterns for VANISH architecture**

- The Logical Interaction component is composed of the Presentation class, its subclasses, and two generic interaction-related utility classes, BSpline and Colour.
- The Presentation component is composed of classes which have a superclass whose name properly contains the substring “Presentation”.
- The Functional Core Adapter is composed of the BaseNode class and the BaseAttribute class and its subclasses.
- The Functional Core is composed of subclasses of the BaseNode class.
- The Dialogue is more complex. In VANISH, it is realized by a visual programming language. The first pattern for the Dialogue specifies that all subclasses of the PrimitiveOp class are included. The second pattern enumerates the other elements that comprise the visual programming language. This enumeration was developed by iterative application of this second pattern, starting with an initial guess at which classes should be included and refining the enumeration as additional classes were identified.
- A final Utilities layer is composed of a set of generic list manipulation classes used by many other classes in the system.

Also, the Graphics component identified in Section 3.4 is incorporated into the Presentation component. The resulting model, both with the Utilities component shown and hidden, is presented in Figure 17.

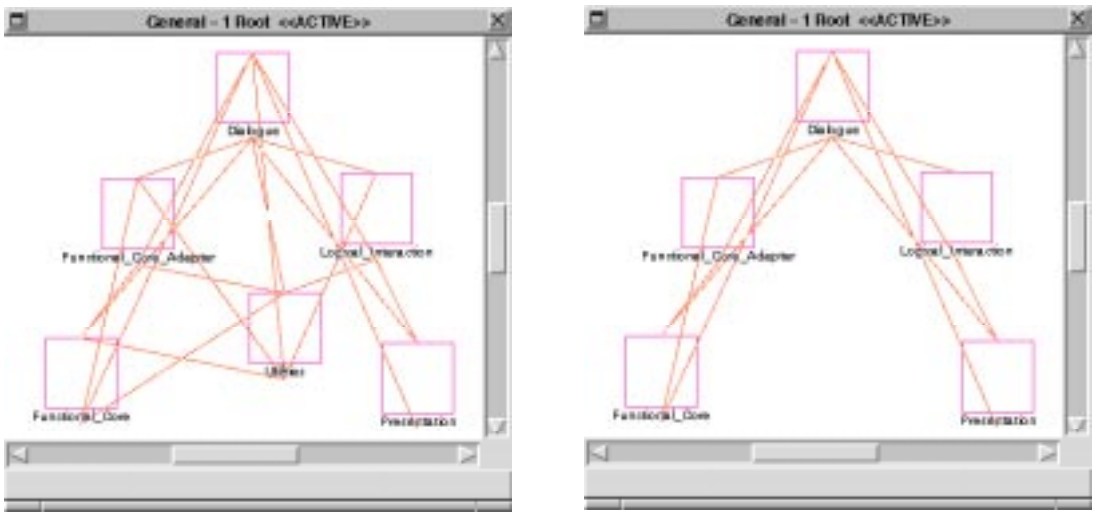


**Figure 17: The VANISH architecture (with and without Utilities layer)**

We are now in a position to consider how well VANISH’s as-built architecture conforms to its as-designed architecture. We can immediately see that the basic arch shape is present; there are no connections between components on opposite sides of the arch. However, it is also immediately apparent that there as several instances of *layer bridging* in the architecture: between the Dialogue and the Functional Core, and between the Dialogue and the Presentation. These are architectural *deviations*. We can identify three classes of deviations:

- **acceptable** - Although the architecture was not designed with a particular feature, the feature does not degrade the conceptual integrity [3] of the architecture. The architectural description against which conformance is being tested need not be updated to reflect the feature.
- **exceptions** - As for acceptable deviations, exceptions do not degrade the conceptual integrity of the architecture. Exceptions should be incorporated into the model against which conformance is being tested.
- **opportunities for improvement** - The architecture's conceptual integrity *is* degraded by the deviation. The implementation should be modified to remove it.

We can consider the existence of the Utilities layer an acceptable deviation: it does not affect the conceptual integrity of the intended architecture, nor does it contribute to its overall structure, so it should be not incorporated into the architectural description.

The connections between the Presentation and the Dialogue are exceptions that should be documented in the system's architectural description. The connection from the Presentation to the Dialogue represents classes in the Dialogue directly instantiating classes in the Presentation (has_instance relations). This appears to be a violation of the constraints of the Arch model. However, this violation is necessary due to the way the Logical Interaction was implemented in VANISH. The Logical Interaction layer (the Presentation class and its immediate subclasses) provides an abstract interface to the concrete classes in the Presentation component—AmPresentation, MIFPresentation, XfPresentation, etc. For the Dialogue to take advantage of this abstraction layer, it must directly create instances of the concrete classes. Thereafter it refers to an abstract class from the Logical Interaction layer, and these references are resolved via polymorphism into calls to the appropriate concrete class. Therefore, the architectural implications of this layer bridging are constrained.

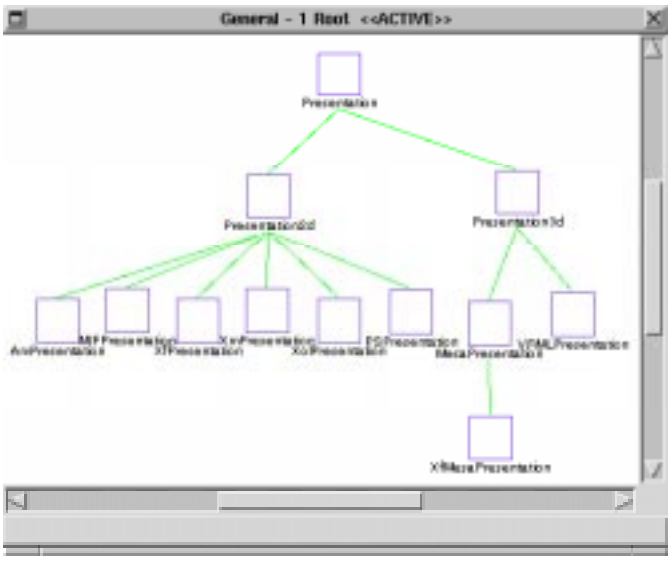


**Figure 18: Presentation and its has_subclass descendants**

The connection from the Dialogue to the Presentation represents calls to the constructors of the classes mentioned above. This, then, is an exception for the same reasons.

The structure underlying the Functional Core Adapter and the Functional Core is much like that of the Logical Interaction and the Presentation: the BaseNode class (in the Functional Core Adapter) provides an abstract interface to the concrete classes that make up the Functional Core. Thus connection from the Dialogue to the Functional Core is identical in nature to that from the Dialogue to the Presentation: it is made up of calls to constructors of the concrete classes. This is also a justifiable architectural exception.

Finally, the connection from the Functional Core to the Dialogue must be examined. One might think that it is analogous to the connection from the Presentation to the Dialogue, for the reasons described above. However, probing the connection shows that the relations which comprise it are not has_instance relations because no class within the Dialogue maintains an instance of a class from the Functional Core. Instead, these relations are calls from classes in the Functional Core to classes in the Dialogue. These calls expose the Functional Core to the details of the Dialogue, bypassing the stated intent of the Functional Core Adapter: to keep these components isolated. This is an opportunity for improvement as it degrades the conceptual integrity of the connection between the Dialogue and the Functional Core Adapter. Also, unnecessary coupling between the components will hamper integration of new visualization domains, which is central to the stated objectives for VANISH.

Analysis of architectural conformance, such as that performed above, can be supported by an automatic tool such as RMTool [14]. We use RMTool by giving it a description of an architectural representation of a concrete model (as reconstructed using Dali) as well as an as-designed architecture and it computes the conformance of the former to the latter. This provides a documented starting point for an examination of architectural deviations.

## 4 Assessing the Evidence: Why Not Any Four Boxes?

What does it mean to reconstruct the software architecture of a system? We started off this paper with the claim that software architecture was, in some ways, a mass hallucination; that it doesn't really exist in anything that you can directly examine. Software designers create it and then hope that what is implemented properly reflects what was designed.

The problem for a software analyst attempting to understand and assess an architecture is therefore one of being software detective; sifting through clues and putting evidence together in coherent patterns. To use a different metaphor, it seems that you could mold any number of software architectures out of the amorphous clay that is a concrete model. So, why can we just not recursively group extracted information together until we have achieved a picture with four boxes and a few connections?

How do we know what is a good, meaningful architecture? This is really asking two questions: what makes a "good" architecture, and how do we know when we've found it?

Our guide in reconstruction is that a good architecture exhibits conceptual integrity [3]: it is built from a small number of structures that are connected in regular ways. But this is only half of the battle. The allocation of functionality to the components and connectors of the architecture should be *congruent* with the structure [9]. Thus, any four boxes are probably not a *good* architecture because the components of those boxes have little to do with each other in terms of the system's functional decomposition.

We have used this rule of thumb as our guide in deciding when to group components together in the examples of Section 3. Consider the application-independent patterns (presented in Section

3.3). These patterns group together components that are functionally coherent, irrespective of the application. Grouping a class's member functions and member variables together is one example of this.

The common application patterns for VANISH show an application-dependent version of this principle: in Section 3.4 we showed how VANISH's Presentation layer consists of all and only those classes that make calls to functions in Mesa, XForms, Xlib, etc. The Presentation layer is functionally coherent, which is to say that the functional decomposition provided by the Arch metamodel is respected by the structure of VANISH: it consists of all and only presentation functionality.

By way of contrast, the structure of UCMEdit does *not* show a consistent allocation of functionality onto structure, which is why, even though we can build a relatively simple architecture for UCMEdit, this architecture does us no good. We, as analysts, get little insight into the system by looking at this simple structure, because the components and their interconnections are not functionally consistent.

# 5 Related Work

Dali owes much to Murphy's work on LSME [16] and RMTool [14]. LSME provides a substantial amount of the source model extraction to populate Dali's database. RMTool has the same goals as Dali, but with a narrower scope, extracting only from source artifacts and lacking the tightly coupled user interaction. More importantly, RMTool provides a language of limited expressive power for construction of architectural components from source elements. As mentioned earlier, RMTool has been incorporated as one of the analysis options in Dali.

Another system that is closely related to Dali is ManSART [25], a tool for recovering architectural information and manipulating views of that information. Dali and ManSART are similar in the following ways:

- Both systems have, as their goals, the reconstruction of architectural information from lower-level sources.
- A view, in ManSART terms, is generated by an operator operating over the results of some recognizer. This is similar to our notion of a view being generated by one or more patterns (SQL queries), operating over a table populated by some extraction tool.
- Both systems visualize the resulting information, and provide some means for an architect to interact with extracted and derived information.

Dali and ManSART are different in some important ways as well:

- ManSART recognizers depend exclusively on parsing and the production of an Abstract Syntax Tree (via REFINE/C [17]). Therefore it is harder for ManSART to switch languages/dialects, because it has to integrate an appropriate parser into the system, and it requires compilable code.
- ManSART concentrates on the code as the artifact of interest; it does not appear to extract non-language artifacts (such as makefiles or profiling information).

- ManSART lacks the notion of combining *the same kind of information* from multiple sources in a fault tolerant manner.
- Our architecture is generic. We store information in a standard relational database, and we can thus accept input from any source, as long as it creates a table in the database. We could, in principle, integrate ManSART as a part of a Dali workbench.
- ManSART's approach is more 'heavyweight' in terms of computation and infrastructure. Recognizers are "complex combinations of feature extractors and view manipulators" [25]. Dali, on the other hand, extracts everything and lets the user iteratively and incrementally build up a view through opportunistic pattern recognition and definition. In this way our approach is more 'lightweight'.

Dali is also related to the Desire system, developed at MCC [2]. Desire was developed with the goal of recovering detailed design information from implemented systems to support maintenance and the harvesting of reusable components. To represent a system, Desire constructs a dictionary of system constructs and applies Prolog queries to probe the dictionary. This is analogous to Dali's use of a repository. Both Desire and Dali apply pattern matching to identify constructs in a model of the system under study. The central element of Desire is a domain model that stores conceptual abstractions to be matched. These conceptual abstractions comprise idioms made up of linguistic and structural patterns. Conceptually, both Dali and Desire recognize the importance of an expert. Desire relies on the knowledge of an expert to identify the conceptual abstractions that populate the domain model.

However, there are some differences between Desire and Dali. While Desire is focussed on the recovery of detailed design information, Dali's goal is the construction of high level architectural representations that can be used to evaluate properties of the system has a whole. Desire has as an additional goal the harvesting of reusable components. In contrast, Dali identifies architectural constructs. Also, Desire is not intended to be open or lightweight. It depends on its "out of the box" functionality to meet its goals. As with ManSART, a Dali workbench could incorporate Desire as appropriate to the analysis task at hand.

# 6 Post Mortem

This paper has presented the Dali workbench for architectural extraction and reconstruction. The tool was born out of our need to have *something* to examine when we do architectural analyses. Frequently we are asked to analyze a system's software architecture and are given only its code and the (limited) time of a designer.

This tool goes a long way in helping an analyst *reconstruct* a software architecture from various artifacts such as source code, makefiles and profiling information. Dali allows an analyst to interact with the recovered information: by assessing the results of the reconstruction effort to see whether composite elements demonstrate functional consistency, and by seeing places where the as-built architecture differs from the as-designed architecture.

We have not attempted a “big-bang” solution here. One of our emphases has been to provide an open, lightweight environment so that tools can be integrated opportunistically. We believe that no single tool is right for all jobs. Certainly, extraction demands different tools for different languages and styles of systems. Our other emphasis is that no extraction technique is useful or complete without user interaction. In some respects, software architecture *is* a mass hallucination, but a convenient and useful one; one that is created by and for humans. So a human must be part of the recovery process: interpreting evidence and creating and testing theories.

# 7 The Next Case: Where Do We Go From Here?

Dali is still an experimental system. In the near term, there are three directions in which we want to extend this work: extend its scope, improve its extraction capabilities, and improve its user interaction. We will discuss each of these directions briefly.

We have only applied Dali to half a dozen systems, all of which used dialects of C or C++. Our intention is to extend Dali’s scope by appyling it to the extraction and analysis of larger systems and other languages (particularly legacy COBOL and Fortran systems). This means augmenting Dali’s extraction capabilities with tools and techniques appropriate to other languages. Similarly, we want to augment Dali’s analysis capabilites by integrating other tools. One avenue is to export and import an ACME [5] representation of the architecture from Dali. We view these enhancements of Dali as relatively rote and easily achieved.

An area of more speculative research is improvement of Dali’s capabilities for combination of extracted information. Extraction is an error-prone process [15]. We realize that no extraction tool is going to reliably retrieve everything of interest about an architecture. So, when dealing with noisy, error-prone input, a sensible thing to do is to not rely on any single source of information. We must do multiple extractions and combine them intelligently. The meaning of “intelligently” is the research area here. There are several possibilities: after characterizing the trustworthiness of individual techniques in extracting particular source elements the techniques would be used to populate only those database relations for which they are “trusted”. Additionally we can take a fault-tolerance view of the problem, and have each of the extraction techniques vote on a particular source element. We would only accept the information into the database when a majority of the relevant extraction techniques agree. Or, we could use a hybrid of these two techniques, where the vote of each extraction technique is weighted according to the trustworthiness of that technique.

Another way of making Dali more useful is to improve its interaction with the user. This, once again, has a near-term and a long-term aspect. In the near term the addition of a history mechanism, with the option of playback will make a user braver in taking exploratory excursions, knowing that these can always be undone. Providing user-guided pattern inferencing, where a user modifies the architecture through direct manipulation and the system *infers* the architectural rules from this interaction, is our long-term user interaction goal.

Finally, we see our holy grail as a “round trip”; proceeding from an as-implemented architecture to a corresponding design, to a redesigned architecture (using a tool such as UniCon [18], that supports architectural design leading to implementation), to a re-implemented architecture. Such a goal is difficult to achieve in practice, because interconnection mechanisms are deeply buried within an implementation. However, a round trip might be practical in a less ambitious form, where one reconstructs an architecture to determine how difficult it would be to change its connection mechanisms. This might be used in making a legacy system “Web-enabled”, or distribute it using CORBA.

# 8 References


[1] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1997 (in press).

[2] T. Biggerstaff, “Design Recovery for Maintenance and Reuse”, *IEEE Computer*, Vol. 22, No. 7, July 1989, pp. 36-49.

[3] F. Brooks, *The Mythical Man-Month—Essays on Software Engineering,* Addison-Wesley, 1975.

[4] R. Buhr, R. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, 1996.

[5] D. Garlan, B. Monroe, and D. Wile. “ACME: An interchange language for software architecture, 2nd edition”, Technical report, Carnegie Mellon University, 1997.

[6] D. Garlan, M. Shaw, “An Introduction to Software Architecture”. *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing, 1993.

[7] R. Griswold, M. Griswold, *The Icon Programming Language*, Prentice-Hall, 1983.

[8] Imagix Corporation, http://www.imagix.com

[9] R. Kazman, G. Abowd, L. Bass, M. Webb, “SAAM: A Method for Analyzing the Properties of Software Architectures," in *Proceedings of the 16th International Conference on Software Engineering*, (Sorrento, Italy), May 1994, pp. 81-90.

[10] R. Kazman, G. Abowd, L. Bass, P. Clements, “Scenario-Based Analysis of Software Architecture”, *IEEE Software*, Nov. 1996, pp. 47-55.

[11] R. Kazman, J. Carrière, “An Adaptable Software Architecture for Rapidly Creating Information Visualizations”, *Proceedings of Graphics Interface ‘96*, (Toronto, ON), May 1996, pp. 17-27.

[12] R. Kazman, M. Burth, “Assessing Architectural Complexity”, http://www.cgl.uwaterloo.ca/assessing.ps.

[13] M. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzales Harbour, *A Practitioner’s Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic, 1993.

[14] G. Murphy, D. Notkin, K. Sullivan, “Software Reflexion Models: Bridging the Gap between Source and High-Level Models”, *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (Washington, D.C.), October 1995.

[15] G. Murphy, D. Notkin, E. Lan, “An Empirical Study of Static Call Graph Extractors”, *Proceedings of ICSE 18*, (Berlin, Germany), March 1996, pp. 90-99.

[16] G. Murphy, D. Notkin, “Lightweight Lexical Source Model Extraction“, *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, July 1996, pp. 262-292.

[17] Reasoning Inc., http://www.reasoning.com

[18] M.Shaw, R.DeLine, G.Zelesnik, “Abstractions and Implementations for Architectural Connections”, *Proceedings of the Third International Conference on Configurable Distributed Systems*, (Annapolis, MD), May 1996.

[19] C. Smith, L. Williams, “Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives”, *IEEE Transactions on Software Engineering*, 19(7), July 1993, pp. 720-741.

[20] Software Emancipation, http://www.setech.com

[21] M. Stonebraker, L. Rowe, M. Hirohama, “The Implementation of POSTGRES”, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 125-141.

[22] UIMS Tool Developers Workshop, “A Metamodel for the Runtime Architecture of an Interactive System”, *SIGCHI Bulletin*, 24(1), January 1992, pp. 32-37.

[23] L. Wall, R. Schwartz, *Programming perl*, O’Reilly & Associates, 1991.

[24] K. Wong, S. Tilley, H. Müller, M. Storey. “Programmable Reverse Engineering”*, International Journal of Software Engineering and Knowledge Engineering*, Vol. 4, No. 4, pp. 501-520, December 1994.

[25] A. Yeh, D. Harris, M. Chase, “Manipulating Recovered Software Architecture Views”, *Proceedings of ICSE 19*, (Boston, MA), May 1997, pp. 184-194.