Design Recovery for Maintenance and Reuse

Ted J. Biggerstaff

Microelectronics and Computer Technology Corporation

Software maintenance and harvesting reusable components from software both require that an analyst reconstruct the software's design. Unfortunately, source code does not contain much of the original design information, which must be reconstructed from only the barest of clues. Thus, additional information sources, both human and automated, are required. Further, because the scale of the software is often large (hundreds of thousands of lines of code or more), the analyst also needs some automated support for the understanding process.

Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains. (I use the term "abstraction" in its general sense and specifically not in the abstract-data-type sense. Thus, the abstractions I discuss are generalized structures that contain fewer details than found in the source code. Any reference to ADTs will be explicit.)

The recovered design abstractions must include conventional software engineering representations such as formal specifications, module breakdowns, data abstractions, dataflows, and program description language. In addition, they must include informal linguistic knowledge about problem domains, application idioms, and the world in general. In short, design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, design recovery deals with a far wider range of information than found in conventional software engineering representations or code.

The Desire system helps software engineers understand programs by analyzing code, relying on the analyst's own reasoning, and drawing on a knowledge base of design expectations.

Design recovery occurs across a spectrum of activities from software development to maintenance. The developer of new software spends a great deal of time trying to understand the structure of similar systems and systems components. The software maintainer spends much of his or her time studying a system's structure to understand the nature and effect of a requested change. In each case, the analyst is involved in design recovery. Thus, design recovery is a common, sometimes hidden part of many activities scattered throughout the software life cycle.

A system expert provides one of the most effective ways to recover the design of a foreign system by answering questions, shifting attention quickly to germane areas of the program, interpreting code segments in human (informal) terms, and so forth. An automated system would need access to the same kind of "in-head" expertise. That is, it would need a knowledge base — a *domain model* — that cap-

0018-9162/89/0700-0036$01.00 ©1989 IEEE

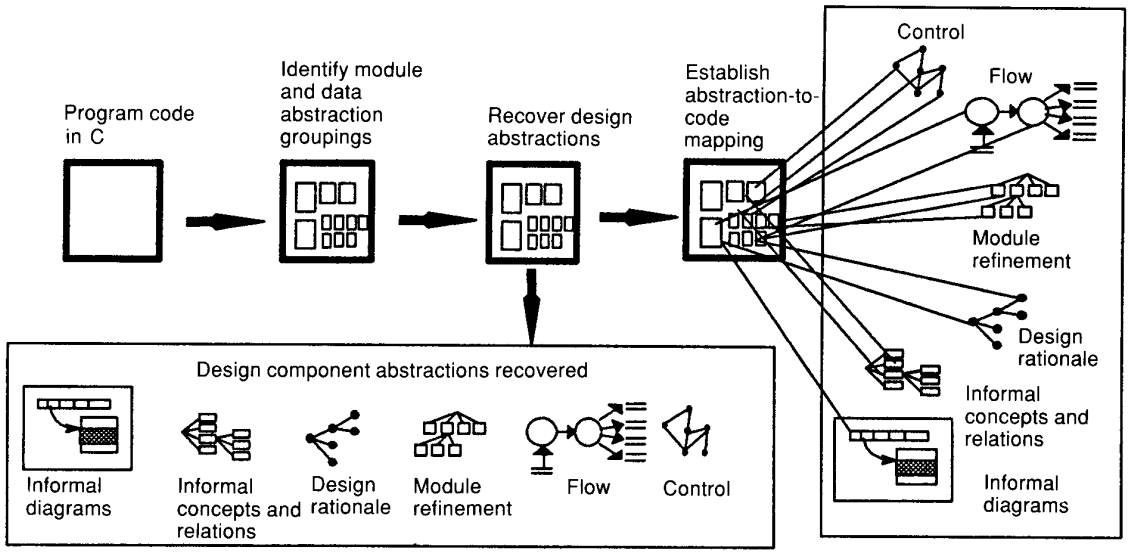


Figure 1. The basic design recovery process.

tures this expertise. The information must be domain oriented, must include more information than the analyst might find in the code alone, and must guide and assist the process of understanding the code. The domain model differentiates design recovery research from such superficially similar efforts as reverse engineering, which automatically abstracts code to a specification level such that the specifications can be modified and revised code can be automatically regenerated. In fact, the domain model is central to the overall success of any attempt to automate portions of the design recovery process.

Design recovery in the broad sense is so inherently unstructured and unpredictable that few tools have been available to help the analyst search through code to find patterns and structures of interest. Exceptions include simple search tools like grep (a pattern searching tool for Unix) and some code analysis facilities in tools like Cscope (an interactive cross-reference tool, also for Unix). Further, there have been few tools to help the software engineer capture, organize, and present the design information once recovered, other than text editors, outliners, and computer-aided software engineering tools.

To show how we might extend the automated assistance available to the software engineer, this article introduces the concept of design recovery, proposes an architecture to implement the concept, illustrates how the architecture operates, describes the progress toward implementing it, and compares this work with other similar work such as reverse engineering and program understanding.

The design recovery process

A key objective of design recovery is to develop structures that will help the software engineer understand a program or system. Understanding is critical to many activities — maintenance, enhancement, reuse, the design of a similar new system, and training, to name a few. This section describes the process of design recovery as it is applied to maintenance and to the population of reuse and recovery libraries. I then outline how a recovery knowledge base (the domain model) can assist in some of the steps of design recovery.

The design recovery process consists of three steps:

Step one: supporting program understanding for maintenance. Figure 1 illustrates the steps of the design recovery process that help a software engineer understand a C program. Other classes of languages, such as object-oriented languages, require a modest variation of these ideas.

The analyst first looks for large-scale organizational structures such as the subsystem structure, module structure, and important data structures. Next, he or she recovers various useful design structures and expresses them in abstracted forms, such as informal diagrams, informal concepts and relations, design rationale, module structures, flow, and control. In the course of this, the software engineer keeps track of the relationship (the mapping) between the various abstractions and the segments of code that implement them. Now, let us look at the kinds of questions a software engineer asks when trying to understand a system.

What are the modules? Some programming languages formalize the notion of a module and provide constructs to define it, so the module and subsystem structures are easy to determine directly from the source code. For those languages that do not provide constructs, the software engineer must use a combination of human intuition and experience, clues from the source code structures, and some knowledge (expectations) of the conventional organization patterns for applications of the type under consideration.

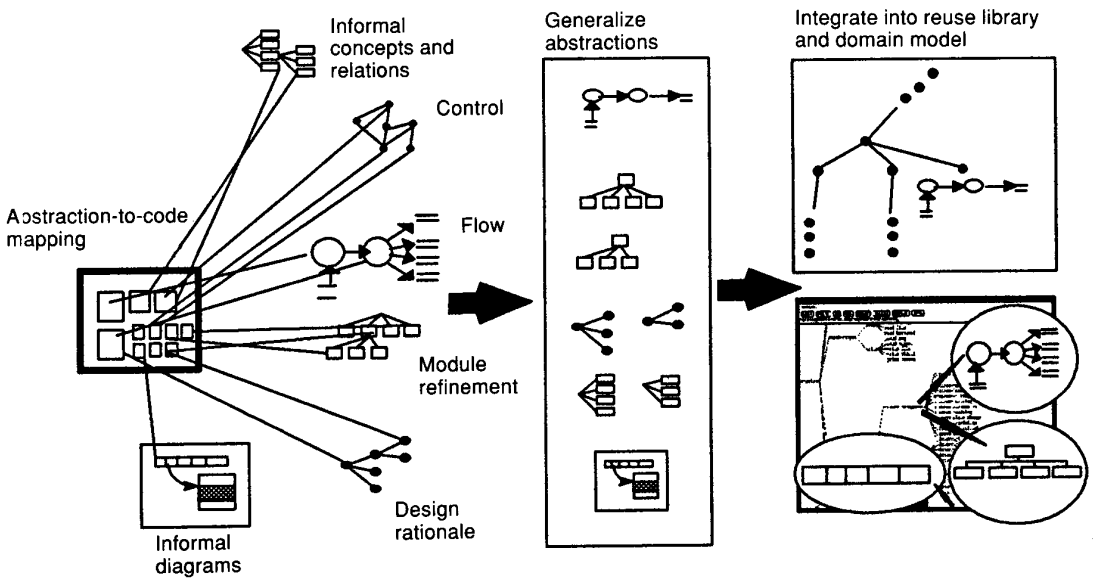


Figure 2. Design recovery extensions supporting reuse library population.

Expectations derived from organizational conventions are powerful and efficient mechanisms for helping the software engineer understand a system. For example, based on their knowledge of typical organizational patterns, experts in the domain of Unix-like multitasking code would expect to find a module that does process management and contains routines for the creation, suspension, and deletion of processes. Of course, such expectations are typically generalizations and, therefore, are only approximations of such multitasking code. Thus, our expectations, drawn from various domains, provide fuzzy patterns to guide our search and analysis of foreign code. But, because of their fuzziness, these patterns can do no more than serve as guides.

In addition to identifying large-scale structures such as modules, we also need to associate the structures with informal semantic concepts. That is, we need to provide semantically rich natural-language abstractions, or *conceptual abstractions*, that represent the essential concept underlying the module. For example, process management would be a good conceptual abstraction to associate with the example module discussed above because the phrase will help the software engineer understand the target system by referencing his or her existing mental concept and activating a variety of important and powerful expectations.

I will formalize these conceptual abstractions to the point that some measure of intelligent computer processing can be implemented on them. I am not suggesting fully automating the design recovery process; the degree of automation is unlikely to ever go beyond the notion of an assistant that can perform wide-ranging searches and suggest domain-based recovery strategies to the software engineer. However, even these limited capabilities would be quite valuable to an analyst faced with hundreds of thousands of lines of foreign code.

What are the key data items? Among the other first questions an analyst asks are

- What are the important data items?
- What abstract informal concepts do they relate to?
- What are their relations to the modules just identified?

For example, in the multitasking window system example, the analyst might find a process table containing entries that describe the processes currently running under the multitasker. The more experience the analyst has with multitasking systems, the richer the set of expectations that he or she will have about such a system.

What are the software engineering artifacts? As shown in Figure 1, the understanding process recreates the software engineering-oriented design artifacts and expresses them wherever possible in terms of the module and data abstractions recovered earlier. The specific artifacts captured are determined to some extent by the process model adopted by the programming organization. For example, some companies will use a program description language, dataflow, module refinement, and a simple data dictionary. Others will depend on different design artifact sets. The techniques under investigation at MCC are flexible enough to apply to a broad range of such artifacts.

What are the other informal design abstractions? For the set of abstractions to be really effective, we need other information structures, many of which are not as well defined and formal as the software engineering-oriented design artifacts. For example, design rationale might be useful, perhaps stated in terms of issue-based information systems (IBIS) nets.[1] Further,

natural-language prose is unavoidable if we want a really effective model of the design. Similarly, informal diagrams describing abstract views of the target system are often quite useful. Thus, we must expect to recover a wide variety of design artifacts that contain a mixture of formal and informal information.

What is the relation of the design abstractions to the code? After recovering the artifacts, we must preserve the relationships among them. That is, once we determine that a context switch is being performed within some dataflow diagram, we would like to know exactly which chunk of code performs it. Code analysis of a concrete example is often required to answer questions that depend on low-level details abstracted out of the dataflow diagram. Once an engineer establishes this abstraction-to-code link, he or she will have an organized, "in-head" framework (the abstraction) in which to put the code-oriented details and, perhaps more importantly, a set of organized structures to help interpret those details. Thus, the engineer can understand the code in terms of the abstractions in the framework.

Step two: supporting population of reuse and recovery libraries. How might we productively use the recovered design components? Populating the component library of a reuse system is an obvious and valuable use, but that requires further steps to generalize the components to enhance their reusability. Figure 2 illustrates this process. Generalization makes the components applicable to a wider spectrum of applications, but it can require that we factor them to decouple independent design aspects. For example, an independent process-management component might apply far more widely than one that is tightly coupled to window management.

The final step in this process integrates the new abstractions into the reuse library and the recovery knowledge base (the domain model). Thus, we expect to reuse this recovered information to help build similar new components and to recover similar components from other systems.

Step three: applying the results of design recovery. The final step of the process cycle applies the newly populated domain model to design recovery (see Figure 3). The abstract design components stored in the domain model now become the starting point for discovering candidate concrete realizations of themselves in a new system's code. Once the software engineer determines that the candidate is truly a concrete realization of the abstract design component, the design recovery system records the finding. For example, domain model information about the expected kinds of functions in the process management example might provide a skeleton for that module and even provide some semantic clues about the names of the various routines in the module.

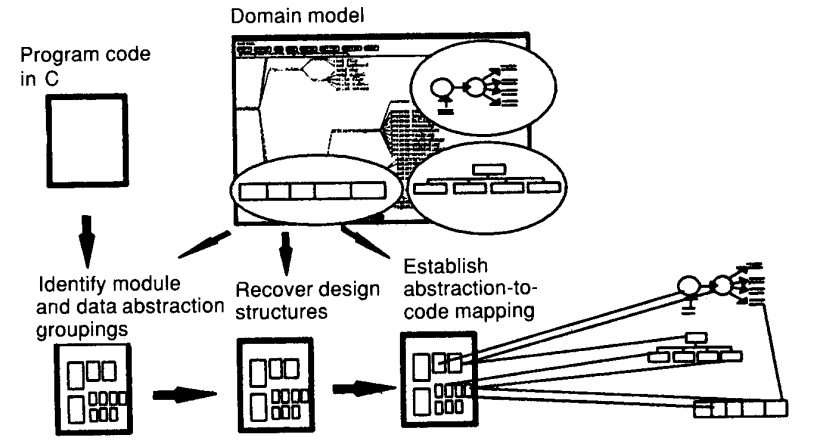


Figure 3. Model-aided design recovery process.

Of course, the expectations in the domain model will seldom be an exact match of the design structures in the source code, and the software engineer will likely have to edit the design abstraction to synchronize it with the code, but even a partial match reduces the overall work. Further, each significant mismatch provides new expectations that help the domain model grow and evolve.

Distinguishing properties of design recovery

Two key properties distinguish this design recovery model from similar models:

(1) *Use of informal information.* The model exploits multiple kinds of information. Importantly, it uses informal information, which exists outside of the sphere of programming languages and opens a new kind of leverage on the recovery problem — one that exploits a human-oriented, associative style of retrieval and analysis.

(2) *Use of a domain model.* This design recovery model also exploits multiple sources of information. In particular, it uses a domain model to help the software engineer understand and interpret foreign systems. The domain model is a knowledge base of expectations expressed as patterns of program structures, problem domain structures, language structures, naming conventions, and so forth, which provide frameworks for the interpretation of the code. These frameworks can be built on to recreate the design information that is missing from the code as written. Heretofore, such expertise has existed only in the minds of expert software engineers or application domain specialists.

Conceptual abstractions: the use of informal information. Among the information developed by the design recovery process are instances of conceptual abstractions that help the user understand the nature of a design in human terms. That is, the conceptual abstraction instances produced by design recovery must go beyond what can be represented in programming languages. They represent the world not only in rigid formal terms, but also in informal and flexible terms. Such artifacts are not simply optional, informal additions to the formalisms expressed in the programming language, but complementary representations that are necessary and critical to the mental structuring and assimilation of the final design by a software engineer.

Note that I distinguish between the no-

tion of a conceptual abstraction ("a process management module") and a specific instance of a conceptual abstraction ("the specific process management module in the Unix system"). This distinction is important because each has a distinct role. Conceptual abstractions are implemented in the domain model as object-oriented classes that take an active role in identifying instances of themselves in the code being interpreted. Thus, they represent the set of realizations of that object type in the target code, whereas an instance represents a single, specific realization of that object in the code. The sidebar, "Concepts of object-oriented programming," further clarifies the distinction between class and instance.

If a recovered design contains this additional kind of entity — the conceptual abstraction instance — how do we identify it? And, what is the character of such an entity?

An instance of a conceptual abstraction has two important properties, one that is structural and one that is semantic or associative. The associative part of the abstraction is represented in the domain model by a "linguistic idiom." The structural part is represented by various kinds of idioms, depending on the kind of information being represented. Introducing associative connections and structural patterns provides a partial formalization for informal conceptual abstractions.

Structural pattern. A conceptual abstraction's first property is its ability to represent (that is, both hide and relate) some set of lower-level details. For example, a single concept such as "a process management module" can be used in many contexts to represent all of the massive detail that is a process management module, keeping the designer from becoming overwhelmed by the detail. This property is similar to the conventional software-engineering notion of expressing designs as top-down refinement structures. Its function is to describe the successively burgeoning levels of detail in a design. A conceptual abstraction's structure has an additional, operational role as a pattern that defines the kinds of source code structures that would express the abstraction. This pattern is used to search for and identify specific source code structures that are

Concepts of object-oriented programming

The domain model in the Desire design recovery system is strongly related to the concepts of object-oriented programming systems (OOPS), such as Smalltalk,[1] C++,[2] and the Common Lisp Object System (CLOS). Central to OOPS is the concept of a *class*, which is a package of local data items that defines the state of an instance of the class, and a set of functions that manage that state. An *instance* of a class (also called an *object*) is a unique copy of the local data items; to put it another way, it is a specific concrete member of the class. Each data item is called an *instance variable*. The functions of the class are conventionally called *methods*.

An example of a class would be line_segment, which might have instance variables *x*, *y*, and *length* that define the position of the line segment's end point and its length. There might be many specific lines in a drawing, and each would be represented by an instance of the class, that is, a data record containing three values for *x*, *y*, and *length*. The methods of such a class might be named create, destroy, move, rotate, stretch, draw, and so forth. These methods would operate on the instance variables to perform various operations on the line.

To call such a method, we would send a message to an instance of the class. Sending a message is a generalization of the notion of a function call, and it requires at least two pieces of information to perform the invocation: a pointer to an instance (from which the system can determine which class to look in for the method definition) and the name of a method (such as move). The method name is called the *selector*. These two pieces of information uniquely determine the specific method to be called. Some object systems, such as CLOS, provide an optional, special case where additional items can be required, allowing a finer-grained determination of the specific method to be called.

A key concept in OOPS is *inheritance*, which allows us to specify a new class by defining only the differences between it and another class, called its *superclass*. For example, we could specify a class fat_line_segment by declaring it as a *subclass* of line_segment and describing the differences. We would say line_segment is the superclass of fat_line_segment. Suppose this new class has an additional instance variable named *width*, which defines the width of the line to be drawn. Its instance records will contain variables *x*, *y*, and *length*, inherited from line-segment, and the variable *width*, from fat_line_segment's definition. Similarly, we would write a new version of the draw and create methods to accommodate the operational differences between simple line segments and those with width. These new methods would be called whenever the draw or create messages were sent to one of fat_line_segment's instances. When other messages, such as stretch, are sent, the inherited methods from line_segment would be called.

Frames, a slight variation of the concept of classes, come from the field of artificial intelligence. They usually have more built-in conventions for the instance variables (commonly called *slots* in frame systems) than simple OOPS classes do. They therefore have more associated runtime support. Frames systems often include conventions and runtime support for expressing relationships between instance records. For example, semantic net applications often provide frame conventions and built-in facilities that search the frame network for sets of instances that resemble but do not exactly match each other. Such frame conventions and support are often built on top of a conventional OOPS system.

References

1. Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.
2. Bjarne Stroustrup, "What is Object-Oriented Programming?" *IEEE Software*, May 1988, pp.10-20.

plausible instances of the conceptual abstraction.

Associative connections. A conceptual abstraction's second property is its rich set of informal, natural-language associations that establish its contextual framework for human understanding. That is, the concept of a process management module has semantic connections to other informal, semantic concepts such as context switching, state saving, and multitasking. Each of these concepts allows association of the concept of a process management module with a large body of knowledge that can help an engineer interpret the design of some specific process management module or plan the design of a new one.

These two properties provide clues to the role of conceptual abstractions in dealing with large complex designs. The structural property provides a way to handle lots of detail without being overwhelmed, as well as a way of describing the application patterns one expects to find in programs. In contrast, the associative linguistic property offers a way to deal with partially specified (fuzzy) design objects within the universe of informal, natural language-based semantics. These properties relate to two parallel and complementary models — the software-engineering representation model and the natural-language semantic model.

The importance of informal information. An example will illustrate the importance of the informal aspect of conceptual abstractions. Consider the C function in Figure 4. This is a real function taken from a multitasking window system[2] with the comments removed and meaningful identifiers mapped to semantically empty symbols. What could an analyst tell about the computational intent of this function? Precious little. About all he or she could do is paraphrase the relations expressed in the programming language. For example, the analyst could describe that the function f0001 calls f0002 with arguments that are global arrays (such as g0001) of structures containing some fields (such as s0001 and s0002). Even if the definitions of all of the functions (f0002, f0003, etc.) were available and similarly transformed, the computational intent would remain unclear. What is worse is that, without the informal information, the computational intent of these functions might not be unique. There could be a number of valid interpretations.

The example severs the connection between the artifact and the semantics of the problem domain, eliminating associations between the program and our informal knowledge of the world. Interpretation and understanding of the program has become impossible in any deep sense. Thus, we can see that connotation plays an important role in the process by which people deal with, interpret, and understand programs.

It is exactly this kind of semantically impoverished representation that we usually give to automated tools. If people have difficulty dealing with this kind of representation, why should we expect a computer to be more successful?

So what sort of informal information is required to understand the program in a nonsuperficial way? Let us consider a slightly enhanced version of this program. Figure 5 maps the symbolic names back to those used in the original code. Here, the names of the functions are more meaningful and, if the reader understands a bit about multitasking and window systems, he or she can probably make some good

```
#include <stdio.h>
#include "h0001.h"
#include "h0002.h"
#include "h0003.h"
f0001(a0001)
        unsigned int a0001;
        {
        unsigned int i0001;
        f0002(g0005,d0001,d0002);
        f0002(a0001,d0003,d0002);
        f0003(g0001[a0001].s0001,g0001[a0001].s0002);
        g0006 = a0001;
        i0001 = g0001[a0001].s0003;
        if(!f0004(i0001) && (g0002->g0003)[i0001].s0004 == d0004)
                f0005(i0001);
        }
```

Figure 4. Function with no informal semantic clues.

```
#include <stdio.h>
#include "proc.h"
#include "window.h"
#include "globdefs.h"
change_window(nw)
        unsigned int nw;
        {
        unsigned int pn;
        border_attribute(cwin,NORM_ATTR,INV_ATTR);
        border_attribute(nw,NORMHLIT_ATTR,INV_ATTR);
        move_cursor(wintbl[nw].crow,wintbl[nw].ccol);
        cwin = nw;
        pn = wintbl[nw].pnumb;
        if(!outrange(pn) && (g->proctbl)[pn].procstate == SUSPENDED)
                resume(pn);
        }
```

Figure 5. Function with some informal semantic clues.

```
#include <stdio.h>
#include "proc.h"
#include "window.h"
#include "globdefs.h"
change_window(nw)              /*Change current window to window nw*/
        unsigned int nw;       /*Number of target window*/
        {
        unsigned int pn;

        /*Restore border of current window to un-highlighted*/
        border_attribute(cwin,NORM_ATTR,INV_ATTR);

        /*Highlight border of new current window*/
        border_attribute(nw,NORMHLIT_ATTR,INV_ATTR);

        /*Move the physical cursor to the new window where the cursor was
        left, and make nw the current window*/
        move_cursor(wintbl[nw].crow,wintbl[nw].ccol);
        cwin = nw;

        /*Resume the process associated with the new window if it is
        suspended.*/
        pn = wintbl[nw].pnumb;
        if(!outrange(pn) && (g->proctbl)[pn].procstate == SUSPENDED)
                resume(pn);
        }
```

Figure 6. Function with many informal semantic clues.

guesses about the operation. The name of the function suggests that it changes which window is currently active, with the new window probably indicated by the argument nw. Further, we can guess that the function border_attribute alters the visual appearance of the windows' borders, the function move_cursor moves the screen cursor to some position in the new window, and the function resume allows some suspended process to run again (probably the process associated with the new window). The variables similarly come alive with meaning: wintbl is probably the window table and probably has fields ccol and crow that keep track of the cursor (inferred from their use in the call to move_cursor).

By restoring the comments from the original code (see Figure 6), we can corroborate several of our guesses and enhance our understanding of some of the functions and variables.

This exercise should make it clear that the informal linguistic information that the software engineer deals with is not simply supplemental information that can be ignored because automated tools do not use it. Rather, this information is fundamental. It provides the ability to determine the computational intent of code in a way that is impossible with just the source code denuded of its informal semantics.

If we are to use this informal information in design recovery tools, we must propose a form for it, suggest how that form relates to the formal information captured in program source code or in formal specifications, and propose a set of operations on these structures that implements the design recovery process. To accomplish these goals, we must first analyze the proposed design recovery system in a bit more detail.

A model-based design recovery system

What would a design recovery system look like? Figure 7 is a system-level description of a model-based design recovery system (called Desire) showing some of the sources of information used to recover designs. They include the code of existing systems because such code contains a large amount of important information, but there must be other sources as well. Much design information cannot be formally captured in the program source code because programming languages do not contain the constructs necessary to express information such as the informal conceptual abstractions behind the code. For example, the informal conceptual abstractions behind the change_window function discussed earlier include windows, processes, cursors, and the operations on these entities. And these conceptual abstractions are woven into a rich set of knowledge about the domain that provides clues to understanding the formal source code structures.

Design recovery results in a hypertext web[4] of information that weaves together informal ideas (e.g., the concept of a process), software engineering artifacts (e.g., a dataflow diagram of a process switch), and details of specific examples of these entities as embodied in code (e.g., one specific subroutine for process switching). This web is projected into externalized reports to help the software engineer understand a specific target system and into internalized data structures for use by the Rose reuse system.[3] Since the web is built out of hypertext frames, the design recovered by the Desire system is simply a set of data structures that represent the conceptual abstractions and express the semiformal relationships among them (see sidebar, "Concepts of object-oriented programming").

To understand how this model-based design recovery system works, the nature of the data items in the model, and how those data items are used, consider the following typical design recovery session using the multitasking window system as the application domain. Using the process model of design recovery as a guide, I will first define a set of data objects (called *idioms*) that implement the structural patterns and associative connections of conceptual abstractions. The example idioms codify the domain model's expectations of the entities and structures in a typical multitasking window system running on a personal computer. I will then informally describe how these domain objects behave during the semiautomated recovery of the design of a specific multitasking window system. This scenario is analogous to a nonautomated design recovery performed by an unaided software engineer.

An example. We start to discover the structure of this multitasking window sys-

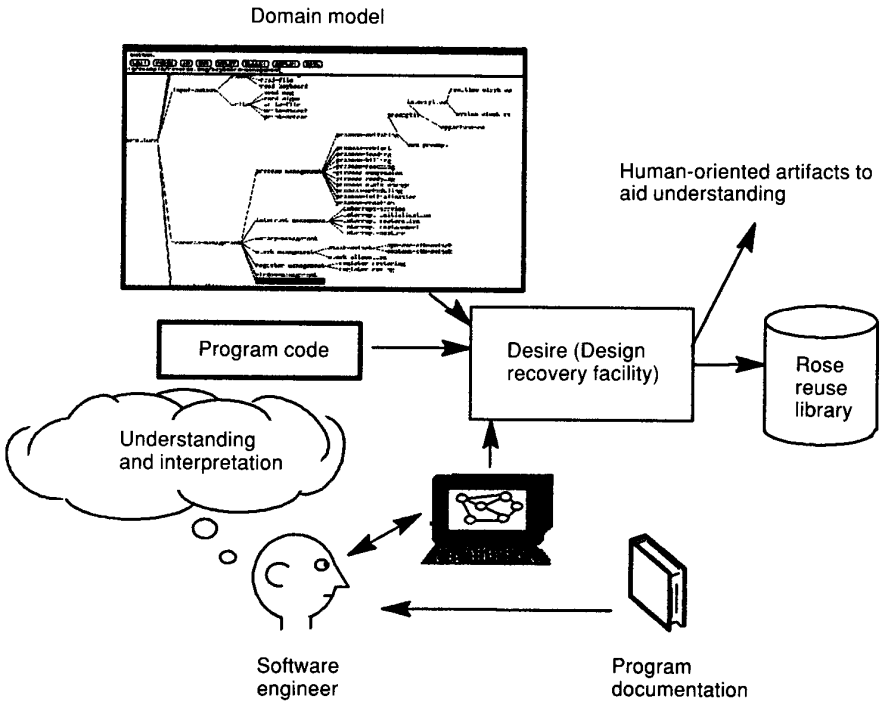


Figure 7. The Desire model-based design recovery system.

tem by looking for key structures, based on our knowledge or expectations of the problem and application domains. A knowledgeable engineer would expect to find a process table, a window table, a window management module, and a process management module, among other structures. (I offer a detailed example of such a system elsewhere.[2])

In Desire's domain model, such expectations are represented by object classes expressed in the Common Lisp Object System (CLOS). In contrast to object classes that *implement* a window management module or a process management module, these domain model classes *operate* on the implementations of window management modules or process management modules. Specifically, domain model objects search for instances of the key structures within the code (perhaps with human help) and bind their instance variables to these key structures, subject to the analyst's approval. An instance of such a domain object represents an occurrence of a concept such as a window management module or a process management module within a specific segment of source code. The instance variables of that instance point to the segments of code that implement the domain object.

Thus, the first step of recovery is to create a set of instances of the idiomatic structures expected. The engineer examines the domain model, finds an object class describing a multitasking window manager, and creates an instance of that object. As a side-effect, other instances that define the detailed substructure of a multitasking window manager are created as a substructure of this first instance. This structure of instance records represents an architectural overview of a multitasking window manager that might look abstractly like the structure in Figure 8, where the relation on the arcs is the subparts relation. Over the course of the design recovery process, the whole set of design details will evolve as a rich substructure beneath this first set of instances. Now let's follow the evolution of that substructure in more detail.

Each of the instances just created can bind to the source code in one of two ways:

(1) It can bind directly to some segment of code (associatively).

(2) It can bind indirectly through a subinstance (that is, through a close match of the substructures to the program code).

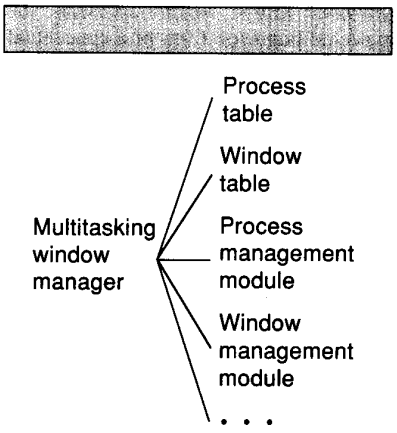


Figure 8. Initial pattern instance records expressing an architectural overview.

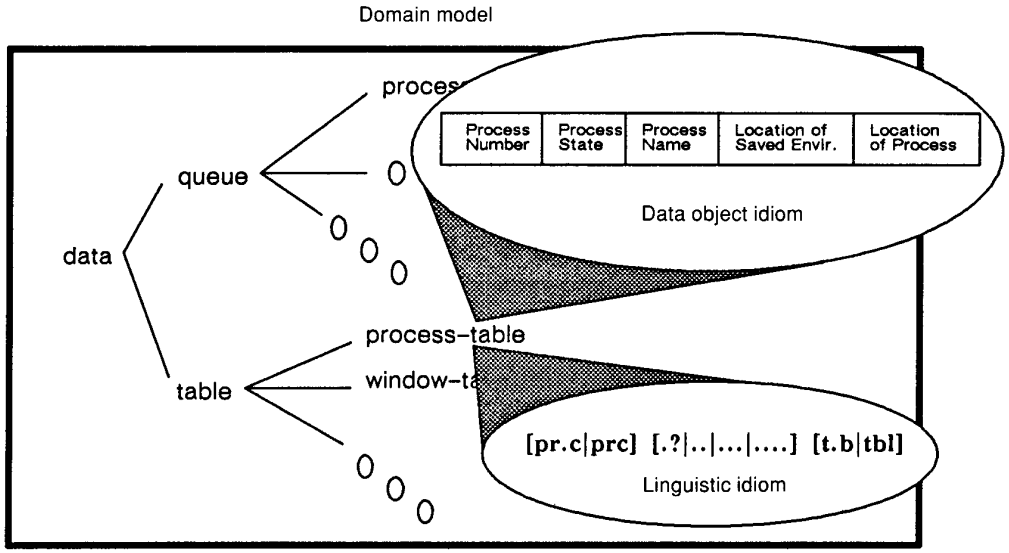


Figure 9. Abstract design idioms within a domain model.

For example, the process table class in Figure 9 contains idioms for both kinds of binding.

Direct binding is implemented via a *linguistic idiom*, which represents the expected linguistic form of a conceptual abstraction such as process table. This idiom might be implemented as a set of regular expression patterns that match the various natural-language forms in source code identifiers or comments. For example, the pattern [pr.c|prc] [.?|..|...|....] [t.b|tbl] defines the linguistic expectations for process table. Of course, we will occasionally encounter an expression of the conceptual abstraction that the existing patterns do not find. The addition of such cases helps the domain model grow and evolve.

How would the recovery system use these patterns? We would seldom want to recklessly search a large system to find all of the associations. Not only would such a search take a large amount of computation time, it also would probably introduce a large number of false positive hits, thereby taking a lot of analyst time to sort out the results. Instead, we would prefer to be more selective and use our knowledge of programs, systems, and domains to focus the search. For example, in the C language we would expect to find the definition of the process table in some header file and so would narrow our initial search to those files. This search might find the chunk of code in Figure 10.

Given this structure as a starting point, the system uses the data object idiom that defines the substructure of a process table and recursively applies the search against the code structure in Figure 10, using the patterns from the classes of each of the subparts of a process table. (Actually, I have simplified this example by ignoring the intervening conceptual structure — the process table entry.) Figure 11 illustrates this recursive step, which binds the fields within the source code definition of the

```
process proctbl[MAXPROCS];          /* Process tbl array */
 . . . .
 . . . .
typedef struct procentry            /* Process table entry */
  {
  unsigned int savesp;              /* Saved sp register */
  unsigned int savess;              /* Saved ss register */
  unsigned int pspseg;              /* PSP seg addr this proc */
  unsigned int windno;              /* Window number this proc */
  unsigned int procstate;           /* Process state */
  char procname[MAXPNAME+1];        /* Process name */
  int pnum;                         /* Process number for this entry */
  . . . .

  } process;
```

Figure 10. Structure found via search of source code.

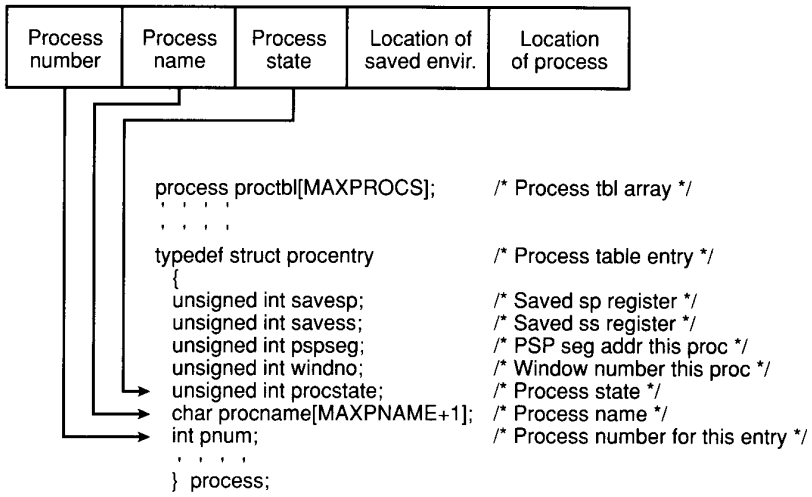


Figure 11. Bindings to substructures in source code.

process table to the instance records that define the expectations of those fields.

The recovered design in Figure 8 has now evolved into that in Figure 12, where the dashed lines show the bindings between abstractions and code. Note that the matching of idiom pattern to code is inexact, with some instances of the idiom unbound and some elements of the struct unexplained. We can typically expect an automated aide to produce only partial matches, leaving part of the interpretational work to the software engineer. Thus, the analyst will likely have to specialize (edit) the idiom further to reflect the specific case. However, the partial match provides enormous benefit by focusing the analysis and establishing a broad framework in which to perform the remaining interpretation. The partial match provides a starting point by identifying some of the substructures of the idiomatic form, making completion of the interpretation far simpler than starting from scratch.

So, for data structures in the source code, at least two kinds of information express the key idiomatic features of the source code. The linguistic idiom expresses the natural-language tokens (generalized into search patterns) that we expect to be associated with key data structures. The data object idioms express the substructure relationship within complex data structures. We can exploit structural idioms to discover design structures for the first time (in a top-down manner) as well as to verify associations with large-scale structures discovered earlier (as a bottom-up verification).

This pattern of linguistic and structural design idioms repeats itself when we examine what kinds of structures the domain model must contain to describe the expectations about module structure, dataflow plans, and so forth. For example, in the context of a Unix-like multitasking system, we would expect to find a process management module with routines for

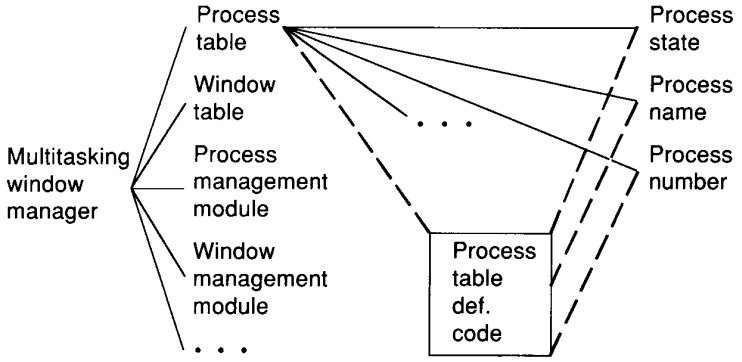


Figure 12. Pattern of instance records with some bindings to code.

creating a process, resuming a process, suspending a process, and so forth. Much as in the data structure idiom, these expectations form an initial framework that might fuzzily match some functions in the code. From here, the software engineer can read and analyze the code and then specialize the domain model patterns to fit the current case. Other artifacts such as dataflow schemas operate in a similar fashion.

Using idioms to guide the search and then act as the skeletal organizing structure for the recovered design offers two important benefits:

(1) The domain model idioms encode expectations as preformed queries, eliminating the need to constantly reenter these query forms during program analysis.

(2) The system records the resulting design in terms of both informal linguistic abstractions and semiformal software engineering structures.

Desire prototype and current work

MCC has developed a prototype of a design recovery system called Desire Version 1.0. The system is intended to explore only that aspect of design recovery that does not depend on the domain model. Thus, it is an interim system designed to lay the foundation for the full Desire system by providing a baseline of facilities to process the information explicitly found in source code.

Figure 13 shows Desire Version 1.0 in operation. The system consists of three major parts: a parser, a set of postprocessing functions, and the PlaneText hypertext system[4] as the presentation engine. The parser processes a set of C files for a given system and produces a set of parse trees. We anticipate the need to use much of the informal linguistic information encoded in variable names, comments, and the like, so the parser must take special care to preserve this information.

A set of postprocessors takes the parse trees as input and produces a dictionary containing information on functions, the

Related work

Commercial reverse engineering tools

A number of reverse engineering tools (closely related to reengineering tools) have appeared on the market recently. These tools solve part of the problem of recovering the design of an existing system. Examples are Cscope, cxref, Bachman/Data Analyst, and Meta's Design/2.0. While these tools find, present, and analyze information in the source code (or in the data dictionary in the case of the Bachman/Data Analyst), they do not reconstruct, capture, and express design abstractions that are not explicitly represented in the source code-related representations. Such design abstractions are a large part of what humans use to understand, modify, adapt, and otherwise deal with systems and programs. The need for and absence of such capabilities should indicate the direction in which these tools will likely evolve, and should be a mandate to push the technology in that direction.

We can loosely classify commercial reverse engineering tools (in the broadest sense of the term "reverse engineering") into the following categories:

- test coverage analyzers;
- debuggers and execution monitors;
- source-to-source translators;
- cross reference facilities;
- code reformatters, pretty printers, and restructurers;
- structure and metric analyzers;
- file comparators; and
- CASE-oriented reverse engineering (and reengineering) tools.

Since these tools are commercial, information about them is limited. Nevertheless, Horton[1] and Aranow[2] provide a short overview plus information on specific products and vendors. In addition, Sneed and Jandrasics[3] describe a prototypical CASE-oriented reverse engineering system.

Related research

In contrast to commercial reverse engineering tools, the tools in the research community come closer to our notion of design recovery. Both sets of tools focus most often on program understanding, but there is a subtle difference in the research goals. Most tools in the area of program understanding have focused on very small-scale problems to achieve precise and complete formal specifications of the source code. Further, they typically have not focused as much on informal information. In contrast, our approach sacrifices formal completeness and precision for scale. In the long run, these two approaches will likely be complementary rather than mutually exclusive, with each providing aspects missing in the other.

Several researchers have been working on the problem of understanding what a program is intended to do.[4-7] Some use information drawn largely from the programming language domain, such as Wills' recognizer.[7] Others incorporate more knowledge from the problem domain. In most cases, the scale of the target programs is quite small — tens or hundreds of lines of code. The most recent Programmer's Apprentice work deals with larger components but does not yet deal with large, industrial sized components.[5]

Most of this work depends on analysis of the low-level, formal details and, therefore, emphasizes a full and exact match of the structure for recognition. The computational load required by such an approach suggests that scaling up to industrial sizes will be quite difficult. People appear to be successful at program recognition because they can attend to a few key features and make tentative, plausible matches based on similarity rather than exactness. Further, most of the time, many of those few key features are informal. Of course, humans supplement such recognition with many other approaches for verification and detailed understanding, but the initial narrowing of attention based on informal, partial clues seems to be critical to handling scale without being overwhelmed by detail.

It seems likely that the understanding and recovery approaches can be productively merged. That is, an initial search strategy based on informal, partial clues, followed by

files that contain them, the global data items defined, where they are defined, and where they are used. Informal information, such as that in the comments, is associated with the target program's function definitions and data item definitions. In addition, the postprocessors compute and store the various relationships between these items (such as calls, uses, and depends) in the dictionary.

Once this information is computed, another postprocessor computes a PlaneText web and invokes the PlaneText browser to exhibit the web. PlaneText computes various views that exhibit some relationships (such as calls) and suppresses others. Thus, if the user wants to see a call lattice or the relationship between data items and files, the system can compute each of these as a separate browser view.

The screen dump in Figure 13 suggests some of the prototype's functionality. The prototype provides a set of predefined Prolog queries for computing a variety of questions about the data in the dictionary. These queries include low-level questions such as "What is the set of functions that call function x?" as well as higher-level questions such as "Does any function defined in file A call any function defined in file B?" or "Compute the set of functions that appear to be utility functions." This set of queries is evolving to include a number of complex program analysis functions. Since the user can build on these queries, the question set can be tailored to any specific application domain.

In Figure 13, the user has used one of the predefined Prolog queries to ask for all functions that refer to a piece of data named call_to_times. The browser responds by highlighting all of the functions found by the query. The user inspects the visual design browser and identifies the name of the file (pow2.c) that contains the definition of one of these functions (power). The user then opens a window on that file (lower left-hand corner) and uses PlaneText's regular expression-based search to find the location of the variable calls_to_times. Of course, as we determine which sequences in the prototype are most useful, we will replace them with a single user command.

the more detailed kind of analysis used in the Programmer's Apprentice, would provide a powerful and scaleable approach to automating more of the program comprehension process. This appears to be the most successful long-term direction for this kind of research.

Other work is more loosely related but nevertheless shares some ideas with our work. Perhaps the overriding theme of this work is the integration of CASE, hypertext (or hypermedia), and knowledge-based technologies. In fact, some of our own work that contributed to our current design-recovery notions (my large-scale reuse[8] and Lubars' Rose[9] system) falls in this class, although the hypertext aspect is largely absent in Rose. The work of Ambras and O'Day[10] and Bigelow[11] shares this theme to a greater or lesser extent, with Bigelow emphasizing hypertext and Ambras and O'Day emphasizing knowledge-based representations. Most of the work in this category is, in principle, able to handle large-scale information bases.

The work by Arango et al.[12] resembles the research reported in the current article. These researchers have solved the problem of scaling up but are not creating the kind of high-level, informal conceptual abstractions that Desire focuses on. Of course, creating such abstractions was not particularly important to Arango et al. because they wanted to port their target program (Draco,[13] in this case) completely automatically from one computing environment to another. From this point of view, their system is more similar to source-to-source translation and restructuring systems than reverse engineering systems of the variety I have discussed.

Abstractly, Arango et al. are also trying to recover designs, but some differences exist between their focus and ours. Their design recovery model focuses more on the structure of the transformations and the operations on transformations than it does on the structure of and operations on the design entities themselves. Our focus is the reverse of this. Further, and perhaps more importantly, their model makes no use of informal information because it is based on a commitment to complete automation. On the other hand, because our model strongly involves people in the design recovery process, we must make heavy use of informal information to help human understanding.

References

1. L. Horton, "Tools are an Alternative to `Playing Computer'," *Software Magazine*, Jan. 1988, pp 58-67.
2. E. Aranow, "CASE for Existing Systems: Taking Yesterday's Systems into Tomorrow," *System Builder*, Oct./Nov. 1988, pp. 20-29.
3. H.M. Sneed and G. Jandrasics, "Inverse Transformation of Software from Code to Specification," *Proc. Conf. Software Maintenance*, CS Press, Los Alamitos, Calif., Order No. 879, 1988, pp. 102-109.
4. S. Letovsky, "Cognitive Processes in Program Comprehension," *Systems and Software*, No. 7, 1987, pp. 325-339.
5. C. Rich and R. Waters, "The Programmer's Apprentice: A Research Overview," *Computer*, Vol. 21, No. 11, Nov. 1988, pp. 10-25.
6. E. Soloway and W.L. Johnson, "Proust: Knowledge-Based Program Understanding," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 3, Mar. 1985, pp. 267-275.
7. L.M. Wills, "Automated Program Recognition," Tech. Report 904, MIT AI Laboratory, Feb. 1987.
8. T.J. Biggerstaff, "Hypermedia as a Tool to Aid Large-Scale Reuse," Tech. Report, STP-202-87, MCC, 1987, also in *Workshop on Software Reuse*, Rocky Mountain Institute of Software Engineering, Boulder, Col., Oct. 1987.
9. M.D. Lubars, "Wide-Spectrum Support for Software Reusability," Tech. Report, STP-276-87, MCC, 1987, also in *Workshop on Software Reuse*, Rocky Mountain Institute of Software Engineering, Boulder, Col., Oct. 1987.
10. J. Ambras and V. O'Day, "MicroScope: A Knowledge-Based Programming Environment," *IEEE Software*, Vol. 5, No. 3, May, 1988, pp. 50-58.
11. J. Bigelow, "Hypertext and CASE," *IEEE Software*, Vol. 21, No. 3, Mar. 1988, pp. 23-27.
12. G. Arango et al., "Maintenance and Porting of Software by Design Recovery," *Proc. Conf. Software Maintenance*, CS Press, Los Alamitos, Calif., Order No. 648, 1985, pp. 42-49.
13. J.M. Neighbors, "Draco: A Method for Engineering Reusable Software Systems," in *Software Reusability*, T.J. Biggerstaff and A. Perlis, eds., Addison-Wesley, 1989.

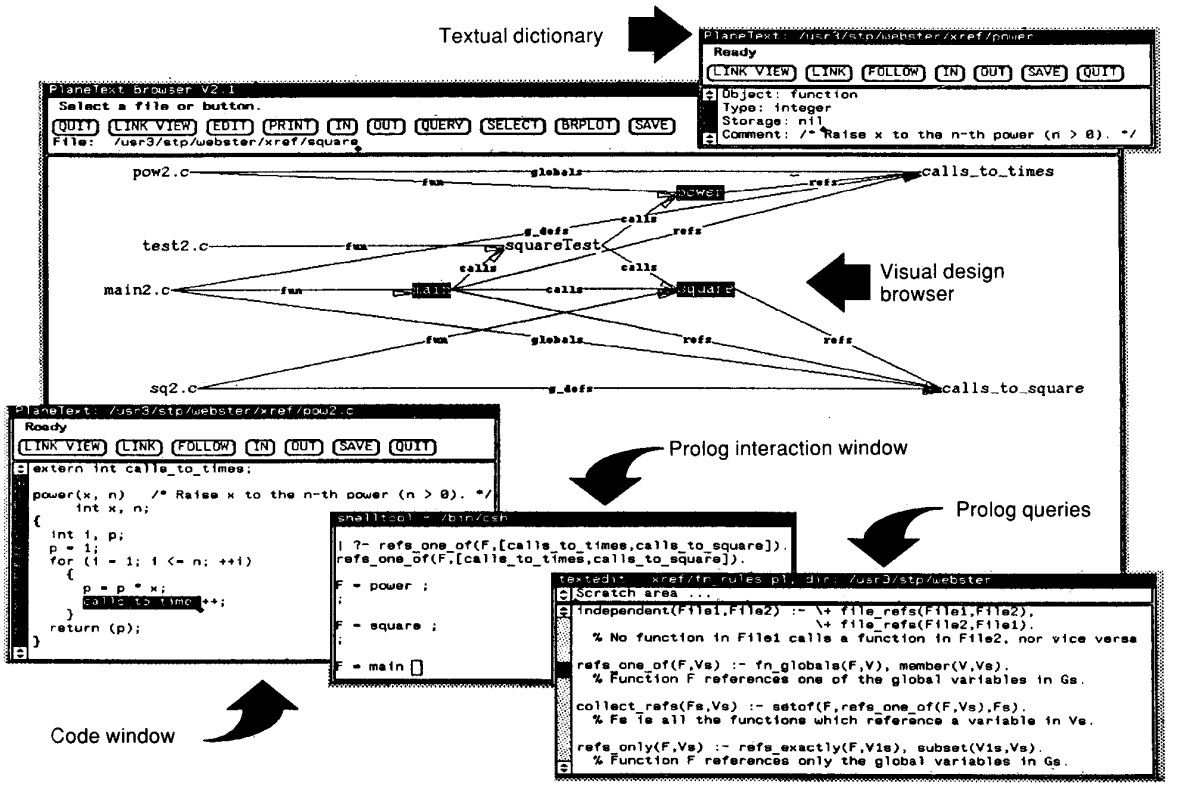


Figure 13. First design recovery prototype: Desire Version 1.0.

Desire Version 1.0 also analyzes the source code and creates a graphical diagram describing its interpretation of the program's module structure. Based on the user's request, this analysis can variously depend on the program's cohesion, its data coupling, and so forth.

As a separate research task, we have used PlaneText[4] to sketch out a domain model drawn from the area of multitasking window systems.[2] This research task uses PlaneText as a simple design aid, and we have not yet integrated the domain model with the Desire Version 1.0 prototype. To do so, we are converting this hypertext design of the domain model into a set of CLOS classes that are the active or implementation form of the domain model. This model bears a strong relationship to the semantic models created with frame languages like KL-One. The primary difference is that the CLOS classes possess more specialized, local behavior. The classes capture

- the *isa* or superclass/subclass lattice of the entities in the domain (e.g., a process-table is a subclass of table);
- the informal patterns for expressing the entities in the domain (e.g., a process might have a variety of abbreviations and synonyms);
- the slots that are to contain the expected substructure of a concept instance (e.g., a queue will have a queue-entry slot);
- restrictions on the slots that constrain what values the slot can have (e.g., a restriction on the class of the value in the slot); and
- methods that provide the entity's behavior (e.g., displaying the concept, searching for possible instances, and binding slots to instances).

A fair amount of work is now aimed at determining what the interface should look like and how it should behave. Specifically, we are working on methods to visually relate the recovered abstract concepts to the portions of the program to which they refer.

More recently, we have begun a series of experiments to refine our notions of search strategies for informal information and to allow fuzzy matches of expectations. We believe that, in addition to the simple search mechanisms described earlier, we will be able to apply ideas from connectionist research[5] to perform some of the fuzzy matches. These searches take advantage of the domain model's structure to allow associative searches that return items based on their indirect associations with the features sought. This part of the research is in an early prototyping stage. □

Acknowledgments

I gratefully acknowledge the work of Keith Andren, Gerry Barksdale, Glenn Bruns, Josiah Hoskins, Peter Marks, Bharat Mitbander, Don Petersen, Dallas Webster, and Mahesh Zarule, who have cast portions of these ideas into working prototypes.

References

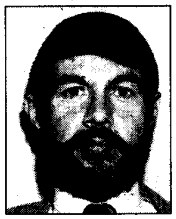
1. J. Conklin and M. Begeman, "The Right Tool for the Job," *Byte*, Vol. 13, No. 10, Oct. 1988, pp. 255-266.

2. T.J. Biggerstaff, *Systems Software Tools*, Prentice Hall, 1986.

3. M.D. Lubars, "Wide-Spectrum Support for Software Reusability," Tech. Report, STP-276-87, MCC, 1987, also in *Workshop on Software Reuse*, Rocky Mountain Institute of Software Engineering, Boulder, Col., Oct. 1987.

4. E. Gullichsen et al., "The PlaneTextBook," Tech. Report STP-333-86, MCC, 1986, republished as nonconfidential report STP-206-88, MCC, 1988.

5. J.A. Feldman et al., "Computing with Structured Connectionist Networks," *Comm. ACM*, Vol. 31, No. 2, Feb. 1988.



Ted J. Biggerstaff joined MCC in 1985 as director of design information. He is directing research in design reusability and design recovery. His research interests include software engineering, knowledge-based approaches to reusability and specification, program generation techniques, program development tools, and natural-language processing.

He is the author of *Systems Software Tools* (Prentice Hall, 1986) and coeditor (with Alan Perlis) of a two-volume book titled *Software Reusability* (Addison-Wesley/ACM, 1989) and the September 1984 special issue of the *IEEE Transactions on Software Engineering* that focused on reusability. He also organized one of the first large-scale workshops on reusability (Newport, R.I., 1983).

Biggerstaff received a Boeing Fellowship in 1974. He received the BA degree in physics from the University of Nebraska in 1964, and the MS and PhD degrees in computer science from the University of Washington, Seattle, in 1971 and 1976, respectively. He is a member of ACM, the IEEE Computer Society, and AAAI.

Readers can contact Biggerstaff at Microelectronics and Technology Corporation, 9390 Research Blvd., Kaleido Building II, Austin, TX 78759.

July 1989