Object-Oriented Design Archaeology with CIA++

Judith E. Grass AT&T Bell Laboratories

ABSTRACT: Increasing numbers of programmers find that they must work on large software systems that they did not write and do not entirely understand. In this situation it is necessary for the programmer to build a working model of the system's design. The process of constructing a working design model from studying the source code may be called *software archaeology*.

This paper demonstrates how software archaeology can be done within the framework of an appropriate design methodology using good static analysis tools. The Object-Oriented Design (OOD) methods described by Grady Booch and James Rumbaugh provide the framework for my investigation. The static analysis tools are based on CIA++.

The *C++ Information Abstractor, CIA++*, builds a relational database of information extracted from C++ programs. The database serves as a foundation for the development of C++ programming tools. Current tools in the CIA++ system include tools for graphical display of various views of the program structure, tools for queries about program symbols and relationships, and tools that extract cohesive components from a larger system. These tools can be used as they are, combined or extended to adapt to specific needs.

This paper briefly describes the CIA++ system and demonstrates how it can be used to extract design information from a significant system: InterViews 3.0, a C++ graphical interface toolkit developed by Mark Linton at Stanford University.

---

1. Introduction

In an ideal world, all programs would be well designed, thoroughly documented and maintained by the same people who originally developed them. Although Object-Oriented (OO) techniques have promise in attacking complexity problems and managing software for change, they do not bring software nirvana. The design of an OO system can become obscured in a haze of detail, documents vanish or fail to be provided, and key programmers decamp for an ashram in Tibet. The result is the same: someone new must come to understand the system well enough to be able to complete or modify it without destroying its architecture.

The process of reconstructing a design from its material remains (the source code) is often referred to as "design recovery," but it might be more accurate to describe this process as "design archaeology." The software archaeologist has to dig through many layers of artifacts to gain an insight into the behavior and structure of a software system. It is rare for this effort to result in a complete and accurate recovery of the original design. Instead, the software archaeologist seeks a working model of that design.

Effective software archaeology requires a knowledge of design methods, a collection of tools and an inquisitive mind. The design methods provide a framework and guidance for the excavation. An appropriate collection of tools can make the digging more effective and less time consuming. The process of software archaeology is experimental. The archaeologist starts out with an initial hypothesis about the program: that its design will conform to the model suggested by

the design methods. Each probing with the tools either confirms that expectation, or shows how the code diverges from that model. The archaeologist adjusts the hypothesis and the excavation strategy accordingly.

The tools traditionally provided in the UNIX environment to support learning about programs have fallen into two categories: simple textual tools and special purpose analytical tools. *Vi* and *grep* are familiar examples of textual tools. Analytical tools include such C language tools as *xref* (a cross-referencing tool), *prof, ctags* and *dbx*. Some of these tools have been adapted for C++ and some new specialized tools added: *hier, publik*[1]. The textual tools are weak tools for analysis because they have no information about program structure or semantics. Real analysis is left to the programmer. The specialized tools generally are inflexible in application and tend to demand specialized parsing of the program text every time they are invoked. This makes these tools inefficient.

Browsers vary in the amount of analysis they do. Usually they function like textual tools. They may help programmers search and navigate from reference to reference in a program, but they cannot provide generalizations about the overall structure or illustrate how major components of the system interact.

The CIA++ system [Grass & Chen] provides a powerful platform for a unified set of analysis tools built around a common relational program database. These tools are fast and efficient because the program is parsed and analyzed only once, when it is entered into the database. The tools are flexible and combine well because they are built on a common relational database schema. CIA++ can support browser systems, but its main goal is higher level analysis of program structure. This paper briefly describes the CIA++ system and the tools built upon it. The main part of the paper illustrates their use and flexibility in analyzing a complex software system: the InterViews library. The intention is not to exhaustively demonstrate the features of CIA++. Instead, this paper presents an effective procedure for design archaeology based on OOD principles. This includes illustrations of how knowledge of design methodology can be applied to the problem and the kind of thought processes a software archaeologist may employ.

1. These are tools developed and used within AT&T that respectively display class hierarchies and list the public interface of a class.

I have chosen the InterViews library as a test case for several reasons: it is sufficiently complex; it is familiar to many people programming in C++; it is publicly available and it is a program that I want to use to implement a graphical interface for CIA++. This paper discusses the InterViews 3.0 alpha release. The 3.0 release is based on the latest version of the AT&T C++ compilation system (2.1) and makes good use of its features. As such, it is a richer and more interesting system to study than earlier versions of InterViews.

2. *An Introduction to The CIA++ System*

CIA++ is made up of three major components: *ciafront,* which analyzes a C++ source module and stores the results of the analysis in a database module; the database linker, which combines the modules into a merged relational database; and the *ShareView* tools, which access the database to answer specific queries about the program. The database schema and basic toolkit are described in detail in Grass & Chen. This section provides only a synopsis of that information.

2.1 Ciafront

Ciafront scans, parses and generates a single database module for a single C++ file. The database it generates includes data from the root file as well as from all the files that get included by the preprocessor. Detailed information is saved about the definitions of five kinds of C++ entities: files, macros, types, functions and variables. Class members are included. Entities declared within a function are considered local and not included. The CIA++ database also saves cross-referencing information. Information is stored about the relationships between any two kinds of tracked entities. Special information is stored about inheritance, containment and friendship relations. *Ciafront* was implemented by reworking *cfront* 2.1.

The complete database for an entire C++ program contains information from all of the individual database modules from all the relevant source files.

2.2 The Basic CIA++ Tools

The core of the *ShareView* toolkit is made up of a small set of packaged query commands: *Def, Ref, Viewdef, Viewref*. *Def* executes basic queries about entity declarations and definitions. For example, the query `Def type ivGlue` extracts database information about the type named *ivGlue*[2]. The formatted response to this query appears in the top half of figure 1. The bottom half of that figure contains an example of the unformatted output for the same query. The formatted query shows the name of the type *ivGlue,* the truncated name of the file in which it is declared, its declared type *class,* the line that the declaration began on and the fact that this declaration is actually a definition (*df*). If there were multiple declarations of *ivGlue* in this library, all declaration instances would be shown in response to this query.

The unformatted version contains much more complete information and untruncated names, but it is not easy to read. This form is intended to be used as input to other analysis, formatting or user interface tools. The CIA++ tools that generate graphic views are implemented in this manner. All formatted displays can be executed by generating unformatted responses to queries and passing that through a formatting filter. This makes it possible to easily customize the displays. The default formatter truncates fields to allow responses to fit into an eighty column display, reflecting the least common denominator terminal environment.

Ref implements cross-referencing queries and presents the information in either a formatted or unformatted style. *Viewdef* and *Viewref* support the same kinds of queries, but extract the relevant lines of code from the source rather than presenting a formatted digest of information.

The basic CIA++ Def query has the syntax:

`Def` *Entity_description Optional_Selectors,*

where *Entity_description* includes the specification of an entity kind: `file`, `func`, `var`, `macro`, `type` or – and a name. The symbol –

2. In InterViews 3.0 there is a convention that most class names from the 3.0 version of the library carry an *iv* prefix. This is not easy to spot in the source code files because the prefix gets applied through macro expansion of the base names. In this example, *ivGlue* appears in the source text as *Glue*. Classes in the library used for 2.6 compatibility are prefixed by *iv26*.

```
falcon 54> Def type ivGlue
file                 dtype                    name                      bline df
==================== ======================== ========================= ===== ==
de/InterViews/glue.h class                    ivGlue                    36    df

falcon 55> Def -u type ivGlue
6442;ivGlue;t;/usr/local/src/X11R4/contrib/toolkits/InterViews/iv/src/include/In
terViews/glue.h;class; ;36;0;49;df; ; ;
```

Figure 1: A simple definition query

```
falcon 120> FuncXmap MakeNewPainter
```

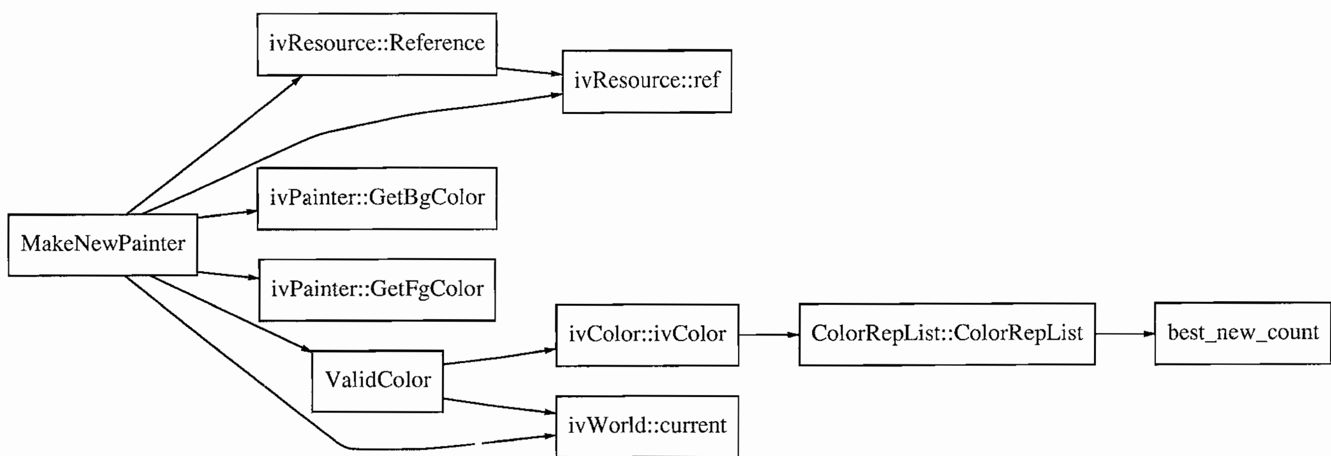


Figure 2: A sample call graph

is a *wild-card* that matches all names and all entity kinds. The *Optional_Selectors* can be used to restrict the search to database entries with specific characteristics. Regular expressions may occur in the selection clauses. For example, the command

```
Def type - dtype="^class$|^$ df="df"
```

extracts a list of all types defined as either `class` or `struct`. The word *dtype* is an abbreviation for "data type." The final selection clause confines the output to only actual data definitions, as the abbreviation *df* stands for "definition flag." The notation used for these commands is terse and rather opaque. The intention is to build GUI interfaces on top of these and hide them from naive users.

The syntax of the *Ref* query and other cross-referencing commands are similar, but describe two entities:

`Ref` *Entity_description_1 Entity_description_2 Optional_Selectors.*

The cross-reference query, *Ref,* presents direct relationships in pairs. Often the most interesting relationships in programs are not the direct relationships extracted by the *Ref* command. Frequently the programmer needs to know about *indirect* relationships.

File inclusion is an interesting illustration of this. In order to compile any particular source code file, the compiler must be able to find all of the header files that define entities used in the source file. The source file may directly include several headers, but it may be dependent on many files that it does not directly include, since its own header files may in turn include other files. The include relationship is transitive. The entire network of direct and indirect inclusion can be established through a process of *transitive closure*. The graph that results from this process contains the name of all files that the source depends on and a map of all the direct and indirect dependencies induced by inclusion.

Several CIA++ tools generate closures of the relationships found in C++ programs. Closure is used to make maps of larger program structures. As for file inclusion, the relationship maps aid in program navigation, but also provide an overview of the organization of a system. Components that are independent stand out, and components that work closely together become apparent. A call graph, for example, is the closure of a simple function cross-reference table. Figure 2 contains the call graph for the function `MakeNewPainter` in the Inter-

Views library. It shows all of the functions that may be called when that function is executed. This could be represented as a table of function pairs defined by the *A_calls_B* relationship. However, a call graph is much more expressive than a table.

CIA++ generates graphs of program relationships with the *Dagen* tool. *Dagen* is written using the *Ref* query, a shell script and the awk language [Aho et al.]. Its output is input for the automatic graph layout program *Dag* [Gansner et al.]. This in turn generates *pic* or *PostScript* output that can either be printed or displayed on a workstation. *Dagen* is a powerful tool that demonstrates how the toolkit can be expanded and adapted using readily available UNIX system tools. It also demonstrates how CIA++ follows the classic UNIX system approach to building tools.

Other tools in the toolkit generate statistics about component connectivity (*Ciafan*), locate unused entities (*Deadcode*) and find strongly connected subsystems (*Subsys*). As an option, *ciafront* generates checksums for all database entities. These may be used to compare two versions of a program database and to analyze the differences between the variants.

Many of these tools will be applied in the analysis of InterViews presented below. CIA++ toolkit commands will be used directly when their command syntax is simple. In some cases the syntax and options for specialized queries may become complex. In these cases wrapper commands are used. A wrapper command is a simple shell script that can execute a specialized query, packaging up the normal CIA++ syntax with specialized selector clauses. Translations of some of the wrapper commands are presented in an appendix.

2.3 Similar Systems

There are several existing systems comparable to CIA++. These include the ParcPlace *ObjectWorks* system, the *Saber C++* interpreter and the class browser tool *cbrowse* [Reiss Meyers] and the *XREFDB* database and *XREF* tools [Lejter et al.] in the *FIELD* system.

These systems differ from CIA++ and each other in several dimensions. Some of these systems implement closed systems which either provide a fixed collection of customized commands, or allow only limited use of pre-existing tools (*ObjectWorks, Saber C++*). These tools seem limited in the amount of code that they can handle. The *cbrowse* system is, strictly speaking, a browser and does not imple-

ment semantic code analysis. *XREF* and *XREFDB* are quite similar to CIA++ and could be applied in the same manner[3].

3. Design Archaeology

The design methodologies developed in the books by Booch and Rumbaugh et al. are based on *Object-Oriented Analysis* [Shlaer & Mellor; Coad & Yourdon]. *Analysis* is the process of investigating system specifications and requirements that yields an abstract model of a client's real-world problem. Once an analysis is completed, the design process can begin. Design is split into two phases: a high-level phase in which a system architecture is developed and low-level design and implementation phase. My assumption in doing design archaeology is that accurate and up-to-date documents describing the design are not available. What is left is the final artifact of that design: the code itself. The task is to discover as much about the original design as possible using the tools provided by CIA++. As a side-effect, it may be possible to make certain inferences about the original modeling of the problem, but the original analytical model of the problem is far enough removed from the implementation that these inferences may not be reliable.

3.1 The InterViews Library

The InterViews library [Linton & Calder; Linton et al. 1988; Linton et al. 1989] implements an OO user interface package in C++ and is built on the X Window System system. The system contained in the InterViews 3.0 *lib/InterViews* directory includes 221 source, header and bitmap files, and 29,996 lines of code that generates 7,674 database entities[4]. It is a substantial system, but this directory does not encompass the entire system. A second directory called *lib/IV-X11* implements an X interface. The *lib/graphic* library directory contains classes describing graphic classes and routines for drawing. Additional directories implement a standard "look-and-feel" (*lib/IV-look*) and tasking (*lib/Task*). If these libraries are also considered, the size of the system increases to more than 470 files and 69,000 lines of code. I built a database that combined the symbols and references from all of

3. Readers interested in a an overview of OOD tools are referred to Booch & Vliot.
4. This information was provided by the CIA++ tool *Ciastat*.

these libraries into a single database. That combined database contains over 17,000 symbols and over 46,000 cross-references. Most of the effort will be focused on the database generated from just the *lib/InterViews* library. This library is the largest and appears to be the most central.

The combined database generated from all the InterViews libraries could be studied as a whole. This would be a good approach for doing a deep study of the system, but using this database would be somewhat unwieldy as a sample database for a technical paper like this one. Queries on such a database would tend to generate too much information to present as examples. However, I appreciate that isolating the core library may distort the analysis somewhat.

3.2 Design Methods and Design Archaeology

OOD archaeology must be guided by a deep understanding of the OOD process and the products of that process. The software archaeologist must think like a designer in order to know which questions to ask about the software and how to understand the answers. The learning process often follows the same path as the creation process, so I will present a synopsis of the process here.

I use a composite of the OOD methods described in the books by Booch and Rumbaugh. Each of these presents a pragmatic and effective procedure for OOD. Their approaches are essentially similar, but each uses a slightly different terminology and different diagraming techniques. The CIA++ system and toolkit does not favor any particular design methodology, as its model reflects the C++ programming language semantics rather than any design methodology.

As described above, the design process must start with an *analysis*. The *analysis* model of a system concentrates on its visible characteristics and behavior. It directly reflects the requirements and specifications of the customer. According to Rumbaugh et al., the analysis model typically can be broken into these subcomponents:

- A Problem Statement;
- The Class Model: a static description of important abstractions and the relationships between them[5];

5. My terminology differs somewhat from Rumbaugh's. He would call this the *Object Model,* which conflicts with the usual terminology used by C++ programmers, who expect objects to be concrete and instantiated.

- The Dynamic Model: a dynamic description of instantiated objects and their interactions[6]. The dynamic model includes state transition diagrams and other time dependent information;
- The Functional Model: a description of data flow.

Actual design starts with the design of a *system architecture*. During this phase the system is partitioned into subsystems and basic decisions are made about hardware platforms, performance targets and fundamental implementation strategies. Some of these decisions will be apparent in the final implementation code from data structures, control constructs and code partitions. Other decisions will not be apparent unless explicitly written into comment blocks in the code. Architectural decisions can cause a review of the analysis and result in changes in the analysis model. The OOD process is inherently iterative.

The complete *design model* includes system architecture combined with detailed class, dynamic and functional models. While the analysis model contains a description of visible characteristics of the system, the design model is oriented towards implementation. All of the information present in the analysis model is present in the design model, but it is augmented. The design model not only describes what the system must do, but how it must do it. Again, as the design model is built, the designer gains new insights into the nature of the problem at hand. This may cause revisions of the analysis and architecture.

A successful design translates into code without much fuss or many surprises. The implementation should be an accurate reflection of the final, complete design. Because analysis, design and implementation are so tightly intertwined, much information about the design and analysis can be recovered by retracing the design process using analysis tools like CIA++. The rest of this paper illustrates this process. Due to length limitations, the analysis presented here is not exhaustive.

3.3 The Class Model

An analysis of an OO system starts by studying classes and the relationships between them. This is the *Class Model* component of the design. The class model describes the foundation of the design, establishing the rules within which the entire system operates.

6. Booch refers to this component as the Object Model.

3.3.1 The Data Dictionary

A *Data Dictionary* is one component of Rumbaugh's class model. This contains class names, attributes and behaviors. It also contains information about interactions with other classes. The software archeologist can derive crucial information about the static system structure from such a dictionary. Rumbaugh discusses this primarily in the context of analysis. A data dictionary augmented with implementation details is also part of the design model.

The data dictionary augmented with diagrams that illustrate class interactions and dependencies present a fairly complete representation of the class design of a system. Class diagrams may be written using any of a number of notations. Whether Booch's *Class Relationship Diagrams,* Rumbaugh's entity-relationship style *Object Model Diagrams* or some other idiosyncratic notation is used, the diagrams will tend to contain the same information. They will show patterns of inheritance, use and inclusion.

Since CIA++ maintains detailed descriptions of type declarations, class membership and cross-referencing, most of the information in a data dictionary is easy to recover from the code. The relationship diagrams can also be recovered, although at this time they cannot be presented directly in the form that Rumbaugh or Booch would use.

Building the data dictionary starts by simply requesting a list of the classes defined in InterViews. This would include both the definitions of `class` and `struct` types in the program. The command `ClassList` produces a list of 335 class definitions. A portion of the response to this query is presented in figure 3. The first column of the table gives the name of the file containing the definition. The second column contains the data type. The third column gives the type name. The fourth column gives the line number on which the definition begins. The last column shows that all of these are actual definitions, rather than simple declarations. The full response to this query is presented in an appendix.

From this list is seems that there are an enormous number of classes that must be entered in the data dictionary. However, the first task in studying the system is to understand its overt behavior. In a well designed C++ program, this should be embodied in public class interfaces and externally scoped data structures and functions. Implementation should be encapsulated in private class members and static or function scoped entities. Efforts to rebuild the data dictionary must

```
falcon 56> ClassList
file                 dtype                 name                     bline df
==================== ===================== ======================== ===== ==
../align.c           struct                __mptr                   -2    df
nterViews/geometry.h class                 ivRequirement            37    df
nterViews/geometry.h class                 ivRequisition            68    df
nterViews/geometry.h class                 ivAllotment              86    df
/InterViews/layout.h class                 ivLayout                 35    df
e/InterViews/align.h class                 ivAlign                  36    df
...                  ...                   ...                      ...   ...
```

Figure 3: Formatted response to ClassList query

start with public information, and the result should reflect the analysis model of the system.

The list of classes presented in figure 3 very much reflects implementation and private information. Consider the struct *__mptr* in the table. It is the type of the table of virtual functions generated by the C++ translator. As such, it is a pure artifact of the compilation process and not part of the design or analysis. Compiler generated types, variables and function calls may easily be identified by their line numbers, which are negative. In fact, 79 of the definitions in the class list are definitions of *__mptr*. Another 42 definitions come from the C++ *iostream* library and other system header files, which makes them external to the analysis. There are also 78 classes that are defined inside of source code files (*.c* files). These class definitions are not directly available to users of the InterViews library because they cannot be accessed by the user's code. They are part of the implementation and design of InterViews, but probably not part of the analysis. If all of these special classes are removed from the list, there are actually only 166 classes that are publicly accessible in this library. These are the classes that must be considered first. The additional 78 hidden classes will be taken up in studying the design of the implementation.

In order to capture the entire public interface of this library, I initially need to know how much of that interface is contained in the class definitions and how much of it is in external functions and other external data structures. The query `Def - ::.*` would extract all symbols that were not class members. It is a bit easier to deal with this data if it is broken up by symbol type. First, a list of the global variables is obtained with the command `GlobalVars` which generates a list like the one presented in figure 4. Once again, the first field given is the file that contains the variable definition. The second field is the data type. The column labeled *bline* shows the definition's beginning line.

The third field, labeled *sc* shows the variable's "*s*torage *c*lass". In the response to this query, most of the variables are either statically allocated and scoped or they are enumerator names. The abbreviation *st* in the storage class field indicates a variable that is *static*ally allocated. A variable with *static* scope is indicated by the abbreviation *st* in the sixth field of this table, which is labeled *ms* for "*m*embership and *s*cope". The constant `false` is both statically allocated and statically scoped. Enumerator names, for example `Dimension_X`, are indi-

```
falcon 57> GlobalVars
file               dtype                 sc name                  bline ms df
================== ===================== == ===================== ===== == ==
terViews/boolean.h unsigned const int    st false                 38    st df
terViews/boolean.h unsigned const int    st true                  39    st df
InterViews/coord.h const float           st fil                   37    st df
InterViews/coord.h const int             st pixels                45    st df
erViews/geometry.h const int             en Dimension_X           34       df
erViews/geometry.h const int             en Dimension_Y           34       df
...                ...                   .. ...                   ...   .. ..
```

Figure 4: Formatted response to GlobalVars query

cated by the abbreviation *en* for "*en*umerator" in the storage class field. All of the enumerator names in InterViews that are not class members are externally visible, so there is no marking in the *ms* field. The rest of the variables have static scope, which means that their visibility is limited to a particular file.

There are about ten object pointers which are externally visible. These include several pointers to *ivCursor* objects and a few pointers to *ivSensor* objects. Not all of the data in this library is entirely encapsulated in classes, but it is only a very small collection of rather simple objects that are visible. This is about what would be expected in an OOD. These external objects will be considered in greater detail when the *Dynamic Model* is built.

The other part of the public interface is made up of publicly declared class members and external functions. Again, I would like to know how much of the behavior of this system is encapsulated in the classes and how much is external to the class system. The query `GlobalFuncs` produces a list of functions not encapsulated by classes. Part of this list is presented in figure 5. This list is similar in format to the lists already seen. The *spec* field is the only new component. This field contains additional specification information peculiar to functions. The letter *i* in this field means that the function is declared to be *inline*. The letter *v* would mean a virtual function. About half of the functions appearing in this list are marked *st* in their scope field. These are static functions that are probably utilities used in implementing some class associated with the code module. The rest are true external functions. Some of these (like the function *sqr* in this table fragment) come from the standard C++ libraries. The ones that are not from the standard libraries will need to be closely inspected. The functions contained in the *minmax.h* header file appear to be simply math utilities. None of these are marked as *const* in either their data type declaration or their specification list. It must be presumed that they have side effects and may affect the operation of the classes in this library. However, from the shortness of this list (about 17 interesting functions) and from the names of these functions, it still appears that the classes encapsulate this library's behavior.

The fact that InterViews 3.0 is written using C++ 2.1 raises another interesting possibility. C++ 2.1 allows type definitions to be nested within class definitions [Ellis & Stroustrup]. This means that some of the 166 public classes found earlier may be encapsulated

```
falcon 58> GlobalFuncs
file            dtype                 name                 bline ms spec df
=============== ===================== ==================== ===== == ==== ==
rViews/minmax.h int ()(int,int)       min                  34    ex i    df
rViews/minmax.h int ()(int,int)       max                  34    ex i    df
./composition.c void ()(int)          grow_arrays          107   st i    df
./composition.c int ()(int,ivCompositi prev_forced_break   134   st i    df
clude/CC/math.h int ()(int)           sqr                  210   ex i    df
clude/CC/math.h double ()(double)     sqr                  211   ex i    df
../tray.c       boolean ()(ivAlignment HAlignment          1328  ex i    df
../tray.c       void ()(ivInteractor* LoadInteractorArray  2101  st i    df
...             ...                   ...                  ...   .. ...  ..
```

Figure 5: Formatted response to GlobalFuncs query

inside other classes and not part of the public interface. The query `NonGlobalTypes` generates the list shown in figure 6. The response to this query shows that the only member type definitions in InterViews are enumerator definitions. The library does not use the newer nested type definition feature. As a result, the investigation of the public aspects of the library must include an inspection of all 166 classes.

The data dictionary entry of public functions and attributes can be filled in using queries similar to this one: `PublicMembers - ivInteractor df=df`. This query requests information about all members of any entity kind (type, variables and functions) defined as members of the class *ivInteractor*. The truncated response to this query is in figure 7[7]. The selection clause `df=df` limits the response to points of definition only, which prevents names that are declared multiple times from appearing here more than once. It shows 39 public member functions in the class *ivInteractor* and no public data or types.

The public members of all the InterViews library classes can be extracted in the same way. These are overwhelmingly function members. Public data members include some encapsulated enumerator names and members of simple "C-style" `structs` defined in the standard C and C++ libraries. The entire list of public data members can be generated with the query `PublicMembers var - df=df`. If these and the public data members defined in the InterViews 2.6 compatibility headers are discounted, there are only 26 classes that expose any of their data members. Most of these classes are those hidden within source code files. Only 10 of the classes with public data members are accessed by library users. Three of these appear to be important because they are frequently referenced: *ivPropertyDef, ivDefaultProperties* and *ivEvent*.

The list of attributes and functions for a particular class that was built using the `PublicMembers` command may not completely describe the public interface of the class. *PublicMembers* only lists members immediately declared within the class definition. The complete public interface of a class also includes all the features that it inherits from other classes. The query `Public - ivInteractor` displays the entire public interface of the *ivInteractor* class, including the portions of its interface that it inherits from its parent classes (an incomplete list of these is in figure 8). The class *ivInteractor* inherits a good deal

7. Many of these sample listings are truncated and otherwise shortened, sometimes severely. The full responses to these queries would generate too many lines to print here.

```
falcon 59> NonGlobalTypes
file                 dtype                 name                     bline df
==================== ===================== ======================== ===== ==
e/InterViews/glyph.h enum                  ivGlyph::BreakType       46    df
/InterViews/canvas.h enum                  ivCanvas::Location       85    df
ispatch/dispatcher.h enum                  dpDispatcher::Dispatcher 41    df
...                  ...                   ...                      ...   ...
```

Figure 6: Formatted response to NonGlobalTypes query

```
falcon 60> PublicMembers - ivInteractor df=df
file               dtype                 sc name                  bline ms df
================== ===================== == ===================== ===== == ==
Views/interactor.h ivShape* ()()            ivInteractor::GetShap 170   pb df
Views/interactor.h ivScene* ()()            ivInteractor::Parent  168   pb df
Views/interactor.h ivPerspective* ()()      ivInteractor::GetPers 169   pb df
../interactor.c    void ()()                ivInteractor::ivInter 45    pb df
../interactor.c    void ()(const char*)     ivInteractor::ivInter 49    pb df
../interactor.c    void ()()                ivInteractor::~ivInte 73    pb df
../interactor.c    void ()(ivAlignment,i    ivInteractor::ivAlign 149   pb df
...                ...                   .. ...                   ...   .. ..
```

Figure 7: Formatted response to ivInteractor query

```
falcon 61> Public - ivInteractor df=df
file                         k name                           bline eline df
============================ = ============================== ===== ===== ==
../listener.c                p ivListener::sensor             49    53    df
../interactor.c              p ivInteractor::Adjust           204   206   df
s/2.6/InterViews/interactor.h p ivInteractor::GetPerspective  169   169   df
../interactor.c              p ivInteractor::Update           208   210   df
../interactor.c              p ivInteractor::SetCanvasType    584   606   df
rc/include/InterViews/glyph.h v ivGlyph::no_break             46    46    df
../glyph.c                   p ivGlyph::allocate              37    37    df
../monoglyph.c               p ivMonoGlyph::body              41    49    df
../monoglyph.c               p ivMonoGlyph::allotment         140   146   df
../resource.c                p ivResource::ref                30    30    df
...                          . ...                            ...   ...   ..
```

Figure 8: Formatted response to Public query

of its interface from other classes. Except for a few enumerators inherited from *ivGlyph,* all that it inherits are functions. The list shows a number of classes that *ivInteractor* inherits from, but it does not show whether the inheritance is direct or indirect. While filling the data dictionary, the focus is on individual classes and not large patterns of interactions.

Inheritance is one way in which classes interact, and probably the most important interaction in an OOD. Other interactions are interesting as well. This includes relationships of containment, friendship and use. My investigations have already shown that there are no classes that contain definitions of other classes, so there are no actual containment relationships between classes in this code. One might suspect that the class definitions embedded in source code files might be candidates for nested definitions in a future version of the system. The command `Ref type ivInteractor type -` shows all of the direct type dependencies of the *ivInteractor* class. A sample of the response is in figure 9. The columns marked *k1* and *k2* display the entity kind of the names *name1* and *name2*. The query only asked about types, so these are types. The column marked *rk* shows the *relation kind.* Relations marked *r* are simple reference relations. It is most likely that the class *ivInteractor* has a data member that is an object of this type or a pointer to an object of this type. The `Viewref` command could be used to extract the code containing the reference to verify its nature. Relations marked *i* are direct inheritance relations. The class *ivInteractor* has two direct inheritance relations, so it is an example of multiple inheritance. The *f* marks friend relations. The class *ivInteractor* is a friend of the classes *ivSensor, ivEvent* and *ivWorld.* This means that one cannot completely understand those classes without a thorough understanding of how *ivInteractor* may change their state.

Since friends may violate normal class access controls, it is important to know about them. The query `Ref - - type ivInteractor rkind=f` shows any classes or functions that could violate the encapsulation of *ivInteractor*. The results in figure 10 shows three such classes.

At this point the *Data Dictionary* for the external view of the *Class Model* is essentially complete. If a problem statement or a set of man pages are available, they can be correlated with entries in the data dictionary to get a higher level view of the functioning of the classes. This information may alternatively be captured using the

```
falcon 62> Ref type ivInteractor type -
k1 file1         name1             k2 file2         name2             rk v pt
== ============= ================= == ============= ================= == = ==
t  /interactor.h ivInteractor      t  ews/handler.h ivHandler         r
t  /interactor.h ivInteractor      t  iews/sensor.h ivSensor          r
t  /interactor.h ivInteractor      t  rViews/view.h ivView            i  n pb
t  /interactor.h ivInteractor      t  Views/world.h ivWorld           r
t  /interactor.h ivInteractor      t  iews/window.h ivWindow          r
t  /interactor.h ivInteractor      t  ws/listener.h ivListener        i  n pb
t  Views/event.h ivInteractor      t  Views/event.h ivEvent           f
t  iews/sensor.h ivInteractor      t  iews/sensor.h ivSensor          f
t  Views/world.h ivInteractor      t  Views/world.h ivWorld           f
.  ...           ...               .  ...           ...               .
```

Figure 9: Formatted response to Ref query

```
falcon 118> Ref - - type ivInteractor rkind=f
k1 file1         name1             k2 file2         name2             rk v pt
== ============= ================= == ============= ================= == = ==
t  /interactor.h ivScene           t  /interactor.h ivInteractor      f
t  /interactor.h ivInteractorWindo t  /interactor.h ivInteractor      f
t  /interactor.h ivWorld           t  /interactor.h ivInteractor      f
```

Figure 10: Formatted response to friend query

Class-Responsibility-Collaboration model described in a paper by Beck and Cunningham.

In this paper building the data dictionary is described as if it all had to be done one query at a time, by hand. This process can be automated with a simple shell script and a few small *awk* programs. Running the script may take some time for a large system, but with very little direct human intervention it is easy to generate a complete formatted report for every class in a system. The layout of the report can be considerably more polished than that generated by the raw commands shown here. The basic CIA++ response format is generalized and meant to run on the meanest of terminals. The automated Data Dictionary generator can be much more sophisticated about layout and text processing tools.

3.3.2 Class Relationship Diagrams

The data dictionary alone does not contain enough information to fully represent the Class Model. So far an atomistic view of individual classes has been built. The descriptions of each class include interaction information, but only about direct interactions. Diagrams that present an overview of these relations are still needed to complete the model.

An analysis of inheritance provides a quick classification of the classes into groups that are closely related and reveals important commonalities between classes. A simple CIA++ query, `HierList` (figure 11), yields a list of 229 class pairs related by inheritance, but a list of this kind is no help in establishing an overview of the system. A graphical view makes the hierarchies immediately visible to the eye and provides a map. The command `HeirGraph` yields a graph of all the inheritance hierarchies in the *InterViews* library. The entire map is too large to print legibly on a single sheet of paper. Figures 12 and 13 show two independent inheritance hierarchies contained in the complete inheritance map. Solid lines in inheritance graphs show public inheritance. Virtual inheritance is indicated by the label *v* on the arc. Private and protected inheritance relations would be indicated by dotted and dashed lines, but there are no examples of these in InterViews. These inheritance graphs are generated by the commands `HierGraph ios` and `HierGraph ivCanvas`. The class hierarchy rooted at *ios* uses multiple inheritance. This is actually the structure of the AT&T *iostream* library and not a part of InterViews proper.

```
falcon 119> HierList
k1 file1         name1             k2 file2         name2             rk v pt
== ============= ================= == ============= ================= == = ==
t  Views/align.h ivAlign           t  iews/layout.h ivLayout          i  n pb
t  Views/group.h ivGroup           t  Views/glyph.h ivGlyph           i  n pb
t  Views/image.h ivImage           t  Views/glyph.h ivGlyph           i  n pb
t  /background.h ivBackground      t  s/monoglyph.h ivMonoGlyph       i  n pb
t  CC/iostream.h ostream           t  CC/iostream.h ios               i  y pb
t  iews/bitmap.h ivBitmap          t  ws/resource.h ivResource        i  y pb
.  ...           ...               .  ...           ...               .  . ..
```

Figure 11: Formatted response to HierList query

```
falcon 120> HierGraph ios
```

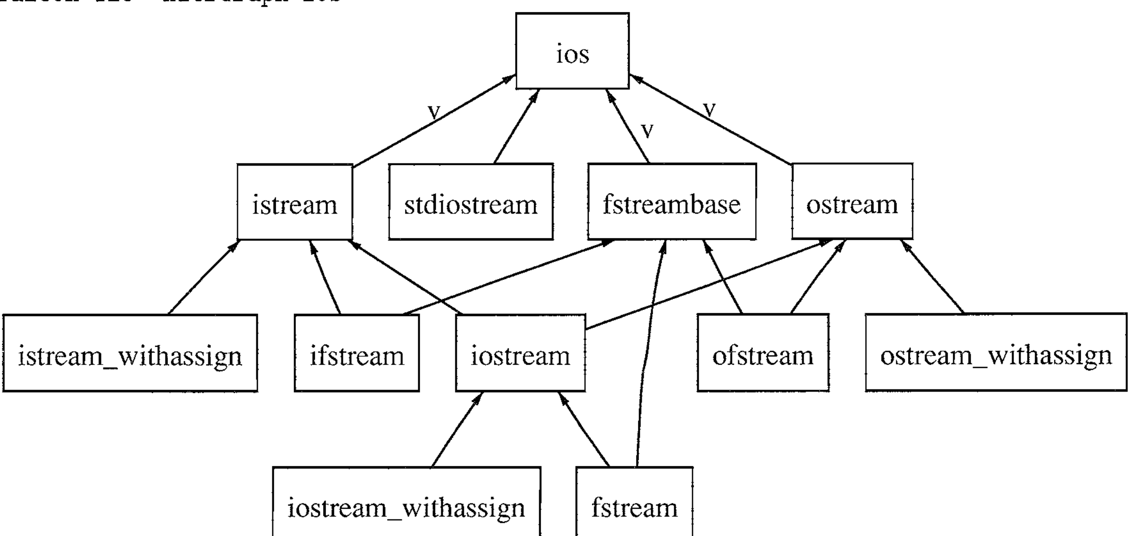


Figure 12: Inheritance Hierarchy for class *ios*

```
falcon 121> HierGraph ivCanvas
```

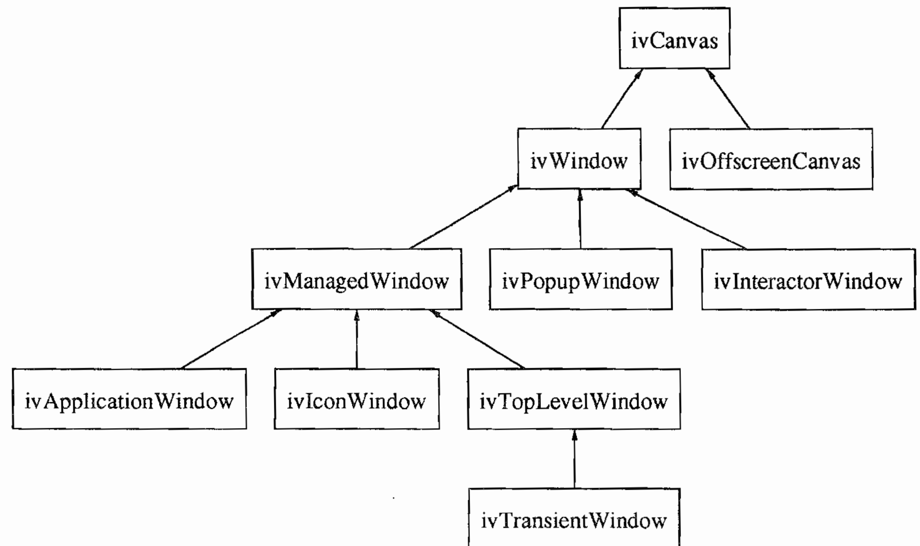


Figure 13: Inheritance Hierarchy for class *ivCanvas*

The class hierarchy rooted at *ivCanvas* is a fairly simple example of a single inheritance hierarchy. The complete inheritance map contains three other small inheritance trees that are externally visible. These are rooted at the classes *ivCompositor, ivLayout* and *dpIOHandler*. In addition there is an immense hierarchy rooted at the class *ivResource*. A total of 112 classes participate in this hierarchy, some of them interrelated by multiple inheritance. The Dagen tools will allow a very large map, like this one, to be printed on several sheets of paper that can be pieced together. Subcomponents of this graph may also be printed. Figure 14 contains a picture of all the nodes that inherit from *ivInteractor*. Figure 15 shows all the classes from which the class *ivInteractor* inherits.

The query `HierList` shows that all inheritance in InterViews is public. InterViews uses inheritance only to establish subtypes and not to establish implementation hierarchies [Liskov]. This also means that the user has access to all of the header defined classes in the library to build new derived classes.

```
falcon 122> HierGraph ivInteractor
```

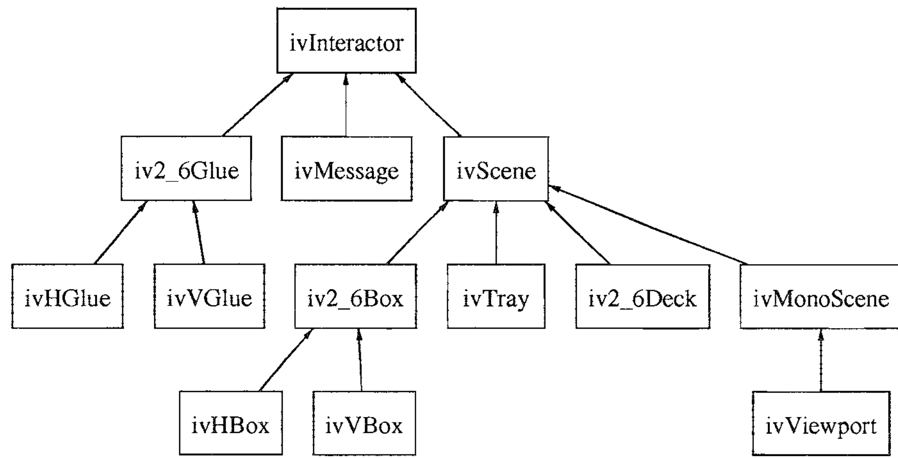


Figure 14: *ivInteractor* and its descendents

```
falcon 124> ParentGraph ivInteractor
```

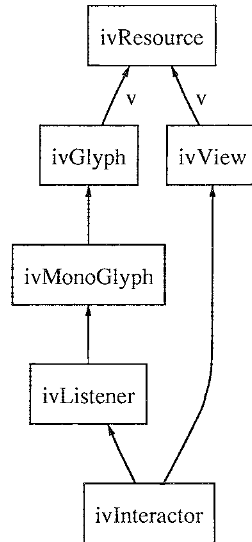


Figure 15: *ivInteractor* and its parents

The inheritance graph needs to be supplemented with a map of friendship relations to highlight places where access controls may be circumvented. Classes that declare friends share a deep dependency on the friends, relying on them not to abuse their special privileges. A complete map of friendship relations can be generated with the command `Dagen t t rkind=f`. Some of the map is shown in figure 16. Several simple relationships involving only two classes were removed from this picture. In general, a complex friendship map in an OOD is not an encouraging sign. Friend access makes it more difficult to isolate the effect of changes during maintenance and localize the cause of errors when problems are found. Some of the friendship relations in this map are inherently less complex than others. The classes lacking the *iv* prefix are local to specific source files, which in itself limits the scope of interactions. Friendship here is probably used for efficiency and convenience. The large friendship relationships between externally visible library components are a bit more troubling as they significantly raise the complexity level of the system.

The friendship system that *ivInteractor* participates in includes classes that are components of the *ivResource* inheritance hierarchy, isolated classes that participate in no inheritance hierarchies and classes belonging to other inheritance systems. The friendship relation ties all of these together in a potentially close dependency. The exact degree of dependency must be investigated by queries that will tell precisely what use the friend makes of its special status.

A complete class relationship diagram must also include a map of simple reference relationships. A direct class to class reference usually indicates that one class contains an object or a pointer to an object of another class. This is a *use relationship*. A use relationship may also result from class references made in a member function. The map of the closure of all simple direct class references can be generated by the command `Dagen t – t rkind="r"`. The resulting map is too large to print here. The command `ClassUseGraph ivInteractor` generates a map showing the closure of the direct use relationship between the class *ivInteractor* and other classes (figure 17). This is not quite a map of all the classes that *ivInteractor* depends on because this map does not necessarily include all the classes used by *ivInteractor*'s member functions. The query `MemberUseGraph ivInteractor` generates a map of these less direct relations.

```
falcon 125> Dagen t t rkind=f
```

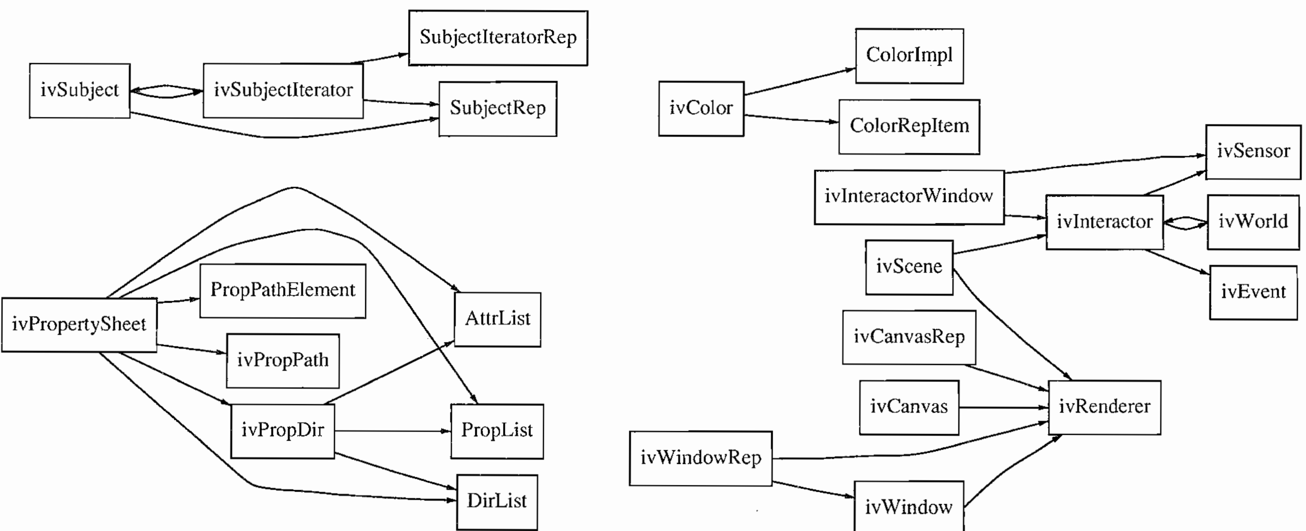


Figure 16: Partial Friendship map for the InterViews library

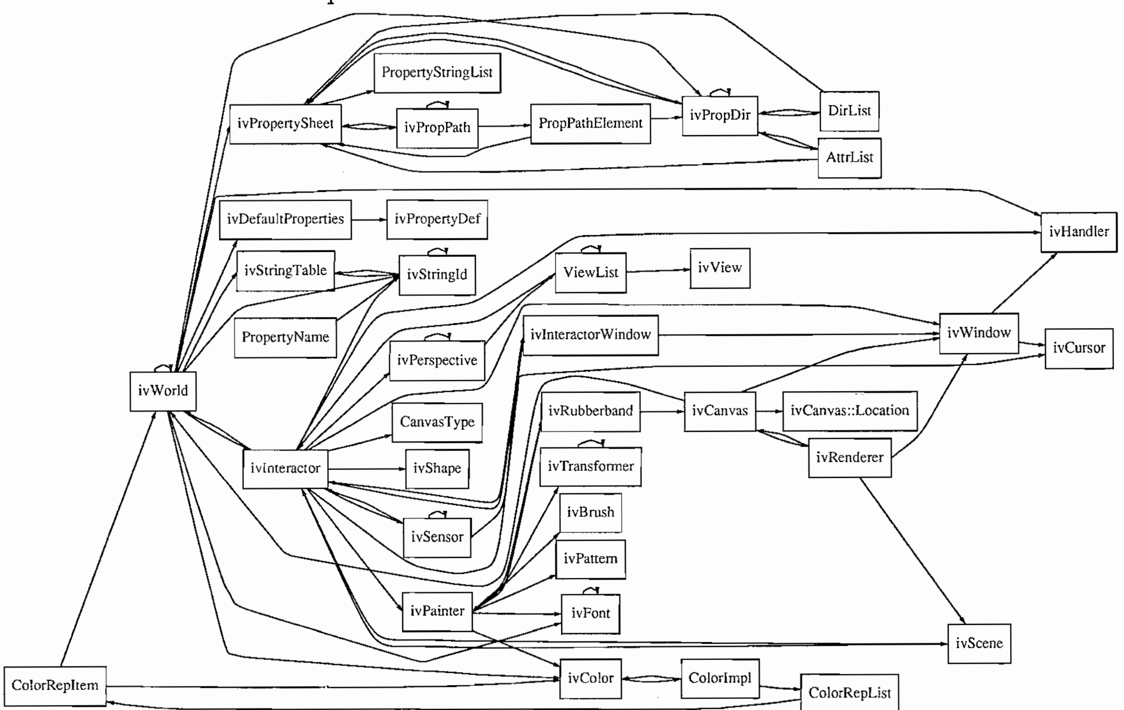


Figure 17: Class to Class use relationships rooted at *ivInteractor*

So far this analysis has concentrated on public aspects of the design. Unfortunately, CIA++ does not distinguish between simple references that occur in the public portions of the class definition and those that are enclosed in private or protected portions of the definition. Once again, the public references are part of the interface of the class, while the private ones pertain to implementation. The distinction is important and observed in the diagrams described by Booch and by Rumbaugh.

With the exception of distinguishing public and private use references, this analysis would allow most of the information that would be in Booch's *Class Relationship Diagram* or in Rumbaugh's *Object Model Diagram* to be rebuilt.

Most of the above analysis concerns the public interfaces of classes. The public interface view strongly suggests the abstract entities and relationships that would have been part of the original problem analysis. The same queries directed at the private and protected parts of the classes can be used to analyze the implementation strategies that realize those abstractions. The implementation design study begins by augmenting the data dictionary with information about data and functions declared within the protected and private portions of the class. The class relationship diagram can be similarly extended.

3.4 The Dynamic Model

Class analysis gives an abstract, static view of the system's type structure. The relationships specified in class declarations hold throughout the life of the system. Objects and object relationships, however, are more transitory because objects are created, destroyed and change state. The *Dynamic Model* captures the way these relationships change over time. This model consists of a series of snapshots of object configurations at critical points in program execution (referred to as the *Object Model*), an analysis of *events, state transition diagrams* and other diagrams that capture timing and the flow of control.

CIA++ only provides tools for static analysis, but some aspects of the static analysis can illuminate some dynamic relationships. For example, object configurations are dynamic relationships that change as the result of calls to constructor and destructor functions. These calls are available in the static analysis. The analysis of events, state transitions and control flow is more problematical. CIA++ does not have

any information about the actual values assigned to program variables, because that can only be determined as the program is running. As a result, CIA++ cannot have information about state. Some inferences may be drawn from the names used and from patterns of function calls, but other tools are required to complete the analysis included in the Dynamic Model[8].

As in class model analysis, a distinction can be made between aspects of the object model that are public and visible and those that are hidden behind the public interfaces. Public objects are available to be manipulated by the library users. Objects protected inside the private portions of other objects, encapsulated by statically scoped definitions in files, or embedded in local function definitions belong to the implementation realm. The analysis will start with public entities and relationships.

The most important objects and object relationships in a system are those objects and relations that exist for relatively long time spans. The objects that exist for the longest time spans are usually global ones, so that is the best place to begin. The command `GlobalVars ms="ex"` lists all the external global variables and their type declarations (some of these are shown in figure 18). The full response to this query shows that there are only 22 external global objects in the InterViews library. Aside from a handful of constants, these consist of a small selection of pointers to *ivCursor* objects and four pointer to *ivSensor* objects. The *ivCursor* object defines a visual representation for a cursor, an identifier and initialization code. This is apparent from the data dictionary. An *ivSensor* object is much weightier, defining many operations to specify the handling of input events. This would include such things as pushing down on a mouse button. These pointer names are externally visible, but as far as is known from this query, these pointers are uninitialized. One way to find out if any of these is statically initialized is to look at the definitions using the command `Viewdef`. This extracts the actual definition from the source code, as shown in figure 19. The variable name in this figure contains a regular expression, so it matches all variables containing the string `Event`.

8. Rumbaugh and Booch split up the modeling task somewhat differently. Rumbaugh's *Dynamic Model* primarily consists of event and state diagrams rather than object configurations. Booch maintains Object Diagrams (configuration snapshots) as a model that is separate from the state and event diagrams. My *Dynamic Model* analysis captures the information contained in Booch's requirements better than it does the information in Rumbaugh's model.

```
falcon 150> GlobalVars ms="ex"
file               dtype                 sc name                  bline ms df
================== ===================== == ===================== ===== == ==
../composition.c   const float           st epsilon               97    ex df
../cursor.c        ivCursor*             st defaultCursor         117   ex df
../cursor.c        ivCursor*             st arrow                 118   ex df
../cursor.c        ivCursor*             st crosshairs            119   ex df
../cursor.c        ivCursor*             st ltextCursor           120   ex df
../forcedraw.c     const float           st epsilon               32    ex df
../xymarker.c      const ivCoord         st th                    31    ex df
../sensor.c        ivSensor*             st allEvents             166   ex df
../sensor.c        ivSensor*             st onoffEvents           167   ex df
../sensor.c        ivSensor*             st updownEvents          168   ex df
...                ...                   .. ...                   ...   .. ..
```

Figure 18: Formatted response to GlobalVars query

```
falcon 14> Viewdef v .*Event dtype=ivSensor df=df
Sensor* allEvents;
Sensor* onoffEvents;
Sensor* updownEvents;
Sensor* noEvents;
```

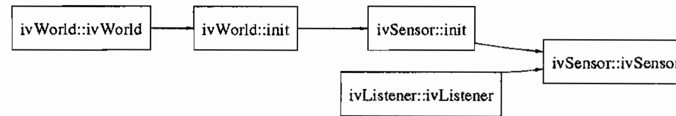
Figure 19: Response to Viewdef query

The selector clauses limit these to the definitions (`df=df`) of variables whose type is derived from *ivSensor* (`dtype=ivSensor`). The response shows that these are declared of the type *Sensor**. Remember, the name *Sensor* is macro expanded when the library is compiled.

From this I must conclude that there are no global objects created within the InterViews library at the time that a system using the library is created. Either other components in the library construct these objects, or the library user has to be responsible for that. This is not an exclusive choice. Since it is only possible to construct new objects through calls to constructors, the patterns of calls to constructors and the nature of the constructors can be used to infer the default initial configuration of the system.

To find the creation of *ivSensor* objects within the library, a graph can be generated that shows all calling sequences of functions that may eventually invoke an *ivSensor* constructor. This graph, generated by the command `Dagen func - func ivSensor::ivSensor`, is the closure of the relationship expressed in the query `Ref func - func ivSensor::ivSensor`, or the set of all functions that directly call any constructor of *ivSensor*. The results are presented in figure 20. This figure also contains a similar chain of calls leading to the instantiations of the cursors. From these graphs it can be seen that nothing in this portion of the library calls the constructor for either *ivWorld* or *ivListener,* so the construction of these objects must be done either by some other component of the library that is not included in this database or by the library user. A quick inspection of the other library directories turn up no calls to the *ivWorld* constructor, so this is apparently a task left to applications. When an *ivWorld* is constructed, all of the *ivSensor* and *ivCursor* objects named above are created at once. The query `Ref fu - v .*Event` shows that all of the *ivSensor* event objects are only referred to by member functions of the *ivSensor* class, at least within this library.

```
falcon 140> Dagen func - func ivSensor::ivSensor
```




```
falcon 141> Dagen func - func ivCursor::ivCursor
```

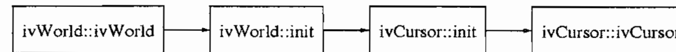


Figure 20: Constructor call graphs reaching *ivInteractor::ivInteractor* and *ivCursor::ivCursor*

Since all of the functions in the constructor call graphs are public, there is some latitude in how this may be done. The function *ivSensor::init* is static and parameterless. It always constructs four specific *ivSensor* objects with default initializations. The function *ivCursor::init* creates a fixed set of cursors with default patterns. The call is parameterized for foreground and background colors. These are generally chosen within the *ivWorld* object. The command `Viewref fu – fu ivCursor::ivCursor` (figure 21) shows some of the constructor calls and reveals the default initializations.

While all of the initialization routines are in the public interface, it is probably unwise to attempt to use them directly because they are "hard-wired" to external object pointers.

Using the CIA++ queries, I can see that the *ivListener* constructor is quite different in its use of *ivSensor* objects. The *ivListener* class contains a private member that is a pointer to an *ivSensor*. An *ivListener* object may be constructed as a dummy that has a null entry for that pointer, or it may create its own sensor object when instantiated as a "working" listener. Over time, the object configuration containing an *ivListener* may change, but it normally will not contain any of the external sensors created with an *ivWorld* object. These are independent.

At this point I can be fairly confident about how the few global objects that have been found are created and initialized, but I have still not entirely constructed the initial Object Model for this system. The key to this problem seems to be the *ivWorld* constructor, as it apparently builds the initial InterViews system configuration. I would like to

```
falcon 136> Viewref fu - fu ivCursor::ivCursor
    noCursor = new Cursor(0, 0, noPat, noPat, fg, bg);
    lowerright = new Cursor(15, 0, lrPat, lrMask, fg, bg);
    upperright = new Cursor(15, 15, urPat, urMask, fg, bg);
    hourglass = new Cursor(6, 8, hourglassPat, hourglassMask, fg, bg);
    crosshairs = new Cursor(7, 8, crossHairsPat, crossHairsMask, fg, bg);
    arrow = new Cursor(1, 15, arrowPat, arrowMask, fg, bg);
    ....
```

Figure 21: Default *ivCursor* initializations

```
falcon 136> ObjectList ivWorld::ivWorld
file            dtype                  name                 bline ms spec df
=============== ====================== ==================== ===== == ==== ==
../strtable.c   void ()(int)           ivStringTable::ivStri 37   pb      df
../sensor.c     void ()()              ivSensor::ivSensor   35    pb      df
../color.c      void ()(ivColor::Inten ivColor::ivColor     52    pb      df
../propsheet.c  void ()(PropertyName)  PropList::PropList   140   pt      df
../propsheet.c  void ()()              AttrList::AttrList   182   pv      df
../propsheet.c  void ()(PropertyName)  ivPropDir::ivPropDir 237   pb      df
../propsheet.c  void ()()              ivPropPath::ivPropPat 291  pv      df
../strpool.c    void ()(unsigned int)  ivStringPool::ivStrin 33   pb      df
../world.c      void ()(ivWorld*,ivWor ivWorld_IOCallback::i 52   pb      df
terViews/font.h void ()(const char*,fl ivFont::ivFont       65    pb      dc
ews/propsheet.h void ()()              ivPropertyDef::ivProp 52   pb i    df
../world.c      void ()(const char*,in ivWorld::ivWorld     107   pb i    df
../color.c      void ()(int)           ColorRepList::ColorRe 38   pb i    df
rViews/cursor.h void ()(short,short,Cu ivCursor::ivCursor   49    pb i    dc
...             ...                    ...                  ..    .. .... ..
```

Figure 22: Constructor calls reachable from *ivWorld* constructor

know the complete list of objects that it creates. A list of all constructor calls possible when the *ivWorld* constructor is called can be obtained with the query `ObjectList ivWorld::ivWorld`. A partial list of object kinds potentially created when an *ivWorld* is created appears in figure 22. Unfortunately, these are only potential calls, because I know of these only from a static analysis. Nor can I tell from this how many of each object may be built or the names of the objects. In order to collect that kind of information, there would have to be a dynamic component to the system that would execute in a manner similar to a debugger and that would do control flow analysis. If I study the code, I may find that many of these constructors are always called, as happens with the constructors for the *ivSensor* and *ivCursor* objects that I have already studied. In any case, the lack of this information means that I can only approximate the contents of the Object Model with this reasoning.

Destructors are never called for any of these objects, either explicitly or implicitly. This suggests that without special intervention in an application program, the object configuration created at the initialization of an *ivWorld* object remains static throughout the life of the program. Although many objects are created when the *ivWorld* constructor is called, no destructor is called by the *ivWorld* destructor. This asymmetrical relation between constructors and destructors suggests that it might be tricky for a library user to dismantle the initial object configuration any time before the entire program is terminated.

The Object Model analysis already presented concerns only the initial state of the system and its final state. Usually the model would contain several snapshots of object configurations occurring at crucial junctures of the system's execution. Identifying these junctures, unfortunately, requires judgement. Fully automating this process is very much a research topic. Some Object Model snapshots can be approximated using reasoning similar to that demonstrated above: by tracking object construction and destruction.

The Object Model is just one component of the Dynamic Model. The components that are missing include state transition diagrams, timing diagrams and control flow diagrams. Since the CIA++ is not a dynamic tool and its database does not capture control flow information, it cannot provide much assistance in uncovering these pieces of the design. This part of the design archaeology needs to be supported by a tool meant specifically for dynamic analysis.

3.5 The Functional Model

The *Functional Model* is the final major component of the Design Model. In the system devised by Rumbaugh, this component describes computations within the design, generally using data flow diagrams supplemented with additional information about invariants that hold during the computations. Not all advocates of OOD would include diagrams of this nature in their designs, considering this to be an artifact of Structured Analysis and Design (see Booch). Booch manages to embed a lot of the information that would be contained in a data flow diagram in his Object and Process Diagrams. In Structured Design, these diagrams often are the centerpiece of the design. In OOD, when these are used at all, they are used to supplement the Class and Dynamic Models and are developed rather late in the process [Coad & Yourdon; Shlaer & Mellor]. The importance of the Functional Model in the overall Design Model can vary. Noninteractive programs that are dominated by their algorithms, like compilers or number-crunchers, will have a major part of their design captured in the functional model. Other programs that just store and retrieve data will not have interesting functional models [Rumbaugh et al.].

Data flow diagrams show *processes, data flows* and objects. Processes take data flows as inputs and transform the data to produce output data flows. Objects in a data flow diagram either serve as passive *data stores* or as *actor* objects that produce or consume data. Processes may contain processes and data-flows, so a data flow diagram may have many layers of nesting with higher level processes in a diagram hiding more detailed and concrete internal data flows. None of the data flow diagrams contain any timing or control flow information. The Functional Model of a system is entirely static.

Although the Functional Model is static, the static program analysis constructed by CIA++ does not contain what is needed to recover that model directly. If processes are identified with C++ functions, then only the nesting of functions within processes is easily recovered from function call graphs generated using CIA++ and Dagen. CIA++ was never intended to do data flow analysis, so it does not save information about sequences of actions, nor the actual variables that are passed as arguments to functions nor the variables that those functions modify. CIA++ records only *direct references* to global objects. It

```
int X,Y,Z;

int F (int i) { return i + Y; }
int H (int i) { return i + Z; }

int G (int j) { j = H(X); return F(j) + X; }
```

Figure 23: Sample program for function references

does not distinguish between references that are simple read operations and those that are writes.

Consider the program fragment in figure 23. The CIA++ database will contain a reference from function `H` to variable `Z` because that global is directly accessed in the function. There will be no reference recorded from function `H` to variable `X`, although that variable appears in a call to the function. That call generates only a reference between function `G` and the functions `F, H` and between function `G` and the variable `X`. The fact that a call to the function `H` precedes a call to the function `F` and that data flows between these functions is not recorded.

While CIA++ does not save information about the actual parameters of function calls, which is a feature of data flow analysis, it does save copious information about the function signature and the types of the formal arguments. This information can be used to indirectly recover the data flow inputs to a process in the functional model. Often the names of the formal parameters and their types reveal important aspects of the data flow. Consider, for example, the process of adding a new component to a complex InterViews graphic interface object, an `ivTray`. The process is invoked by calling the `Insert` function that `ivTray` inherits from its `ivScene` base class. The query `Signature` displays the formal parameter types of this function (figure 24). From this alone it is apparent that insertion requires at least one `ivInteractor` component, and sometimes requires coordinates and alignment information. Looking at the function definition headers adds more detail (figure 25). This gives some names for the coordinates. I would like to know what the reference point for the `x` and `y` coordinates are (relative to the root window? relative to a component window?), but that is a level of semantics at which no static analysis tool can help.

```
falcon 33> Signature ivScene::Insert df="df"
void ivScene::Insert (ivInteractor*)
void ivScene::Insert (ivInteractor*,ivIntCoord,ivIntCoord,ivAlignment)
```

Figure 24: Signatures for *ivScene::Insert*

```
falcon 34> Viewhead ivScene::Insert
void Scene::Insert(Interactor* component) {
void Scene::Insert(Interactor* component, IntCoord x, IntCoord y, Alignment a) {
```

Figure 25: Definition headers for *ivScene::Insert*

```
falcon 41> Signature ivScene::DoMove df="df"
void ivScene::DoMove (ivInteractor*,ivIntCoord&,ivIntCoord&)
```

Figure 26: Signature for *ivScene::DoMove*

In some cases the signature can help distinguish parameters that are read-only parameters from those that are modified. For example, parameters that are declared as `const` are read-only parameters, as are parameters that are not pointers or references. Reference parameters usually indicate the intention to modify a parameter, although sometimes the intention is only to optimize the mechanics of parameter passing. Reference parameters may be both inputs and outputs. In the `Insert` function, the coordinate and alignment parameters are passed by value, so they are only input parameters. The function `ivScene::DoMove` uses coordinate reference parameters (figure 26), which strongly suggests that the coordinate parameters are used both for input and output. Pointer parameters may or may not be modified within a function. Distinguishing the role of a pointer parameter takes extra work. In order to know the role of the `ivInteractor` parameter for the `Insert` or `DoMove` functions, the function code must be inspected.

The database records the function return type, which helps recover more information about output. Inputs and outputs to the process that happen as a result of function side-effects (e.g. direct access to global data or class data members not explicitly passed in the parameter list) can be inferred from the function's list of references to variables. However, that list does not indicate the roles that the variables play. Unfortunately, CIA++ does not record which variables appear as the argument of a return statement, so recovering that information requires reading the code. The return type of `Insert` is `void`, so it does not explicitly return anything.

The actual parameters passed to the `Insert` function can be found by viewing function references, as in figure 27. The cross references show that there are no calls to the multi-parameter version of the `Insert` function within the library code. That is an option that would be used by InterViews application programs. Most of the calls occur in member functions of the classes `ivTray` and `ivViewport`, both of which inherit the function from the class `ivScene` (see figure 14).

These member functions are all fairly small. A model of the data flow in any one of them can be built by looking at the definition. The function `ivTray::ivHBox` is representative (figure 28). In this process, up to six pointers to `ivInteractor` objects are passed into a function that packs them into an array (`LoadInteractorArray`). The array becomes input to `ivScene::Insert`. The `Insert` function ul-

```
\begin{verbatim}
falcon 69> Ref fu ivTray::.* fu ivScene::Insert
k1 file1         name1             k2 file2         name2             rk v pt
== ============= ================= == ============= ================= == = ==
p  ../tray.c     ivTray::ivAlign   p  ../scene.c    ivScene::Insert   r
p  ../tray.c     ivTray::ivAlign   p  ../scene.c    ivScene::Insert   r
p  ../tray.c     ivTray::ivAlign   p  ../scene.c    ivScene::Insert   r
p  ../tray.c     ivTray::ivHBox    p  ../scene.c    ivScene::Insert   r
p  ../tray.c     ivTray::ivVBox    p  ../scene.c    ivScene::Insert   r

falcon 87> Viewref  -n -f fu - fu ivScene::Insert
../tray.c:2089: Insert(i2);
../tray.c:2086: Insert(i1);
../tray.c:2096: Insert(i);
../tray.c:2128:     Insert(i[k]);
../tray.c:2149:     Insert(i[k]);
../tray.c:2178:     Insert(i[k]);
../viewport.c:54:       Insert(i);
```

Figure 27: Some function references to *ivScene::Insert*

```
falcon 82> Viewdef -f -n fu ivTray::ivHBox df=df
../tray.c:2136:void Tray::HBox(
../tray.c:2137:    Interactor* i0, Interactor* i1, Interactor* i2,
../tray.c:2138:    Interactor* i3, Interactor* i4, Interactor* i5, Interactor* i6
../tray.c:2139:) {
../tray.c:2140:    const int n = 7;
../tray.c:2141:    Interactor* i[n];
../tray.c:2142:    int k, last = n - 1;
../tray.c:2143:
../tray.c:2144:    LoadInteractorArray(i, i0, i1, i2, i3, i4, i5, i6);
../tray.c:2145:
../tray.c:2146:    for (k = 0; k < n && i[k] != nil; ++k) {
../tray.c:2147:        if (!AlreadyInserted(i[k]))  Insert(i[k]);
../tray.c:2148:    }
../tray.c:2149:    for (k = 1; k < n && i[k] != nil; ++k) {
../tray.c:2150:  if (TrayOrBg(i[0]) && k == 1)
../tray.c:2151:      tsolver->AddAlignment(Left, this, Left, i[1]);
../tray.c:2152:         else if (TrayOrBg(i[k]) && (k == last || i[k+1] == nil))
../tray.c:2153:             tsolver->AddAlignment(Right, i[k-1], Right, this);
../tray.c:2154:         else
../tray.c:2165:             tsolver->AddAlignment(Right, i[k-1], Left, i[k]);
../tray.c:2166:    }
../tray.c:2167:}
```

Figure 28: Definitions for *ivTray::ivHBox*

timately calls the ivTray::DoInsert function, which modifies the ivTray object, making that object an input and output of the insertion and the entire ivHBox function. This is not apparent from the CIA++ database, because knowing this requires reading the code and understanding the dynamic binding of virtual functions. Finally, an ivTSolver member (the data member tsolver) is used to compute alignments, using the function AddAlignment.

The object tsolver appears to be a passive object that stores and calculates representation information for the ivTray class. Its class definition is buried within the code module for the ivTray class, which clearly indicates that it is part of the implementation of the class, and not part of its public interface. Representation information from the tsolver object is input to the AddAlignment function. Nothing is explicitly output, but the representation information is modified by the call. That function also takes pointers to ivInteractor objects as inputs. Those pointers are stored within the tsolver object, but they are not altered by it.

The description above is a prose description of the data flow diagram for a process that uses the Insert function. The information came both from the results of CIA++ queries and manual inspection of the code. Doing the data flow analysis manually is slow and error prone. Eventually a new tool should be devised to help with this chore, or CIA++ will need to be extended. Much of the information needed to do data flow analysis is present during the compilation process. Building a tool for this purpose is largely a matter of picking a representation for the information and then extracting it in the same way that CIA++ extracts the static definition and cross-referencing information.

3.6 Physical Design

The design analysis that has been presented misses some aspects of the physical design of C++ systems. The Class, Dynamic and Functional Models capture system abstractions, but software archaeologists often are concerned with the way the abstractions are packaged and implemented. The models already investigated cover a great deal of the implementation, but since these focus on features of the C++ programming language, there are issues concerning file partitioning and macro expansions that have been largely ignored. These physical design issues

have a serious impact on performance, code size, compilation time, portability and maintenance.

The file inclusion map for any InterViews file is easy to generate with the FileMap command. Figure 29 shows the file dependencies for the file interactor.c. This map was hand edited to shorten some of the file names by abbreviating the directory paths. The absolute paths to the include files in the local environment are long, some over 75 characters in length. This made the file inclusion graph grow in size. Filtering the names and replacing the long paths made it possible to fit this map on a single page.

The file dependency map for all the files in the InterViews library is so large and complex that a readable copy covers an area the size of an office door. To be precise, the graph is 30 inches wide and 80 inches tall. This takes 32 pages of 8.5 × 11 paper to print. Keep in mind, that map contains only the core library of InterViews. It does not contain the X interface, graphic or "look and feel" libraries or any of the auxiliary libraries.

The file dependency graph for interactor.c is not a simple tree. Some of the files are used by multiple header files, and these might actually be included multiple times in the preprocessing of the interactor.c file. This is not at all unusual in a large C++ program, and a standard method is used here to prevent the C++ defini-tions in such files from being included more than once during preprocessing. Most header files are wrapped within conditional compilation statements (#ifndef) that guard against multiple inclusions. However, the assumption that none of these header files should be multiply included would be false. The headers _enter.h, _leave.h and iv.h, for instance, are used to toggle some name conversion macros on and off. The conversion macros expand key class and function names within the InterViews library with a prefix that prevents the InterViews library names from colliding with names used in either the application or the names or the existing system library environment.

This is why in figure 25 the query is written with the class name ivScene and the response to that query uses the name Scene. The latter is a macro. Figure 30 contains the macro definition queries and extracted macro definitions for Scene and iv.

The remaining scheme is an important part of the design and packaging of the InterViews library. The earlier release of the library

```
falcon 120> FileMap ../interactor.c
```

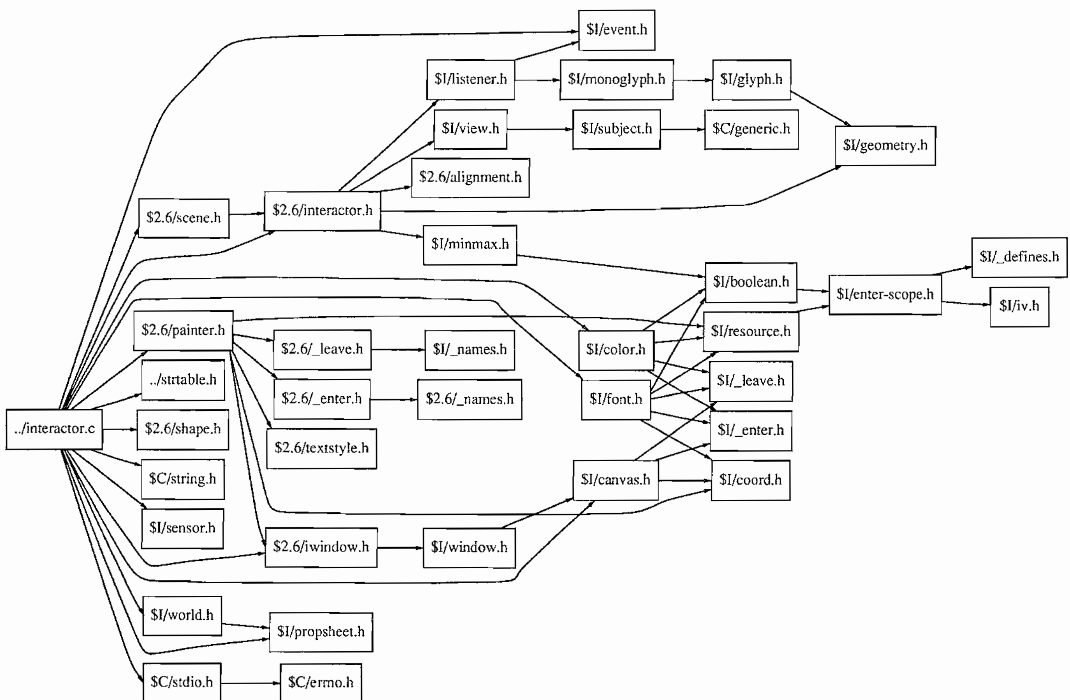


Figure 29: File inclusion map for the file *interactor.c*

```
falcon 171>  Def macro Scene
file                            name                        bline eline mkind
=============================== =========================== ===== ===== =====
c/include/InterViews/_defines.h Scene                       185   185   df

falcon 172> Viewdef macro Scene
#define Scene iv(Scene)

falcon 173> Def macro iv
file                            name                        bline eline mkind
=============================== =========================== ===== ===== =====
/iv/src/include/InterViews/iv.h iv                          29    29    df

falcon 174> Viewdef macro iv
#define iv(name) iv##name
```

Figure 30: The macro *Scene*

did not attempt to isolate its names from those already in the user's environment. This led to serious problems building the library for users unlucky enough to have conflicting names.

3.7 Gaining Perspective

The practice of software archaeology, as shown so far, has concentrated on details. Reconstructing the various analysis and design models pulls the archaeologist into a rather myopic view of a system as characterized by lists of symbols and the relationships between them. A complete understanding of the system also requires stepping back from the details to consider the design as a whole. Inevitably, the archaeologist will want to evaluate the design either against known design principles or against other OO systems. Some comparisons can be captured in numbers: the number of symbols and lines of code in a program, the size of class and function definitions. Metric tools can be used to study these aspects of a design. Some evaluations are harder to characterize: how well inheritance is used, how complete class definitions are. The archaeologist forms a general impression about these as the digging proceeds.

From consideration of the InterViews design, it seems that its designers followed some basic OOD principles. The library is built around a number of basic classes that appear, from the names of their functions, to define coherent subsystems. InterViews builds a relatively shallow hierarchy of subclasses from these. As discussed above, most of the functionality of the library is encapsulated in the classes and virtually all of the data structures used in InterViews are hidden in the private declarations of these classes. These characteristics of the library should be helpful when the library must be extended [Rumbaugh et al.].

When I built the various design and analysis models, it seemed that the classes and functions in the system were rather small. This impression was confirmed using a few simple metrics tools implemented in the CIA++ toolkit.

Typical classes actually are quite small. The average InterViews class has eight members, with the majority of the members being public functions. The small class size seems to be the result of factoring the feature set of major classes in the library into small classes capturing discrete pieces of the feature set. The small classes are used to

compose larger ones through inheritance and delegation. This approach should increase the flexibility of the library and enhance reusability of the components.

The functions in InterViews also tend to be small. In InterViews the largest function (`regmatch`) has 170 lines. There are only three functions that contain over 90 lines. The average number of lines in an InterViews function is eight[9]. InterViews functions are *fine-grained* and fairly primitive. According to Booch, this is a desirable feature in OOD. Small, coherent functions are more likely to be reusable than large functions that do several unrelated things [Rumbaugh et al.].

Multiple inheritance in C++ is a controversial feature that many designers feel should be avoided [Cargill]. In this library only the `ivInteractor` class uses multiple inheritance. This class inherits output characteristics and input sensing from the `ivListener` base class and the ability to update its presentation when it senses an event through its `ivView` base class. The designers have anticipated that users may want to extend the InterViews classes using multiple inheritance. In some cases virtual inheritance is used to derive some "terminal" classes that should be maintained as a single resource rather than duplicated when these are used in multiple inheritance derivations.

Consideration of these issues is part of a fully characterized design, although this may be hard to capture in any kind of formal way. CIA++ can generate lists of statistics, but from these alone it is hard to conclude that the system is or is not a good example of OOD. There is no consensus on metrics for OOD. Judgement about the quality of the design and the overall value of the system usually lies with its users.

3.8 Evaluating CIA++

At this point I should admit that the original purpose for undertaking this study of InterViews was to test the CIA++ toolkit and schema against the demands of a realistic problem. Experience with the CIA++ system prior to this effort was somewhat limited. Mostly it had been used for browsing and generating very specific information to support fixing software that already was fairly well understood. It

9. This partially explains why it was difficult to find an interesting function for data flow analysis.

hadn't been used to do the kind of all-encompassing dig on a large and unfamiliar piece of software that is described here.

Lots of things happened as a result of this experiment. Various kinds of information were added to the database schema when I found that they were missing. The original database schema did not differentiate abstract classes or functions from fully defined ones, nor did the database have an adequate containment relationship. Both of those were added as a result of this experiment. These are just examples, there are others as well.

Many new tools were added to the CIA++ toolkit as a result of this study. Some of these were added just to save typing long, error-prone and common queries (`ClassList` for example). Others combined sequences of queries where each query depended on C++ semantics and the results of an earlier query. One example is `Public`, which generates a complete public interface for a class. `Public` generates a list of class names from an input regular expression, generates the inheritance graph for each of these classes, computes the list of members for the entire inheritance graph, and then applies visibility rules to prune the member list down to those that would be publicly visible for the requested classes. This study identified a few more query tools that still remain to be written.

In doing this experiment, I tried to apply CIA++ in lots of places where it was never meant to go. It was never intended that CIA++ would generate a complete program representation. A database containing a *complete* program representation would be a lot larger and heavier on its feet than the CIA++ database. Completeness was sacrificed for compactness and speed. Primarily this accounts for the lack of local (function scoped) variables in the database. It also accounts for the lack of data flow information, even though that is also static information. Extracting data flow information is hard, and modeling it may also be hard. The extraction could not be done with the simple lexer-parser based abstractors that generated the database for CIA++'s predecessor, the C Information Abstractor (CIA) [Chen et al.]. It can, however, be done within an abstractor that uses the power of a complete compiler, as does CIA++. This experiment persuaded me that I must look at the problem.

Dynamic analysis is another area that should be pursued. What is surprising about applying CIA++ in this realm is not so much where it fails, but rather the number of places where it seems to succeed. The

database contains a fairly complete picture of the potential interactions between components. This is like a map: it shows all the places you can go, and all the ways to get there. However, on any particular trip only some of the places will be visited and only some of the roads traveled. The static database combined with a runtime component could generate the snapshots required for recovering the Dynamic Design Model, and do a lot more besides. This is work in progress.

CIA++ proved to be a powerful tool for software archaeology when used inside the framework provided by OOD methods. This experiment confirmed that this kind of tool has broad applicability. It also pointed out where some changes need to be made and some new tools need to be devised.

4. Conclusions

This paper has demonstrated how CIA++ can be used, within the framework of OO Design methods, to study a significant C++ system. The work described here confirmed that CIA++ is a powerful tool for design analysis. It also identified useful changes and additions to the CIA++ database schema and resulted in many new tools being added to the CIA++ toolkit.

Static analysis tools, like CIA++, are not a complete solution to the problems of design archaeology. Such tools are strong in supporting analysis of static syntactic and semantic relationships and can be used effectively for some kinds of browsing tasks. They can also be used to generate interesting program statistics. On the other hand, static tools provide only weak support for analysis of the dynamic behavior of programs. Other kinds of tools must supplement static tools to effectively recover these aspects of design. Other than debuggers and profilers, tools adapted to this kind of analysis are not common. This study helped identify some of the characteristics of a tool that would support dynamic analysis for OOD.

Effective use of whatever tools are available depends on knowing something about the design principles and methods because these provide the framework that underlies the analysis. Reference to the framework helps the archaeologist plan a dig and it makes it possible to evaluate the results of each step of that excavation. As an example, consider the class relationship diagram for the InterViews library. If

the only graph generated for the class relationships were the graph of the inheritance relationship, the class relationships would appear to be rather simple. The assumption would be that classes implemented well encapsulated abstractions. When the inheritance relationship is supplemented by class usage relationships, in particular the friendship graph, it becomes clear that more careful study is merited. The friend classes may violate some of the encapsulation expectations of the OOD model.

Although using appropriate analytical tools inside a good theoretical framework can significantly help in understanding a design, there are many aspects of design that will not be captured that way. The original design documents would specify class invariants, performance characteristics, and probably contain references to specific algorithms or domain specific abstractions. Some of this may be rediscovered by reading the code. Some of it may only be learned by asking a domain expert. In any case, the design information recovered by the dig provides a strong context for direct code study and identifies where expert help is needed.

The InterViews system appears to be a good example of OO Design. As such, it is probably easier to use the CIA++ tools and get a good result. The same tools applied to a program that was poorly designed would throw a spotlight on the design problems. This is also a useful result, if not as satisfying.

The CIA++ system is a research prototype, however it is in daily use by many C++ programmers. This study has already helped improve CIA++. I hope that this paper will also help the users of these tools (and similar tools) to reap more benefit from them.

References

UNIX System V AT&T C++ Language System: Release 2.1, Library Manual, 1990. Select code 307–158.

A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, Reading, MA, 1988.

K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. *SIGPLAN Notices,* 24(10), October 1989.

G. Booch. *Object-Oriented Design with Applications*. Benjamin Cummings, Redwood City, CA, 1991. ISBN 0-8053-0091-0.

G. Booch and M. Vliot. Tools for object-oriented design. *The C++ Report,* 3(4), April 1991.

T. A. Cargill. Controversy: The Case Against Multiple Inheritance in C++. *Computing Systems,* 4(1), 1991.

Y. F. Chen, M. Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction system. *Transations on Software Engineering,* 16(3):325–334, March 1990.

P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press (Prentice Hall), Englewood Cliffs, NJ, 1990. ISBN 0-13-629122-8.

M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, MA, 1990. ISBN 0-201-51459-1.

E. R. Gansner, S. C. North, and K. P. Vo. DAG—A Program that Draws Directed Graphs. *Software—Practice and Experience,* 18(11), November 1988. Also appeared as AT&T Bell Laboratories TM 59554-871019-04TM.

J. E. Grass and Y. F. Chen, The C++ Information Abstractor. In *USENIX C++ Conference Proceedings,* pages 265–278, San Francisco, CA, April 1990.

M. Lejter, S. Meyers, and S. P. Reiss. Adding Semantic Information to C++ Development Environments. In *Proceedings of the C++ at Work Conference,* pages 103–108, Secaucus, NJ, September 1990.

M. A. Linton and P. R. Calder. The Design and Implementation of InterViews. In *USENIX C++ Workshop Proceedings,* pages 256–267, Santa Fe, NM, November 1987.

M. A. Linton, J. M. Vlassides, and P. R. Calder. Applying Object-Oriented Design to Structured Graphics. In *USENIX C++ Conferencea Proceedings,* pages 81–94, Denver, CO, October 1988.

M. A. Linton, J. M. Vlassides, and P. R. Calder. Composing User Interfaces with InterViews. *Computer,* 22(2):8–22, February 1989.

B. Liskov. Data Abstraction and Heirarchy. *OOPSLA '87 Addendum, published as Sigplan Notices,* 23(5):17–34, May 1988.

S. P. Reiss and S. Meyers, FIELD Support for C++. In *USENIX C++ Conference Proceedings,* pages 293–299, San Francisco, CA, April 1990.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991. ISBN 0-13-629841-9.

S. Shlaer and S. J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press (Prentice Hall), Englewood Cliffs, NJ, 1988. ISBN 0-13-629023-X.

[submitted Aug. 23, 1991; revised Dec. 6, 1991; accepted Dec. 19, 1991]

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the *Computing Systems* copyright notice and its date appear, and notice is given that copying is by permission of the Regents of the University of California. To copy otherwise, or to republish, requires a fee and/or specific permission. See inside front cover for details.

Appendix A: Some CIA++ Commands

ClassList: `Def type - dtype="^class$|^struct$" df="df"`
ClassXref: `Ref type - type - dtype1="^class$|^struct$"\`
`dtype2="^class$|^struct$".`
HierList: `Ref type - type - rkind="i"`
HierGraph: `Dagen type - type - rkind="i"`
Friendof: `Ref - - type "$1" rkind="f"`
AllFriends: `Ref - - type - rkind="f"`
FriendGraph: `Dagen type - type - rkind="f".`
GlobalVars: `Def var ::.* df=df`
Constructors: `Ref - - func "$1::$1"`
FuncXref: `Ref func "$1" func -`
FuncGraph: `Dagen func "$1" func -`
ObjectList: `Subsys -u fu "$1" fu - |\`
`grep "__ct" | Ciadef_filt | Dispdef fu -`

Appendix B: An Example of Uncondensed Query Output

I have avoided presenting complete listings of CIA++ query responses in the body of this paper. This complete listing is provided in order to demonstrate how complete and detailed the information generated by the queries can be. Comprehensive listings of this kind can be overwhelming. The scope of this query can be narrowed by providing regular expressions for the class name (`ClassList ivReq.*` for example) by adding additional attributes to the query (`ClassList -file="window.h"`). As I mentioned above, the multiple definitions of `_mptr` in this list are an artifact of the way *cfront* generates virtual function pointers.

```
falcon 99 ClassList  -
file                 dtype   name                    bline df
==================== ======= ======================= ===== ==
../pattern.c         struct  __mptr                  -2    df
nterViews/resource.h class   ivResource              34    df
InterViews/pattern.h class   ivPattern               39    df
../strut.c           struct  __mptr                  -2    df
de/InterViews/font.h class   ivFontFamily            45    df
de/InterViews/font.h class   ivFont                  63    df
nterViews/geometry.h class   ivRequirement           37    df
nterViews/geometry.h class   ivRequisition           68    df
nterViews/geometry.h class   ivAllotment             86    df
nterViews/geometry.h class   ivAllocation            110   df
nterViews/geometry.h class   ivExtension             131   df
e/InterViews/glyph.h class   ivGlyph                 44    df
e/InterViews/strut.h class   ivStrut                 36    df
e/InterViews/strut.h class   ivHStrut                55    df
e/InterViews/strut.h class   ivVStrut                72    df
../textbuffer.c      struct  __mptr                  -2    df
/InterViews/regexp.h struct  regexp                  44    df
/InterViews/regexp.h class   ivRegexp                61    df
erViews/textbuffer.h class   ivTextBuffer            36    df
1/include/CC/stdio.h struct  _iobuf                  66    df
clude/CC/sys/types.h struct  _physadr_t              56    df
clude/CC/sys/types.h struct  label_t                 57    df
clude/CC/sys/types.h struct  _quad_t                 67    df
clude/CC/sys/types.h struct  fd_set                  101   df
cal/include/CC/pwd.h struct  passwd                  26    df
cal/include/CC/pwd.h struct  comment                 42    df
/include/CC/malloc.h struct  mallinfo                40    df
```

```
falcon 99 ClassList  -
file                 dtype   name                      bline df
==================== ======= ========================= ===== ==
../psfont.c          struct  __mptr                    -2    df
e/InterViews/world.h class   ivWorld                   87    df
/InterViews/psfont.h class   ivPSFont                  37    df
../psfont.c          class   PSFontImpl                38    df
terViews/propsheet.h class   ivPropertyDef             46    df
terViews/propsheet.h class   ivPropertySheet           56    df
e/InterViews/world.h class   ivPropertyData            49    df
e/InterViews/world.h class   ivOptionDesc              64    df
e/InterViews/world.h class   ivDefaultProperties       72    df
../action.c          struct  __mptr                    -2    df
/InterViews/action.h class   ivAction                  36    df
/InterViews/action.h class   ivMacro                   45    df
../align.c           struct  __mptr                    -2    df
/InterViews/layout.h class   ivLayout                  35    df
e/InterViews/align.h class   ivAlign                   36    df
../strtable.c        struct  __mptr                    -2    df
../strpool.h         class   ivStringPool              37    df
../strtable.h        class   ivStringId                34    df
../strtable.h        class   ivStringTable             47    df
../composition.c     struct  __mptr                    -2    df
/InterViews/canvas.h class   ivCanvas                  40    df
/InterViews/canvas.h class   ivOffscreenCanvas         97    df
ude/InterViews/hit.h class   ivHit                     46    df
terViews/monoglyph.h class   ivMonoGlyph               36    df
../composition.c     class   ivCompositionComponent_L  43    df
../composition.c     class   ivBreak_List              94    df
erViews/compositor.h class   ivCompositor              37    df
../composition.c     class   ivBreak                   49    df
rViews/composition.h class   ivComposition             35    df
rViews/composition.h class   ivLRComposition           89    df
rViews/composition.h class   ivTBComposition           100   df
terViews/fixedspan.h class   ivFixedSpan               34    df
terViews/forcedraw.h class   ivForcedDraw              34    df
de/InterViews/glue.h class   ivGlue                    36    df
de/InterViews/glue.h class   ivHGlue                   51    df
de/InterViews/glue.h class   ivVGlue                   60    df
ude/InterViews/hit.h class   ivHitIterator             73    df
ude/InterViews/box.h class   ivBox                     40    df
ude/InterViews/box.h class   ivLRBox                   81    df
ude/InterViews/box.h class   ivTBBox                   91    df
ude/InterViews/box.h class   ivOverlay                 101   df
de/InterViews/tile.h class   ivTile                    34    df
de/InterViews/tile.h class   ivTileReversed            50    df
../composition.c     class   CompositionComponent      38    df
../rubcurve.c        struct  __mptr                    -2    df
/InterViews/window.h class   ivWindow                  41    df
/InterViews/window.h class   ivManagedWindow           103   df
```

```
falcon 99 ClassList  -
file                 dtype   name                       bline df
==================== ======= ========================== ===== ==
/InterViews/window.h class   ivApplicationWindow        149   df
/InterViews/window.h class   ivTopLevelWindow           157   df
/InterViews/window.h class   ivTransientWindow          168   df
/InterViews/window.h class   ivPopupWindow              180   df
/InterViews/window.h class   ivIconWindow               188   df
InterViews/iwindow.h class   ivInteractorWindow         38    df
InterViews/painter.h class   ivPainter                  50    df
InterViews/rubband.h class   ivRubberband               41    df
nterViews/rubcurve.h class   ivRubberEllipse            34    df
nterViews/rubcurve.h class   ivSlidingEllipse           55    df
nterViews/rubcurve.h class   ivRubberCircle             71    df
nterViews/rubcurve.h class   ivRubberPointList          83    df
nterViews/rubcurve.h class   ivRubberVertex             98    df
nterViews/rubcurve.h class   ivRubberHandles            113   df
nterViews/rubcurve.h class   ivRubberSpline             126   df
nterViews/rubcurve.h class   ivRubberClosedSpline       136   df
nterViews/rubcurve.h class   ivSlidingPointList         146   df
nterViews/rubcurve.h class   ivSlidingLineList          162   df
nterViews/rubcurve.h class   ivScalingLineList          172   df
nterViews/rubcurve.h class   ivRotatingLineList         193   df
lude/floatingpoint.h struct  quadruple                  27    df
lude/floatingpoint.h struct  decimal_record             75    df
lude/floatingpoint.h struct  decimal_mode               100   df
al/include/CC/math.h struct  exception                  367   df
al/include/CC/math.h struct  ______complex              562   df
../scene.c           struct  __mptr                     -2    df
e/InterViews/event.h class   ivEvent                    63    df
erViews/interactor.h class   ivInteractor               62    df
nterViews/listener.h class   ivListener                 40    df
de/InterViews/view.h class   ivView                     43    df
InterViews/subject.h class   ivSubject                  38    df
InterViews/subject.h class   ivSubjectIterator          64    df
InterViews/subject.h class   IntSubject                 112   df
InterViews/subject.h class   LongSubject                113   df
InterViews/subject.h class   FloatSubject               114   df
InterViews/subject.h class   DoubleSubject              115   df
6/InterViews/scene.h class   ivScene                    36    df
6/InterViews/scene.h class   ivShape                    37    df
6/InterViews/scene.h class   ivMonoScene                75    df
../cursor.c          struct  __mptr                     -2    df
e/InterViews/color.h class   ivColor                    35    df
/InterViews/cursor.h class   ivCursor                   47    df
../stencil.c         struct  __mptr                     -2    df
/InterViews/bitmap.h class   ivBitmap                   42    df
nterViews/renderer.h class   ivRenderer                 41    df
InterViews/printer.h class   ivPrinter                  40    df
```

```
falcon 99 ClassList   -
file                  dtype   name                   bline df
===================== ======= ====================== ===== ==
InterViews/stencil.h  class   ivStencil              37    df
../transformer.c      struct  __mptr                 -2    df
rViews/transformer.h  class   ivTransformer          38    df
../center.c           struct  __mptr                 -2    df
/InterViews/center.h  class   ivCenter               36    df
/InterViews/center.h  class   ivHCenter              56    df
/InterViews/center.h  class   ivVCenter              62    df
../forcedraw.c        struct  __mptr                 -2    df
../table.c            struct  __mptr                 -2    df
../geometry.c         struct  __mptr                 -2    df
../tray.c             struct  __mptr                 -2    df
../tray.c             class   ivTrayElement          1835  df
../tray.c             class   ivTSolver              1246  df
.6/InterViews/tray.h  class   ivTGlue                40    df
.6/InterViews/tray.h  class   ivTray                 51    df
../tray.c             class   TElement               38    df
../tray.c             class   TList                  176   df
../tray.c             class   TElementList           265   df
../tray.c             class   TTermination           327   df
../tray.c             class   TLoop                  346   df
../tray.c             class   TNode                  362   df
../tray.c             class   TNodeList              577   df
../interactor.c       struct  __mptr                 -2    df
/InterViews/sensor.h  class   ivSensor               38    df
../group.c            struct  __mptr                 -2    df
terViews/aggregate.h  class   ivAggregate            38    df
e/InterViews/group.h  class   ivGroup                33    df
../rubverts.c         struct  __mptr                 -2    df
nterViews/rubverts.h  class   ivGrowingVertices      35    df
nterViews/rubverts.h  class   ivGrowingMultiline     57    df
nterViews/rubverts.h  class   ivGrowingPolygon       67    df
nterViews/rubverts.h  class   ivGrowingBSpline       77    df
nterViews/rubverts.h  class   ivGrowingClosed BSpline 87   df
../page.c             struct  __mptr                 -2    df
de/InterViews/page.h  class   ivPage                 34    df
../page.c             class   ivPageInfo_List        58    df
../page.c             class   PageInfo               39    df
../color.c            struct  __mptr                 -2    df
../color.c            class   ColorImpl              40    df
../color.c            class   ColorRepItem           29    df
../color.c            class   ColorRepList           37    df
../character.c        struct  __mptr                 -2    df
terViews/character.h  class   ivCharacter            37    df
../box.c              struct  __mptr                 -2    df
../box.c              class   ivBoxComponent_List    43    df
../box.c              class   ivBoxAllocation_List   55    df
```

```
falcon 99 ClassList   -
file                  dtype   name                   bline df
===================== ======= ====================== ===== ==
terViews/superpose.h class   ivSuperpose            34    df
../box.c             class   BoxComponent           38    df
../box.c             class   BoxAllocation          46    df
../label.c           struct  __mptr                 -2    df
e/InterViews/label.h class   ivLabel                37    df
../compositor.c      struct  __mptr                 -2    df
../strpool.c         struct  __mptr                 -2    df
../printer.c         struct  __mptr                 -2    df
e/InterViews/brush.h class   ivTransformer          38    df
e/InterViews/brush.h class   ivBrush                40    df
nclude/CC/iostream.h class   ostream                463   df
/InterViews/raster.h class   ivRaster               40    df
nclude/CC/iostream.h class   streambuf              179   df
nclude/CC/iostream.h class   ios                    49    df
nclude/CC/iostream.h class   istream                346   df
nclude/CC/iostream.h class   iostream               560   df
nclude/CC/iostream.h class   istream_withassign     568   df
nclude/CC/iostream.h class   ostream_withassign     576   df
nclude/CC/iostream.h class   iostream_withassign    584   df
nclude/CC/iostream.h class   Iostream_init          605   df
../printer.c         class   PrinterInfo            75    df
../rubgroup.c        struct  __mptr                 -2    df
nterViews/rubgroup.h class   ivRubberGroup          34    df
../rubgroup.c        class   RubberList             31    df
../image.c           struct  __mptr                 -2    df
e/InterViews/image.h class   ivImage                36    df
../simplecomp.c      struct  __mptr                 -2    df
erViews/simplecomp.h class   ivSimpleCompositor     30    df
../arraycomp.c       struct  __mptr                 -2    df
terViews/arraycomp.h class   ivArrayCompositor      36    df
../discretion.c      struct  __mptr                 -2    df
erViews/discretion.h class   ivDiscretionary        34    df
../deck.c            struct  __mptr                 -2    df
de/InterViews/deck.h class   ivDeck                 32    df
../deck.c            class   ivDeckInfo_List        35    df
../deck.c            class   ivDeckInfo             30    df
../margin.c          struct  __mptr                 -2    df
/InterViews/margin.h class   ivMargin               34    df
/InterViews/margin.h class   ivHMargin              68    df
/InterViews/margin.h class   ivVMargin              80    df
/InterViews/margin.h class   ivLMargin              92    df
/InterViews/margin.h class   ivRMargin              99    df
/InterViews/margin.h class   ivTMargin              106   df
/InterViews/margin.h class   ivBMargin              113   df
../hit.c             struct  __mptr                 -2    df
../shape.c           struct  __mptr                 -2    df
```

```
falcon 99 ClassList   -
file                 dtype   name                     bline df
==================== ======= ======================== ===== ==
../texcomp.c         struct  __mptr                   -2    df
InterViews/texcomp.h class   ivTeXCompositor          30    df
../texcomp.c         class   BreakSet                 32    df
../subject.c         struct  __mptr                   -2    df
../subject.c         class   SubjectRep               37    df
../subject.c         class   SubjectIteratorRep       92    df
../subject.c         class   ViewList                 34    df
../rubrect.c         struct  __mptr                   -2    df
InterViews/rubrect.h class   ivRubberRect             34    df
InterViews/rubrect.h class   ivRubberSquare           53    df
InterViews/rubrect.h class   ivSlidingRect            65    df
InterViews/rubrect.h class   ivStretchingRect         80    df
InterViews/rubrect.h class   ivScalingRect            96    df
InterViews/rubrect.h class   ivRotatingRect           112   df
../glyph.c           struct  __mptr                   -2    df
../xymarker.c        struct  __mptr                   -2    df
nterViews/xymarker.h class   ivXYMarker               36    df
../fixedspan.c       struct  __mptr                   -2    df
../glue2_6.c         struct  __mptr                   -2    df
.6/InterViews/glue.h class   ivGlue                   37    df
.6/InterViews/glue.h class   ivHGlue                  47    df
.6/InterViews/glue.h class   ivVGlue                  57    df
../world.c           struct  __mptr                   -2    df
Dispatch/iohandler.h class   dpIOHandler              34    df
ispatch/dispatcher.h class   dpDispatcher             39    df
InterViews/handler.h class   ivHandler                34    df
../world.c           class   ivWorld_IOCallback       51    df
../world.c           class   ivWorld_HandlerCallBack  472   df
../viewport.c        struct  __mptr                   -2    df
rViews/perspective.h class   ivPerspective            39    df
nterViews/viewport.h class   ivViewport               39    df
../rubline.c         struct  __mptr                   -2    df
InterViews/rubline.h class   ivRubberLine             34    df
InterViews/rubline.h class   ivRubberAxis             53    df
InterViews/rubline.h class   ivSlidingLine            65    df
InterViews/rubline.h class   ivScalingLine            80    df
InterViews/rubline.h class   ivRotatingLine           95    df
../resource.c        struct  __mptr                   -2    df
../patch.c           struct  __mptr                   -2    df
e/InterViews/patch.h class   ivPatch                  34    df
../regexp.c          struct  __mptr                   -2    df
include/CC/iomanip.h class   smanip_int               105   df
include/CC/iomanip.h class   sapply_int               105   df
include/CC/iomanip.h class   imanip_int               105   df
include/CC/iomanip.h class   iapply_int               105   df
include/CC/iomanip.h class   omanip_int               105   df
```

```
falcon 99 ClassList  -
file                 dtype   name                     bline df
==================== ======= ======================== ===== ==
include/CC/iomanip.h class   oapply_int               105   df
include/CC/iomanip.h class   iomanip_int              105   df
include/CC/iomanip.h class   ioapply_int              105   df
include/CC/iomanip.h class   smanip_long              106   df
include/CC/iomanip.h class   sapply_long              106   df
include/CC/iomanip.h class   imanip_long              106   df
include/CC/iomanip.h class   iapply_long              106   df
include/CC/iomanip.h class   omanip_long              106   df
include/CC/iomanip.h class   oapply_long              106   df
include/CC/iomanip.h class   iomanip_long             106   df
include/CC/iomanip.h class   ioapply_long             106   df
ude/CC/stdiostream.h class   stdiobuf                 18    df
ude/CC/stdiostream.h class   stdiostream              37    df
include/CC/fstream.h class   filebuf                  17    df
include/CC/fstream.h class   fstreambase              49    df
include/CC/fstream.h class   ifstream                 71    df
include/CC/fstream.h class   ofstream                 86    df
include/CC/fstream.h class   fstream                  101   df
../monoglyph.c       struct  __mptr                   -2    df
../target.c          struct  __mptr                   -2    df
/InterViews/target.h class   ivTarget                 42    df
../border.c          struct  __mptr                   -2    df
/InterViews/border.h class   ivBorder                 36    df
../listener.c        struct  __mptr                   -2    df
../lrmarker.c        struct  __mptr                   -2    df
nterViews/lrmarker.h class   ivLRMarker               36    df
../brush.c           struct  __mptr                   -2    df
../shapeof.c         struct  __mptr                   -2    df
InterViews/shapeof.h class   ivShapeOf                34    df
../box2_6.c          struct  __mptr                   -2    df
../box2_6.c          class   ivBoxElement             34    df
../box2_6.c          class   ivBoxCanonical           48    df
2.6/InterViews/box.h class   ivBox                    39    df
2.6/InterViews/box.h class   ivHBox                   70    df
2.6/InterViews/box.h class   ivVBox                   95    df
../box2_6.c          class   BoxDimension             41    df
../propsheet.c       struct  __mptr                   -2    df
../propsheet.c       class   ivPropDir                89    df
../propsheet.c       class   ivPropPath               118   df
../propsheet.c       class   PropertyStringList       39    df
nclude/CC/sys/stat.h struct  stat                     28    df
../propsheet.c       class   PropList                 42    df
../propsheet.c       class   AttrList                 61    df
../propsheet.c       class   DirList                  75    df
../propsheet.c       class   PropPathElement          110   df
```

```
falcon 99 ClassList  -
file                 dtype   name                     bline df
==================== ======= ======================== ===== ==
../background.c      struct  __mptr                   -2    df
erViews/background.h class   ivBackground             38    df
../shadow.c          struct  __mptr                   -2    df
/InterViews/shadow.h class   ivShadow                 36    df
../tile.c            struct  __mptr                   -2    df
../textdisplay.c     struct  __mptr                   -2    df
rViews/textdisplay.h class   ivTextDisplay            46    df
../textdisplay.c     class   TextLine                 37    df
../space.c           struct  __mptr                   -2    df
e/InterViews/space.h class   ivSpace                  33    df
../table2.c          struct  __mptr                   -2    df
../sensor.c          struct  __mptr                   -2    df
../superpose.c       struct  __mptr                   -2    df
../perspective.c     struct  __mptr                   -2    df
../perspective.c     class   ViewList                 32    df
../aggregate.c       struct  __mptr                   -2    df
../aggregate.c       class   ivAggregateInfo_List     43    df
../aggregate.c       class   AggregateInfo            34    df
../glue.c            struct  __mptr                   -2    df
../message.c         struct  __mptr                   -2    df
InterViews/message.h class   ivMessage                36    df
../layout.c          struct  __mptr                   -2    df
../rule.c            struct  __mptr                   -2    df
de/InterViews/rule.h class   ivRule                   36    df
de/InterViews/rule.h class   ivHRule                  51    df
de/InterViews/rule.h class   ivVRule                  57    df
../painter.c         struct  __mptr                   -2    df
../rubband.c         struct  __mptr                   -2    df
../deck2_6.c         struct  __mptr                   -2    df
.6/InterViews/deck.h class   ivDeck                   36    df
../deck2_6.c         class   Card                     35    df
../tformsetter.c     struct  __mptr                   -2    df
rViews/tformsetter.h class   ivTransformSetter        31    df
```