HEWLETT PACKARD

Using Hypertext in Selecting Reusable Software Components

Michael L. Creech; Dennis F. Freeze; Martin L. Griss
Software and Systems Laboratory
HPL-91-59 R.1
September, 1992

software reuse; software libraries; component selection; hypertext; Kiosk.

Recently, there has been increasing interest in software reuse as a way to improve software quality and productivity. One of the major problems with reusing libraries of software components is helping users effectively select (find and understand) components of interest. This paper explores the use of hypertext to enhance the process of component selection through a prototype system called Kiosk. Included are discussions of the selection process, why hypertext is well suited for supporting selection, and important characteristics of hypertext systems intended to support reuse. Also discussed are how reusable libraries can be structured using hypertext, how these structured libraries can be mechanically built, and how their use enhances the component selection process.

1 Introduction

In recent years, there has been increasing interest in software reuse as a way to improve software quality and productivity. Software reuse simultaneously offers improved time-to-market, increased software quality, and reduced costs for both development and maintenance. As reuse levels increase, costs generally go down while overall quality improves.

Many sociological and technical factors inhibit the successful reuse of software[Tra87, CHSW90]. In our work, we have focused on two of the major technical inhibitors: how users of large reusable libraries can efficiently *select* (i.e., find and understand) software to reuse, and how library builders can *structure* reusable libraries to enable efficient selection.

We believe that the appropriate use of hypertext to structure reusable components can greatly improve a user's ability to find and understand components. Hypertext provides a means to link together all of the related *workproducts* that comprise a module, such as source code, tests, documentation, and design notes, into a conceptual entity called a *component*. This linking provides ready access to all workproducts within a component, as well as to related or similar components. It also allows multiple classification schemes to be overlaid on the data, supporting a number of different ways to browse and search through a library.

At Hewlett Packard Laboratories, we have built a set of hypertext-based tools, collectively called Kiosk, that attempts to augment the selection process. One tool, Cost++, mechanically links together all of the workproducts for a module into a component, and then links the resulting components into libraries which are structured with multiple classification lattices. Another tool provides interactive access to these libraries, allowing users to browse, navigate, perform queries, and create additional nodes and links.

In this paper, we will discuss how hypertext can support the software component selection process through automatic construction of links, structuring of libraries with multiple classifications, and flexible navigation and search techniques.

2 The Selection Process

2.1 User Framework—Our Reuse Process Model

We view the use of Kiosk within the context of this simple reuse process model. *Consumers*[1] make use of pre-written components when building their programs, so they want to select and use components as quickly as possible. They need to find potentially useful software in a library, and then efficiently evaluate it to determine whether or not it meets their needs. They use the browsing, query, and navigation tools of Kiosk to help them select these components. *Producers* create components to be reused by others. *Librarians* manage libraries of components by helping producers package, catalog, classify, and release their components. They also help consumers in selecting components, and then gathering their feedback. Librarians use Cost++ to structure and

[1]From this point forward, we will use the term "user" to mean consumer.

classify component libraries.

2.2 Software Reuse and Component Selection

In this section, we discuss the selection process whereby users attempt to find and evaluate potentially useful software in a library. Our system is aimed at those users who reuse code written by others, and we believe that it can help both novice and expert users in the selection process.

There are a number of factors that can influence how successful a user might be at locating a potentially reusable component. One obvious factor is how well a user might already know a library they are searching. Another factor that applies to both experts and novices is the degree to which they can specify the characteristics of a component they'd like to reuse, since having only a vague idea will probably cause them to examine different information than if they were searching for a known component. Since users have different styles for searching, individual preferences can also affect their success. Some users prefer a graphical view, while others want to see ordered lists of potential components. Furthermore, users' level of experience in software development could determine whether they wanted to see an implementation-oriented view of a library, a functional view, or both. Ease-of-use and familiarity with the browsing tools can also have a definite impact.

2.3 The Application of Hypertext to the Component Selection Process

We have chosen to use hypertext as the base technology for connecting the workproducts that comprise a library. We believe that hypertext is an appropriate and powerful implementation technology for these reasons:

Retrieval techniques: Hypertext naturally supports the application of a variety of techniques for examining a library, including free navigation, distinguished paths, and pre-built filters and queries. It is also conducive to the use of keywords and free-text searching. Since different techniques retrieve different items[FP90], this gives us the ability to empirically determine which retrieval techniques work best for software.

Flexible library structure: Since we are experimenting with how to structure libraries, this approach allows us to modify an existing library structure by simply relinking nodes in different arrangements.

Multiple classifications: A number of different classification schemes can overlay the same nodes, and then be made evident to a user by simply traversing different links.

User customization: Users can modify the existing library structure dynamically to suit their needs, as well as add new kinds of structure. In addition, annotations to a library can be made without disrupting usage of the library.

2.4 Required Hypertext Features for Component Selection

The development of libraries of reusable components imposes certain requirements and constraints on a supporting hypertext system, including:

Open system: Since reuse is only one part of the overall software process, we believe that a supporting hypertext system must:

- Allow for link creation and manipulation by external tools.
- Provide for integration with other environmental tools, such as structured design, configuration management, and versioning tools.
- Maintain the high performance and accessibility expected by users. This means avoiding dependence on implementation strategies that might adversely affect performance or limit the potential audience, such as object-oriented databases.

Non-intrusive linking: When dealing with software, link information should not be embedded in the files in which links appear. The overhead of having to pre-process files to remove link information in order to compile a program or format a design document would make a system less attractive to potential users. Furthermore, much reuse occurs with product or other useful code, where a user may not even have write access to the code. Non-intrusive linking allows users to link to a component even if they do not have write privileges, thereby permitting them to annotate interesting things they've found.

Point-to-point linking: Since most software today is still stored in files, many interesting entities can exist within a single file, such as class definitions and method definitions. Accordingly, a hypertext system for reuse must support linking between these finer-grained entities, thereby allowing users to examine class structures or specific sections of source code that are linked to design notes or documentation.

Content-independence: Since reusable software can be written in any of a number of languages, the hypertext system should not depend on any particular structure in the contents of nodes. It is certainly desirable, however, that the system be able to take advantage of whatever structure might be in a node, whether it be class structure in source code or a particular documentation format.

2.5 Hypertext Features of Kiosk

The Kiosk prototype is written in C++, using InterViews[LVC89] to build the user interface. A major component of Kiosk is a simple, general-purpose hypertext system similar to PlaneText[GDLT86]. This system allows typed[2], non-intrusive links to be created between nodes, where a node is considered to be a *Unix*^{tm} text file plus any links that point into or out of that file. Point-to-point linking is represented by storing character offsets into each node in which a link points. Links are generally bi-directional and stored in "shadow files" associated with the nodes they link, not in a special database. This improves the availability and openness of the system, since standard tools can read and write node and link files.

The editing facilities of this hypertext system allow standard operations, such as inserting and deleting text and links. The point-to-point link information is kept up-to-date as editing is performed. Nodes and links are represented in Kiosk using a *buffer model*, similar to Emacs, where

[2]By typed, we mean that a link contains an attribute that describes the relationship between the nodes connected by the link.

an in-memory representation of them is created using C++ node and link objects. All changes are made to the in-memory objects; when changes are saved, the appropriate nodes and links are stored back into files.[3]

As proof-of-concept, three structured libraries are supplied with Kiosk—Codelibs, InterViews (version 2.6), and Kiosk itself. Codelibs[Pat90] is a generic library of miscellaneous C and C++ functions and classes, and InterViews is a public-domain user-interface construction toolkit written in C++. These libraries have C and C++ workproducts intermixed, including supporting documentation and example programs. The hypertext structure for the libraries is automatically generated from a batch tool called Cost++, which uses a description of the components and desired classifications to create links.

Through the editing facilities of Kiosk, users can annotate the contents of libraries for their own purposes, and we are promoting a methodology that encourages them to provide feedback to librarians that can be incorporated into the libraries. This feedback could include descriptions of attempts to make use of specific components, as well as suggestions about the structure of the libraries themselves.

2.6 Related Work

The use of hypertext for reusable libraries has been proposed by several others, including Latour[LJ88] and Biggerstaff[Big87]. In Latour's SEER system, hypertext is used to link the reusable Booch[Boo87] ADA components, but the system has no means of presenting different classifications of components. Biggerstaff has advocated the use of hypertext to capture most of the textual artifacts of the software process, with particular interest in design documents. Neither provides a means of mechanically generating libraries structured with hypertext—something we believe to be a necessity. Kiosk's clustering of workproducts into components is most similar to the ORT system's[Mon89] class objects, which act as the central organizing entities for connecting software workproducts. Kiosk uses a more general approach, though, where components do not have to be object-oriented.

Kiosk's underlying hypertext system is similar in philosophy to both the PlaneText system and the Sun Link Service[Pea89]. Both advocate an open systems approach and non-intrusive linking. Kiosk's hypertext system is also similar to PlaneText in using file-based storage for links and nodes. However, Kiosk handles point-to-point linking, and it includes more attributes by which links can be manipulated than PlaneText (e.g., filtering by link role).

3 Structuring Reuse Libraries for Selection Using Hypertext

By enhancing library workproducts with more structure and using the appropriate browsing, editing, and query tools, a user's ability to select components can be greatly enhanced. In this section, we discuss how hypertext is used to structure Kiosk libraries, and how these structured libraries are built.

[3]Kiosk has a fairly complex consistency management scheme to ensure that the appropriate set of nodes and links are written out to avoid consistency problems.

3.1 The Form of Hypertext-Based Structured Libraries

The libraries used with Kiosk exist as *structured libraries*—they contain all the workproducts of the original libraries, with the addition of *links* and *structure nodes.*

Links connect workproducts to other workproducts or nodes. All links are typed, using a *role* attribute to describe the relationship between the nodes connected by each link. Links are used to make access to related workproducts easier, and link roles aid in filtering workproducts from different perspectives.

Structure nodes are created to help improve a user's ability to find and understand library workproducts. They help to improve access to workproducts, as well as enhance understanding of what a library contains and how it is structured. Structure nodes come in two forms:

Classification Nodes classify and catalog other nodes. When used with links, they generally form *lattices* or *webs* of workproducts. They can display the structure of a library that would otherwise not be apparent. They are also used to create different views of a library's components.

Component Nodes cluster related workproducts into *components* for easy access. Component nodes usually correspond to the functionality described by a manual page or equivalent documentation. For example, the Codelibs library has a manual page for a set of string routines called `Stringx`. A component node exists for `Stringx` that represents the compilation of all related workproducts into a unique entity—a component—which links together all of the `Stringx` header files, source code, documentation, tests, and configuration files.

Links and structure nodes are used to construct structured libraries by first clustering related workproducts into components, and then placing these components into one or more classification lattices.

3.1.1 Classifications and Views

Components are combined into classification lattices by linking together classification nodes and component nodes. These lattices allow a user to see the components that make up a library from different *views.* Having a variety of views is important because different users may find different views easier to comprehend, and one view may be more useful for a particular task. For example, users can view components based on the function they perform, the objects they manipulate, their implementation structure, or the architecture of the system in which they are used.

We have only begun to explore a few of these views—our current system provides four. The *object* view is based on the objects that are manipulated by library components (e.g., strings, buffers, windows), and the *functional* view is based on the functions components provide within each library (e.g., add, delete, move). The *inheritance* view models the class inheritance of object-oriented components in each library, and the *architectural* view describes the overall systems in which the components appear. A classification also exists that combines all libraries, so that users do not have to know in which library components reside to successfully select them.

Since each view is represented by links with the same role, users can see these different views by filtering links based on these roles. For example, all object view elements are connected by links with the role `object_view`, so users who wish to see components from an object perspective can simply filter the viewpoint based on those links.

To simplify the use and understanding of these different views, they have themselves been placed in a classification lattice, called *Views*, that connects to the root classification node of each classification.

To more clearly understand library structure, Figure 1 shows part of the object view of Codelibs, along with the `Boolean` component node and its associated workproducts. Nodes like `Data_Structures` are classification nodes, and the nodes that radiate out from `Boolean` are workproducts. At the lower left, a component browser displays part of the contents of the `Boolean` component node.

3.1.2 Direct Workproduct Linking

Besides linking workproducts into components and components into views, workproducts are also directly linked together to enhance understanding by having related information quickly accessible. Common relationships of this form include the linking of manual page "see also" sections to the manual pages to which they refer, linking child and parent class definitions in object-oriented code, and linking functions in source code to their descriptions in manuals and header files.

To illustrate direct workproduct linking, the manual page for the `Interactor` component of the InterViews library can be seen at the left of Figure 2. All manual pages related by `see_also` links are shown, and any of these nodes can be clicked on to read the actual manual pages.

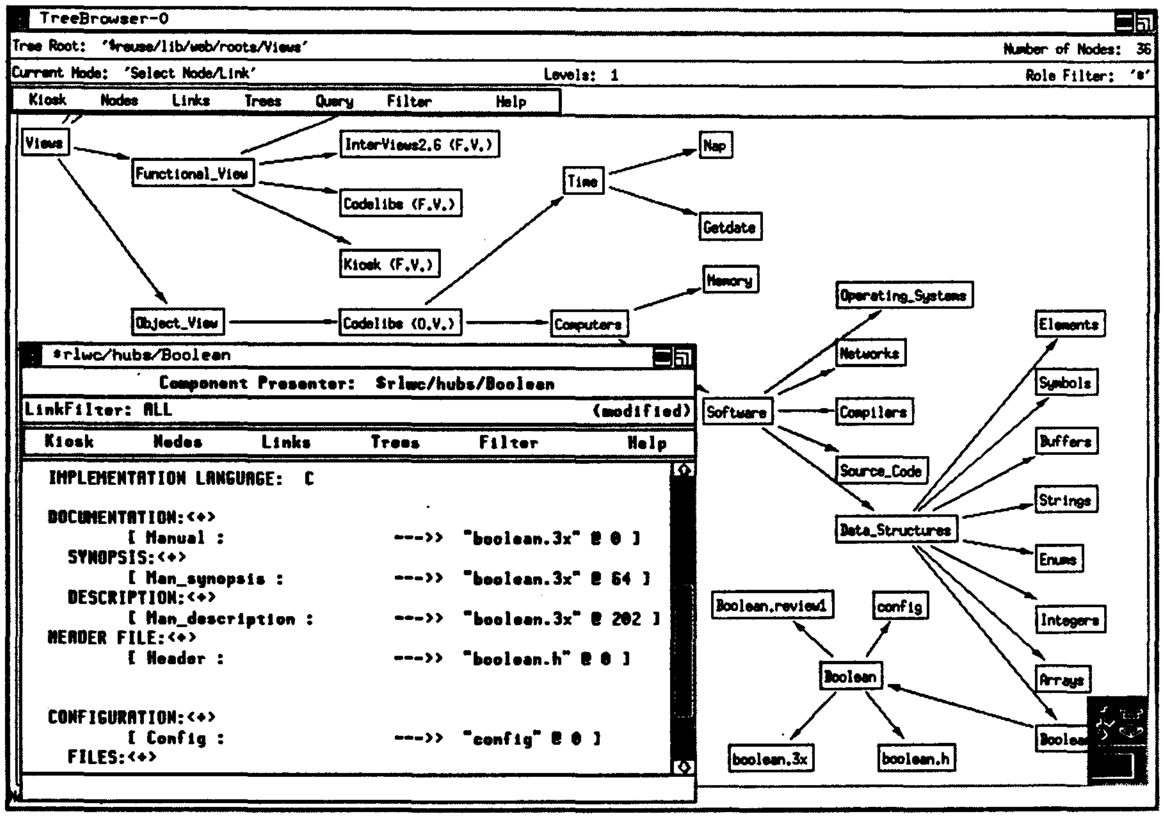
3.2 Building Structured Libraries Using Hypertext

In principle, librarians could build structured libraries manually using standard hypertext editing facilities. We believe this to be impractical, since it would be quite time-consuming and error-prone to interactively build library classification lattices, link all library workproducts, and then repeat this for each release of a library. We have developed a tool, called Cost++, that builds structured libraries from lists of standard workproducts.

Although some work has been done in automatic generation of hypertext structures[FPS89, SB89], it has mostly focused strictly on link generation for text-based documents. Structured libraries, however, require the linking of many types of workproducts, as well as the creation of structural nodes.

Although the main purpose of Cost++ is to build structured libraries for use by other Kiosk tools, it can be used to classify, catalog, and structure virtually any set of text files. For example, Cost++ could be used to structure mail messages, link together on-line documentation, or build a hierarchy chart for an organization. Cost++'s behavior is determined by the declarative information placed in *data files*, which specify:

- Exactly which workproducts should be linked together.



```
Tree Root:  '$reuse/lib/web/roots/Views'                 Number of Nodes:  36
Current Mode:  'Select Node/Link'        Levels:  1      Role Filter:  's'
```

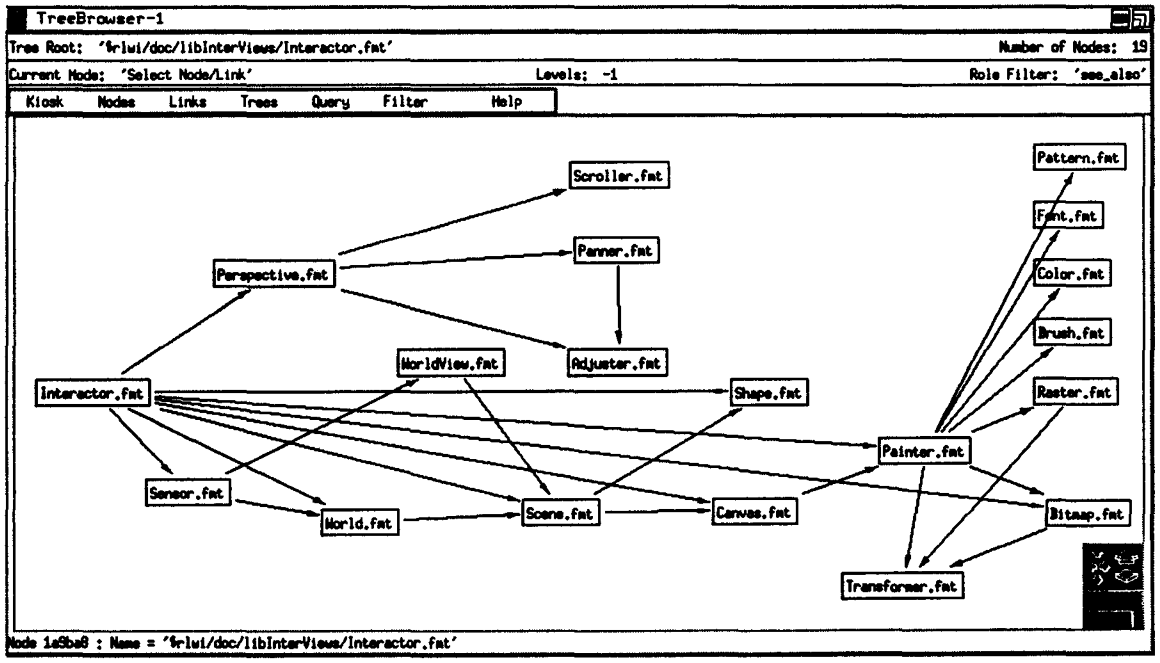
```
            Component Presenter:  $rlwc/hubs/Boolean
LinkFilter: ALL                                        (modified)

  IMPLEMENTATION LANGUAGE:  C

  DOCUMENTATION:<+>
          [ Manual :                 --->>  "boolean.3x" @ 0 ]
    SYNOPSIS:<+>
          [ Man_synopsis :           --->>  "boolean.3x" @ 64 ]
    DESCRIPTION:<+>
          [ Man_description :        --->>  "boolean.3x" @ 202 ]
  HEADER FILE:<+>
          [ Header :                 --->>  "boolean.h" @ 0 ]


  CONFIGURATION:<+>
          [ Config :                 --->>  "config" @ 0 ]
    FILES:<+>
```

Figure 1: Views and Classification, Component, and Workproduct Nodes



```
Tree Root:  '$rlwi/doc/libInterViews/Interactor.fmt'     Number of Nodes:  19
Current Mode:  'Select Node/Link'        Levels:  -1     Role Filter:  'see_also'
Node 1a9ba8 : Name = '$rlwi/doc/libInterViews/Interactor.fmt'
```

Figure 2: Direct Workproduct Linking—"See Also" Links from Manual Pages

- Creation of classification and component nodes to help classify, cluster, and catalog workproducts.
- Where the linked workproducts should be persistently saved.
- Feature Extraction—the ability to find interesting features within workproducts and link them to interesting features in other workproducts or structure nodes.
- New terms that can allow Cost++ to be extensible for structuring and linking new types of workproducts.

These data files can be arbitrarily complex, making use of pre-defined or user-defined functions that take advantage of particular knowledge about a library's domain or implementation language. Data files can also specify connections to other libraries. Cost++ invokes *feature extractors* to link "interesting places" within a workproduct to "interesting places" in other workproducts or nodes. Feature extractors can be built-in functions, programs, or *Unix* shell scripts, allowing them to be written without modifying the Cost++ program. When feature extractors are executed, they are passed the node from which to extract features, and they return possible places to create links. To link source code functions to the manual pages that describe them, we would write two feature extractors: one that returns the locations of source code functions using their function names as identifiers, and another that finds the names and locations of functions defined in manual pages. Cost++ would then use the information returned from these two feature extractors to actually create links between functions with the same identifiers.

Adding new workproducts and changing how existing workproducts are classified, clustered, or cataloged is relatively easily done by restructuring data files and by defining new data file terms. A library of useful feature extractors is provided, as well.

Cost++ was used to generate the structured libraries supplied with Kiosk. It was also used to develop the hypertext-based on-line documentation for Kiosk itself.

4 Selecting Components from Structured Libraries

Now that we have looked at how libraries are structured and built for selection, we will show how these libraries are used. We will start with a high-level view of how Kiosk enhances selection, and then look at the actual mechanics of how components are selected.

4.1 How Kiosk Enhances Selection

Users find components of interest by graphically navigating through the supplied views of the libraries, by direct selection, and by performing text-based pattern matching queries. Users also use the supplied views to better understand the contents of the libraries. The different views created by librarians and producers provide a global perspective of how components relate to one another. Once users learn these views, they can use them to more effectively select workproducts. For example, a user might initially navigate along an object classification of the Codelibs library finding a `Data_Structures` classification node which represents components that manipulate data structures, like "Tables", "Strings", or "Arrays". In later sessions, they could begin by directly reading in this node, thereby gaining immediate access to all the Codelibs data structure components.

In conventional software development systems, a user must locate and review several different files to understand a component well enough to select it. For a user to really understand a component, they might need to view the manual page for a component, its interface description, and a file of sample usages. Through the direct linking of related workproducts into components, their understanding is facilitated because they will not waste time in trying to find exactly where workproduct information is found—they simply follow links between workproducts.

Another important way users understand workproducts is by reading other users' annotations about workproducts, as well as making annotations of their own. Such annotations include how a workproduct works, results of attempts to use a workproduct, and design decisions. Kiosk's annotation facility allows for just such comments. Such annotations could even be used to present a history of design decisions leading up to the current state of a workproduct, instead of just seeing the workproduct in its final form.[4]

A less obvious benefit is how navigating through structured libraries can act as an idea generator.[5] This is an indirect benefit we have found that is best summarized through an example:

> As a user, you might want a way to quickly access items using an index, so you might start by looking for a hashtable in a library. Using navigation, you find a `Data_Structures` classification node that points to a hashtable component, along with some other components that provide indexed access to items. One of these is a B-tree component that you had not thought of as a possibility. After looking at the documentation for some of these components, you might decide that the B-tree component makes the most sense for your application. This, then, makes you think of representing things as trees, and you can then go off in search of tree-displaying components for an interface.

4.2 The Mechanics of Selection

There are a wide range of techniques that can be used to select components. Although work has begun in evaluating different retrieval techniques for reusable components[FP90], it is by no means clear which techniques or combination of techniques are best. We provide several different selection techniques in Kiosk: browsing and navigation, full-text pattern matching, and direct selection.

[4]Although the annotation facility is available for use, the structured libraries that come with Kiosk are not currently annotated.

[5]We believe this occurs much more frequently with navigation than with queries. In the example below, searching for "hashtable" would not likely match with the other data structures available (like B-trees).

4.2.1 Browsing and Navigation

Since one important way for users to select components is by browsing and navigating through classification views looking for components, there are several operations users can perform to show more detail, remove visual clutter, and generally manipulate which links and nodes are displayed.

At any time, a user can graphically view the set of all component views in a directed acyclic graph (DAG) rooted by the node `Views`. Component nodes, classification nodes, and workproduct nodes appear in different, user-customizable colors. Clicking on a classification node in this structure shows a short description of the node. Existing portions of classification views can be expanded to see more detail or pruned to remove visual clutter, and roots of other structures can be added to the display. Existing nodes can also be selected to become the root of an existing DAG structure, thus removing higher level nodes that are no longer of interest.

Any set of linked nodes can be viewed by users by simply specifying a starting node to be the DAG's root and the link roles that should be followed to derive the tree. Users can also specify how many levels (hops from node to node via a link) to traverse to derive the structure. For example, by filtering on the link role `see_also`, a user could view all manual pages that are related to a specific manual page, and then click on any of these nodes to browse a particular manual page of interest (see Figure 2).

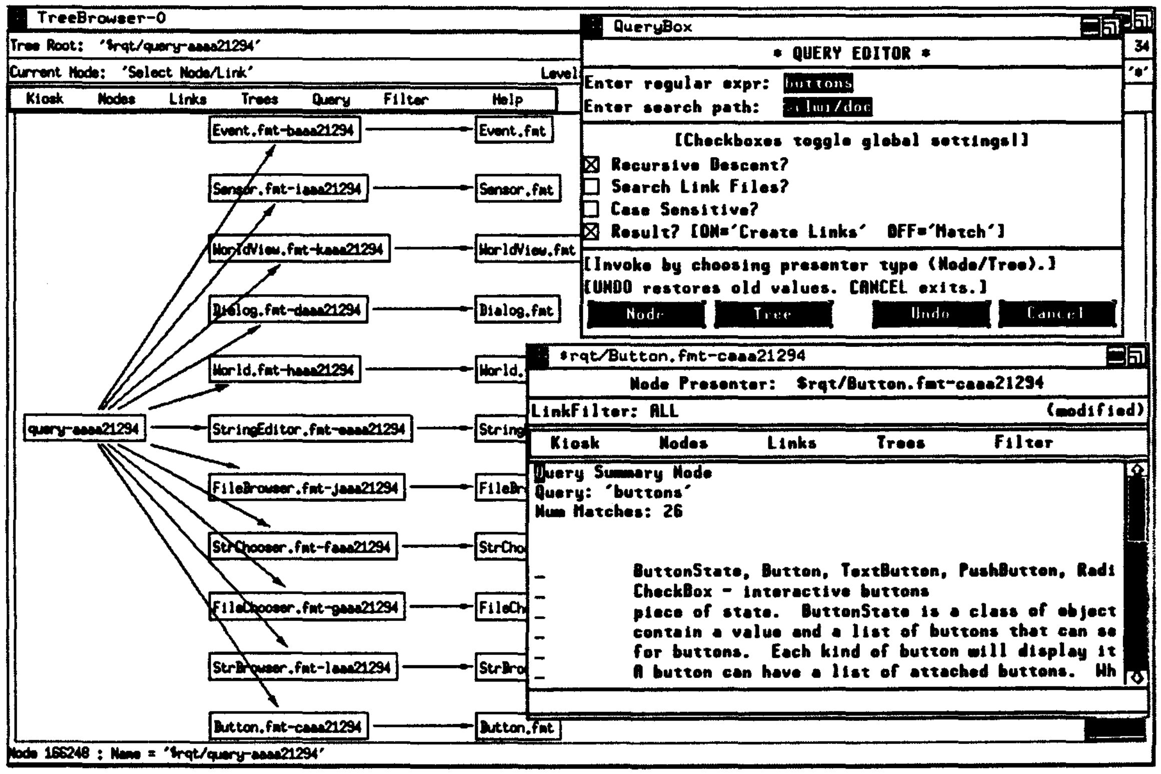
4.2.2 Full-Text Pattern Matching

Although browsing and navigation can be very helpful in locating components of interest on a local to intermediate scale, they are not as effective when searches are conducted on a global scale. For example, a user might want to search all libraries for something that implements "buttons", but navigating around multiple libraries to find such a component could be tedious. For this reason, we have implemented a simple full-text query mechanism that permits users to enter a regular expression to search all nodes designated in one of two ways. We can restrict searches to all nodes accessible from some *Unix* pathname, or to all nodes reachable by walking a hypertext lattice and following only links with a specified role.

The result of a query is returned as a linked hypertext with its own root pointing at selected nodes. The root is the *query head node*, which is linked to a *query summary node* for each matched node. Each summary node acts as a "contents browser" for its corresponding match node. These summary nodes contain the original query, the number of successful matches within the matched node, and a text line from the matched node where each match occurred. When users click on a match line, they are presented with the actual match within the node where the match occurred.

Because the results of searches are hypertext structures in their own right, users can transparently move from the results of a query to the workproducts where matches occurred, and then to the components and classifications to which these workproducts belong. A query hypertext can be saved and restored, used as the starting point for other searches, and combined with other hypertexts via union, intersection and negation operations.

Figure 3 shows the results of searching for the string "buttons" through all of the InterViews documentation. The query head node, in the left center of the figure, connects to 11 query summary nodes. Each of these connects to one node where a match occurred. In the lower right, the query summary node for the manual page match node button.fmt is shown. Each line beginning with an underscore corresponds to a line in button.fmt where a match occurred. When a user selects one of these lines, they are taken to the location of the match within button.fmt. In the upper right, the dialog box used to perform the query is shown.



```
* QUERY EDITOR *
Enter regular expr: buttons
Enter search path: [illegible]lmi/doc
[Checkboxes toggle global settings!]
☒ Recursive Descent?
☐ Search Link Files?
☐ Case Sensitive?
☒ Result? [ON='Create Links'  OFF='Match']
[Invoke by choosing presenter type (Node/Tree).]
[UNDO restores old values. CANCEL exits.]
```

```
Node Presenter:  $rqt/Button.fmt-caaa21294
LinkFilter: ALL                                  (modified)
Query Summary Node
Query: 'buttons'
Num Matches: 26


_        ButtonState, Button, TextButton, PushButton, Radi
_        CheckBox - interactive buttons
_        piece of state.  ButtonState is a class of object
_        contain a value and a list of buttons that can se
_        for buttons.  Each kind of button will display it
_        A button can have a list of attached buttons.  Wh
```

Figure 3: Query Result While Searching for "buttons"

4.2.3 Direct Selection of Components

For users who know both a library and exactly what they are looking for, a *known-item* search is clearly faster than browsing or querying. Accordingly, we offer lists of components that allow users to directly select a known component. These component lists are automatically constructed by Cost++ by connecting all components to a "meta-component". We plan to expand this to allow users to arbitrarily designate any set of items as a group for selection purposes.

5 User Experiences and Lessons Learned

For our initial release of Kiosk, we made a network-installable version available on HP workstations in October 1990. About 50 users within HP installed Kiosk. After receiving little feedback from installation sites, we sent out a questionnaire to find out how Kiosk was being used. Over 50% of the sites responded; however, few had used Kiosk (except 2 local individuals we knew), and none had used the structured libraries. Users had no trouble installing the system, but the time required to figure out how to effectively use Kiosk was longer than installers were willing to spend. Users complained of the lack of higher-level documentation, on-line help, tutorial examples, and general information on how to use Kiosk. Also, from our own use, we found the query interface difficult to use. We concluded that Kiosk was not mature enough to be picked up remotely by novice users and effectively used.

As a result, we focused on working directly with a small group of users in evaluating Kiosk. After updating the documentation, adding some tutorial examples, and upgrading the query interface, we gave Kiosk to several hypertext-naive Computer Science PhD students. After a few days of work with Kiosk, we had them fill out a more detailed questionnaire about using Kiosk. Some preliminary results follow:

- It took 3 days for the students to feel comfortable using Kiosk.

  We expected it would only take an afternoon. Although we had improved the documentation, the students still wanted more task-oriented examples and higher-level documentation. We theorize that the difficulty in learning a new technology (hypertext), aggravated by incomplete documentation, is a main reason for the longer than expected learning time. Anecdotal evidence from coworkers suggests that decreasing the visibility of the hypertext links within the interface might simplify the learning process.

- Getting lost within the hypertext web.

  Initially, the students would easily get lost in the "sea of windows" caused by traversing links to other nodes. They both students requested a way to "go back" to their previous location, and wanted a familiar "home node" to which they could return. The bi-directional nature of Kiosk links did little to help because traversing a link would commonly take the students to where a multi-way branch of links existed. For example, in our on-line documentation, following a reference to a glossary term from the user guide would take the students to a glossary node referenced by many other nodes. We did, however, observe that the longer the students used the system, the less frequently they got lost.

- The structured libraries were indeed useful for selection.

  The students found the object view and architectural view easy to understand and helpful in selecting components. They thought components were easier to select using Kiosk than using "grep" or "man" in *Unix*, or "tags" in Emacs.

6 Future Work

Because mastering the concepts of hypertext was found to be difficult and time-consuming, we will explore the use of interactive demonstrations and training to make the learning task easier. To further focus our work, we will work closely with an appropriate pilot project—a group already using a large library of reusable components. We will bring in and customize Kiosk, help them learn to use it effectively, and adapt it to their reuse process.

Based on the above user feedback and our own experiments, we are exploring a number of extensions. We will improve our hypertext base by adding more types of nodes (executable and graphical) and new types of links (active, one-way, and virtual). We will also add an interactive interface to Cost++ to allow users to augment and restructure existing libraries. Finally, we will add a history mechanism for tracking users' actions, allowing them to return from dead-end searches. This will not only mitigate the lost-in-hyperspace problem, it will also provide a base for experimenting with automatically removing "old" windows to reduce screen clutter.

The current prototype does not provide for multiple users annotating or modifying the same library, or for managing evolving versions of the hypertext. As a consequence we have focused on libraries that change infrequently, such as releases of a system library where a snapshot is taken every few months to be packaged up by a librarian. While this has not been a major problem to date, we have begun to experiment with several users annotating a read-only library, using a single Kiosk "database" shared between several users. This raises issues about versioning of changes, and if and how annotations to a library can be carried over to newer versions of that library. We do not handle the moving of links or annotations from an older version of nodes to a newer version; however, we provide limited support to re-create and repair manually created or changed links after mechanical linking is performed using Cost++.

We plan to improve the selection ability of the query mechanism by using keywords and information retrieval techniques (e.g., removing stop words and building inverted indexes of interesting words) to improve the query mechanism, as well as supplying standard queries. We also plan to provide ways to conceptually specify where to search, rather than using pathnames.

We also want to improve the ability to use other tools with Kiosk by providing a mechanism to programmatically access the hypertext engine. In particular, this will let us use GnuEmacs as the node editor, in place of our enhanced InterViews editing "widget". This approach will also provide basic facilities to help coordinate the interaction of several Kiosk systems, thereby providing better support for multi-user access to a shared hypertext.

7 Summary

Before reuse of library components can be effective, users must be able to find and understand components of interest. Success in the selection process varies according to many factors, including users' experience with libraries, how focused their needs are, individual differences that affect preferences for search strategies, how the libraries are structured, and the availability of different techniques for browsing through libraries.

The use of hypertext for structuring libraries can greatly facilitate the selection of components, but the hypertext system must meet certain criteria: it must be open, links must be non-intrusive, and links must be point-to-point connections. Another important requirement is the ability to generate links and structure nodes mechanically, since the cost of hand-linking large libraries is prohibitive. Given that a hypertext system meets these requirements, its application to software reuse is natural, providing a flexible model, ease of customization, a variety of browsing techniques, and the ability to have multiple classifications overlaying the same libraries.

By linking related workproducts into components, and components into classification hierarchies, finding and understanding are enhanced. The inherent structure of components placed in classifications gives users the chance to see components from a global view, shows them how components are related, and allows them to find similar components. Selection is thereby enhanced by having more information about components readily accessible. The ability to annotate workproducts also improves selection, since usage information about workproducts can be incrementally added, and users can mark interesting things for later retrieval.

Because of the added complexity of learning hypertext technology, extremely good documentation and a good user interface are necessary before a system can be used without direct help. Future work on Kiosk will address these areas.

References

[Big87] Ted Biggerstaff. Hypermedia as a tool to aid large scale reuse. Technical Report STP-202-87, MCC, July 1987.

[Boo87] Grady Booch. *Software Components With Ada.* The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings, Menlo Park, California, 1987.

[CHSW90] Joachim Cramer, Heike Hanekens, Wilhelm Schafer, and Stefan Wolf. A process-oriented approach to the reuse of software components. Memo Nr. 43, University of Dortmund, Helenenbergweg 19, D-4600 Dortmund 50, FRG, March 1990.

[FP90] William B. Frakes and T. P. Pole. Proteus: A software reuse library system that supports multiple representation methods. *SIGIR Forum*, 24(3):43–55, October 1990.

[FPS89] Richard Furuta, Catherine Plaisant, and Ben Shneiderman. Automatically transforming regularly structured linear documents into hypertext. Technical report, University of Maryland, December 1989.

[GDLT86] Eric Gullichsen, Dillip D'Souza, Pat Lincoln, and Khe-Sing The. The planetext book. Technical Report STP-333-86(P), MCC, Austin, Texas, October 1986.

[LJ88] Larry Latour and Elizabeth Johnson. SEER: A graphical retrieval system for reusable Ada software modules. *Third International IEEE Conference on Ada Applications and Environments (Cat. No.87CH2470-3)*, pages 105–113, May 1988.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, pages 8–22, February 1989.

[Mon89] Michael D. Monegan. An object-oriented software reuse tool. A. I. Memo no. 118, MIT - AI Lab, Technology Square, Cambridge, Mass., April 1989.

[Pat90] Parag Patel. Codelibs – About a C++ Code Re-use Library. In *Proceedings of the 3rd International TOOLS Conference*, page 79, 1990.

[Pea89] Amy Pearl. Sun's link service: A protocol for open linking. In *Proceedings of Hypertext'89*, pages 137–146, New York, NY, November 1989. ACM.

[SB89] Gerard Salton and Chris Buckley. On the automatic generation of content links in hypertext. Technical Report TR 89-993, Cornell University, Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, April 1989.

[Tra87] Will Tracz. Software reuse: motivators and inhibitors. *Digest of Papers. COMPCON Spring '87. Thirty-Second IEEE Computer Society International Conference. Intellectual Leverage (Cat. No.87CH2409-1)*, pages 358–363, February 1987.