Supporting Compositional Reuse in Component-Based Web Engineering

Martin Gaedke Jörn Rehse

Telecooperation Office
University of Karlsruhe
Vincenz-Prießnitz-Str. 1
76131 Karlsruhe
Germany
Tel.: ++49 721 690279
{gaedke, rehse}@teco.edu

ABSTRACT

The World Wide Web's anticipated scope as an environment for knowledge exchange has changed dramatically. Without major modifications to its primary mechanisms the Web has turned into a platform for distributed applications. The originally simple and well-defined coarse-grained implementation model of the Web now hinders Web application development. Fine-grained development artifacts, design patterns, and other well-established Software Engineering methods are hard to reuse in the Web after they have found their way into implementation resources. The application of Software Engineering practice to development for the Web, which is also referred to as Web Engineering, and especially the systematic reuse of components for Web-application development at low-costs is a main goal to achieve. This paper presents a systematic approach to code reuse with the WebComposition Repository, which is an essential tool for retrieval and classification of large component sets. The Repository's architecture is crafted to support multiple representation and classification approaches. It facilitates reuse in component-based Web Engineering.

Keywords

Repository, Reuse, Pattern, WebComposition, Component Retrieval.

1. INTRODUCTION

At the beginning of its existence the WWW was seen as a medium for knowledge exchange and proliferation. Roughly nine years have passed and the applications put to use in the Web now have moved a long way from the plain hypermedia system envisioned then. Today (1999) for many organizations the WWW has become the platform of choice when deciding on a system independent way for deploying and running distributed applications. WWW browsers and HTML have become the base technologies for thousands of Intranets allowing members of an organization remote access to workflow and other systems of increasing complexity.

Even though requirements have reached levels of complexity previously known only in "conventional" software development the development process of numerous companies and organizations working for the Web is still mostly ad-hoc and chaotic [8]. In software development this problem has been approached by the introduction of software engineering. Consequently, applying software engineering practice to the Web is widely regarded to be a solution [4, 10, 15]. To denote this approach the term Web Engineering is commonly used but still it remains undefined.

A large gap between the granularity of design models and the granularity of the implementation model of the Web has been recognized to be one of the main reasons for the low acceptance of disciplined development in the community of Web developers [8, 10]. Another argument that made the introduction of a new implementation model desirable are the results of the research on compositional reuse that also stem from the field of software engineering and have resulted in a new sub-discipline: Component-Based Software Engineering (CBSE). CBSE aims at assembling large software systems from previously developed components (which in turn can be constructed from other components). Once again it is the nature of the implementation model of the WWW that limits the systematic compositional reuse of previously developed artifacts (code fragments).

The object-oriented WebComposition model and the XML based WebComposition Markup Language (WCML) that allows defining components at any granularity have been introduced [9, 10]. Using WCML it is possible to represent abstract design concepts in an implementation. WCML facilitates Web application development by means of composing Web applications from components.

A number of components are required to have a reasonable chance of finding a matching component for a given task. Finding that component among all the others has been identified as a major problem of compositional reuse. It has been found that a repository for storing components and aiding their location and retrieval is required to facilitate the reuse of components enough to make it attractive [6, 16]. Such a repository should utilize representations of the components that differ from the information extractable

SAC'00 March 19-21 Como, Italy
(c) 2000 ACM 1-58113-239-5/00/003>...>$5.00

from the components themselves to be able to offer advanced methods for location and retrieval.

To fully exploit the advantages of code reuse when using the WebComposition system a repository for WCML has been designed and implemented. A Key feature of the Repository is its openness towards adding new representation and classification methods for enhanced retrieval capabilities.

In the following chapters a synopsis is given of how WebComposition and the related WCML technology address software engineering issues specific to the Web. Furthermore, the architecture for the WebComposition Repository will be presented. The paper closes with a discussion on different aspects of the Repository's implementation. Finally, conclusions and a perspective of the future work are given.

2. APPLYING COMPONENT TECHNOLOGIES TO THE WEB

The idea behind Component Based Software Engineering (i.e. the construction of software from existing components) has been around for about three decades [17]. Definitions of software components are manifold and we do not feel the calling to supply one ourselves (yet). Instead we will subscribe to the definition given in [20], which requires components to be *self-contained, clearly identifiable artifacts.*

CBSE is said to allow the construction of more complex software at lower costs. It is supposed to lead to easier maintenance and evolution (i.e. a higher flexibility of a software product throughout its entire life cycle), as well as an overall increase of quality if performed systematically [2, 6, 11].

2.1 Applying Component-Based Software Engineering to the Web

In analogy to CBSE the term Component-Based Web Engineering is introduced to denote the construction of Web Applications according to a disciplined process involving systematic compositional reuse.

Definition "Component-Based Web Engineering" (CBWE): The production of Web applications by composing existing components using a defined process that includes systematic reuse of components and of domain knowledge.

Additionally to the common problems of software engineering every field of activity in the subject faces its own sort of specific problems and Web Engineering is no exception. One of the well known problems of software-engineering for the World Wide Web lies in the fact that its resource-based (file) implementation model was never truly intended for the kind of complicated applications that are put to use in the Web today [10]. During the design of Web applications the entities handled by the designers are often defined at a much higher resolution than possible in the actual code produced during the development process. Objects seen as different system parts by the designers will have to be integrated into a single resource while a single design entity may very well appear in different documents of the implementation [10]. Software engineering methods have been brought to the Web through design models especially suited for Web and other hypermedia applications, for example OOHDM[21], RMM[13], and JESSICA[1]. Still, the implementation model remains far too coarse-grained to reflect designs crafted with those powerful methods. While the granularity of the design models used has become finer and finer along with the complexity of the applications developed, the implementation model of the Web has remained as it was in the beginning [4, 10]. By the end of the first decade of the Web a wide gap has opened between the expressiveness of design methods and the semantically deprived Web implementation model. It is always hard and in many cases impossible to represent and maintain abstract design decisions based on higher level concepts in code. Consequently, mapping changes in the design to changes in the implementation becomes a tedious and error-prone task. As the expressive power of the implementation model is not up to the variety of abstract concepts used during design these decisions will get lost on their way to code. As a result a reverse mapping from Web implementations to higher-level (fine-grained) concepts is impossible. The impact of this is that the artifacts used in design cannot be used during any later part of the application's life cycle. Given the evolutionary approach towards development often encountered in the Web this is a particularly undesirable phenomenon. This effect also hinders reuse as the level of abstraction is reduced during the journey from design to code. The actual code is typically more specialized than the design (because of the implementation model).

Summarily, the increasing complexity of requirements has made the coarse-grained implementation model of the Web a burden to developing and maintaining parties.

2.2 WebComposition – a Component-Based Approach to Web Application Development

The WebComposition approach introduced in [10] allows modelling (web) applications from components. It bridges the gap between design and implementation by allowing to capture whole design artifacts in components of arbitrary granularity. The resolution of a component is not preset but can vary depending on the level of detail required by the design concept in question. A component may represent for example an atomic feature such as the font size attribute, a complex navigation structure, implementations of hypermedia design-patterns [20], or simply compositions of other components. In this way, WebComposition supports the bridging of the gap between design and implementation model by offering a high-resolution implementation model relying on code-abstractions. Complete target language resources are constructed by compiling compositions of these components.

The WebComposition model is object-oriented, and consequently supports code-reuse by creating new components from existing ones through inheritance. Inheritance in WebComposition works according to a prototype-instance model as in SELF [22] and not according to the class-based inheritance models well known from languages such as C++ and Java. Prototyping and referencing other components are techniques of code sharing and thus a form of reuse.

The Web Composition Markup Language (WCML) was introduced in [9] to offer a convenient way to define and represent WebComposition components. The WebComposition Markup Language (WCML) is an application of the eXtensible Markup Language (XML) [23] and allows a (tag-based) definition of components, properties, and relationships between components on top of Web Composition's prototype-instance model. XML is the basis for developing a markup language for components because of its good qualities as described in [23]. WCML is platform in-

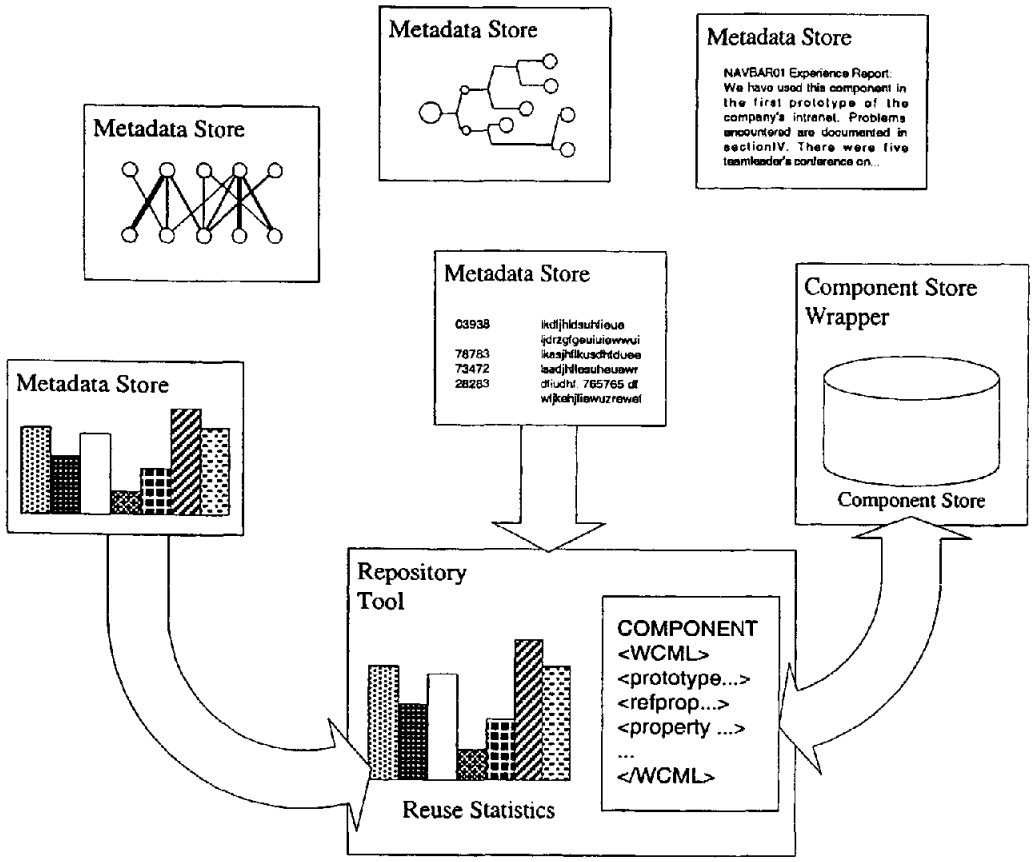

Figure 1: A Repository Tool has created a Context containing two Metadata Stores, a Component Store Wrapper, and itself. Using Metadata the tool gives an augmented perspective of the components

dependent, easy to parse, and it is rigorous in terms of well-formed or valid documents.

3. THE WEBCOMPOSITION REPOSITORY MODEL

It is hoped that growing numbers of components increase the probability that a component fitting a certain purpose exists. On the other hand the difficulty associated with finding such a component also increases with larger numbers of components. As soon as a lot of components are available finding appropriate components becomes one of the main problems of code reuse and of the CBSE approach especially [6]. Component repositories can be an answer to problems posed by a situation in which a human developer cannot be acquainted with all of those components (let alone know all the details about them) [21].

3.1 Providing a flexible Platform

Repositories intended for reuse can employ different methods for the classification and representation of components to improve the chance of finding a component matching a given development problem and to present an augmented perspective of the stored components. The commonly used representation methods usually belong to (at least) one of the following categories: *controlled and uncontrolled indexing* (e.g. [18]) or *methods that contain semantic information* (e.g. [12]). Also *Hypertext-based systems* are mentioned sometimes [5].

In an empirical study [5] Wiliam Frakes and Thomas Pole compared four representation methods belonging to the first category discovering that none of them was inherently superior to the others. The results given by all methods were useful and differed slightly. None of them was perfect. From this Frakes and Pole concluded that a set of reuse entities should be represented *"in as many ways as you can afford."*

Even without considering these results it seems desirable to design a repository in such a way that newly developed classification and representation methods can be added at any time. Especially in a research environment concerned with applying new representation methods (using for example conceptual modeling) to compositional reuse development advances at a fast pace in potentially unforeseeable directions resulting in the need for an extensible, flexible repository.

In the following sections an overview of the Repository Architecture is given. After that the main system parts of this architecture are explained informally.

3.2 The Repository Architecture

There is no single program, which constitutes the Repository as illustrated in Figure 1. Instead the basic mode of operation of the WebComposition Repository is the cooperation of at least three system entities: a Component Store Wrapper (as described above), at least one Metadata Store, and a search or browsing tool *(Repository Tool)*. The tool can utilize the information stored in the Metadata Stores to provide advanced retrieval abilities or it can display information from the Component Store wrapper augmented with additional information provided by the Metadata Stores. Tools will be shaped to work with the information of certain sets of Metadata Stores. A browsing tool for graphs for example could not cooperate with the Metadata Store that contains statistical data. A Component Store Wrapper, some Metadata Stores, and a tool working together are referred to as a *Repository Context*. To the user the Repository is the Repository Context she or he is using.

A major design goal of the Repository Architecture is the possibility of adding Metadata Stores and Tools at any time. This design goal is referred to as *openess*. A brokerage mechanism is provided in the architecture to ensure the fulfillment of this goal.

3.3 Metadata Stores

The Metadata Stores are the key to the functionality of the Repository. They can offer data on the components stored in the Component Store. In this way Metadata Stores can be used to create representations of the Components available that contain information differing from the information contained in the components themselves. The Component Store offers only a "flat" representation of the WCML Components. It provides simple DBMS like query functionality but for example doesn't classify Components neither by functionality nor by any other criteria. Especially in a large set of undocumented components it is hard to find the appropriate component using text search only.

3.4 Repository Tool

The most prominent part of a Repository Context is the Repository Tool. It supplies the user interface and is the part that is directly visible to the user. The user interface may be a GUI or command line interface that is used by a human user but may as well be an API that allows the use of the tool from other programs.

Typically, the tool is a viewer, query tool, or browser for retrieving components, analyzing components, or offering other services using the Repository. Retrieving, sorting or displaying sets of components in a way defined by structures with varying semantics is possible using Metadata Stores. Additionally it is possible for the Metadata Store to manipulate metadata as a reaction to query tool user actions.

4. THE WEBCOMPOSITION REPOSITORY SYSTEM

The WebComposition Repository System is designed to allow access to the same set of WCML components using different classification and retrieval methods. The system allows adding different representations for existing components at any time. In the following chapter a more detailed presentation of how the system parts cooperate is given.

4.1 Repository Architecture Implementation

The architecture requires that the Repository works in the domain of an established component system (e.g. CORBA, COM). This is called the *implementation component system*, because it is the component system (a.k.a. component integration technology) used to implement the Repository. The architecture is independent from which concrete component system is used, if the component system fulfills a certain set of basic requirements. Informally speaking it is assumed that the component system provides unique component handles, which are registered somewhere in the component system. Those components can offer interfaces, which can contain methods. It is also assumed that a component can be instanciated and used from anywhere in the component systems domain if its component handle is known. These requirements are fulfilled by today's popular component systems. Components of the component system that is used to implement the Repository should not be confused with WCML components.

The Component Store Wrapper and the Metadata Stores are components of the implementation component system. The tool does not need to be a component. It is the system part that instanciates the Metadata Store, and the Component Store Wrapper. To enable the tool to contact these components the tool must receive their component handles. The Repository Broker will provide these handles. The broker itself is implemented as a component in the component system. A set of one or more interconnected implementation component system domains in which one and only one component handle for a Repository Broker exists is called a *Repository Domain*. Thus the broker is unique in a Repository Domain.

The tool communicates with Metadata Stores and a Component Store Wrapper through their implementation component system interfaces. The question whether or not a Metadata Store and a tool can cooperate boils down to the question whether or not the Metadata Store supports the interface the tool requires for its work.

The interface of the Component Store Wrapper is standardized. Metadata Stores may offer different interfaces depending on the kind of Metadata stored in them. A Metadata Store that contains statistical data on prior uses of a WCML component will have a different interface than a Metadata Store containing a graph structure.

The Repository Context is a tupel $(T_{IS}, C, \{ M_{1,IO1}, \ldots, M_{n,Ion} \})_R$, where $i \in \{1,..,n\}$ and $IO_i \subseteq IS$ (for all values of i). R denotes the Repository Domain in which the context is instantiated. T_{IS} is the tool that created the context. IS is the set of interfaces which are used by T to exchange data with Metadata Stores in this context. C is a Component Store Wrapper. $M_{i, IOi}$ is a Metadata Wrapper that is used by T. IOi is the set of interfaces supported by M_i that are used in this context. Interfaces that are not used in this context are not listed. The context is created and destroyed by T. It exists only and completely in the Repository Domain R.

4.2 The Repository Broker

In our view the requirement "openness" is satisfied if there is a procedure for adding Metadata Wrappers M_{IO} and Tools to a Repository Domain in such a way that any Tool T_{IS} can always create a Repository Context with this Metadata Wrapper if $IO \subseteq IS$.

The system part responsible for guaranteeing openness is the Repository Broker. It allows selecting Metadata Stores by interfaces.

On installation of a new component in a Repository Domain the Repository Broker's register method is called to notify the broker of the installation. The registration method offered in the broker's implementation component system interface takes the new components component handle as parameter. The broker provides persistent storage of the handles of registered Metadata Stores. As it is necessary that the broker knows about a Metadata Store's interface it must either be able to query those interfaces or they must be passed during registration. A tool can pass an interface (or a unique identifier of an interface, this is a question of implementation) to the broker to request a list of registered Metadata Stores that support that interface. The broker can then supply the component handles of Metadata Stores matching the query. As the implementation component system is required to enable the tool to instantiate a component as soon as its handle is known the tool can now create a Repository Context using the newly registered Metadata Store.

It could be argued that if an implementation component system is used that features its own brokerage mechanism (such as CORBA) the system provided broker could fulfill the tasks of the Repository Broker.

4.3 Metadata Store and Repository Tool Implementation

A Metadata Store is an implementation component system component offering at least one interface. The qualifying criterion that makes a component a Metadata Store is that (at least some of) its interface methods have references to WCML Components as return values or parameters. The Metadata Stores interfaces offer methods that depend on the type of metadata stored. Tools able to work with one of these interfaces can use that Store.

As stated before the primary purpose of these Metadata Stores is to supply the information that is usually associated with a Repository (as in the definition of a repository in [20]). This information is (for example) information on component design, their history, interactions with other components, classification, semantic information, and documentation. The Metadata Store can also be used to store additional indexing-, browsing-, hierarchy-, or any other Meta-information. The Metadata Store brokerage mechanism supplied by the Repository allows adding Metadata for retrieval mechanisms of various (and possibly yet unknown) types.

A *Repository Tool* (tool) is defined as a program (e.g. a standalone program or a component) that creates a Repository Context (by instanciating a Metadata Store that has a matching interface and a Component Store wrapper). It uses references to WCML components that are returned through the interface of Metadata Stores to retrieve the corresponding components from a Component Store.

5. DIFFERENT REPRESENTATIONS OF COMPONENT SETS

In this paragraph some examples of the kind of information Metadata Stores can supply will be given, how a tool can utilize this information, and how the information can be used to access the Component Stores. Although the presented examples are completely different, they are all based on the open approach of the WebComposition Repository.

5.1 Examples

5.1.1 Example 1: Additional Textual Information

A very obvious example is a Metadata Store that contains textual information on components. Usually it is hard to deduce the effects of a component from undocumented code. Third party component suppliers could deliver such Metadata Stores together with the actual components. Together with an appropriate tool these can easily be used to implement an online documentation and help system.

5.1.2 Example 2: Indexing Information

A Metadata Store can also contain representations of the first category of representation methods. The Metadata Store could be filled either by a librarian according to an enumerated or facetted classification scheme or by free-text indexing.

Different user groups could maintain their own index to support their own team vocabulary. In this Repository Architecture user groups could achieve separate sets of indices by working in Repository Contexts using the same tool but different Metadata Stores (that have the same interface).

5.1.3 Example 3: Statistical Information

The interface of a Metadata Store may also offer methods that allow modifying the contents of the Store. A WCML development tool that has the tool role in a Repository Context may report the use of WCML components to a Metadata Store that recording component usage. In this way an organization can gather data on the success of their reuse attempts. An evaluation tool can use these data to produce statistics measuring values like reuse efficiency.

5.2 An Example Tool

To give a more lucid presentation of the concept we want to give a condensed overview of a simple tool that we have implemented, refer to Figure 2. The tool works together with Metadata Stores that contain directed graphs:

Metadata that can be used by the tool to display components defines a directed graph on the set of components. Metadata Stores cooperating with the tool have to offer a (standardized) interface that supports iteration on this graph. Vertices are references to WCML components. Semantics of the weighted edges are not defined in the tool and may differ between different Metadata Stores (offering the same interface). Consequently the same tool can be used to view for example an inheritance hierarchy or a graph based on the conceptual closeness of components. All that is required of the information is that it can be represented as a directed graph.

After receiving a list of handles of Metadata Stores supporting the interface in question from the broker the tool allows the (human) user to select from the list. The selection is used to create a Repository Context and the user can start browsing the graph a portion of which is displayed on the screen. Several display modes can be selected. Vertices (i.e. components) can be chosen and their WCML code can be displayed and exported. It is possible to change the recursion depth that is used when retrieving a neighborhood of the current iterator position. To allow changing the position in the graph more rapidly it is possible to issue queries that are dispatched directly to the Component Store. These queries are made up from Boolean operations and predicates on

WCML component properties. Resulting components, represented with their unique identifier (UUID) – a concept used in WCML, can be jumped to from the current iterator position. The tool can also extract the entire contents of the Metadata Store and create a hypertext structure from it. Obviously, this enables the distribution of metadata along with components over the Web, while the component distribution mechanism is reused from the WCML system.

One way to fill a Metadata Store offering the interface described (and since being able to work with that tool) is by evolution. Evolution of Repositories has been described in [12,14]. At TecO a Metadata Store is being constructed that uses adaptive clustering techniques to create new indices into the set of components. The Metadata Store records the paths the Repository user takes when browsing through components (this requires another Metadata Store). According to the paths taken a genetic algorithm tries to put like components into clusters (and similar clusters into higher-level clusters). This way a tree hierarchy is constructed automatically.

6. CONCLUSIONS AND FURTHER WORK

We have pointed out that the coarse-grained implementation model of the World Wide Web hinders the representation of abstract design concepts in actual code. The resulting gap between implementation and design model is a burden to the use of modern software engineering practices in Web projects. The WebComposition approach with its implementation technology WCML bridges this gap and allows designing for reuse. In component-based software engineering the problem of storing, retrieving, and managing components becomes a center-point of considerations. The WebComposition Repository for storing WCML components has been presented as an approach to address this problem. Because of its flexible nature the Repository Architecture is also suited as a framework for further research on developing and comparing representation and classification methods and their capabilities when applied to retrieval of components for reuse.

Two criteria for the adequacy of a repository were used in [16]. At first a sufficient number of components must be provided, secondly the appropriate code should be easy to locate and retrieve. The existing WebComposition Repository addresses the second issue.

Our current research focuses on more complex search tools and Metadata Stores. One of the main goals is to create Metadata Stores that can store domain knowledge about WCML and the Web environment. Consequently, we also work on a tool that can use this knowledge for retrieving data from the Component Store.

In parallel tools are developed that will acquire bulks of HTML data from the WWW for statistical evaluation and semi-automatic component extraction. As massive amounts of code exist in the Web this seems to be a promising approach to meeting the second criterion of adequacy formulated by Maarek, Berry, and Kaiser. Tools for the extraction of recurring patterns from high level lan-

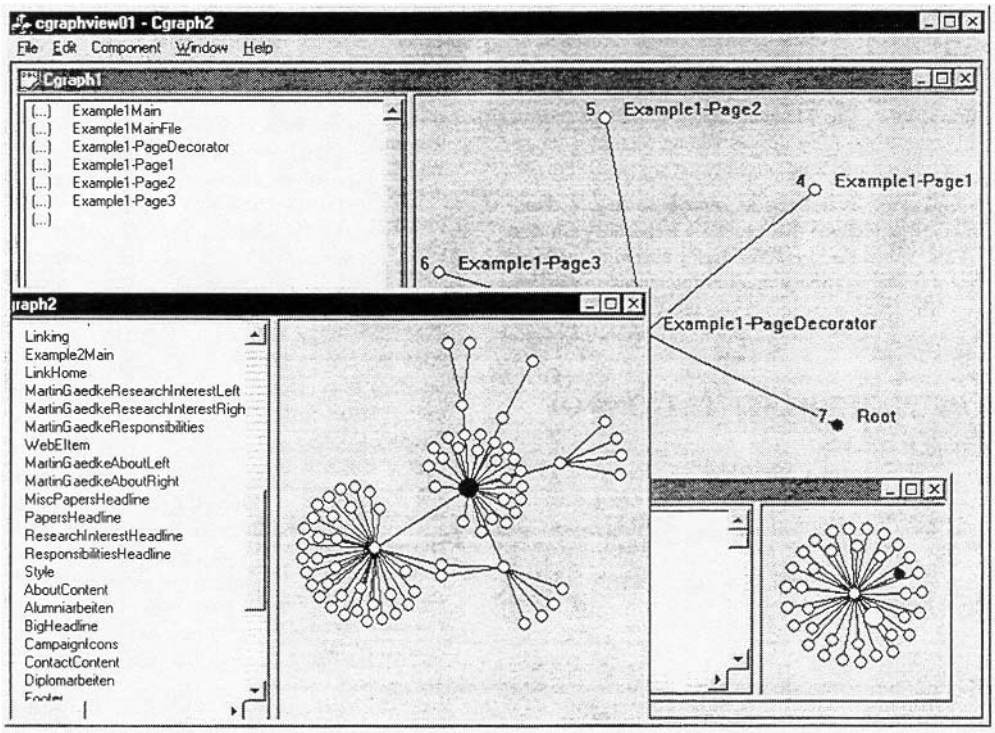

Figure 2 A Repository Tool has created a Context containing two Metadata Stores, a Component Store Wrapper, and itself. Using Metadata the tool gives an augmented perspective of the components

guages exist (such as PEEL for LISP [12]). As the structures of a layout language like HTML are far simpler than those of LISP (for example) we believe that level of human aid required for such an extraction program to work will be lower than in similar tools for "real" programming languages.

7. REFERENCES

1. A. Barta, M.W. Schranz (1998). *JESSICA: an object-oriented hypermedia publishing processor.* In Computer Networks and ISDN Systems 30(1998), Special Issue on the 7th Intl. World-Wide Web Conference, Brisbane, Australia, April 1998, 239-249
2. K. Berg (1997). *Component-Based Software Development: No Silver Bullet*, Object Magazine, 3-97
3. T.J. Biggerstaff, A.J. Perlis (1989), *Software Reusablity*, Volume I, ACM Press
4. F. Coda, C. Ghezzi, G. Vigna, F. Garzotto (1998). *Towards a Software Engineering Approach to Web Site Development.* In Proceedings of 9th International Workshop on Software Specification and Design (IWSSD), Ise-shima, Japan
5. W. B. Frakes, T. P. Pole (1994), *An Empirical Study of Representation Methods for Reusable Software Components*, IEEE Transactions on Software Engineering, Vol. 20, No. 8, pp. 617
6. W.B. Frakes, B.A. Nejmeh (1987). *Software Reuse through Information Retrieval*, In: Proceedings of the 20th Annual Hawaii International Conference On System Sciences, 1987
7. M. Gaedke, M. Beigl, H. W. Gellersen, C. Segor (1998): *Mobile Information Access: Catering for Heterogeneous Browser Platforms.* In: Proceeding of the International Workshop on Mobile Data Access in conjunction with 17th International Conference on Conceptual Modeling (ER98), Singapore, p. 201-212
8. M. Gaedke, H.-W. Gellersen, A. Schmidt, U. Stegemüller, W. Kurr (1999). *Object-oriented Web Engineering for Large-scale Web Service Management.* In: R. H. Sprague (Ed.) Proceedings of the 32nd Annual Hawaii International Conference On System Sciences, Maui, Hawaii, (CD-ROM)
9. M. Gaedke, D. Schempf, H.-W. Gellersen (1999): *WCML: An enabling technology for the reuse in object-oriented Web Engineering.* Poster-Session at the 8th International World-Wide Web Conference (WWW8), Toronto, Ontario, Canada
10. H.-W. Gellersen, R. Wicke, M. Gaedke (1997). *Web-Compostion: an object-oriented support system for the Web Engineering Lifecycle.* In: Computer Networks and ISDN Systems 29, Special Issue on the 6th Intl. World-Wide Web Conference, Santa Clara, CA, USA, p. 1429-1437
11. Capt. G. Haines, D. Carney, John Foreman (1997). *Component-Based Software Engineering / COTS Integration.* Software Technology Review, Software Engineering Institute, Carnegie Mellon University. http://www.sei.cmu.edu/str/descriptions/cbsd.html 01/02/1999
12. S. Henninger (1997). *An Evolutionary Approach to Constructing Effective Software Reuse Repositories*, ACM Transactions on Software Engineering Methodology, 1997
13. T. Isakowitz, E.A. Stohr, P. Balasubramaninan (1998). *RMM: A Methodology for Structured Hypermedia Design*, Communications of the ACM, Vol. 38, No. 8, August 1995, pp. 34-44
14. A. Johnson, F. Fotouhi (1996), *Adaptive Clustering of Hypermedia Documents*, Information Systems, Vol. 21, No. 6, pp. 459
15. A. Kristensen (1998). *Template resolution in XML/HTML.* In Computer Networks and ISDN Systems 30, Special Issue on the 7th Intl. World-Wide Web Conference, Brisbane, Australia, April 1998, 239-249
16. Y.S. Maarek, D.M. Berry, G.E. Kaiser (1991), *An Information Retrieval Approach for Automatically Constructing Software Libraries*, IEEE Transactions on Software Engineering, Vol. 17, No. 8, pp. 800
17. M. D. McIlroy (1968), *Mass Produced Software Components*, Scientific Affairs Division, NATO Software Engineering Conference
18. R. Prieto-Diaz (1987), *Classification of Reusable Modules*, in [3]
19. G. Rossi, A. Garrido, S. Carvalho (1996). *Design Patterns for Object-Oriented Hypermedia Applications.* In: Pattern Languages of Programs 2, Vlissides, Coplien and Kerth (eds.), Addison-Wesley
20. J. Sametinger (1997), *Software Engineering with Reusable Components*, Springer Verlag
21. D. Schwabe, G. Rossi, S. Barbosa (1996). *Systematic Hypermedia Design with OOHDM.* In Proceedings of the ACM International Conference on Hypertext, Hypertext '96, Washington, March 1996
22. D. Ungar, R. B. Smith (1987). *Self: The power of Simplicity*, In: OOPSLA'87 Proceedings, p. 227-242
23. World-Wide Web Consortium (1998). *XML: eXtensible Markup Language.* http://www.w3c.org/XML/