View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Infoscience - École polytechnique fédérale de Lausanne

An Automated, Language-Based Approach to the Creation of Component Libraries

Technical Report DSC/2001/028

Institute for Computer Communication and Applications

Department of Computer Science
Swiss Federal Institute of Technology
1015 Lausanne, Switzerland

Christian Och

Database Research Laboratory
Department of Computer Science
University of Colorado
Boulder, Colorado 80309 - 0430, USA

Tel. (303) 492 - 5964
Fax (303) 492-2844
och@cs.colorado.edu

Richard M. Osborne

Database Research Laboratory
Department of Computer Science
University of Colorado
Boulder, Colorado 80309 - 0430, USA

Tel. (303) 492-4929
Fax (303) 492-2844
rick@cs.colorado.edu

Otto Preiss

Department of Information Technologies
ABB Corporate Research Ltd
5405 Baden -Dättwil, Switzerland

Tel. ++41 (56) 486 80 69
Fax ++41 (56) 73 65
otto.preiss@ch.abb.com

Alain Wegmann

Institute for Computer Communication and Applications
Department of Computer Science
Swiss Federal Institute of Technology
1015 Lausanne, Switzerland

Tel. ++41 (21) 693 43 81
Fax ++41 (21) 693 47 01
alain.wegmann@epfl.ch

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

An Automated, Language-Based Approach to the Creation of Component Libraries

Abstract

A major promise of component-based software engineering is the reduction of application development time and costs by reusing software components. The existence of and access to component libraries, as well as the ability to build such libraries in the first place, is therefore key in a component-based development infrastructure to facilitate the envisioned reuse. We believe that the future demand for component libraries will increase substantially because many companies are about to adopt the software product line ideas. Due to the lack of a general standard for component libraries and because of the restrictions of existing ones, it is very often the case that new reuse libraries are written from scratch to fit a user's specific requirements. An additional problem is the size and the focus of (existing) component libraries or component repositories. With an increasing size and a broadening focus of the library it becomes harder to retrieve the "right" components due to more complex classification structures, which hinders effective reuse. This will lead to a large number of specific libraries, rather than to the global all-encompassing single one. This work introduces a component library description language, which allows for the definition of a component library, its provided functionality, and its associated semantics on a high level of abstraction. The language compiler uses those definitions made in the (XML-based) language to automatically generate a component library, which is a user-defined, customized application, including a web-based user interface and a persistent storage facility to store the components handled by the library. The language approach allows for the rapid development of domain specific component libraries in a time and cost efficient manner and therefore supports the need for fast creation of reuse libraries with minimal up-front investment.

Keywords

Component Library, Software Reuse, Component Based Software Engineering, Description Language, XML

1. Introduction

Although the approaches to software related reuse have changed over time, systematic software reuse is still the most attractive idea to shorten development time, save costs and improve quality. Component-based software engineering (CBSE) is the software community's next attempt on large-scale software reuse. The success of reuse depends on a number of equally important factors (Jacobson et al., 1997): organizational issues, component and system architectures, a more or less stable market environment, existing domain and technology standards, and sustained managerial support. However, reuse will only happen, if the reuse artifacts are carefully designed and readily accessible by others. If the search for and integration of components is more troublesome than developing the required functionality from scratch, reuse will not happen. Similarly, if building the infrastructure for component reuse is overly time-consuming and bothersome, reuse will not happen, too.

One of the key infrastructure elements in a reuse oriented component-based development process is a component library, or sometimes also called component repository. Note our distinctive definition of these two terms later in Section 2.1. However, a component library's main intention is to explicitly support software component reuse, in that it provides easy access to potentially reusable existing software components and related artifacts. It must therefore support a (Web-based) GUI to easily upload, browse, search, and retrieve components, a DBMS to store components and their associated descriptive and classification information, and effective retrieval methods.

Although a library is of central importance and needs to be set up per project, product, or product line, there is no current standard for efficiently building component libraries. With

more software companies embarking on the software product line (SPL) ideas (Bass et al., 2000), the tool environment and in particular component libraries will continue to gain in importance. It is our belief that SPL will increase the need for the creation of specific component libraries with specific features as well as the need for library interoperability. Hence, the process to accomplish both must be automated.

Various reuse library systems of both academic and commercial nature currently exist (a recent survey can be found in (Guo and Luqi, 2000), or as an application example, an existing Internet based software component registry at http://www.componentregistry.com/entrypage.jsp). These systems have in common, that the libraries were developed with a clear picture of what the target artifacts would be. Consequently, the characteristics of the entities to be accommodated by the library were known at library development time and the library structure and functionality is hard-coded with that knowledge in mind. However, besides fundamental differences in the internal architecture, there are also big differences in the functionality and the features provided by those existing component libraries (or reuse libraries), both as seen by the library end user - the application programmer and the librarian - and as seen by the library builder. The different systems provide, for example, different techniques to locate, store, and retrieve components in the system. Another example of the differences among the existing approaches is that some libraries are restricted to a specific component technology and, therefore, the library can handle only components of that specific component type.

Due to the lack of a general standard for component libraries and because of the several restrictions of existing ones, it is very often the case that new libraries are designed and written from scratch. This is not only expensive with respect to both time and money, but also usually done in an ad hoc fashion, which certainly hinders its further evolution and

almost precludes interoperability with other libraries, if the interoperability requirements and inter-operating systems were unknown at library design time.

We propose an automated approach to building libraries and therefore introduce the "Clan" component library description language. It allows for the definition of a component library and its provided functionality on a higher level of abstraction. All aspects of the library (e.g. the retrieval capabilities, including its GUI features) are defined and expressed in the library description language. The specific information about the components that are handled by the library (i.e. component metadata, classification schemas, and component descriptions on different abstraction levels) is also defined in the Clan language. The Clan compiler uses those specifications to generate a component library, which captures the features and semantics of the library, defined in the Clan library definition. The generated component library is a user-defined, customized library application, which provides the functionality expressed in the Clan library definition. This includes a well-defined web interface, as well as an interface to a database management system, which is used to store the components handled by the library. Because of the reproducible nature of the creation process and the uniform architectural solution, such libraries can more easily be enhanced, merged or made interoperable, especially when interoperability standards such as the data model for reuse library interoperability (BIDM) (Browne and Moore, 1997) are automatically accommodated for.

The remainder of this article will first provide the context of our work by giving some basic and important definitions (Sections 2 and 2.1), by showing the relation to other research work (Section 2.2), and by elaborating on the commercial and technical motivation for the presented approach (Section 2.3). Sections 3 and 4 will then discuss the important concepts of the library architecture with its tool support and the component

library features that are supported by the proposed library description language, respectively. Finally, Section 5 concludes by summarizing the main promises of a language based and automated component library creation process and indicating the directions of the future work.

2. Component Libraries and Repositories

Research on component libraries can be seen as one stage in the continuum from early general work on information retrieval, over reuse libraries, to software (code) reuse libraries, and finally to the specialization on component libraries. We see the latter as a specialization of code libraries, because of our working definition of a component, which is the following:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." (Szyperski, 1998).

"…a component is a static abstraction with plugs. By static, we mean that a software component is a long-lived entity that can be stored in a software base, independently of the applications in which it has been used. …" (Nierstrasz and Tsichritzis, 1995).

In our context, the key characteristics of a component are its ability to be deployed independently, its value as a unit of composition, and its ability to be stored in repositories or libraries. Hence, components adhering to the current mainstream component technologies (JavaBeans, EJB, COM/ActiveX) would qualify as components in our sense. By this definition, it is also implicit that component libraries support the notion of component-based reuse as opposed to generative reuse.

The major difference between a component library and a code reuse library is the latter's focus on a special (usually single) programming language and the dependency

on source code. A software component can principally be written in any language and distributed as well as stored in binary form.

2.1 Repository versus Library

The terms *repository* and *library* are used somewhat carelessly in the software engineering community and are often treated as synonymous terms. We mentioned above that component libraries would fall under the category of code reuse libraries. Thus, it is important for us to clearly distinguish their meaning by briefly mentioning our understanding of the major characteristics of these two terms.

Repository: The term is often used in the context of Computer-Aided Software Engineering (CASE) tools or Integrated Development Environments (IDE). It refers to object or component repository products that are either proprietary, dependent or completely independent. These three categories were set forth in (King, 1997) and indicate how tightly a repository product is coupled to a larger application development system, i.e. to a CASE tool or an IDE. Object repositories are intended to keep track of the various development project artifacts (documents, code, images, etc.), and their properties, dependencies, and relationships (in the form of metadata). Object repositories are essentially there to avoid the chaos in software development projects. Prominent commercial representatives are the Universal Repository (UREP) from Unisys Corp. (http://www.unisys.com/marketplace/urep/arch-fab.html), Microsoft's Object Repository (http://msdn.microsoft.com/repository/technical/whatismr/whatismr.asp), or the Build-IT Object Manager from Wallop Software Inc (http://industry.java.sun.com/solutions/products/by_company/0,2343,all-2839,00.html). All of them are independent repository products in that they can be integrated into

development environments through sophisticated application programming interfaces. King (King, 1997) reported the existence of 30 to 35 such repository products in 1997.

Library: Software libraries, code libraries, or code reuse libraries derive their name from traditional book libraries. Note that some research work also uses the term software reuse repositories. The basic idea is to store and retrieve artifacts for the main purpose of reuse. The efficient and effective identification and retrieval of artifacts based on a clever classification system is key. Code reuse libraries are not meant to manage development project artifacts and their dependencies, but mainly to provide potentially reusable software components for the next developer or project to benefit from. As opposed to repositories, libraries are mostly used as standalone products with a direct end user value. Metadata are mainly introduced to improve the classification and description of the components residing in the library.

Hence, our work described in this article aims to improve the building of component libraries.

2.2 Reuse Library Related Research Work

Because information storage and information retrieval are key areas in computer science, a lot of research work exists on information retrieval methods in general, and applied to code reuse libraries in particular. Henninger (Henninger, 1997) not only gives a good overview and explanation of retrieval methods but also many useful references to related work. The major ideas of his approach are described later in this section. Guo et al. (Guo and Luqi, 2000) provide a recent survey on existing commercial and governmental reuse libraries. It is important to mention that although they use the term repositories and libraries interchangeably, their focus is primarily on code reuse libraries, as defined in Section 2.1. Besides describing the purpose of the different products, the

survey compares 15 different library/repository systems based on the following key features: Web integration, CASE tool integration, security control, and the retrieval methods. However, a ranking was not attempted.

In general, the reuse library specific research work found in the literature focuses on particular aspects of libraries, mostly centering on information retrieval methods or information structuring (Damiani et al., 1999). We would categorize them into the following topical clusters:

- Retrieval and information structuring approaches such as facetted classifications, enumerated classifications, free-text, or any refinement or combinations of it.
- Automatic extraction of classification indexing information, including automatic extraction of reusable code fragments.
- Methods that allow finding components that nearly provide the requested functionality (distance functions, query reformulation) as well as to find those components that suffer from an ill-chosen classification index.

Although this article and our work are not primarily concerned with the retrieval methods, it is assumed that an enumerated or facetted classification approach is employed to structure information in component libraries. The often binary or source code nature of components makes a free-text based approach a less interesting choice because the needed linguistic information is missing in the stored artifact. This is at least true, if we assume to have little influence to enforce high quality textual description from our component providers.

Substantial work is found in the area of knowledge management for software development with special focus on reuse. It hardly addresses the specifics of building

libraries but rather concentrates on reuse-based process methodologies and their possible tool support (Henninger and Schlabach, 2001) or metadata modeling.

An active research community has also formed around the area of reuse library interoperability. A focal point is the Reuse Library Interoperability Group (RIG) (Browne and Moore, 1997). The group was formed in 1991 to draft standards, which shall enable interoperability among standalone software reuse libraries. Tangible outcomes are the Basic Interoperability Data Model (BIDM, also standardized as IEEE 1420.1) and formal extensions to it, such as the Asset Certification Framework and the Intellectual Property Rights Framework. The details of these models as well as the beneficiaries and the concrete benefits of interoperability standards are discussed in (Browne and Moore, 1997). We will briefly touch upon some of the benefits when we discuss the motivation for our Clan language.

Finally, a relatively young research field is Web Engineering, with its goal to apply established software engineering processes and methods to the development of Web applications. For that purpose (Gaedke et al., 1999) introduce the WebComposition Markup Language (WCML), which attempts to support a compositional approach to Web application building with a fine-grained component model. Components are code abstractions of any target language. They also introduce a WebComposition repository to facilitate code reuse by providing storing and retrieval functionality for components. However, the building of the repository as such is not an issue for them. Web engineering and the Web application domain are of further interest to us, because the reuse of existing software artifacts is perceived as a natural and professional necessity. For instance, Web page designers live from the reuse of existing images, animations, ready-made scripts, etc.

It is only in (Henninger, 1997) and partly in (Maarek et al., 1991) that we found significant research interest and work that focused on speeding up the development and deployment of an entire library. Henninger's approach is to avoid high initial costs by designing libraries with minimal structuring effort, thereby saving the time for the complex and hard to design structuring schemes. The retrieval techniques, which he employed, are designed such that they compensate for the minimal retrieval structure and that the structures are enhanced over time by adaptive indexing techniques. The approach of Marek et al. is to automate the assembly of software libraries by using information retrieval techniques in order to automatically extract attributes from a set of components and subsequently create a browsing hierarchy for these components based on the attributes.

2.3 Motivation for an Automated Approach to Building Libraries

Similar to the motivation behind Henninger's work (Henninger, 1997), we believe that it is of utmost importance to be able to create and modify component libraries as fast as possible and with as small an up-front investment as possible. The reasons are the following:

- As for every other software artifact, the requirements for a component library are context dependent. If the context changes, the original requirements, which lead to the solution, lost applicability, and thus the solution might not be adequate anymore. Since changing contexts (organizations, user groups, merging of product lines, technology constraints, etc.) are becoming commonplace we must be able to react accordingly.
- Increasingly less time is given to organizational entities and their managers to justify and prove the return on investments. This prohibits large reuse libraries

with sophisticated classification structures because they simply take too long to be built and reuse cannot be shown to pay off.

- Although we agree with the notion of systematic reuse, where the reuse artifacts must explicitly be designed for reuse, it is still the fact that organizations want to reuse what they have designed up to now, which was originally designed without reuse in mind. Therefore, some trial and error in dealing with such elements is inevitable and must be supported through fast and easy to build libraries.
- Practitioners, i.e. developers with a given project deadline, are more interested in having a sub-optimal solution quickly than a close to optimal solution that requires more time to be ready.

Rather than streamlining one single aspect of a library system in order to speed up library development time, our proposition is to substantially automate the overall creation and deployment process of an entire component library, and therefore strive for a rapid library creation process. This shall include the library front-end, the retrieval structure and methods, the database connection and configuration, and the library interoperability features. Our approach is based on the idea of a component library description language (working name "Clan"), which allows for the definition of a component library and its provided functionality on a higher level of abstraction. Chapter 3 will discuss the language features in more detail. But basically, a Clan compiler generates the library, i.e. its Web-based front end, the queries based on the selected retrieval methods, the library structure, and the database connection.

One might argue that the development of a description language is an extra effort that would never pay off. We think not because we assume that there will be many specific libraries, even within the same company or department. Although a single, global library (and be it virtual) sounds like the ultimate goal, it is technologically not feasible and is

also not viable from the business standpoint. With respect to the former, the feasibility is seriously doubted because of the usability problems that occur with component retrieval. For example, it is nearly impossible to reasonably handle the component retrieval based on enumerated classification if the classification space is big. The same holds true for facets because the facet combinations become extremely complex when multiple domains and multiple technologies are captured in the same library (Henninger, 1997). As for the business viability, unless a library service is not a business on its own, it is not conceivable that organizations have a commercial interest to build large and expensive libraries to accommodate components that do not support their own reuse efforts. Also, because this would require a higher up-front investment and might take longer, than just developing the library functionality that suits their own immediate needs.

In addition, the reasons that reinforce our belief in the need for specific and consequently rather many libraries are:

- Library retrieval efficiency is directly related to the classification structure, i.e. the more domain specific the structure, the more efficient and effective the component retrieval. As discussed above, large classification schemes are prohibitive, which tends to keep efficient libraries rather domain specific. If, for instance, the different non-functional aspects of components shall be considered as facets in a classification structure (e.g. QoS criteria), it is evident that different domains have entirely different priorities.
- The usability of a library depends on the classification vocabulary. Communities or even small user groups (e.g. development teams) tend to develop their own jargon, which should be reflected in the library interface to yield effective search results. However, the community language specific interface might not be intuitive for the occasional, community foreign, library user.

- Software components are developed with an "architecture-first" attitude (Bachmann et al., 2000). These architectural characteristics will be reflected in the library structure for effective component retrieval. Accommodating conceptually different architectures in one library would again compromise the library's effectiveness for the average user, who is normally interested in finding components that belong to a certain architectural style.
- Companies that streamline towards software reuse will start to do so in controlled environments, and only if successful, take further and more risky steps. For instance, they might try to achieve synergies by coordinating reuse efforts across department or product line boundary. It is then much more likely that they try to have their libraries inter-operate rather than develop a common library system from scratch.

All the above arguments suggest that we will see many customized component libraries on the basis of projects, product and product families, and per application domains or interest groups.

3. The Clan Component Library Description Language

As described above, the Clan library description language allows for the definition of a component library on a high level of abstraction. The Clan compiler then generates a component library, which captures the definitions expressed in the Clan specification. Note that the output of the Clan compiler is a component library in the form of a user-defined executable application, where no actual code had to be written. The library designer expresses the basic features of the library entirely in the Clan language. This library definition is then used by the Clan compiler to create the actual component library. Hence, the Clan language approach allows the library designer to concentrate on

the details of the semantics of the library without having to worry about implementation details.

The Clan language is an XML-based language and therefore it is possible to use a standard XML tool to create a Clan library definition. Using an XML-based approach provides, as usual, additional benefits: it is, for example, possible to use a standard XML viewer to verify the (syntactic) correctness of a library definition, as well as standard XML parsers as part of the Clan compiler. In addition, it is relatively easy to understand, change, and maintain the Clan library definition, which greatly supports changes to the semantics of the component library, e.g. due to evolving requirements.

As presented in Figure 1, the compilation of a Clan component library involves two different phases (2-phase compilation process). In the first phase the Clan compiler is used to verify the syntactic and semantic correctness of the Clan library definition and to generate Java source code for the component library. Hence, the output of the Clan compiler is not binary code of some kind but source code in another programming language. In our case, it is the source code of a Java package with the name of the component library as defined in the Clan library definition. Therefore, the Clan compiler is classified as a cross-compiler. In the second compilation phase, the host language compiler (a standard Java compiler) is used to generate the component library executable code, i.e., the Java compiler uses the source code for the component library, which was generated by the Clan compiler, to generate the actual Java byte code for the component library in the form of a Java servlet. The generated Java servlet package can now be used like any other Java servlet in the Java development environment.

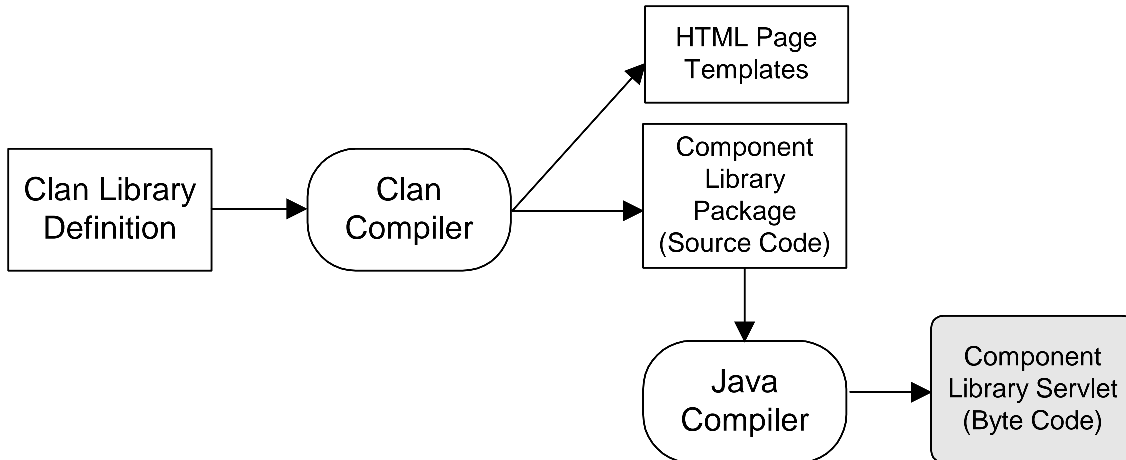

Figure 1 – The Clan Component Library Compilation Process

The Clan compiler also generates templates for HTML pages, which are later used (at runtime) to provide the library's user interface, and a database schema. The latter is in the form of JDBC code (as part of the Java servlet) and is later used to create the database that stores all information handled by the component library. Because the actual deployment of the library system is, among others, dependent on the underlying database product, it is not discussed here. The current idea is to use the JDBC schemas to create the database tables, in a possibly semi-automated way.

In general, a Clan component library is a user-defined application, which provides a web-based user interface for client access. The actual semantics of the library (e.g. user policies, retrieval and search techniques, component metadata definitions), the definition of the presented user interface, and the specification of the persistent storage facilities used by the library are all captured in the XML-based Clan library definition. The different aspects involved in a Clan library definition as well as the language features of the Clan language are described in more detail in Section 4 below.

The output of the Clan compiler, the actual component library application, its functionality, and the general system architecture of a Clan library are presented in the next section.

3.1 The System Architecture of a Clan Component Library

Conceptually, a Clan component library consists of a three-layer architecture as depicted in Figure 2 below. The system architecture of a Clan-based component library consists of the following three architectural layers:

- the user interface layer
- the library layer
- the persistent storage layer

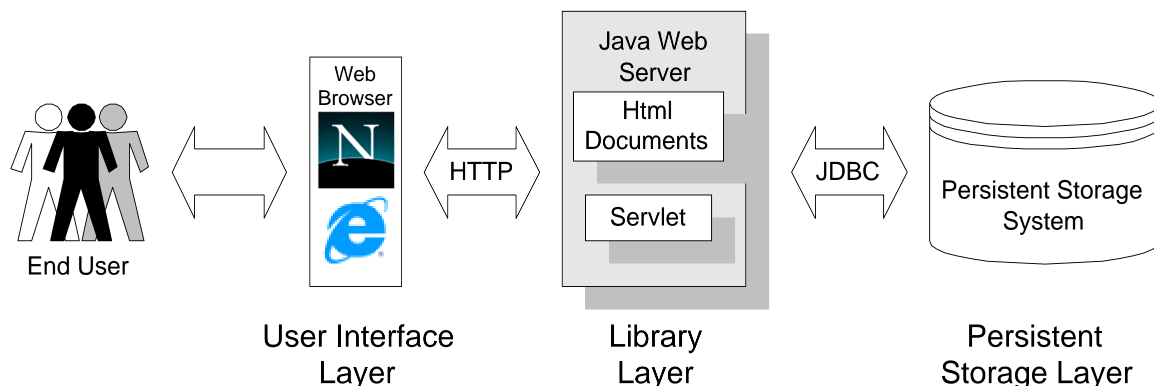

Figure 2 – The Three-Layer System Architecture of a Clan Component Library

The user interface layer. In general, a Clan library provides a well-defined user interface in the form of automatically generated HTML pages, which are used to browse, search, upload, and retrieve components from the component library. The user interface layer therefore consists of various HTML pages, which can be displayed using a

standard web browser. All user interactions with the component library are handled through the provided web interface. Note that, besides some basic static pages, those web pages are produced by the Java servlet, which implements the component library

The library layer. The library layer is the central architectural layer of the system and mainly consists of a Java servlet. All the semantics of the component library is encoded in the Java servlet. The servlet enforces the semantics of the component library such as user policies and search/browsing/retrieval techniques at runtime. In addition, the library layer also handles the interaction with the persistent storage layer and the generation of the user interface in the form of HTML pages as described above. In general, all the specific policies and capabilities of the component library as expressed in the Clan library definition are implemented in the library layer.

As an example for the functionality of the library servlet, consider the handling of a search request issued by a user of the component library: the client sends a search request for specific components through the web-based interface to the library servlet. The library then follows the implemented search techniques/features (as defined in the Clan definition), which will most likely include a query against the persistent storage layer to retrieve the desired information. The library servlet then generates an HTML page, which includes the results from the underlying database system and returns the generated HTML page to the client. Depending on the Clan XML specifications of the component library, the servlet may, for example, return an HTML page that contains the results using a weight system or a hierarchical structure.

As another example of the functionality of the library layer, consider the uploading of a component to the component library: a client of the component library uses the provided user interface to upload the software component she wants to share with the component library as well as additional information about the component (e.g. description of the

component, keywords, or the source code of the component). The library servlet then performs a basic verification of the uploaded information. Based on the success of the verification check, the component along with the other information is then stored in the underlying database system and a response about the success/failure of the upload operation is sent back to the user. Note that the semantics of the verification check performed by the library layer is also defined in the Clan library specification (see 4.2.3 below). In general, the library layer performs a basic verification of the uploaded components and information based on the definitions in the Clan XML specifications of the library.

The persistent storage layer. The persistent storage layer consists of a persistent storage system, which is used to store the components and additional information about the components. As depicted in Figure 2, a JDBC interface is the only requirement for the persistent storage system to participate in the component library. This minimizes the effort to use a standard OTS product (e.g. Oracle) as the library's underlying storage system. Note that most often a relational DBMS will be used as the persistent storage system but any other system thatsupports a JDBC interfacecan be integrated. As described above, the library layer (the servlet) handles all interactions with the storage system through the provided JDBC interface. In addition, the servlet also creates the initial database (i.e. database schema) that is needed to store the components and the other information handled by the component library.

4. Clan Component Library Features - An Overview

As described in Section 3 above, the general system architecture of a Clan component library consists of three architectural layers. This structure is reflected in the structure of the Clan library definition file(s), an XML document. In particular, it consists of three

sections (three nodes in the XML document), each of which represents one of the three architectural layers. Thus, each section in the XML document is used to make the specific definitions for a particular architectural layer. An overview of the different features and attributes that may be defined for a Clan component library is presented below. The discussion of the features is organized in three sub-sections, one for each architectural layer/section in the XML document. A more detailed description of the Clan language, its features and its syntax (including the complete XML schema of the Clan language) is presented in the Clan language reference manual (Och, 2000).

4.1 User Interface Features

The user interface section of the Clan XML document is used to define the general features (a basic “look and feel”) of the component library’s user interface. Its sole purpose is to enable the provision of a basic HTML template, which contains the servlet connections and, if not further modified, can be used as a simple start page. To obtain the latter, some basic definitions can be made, such as background colors, welcome message, inclusion of a logo, etc. Since it is expected that Web page designers want to have the freedom to change the basic appearance based on HTML, the Clan language does not attempt to compete with it. All GUI features that are related to the semantics of the component library (e.g. browsing capabilities, how search results are displayed, etc.) have to be defined in the library section, which is described below. In general, the basic look of the generated HTML pages is defined in the user interface section, whereas the actual content of the pages is defined in the library section of the Clan XML library definition.

4.2 Library Features

The library section of the Clan XML document is the most complex part of a Clan component library definition. The library section allows for the definition of the complete functionality and all semantics of the component library. The main focus in the definition of the semantics and the functionality of the library is on the following issues:

- Library Metadata Definition
- Search/Browse Capabilities
- Component Uploading/Verification
- User Policies and Profiles

4.2.1. Library Metadata Definition

The metadata definition of the information stored in the component library is central to the library's functionality. Clan is kept simple and flexible, i.e., it does not impose restrictive rules on metadata definition. Some basic support is given to define the types of components that may be stored in the library (e.g. JavaBean, ActiveX component, etc.). But Clan is open to accept definitions for different other forms of additional information that may be stored with the component (e.g. plain text description, key words, classification index, etc.). Metadata can potentially be coupled to the user policy mechanism (see below), which would allow selective access to them. In addition, metadata may have importance attributes, which serve to define the importance levels of the information for check-in (upload) purpose. It is, for example, possible to define "critical" information, which must be provided in order to store the component in the library (e.g. the component itself). Other information might be defined as "non-critical" (e.g. a text description of the component), indicating an optional information, which is not

necessary for storing the component in the library. The importance levels are verified by the verification system, which was mentioned above and is described in more detail below.

4.2.2. Search/Browse Capabilities

Clan provides several features for searching and browsing for components in the library. The library designer can choose among the different features and pick the best solution(s) for his specific needs. Some of the provided search functionality is very similar to the approach implemented by the Agora system (Seacord et al., 1998). Clan allows, for example, for the definition of searches for components by their types, their name, or by the structure of their presented interfaces (e.g. method/operation names, parameter/attribute/property names, and event/exception names).

In addition, it is possible to define a keyword search for specific components in the library. The library designer has to define in which part of the metadata of a component (e.g. text description, keyword fields) the keyword search shall be performed. If the keyword search feature is activated, all information about a component (as defined in the library metadata definition described above) may be used to perform the keyword search.

Clan also allows for the definition of how the results of a search are presented to the user and which stored information has to be presented. Optionally, Clan is prepared to define a weights system for the accuracy of the search results, which may be used to present the results in a ranked fashion.

In addition to the query-based search functionality described above, Clan supports the definition of navigational browsing capabilities. Browsing the component library in an navigational manner through a tree-like structure is based on a classification

hierarchy/scheme that has to be defined by the library designer. The library designer may, for example, define the name of the components or the type of the components (or both) as the browsing classification scheme for the library so that the user can browse the components in the library by their name (or their type respectively). It is also possible to define more detailed classification schemes. The details can be found in (Och, 2000).

4.2.3. Component Uploading/Verification

As described above, the library designer may define importance levels for the different information that is stored about a component in the library. That information (more specifically the “critical” information) about a component has to be provided to successfully store or upload a component into the library. It is the library servlet that performs a basic verification of the uploaded information at runtime based on the definition of which essential information about the component must be provided to permit storing (e.g. a text description of the component is mandatory). The library designer may also define additional verification checks. For instance, one could require a minimal component type check against some component characteristics that must be met to be compatible with a certain software product line (e.g. mandatory interfaces).

4.2.4. User Policies and Profiles

Since the balance between automated generation versus deployment/administration effort is a trade-off consideration, the user policy and profiling features are just mentioned for completeness. Hence, it is not yet decided how much of it will go into the Clan language support and how much directly to the deployment infrastructure.

However, the requirements are to support different user levels, which have different access rights to the component library. If a user policy is defined for the component

library every user has to provide a login/password to use the services of the component library. Some users may, for example, only be enabled to retrieve components from the library but may not be allowed to upload information. Other users may only be allowed to browse the library but cannot retrieve any components from the library unless they get granted the right access rights. The requirements are driven by the desire to use such a library as a tool for a component market, to implement online sales (e-commerce) of software components on top of it.

The library shall also provide support for user profiles, enabling the library administrator to gain statistical information about the usage of the library by specific users. In general, the user profile feature enables the library administrator to store additional information about specific users, but we tend to treat the details of this feature outside the Clan language scope.

4.3 Persistent Storage Features

The persistent storage section of the Clan XML document is used to define the general features of the component library's underlying persistent storage system. As mentioned before, the library layer handles all interactions with the database system and also creates the database schema (through the JDBC interface), which is used to store all the information handled by the component library. Therefore, only a few definitions about the storage system itself remain to be defined, namely the few entries that allow the library layer to connect to and to interact with the database system. It consists of:

- The name of the persistent storage system (name of the database system)
- Name of the (empty) database which will be used as the library's underlying database (connection URL)

- The name and the location of the Java class files for the JDBC driver provided by the database system (class name and code base of the JDBC driver)
- The user name and (if necessary) the user's password

5. Conclusions and Future Work

We have presented our motivation and a language-based, automated approach to the fast creation of component libraries. We argued that rather many but relatively small and domain specific component libraries would be used in the near future. This leads us to the conclusion to rigorously automate the creation and modification of relatively simple (compared with metadata repositories) software libraries. We believe that an inexpensive lead-in to the usage of component libraries is necessary (with respect to both lead-time and costs). Especially, because we have not yet fully understood how we best achieve component-based reuse or how reuse processes will be implemented in companies, and consequently, how such libraries must be built to optimally support component reuse. We therefore expect a large amount of probing, i.e. trial and error in employing component libraries. This hardly justifies large up-front investments and prohibits the deployment of heavy and hard to modify library systems. In addition, if reuse efforts do not show early results, sustained managerial support is hardly conceivable.

The underlying pattern of our approach is "abstract - standardize - automate", which, according to (Otway, 1993), is the most promising strategy to master the challenges of software development in the future, if applied rigorously. We therefore introduced the Clan library description language, which is used to describe a component library at a high level of abstraction in a standardized way. The Clan compiler cross-compiles such a library description and produces Java source code (in form of a Java servlet), HTML

templates, and JDBC code. The result is an automatically created component library, composed of the Web front end, the actual library functionality in the middle tier (storing, searching, browsing, retrieval, verification) and a connection to a commercially available persistent storage system. The declarative language approach shall enable non-programmers (e.g. domain experts or librarians) to modify or even create domain specific component libraries.

We are aware that our approach is feasible for component libraries of limited complexity only, but see a clear benefit in the speed and ease to create fully functional and complete, ready to use library systems.

Since our work is in an early stage , the next steps are to finalize the Clan language definition and the development of the Clan compiler. Both are scheduled to be tested in a case study together with a business unit of a globally operating process automation company. In parallel, further research is necessary in order to incorporate the library interoperability feature into the language and the architecture, as well as to optimize the library modification process, which is currently based on a full check-out/check-in approach.

References

- Bachmann, F., Bass, L., Comella-Dorda, S., Long, F., Seacord, R. and Wallnau, K. (2000). Volume II: Technical Concepts of Component-Based Software Engineering, Pittsburgh: Carnegie Mellon University - Software Engineering Institute.

- Bass, L., Clements, P., Donohoe, P., McGregor, J. and Northrop, L. (2000). Fourth Product Line Practice Workshop Report, Pittsburgh: Carnegie Mellon University - Software Engineering Institute.

- Browne, S. V. and Moore, J. W. (1997). "Reuse Library Interoperability and the World Wide Web," Symposium on Software Reuse (SSR) as part of ICSE 97, Boston, pp. 684 - 691.

- Damiani, E., Fugini, M. G. and Bellettini, C. (1999). "A Hierarchy-Aware Approach to Faceted Classification of Object-Oriented Components," ACM Transactions on Software Engineering and Methodology, Vol. 8, No. 3, pp. 215-262.

- Gaedke, M., Rehse, J. and Graef, G. (1999). "A Repository to facilitate Reuse in Component-Based Web Engineering," International Workshop on Web Engineering at the 8th International World-Wide Web Conference (WWW8), Toronto, Ontario, Canada.

- Guo, J. and Luqi. (2000). "A Survey of Software Reuse Repositories," 7th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS), Edinburgh, Scotland, pp. 92-100.

- Henninger, S. (1997). " An Evolutionary Approach to Constructing Effective Software Reuse Repositories," ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, pp. 111 - 140.

- Henninger, S. and Schlabach, J. (2001). "A Tool for Managing Software Development Knowledge (Draft Version)," Submitted to International Conference on Software Engineering (ICSE), Toronto.

- Jacobson, I., Griss, M. and Jonsson, P. (1997). Software Reuse - Architecture, Process and Organization for Business Success, Addison Wesley.

- King, N. (1997). "Overcoming the Object Onslaught," Internet Systems, April.

- Maarek, Y. S., Berry, D. M. and Kaiser, G. E. (1991). "An information retrieval approach for automatically constructing software libraries," IEEE Transactions on Software Engineering, Vol. 17, No. 8, pp. 800 - 813.

- Nierstrasz, O. and Tsichritzis, D. (1995). Object-Oriented Software Composition, Prentice Hall, pp. 5.

- Och, C. (2000). The Clan Component Library Description Language: Reference Manual, Boulder. (Draft version available as of Nov. 2000 on http://www.och.net/Clan.html)

- Otway, D. (1993). Abstract and Automate or How to stop burying your head in the software, Cambridge: ANSA.

- Seacord, R. C., Hissam, S. A. and Wallnau, K. C. (1998). Agora: A Search Engine for Software Components, Pittsburgh: Carnegie Mellon University - Software Engineering Institute.

- Szyperski, C. (1998). Component Software - Beyond Object-Oriented Programming, Essex: Addison-Wesley.