Reuse Frequency as Metric for Component Assessment

Till G. Bay[1] and Karl Pauls[2]

[1] Eidgenössische Technische Hochschule Zürich,
Chair of Software Engineering,
ETH Zentrum, RZJ 22, CH-8092 Zürich, Switzerland,
bay@inf.ethz.ch

[2] Freie Universität Berlin,
Fachbereich Mathematik und Informatik,
Takustr. 9, D-14195 Berlin, Germany,
pauls@inf.fu-berlin.de

Abstract. In component based software engineering, the quality of the produced software directly depends on the quality of the components involved. As component quality measurement is gaining attention, discovery of good quality components is an advancing topic. This paper presents a new metric for component assessment. The contribution of this paper is threefold. First, we define the Reuse Frequency of a component. We observe how Reuse Frequency correlates with component quality. Second, we present a Component Assessment System, we use discover and assess components automatically. Third, we introduce the Component Graph we use to relate components to each other. Applying our technique to a large component repository allows to classify the found components according to their Reuse Frequency.

1 Introduction

Component orientation is a current trend for creating modern applications that increasingly center around component technologies. Component based software engineering is applied in almost all areas of application development including distributed systems, ubiquitous computing, embedded systems, and client-side applications. The concept of a component includes any unit of modularization ranging from a class file to plug-ins of an application. In that respect our definition of a component is a superset of Szyperski's where he states that a component is an independently deployable executable unit of composition [1] . Additionally, software reuse is one of the main benefits of component based software development. More specifically, in most cases it is due to component based software development that reuse can take place.

The ability to compose a component is related to its ability to express dependencies on other components. Dependencies describe prerequisites for a component that are needed for it to function. Component dependencies may exist at deployment unit level, such as a dependency on a resource like a library, or at instance level, such as a dependency on a service provided by another component instance.

This paper presents an approach to combine discovery of components with the Reuse Frequency of a component as a means of quality assurance heuristic for component assessment. The contribution of this paper is threefold. First, we define the Reuse

Frequency of a component. We observe that Reuse Frequency correlates with component quality. Second, we present a scalable distributed web crawler we use to isolate components found in source repositories and to determine their Reuse Frequency automatically. Third, we introduce the Component Graph we use to relate components to each other. Applying our technique to a large component repository allows to classify the found components according to their Reuse Frequency. The underlying assumption is that the Reuse Frequency of a component directly correlates with a certain degree of quality of the component and/or reliability, respectively. This assumption is based on the fact that a high Reuse Frequency of a component does increase the likelihood of the component to be reused again. Subsequently, this leads to a chain reaction increasing the overall quality of the component as increased usage usually leads to more feedback, more specific bug-reports, and reports about the fitness of use of the component. Hence allowing further improvement and refinement.

2 Reuse Frequency

An important advantage of Component Based Software Engineering is reuse. By reusing existing solutions to problems can one reduce time to market. Components capture these solutions in a way they can be reused easier.
Component dependency is a structural attribute of Component Software. Component dependency represents how a component uses functionality provided by another component. Such a relationship yields a functional component dependency. In order to function a component needs to satisfy the transitive closure of all the components it depends on. Let c_1, c_2 be two components. A dependency of c_1 on c_2 is denoted by $c_1 \rightarrow c_2$.

Definition 1. *Component Rank*

$$r(c) = (1-d) + d(r(c_1)/u(c_1) + \ldots + r(c_n)/u(c_n))$$

Where c is a component that used by the components $c_1 \ldots c_n$, $u(c_i)$ denotes the number of components c_i depends on and $0 \leq d \leq 1$ is a damping factor.

The Component Rank is derived from Google's PageRank [2] equation. Just like PageRank Component Rank gives an approximation of a component's importance or it's quality.

Definition 2. *Reuse Frequency*
To normalize the Component Rank, we use the sum of all Component Ranks in the Graph. This gives the Reuse Frequency that forms a probability distribution.

$$f(c_i) = \frac{r(c_i)}{\sum_{k=1}^{n} r(c_k)}$$

A component has a higher Reuse Frequency if many components depend on it, or if some components depend on it that have a high Reuse Frequency. Intuitively a component that is used by many other components has a higher probability of being reused again. Also components that are used by only few other components, that are highly reused get a higher Reuse Frequency. If a component was not of high quality a highly reused one would probably not depend on it. The recursive propagation of Reuse Frequency handles both dependencies (being used by an important component and being used by many components). Additionally, the Reuse Frequency denotes the importance of a specific component for other components that depend on it. More specific, if a component enables another component to function since it satisfies the other component's only dependency it will gain a relatively higher Reuse Frequency than a component that is only a part of a hole set of other components that are needed to satisfy the dependencies of yet another component. Therefore, our notion also covers the likelihood that a specific component can be useful in a certain environment (i.e., enable as much other components to function as possible).

3 Component Assessment System

To address the challenges involved in assessing components we built a Component Graph that uses the above mentioned Reuse Frequency. By querying the Component Graph a user can get information on how extensively a specific component is used by other components. The user can also find out what other components the component itself depends on and uses. And finally the Component Graph shows what other components are in use in systems that use the component in question. Figure 1 illustrates these three queries a user can ask the Component Graph.

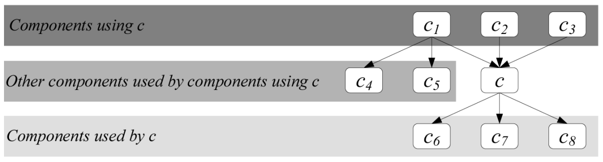

Fig. 1. Component Graph showing three possible queries

The Component Graph is the core of the Component Assessment System we use to improve component assessment. Building the annotated Component Graph involves finding the information about the components, preprocessing it and storing it in the Component Graph. In Figure 2 the overview of the Component Assessment System we use is shown.

3.1 Code Crawler

The first part of the Component Assessment System is the Code Crawler. The Code Crawler addresses the first two issues mentioned above: The finding, processing and storing of all the information we can find about a component while only looking at its source code or meta information. The Code Crawler is a fully configurable, scalable and distributed web crawler. It can crawl the web for files containing source code of any given programming language. By adding filters for specific source code or meta data files one can retrieve semantic information about the component to which a file belongs. The Code Crawler can be configured to crawl source code repositories that can be reached via a URL, or it can also query specialized component repositories like Eureka [3]. Again a configurable filtering mechanism allows to use any source or component repository. Assessing components should not be limited to one component technology and we decided to design Code Crawler to be able to talk to component repositories but to also allow crawling the web for component related information (i.e., source files available in source code repositories). This separation is shown by the redirection of the output of Code Crawler into different component meta data containers after the crawling - see Figure 2.

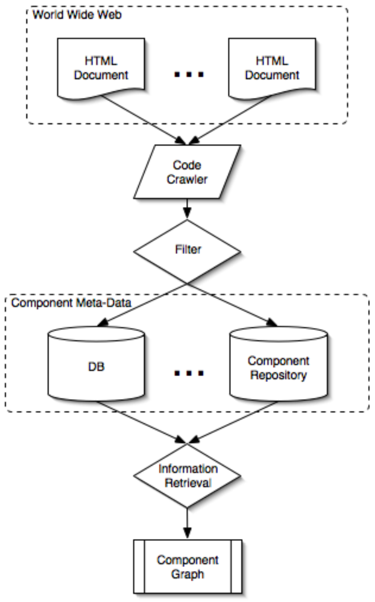

Fig. 2. Component Assessment System Overview

Crawling Source Code Repositories The current implementation focuses on ViewCVS [4], as it is the most widely-used script to publish a source code repository on the web. The Code Crawler stores the crawled and filtered source files in a database. The database is indexed with the Apache Jakarta Lucene library [5] and can be efficiently searched for keywords. Our component assessment approach focuses on information that can be found in a component's source code. However it is also possible to retrieve relevant data

from components that don't make their code available to the public. See Section 6 for suggestions how to handle such components.
For every file type that we download during a crawl, we specify the format of the dependency relation. In addition to the filters that discriminate source files of one programming language from source files from another one, we define a pattern of the dependency relation. If we take the filter that crawls all files ending with *.java, it is straight forward to list all the import statements or the fully qualified class names, that exist in one of the crawled Java files. We identify component dependencies from the collected import statements and the found fully qualified class names.

Component Repositories Component repositories allow communication through an API they provide. Therefore we don't need to crawl them to find information about components. The name Code Crawler is misleading in a situation where we use an API to communicate with a component repository. Nevertheless we decided to include that part of the Component Assessment System in the Code Crawler as well, as it also falls into the information collection phase. See Figure 2 to see how we store the found information in different component meta data containers for the two different cases.

3.2 Component Graph

Like the structure of Html documents that are linked to each other, components depending on each other span a directed graph. The component dependencies can be viewed synonymous to the hyper-links in Html documents. See Figure 1 for an illustration of an ensemble of components that use each other - the resulting graph looks similar to what we know from linked Html documents. The analogy is not complete - it is for example very common to have cycles for the Web Graph while it is seldom for the Component Graph.

Definition 3. *Component Graph*

$$CG = (N,E)$$

where each node $n \in N$ is a component and each edge $e \in E$ is a dependency between two nodes.

See Figure 1 for an illustration of a very small Component Graph.

Weighting the Nodes After constructing the Graph with Components as nodes and Component dependencies as edges, the Reuse Frequency of the components is calculated. The calculated Reuse Frequency is stored along with the Graph nodes. The Component Graph is now complete and can be used for assessing components.

4 Usage Scenario

This section presents how we applied our Component Assessment System to a concrete component repository. The next paragraph briefly introduces the three used technologies namely OSGi [6], Eureka [3], and Gravity [7], followed by the case-study.

The Open Services Gateway Initiative (OSGi) framework and service specification, was defined by the OSGi Alliance to deploy, activate, and manage service-oriented applications dynamically. The OSGi framework sits on top of a Java virtual machine, is an execution environment for services. It defines a unit of modularization, a bundle, that is both a deployment and an activation unit. Physically, a bundle is a Java JAR file containing a single component. After installing a bundle is installed, it can be activated if all of its Java package dependencies are satisfied. Package dependency meta data is contained in the manifest of the JAR file. Bundles can export/import Java packages to/from each other - these are deployment-level dependencies. After a bundle is activated it can provide or use service implementations of other bundles within the framework. A service is a Java interface with externally specified semantics. When a bundle uses a service, an instance-level dependency on the provider of that service is created. Technically, the OSGi service framework can be boiled down [8] to a custom and dynamic Java class loader and service registry that is globally accessible within a single Java virtual machine. The custom class loader maintains a set of dynamically changing bundles that share classes and resources with each other and interact via services published in the global service registry.

Eureka is a network-based resource discovery service supporting deployment and run-time integration of components into extensible systems using Rendezvous' DNS-based approach [9]. Publishing and discovery of components can be performed in both wide-area and local-link (i.e., ad-hoc) networks.

Figure 3 is a conceptual view of the Eureka architecture. Each Eureka server has an associated DNS [10] server, whose resource records the Eureka server can manipulate. An Eureka server has a client API, that allows clients to publish components, discover available components and discover other Eureka servers. Component discovery occurs in the DNS/Rendezvous cloud of the Figure representing the unified local-link and wide-area networks accessible through mDNS [11] and standard DNS respectively.

Gravity [7] is a research project investigating the dynamic assembly of applications and the impact of building applications from components that exhibit dynamic availability, i.e., they may appear or disappear at anytime. Gravity is built as a standard OSGi bundle and provides a graphical design environment for building application using drag-and-drop techniques. Using Gravity, an application is assembled dynamically and the end user is able to switch at anytime between design and execution mode. Eureka was integrated into the Gravity user interface to enable end user discovery of components for integration into his running application. Figure 4 Gravity's user interface with the context menu, that is used to find components.

4.1 Component Discovery and Deployment Case Study

As mentioned above, the Component Assessment System can communicate with component repositories of a specific component technology (see Figure 2). For this case-

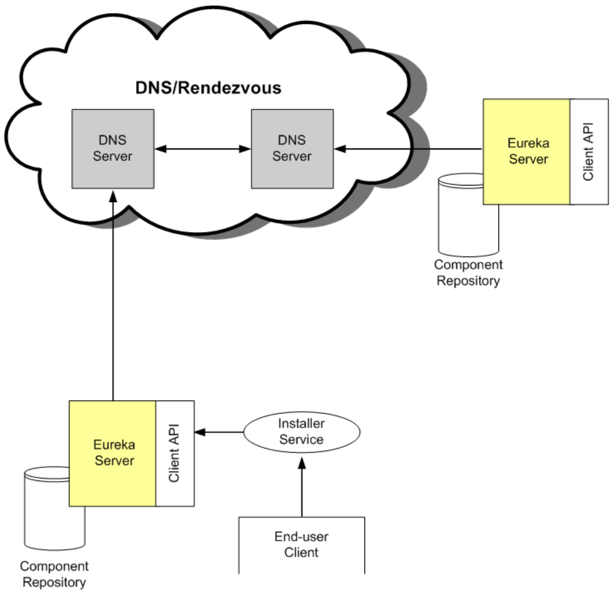

Fig. 3. Eureka Architecture

study we communicate with an OSGi component repository. The discovered components are then fed back into an Eureka network, annotated with their Reuse Frequency. This allows two things: First, we can also feed other OSGi components that we find during a crawl into Eureka and second we get an ordering of the displayed components in Gravity.

Eureka as a Component Meta Data Provider allows to extract dependencies of discovered Bundles. Additionally, existing repositories can also be queried using the Eureka API. In case that an entry point to such a repository is discovered during a crawl, the Component Assessment System queries and retrieves information about published bundles and their dependencies. In the next step, the Component Graph is created showing a network of bundles connected by their dependencies. Since our Component Assessment System is component model agnostic deployment and instance level dependencies can be treated equally in the resulting view.

Component Discovery is enabled via a special filter integrated into the Component Assessment System and applied in case a Bundle is discovered. Subsequently, the filter uses Eureka to extract the dependencies of the Bundle while storing the meta data in an Eureka controlled component repository. The component is now available to clients via Eureka using this repository. Additionally, we write the Reuse Frequency the component's meta data.

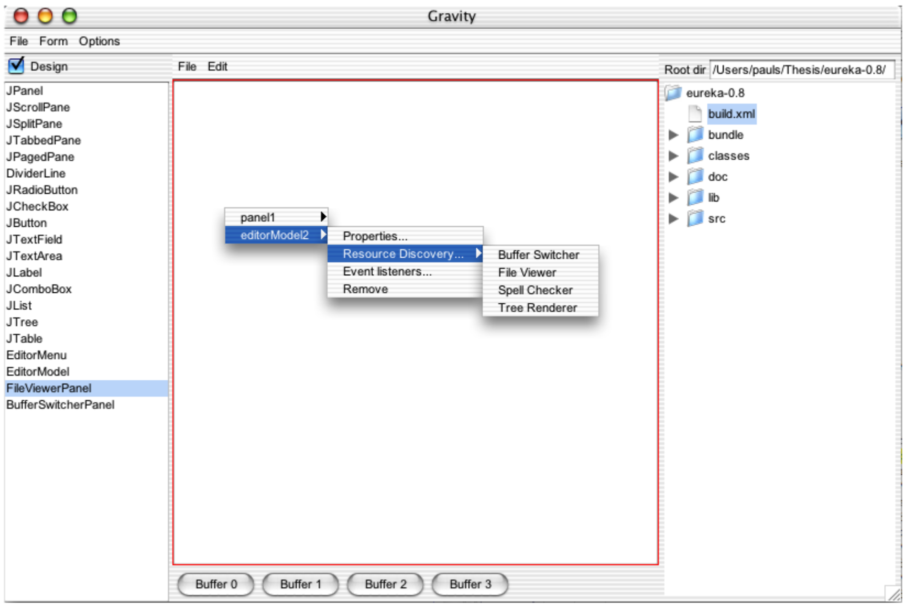

Fig. 4. Gravity showing a list of discovered bundles

Reuse Frequency used as Order Relation in OSGi based applications has been evaluated using the Eureka enhanced Gravity. As mentioned above, Gravity provides a resource discovery that enables the user to extend her application at runtime. For example, an editor component could be extended by a spell-checker or a buffer switcher. More specifically, the dependencies of the underlying component serve as a means of filtering the suggested components. In a situation where the amount of suggested components is small the order of the suggestions has low importance. If many components are found, that resolve a specific dependency, Reuse Frequency is used to order the displayed suggestions. The order of the suggestion list provides the user with additional information. First, it is likely that by choosing one of the more prominent suggestions (i.e., one with a higher Reuse Frequency) over a less prominent one with a similar or equal functionality (e.g., two different spell-checkers are available) the one with the higher importance or quality is chosen. Second, by following the former approach the assembled application will be more extendable since heavily reused components will be added and therefore more suggestions will become available.

4.2 Component Graph Case Study

At the moment two free OSGi R3 [12] framework implementations are available. Both projects provide a small component repository. Both contain the implementation of the OSGi R3 service specification. Oscar [13] from Richard S. Hall is part of ObjectWeb [14] and Knopflerfish [15] is based on the Gatespace GDSP OSGi framework. In order to present the Component Graph and to intuitively validate the assumption that the

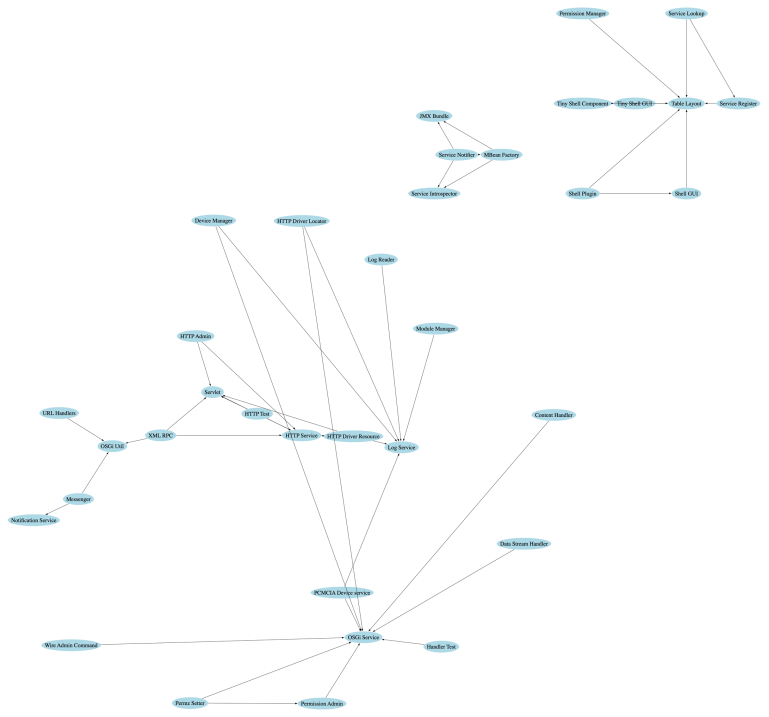

Fig. 5. Component Graph of Oscar's Bundle Repository

visualization of component dependencies combined with the calculation of their Reuse Frequency allows reasoning about importance or quality issues both repositories have been inspected. First we published the components of each repository under a different scope in Eureka. Subsequently, our Component Assessment System retrieved the information about the components together with their dependencies from Eureka and created a Component Graph for each of the two repositories. Figure 6 shows the visualization of Knopflerfish's repository while Figure 5 shows the visualization of the Oscar repository. Currently we are not able to draw a conclusion about a component's quality looking at his Component Graph. However, future work will include empirical analysis of other repositories and focus on conclusions that can be derived directly from the visualizations or the calculated Reuse Frequencies respectively.

Figure 7 shows a subset of the Oscar repository. Table 1 shows the Reuse Frequencies of the Bundles in the Component Graph shown in Figure 7 calculated using a damping factor of 0.85. Due to the Reuse Frequency of the example one can reason about the importance of the participating components. Furthermore, information about the likeliness that a component may function because all of its dependencies are satisfied is conveyed. The two components with the highest Reuse Frequency (JMX Bundle and Service Introspector) are self-contained (i.e., deployable without any assumptions about the availability of other components). One step down the hierarchy the MBean

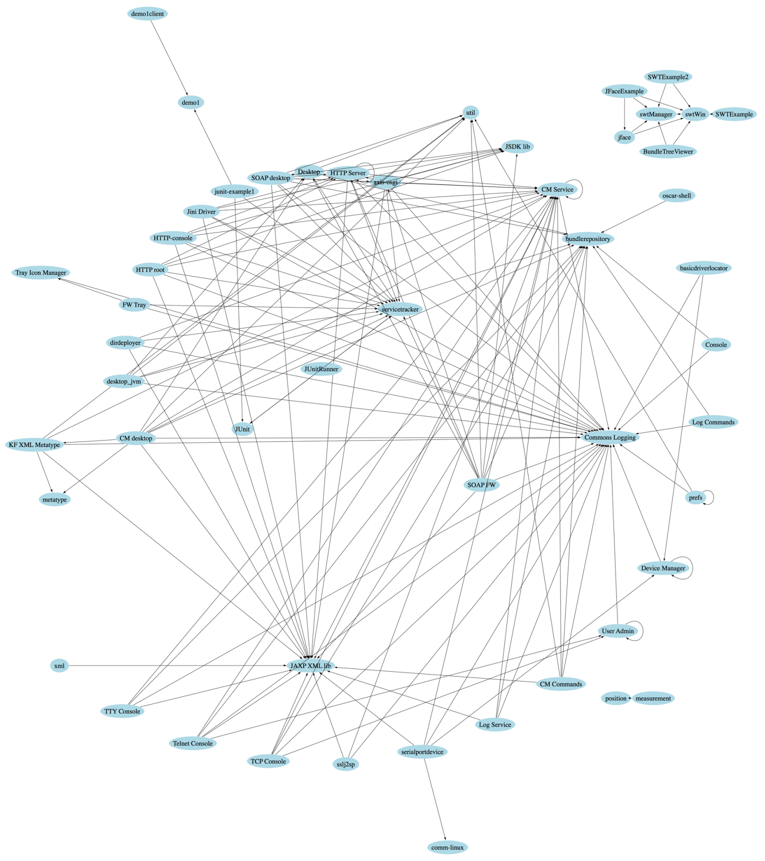

Fig. 6. Component Graph of Knopflerfish's Bundle Repository

Factory can be found - it has dependencies on the two aforementioned Bundles. At last the Service Notifier depends on all of the other inspected components and has the lowest rank, because no other components depend on it. This observations empirically support the intuitive assumption that the Reuse Frequency can be used as an order relation as in our Gravity case-study. Additionally, the given order indicates which component should be of higher quality. A high Reuse Frequency indicates that a larger amount of other components depends on a component and a developer should thus pay more attention to the quality of such a component.

Apart from Oscar and Knopflerfish a third free OSGi implementation exists supporting underlying the last release of eclipse [16]. Eclipse is a kind of universal tool platform, an open extensible IDE for anything and nothing in particular as stated on their web-site. The interesting thing to note however, is that eclipse uses it's own OSGi framework implementation as a plug-in mechanism and provides the possibility to discover, deploy, and dynamically integrate plug-ins (i.e., Bundles) from remote sources. The entry point for the remote repositories is the eclipse web site. Future Work will use

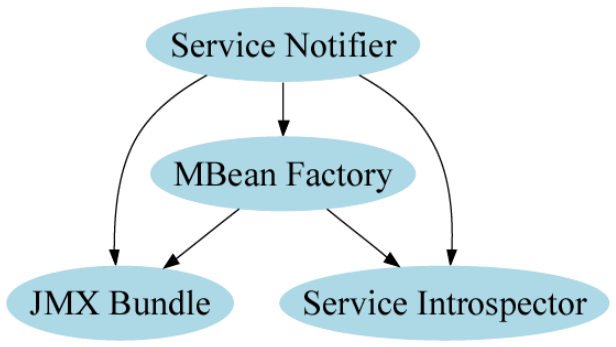

Fig. 7. Partial Component Graph of Oscar's Bundle Repository

Component Name	u(c_i)	r(c_i)	f(c_i)
Service Notifier	0	0.15	0.17
MBean Factory	1	0.19	0.21
JMX Bundle	2	0.27	0.31
Service Introspector	2	0.27	0.31

Table 1. Reuse Frequencies of the Component Graph shown in Figure 7, d=0.85

the presented Code Crawler in order communicate with eclipse's bundle repository and make this huge repository available via Eureka. A Component Graph created using all eclipse plug-ins that can be found this way promises to be an overwhelming source for empirical validations of our assumptions.

5 Related Work

Automatic component discovery is closely related to other search and matching problems such as: text document matching, web search and web service matching. Component assessment on the other hand is related to software quality assurance. Additionally, component repositories like OBR become interesting.

Text Document Matching and classification is a well studied problem in information retrieval. Popular solutions to the problem are based on term frequency analysis [17], [18], [19] In our case term frequency can be used once we extend our automatic component discovery infrastructure to also include component documentation into the assessment process. However it will be a supplementary information source to the dependency relations that we are able to extract from the source code or the component repositories.

Web Search inspires techniques proposed in this paper. We compare component architectures to the world wide web. We suggest addressing the component searching problem specifically by using component specific information. Web search should nevertheless influence component search since for example component documentation is normally deployed on the web.

Web Service Matching In Woogle [20] the authors propose unsupervised matching of web services at the operation level. Web services comply to the notion of a software component and the technologies shown for matching on the operation level can contribute to the information stored in the Component Graph.

OBR The oscar bundle repository [21] is an incubator and repository for OSGi bundles. OBR provides a repository of useful and/or didactic bundles that can be easily deployed into existing OSGi frameworks. It promotes a community effort around bundle creation by increasing the visibility of individual bundles. OBR provides simple access mechanisms for the bundles in the repository. Consequently, there are multiple ways to access the repository bundles namely, web access (via a web-site), programmatic access (via a provided OSGi service) for dynamically deploying repository bundles and interactive access (using the Oscar Shell).

6 Future work and Conclusion

As mentioned before there are a open issues where development of our system continues. In the following we would like to list suggestions for each one of them. We also mention how the system can be extended to become even more general.

Closed source components Our system should be able to access and use the dependency information of closed components. Closed components are components that do not make their source code available on the web.
The strongest argument why getting the information will always be possible is that it is in the nature of components to advertise how it can be composed with other components and what requirements need to be satisfied in order to do so. Therefore the dependency information will also be visible to a tool that is trying to retrieve it automatically. It will merely be an issue of finding out what has to be done in order to access that information. Once the retrieval method is found it will remain an implementation issue to integrate it into the Code Crawler or some other automatic information gathering system.

For example the meta data available with .net assemblies can be used to gather information about component dependencies. The documentation of components can also be taken into account, so that the component relationships described there can serve to enrich the data in the Component Graph.

Clustering and Information Retrieval The second open issue is the clustering of the dependency relations. Since we want to operate on a coarser level of granularity, than the one of a simple class file - we will need to cluster the found dependency relations. Clusters of similar package names - getting back to the previously mentioned example

with Java files indicate that all the files belong to the very same project, library or component.
Smart information retrieval algorithms would also allow to combine documentation search with component search - leading to a general search engine that could list the most widely-used components for a certain functionality.
Bridging the gap between our technology and Woogle's argument level operation matching algorithms it would be possible to find more important components to fulfil a specified purpose. More precisely it would be possible to find the most widely-used implementation of a specific API.

Extension - User Interface Providing a user interface that allows searching the Component Graph for other users of a certain component and for related components will be a very important extension of our system. The challenges will include the following:

– Visualization of component dependencies.
– Visualization of component quality attributes.
– Providing a user-friendly search interface.

Extension - Google API As mentioned above it will be very important to also take a component's documentation into account when trying to assess it's quality or fitness for a purpose. Most of the documentation developers use nowadays is available online. The most natural step is therefore to start integrating information we can access through the Google API into our Component Assessment System.

Conclusion In this paper we presented how Reuse Frequency can be used for component assessment. Together with the Component Assessment System and the Component Graph and the Reuse Frequency calculation it contains, our method can be applied to many different component technologies. Using Eureka as a concrete example we demonstrated how Reuse Frequency establishes an order on the components involved and can be used to compare the importance of different components. The possibility to generalize our method to other component technologies or component information of different granularity makes it attractive for general component assessment. More assessments on large component repositories will show how good Reuse Frequency is as a measure of user perceived component quality.

References

1. C. Szyperski: Component Software: Beyond Object-Oriented Programming. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1998)
2. Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems **30** (1998) 107–117
3. Karl Pauls and Richard S. Hall: Eureka - A Resource Discovery Service for Component Deployment. In: Proceedings of the 2nd International Working Conference on Component Deployment (CD 2004). (2004)
4. ViewCVS: ViewCVS - Official Web Site (retrieved October 2004) `http://viewcvs.sourceforge.net`.

5. Jakarta Lucene: Jakarta Lucene - Official Web Site (retrieved October 2004) http://jakarta.apache.org/lucene/.
6. OSGi Alliance: OSGi Alliance. Official Web Site, http://www.osgi.org (2004)
7. Richard S. Hall and H. Cervantes: Gravity: Supporting Dynamically Available Services in Client-Side Applications. In: Poster paper in Proceedings of ESEC/FSE 2003. (2003)
8. Richard S. Hall and H. Cervantes: An OSGi Implementation and Experience Report. In: Proceedings of IEEEConsumer Communications and Networking Conference. (2004)
9. Apple Computer, Inc.: Rendezvous. Official Web Site, http://developer.apple.com/macosx/rendezvous/ (2004)
10. P. Mockapetris: Domain Names - Concepts and Facilities. RFC 1034 (1987)
11. S. Cheshire and M. Krochmal: Multicast DNS. Internet Draft, http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt (2004)
12. The Open Services Gateway Initiative: OSGi Service Platform. IOS Press, Amsterdam, The Netherlands (2003) Release 3.
13. Oscar Community: Official Web Site (2004) http://oscar.objectweb.org.
14. Object Web: Official Web Site (2004) http://www.objectweb.org/.
15. Knopflerfish OSGi: Official Web Site (2004) http://www.knopflerfish.org/.
16. The Eclipse Foundation: Eclipse Platform - Technical Overview. Technical report, Object Technology International Inc. (2003)
17. Scott Cost and Steven Salzberg: A Weighted Nearest Neighbor Algorithm for Learning with Symbolic Features. Machine Learning **10** (1993) 57–78 http://citeseer.ist.psu.edu/cost93weighted.html.
18. Larkey, L.S., Croft, W.B.: Combining classifiers in text categorization. In Frei, H.P., Harman, D., Schäuble, P., Wilkinson, R., eds.: Proceedings of SIGIR-96, 19th ACM International Conference on Research and Development in Information Retrieval, Zürich, CH, ACM Press, New York, US (1996) 289–297
19. Yang, Y., Pedersen, J.O.: A comparative study on feature selection in text categorization. In Fisher, D.H., ed.: Proceedings of ICML-97, 14th International Conference on Machine Learning, Nashville, US, Morgan Kaufmann Publishers, San Francisco, US (1997) 412–420
20. Xin Dong et Al.: Simlarity Search for Web Services. In: Very Large Data Bases. (2004) 582–599
21. Richard S. Hall: Oscar Bundle Repository - Official Web Site. http://oscar-osgi.sf.net (2004)