GRIDLE Search for the Fractal Component Model

Diego Puppin[1], Matthieu Morel[2], Denis Caromel[2], Domenico Laforenza[1] and Françoise Baude[2]

[1] ISTI-CNR
via Moruzzi 1, 56100 Pisa, Italy
{diego.puppin, domenico.laforenza}@isti.cnr.it
[2] INRIA
2004 route des lucioles - BP 93, FR-06902 Sophia Antipolis, France
{matthieu.morel, denis.caromel, francoise.baude}@sophia.inria.fr

Abstract. In this contribution, we discuss about the features needed by a component-oriented framework, in order to implement a tool for component search. In particular, we address the Fractal framework: we describe its main features and what is needed to perform effective component search in this framework.

1 Introduction

There is a growing attention to component-oriented programming for the Grid. Under this vision, complex Grid applications can be built simply by assembling together existing software solutions. Nonetheless, two main features are still missing: a a standard for describing components and their interactions, and a service able to locate relevant components within the Grid. An important step in the first direction was taken by adopting Fractal as a base for the component model for the CoreGrid project. The use of a common component model will accelerate the convergence of several initiatives in the Grid. Still, the research in tools for an effective component search is lagging behind.

This contribution is structured as follows. After an overview of relevant work in the field of component search and ranking (Section 2), we introduce our vision of a search engine based on the concept of component ecosystem (Section 3). Then, we describe the Fractal framework (Section 4) and the features we need to implement our search strategy over it (Section 6). Finally, we conclude.

2 Related work

Valid ranking metrics are fundamental in order to have relevant search results out of the pool of known components. The social structure of the component ecosystem is, in our opinion, the strongest source of information about component relevance: components that are used by many other trusted components are probably *better* than those that are rarely used. Below, we illustrate different approaches to ranking discussed in the literature.

In [3], the authors cite an interesting technique for ranking components within a set of given programs. Ranking a collection of components simply consists of finding an absolute ordering according to the relative importance of components. The method followed by the authors is very similar to the method used by the Google search engine to rank Web pages: PageRank [1]. In ComponentRank, in fact, the importance of a component is measured on the basis of the number of references (imports, and method calls) other classes make to it within the given source code. Unfortunately, ComponentRank is designed to work with isolated software projects, and it is not able to grasp social relations among independently developed components as we do.

There has been a number of interesting works also in the field of *workflow mining*. The approach we propose is complementary to that presented in [5]: the dynamic realization of a workflow is analyzed in order to discover profiling properties, frequently used activities and so on. This analysis requires access to the actual workflow execution, which is usually kept secret by the application providers. We believe that an approach focused on the static structure of the workflow has a stronger potential.

Another related result was recently presented by Potanin et al. [4]. The authors examined the heap of large Java programs in order to measure the number of incoming and outgoing references relative to each object in the program. They verified that the number of references is distributed according to a power-law curve w.r.t. the rank. Their work is based on the *dynamic* realization of a program. We show that this relationship exists also in the *static* links among classes in their coding.

3 The GRIDLE Search Framework

We would like to approach the problem of searching software components using the mature Web search technology. In our vision, an effective searching and ranking algorithm has to exploit the *interlinked structure* of components and compositions. Our ranking scheme, in fact, will be aware of the context where components are placed, how much and by who they are used.

Figure 1 shows the overall architecture of our Component Search Engine called *GRIDLE*: *Google^{TM}-like Ranking, Indexing and Discovery service for a Link-based Eco-system of software components*. The main modules of GRIDLE are the *Component Crawler*, the *Indexer* and the *Query Answering*.

The *Component Crawler* module is responsible for automatically retrieving new components. The *Indexer* has to build the *index* data structure of GRIDLE. This step is very important, because some information about the relevance of the components within the ecosystem must be discovered and stored in the index. The last module of GRIDLE is the *Query Answering* module, which actually resolves the queries on the basis of the index.

In the next paragraph, we discuss some of the properties of the social network of components, and the way we used it to perform ranking.

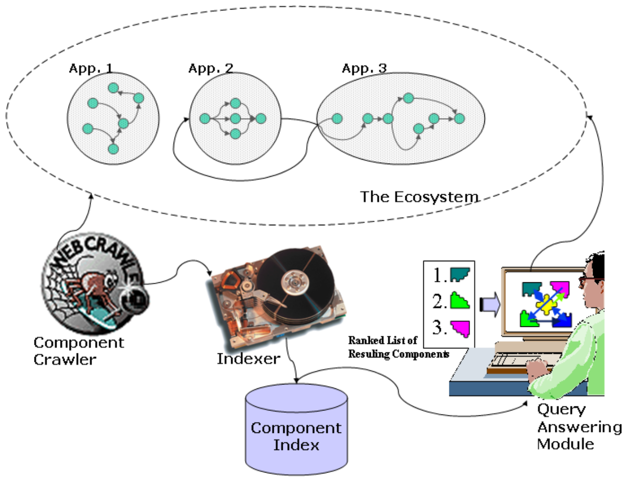

Fig. 1. The architecture of GRIDLE.

3.1 Component Ecosystem: Experiments

In order to explore the characteristic of the component ecosystem, we performed some initial experiments on Java classes. We were able to collect a very large number of Java classes from around the Internet. Clearly, Java classes are only a very simplified model of software components, because they are supported by only one language and they cannot cooperate with software developed with other languages, but they also support some very important features:

1. their interface can be easily manipulated by introspection;
2. they are easily distributed in form of single JAR files;
3. they have a very consistent documentation in the form of JavaDocs;
4. they can be manipulated visually in some IDEs (BeanBox, BeanBuilder etc.);
5. they have a natural unique ID (given by the package name);
6. it is easy to see which class uses which others.

In our experiments, we looked for *documented* Java classes, which can be used by independent developers in their programming tasks: our target were all Java classes with consistent Java Documentation files (JavaDocs). We search the Web looking for JavaDocs. Starting points of our searching were those documents reached through standard web search engine and the documentation site of projects we were aware of. Within the collected JavaDocs, we also found links to external libraries used by the developers.

This way we collected an initial set of 7700 classes, then grown to 49500. We were able to retrieve very high-quality JavaDocs for big software projects, including: Java 1.4.2 API; HTML Parser 1.5; Apache Struts; Globus 3.9.3 MDS; Globus 3.9.3 Core and Tools; Tomcat Catalina; JavaDesktop 0.5, JXTA; Apache

Lucene; Apache Tomcat 4.0; Apache Jasper; Java 2 HTML; DBXML; ANT; Nutch; Proactive; Fractal; Eclipse.

We parsed the JavaDocs files, and we recorded a link between Class A[3] and Class B everytime a method in Class A used as an argument, returned as a result, or listed as a field, an object of type B. This way, we generated a directed graph describing the social network of the Java libraries.

The basic assumption behind this is that a Java programmer behaves somewhat like a general service user: s/he will pick up the most useful service (Java class) out of a large repository of competing vendors/providers (libraries). Thus, s/he will have an eye to the most useful and general classes.

We then counted and plotted the number of inlinks and outlinks from each class. We observed a very interesting power-law distribution (see Figure 2): the number of inlinks, i.e. the number of times different classes refer each class, is distributed following a power-law pattern: very few classes are linked by very many others, while several classes are linked by only a few other classes. This is true both for our initial sample of 7700, and for the bigger collection.

In the figures, the reader can see a graph representing the number of incoming links to each class, in log-log scale. Classes are sorted by the number of incoming links. The distribution follows closely a power-law pattern, a small exception given by the first few classes (Object, Class etc.) which are used by almost all other derived classes to provide basic services, including introspection and serialization.

This is a very interesting result: within Java, the popularity of a class among programmers seems to follow the pattern of popularity shown by the Web, blogs and so on (see [2]). This supports our thesis that methods for Web search can be used effectively also for component search.

3.2 GRIDLE 0.1: Searching Java Classes

Out of our preliminary results, we developed a prototype search engine, able to find high-relevance classes out of our repository. Classes matching the query can be ranked by TF.IDF (a common information retrieval method, based on a metric that keeps into account both the number of occurencies of a term within each document and the number of documents in which the term itself appears) or by GRIDLERank, our version of PageRank for Java classes, based on the class usage links. Our first implementation is available on-line at: http://gridle.isti.cnr.it/.

GRIDLERank is a very simple algorithm, that builds on the popular PageRank algorithm used in Google[1]. To determine the rank of a class C, we iterate the following formula:

$$rank_C = \lambda + (1-\lambda) \sum_{i \in inlinks_C} \frac{rank_i}{\#outlinks_i}$$

where $inlinks_C$ is the set of classes that use C (with a link into C), $\#outlinks_i$ is the number of classes used by i (number of links out of i), and λ a small factor, usually around 0.15.

[3] We added the Interface files to our collection of Classes.

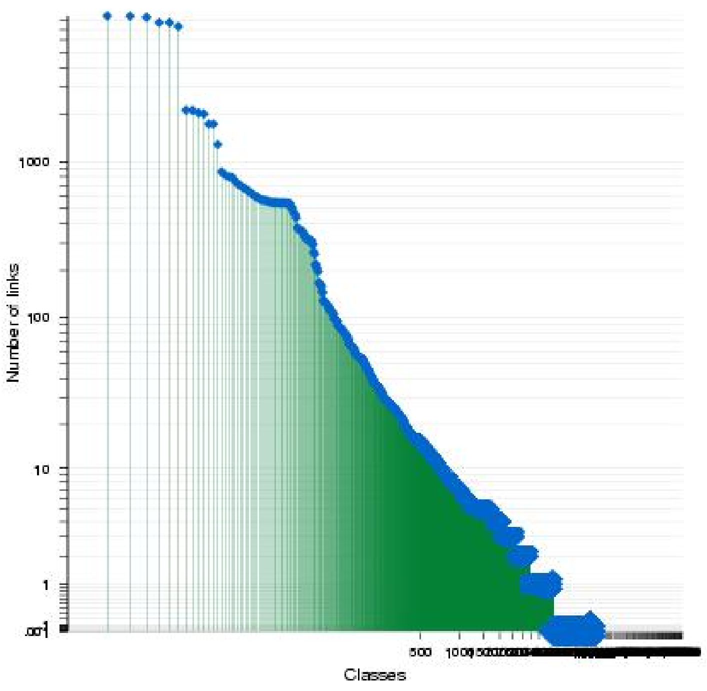

Fig. 2. Distribution of inlinks, 49500 classes.

Interesting Observations We could observe some very interesting results. The highest-ranking classes are clearly some basic Java API classes, such as `String`, `Object` and `Exception`. Nonetheless, classes from other projects are apparently very popular among developers: #7 is Apache `MessageResources`, #11 is Tomcat `CharChunk`, #14 is DBXML `Value` and #73 is JXTA `ID`. These classes are very general, and are used by developers of unrelated applications.

We could verify that in most cases GRIDLERank is more revelant than TD.IDF, especially when the class name is not textually similar to the function we are looking for. For instance, if the developer is trying to write data to a file, and performs a query such as *"file writer"*, TF.IDF will return, in order: (1) javax.jnlp.JNLPRandomAccessFile, from JNLP API Reference 1.5; (2) javax.-swing.filechooser.FileSystemView, from Java 2 Platform SE 5.0; (3) java.io.-FileOutputStream, from Java 2 Platform SE 5.0; (4) java.io.RandomAccessFile, from Java 2 Platform SE 5.0. The second class is clearly unrelated with the problem under analysis, and only the third is probably what the user was looking for.

On the other hand, GRIDLERank will return four classes from the Java API (Java 2 Platform SE 5.0): (1) java.io.PrintWriter; (2) java.io.PrintStream; (3) java.io.File; (4) java.util.Formatter; which are all probably better matches. To verify this claim on result quality, we will need to test the search engine with Java developers.

4 Fractal and the Grid

4.1 The Fractal Component Model

Fractal is a simple, flexible and extensible component model developed by the ObjectWeb consortium. It can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces.

The main design goal of Fractal was to reduce the costs of developing and maintaining big software projects, in particular the ObjectWeb projects. The Fractal model already uses some well known design patterns, such as separation of interface and implementation and, more generally, separation of concerns, in order to achieve this goal.

Recently (June 2005), Fractal was chosen as a basis for the component model for the CoreGrid network. As a matter of fact, Fractal has several features that are fundamental for Grid programming.

– Several different communication patterns can be implemented using Fractal, so to cope with communication latency, including asynchronous communication, subscribe/pulling.
– It allows structural and hierarchical composition, which is a fundamental requirement to build more and more complex applications.
– Sub-components can be shared among components, this way implementing a coherent single-state image of the system.
– Components can be reconfigured, and external controllers can be used to wrap a component.

Currently, the Fractal project is a big effort, organized into four sub projects:

1 The Component Model sub project deals with the definition of the component model. The main characteristics of this model are recursivity (components can be nested in composite components - hence the ”Fractal” name) and reflexivity (components have full introspection and intercession capabilities). The Fractal model is also language independent, and fully modular and extensible.
2 The Implementations sub project deals with the implementation of Fractal component platforms, which allow the creation, configuration and reconfiguration of Fractal components. Julia, the reference implementation, is developped in this sub project.
3 The Components sub project deals with the implementation of reusable, ready to use Fractal components, such as protocol or Swing components.
4 The Tools sub project deals with the implementation of Fractal based applications dedicated to Fractal, such as tools to define component configurations.

One of the core principle behind Fractal is the so-called *separation of concerns*, this means the need to separate, into distinct pieces of code or runtime entities, the various concerns or aspects of an application: implementing the service provided by the application, but also making the application configurable, secure, available etc. There are three main aspects:

1. Separation of interface and implementation.
2. Component oriented programming.
3. Inversion of control.

The first refers to need to separate the design and implementation concerns. A component must stick to a precisely defined *contract*, at syntactic and semantic level. The component contracts must be designed with care, so as to be the most stable as possible w.r.t. internal implementation changes: interfaces must deal only with the services provided by the components — not implementation or configuration of the components. Configuration concerns are to be be dealt with separately, as well as other concerns such as security, life cycle, transactions...

The second means the separation of the implementation concern into several composable, smaller concerns, implemented in well separated entities called components. Wrapper components can add levels of security, authentication, reconfiguration etc. A wrapper component can take an existing component, wrap it, and export the interface to a similar component, with added features. For instance, a wrapper component can take a sequential component and launch it in another thread, offering explicit control of its execution (start, stop etc.).

As an implementation optimization, wrappers can be implemented *inline* by rewriting the Java bytecode. Julia (a Java implementation of Fractal) offers life-cycle controllers and other wrappers in the form of code generators.

The third principle, *inversion of control*, is the separation of the functional and configuration concerns. Components are configured and deployed by an external, separated entity. This way, components can be replaced, updated etc. *at run-time*, not only at design-time.

In other words, a Fractal component is indeed composed of two parts:

1. a content that manages the functional concerns,
2. a controller that manages zero or more non functional concerns (introspection, configuration, security, transactions, ...).

The content is made of other Fractal components, i.e. Fractal components can be nested (and shared) at arbitrary levels.

In order to be used with the framework, a component must stick to a set of design conventions. In particular, within the Julia implementation, it has to implement a set of Java interfaces with the methods for binding other components, export the interface and so on (see Table 1).

When this standard is respected, components can be easily manipulated within the included graphical shell (see Figure 3). The recursive structure of composition is clearly visible.

```
public interface Component {
   Object[] getFcInterfaces ();
   Object getFcInterface (String itfName);
   Type getFcType ();
}
public interface ContentController {
   Object[] getFcInternalInterfaces ();
   Object getFcInterfaceInterface(String itfName);
   Component[] getFcSubComponents ();
   void addFcSubComponent (Component c);
   void removeFcSubComponent(Component c);
}
```

Table 1. Basic methods to be implemented by a Fractal component within the Julia framework (Java implementation of Fractal).

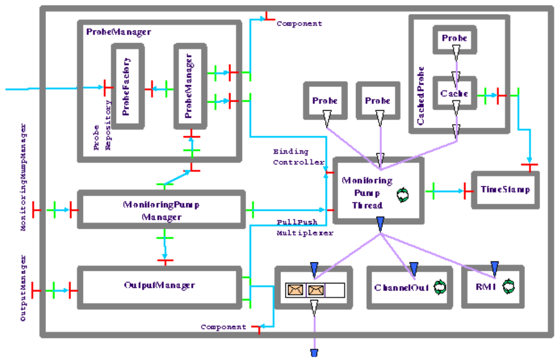

Fig. 3. The logical structure of assembled Fractal components. Manipulation within the graphical environment (screen snapshot).

4.2 Fractal and Proactive

Recently, the Fractal framework was ported to Proactive. This way, Fractal components can be also active objects, as described by the Proactive framework. Components can so migrate from one machine to another, can use asynchronous communication and so on. The Proactive framework itself is currently being re-factored to stick to the Fractal specification.

5 Needed Features

In this section, we discuss the features we need to implement our component search for Fractal components. We believe that some crucial tool and standard are still missing.

5.1 Documentation

As described, GRIDLE bases its analysis on the application structure and on free-text search among documentation files. With these two requirements, GRIDLE can select the components that match a given query and arrange the results according to our ranking metrics.

The current Fractal model does not include a strong standard for documentation file nor a way to distribute them: currently, only interfaces have a standard description with their ADL file.

We analyzed some Java application developed using the Fractal framework (including the reference implementation of the framework itself). In all cases, the detailed description of the interfaces (expected behavior etc.) is usually released in the form of a Java documentation file for the corresponding Java interface.

This will allow us to implement a search tool that chooses among different types of services, but not to choose among different service providers (e.g. different component implementations).

When more detailed information about a given implementation is needed, there is no clear standard for composite components, while, for basic components, we can observe a slight confusion among Fractal components and Java classes: the Java classes implementing the components are described with the standard JavaDoc files. This is not satisfactory for several reasons.

1. Methods and fields that are peculiar to the framework (e.g. the *bindFc* mechanism to bind components) are listed among the methods that implements the class logic.
2. Component dependencies (client interfaces) to other components can be inferred only through a detailed analysis of class fields, among which the binding variables are listed.
3. Redundant information is present about, for instance, methods inherited through the Java class structure (e.g. *clone, equals, finalize, getClass, toString...*) or interfaces implemented for other reasons related to implementation details (e.g. *javax.swing.Action, java.lang.Cloneable, java.io.Serializable...*).

This can be partly solved by browsing the interface documentation but it is not satisfactory, as this mixes the Java implementation with the component structure.

5.2 Packaging

A standard for component distribution and management is not yet part of the Fractal model. There is an on-going effort, within the developers' community, to include some packaging tools within the Fractal framework. An experimental tool, called FractalJAR, is currently under development.

A packaging standard should describe a way to ship components in a compact form (single-file archive), including:

– ADL files describing the internal composition of the component;

– implementation (executable) files;
– description files, with clear interface, dependency and behavior description.

Also, the package should have a strong version system, which should manage dependencies with different component versions and component upgrade. It should verify that an update does not break the coherence of the software infrastructure.

As a matter of fact, an interesting goal of searching would be to find complete packaged components (compositions), rather than individual components with outstanding dependencies.

Packaged components should also be easier to store and archive in specific repositories.

6 Conclusion

In this contribution, we presented our vision of a tool searching software components, and we discussed how to adapt it to Fractal.

We believe that in the near future there will be a growing demand for ready-made software services, and current Web Search technologies will help in the deployment of effective solutions.

When all these services become available, building a Grid application will become a easier process. A non-expert user, helped by a graphical environment, will give a high-level description of the desired operations, which will be found, and possibly paid for, out of a quickly evolving market of services. At that point, the whole Grid will become as a virtual machine, tapping the power of a vast numbers of resources.

References

1. S. Brin and L. Page. The Anatomy of a Large–Scale Hypertextual Web Search Engine. In *Proceedings of the WWW7 conference / Computer Networks*, volume 1–7, pages 107–117, April 1998.
2. Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of SIGCOMM*, 1999.
3. Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Component rank: relative significance rank for software component search. In *Proceedings of the 25th international conference on Software engineering*, pages 14–24, Portland, Oregon, May 2003. IEEE, IEEE Computer Society.
4. Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in oo programs. *Communications of the ACM*, 48:99–103, May 2005.
5. W.M.P. van der Aalst and B.F. van Dongen and J. Herbst and L. Maruster and G. Schimm and A.J.M.M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47:237–267, 2003.