Software Semantic Provisioning: *actually* reusing software

S. Sguera[a], A. Stellato[a], P. Ombredanne[b], M. T. Pazienza[a]

s.sguera@ieee.org
{pazienza, stellato}@info.uniroma2.it
philippe.ombredanne@eclipse.org

a: Università di Roma Tor Vergata, Dipartimento di Informatica, Sistemi e Produzione
b: The Eclipse Software Foundation

Abstract. Software development nowadays largely consists of adapting existing functionalities or components to perform in a new environment, and is biased towards delivering component-oriented architectures. Finding, choosing, provisioning and integrating the right libraries or components is still an ad-hoc and error prone task. This paper describes the SSP (Software Semantic Provisioning) project, funded in its early stages by Google™ Inc., developed during the Google Summer of Code™ 2007 program, and incubated by the Eclipse Software Foundation; the project aims to actually achieve software reuse in an effective, reliable and developer-friendly fashion, integrating cutting edge technologies in the component provisioning and integration areas, and providing support to decision-making in choosing the right dependencies set. A prototypical RESTful repository, and an Eclipse plug-in consuming the repository services have been implemented and will be discussed.

1. Introduction

Software development nowadays largely consists of adapting existing functionalities or components to perform in a new environment, and is biased towards delivering component-oriented architectures. Finding, choosing, provisioning and integrating the right libraries or components is still an ad-hoc and – thus – error prone task. Furthermore, it is sadly well know that object-oriented programming promised a lot about code reuse, but so far it never delivered it that much.

The problem of component provisioning, choosing the right software libraries set, and integrating it affects software developers and libraries providers. The impact of this is library choosing, component provisioning and integration tasks are carried out by developers, with little or no help at all.

The very general concept which lies behind software collection and reuse can be observed (in terms of needs) and applied (through successful methodologies and technical solutions) at very different level of specializations. While very general frameworks for software delivery and provisioning may offer services for accessing

and contributing to large library repositories, relying on dedicated metadata for organizing and retrieving the archived objects, there could be specific fields of interest where a more complex and organized description of the repository, tailored upon explicit needs and requirements which characterize the given domain, would improve the shareability of data, information and tools inside really active and participating communities.

Following previous research in the software components and libraries provisioning and integration by the ART group[1] at University of Rome Tor Vergata, this paper describes the SSP (Software Semantic Provisioning) project, funded in its early stage by Google™ Inc., developed during the Google Summer of Code™ 2007 program (details in [6]), and incubated by the Eclipse Software Foundation.

In Section 2 we will briefly introduce the main provisioning, build and integration support technologies currently available. Representative use case scenarios have been studied exploiting the prototypical implementation provided, and will be presented in Section 3, giving the reader a more thorough understanding of the surrounding environment and the actual benefits delivered to developers and component providers by the project. Section 4 will describe our approach, key goal and significant design issues. The software component domain has been formalized in the Software Provisioning Ontology (SWPO) whose main classes, properties and possible evolutions will be discussed in Section 5. Section 6 and 7 will be dedicated respectively to the discussion of architectural choices and issues we took both in server and client side development, while Section 8 will hold our conclusions and future directions of work and research.

2. State-of-the-art

A number of existing projects and efforts aim to describe software. Each one focuses upon a peculiar aspect, but no known product provides a thorough description enabling complex search and integration features. Hereafter we discuss the main characters populating the current component provisioning and integration panorama.

DOAP
The DOAP[2] (Description Of A Project) effort aims to describe a software project in terms of URI, maintainers, code repository and other product release-related features. No hints about what a given piece of software does or does not are given.

Maven
Maven[3] is one of the cutting edge integration and build management technology, and gained a significant market share in latest years. Its main goal is helping developers in

[1] http://ai-nlp.info.uniroma2.it
[2] http://usefulinc.com/doap
[3] http://maven.apache.org

handling dependencies and relieve the burden of integration and build process. The m2eclipse plug-in[4] allow developers to use POM files directly from the Eclipse[5] IDE.

Even if the folksonomy feature provided by the repository is quite functional and easy to use, and perfectly in line with the Web 2.0 hype, it does not provide a reliable mechanism to spot functional resemblance or more formal mappings and correspondence between components, as we propose in this paper.

OSGi Bundle Repository

OSGi[6] is the technology which enabled – among other things – the major shift in Eclipse's aims, from being a tooling platform (versions before 3.0) to a Rich Client Platform [3], and the subsequent changes in the requirements set, in terms of dynamic plug-in management, services, security, and performance. It provides an excellent platform for bundle provisioning and building dynamically extensible applications. A still evolving specification for building OSGi bundle repositories is given in [5].

Orbit

Orbit[7] mainly aims to reduce component duplication: it provides a repository of bundled versions of third party libraries that are approved for use in one or more Eclipse projects. It also clearly indicates the status of the library (i.e., the approved scope of use). Yet our aim is a bit more general, not simply attempting to reduce duplication, but collapsing – where possible – two or more libraries' functionalities in just a single one.

Buckminster

Buckminster[8]'s goal is to leverage and extend the Eclipse platform to make mixed-component development as efficient as plug-in development. It is very much focused on dependencies handling as well, while our approach is mainly aimed to improve components search and facilitate software reuse.

Kepler

The purpose of Kepler[9] is to address the complexities involved with provisioning, managing, and to use a shared infrastructure in order to support a community-oriented development model. The focus remains much tied to community-oriented development, more than component-oriented as in our effort.

Ivy

Ivy[10] is a project incubated by the Apache Software Foundation: it provides a tool for managing (recording, tracking, resolving and reporting) project dependencies. An

[4] http://m2eclipse.codehaus.org
[5] http://www.eclipse.org
[6] http://www.osgi.org
[7] http://www.eclipse.org/orbit/
[8] http://www.eclipse.org/buckminster/
[9] http://www.eclipse.org/proposals/kepler/
[10] http://incubator.apache.org/ivy/

interesting feature is transitive dependencies management: it shows simple inference capabilities, but no support for functionalities-driven smart search and reasoning, which characterize our approach, and are essential to us to enhance software reuse possibilities.

3. Main use cases and benefits

Despite the proliferation of provisioning systems and frameworks, the component search and choice activities are still carried out by developers with little or no help at all. Programmers are left to themselves scouting the web to find libraries and components, and no systematic approach nor thorough frameworks exist.

In the next paragraphs we will discuss some of the most representative use cases and the benefit they deliver to developers and components providers, stressing how our system tackles various aspects which currently undermine software reuse and often lead to write ex-novo already existing code.

Assert and spot functional equivalence between components

The number of components and libraries, along with their versions, makes practically impossible for a developer to know them all. On the other hand, there may exist more than a piece of software accomplishing the same task, fulfilling the same requirements set, or even implementing the same specification. To some extent, such components could be considered *functionally equivalent.*

This is the case, for instance, of Hibernate[11], Apache Cayenne[12] and all of the other frameworks implementing the Java Persistence API, or any implementation of the Java Servlet API, any JDBC driver, or any HTTP server (or client as well). The list would go a long way.

Furthermore, the equivalence is symmetrical, reflexive and transitive; the inference mechanism helps building relations upon social-generated contents: relations and functional equivalence among software components are both explicitly declared and inferred by the system, thus building a dense semantic network with a little effort. Machine-readable metadata allow much more granularity and raise the formal level and the *intelligence* of search-related features.

Let's suppose we just finished developing, for some obscure reason, a novel implementation of the Java persistence API. Let's suppose also that metadata about two common frameworks implementing the same API – i.e. Hibernate and Apache Cayenne – are already present in the repository, and (just as an example) that the two are declared as *functionally equivalent*. As we declare our library as equivalent to Hibernate, since they implement the same API, the inference engine can conclude my library is equivalent to Cayenne as well; Cayenne's mapping to our product is nowhere in the repository, but was just inferred. A developer looking for "Hibernate or equivalent" or "Cayenne or equivalent" libraries, or again "Java Persistence API

[11] http://www.hibernate.org
[12] http://cayenne.apache.org/

implementation" will then see our implementation among the query results, obtain information and in case decide to use it.

Find components providing a set of tasks
Describing a software component or library in terms of the tasks it fulfills is the very first way to tell whether a piece of software fits our needs or it does not. During the analysis and design phase developers must choose the right set of enabling technologies and components which will drive further development phases, and will construct the base for building our application's architecture.

Let's suppose – just as an example – we are planning to develop two components, one carrying out the *"dom-parsing"* task and the other fulfilling the *"sax-parsing"* task, and we would like to know if there is already a unique component providing both the tasks. It would be useful to browse the repository and discover at design time that *xerces-j* actually carries out both *sax* and *dom* xml parsing. We might then decide to use it if it fits our project's requirements.

Assessing reputation of components
Whenever a developing team picks up third-party code to underlie its application, it is implicitly taking responsibility someone else's code, which could affect their product's security and credibility. To this purpose, we could want to know which – and how many – components actually use one: this may give us valuable information about its reputation. On the other hand, if we developed a new component – and added it to the repository, it could be interesting to know which and how many components rely on our work.

4. Approach and design goals

Our key goal is to provide developers with a complete environment to exploit semantic metadata in order to effectively find and provision software components.

We tried to overcome the main limitations in current mainstream provisioning systems and frameworks, which are in turn tied to a particular technology or show a formalization level which grants no access to technology-independent, high level and enough granular information for a component.

Moreover, even if current provisioning technologies follow different approaches and stress different aspects proper of the software domain, there is a substantial overlap among the components' description they provide and rely upon.

Thus an ontology, meant to be a shared, higher level domain vocabulary among developers, allowing to semantically describe software and eventually mapping a subset of available metadata to one of the technologies available, would enable a thorough description of a component, aimed to stress *what* does the component do in an unambiguous fashion; this supports interoperability among developers and among technologies, provide some ground concepts to establish, declare or infer relationships among software components, and eases the reuse of existing software, giving developers a significant help in the early discovery phases.

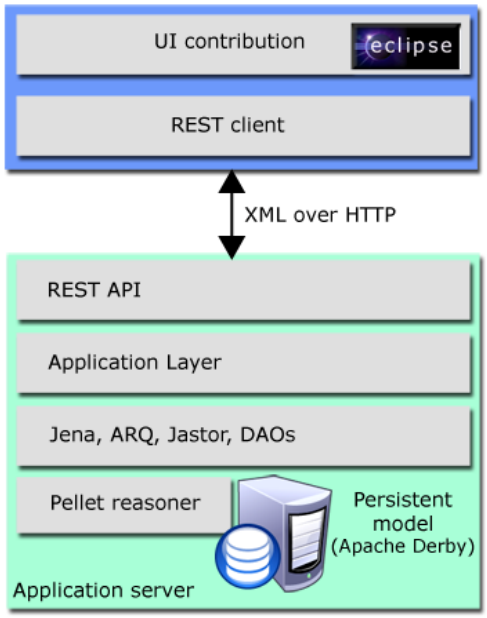

Figure 1: Server and Client side full stack architecture

A RESTful semantic repository (Figure 1), as it will be clearer in the next sections, easily allows the developing of a multitude of clients (i.e. browsers extensions, IDE plug-ins, et cetera), and broadens the field of possible applications.

5. Knowledge Model

The Knowledge Model of the SSP environment offers, at the current state of development, those concepts and relations which are necessary for providing a sufficiently detailed description of software entities and for modeling the functionalities which have been presented in the use-cases section.

Reference to past research work on modeling ontologies, like [4], for describing software systems has been made by reusing concepts from these ontologies for describing common software entities like: *component*, *library* and *software license*.

As it can be seen in Figure 2, our framework is centered about the description of software objects, providing several semantic anchors through which they can be identified, classified according to different perspectives and needs, and thus easily retrieved on these same aspects.

SoftwareObject(s) can be mainly distinguished according to two different categories: Components, which are "Program modules that are designed to interoperate with each other at runtime", that is software objects for which there is a well-defined runtime behavior, and Library(ies) which define "collections of subprograms used to develop software".

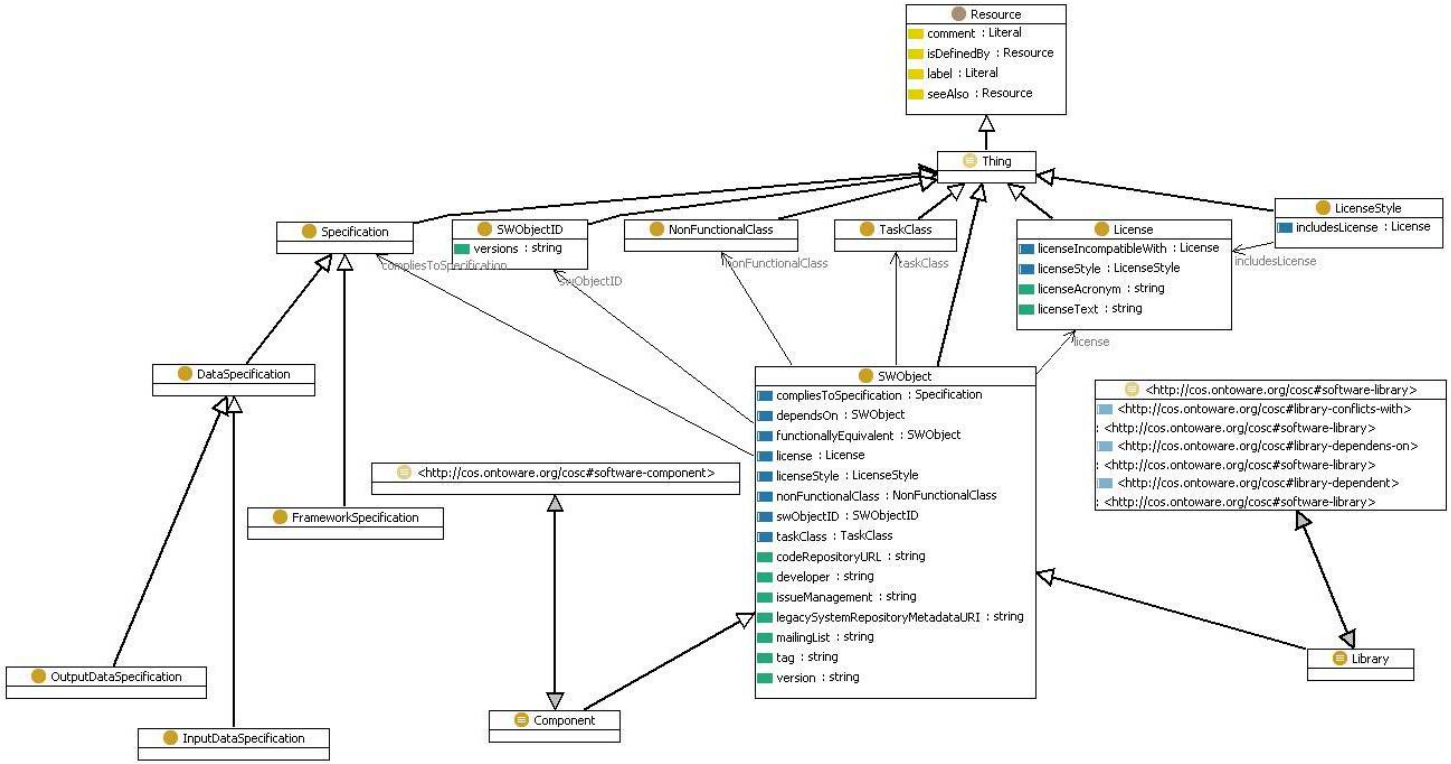

Figure 2: Knowledge Model of the SSP Framework

Other classes offer further perspectives over which software objects registered in the SSP repository may be clustered and accessed: License has been introduced to describe the diverse software licenses adopted by software developers and vendors. This way users may filter their choice if, as an example, they need only software licensed under a specific contract. This filtering can even less explicit, by automatic reasoning over class of licenses and the relationships between them. A property licenseIncompatibleWith allows to establish incompatibilities between use of components licensed under different contracts, while the class LicenseStyle describes categories of licenses which share common aspects. A reification technique (see [2] for a wider discussion on this topic) has been adopted to describe license styles both as objects of the domain as well as classes of licenses (so, as rdfs:subClassOf License), still remaining inside a first order description of the domain. This way we can "talk about" software license styles as ground objects (which may exhibit specific contractual expressions, have a reference web site for their general specifications etc…) and, at the same time, consider them as set of licenses, offering class level restrictions on the values that their belonging instances should expose on their properties. The explicit links between the objects (instances of LicenseStyle) and the set of Licenses (subclasses of License) is given by a restriction on a property which describes the specific style (if present) of any given license; the semantic repository thus automatically generates subclasses of License for each new introduced license style, together with their associated restriction.

With the same approach, it is possible to describe software with licenses according to a specific style, as for the following example:

$$\text{ApacheStyleLicensedSoftware} \equiv \exists \text{license.ApacheStyledLicense}$$

which describes (in description logic syntax) software distributed according to a license instantiating class ApacheStyledLicense, where this last is defined as:

$$\text{ApacheStyledLicense} \equiv \text{style} \ni \text{apacheLicense}$$

The same reification technique described above is used to automatically generate subcategories of SWObject which cluster sets of components and libraries according to their purposes, which are considered first class citizens inside the repository and not mere simple attributes for describing software. Specific Tasks can thus be defined in the repository and fully qualified according to their specifications and to descriptive information thought for human inspection; software objects can then be accessed, among the other ways, according to the task(s) they fulfill (e.g XML parsing, object persistence, text indexing etc…)

6. Server-side: the SSP Semantic Repository

The semantic repository publishes a set of REST API, in compliance to the well known architectural style described in [1] allowing clients to easily consume its services, and enabling any kind of Web 2.0 buzzword-compliant mashup. The RESTlet framework was embedded into a servlet container to deploy the repository as a web application.

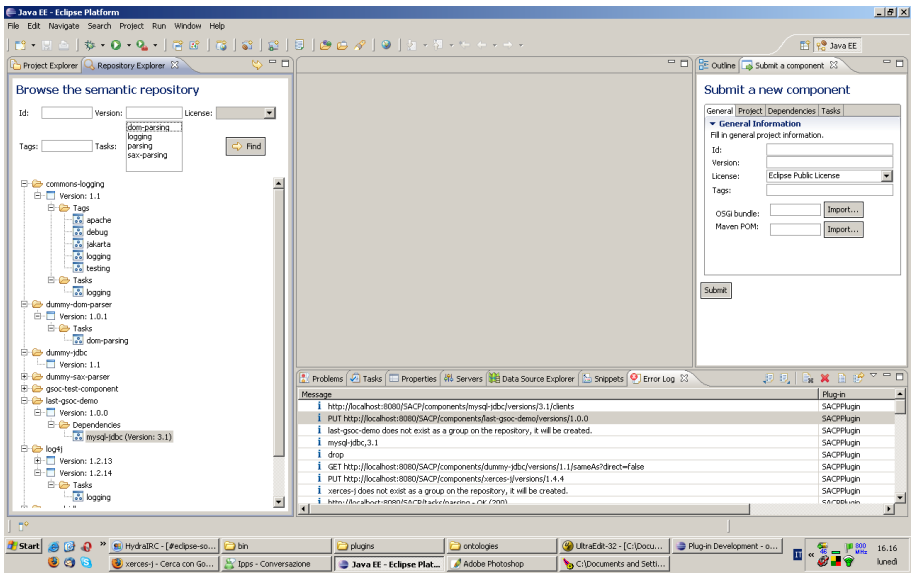

Figure 3: SSP Eclipse plug-in - UI contribution

Data serialization (beans to XML and vice versa) and complex services are handled by the application layer, while to access RDF triples stored in a persistent Jena model we took advantage of the IBM Jastor framework, providing OWL to Java mapping. Anyway, a further level of indirection was introduced not to tie the topmost layers to the specific technologies (i.e. Jastor) used in the data access layer. To enable inference-based web services we plugged Jena with the Pellet DIG reasoner.

7. Client-side: Eclipse SSP Plugin

We developed a RESTlet client consuming the repository's web services, decoupling the client-server interaction from the UI contributions.

The repository location can be both local (i.e. this can be achieved simply deploying the repository web application inside Eclipse itself, exploiting the embedded Jetty server used by the *help* plugin), or remote, and it can be chosen using the provided preference page, accessed in the usual Eclipse way.

Two views were implemented (Figure 3): the *Repository Explorer*, on the left, allows the developer to browse components by name, version, license, tags, tasks or navigate the semantic relations among the components; the *Submit a new component* view makes use of the Eclipse SWT Forms widgets to provide developers with an elegant and fast way to submit a new component to the repository. It is possible to define a component's dependencies, simply by dragging a component from the *Repository Explorer* on the left, and dropping it on the *Dependencies* tab in the component submission form, on the right. It is also possible to choose among the tasks already described in the repository, or add a new one throughout the submission process.

8. Conclusions and future works

In this paper we introduced a novel approach to software components and libraries discovery and provisioning. Indeed we believe current mainstream provisioning systems lack a shared vocabulary and technology-independent formalization of the software domain, supporting richer semantic description to support reasoning and the generation of a consensus based upon the specific domain the considered software belongs to.

Future iterations will involve a deeper axiomatization of *License* and *License-style* concepts, since they represent the contract between the product provider and the consumers, and often is a strict non-functional requirement to be satisfied when a third-party software is chosen. A strong investigation on "software specifications" could contribute to further discriminative arguments for facilitating classification (and thus more precise retrieval) of software objects in the repository. Integration with – and metadata reuse from – OSGi and Maven, and user interface improvements are top priorities for the project.

Acknowledgments

This work was funded by Google™ Inc. as part of the Google Summer of Code™ 2007 program, and developed by Savino Sguera mentored by Philippe Ombredanne – details in [6] – as a result of previous research work done in the area of software component provisioning by M. T. Pazienza, S. Sguera and A. Stellato at the ART research group at the University of Rome Tor Vergata.

We would like to thank Leslie Hawthorn and the whole Google Summer of Code™ team for the great job they did, and the Eclipse open source community for supporting the project and giving invaluable feedback throughout the development.

References

1. Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures, University of California Irvine, PhD Dissertation
2. Gangemi, A. & Mika, P. (2003). "Understanding the Semantic Web through Descriptions and Situations." Proceedings of the DOA/CoopIS/ODBASE 2003 Confederated International Conferences. LNCS 2888. Springer Verlag, 2003
3. Gruber, O., et al., 2005. The Eclipse 3.0 platform: Adopting OSGi technology, IBM Systems Journal, Vol 44, No 2, 2005
4. Oberle, D., Lamparter, S., Grimm, S., Vrandecic, D., Staab, S. Gangemi, A. Towards Ontologies for Formalizing Modularization and Communication in Large Software Systems Journal of Applied Ontology 1 (2): 163-202. 2006
5. OSGi RFC0112, 2005. http://www2.osgi.org/Download/File?url=/download/rfc-0112_BundleRepository.pdf
6. Sguera, S., 2007 http://code.google.com/soc/2007/eclipse/appinfo.html?csaid=1221666D7EBA3415