

Jedi : A Documentation Generator for Extensible Domain-Specific Languages

Roberto E. Lopez-Herrejon and Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712
{rlopez,batory}@cs.utexas.edu

Keywords: Domain Specific Languages (DSL), generator, language extensions, program documentation, template engines

Classification: 3rd year's work (Ph.D. middle)

1 Introduction

Program documentation is a very useful and indispensable resource for programmers and system users. However producing and keeping it up to date can be an expensive and time-consuming endeavor. Many tools have been developed to address this issue, but they often constrain the programmer to a fixed comment syntax and semantics, and the system user to a limited set of output formats and contents [3][5][6].

In recent years the software community has begun to realize the benefits in terms of *understandability*, *maintainability*, and *extensibility* that the use of *Domain Specific Languages (DSL)* brings to the software design and development process [1][4]. Unfortunately there have been no attempts, that we are aware of, to provide tools that support the generation of documentation for DSL programs.

This same situation is also present in extensions to popular languages like Java, where the documentation generator *javadoc* [6] provides a fixed comment compiler that does not allow any syntactic extensions to the Java language, and a limited form of extensions to documentation contents or format.

Our research addresses this problem: we are developing an extensible version of *javadoc*, called *Jedi (Java Extensions DocumentatIon)*. It supports the creation of *javadoc*-like HTML documentation pages for domain-specific languages, including DSL-extended versions of Java. Jedi is part of a larger project to develop a suite of tools - an *integrated development environment (IDE)* - for feature (or aspect) oriented software development.

2 Jedi Architecture

Jedi is based on the GenVoca product-line methodology [2], and is implemented using the *Jakarta Tool Suite (JTS)* [1]. Figure 1 is an example of the Jedi architecture for Java extensions that implements *State Machines (SM)* and Templates. Java and each of its language extensions (SM, Templates) is represented as a layer in this model that contains the following four elements:

- **API:** contains the class hierarchy of the data structures that hold the information of the comments and program structure (methods, fields, classes, etc.). We based our API designs on Sun's Doclet API [6].
- **Harvester:** consists of a BNF grammar file that defines the syntax of the language extension and its corresponding parser. It is responsible for collecting the documentation information for the syntactic structures defined in its grammar and for storing information in the corresponding API data structures.
- **Writer:** coordinates the order in which harvesting and output generation is done among the different layers. For example, given a package name to document, it first collects the information of all the classes in the package, resolves cross-references, and finally calls the output generation routine for each of those classes.
- **HTML doclet:** produces as output the HTML documentation for a parsed program.

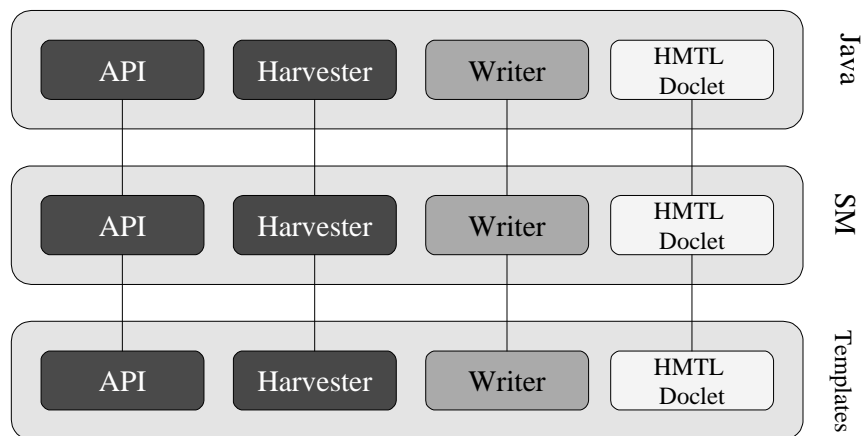


Figure 1: Example of Jedi Architecture

Jedi architecture resembles a matrix where the rows are language extensions and the columns are the application features.

The architecture has been illustrated for extensions of Java, a general purpose language, but it can be easily tailored to fit any DSL. To do that, the language designer has to: add to the DSL parser grammar productions to collect comment information (Harvester), provide data structures to hold that information (API), decide how and

when the documentation is harvested and generated (Writer), and design the output formats for each document type (Doclet).

One of the key concepts of Jedi is the composition of documents. To generate the documentation, we make use of *Velocity* [11], a *template engine*, that greatly facilitates the composition process. A template engine is a tool that decouples the task of designing a document from that of writing a Java program that generates this document [9]. A *Velocity template* is a text file that follows the format of the document you want, like HTML or XML, but that has a set of embedded directives that tell the template engine the places where the information that varies among the different instances of that document will be, and how this information will be obtained.

The Velocity templates for Jedi can be conceptually divided in two categories:

- **Base templates** that correspond to the main program structures that we want to document, like classes or states machines. These templates contain *extension frames* that are the places where subsequent extensions will add new information.
- **Extension templates** that add contents to the base templates in the corresponding extension frames.

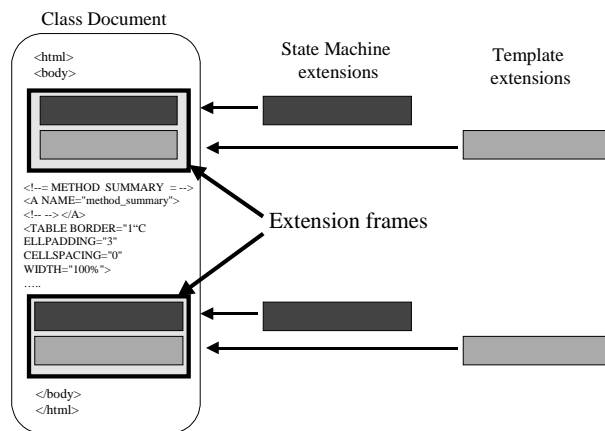


Figure 2: Jedi Composition of Class Document

Figure 2 illustrates this distinction. It sketches the HTML layout of a Class Document that contains the information of a normal Java class in a format similar to *javadoc*'s standard doclet, i.e with class headers, followed by summary and details sections of each type of class member (constructor, method, variable, etc.). If a class is extended, for example, with States Machines, such extension may have to provide (for the State Machines declared in a class) a summary section and a details section. These new sections would be added at the *extension frames* of the Class *base template*, and each of them would be defined by means of an *extension template*.

3 Future Work and Conclusions

Jedi is an application capable of generating documentation for extended Java programs. We showed its components and architectural model, and described how those could be applied to DSL programs. We also sketched how the composition of documentation is performed and the technologies it relies on.

While writing Jedi we noticed that harvesters contained a set of common patterns. This lead us to believe that a harvester can be automatically produced from a DSL (which is a grammar file plus some annotations) that relate patterns to grammar rules and API structures. If that is the case, this finding will prompt the creation of a new tool that will significantly reduce the effort to develop these elements, which are the most complex parts of a Jedi application.

Separation of concerns (SOC) [10] has been an evasive goal of the software engineering discipline. Jedi is one of the most conceptually complex applications built with the GenVoca product-line methodology, however we were able to separate the concerns relevant to a documentation generation application, namely API, Harvester, Writer, and Doclet; which can be expressed in a matrix-oriented manner as in Figure 1. This finding gave us a strong intuition that GenVoca could be used to address the SOC problem.

We need to study how and for what application domains (other than documentation generation) it can be effectively utilized, and if it requires further adaptations or modifications to provide that kind of capability. Furthermore we will explore the relationships that such revised GenVoca methodology might have with other approaches, such as Aspect-Oriented Programming [7][8] and Hyperspaces[10], that have been recently proposed to address the SOC problem. It is here where we believe the most promising results of our research are expected to come.

4 References

- [1] D.Batory, B.Lofaso, and Y.Smaragdakis. *JTS: Tools for implementing Domain-Specific Languages*, 5th International Conference on Software Reuse, Victoria, Canada, June 1998.
- [2] D. Batory, R. Cardone, and Y.Smaragdakis. *Object-Oriented Frameworks and Product Lines*, 1st Software Product-Line Conference, Denver, Colorado, August 2000.
- [3] S. Benz. *DocJet Documentation*. <http://www.talltree.com/docjet/index.html>
- [4] A. van Deursen and P. Klint, "Little Languages: Little Maintenance?", *SIGPLAN Workshop on Domain-Specific Languages*, 1997.
- [5] Friendly, L. *Design of Javadoc. The Design of Distributed Hyperlinked Programming Documentation (IWHB)*. Springer-Verlag, Montpellier, France, 1995.
- [6] *Javadoc-The Java API Documentation Generator*. Sun Microsystems Web site. <http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javadoc.html>

- [7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.J. and Irwin, J. *Aspect-oriented programming*. In Proceedings of the 11th European Conference on Object-Oriented Programming, June 1997.
- [8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G. *An overview of AspectJ*. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Hungary, 2001. Springer-Verlag.
- [9] Messerschmidt, L. *Take the fast track to text generation. Create text content with template engines and save time and frustration*. Java World, July 2001. <http://www.javaworld.com/javaworld/jw-07-2001/jw-0727-templates.html>
- [10] Ossher, H. and Tarr, P. *Multidimensional separation of concerns and the hyperspace approach*. In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2001.
- [11] Velocity Template Engine. <http://jakarta.apache.org/velocity/index.html>