Generating New Types of Documentation

Ian Hopkins

October 23, 2007

Abstract

Software documentation is an important artifact of any software development project, yet it may be costly to produce and keep up to date. Many tools have been proposed to assist in generating and locating documentation automatically so as to reduce the burden of having up to date documentation available. In this paper I present a model for classifying different types of documentation, classify existing tools using this model, and then explore prototype tools that produce documentation types not well covered by existing tools. After presenting their implementations, I evaluate the prototype tools in respect to the original model and show that they succeed in providing new information without adding major overhead to the development process.

1 Introduction

Documentation can be valuable in software projects during all phases of software development and as a deliverable. When re-designing a system, building an extension, or fixing a bug, developers need to know how the software works so the effect of changes can be minimized. Maintaining documentation takes time and is not always done well; Briand found that in practice the documentation available for software projects is poor, incomplete, and out of date [1]. Yet Briand's study also found that this same documentation was useful to developers in most cases.

Developing documentation can be a large part of the software development process. Because of the importance of accurate, quality documentation in many projects, Preistley and Utt developed an extension to the Rational Unified Process (RUP) that augments the famous software development process with processes for developing documentation [11].

The accuracy of a software project's documentation can be critical to its success. Ensuring the correctness of an application programming interface (API) for a closed source library is essential to allow third parties to use and/or extend that library. Automatically generated documentation has become a popular way to create accurate API documentation. In Forward and Lethbridge's study of documentation tools, automated documentation systems like Javadoc, Doxygen, and Doc++ rank second only to Word processors for popularity amongst software developers in industry [4]. The popularity of traditional word processors indicates that although some good tools for generating documentation exist, not all documentation needs are satisfied by these automated tools.

To better explore what types of documentation can be generated automatically, this paper presents a model for classifying documentation in section 2, explains related work on automated documentation tools in section 3, and classifies those systems in section 4. Section 5 introduces prototype tools for generating documentation, followed by an explanation of their implementation in section 6 and finally justifies these tools by showing they satisfy new portions of the documentation model in section 7.

2 Model of Documentation

Software project documentation can refer to many parts of a given project including: requirements documentation, source-code comments, a user's manual, unit tests, or even a project post-mortem report. To focus my exploration of generated documentation, I propose a model for classifying software documentation. This model is aimed at classifying different artifacts of documentation and specifically does not partition artifacts by their presentation (textual or graphic, static or dynamic), underlying information source (source code, profiles, runtime information), or quality, but rather by the types of questions the artifacts might help answer. The dimensions on which the model classifies documentation artifacts are explained below:

2.1 Audience

Audience captures the intended user of the artifact. This ranges from the application end-user (How do I install this application?), to a third party developer (How do I include this library in my project?), to an internal developer (How do I build this application?). The artifact's content needs to be appropriately adjusted depending on its audience.

2.2 Abstraction

Abstraction is the degree to which the document abstracts the underlying details. Abstraction ranges from providing an project overview (How is the project designed/structured?) to explaining in depth details (How is this sort implemented?).

2.3 Temporality

Temporality describes the intended lifetime of the artifact. This ranges from long-term concerns (What are the requirements and how have they evolved?) to short-term queries (What was the last change to this source file?).

2.4 Phase

Phase indicates project phases in which the artifact is useful. This may vary across different development processes (extreme programming vs. waterfall), but might range from design (What were the motivations for this decision?), to development (What is the coding convention of this package?), and to testing (What test cases must this class/method satisfy?), to maintenance (How does the latest version affect performance?), to post-mortem (Was the original design achieved?).

2.5 Focus

Focus describes what aspect o the project is primarily described by an artifact. Parnas introduced this classification of documentation in 1998 and proposed three categories: constructive which describes the implementation (How does the is this requirement implemented?), behavioral which describe's the runtime behavior (How does the application perform at runtime?), and before/after which describes state changes over time (What call sequence will put this object into a desired state?) [9].

3 Related Work

Javadoc is a tool that automatically generates API documentation from special mark up comments, method signatures, and class and method names in Java source code [6]. Javadoc was invented by Sun to help generate the documentation for the Java platform API. In Kramer's discussion of Javadoc, he makes two important points: first, that keeping documentation in-line with source code helps keep that documentation up to date and concise; and second, that developing documentation to satisfy every audience is impractical.

Doxygen is a tool for automating the documentation of C and C++ projects based on DOC++ and similar to Javadoc (Doxygen even supports the Javadoc comment format) [14]. Beyond generating API descriptions like Javadoc, Doxygen is able to generate dependency graphs, inheritance diagrams, and class diagrams in UML that are included in the generated output.

Collard et al. present *srcML*, an XML format for representing program source code for which the authors have developed an ANTLR-based generated translator that can mark up C++ source with srcML tags [2]. Once marked up in srcML, standard XML tools like XPath and XSLT can be used to execute ad hoc queries on the program source code. Collard et al. show that srcML is superior at Linking/Querying, and Software Visualization as compared with both plain text source code or an abstract syntax tree (AST) representation. Furthermore, srcML can be generated from a partial parse while ASTs generally require a complete parse.

Rose is a part of Rational's Enterprise Suite for software engineering [10]. Although primarily a forward engineering tool, Rose can be used to reverse engineer some documentation from C++ source code as well as DLL, JAR, and

COM exported binaries. In particular, Rose can be used to generate UML class and inheritance diagrams.

Čubranić and Murphy developed *Hipikat*, an application that mines software project knowledge from the project's CVS repository, developer mailing lists, bug tracking database, and website [3]. Hipikat uses the collection of documentation mined from these sources to suggest relevant documentation to programmers as they are working on a task. Relevance is determined by inferred links and information retrieval techniques like cosine similarity of keyword frequency vectors (for example between a query and indexed each document). Hipikat mines documentation, but generates relationships between documents. Furthermore, the authors mention that it could be extended to generate some missing documentation.

Calliope is an IDE plug in developed by Madsen and Nürnberg to record and look up documentation in a better way than mailing lists (the primary source of documentation in open source projects) allow [8]. Madsen and Nürnberg specifically model multi-valence which allows Calliope to support multiple perspectives or opinions on a given artifact. Although their results were not statistically significant (likely due to the small sample size), the authors did find modeling multi-valence reduced the time spent per documentation change by the documentation writer as compared to not modeling multi-valence. Documentation is not generated with Calliope, but navigation, development, and integration of multi-valent documentation is automated.

Xie and Pei developed *MAPO* in response to the insufficient documentation for Apache's Byte Code Engineering Library (BCEL) [15]. MAPO mines source code from open-source repositories to find common usage sequences of API methods, and then presents these to the user. The project aims to augment Javadoc-like documentation (what methods and their parameters do), with API usage that shows the context (of other method calls) within which those methods are usually found.

4 Classification of Related Work

4.1 Audience

Javadoc, Doxygen, and MAPO produce API documentation that targets 3rd party developers who will be consumers of those APIs. srcML, Hipikat, and Calliope generate documentation targeted at internal developers, and Rose's UML diagrams could be used to compliment either type of documentation. No tools attempt to automatically generate end-user documentation, likely due to the knowledge of both the user and domain required to create user documentation.

4.2 Abstraction

Javadoc and Doxygen allow users to drill down from an overview using hypertext links, but this drill down changes the level in the code structure, not the level of abstraction. The other tools respond to user queries (srcML via XPath) and give specific answers to those queries. In general, automated tools do not provide good overview abstractions.

4.3 Temporality

Hipikat, MAPO, and srcML are designed to answer user queries and generate documentation on the fly for short term use. Javadoc, Doxygen, Rose, and Calliope produce artifacts like API references that are long-term and can even be deliverables that live beyond the development project.

4.4 Phase

Hipikat, Calliope, and Rose are intended to be used by developers in change tasks likely during the maintenance phase. srcML can be used in both development and maintenance, while the other tools are used by 3rd party developers after delivery. Rose may also be used for forward engineering during design/development, but Pierce and Tilley did not discuss this in depth. It is likely that these forward engineering tools are focused on generating code not documentation.

4.5 Focus

Most of the tools use source code to generate documentation and as such focus on constructive documentation. MAPO uses source code to gather usage patterns to produce before/after documentation. Hipikat and Calliope are generally constructive, but can retrieve other information if it is written manually and added to the system's repository.

5 Prototype Tools

To explore parts of the documentation model that are not well addressed by existing tools, I have begun work on a set of prototype tools. The prototype tools generate documentation from mostly existing information sources thereby providing developers with new types of documentation and limited additional overhead. The prototypes are designed to be interactive so as to produce customized (to the current need), disposable documentation with ability to select different levels of abstraction. The prototypes are also designed as small interacting components so that they can be combined and replaced to answer ad hoc questions about a given application.

5.1 Generated

Generating documentation with the prototype tools involves two major processes. First, facts must be extracted from the source code and test results and stored in a facts database. Second, when users wish to access a certain type of documentation they use one of the prototype tools to query the facts database. This process is described further in section 6). By generating documentation when it is needed, developers can easily obtain documentation for projects when they need it and do not need to worry about cataloging, persisting, or sharing a repository of documentation. Furthermore, developers can generate documentation for code that is part of a private branch and has not yet been approved for the central repository.

By utilizing a process that is similar to compiling source code, developers can use common applications for both documentation and source code. In deploying the Javadoc system at sun, Kramer found using similar processes and tools like version control and bug reports was essential to getting programmer buy-in [6].

Although Forward and Lethbridge found that documentation is generally not kept up to date, testing and quality assurance documents are [4], thus test cases and their results (as well as source code and application binaries) are good, up to date sources of information. Generating documentation automatically from these sources may therefore lead to more up to date documentation than manually writing documentation.

5.2 Information Sources

To minimize the overhead for developers of using the prototype tools, all facts used by the tools are gathered from the following sources:

5.2.1 Source Code

The source code for the application and the comments embedded in that source code are a valuable source of constructive information. Furthermore, tutorial example, unit test, load test, and profile test source code can also provide information about how to use and interface with portions of the application. Since unit, load, and profile tests are generally written for essential, core functionality, these tests likely use and test important features of the application.

5.2.2 Test Execution Results

Unit, profile, and load test results give us a view of the application's behavior in terms of test timing, operation counters, and memory usage. Being able to execute live profile queries on a running application may also provide a great source of information about the behavior in response to events (e.g. pressing a button or executing a command). Live profiling is not yet implemented.

5.2.3 Version Information

Although the prototype tools do not directly interact with the version control system, when the facts database is updated, the current version of the source code is recorded with each fact. Indexing information by version allows us to easily document changes over time and facilitates inter-version comparisons.

5.3 Interactive

Documentation is generated on the fly in response to a user query. Instead of consulting voluminous documents, developers can simply generate relevant documentation at the desired level of abstraction which answers their query. Furthermore, some of the prototype tools allow the user to drill down to details of the system from a high-level overview.

Sillito found that programmers ask four distinct types of questions when doing maintenance tasks [12]. Further, Kramer states that the documentation needed for a particular software project can depend on business goals [6]. Finding a single representation that is appropriate for all types of questions would be difficult. Instead, the prototype tools provide a number of different representations of the information in the facts database that the developer can choose from when working on a given task.

Kirk found that documentation can be difficult to navigate, split up, and index for large frameworks [5]. Allowing the user to determine what documentation to generate, using manual filtering, and allowing drill down from high-level abstractions down to source code details may help simplify navigating documentation.

5.4 Small tools

Instead of using an integrated, heavy-weight framework for documentation, I have built a collection of small tools that developers can easily combine or replace to suit their needs. The tools are command line based and are connected by pipes. The tools are simple and could be added to a makefile for automating the generation of some standard documentation.

As a conclusion of Forward and Lethbridge's study of documentation tools, he recommends that future systems strive to be light-weight and disposable [4]. I have designed these tools to be light-weight and simple to replace so that developers can customize the documentation system to meet their needs. Kramer reports that users have customized the output formatting of Javadoc by replacing the default *doclets* produced by Sun [6]. Javadoc can also be customized with taglets which allow for custom mark up tags in source code comments.

6 Implementation

6.1 System Overview

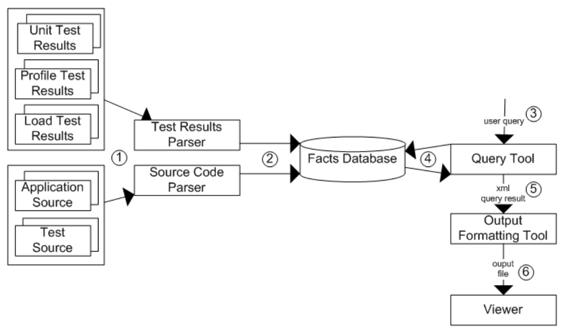


Figure 1: System Diagram

1. Up to date, raw source code is passed to the source code parser and the output from unit, load and profile tests are passed to the test result parser.
2. Both the source code and test results parsers direct their output to the SQLite3 facts database. When information is stored in the database it is indexed by the source code version from which it was generated.
3. A user executes a command line query to one of the prototype tools.
4. The command line tool makes specific queries to the database which returns matching facts about the application being queried.
5. Tools generate XML output which is then sent to an output formatting tool.
6. The resulting formatted output file (JPG, PDF, or HTML file) is opened by the user with a viewer for that output format.

6.2 Tools

6.2.1 Source Parser

The source parser is implemented as a series of PERL and TXL scripts that mark up source code with XML tags and is an extension of the parser by Andrew Sutherland [13]. The parser recognizes and marks up package, class, method, parameter, constructor, thrown exception, and member variable declarations as well as method and constructor calls. Once the source code is marked up, the declarations, calls, line numbers, comments, and current version are recorded in the facts database by a PERL script. The source code parser can also be used to infer facts based on keywords in the source files. This process of updating the facts database could be automated as part of the build process for the application so that the fact database is always in sync with the application binaries.

The current version marks up some references but does not completely resolve them as this can get quite complicated (e.g. parsing types from Runtime.getRuntime().getTotalMemory()). Although the parser is currently only able to process Java source code, the TXL Java language grammar could likely be changed to handle other C-based object oriented language with some modifications. Furthermore, source code marked up in a similar XML format, srcML for example [2], could be transformed with a short XSLT script.

6.2.2 Test Results Parser

As part of the build process, the unit and profile tests will need to be executed and their output parsed for inclusion in the facts database. The profile and unit tests export XML which can easily be parsed by a PERL script. The results are linked to both the source code which implements the tests as well as the source code that is tested. These links are only preliminary in the prototype tools.

6.2.3 Facts Database

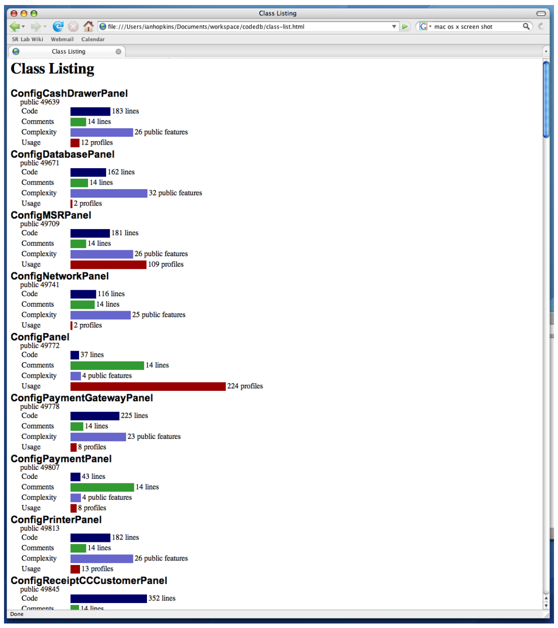
The facts database is implemented as an SQLite3 file-based database that indexes facts by type (package, class, method, variable, etc.), version, and parent (the Test class is the parent of its setUp method). The facts database has a simple schema that allows application developers to easily extract the information they want. Furthermore, the database is extensible in that new columns can be added to the data tables without changing how the prototype tools use the existing data. For example, the database could be extended to attribute changes to CVS users.

Although exceptions and method/constructor calls can be recorded in the facts database, this is not implemented in the current version.

6.2.4 Complexity Analysis

This prototype tool uses the facts database to extract information about the source lines, comment lines, external complexity, and profile for each method in a given project. This information is visualized in a Dynamic Hypertext Markup Language (DHTML) document. The resulting document can be viewed in any modern web browser and allows the user to filter classes by keyword, and drill down to individual methods. External complexity is defined as the number of unique types one must know about to use a given method which is the size of the set formed of the return type, the parameter types and any thrown exceptions.

The current implementation does not consider exceptions as they are not recorded in the facts database. Figure 2 shows the prototype Complexity Analysis tool run on an open-source Point of Sale solution called synPOS [7].


Class Listing
file:///Users/ianhopkins/Documents/workspace/codedb/class-list.html
SR Lab Wiki
Webmail
Calendar
Class Listing
Done

```
Class Listing

ConfigCashDrawerPanel
  public 49639
   Code         183 lines
   Comments     14 lines
   Complexity   26 public features
   Usage        12 profiles
ConfigDatabasePanel
  public 49671
   Code         162 lines
   Comments     14 lines
   Complexity   32 public features
   Usage        2 profiles
ConfigMSRPanel
  public 49709
   Code         181 lines
   Comments     14 lines
   Complexity   26 public features
   Usage        109 profiles
ConfigNetworkPanel
  public 49741
   Code         116 lines
   Comments     14 lines
   Complexity   25 public features
   Usage        2 profiles
ConfigPanel
  public 49772
   Code         37 lines
   Comments     14 lines
   Complexity   4 public features
   Usage        224 profiles
ConfigPaymentGatewayPanel
  public 49778
   Code         225 lines
   Comments     14 lines
   Complexity   23 public features
   Usage        8 profiles
ConfigPaymentPanel
  public 49807
   Code         43 lines
   Comments     14 lines
   Complexity   4 public features
   Usage        8 profiles
ConfigPrinterPanel
  public 49813
   Code         182 lines
   Comments     14 lines
   Complexity   26 public features
   Usage        13 profiles
ConfigReceiptCCCustomerPanel
  public 49845
   Code         352 lines
   Comments     14 lines
```

Figure 2: Complexity Analysis Tool's output. Comparing memory usage and timing of four sorting algorithms.

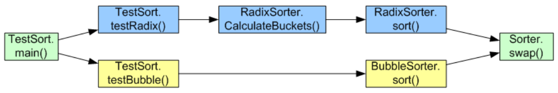


Figure 3: Callgraph Analysis Tool's output. Compares the call graphs of two sorting tests from the main() function to the swap() function.

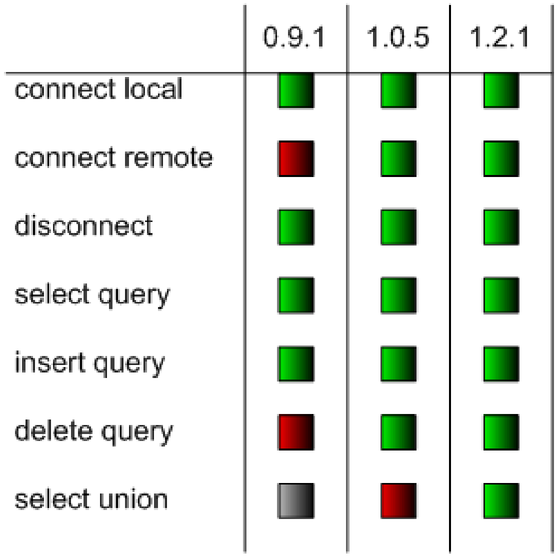
6.2.5 Callgraph Analysis

This prototype tool uses both static analysis and profile traces to determine call graphs in the application. This tool could compare two test cases by generating a call graph for each and using the edit distance algorithm to find how the two graphs differ (possible overlaying the two graphs in a visualization). Callgraphs can be difficult to generate due to the number of calls in a given application, but this could be reduced by allowing users to filter the graphs to include only calls the user is interested in (by keyword or depth).

The callgraph analysis tool is not yet implemented and is only an idea. An idea of how the output of this tool might be formated is presented in figure 3.

6.2.6 Unit Test Analysis

The unit test analysis tool records the results of unit tests (pass or fail) and stores them in the facts database. The information recorded in the database can



	0.9.1	1.0.5	1.2.1
connect local	🟩	🟩	🟩
connect remote	🟥	🟩	🟩
disconnect	🟩	🟩	🟩
select query	🟩	🟩	🟩
insert query	🟩	🟩	🟩
delete query	🟥	🟩	🟩
select union	⬜	🟥	🟩

Figure 4: Unit Test Analysis Tool's output. Comparing memory usage and timing of four sorting algorithms.

provide developers with a number of interesting perspectives on the state of the application. First, a developer can get an overview of how many tests passed and how many failed. Second, developers can determine their progress by comparing the number of tests passed now with a previous version. Third, developers can look for how new tests may break old versions possibly suggesting potential patches. Fourth, the unit test analysis tool can provide detailed analysis on what code has been exercised as the result of a given unit test.

The analysis and output formatting portion of this tool are not yet implemented, but unit test information is being stored in the facts database. Figure 4 shows a potential output format for this tool.

6.2.7 Profile Test Analysis

Profile tests are similar to unit tests, but are marked up with special profile start, count, and stop instructions. The start and stop instructions record a profile of both CPU time and memory usage for the test. The count instruction is used to count individual operations (e.g. swap operations in a sort implementation). This information is collected while the tests are running and added to the facts database. The results are then exported to XML for easy manipulation and visualization. The information collected by the profile tests can help developers consider a number of different aspects of the application's performance: first, an overview of memory and CPU usage over the entire suite can help pinpoint problem areas; second, regressions in performance can be spotted easily by comparing results from two different versions; third, developers can easily see

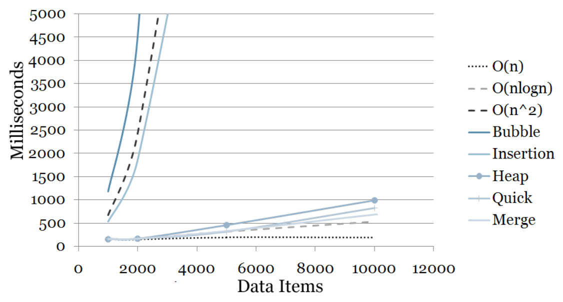


Figure 5: Profile Analysis Tool's output. Comparing sorting algorithm execution time against included baseline algorithms.

how the application reacts to different situations; and fourth, the application can be compared to known baselines, like an O(n) algorithm or a competitor's solution, for both space, operation counting (difficult with competitor solutions, but may be possible with byte-code instrumentation), and time usage.

The profile instrumentation tools work and visualization is done with graphs in a Microsoft Excel spreadsheet. This could easily be ported to gnuPlot to be better in line with the goal of using small components in the tool chain. The profile instrumentation also includes algorithmic baselines (e.g. Waste $O(n)$ space, Waste $O(n \log_d m)$ time) for comparison. Figure 5 illustrates comparing baselines to sorting algorithms.

6.2.8 Output Formats

The implemented prototype tools use XML and DHTML as output formats. I have also done some investigation into an interactive AJAX profiler that would allow the user to view live application performance as they used an application from within a web browser. This is not fully implemented, but seems both a promising tool and format for visualizing source code facts.

7 Evaluation

7.1 Proof of concept

The first portion of evaluating the ideas in this paper is to prove that I can build tools to extract, store, and visualize the types of information developers might be interested in that are not well implemented in existing tools. To do this I have implemented a complexity analysis prototype, part of the unit test prototype, and the profile analysis prototype tools. Evaluating these tools required an application with some complexity and known performance characteristics (for comparing the results with).

Sorting is an important problem in computer science which has a number

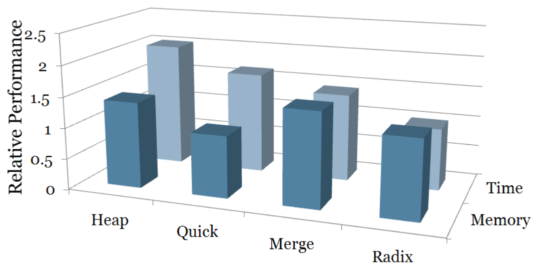


Figure 6: Profile Analysis Tool's output. Comparing memory usage and timing of four sorting algorithms.

of different solutions each with well documented time and space analyses. Although sorting is a simple problem, some algorithms are sufficiently complicated and exhibit well defined performance differences. For testing I implemented seven sorting algorithms listed below with their analyses below where n is the number of elements to sort:

1. **Bubble** sort. $O(n^2)$ time, $O(1)$ extra space. Usually slower than selection and insertion sort.
2. **Selection** sort. $O(n^2)$ time, $O(1)$ extra space.
3. **Insertion** sort. $O(n^2)$ time, $O(1)$ extra space.
4. **Quick** sort. $O(n^2)$ time, $O(1)$ extra space. On average takes $O(n \log_2 n)$.
5. **Heap** sort. $O(n \log_2 n)$ time, $O(1)$ extra space.
6. **Merge** sort. $O(n \log_2 n)$ time, $O(n)$ extra space.
7. **Radix** sort. $O(n \log_d m)$ time, $O(n)$ extra space, where d is the integer base (2 or 10) and m is the maximum integer in the collection. Radix sort only works for types that can be decomposed by digit (integers only in this implementation).

The results of the profile test are presented in Figure 6. This figure shows that Merge and Radix sort use additional memory compared to quick sort and heap sort, radix sort is slightly faster than the others, and quick sort executes in about $O(nlog_2n)$ time. The difference in memory usage between quick sort and merge does not appear as exactly the data size, which is possibly due to the extra stack frames created by recursion in quick sort. All of these observations match the known algorithm analyses.

There are some problems with the profile analysis prototype. First, memory usage is only estimated by making queries to the Java runtime (java.lang.Runtime) and the numbers reported are not exact. This is because of over allocation and delayed collection by the garbage collector, but is limited by invoking the garbage collector each time a profile is collected. Second, no optimizations were implemented even though there are well known cases in which performance can be drastically increased. One easy optimization for quick sort is to use another algorithm on short lists since quick sort is inefficient on small lists. Like any application profiler, the profile analysis prototype introduces a small performance overhead by periodically polling the Java runtime and invoking the garbage collector.

I also implemented a complexity visualization tool and tested this on an open-source point of sale (POS) application called synPOS [7]. This tested both the TXL source parser, the facts database, and the complexity visualization tool. The resulting visualization is shown in Figure 2.

7.2 Relation to Model

Finally, the prototype tools and the information they expose must be validated in terms of the documentation type model introduced in section 2.

The profile and unit test prototype tools allow us to generate documentation that not only focuses on how the application was constructed, but also its behavior at runtime. The profile and unit test analysis tools can can abstract test results to a few numbers or zoom in on timing and memory usage details of a single test. The facts database allows us to compare a development build against a stable release on the fly or generate a comparison of feature tests from one version to another (a long-term artifact that could be used to encourage customer upgrades). The profile test suite could be used anywhere from the design phase (to prototype an idea to see basic performance implications) up to the maintenance phase (run and report on regression tests on a potential patch. The profile tool could be used by both internal developers to improve performance and by 3rd party developers to understand the performance of API calls.

Profile tests are easy to create from unit tests (in this case, only two lines were added to the unit tests), introduce only a minor performance overhead in testing only, and provide an accurate account of the application's memory usage and processing time.

8 Conclusion

In this paper, I have introduced a useful model to describe different types of documentation that I used to classify existing tools. This analysis helped me explore a set of prototype tools to produce types of documentation not generated by existing tools. I implemented two of these prototype tools and showed that the profile analysis tool is effective at identifying both processing time and memory usage trends. Finally, I have presented the start of my work on other promising prototype tools for automatically generating documentation.

9 Further work

I have plans to extend this work by implementing more of the tools described in this paper. Once I have a set of working tools I would like to perform a study of developers using these tools to find out if they do satisfy the dimensions of the model that I predict they will, if they are useful to developers, and how they might be improved to provide even better documentation.

Beyond further evaluation of this work, it would be interesting to see how this work could be combined with management techniques to encourage good documentation habits within a software project?

Another interesting area not covered by this work, is that of consequences inherent in linking documentation to source code and test cases via automated generation. This may have positive benefits including more up-to-date documentation, but it may reduce the amount of documentation available early in the project since documentation can be generated quickly later. The documentation may be determined by the implementation when it may be preferable to have the inverse.

References

[1] Lionel C. Briand. Software documentation: How much is enough? In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Michael L. Collard, Jonathan I. Maletic, and Andrian Marcus. Supporting document and data views of source code. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 34–41, New York, NY, USA, 2002. ACM Press.

[3] Davor Cubranic and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, New York, NY, USA, 2002. ACM Press.

[5] Douglas Kirk, Marc Roper, and Murray Wood. Identifying and addressing problems in framework reuse. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 77–86, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *SIGDOC '99: Proceedings of the 17th annual interna-*

tional conference on Computer documentation, pages 147–153, New York, NY, USA, 1999. ACM Press.

[7] Baycloud LLC. synpos. Corporate Website, 2006. `http://www.synpos.com/`.

[8] Fredrik H. Madsen and Peter J. Nörnberg. Calliope: supporting high-level documentation of open-source projects. In *MIS '05: Proceedings of the 2005 symposia on Metainformatics*, page 10, New York, NY, USA, 2005. ACM Press.

[9] David Lorge Parnas. Precise description and specification of software. pages 93–106. Addison-Wesley Professional, 2001/1998.

[10] Robert Pierce and Scott Tilley. Automatically connecting documentation to code with rose. In *SIGDOC '02: Proceedings of the 20th annual international conference on Computer documentation*, pages 157–163, New York, NY, USA, 2002. ACM Press.

[11] Michael Priestley and Mary Hunter Utt. A unified process for software and documentation development. In *IPCC/SIGDOC '00: Proceedings of IEEE professional communication society international professional communication conference and Proceedings of the 18th annual ACM international conference on Computer documentation*, pages 221–238, Piscataway, NJ, USA, 2000. IEEE Educational Activities Department.

[12] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34, New York, NY, USA, 2006. ACM Press.

[13] Andrew Sutherland. Softsphere - dynamically visualizing coupling in a 3d environment. Tech Report, 2004. `http://www.cs.usask.ca/research/techreports/2006/TR-2006-03.pdf`.

[14] Dimitri van Heesch. Doxygen. Project website, 1997. `http://www.stack.nl/~dimitri/doxygen/`.

[15] Tao Xie and Jian Pei. Mapo: mining api usages from open source repositories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57, New York, NY, USA, 2006. ACM Press.