Software Measurement and Metrics - Assignment 3
11 November 2011

A Retrospective, Data-Driven Case Study for the Django Project

Authors:
Eric Huneke, Brendan Long, Gary Wilson Jr.

1 Introduction

1.1 This Study

This report describes our case study of Django, an open source web application framework written in Python. The overall goal of our study was a data-driven exploration of Django's code base, issue tracker, and wiki space, targeted to answer our research questions related to code size, complexity, and defect counts. We explored both the current state and the history of the Django project's artifacts, and produced several visualizations of the most interesting data.

This paper is organized as follows: Section 2 lists our research questions, Section 3 details our data gathering techniques and procedures, Section 4 presents our data and analysis, and Section 5 provides a brief summary of the report.

1.2 Django Project

Django's roots lie in a tool developed by two World Online developers that allowed them to quickly build web applications. Work on the tool began in 2003, and in 2005 the product was placed under an open source license and released to the public, as Django. As of November 2011, Django boasts over 20 core developers, as well as a bustling community with a 21,000+ member mailing list, documentation that has been translated to over 65 languages, and attribution to over 500 individuals who have contributed code to the project.

Throughout Django's history, the project has used a single subversion repository and a single bug tracker. Much of the raw data from Django's subversion repository and bug tracker are readily available [1] from the project's website.

2 Research Questions

Our research questions were as follows:

- What is the size of the code base, and how fast is it growing?
- What is the cyclomatic complexity of all the modules? How has complexity changed over time? Which modules should be refactored or redesigned due to excessive complexity?
- What is the quality of various modules? How has this changed over time leading up to the most recent release?
- What is the size of Django's wiki and how often is it maintained?
- What is the code coverage of the unit test suite? How has code coverage changed over time?

Because it was our ultimate goal to answer our research questions by the end of the study, the research questions provided general guidance as we collected our data. In our analysis, we answer these questions using the data visualizations we created.

3 Data Gathering Techniques

The main objects of interest relating to our research questions were the code, tests, issue database, and wiki. To gather the data relating to these objects, we used a variety of techniques and tools. In most cases, we wrote scripts to either generate reports or parse reports generated by external tools. Our scripts and raw data can be browsed at [2]. The data views on this website are powered by Trac [3], which also powers the same views in the code section of Django's website.

The first commit to Django that included code was commit number 3, which occurred on July 12, 2005. For our historical analysis, where applicable, we decided to analyze data at 6-month intervals starting July 13, 2005 and ending July 13, 2011. For some data, we placed an additional interval at October 26, 2011 in order to close the gap between the last interval and the start of this study.

3.1 Static Code Analysis

To answer our research questions regarding code growth and complexity, we analyzed checkouts from the Django repository [4] with a static analysis tool for Python called PyMetrics [5]. For each file in the project, PyMetrics measures basic syntactic details such as block indentation and number of characters in a file. It also measures the percentage of self-documenting classes and functions. Additionally, PyMetrics provides the source lines of code (SLOC) and the McCabe complexity metric for each Python function in the file.

All of PyMetric's output is aggregated in an easy-to-read text file. In order to plot trends over time, we developed a Python script to parse the text file and translate it to a comma-separated format.

3.2 Issue Tracker and Wiki

The code browser section of Django's website provides several tabs that allow users to find wiki and ticket data. The most primitive of these is the search tab, which lets you find wiki articles or tickets containing queried text. Tickets can be searched for in a more powerful way by using the custom query tab, which allows you to filter the resulting tickets shown based on a large set of fields. The timeline tab lets you view ticket and wiki changes that took place during a set period of time. Since they must be browsed manually, these tabs are not useful for gathering large amounts of data. Fortunately, Django exports several functions via a remote procedure call (RPC) interface that offers similar functionality to the tabs. Using the RPC functions, we were able to write a script to gather large amounts of ticket and wiki data into a spreadsheet, which we were then able to generate graphs from using Excel and Matplotlib [6].

3.3 Code Coverage

Examining the code coverage of Django's unit test suite was challenging because there are numerous (and sometimes disjoint) configuration combinations. For example, Django is designed to support multiple major version of Python, as well as various database backends, cache backends, and other optional features. Depending on these various environment and configuration combinations, there may be blocks of code that do or do not get executed during a run of the test suite.

For this study, we only looked at code coverage using a single, out-of-the-box configuration that was specifically made as a quickstart to running Django's unit tests. More specifically, this configuration makes use of the SQLite database backend, a simple in-memory or file-based database implementation that requires no third-party libraries and no additional setup.

In order to keep the coverage comparisons as similar as possible across historical revisions of Django, we used equivalent settings across all test runs:

- For recent revisions, we used an out-of-the-box test configuration included in the source code.
- For older revisions of Django, where no out-of-the-box test configuration existed in the source code, we used a custom settings configuration that is equivalent to the out-of-the box version included in more recent revisions of Django. Below is the content of the custom settings used for older revisions:

```
DATABASE_ENGINE = 'sqlite3'
TEST_DATABASE_NAME = 'testdb'
ROOT_URLCONF = 'foo'
ADMIN_MEDIA_PREFIX = 'foo'
```

- We did not collect coverage data for revisions of Django prior to 2007, as those revisions did not include a database backend implementation for connecting to a SQLite database.

- We followed Django's "Unit Tests" documentation [7] and installed additional packages to increase the number of modules executed by the test suite. Namely, we installed the following Python packages: pyyaml, markdown, textile, docutils, and setuptools.

We used the coverage.py [8] tool for measuring the code coverage during execution of Django's unit test suite. Below is an example of the command we executed to generate the coverage data:

```
coverage run
--omit="*/pyshared/*,modeltests/*,regressiontests/*,*test_sqlite
*,*runtests*,*urls*,*pkg_resources*,*site-packages*"
./runtests.py --settings=test_sqlite
```

Note our use of several omit patterns added when running the coverage command. This allowed us to filter out unwanted coverage data from:

- The test suite code itself
- System-installed Python libraries that are not part of the Django codebase
- Several specific files that are used to bootstrap execution of the test suite, including the settings file, the runtests.py test harness script, and the code defining the base URLs used by the test suite.

After the “coverage run” command completed, text-based reports were generated using the “coverage report” command. Results and analysis of the coverage data is presented in Section 4.3.

4 Data and Analysis

Due to the large amount of data we gathered, we found the best way to present it was in graphical form. In some cases, we also present single-number metrics or data in tabular form. In our analysis, our main goal was to answer our research questions, but we also highlight other pieces of information that were unexpected or otherwise interesting.

4.1 Static Code Analysis

The following data was collected as described in section 3.1 using the PyMetrics tool suite on snapshots of the project every six months over its lifetime.

4.1.1 Size

Figures 4.1 and 4.2 below illustrate the project growth over time, as measured by the total SLOC and the total number of source files in the project. Both trends appear nearly linear with a slight acceleration. In fact, the fastest growth rate appears in the last two years, indicating that the project will likely continue to grow for some time.

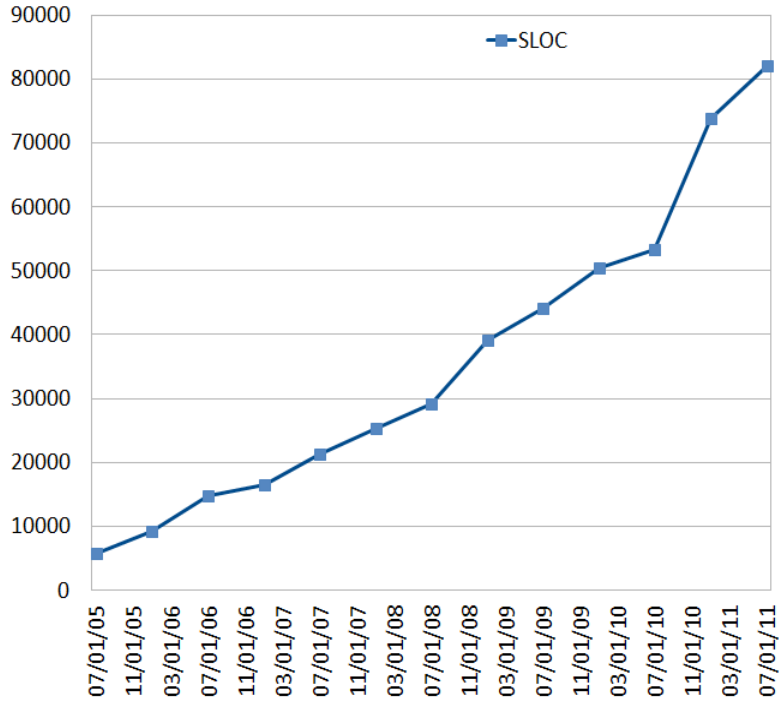

Figure 4.1: SLOC Growth

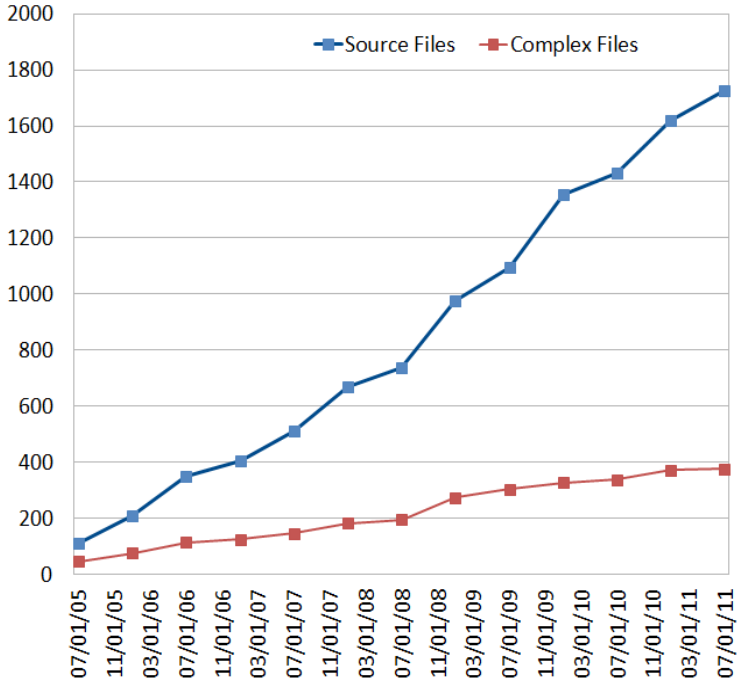

Figure 4.2: File Growth

4.1.2 Complexity

Figure 4.2 above presents the proliferation of complex files over time. We heuristically define a "complex file" as a source file containing at least one function with cyclomatic complexity greater than 10. The growth rate of these files has remained much lower than the growth rate of files in general. This might indicate that as the project grows, more effort is put into producing simpler source files such as translation definitions used in localization.

Below, Figure 4.3 displays the average complexity of the project. Here, the complexity of the project is measured by averaging the complexities of all the source files, weighted by their SLOC. Aside from some volatility early in the project's life, average complexity has been steadily decreasing since 2008.

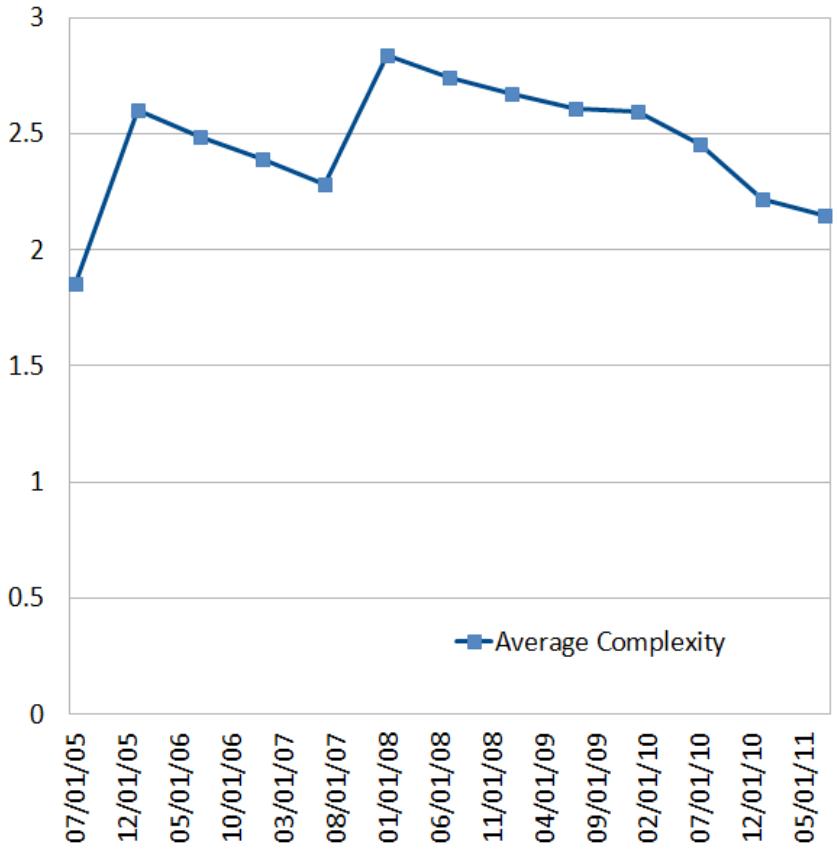

Figure 4.3: Average Complexity

Figure 4.4 tracks the highest complexity found in the project over time. Ignoring one outlier from the first check-in, the max complexity has been monotonically increasing. With maximum complexity of over 50, this trend could point to some warning signs. With a concerted effort to simplify the more complex modules, this trend could easily be reversed.

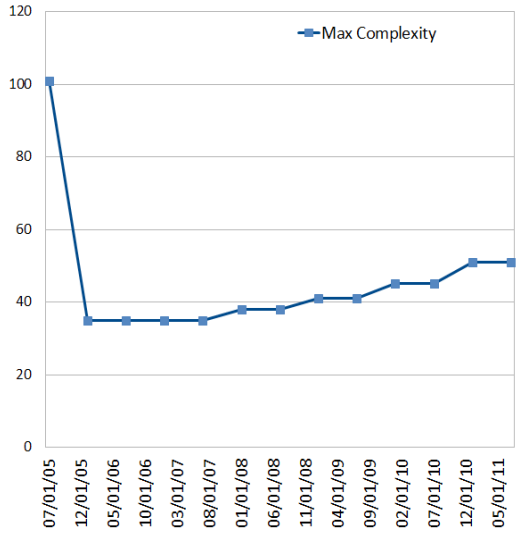

Figure 4.4: Max Complexity

Figure 4.5 illustrates the percentage of source files considered complex. The downward trend is not due to purposeful refactoring, but rather to the increase in the number of simple source files.

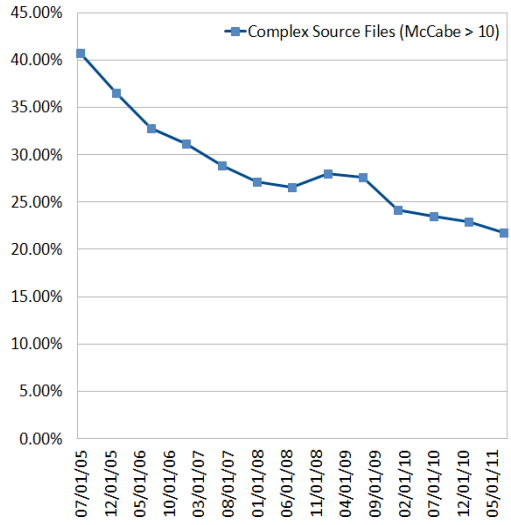

Figure 4.5: Proportion of Complex Files

Table 4.1 below lists the python files containing the most complex functions. Aside from these 10 modules, no file contains a function with complexity greater than 20. Therefore, these modules indicate the greatest areas for improvement and simplification.

File	**McCabe Complexity**
/django/core/management/commands/loaddata.py	52
/django/contrib/admin/validation.py	47
/django/views/i18n.py	37
/django/core/management/commands/syncdb.py	35
/django/utils/regex_helper.py	29
/django/contrib/gis/utils/ogrinspect.py	26
/django/core/management/commands/dumpdata.py	26
/django/contrib/auth/management/commands/createsuperuser.py	25
/django/core/servers/fastcgi.py	22
/django/contrib/contenttypes/management.py	19

Table 4.1

4.1.3 Documentation

Figure 4.6 on the following page illustrates the percentage of self-documenting classes and functions. Both of these trends saw a sharp increase up to September 2009 followed by decline. It would appear that right around the time that more simple files were added to the project, emphasis on self-documentation decreased. It could well be that these simple files are straightforward enough to understand without comments. Another explanation for the sharp increase in documentation seen around May of 2009 could even be the contributions of a small group of contributors. The developer community for Django is small enough for individual contributions to impact metrics such as these.

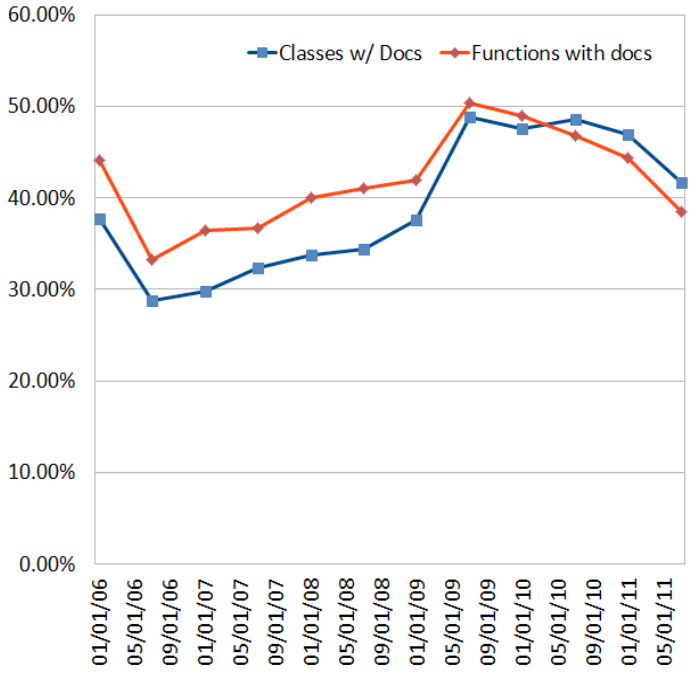

Figure 4.6: Self-Documentation

4.2 Issue Tracker and Wiki

One of the most basic questions to ask regarding tickets is how long they take to close. Figures 4.7 through 4.9 show the answer to this question based on three different ticket categorizations. Note that tickets can be reopened after being closed; for our purposes a ticket is closed the last time it is closed in its history. One point of commonality between the graphs is the large number of tickets closed the same day they are opened. This usually happens in one of two ways: the ticket should have never been opened (e.g. it was invalid or duplicate), or the fix for the ticket comes in at the same time as the ticket itself.

Figure 4.7 shows time to close (TTC) based on ticket type. The outstanding line at the bottom is tickets relating to new features. Sure enough, new feature tickets take longer to close than other types. At one month, only 35% of new feature tickets are closed compared to 50-80% for the other types. At the one year mark, the percent of new features closed goes up by about 10% per year until the present.

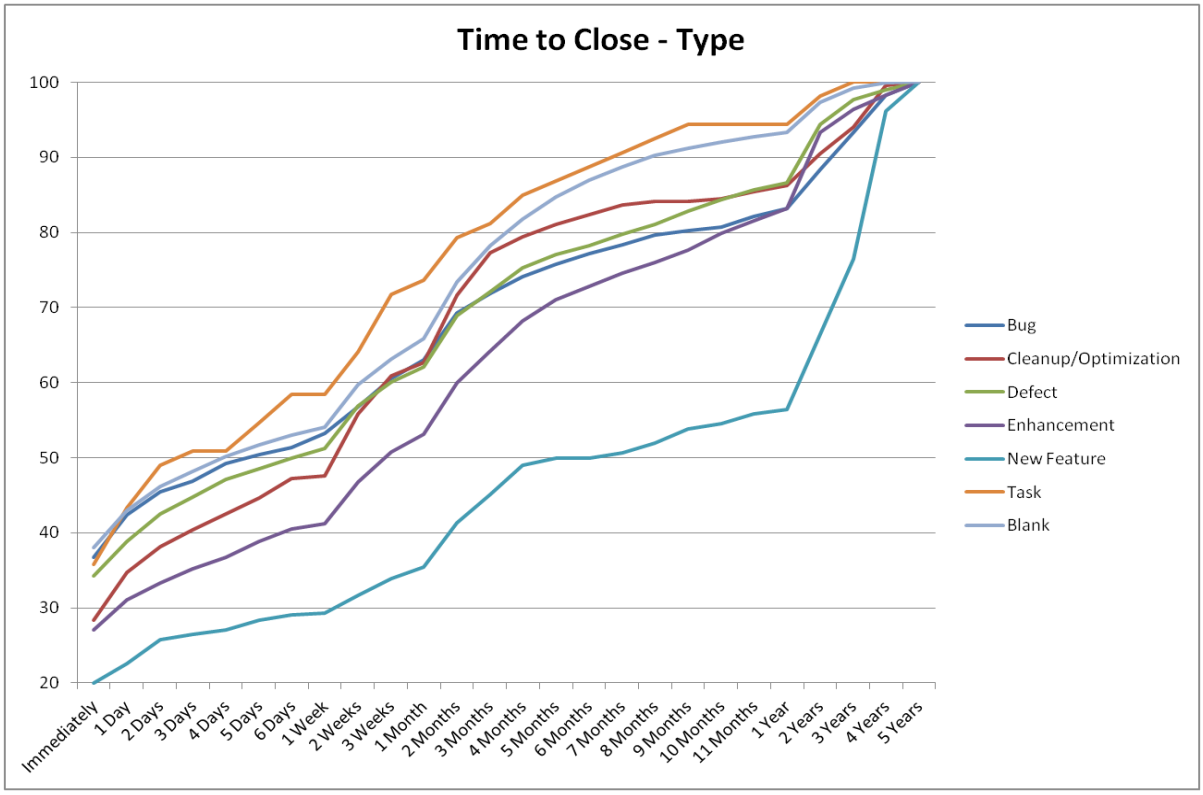

Figure 4.7: Time to Close Tickets Based on Type

Figure 4.8 is centered around ticket resolution. Line growth intuitively corresponds to resolution type. For example, invalid tickets are almost always closed the first day since there is no reason for them to stay open if they are invalid. On the other end of the spectrum, fixed resolution means that there wasn't a problem with the ticket and thus they are closed more slowly. The 'won't fix' category is perhaps unexpected, but can be explained by the observation that the 'won't fix' label is not determined immediately; the person who would be responsible for fixing the ticket may have to argue with the opener to determine whether or not to fix it.

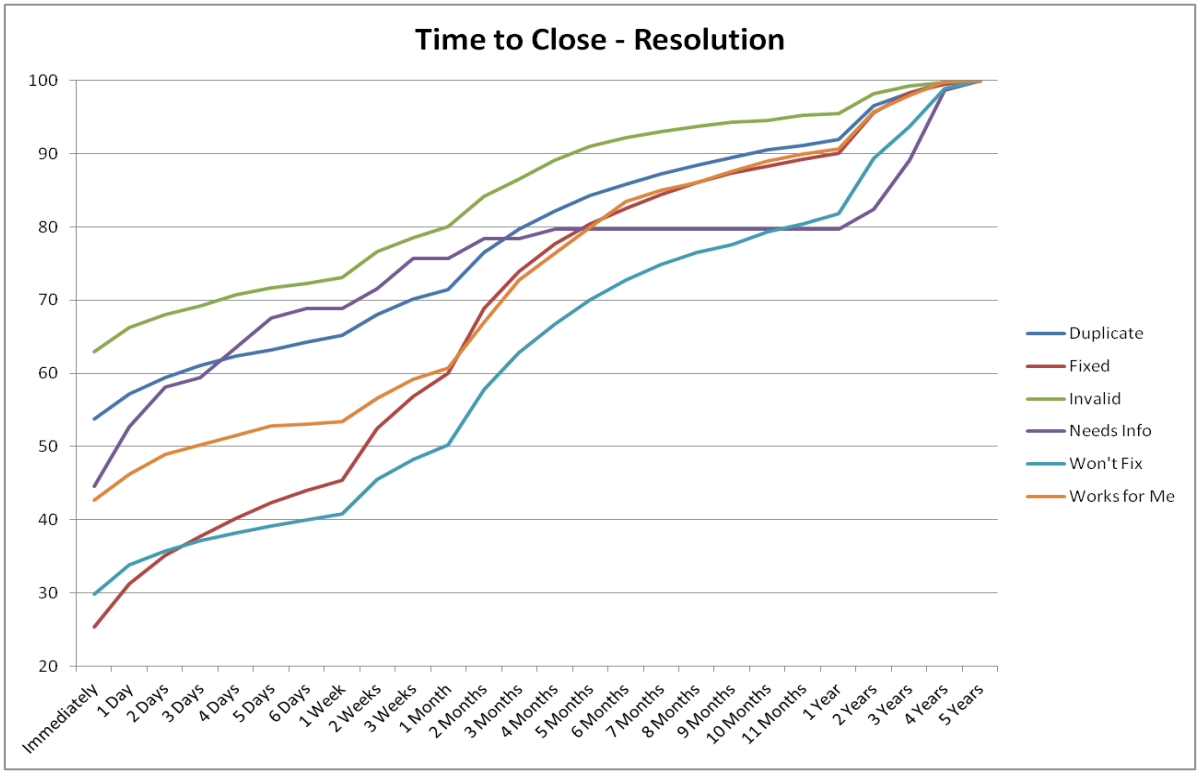

Figure 4.8: Time to Close Tickets Based on Resolution

Figure 4.9 shows TTC for tickets based on severity. Critical tickets are the fastest to start, but are taken over by the release blockers. Most of the lines are roughly in order of severity, with more severe tickets being closed faster. However, trivial tickets are actually faster to close than, for example, minor tickets, even though by strict ordering they should be last. This makes sense considering trivial tickets can be easily fixed and closed, and many developers would try to get them out of the way to reduce the number of open tickets to deal with.

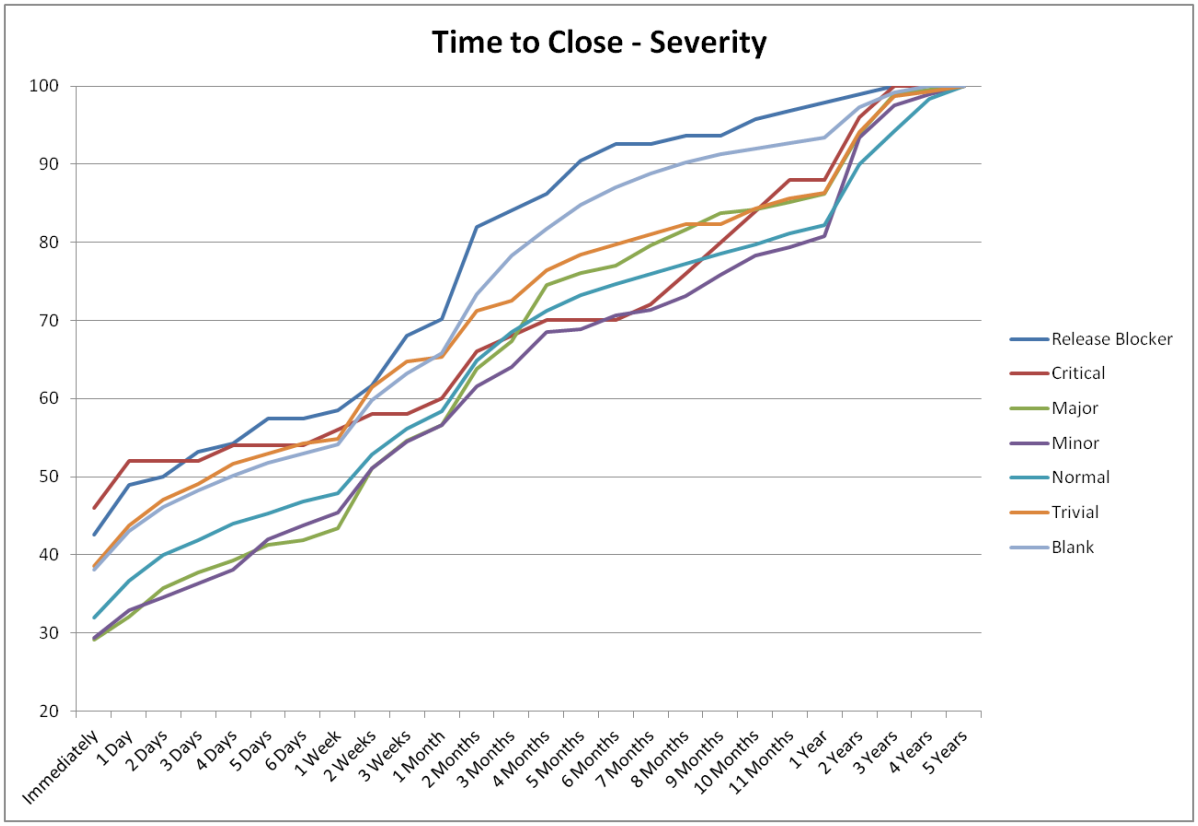

Figure 4.9: Time to Close Tickets Based on Severity

Figure 4.10 below is a Pareto chart showing the tickets belonging to each of the top 30 components. Since tickets usually indicate something is wrong with the code, this figure gives a rough estimate of component quality. The top violating component is the documentation, which may be expected since the documentation covers the entire codebase. Looking at the percentile line, it can be seen that a small number of modules contribute a large number of the total tickets.

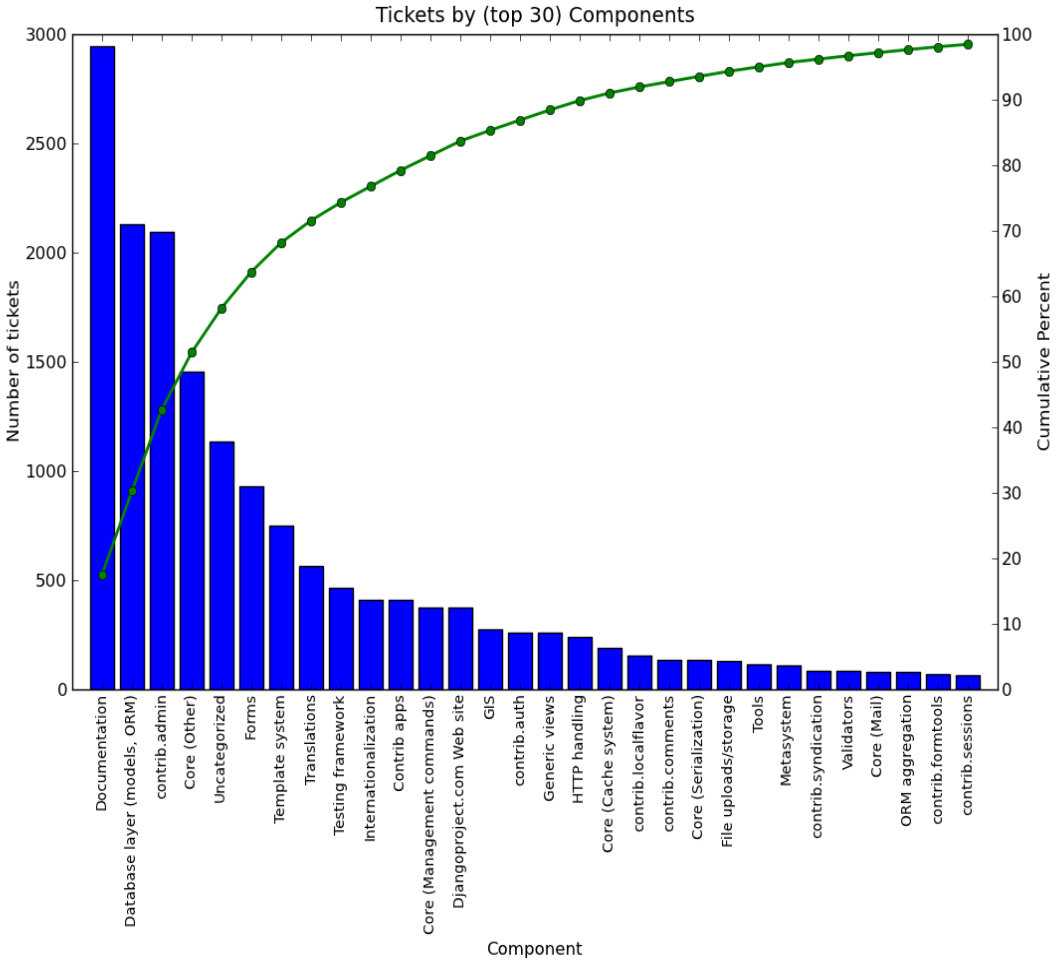

Figure 4.10: Tickets by Component

At the time of our study, Django's wiki consisted of 600 articles. Figure 4.10 shows the number of number articles for ranges of revisions. A large number of articles have a small number of revisions. The number of articles in each revision category slowly dwindles, but never disappears. There are a large number of articles that are updated frequently due to the nature of the page. For example, the article with the highest number of revisions is "DevelopersForHire", which posts job openings related to the project.

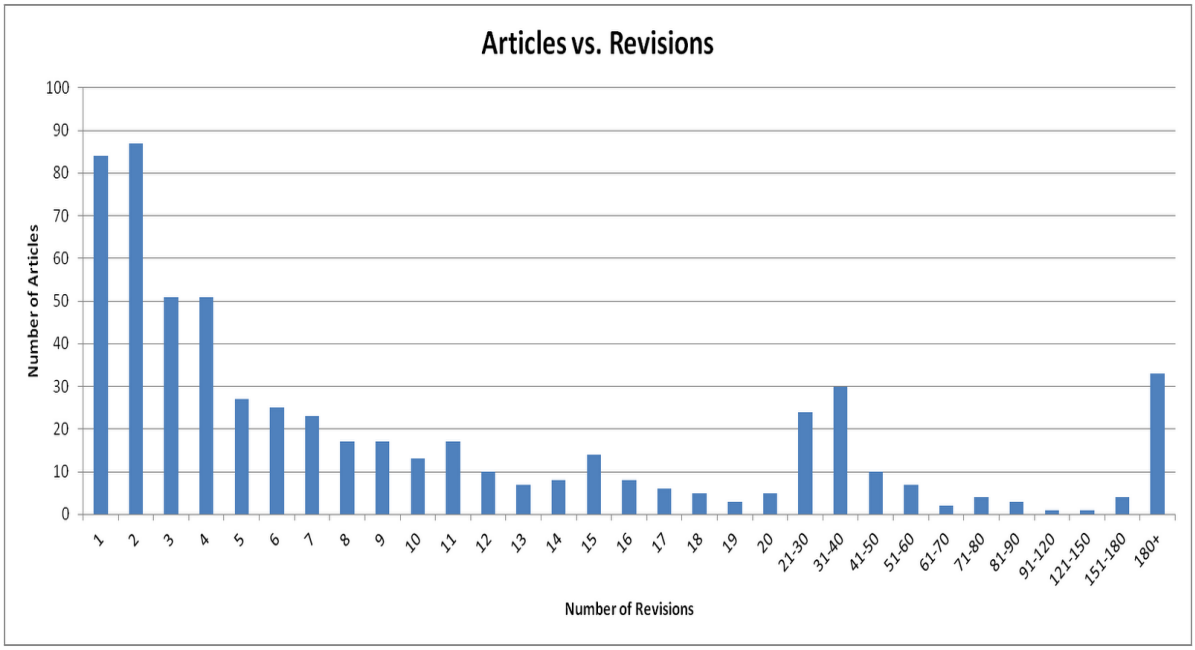

Figure 4.11: Articles vs. Revisions

Table 4.2 below shows some other wiki metrics we were able to gather.

	#Attachments	#Revisions	#Characters (Rev.1)
Maximum	16	818	61857
Sum	155	11019	2112963
Average	0.258	18.365	3521.605

Table 4.2: Wiki Metrics

4.3 Code Coverage

An important note about the coverage.py tool used in this analysis is that the total lines used in the calculation of the tool's reported percent coverage only included modules that were actually imported during a run of the test suite. Thus, any "dead" modules or modules not tested at all by the test suite are not reflected as part of the total number of SLOC.

Below, in Figure 4.12, we have plotted the overall line coverage percentage in the time periods between 2007 and 2011. Over this time period, Django's percent line coverage executed by the test suite more than doubled, increasing from 39.0% to 90.0%. There was only one decline in coverage seen during the time periods studied: from January to July, 2007, the percent line coverage decreased by about 0.3%. In all other periods, coverage increased; the largest increase was seen between July 2007 and January 2008, where coverage nearly doubled (from 38.8% to 74.0%) in a single 6-month period.

The blue vertical lines represent Django's major releases. Interestingly, the large increase in coverage noted above occurred between the 0.96 and 1.0 releases (the 1.0 release, of course, being a major project milestone).

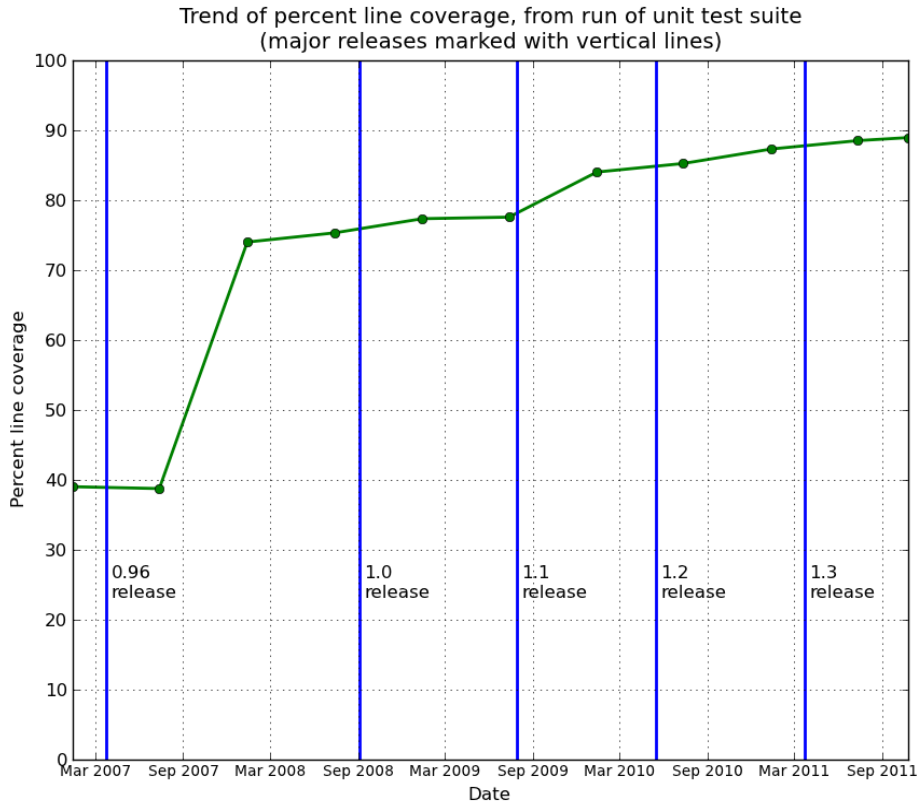

Figure 4.12: Coverage

In the latest snapshot analyzed (from October 26, 2011), 59.3% of the non-zero length modules executed during a run of the test suite had 95% or greater line coverage, and 40.3% of the non-zero length modules had 100% coverage. See Figures 4.13 and 4.14, below.

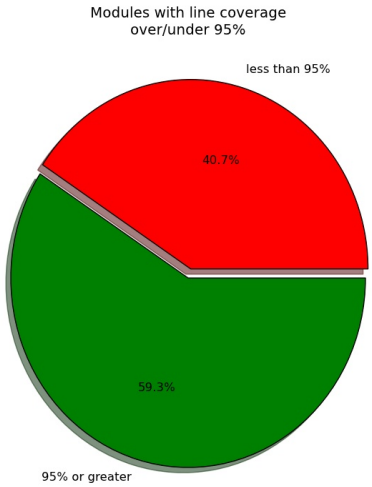

Figure 4.13: 95% Coverage

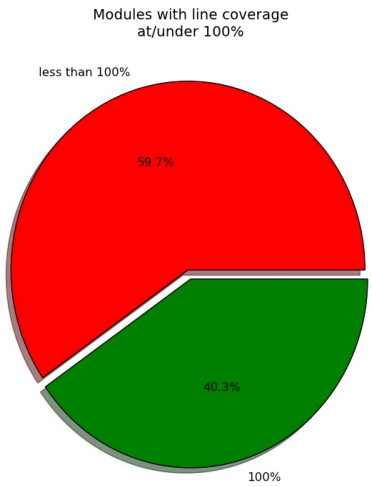

Figure 4.14: 100% Coverage

For a more detailed view of the line coverage of the modules within the Django project, we plot a couple histograms in Figures 4.15 and 4.16 (non-cumulative and cumulative, respectively) showing the number of non-empty modules within each 5% range of line coverage.

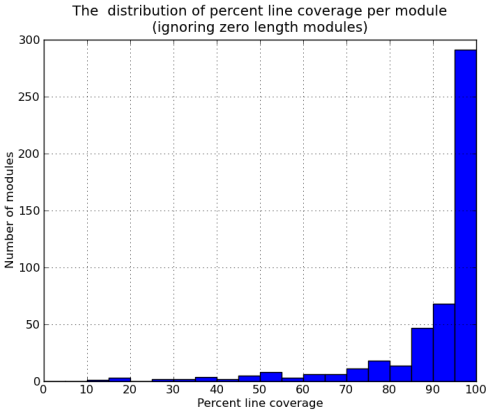

Figure 4.15: Coverage Distribution

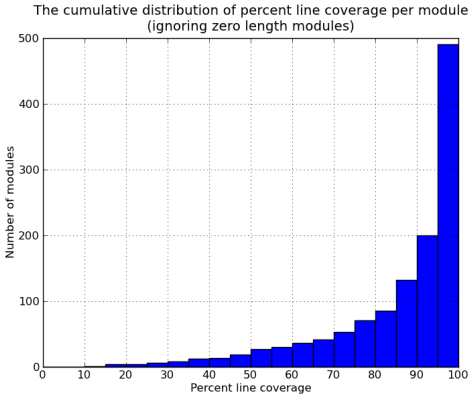

Figure 4.16: Coverage Distribution (Cumulative)

5 Conclusion

This report presented the data we collected as part of a case study examining various aspects of Django, an open source web application framework. We found that the codebase has been growing at a steady rate since its start and will likely continue to grow. Overall complexity has been manageable, with just a few files containing egregiously complex functions. A refactoring effort focused on the files presented in table 4.1 is recommended. Ticket data relating to Django reflect basic intuitions about how they should behave. We were able to get a rough measure of component quality by looking at the number of tickets relating to each component, and showed that the top three components by ticket count (documentation, database layer, and contrib.admin) accounted for over 40% of all reported tickets. Test coverage of Django's components is high, with most components having at least 95% coverage. Coverage has trended upward for almost the entire project's history, and saw an extreme increase in the time periods before its 1.0 release.

6 References

[1] https://code.djangoproject.com/
[2] https://utforge.its.utexas.edu/trac/projects/metrics
[3] http://trac.edgewall.org/
[4] https://code.djangoproject.com/svn/django/
[5] http://sourceforge.net/projects/pymetrics/
[6] http://matplotlib.sourceforge.net/
[7] https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/unit-tests/
[8] http://nedbatchelder.com/code/coverage/