Software Measurement and Metrics
Assignment 2 – Case Study
Submitted on November 11, 2011

Django Project Case Study

Submitted by:

Kevin Boos
Richard Campbell
Evan Grim
Hasanain A. Jwair

1.0 INTRODUCTION

This case study examines the Django project, a high-level Python web framework that allows for rapid creation and clean development of web applications. The entirety of Django is written in Python and emphasizes reusability and interoperability of various components, as well as the DRY (Don't Repeat Yourself) development principle. Django is an open-source development effort, meaning that anyone can contribute to its code base. As such, Django has a massive amount of information waiting to be collected and analyzed, in the form of a Git/SVN version control repository, ticket tracking database, and actual source code.

Our motivation for conducting this case study, aside from the obvious assignment requirements, is to answer a variety of research questions corresponding to several standard software metrics: code size, modularity, complexity, unit test coverage, and component quality relative to defect quantity. We believe that obtaining and analyzing data from the Django repositories and databases will allow us to generate graphical representations of important relationships between developer behavior, component organization, testing and debugging procedures, and a wide variety of other project characteristics. From these relationships, we can then draw conclusions about how to improve the software development process and other facets of the Django project.

This study begins by outlining our methods for acquiring and modifying data, including the sources of the data, the tools used to collect the data, and the perceived quality of the data itself. We then progress to displaying the results of our analyses, which primarily consists of graphical and tabular representations of the data, as well as explanations of each figure. We discuss the trends demonstrated by each figure and draw conclusions from the data about what aspects of Django have cause-and-effect relationships. We also make cross-comparisons across different measurement categories and highlight the similarities of corroborating relationships and the conflicts of contradictory data. Finally, we answer the research questions initially used as motivation for the study, and recommend several courses of action that could improve the development process and quality of Django.

2.0 DATA COLLECTION PROCEDURE

One of the primary goals of this case study is to perform as scientifically rigorous an evaluation of the Django project as possible. Towards this end, we chose to use a variety tools to assess many quantitative and qualitative categories of Django's development process. We preferred automation whenever possible to avoid any bias that might arise from more manual mechanisms. The following sections describe the method we used to collect the specific measurements needed to answer the aforementioned research questions. We also discuss any shortcomings of the underlying tool, any outside influences effected upon the data by the tool, the source from which each tool extracted the data, and the overall quality of the data as gathered by each tool.

2.1 Code Size, Rate of Growth, and Complexity

To gather fundamental code size and growth rate data, we turned to Django's source control repository to analyze historical revisions and extract the story of its software evolution since emerging to the public back in 2005. While Django's primary revision control is managed in a subversion repository, we turned to a mirror of this history hosted as a git repository on github. Git provides a complete clone of the revision history to be obtained once and stored locally. This allows for historical data to be mined much quicker since each version of interest can be reconstructed without requiring a connection to the central server.

With the local repository in hand, we progressively performed static analysis on snapshots of the source code as it stood at the first of every month since the project began. For this we used two tools designed to extract useful measurements from python source code: pymetrics and sloccount. On our first pass we intended to gain all the desired measurements from pymetrics, but after sanity checking its software lines of code (SLOC) data we discovered that its counting methodology was not in-line with mainstream industry standards[1]. Instead, we supplemented our data with the purpose built sloccount utility which provided counts more in line with our expectations.

After obtaining the raw data from the output of these tools, we performed post-processing using our own custom designed scripts to obtain data for the following measurements:

- software lines of code
- number of classes
- number of functions/methods
- number of modules
- percent of classes with docstrings
- percent of functions/methods with docstrings
- percent of modules with docstrings
- McCabe's cyclomatic complexity for each method

Gathering this data at regular intervals over the complete history of the project provides a

[1] A quick Google search confirmed we were not alone in our assessment of the SLOC data pymetrics provides.

unique viewpoint from which to analyze the growth of the code and various maintainability properties as the project grew and matured.

2.2 Ticket Tracking and Defect Analysis

In order to answer the third research question, we needed to assess the modular quality of Django by measuring the defect characteristics of each component. We also wanted to examine the trends leading up to the most recent release, version 1.3, to determine how the development process changes immediately before a deadline. Django's website utilizes *Trac*, a software project management and bug-tracking system, to keep a version-controlled record of all defect reports, feature requests, and other issues that arose during development and testing. Each of these entries is known as a *ticket*, and consists of several types of information:

- ID number
- Summary and description of issue
- Status (new, assigned, closed, reopened)
- Type (enhancement, bug/defect, cleanup/optimization, new feature, etc.)
- Component (Core, Database, Contrib., Documentation, etc.)
- Version (release number)
- Bug severity (blocker, critical, major, normal, minor, trivial, etc.)
- Submission date
- Most recent change date
- Resolution action (duplicate, needs info, fixed, CNR, invalid, etc.)

All of the above data is available using queries provided by the Trac ticket logging system. The query feature allows a developer to search for submitted tickets that match certain criteria, such as "easy" tickets submitted before March 2008 with a bug severity level of "critical." We downloaded a query of every ticket in the tracker's history, which began July 13, 2005, and did not filter or exclude any tickets for the purposes of obtaining universally-relevant data. As of October 31, 2011, this produced a spreadsheet of approximately 16,000 ticket entries, each entry tagged with the descriptors listed above.

However, one unfortunately limitation of Trac is that it does *not* provide the dates when defects were resolved. In order to obtain this, we parsed the Git repository history and recorded check-in dates when tickets were marked "resolved." We then matched each ticket with its repository fix date based on the unique ticket ID number, and subsequently appended the fix dates to the spreadsheet. At this point, all of the necessary data was satisfactorily collected, and we were fully prepared to analyze defect characteristics, answer the third research question, and even contribute to the analyses of other measurement areas. The results garnered from this data are presented, discussed, and examined in Section 3.3.

Since this data was gathered without the knowledge or participation of the developers, there is no bias or Hawthorne effect influencing the data. Furthermore, our measurements do not intrude upon or restrict the development process, as everything was conducted offline, separately from writing and debugging source code and documentation. Aside from Trac's deficiency

in recording defect fix dates, which was mitigated through use of the Git repository history, it proved to be an excellent tool for adequately collecting data. Therefore, we are absolutely confident that all data collected from the ticket tracker is accurate, free from external influences and skew, fully comprehensive, and valid.

2.3 Unit Testing Coverage

To measure how well covered the Django code is by the tests, we used the coverage.py tool created by Ned Batchelder. The tool was run on Ubuntu with the source code and tests extracted from the Django subversion server on October 31, 2011.

The coverage.py report gave us results for each individual file in the Django source. The results included how many lines each file contained, how many lines not covered, the percentage of lines covered, the number of branches in a files, and how many branches were not covered. Grouping the files according to which component the files were in allowed us to make comparisons with the ticket tracking results.

The only bias in our test coverage data came from running the test tool on Linux. Some files had code that ran depending on whether the operating system was Windows or Unix. For us, the Unix lines ran so the Windows lines were reported as not covered. This is not a significant source of bias as less than 200 lines out of 86267 were operating system dependent.

2.4 Coding Standards and Style Guidelines

Following a set of coding standards plays an important role in open source projects, specially those that undergo continuous development from a large number of developers. Additionally, it is crucial in such projects to develop code that is well commented and documented. To measure all these aspects of the Django project we used PyLint. Pylint a code analysis tool that is used to:

- Reports standard violations, stylistic problems, and “good practice” violations[2].
- Produces a report about comments and documentation of code elements.
- Calculates the number of duplicate lines.

PyLint was run on Ubuntu with the source code extracted from the Django subversion server on October 31, 2011.

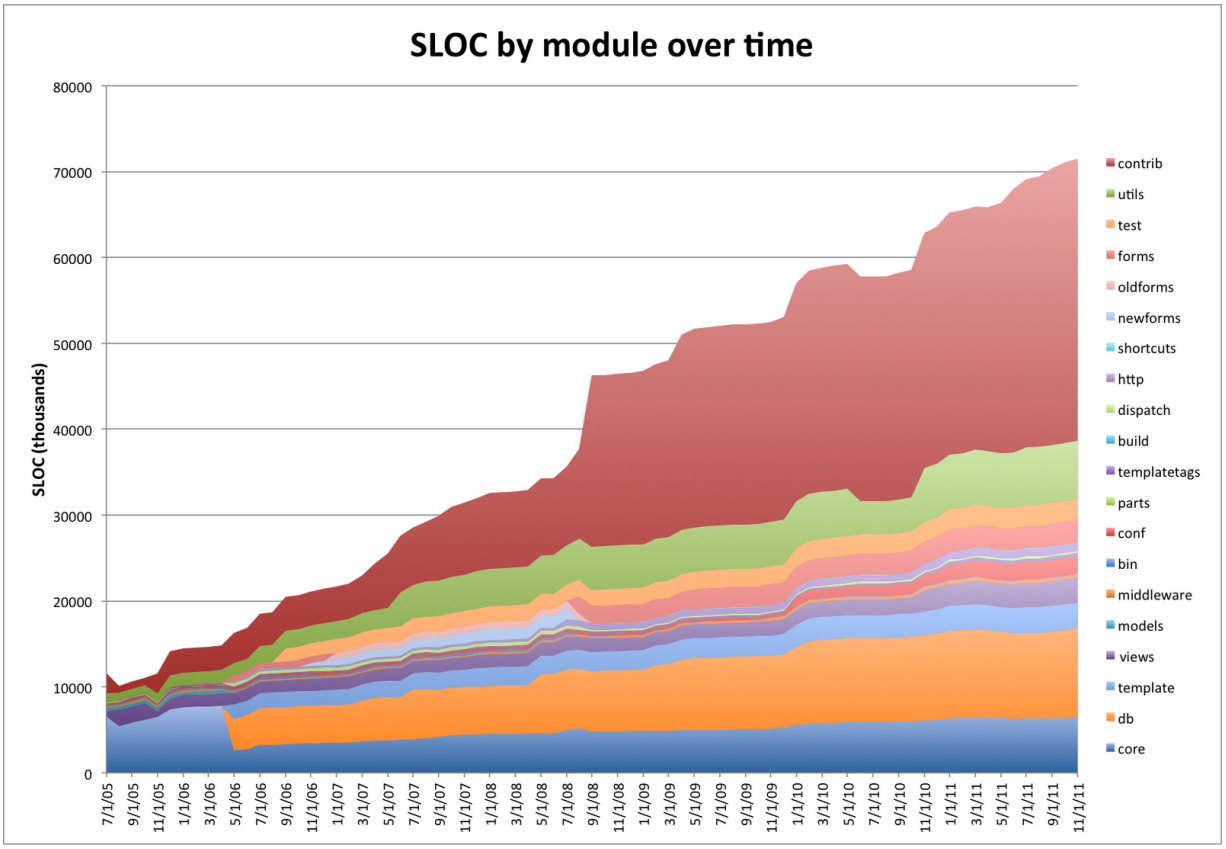
[2] The default coding standards and style guidelines used by PyLint are close to PEP 8, The style guide for python code, by Guido van Rossum and Barry Warsaw. [http://www.python.org/dev/peps/pep-0008/]

3.0 PRESENTATION AND ANALYSIS OF DATA

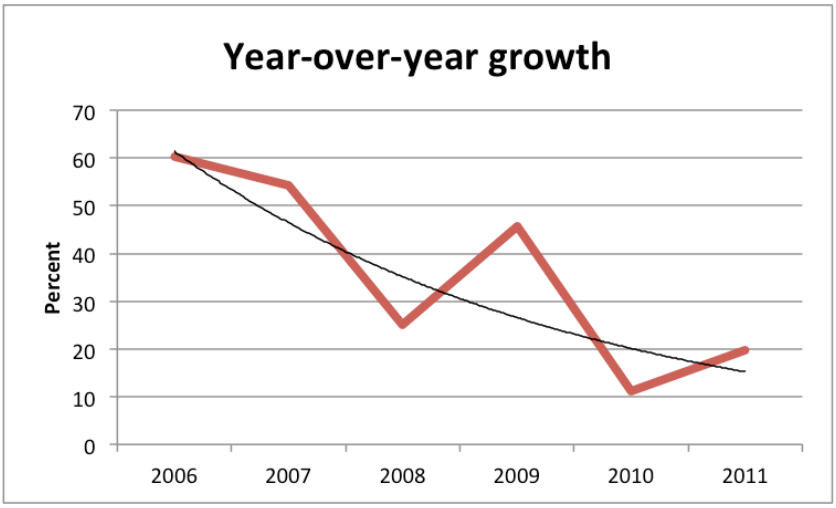
The following sections display our collected data through various visual representations. We offer explanations of how and why each graph was generated, as well as highlighting important trends in the data. We also draw conclusions about Django's development process from the most significant trends in the data, and evaluate the inherent assumptions we made while analyzing these trends. Finally, we identify contradictions within the data by making comparisons across multiple categories and sources of data.

3.1 Code Size and Rate of Growth

To understand the nature of the Django project it's useful to first get a feel for its size and scope. For this we turn to the historical data we collected (as described above) and transform the raw data into a more visual form amenable to provide analysis and insight. First, we'll take a look at that most basic measurement of software size: source lines of code (SLOC).



This stacked area graph shows the growth in SLOC over time and provides a breakdown of the size of each of the underlying Django modules. As you can see, the project started with about 11K SLOC in July of 2005 and has grown to over 70K SLOC today. To characterize this growth, it's useful to take a deeper look at its year-over-year growth.



Here we see that growth is slowing, as would be expected from a project reaching maturity. digging deeper, we see that nearly half of Django's code reside in the "contrib" module, a holding ground for plug-ins useful for people using Django but not essential for the basic featureset. Furthermore, on average a third of Django's recent growth is attributable to this module. The existence of the contrib module has roots in Django's "batteries included" philosophy. This certainly made sense in the early days of the project, but projecting forward it appears likely that the contrib module will soon grow larger than the rest of the django project combined. This leaves the project in a situation where a large and growing part of the codebase is identified as not essential to the project and admittedly used by only a subset of the project's users. We believe serious consideration should be given to segmenting development and distribution of the project into smaller units. A possible division could be:

- django-core: the core/essential django components (e.g.: tempates, ORM, etc.)
- django-batteries: the contrib module
- django-full: combined

This would mirror similar projects that have grown to include a large quantity of extras (i.e.: vim).

Recommendation: Break the Django project up into distinct smaller components for purposes of development and distribution in order to improve code modularity and encourage reuse.

3.2 Code Maintainability

Turning to measurements with more nuance, we now explore data gained from static analysis that indicate the maintainability of the code. For this, we highlight two categories of maintainability: in-line documentation (comments) and complexity.

First off, in-line documentation: python provides a mechanism for attaching a string to each element defined in the code (e.g.: classes, methods, etc.). These strings are known as docstrings, and provide a built-in method for in-line documentation supported by the language itself that gives a fantastic level of automatic documentation information that can be provided during development and even at runtime. Python heavily encourages the use of docstrings,

and doing so directly contributes to code readability and improves maintainability throughout the software development lifecycle.

Analyzing the historical data available in the repository shows that Django has been pretty consistent in the percentage of code elements that include in-line documentation, so we'll focus on the *status quo* in our analysis here. That being said, on average roughly half of Django's code elements currently carry a docstring. Some modules are bigger offenders than others:

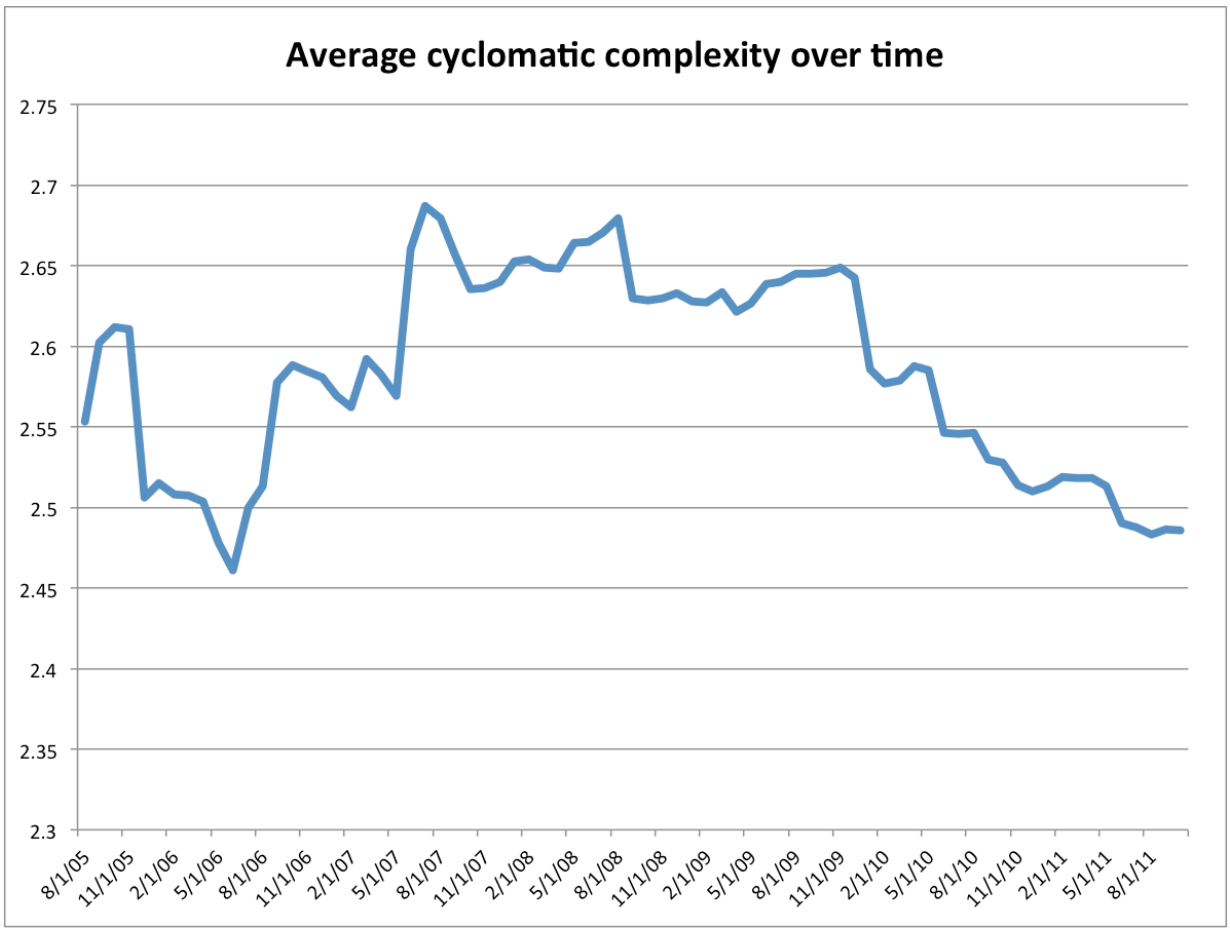
Lowest 5 docstring percentages by module

	Classes	Methods/Functions
1	templatetags (8.3%)	http (30.2%)
2	template (32.9%)	templatetags (36.7%)
3	http (36%)	forms (39.1%)
4	db (50.5%)	db (39.6%)
5	core (52.6%)	template (40.6%)

This table calls out the five modules in the Django code-base that have the lowest percentages of class and methods with docstrings respectively. These modules should be targeted for clean-up and in-line documentation improvements.

Recommendation: Target classes and methods with poor in-line documentation for clean-up for improve code readability.

Next we turn to complexity. Here Django is doing very well. Analysis shows that from the beginning the average McCabe cyclomatic complexity metrics has been a quite respectable 2.49. Looking at historical data we see a project which started out good, had some growing pains, but has effectively been reducing cyclomatic complexity down for years now.



However, when digging deeper into individual modules we do notice methods with very high cyclomatic complexity measurements.

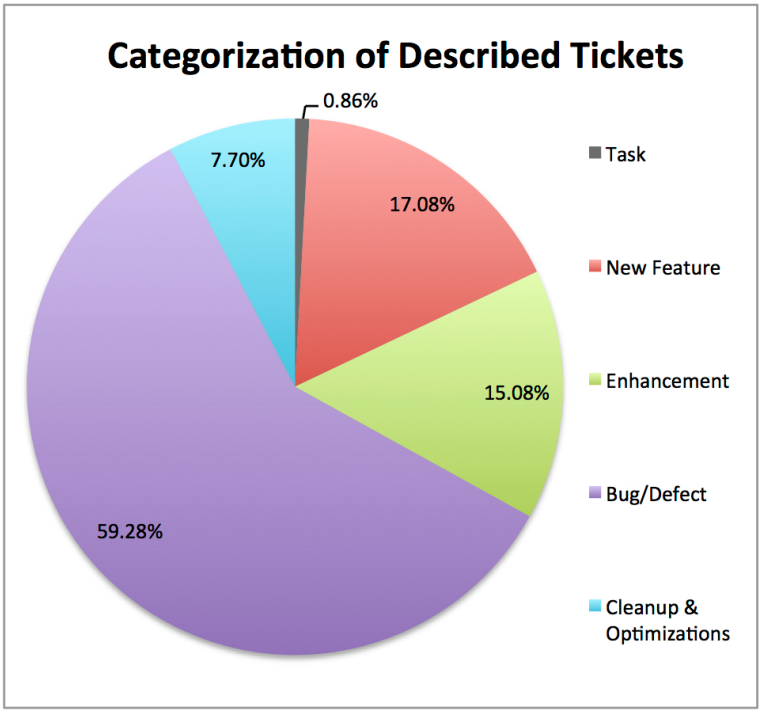
	Module:Method	Complexity
1	core:get_validation_errors	103
2	db:ModelBase.__new__	53
3	contrib:validate	47
4	utils:templatize	45
5	views:javascript_catalog	37

These represent methods with extreme complexity that should be targeted for refactoring and simplification

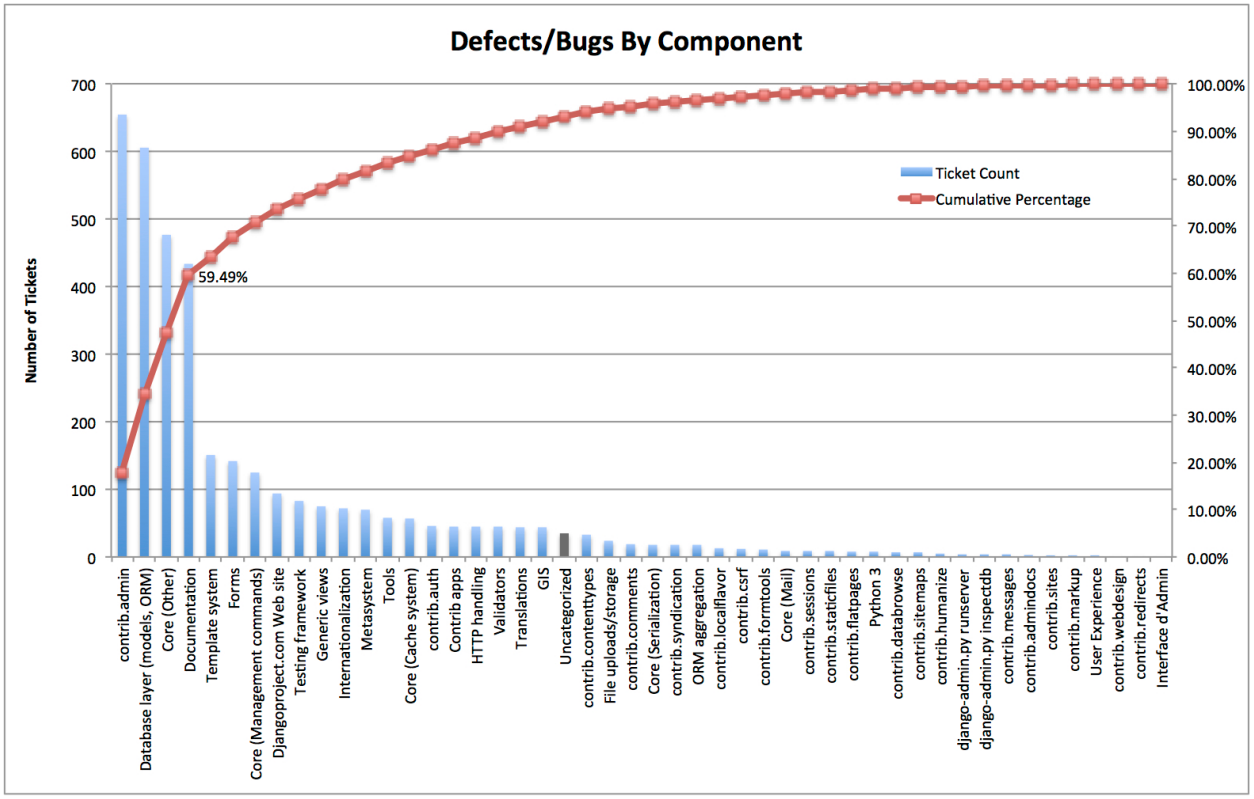
Recommendation: Target methods with extremely high cyclomatic complexity for refactoring to improve code maintainability.

3.3 Ticket-Based Defect Analysis

Mining the *Trac* ticket-tracking system produced a wealth of data that allowed us to evaluate the quality of each component with respect to defect quantity and characteristics. We begin by dividing the ticket database into multiple categories to illustrate the developers' unique tendencies to submit, process, and resolve tickets. In particular, developers tend to spare the details of the issue when submitting tickets, as over 63% of the 16,000 tickets were not categorized; we will denote the remaining 37% of tickets as "described tickets," meaning that the developer chose a category classification for the ticket when submitting it. As shown in the pie chart below, defects accounted for nearly 60% of the tickets, with new feature requests and enhancements or optimizations comprising the remainder of the described tickets. A total of over 3600 defect tickets is a sufficiently large sample size to serve as the basis for the following conclusions in this section.



The remainder of this section will analyze these defect tickets only, starting with the proportion of defects identified in each subcategory of Django's source code. The following graph illustrates the number of reported defects per component (blue bars) and the cumulative percentage of the total defect reports comprised by each component (red line). This graph includes every defect ticket submitted over the entire lifetime of the Django project, so it is representative of the full history of Django's development. The four most defect-ridden components of Django are Contrib.Admin (admin-related side contributions), Database, Other Core functionality, and Documentation. These four categories account for nearly 60% of all Django defects, and that figure jumps up to 71% if Templates, Forms, and Core Management Commands are included.

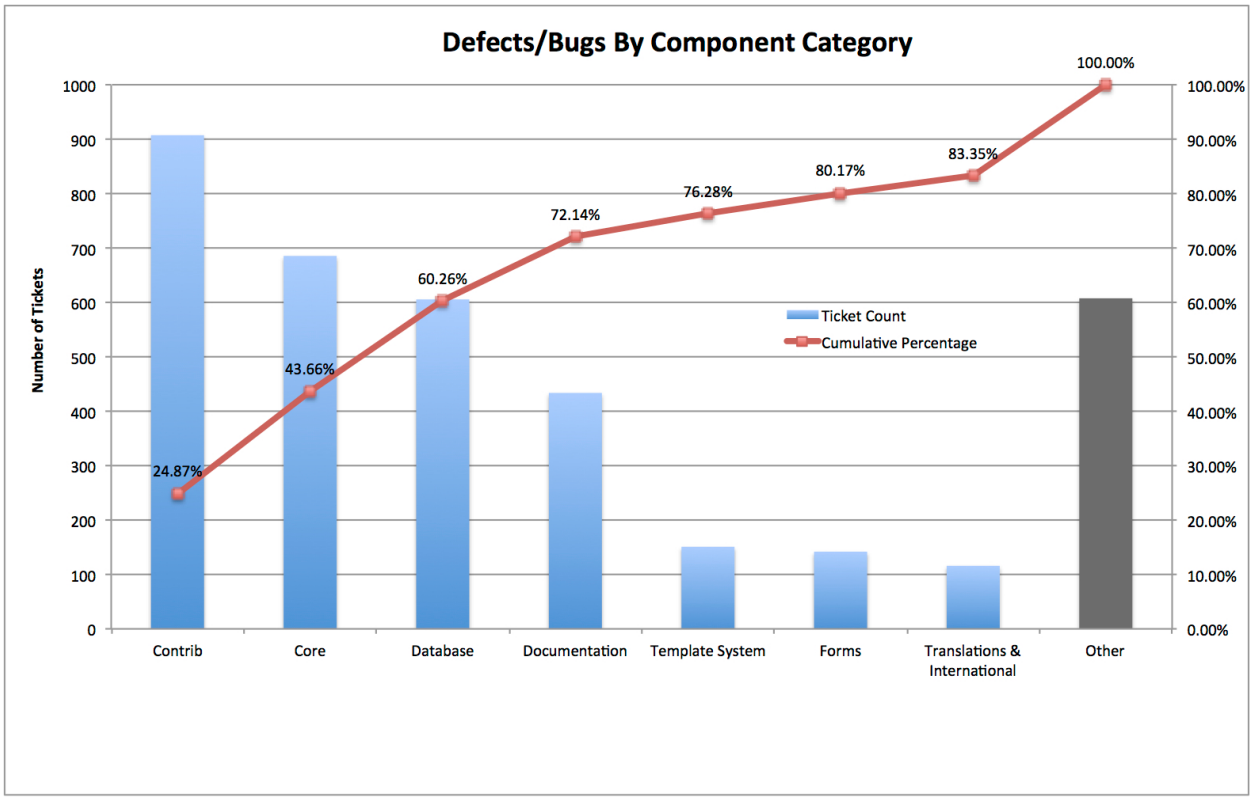


A natural conclusion from the above data is that the overall quality of Django could be dramatically improved by reducing the defects in the aforementioned components. One could also infer that those components are the largest and/or most complex, as complexity and module size is a clear cause of a high defect quantity. The remaining dozens of components have significantly fewer defects, due to a combination of relative ease of implementation and low modular complexity. These inferences are supported by the code size and complexity data presented in Section 3.1 and 3.2.

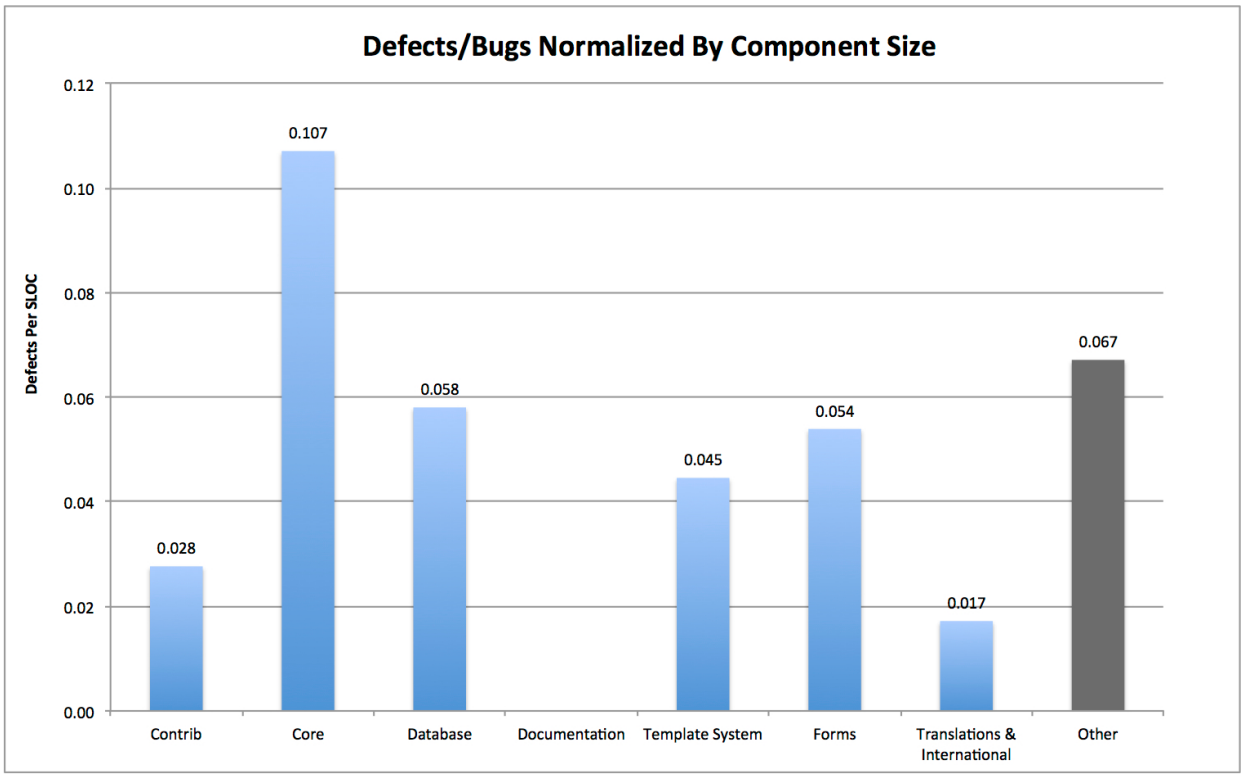
Recommendation: reducing the defects in the Contrib.Admin, Database, Core (Other), and Documentation modules will drastically reduce the overall defective percentage of Django modules.

We further merged the component subcategories discussed above into a few broader, overarching categories that generically represent the core functionality of Django. The following graph displays the same data as the previous Pareto graph, but provides a comparison of the major types of functionality instead of the functionality of each specific module. When examining the graph below, we can reach similar conclusions as the previous graph. The “Contrib” category, which is essentially a collection of useful but non-essential side modules, along with the Core and Database functionality categories comprise over 60% of all Django defect reports. Since the Contrib category consists of additional modules that provide non-essential functionality, and that Documentation has no effect on the correct operation of the Django

framework, we can assess that the most important portion (35%) of defects lurk in the Core and Database source.



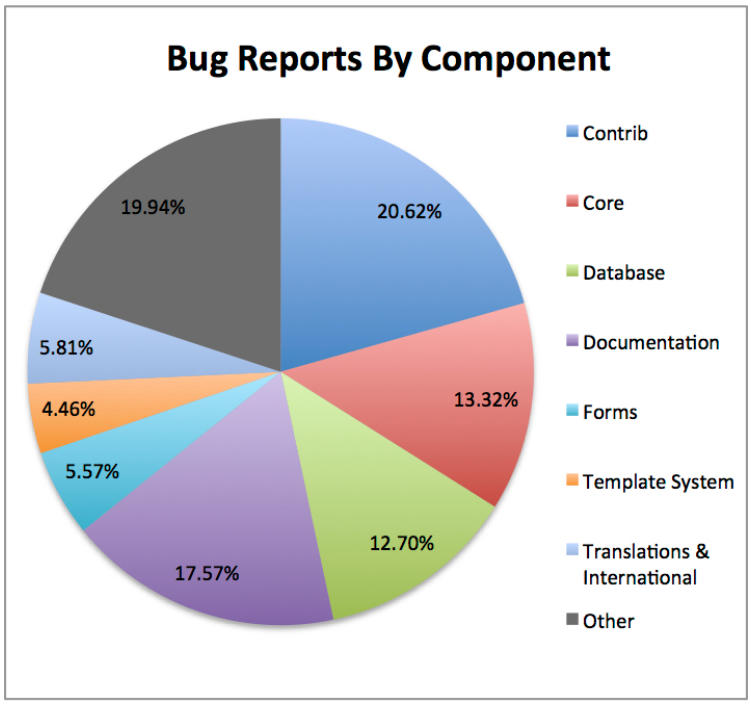
However, these results may be misleading, as a very large component will undoubtedly have many more defects than a very small one. Therefore, the next graph takes the size factor into account by normalizing each component's defect quantity according to its size, measured in SLOC. Consequently, this represents defect density, or the number of defects per SLOC.



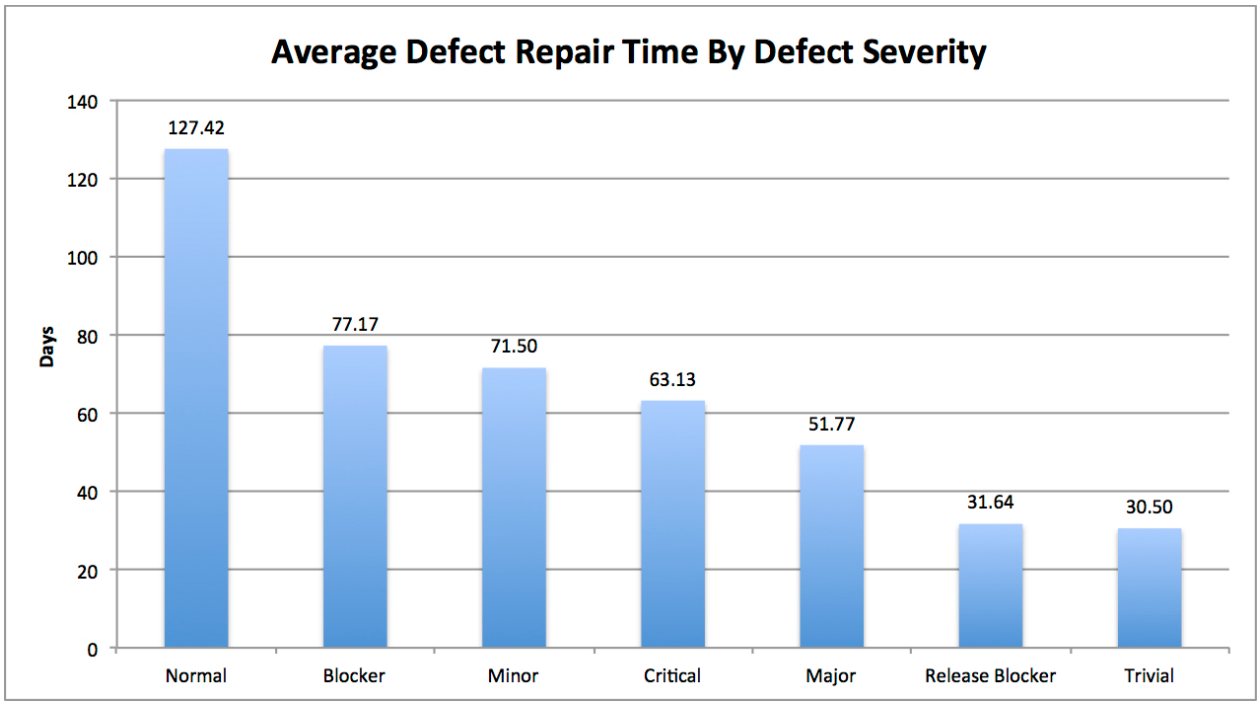
When size is considered, it is quickly apparent the the most defect-ridden component of Django is the Core, followed by the Database, Forms, Templates, and Contrib components respectively. The Documentation component does not have a size measured in SLOC, therefore it has been omitted. As we expected, this figure still agrees with the previous evaluation of the Core and Database components as the most defect-ridden components. Interestingly, the Template and Forms components exhibit a high defect density, which was not obvious from the previous graphs. Given the high concentration of defects in the Core, Database, Forms, and Template components, those four should be prioritized by Django's developers in order to reduce the overall quality of the framework.

Recommendation: prioritize repairing defects in the Core and Database modules.

Another way to asses defect characteristics is to measure the tendencies of developers to repair defects. The following pie chart demonstrates the number of defects that were repaired, grouped by component category. As with the previous metrics, the Contrib, Documentation, Core, and Database categories suffered from the highest number of defects, but this graph displays only the repaired defects. We can therefore infer that the majority of developer time is devoted towards repairing defects in these four categories.

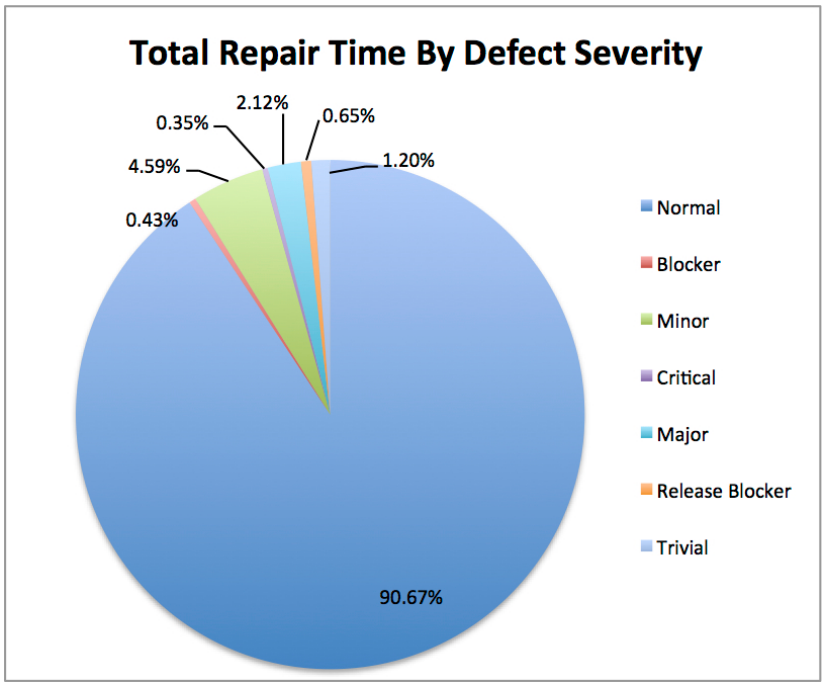


In addition to dividing the defect reports by component category, we can also divide them according to defect severity. The seven levels of defect severity as defined by Django are (in descending order): Release Blocker, Blocker, Critical, Major, Normal, Minor, and Trivial. On average, the time required to fix a defect was approximately 65 days, including defects from every category. However, a blanket average of all defect repair time is not sufficient to provide a practical analysis. Therefore, the following graph demonstrates the average time required to fix defects of a certain severity, which was measured across all categories of Django components. As expected, many of the defects are reported with a severity classification of “Normal,” meaning they that aren’t high priority, but should be tended to as soon as possible, barring extraneous interruptions by higher-priority defects. Release Blocker defects are defects that prevent Django from releasing a new version of the framework; therefore, these defects are the highest priority and are repaired with much haste.

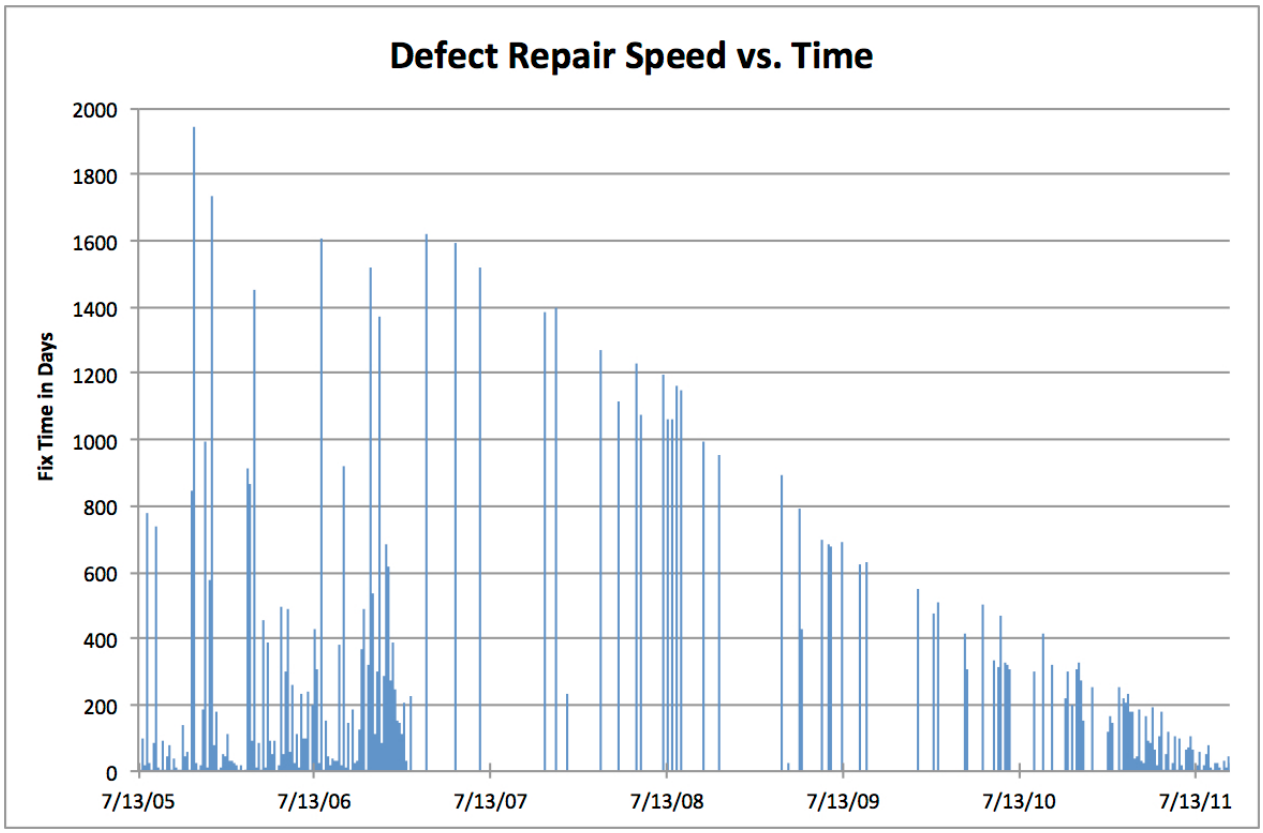


However, this analysis is fairly misleading, as it is not representative of the total time required to fix defects. The following pie chart shows the proportion of total time spent repairing defects of each severity level over the entire lifetime of the Django development process. It clearly demonstrates that 90% of debugging time is devoted towards repairing “Normal” defects, indicating that the vast majority of defects are classified that way. Surprisingly, the next largest section is the time spent repairing “Minor” defects, which suggests that Django developers spend far too much time on defects that are only slightly more challenging than “Trivial” ones. However, the developers of Django do prioritize efficiently; the most severe defects (Release Blocker, Blocker, and Critical) are the least time-consuming repairs of all. Thus, we can infer that the developers are alerted of high-severity defects quickly, and repair them quickly and efficiently in order to keep the development process on track.

Recommendation: devote more time to handling defects classified as “Normal” severity.



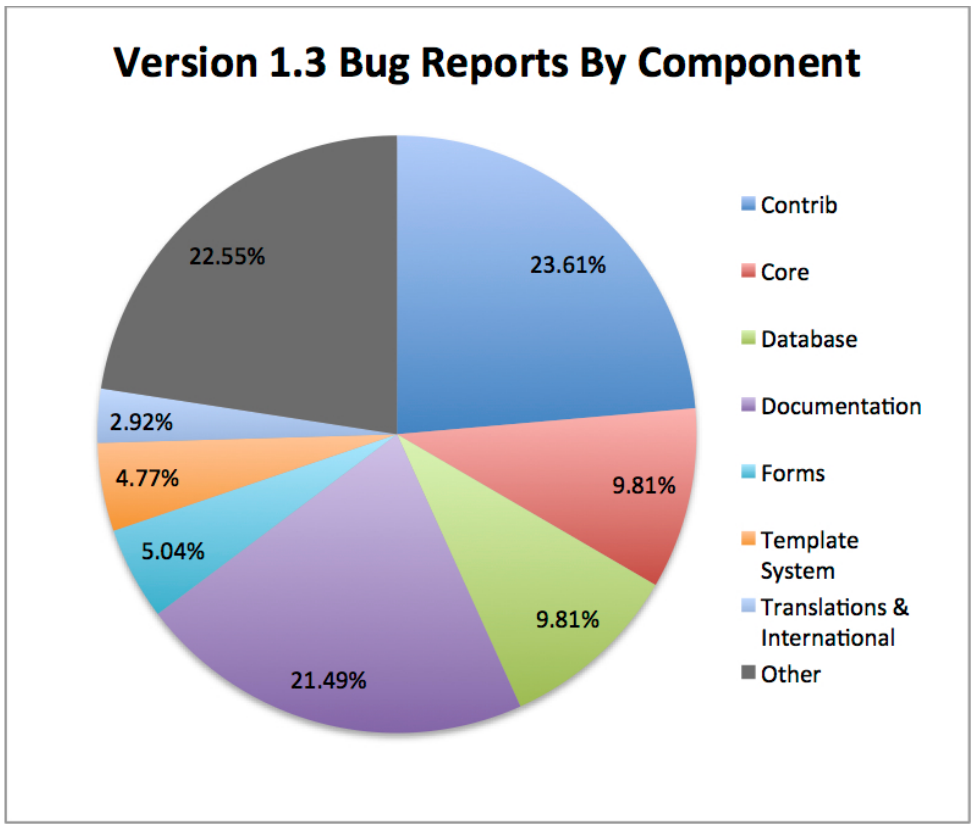
Finally, we evaluate the progression of developer debugging skills over the lifetime of the project. The following graph exhibits the time needed to repair a defect from the initiation of the project through October 2011. The obvious trend displayed here is a decreasing mean time to repair (MTTR) for the typical defect report. The clusters of short repair times indicate an impending version release, which was most likely due to the developers' sudden initiative to tidy up the code base before said version change.



The overall decreasing MTTR trend speaks well of Django developers, indicating that the they are 1) more responsive, 2) increasingly talented debuggers, 3) growing in number, and 4) aware of impending goals and deadlines. For the most part, as the severity of the bugs increased, the average time spent fixing the bug decreased. The only room for improvement lies with the inconsistency of each defect repair – most defects are repaired within a couple months, but occasionally one defect report goes 4-5 months without even being acknowledged. If developers checked and attended to defects more consistently, they could alert fellow Django contributors about the defect report even if they lacked the personal knowledge or free time to handle it.

Recommendation: keep average repair times low by setting a maximum limit on the time a defect can go unacknowledged before repairing or dismissing it.

Another research question we set out to answer was whether Django's defect characteristics changed leading up to the most recent release, version 1.3. Less than 6% of all tickets were submitted before version 1.0. Version 1.3 is responsible for 20% of the total tickets ever submitted to Django, while Versions 1.0, 1.1, and 1.2 are responsible for 30%, 18%, and 25% respectively. This demonstrates how the ticket submission and response system wasn't being fully utilized until after Version 1.0 began. In addition to analysis of defect reports from the previous graph (clustered quick fixes directly preceding version releases), we also generated a graph of all defects specifically reported for version 1.3 only, pictured below. Given that the data presented below is relatively similar to the data for all other versions, we believe that the development effort that went into version 1.3 was not exceptionally remarkable or noteworthy.



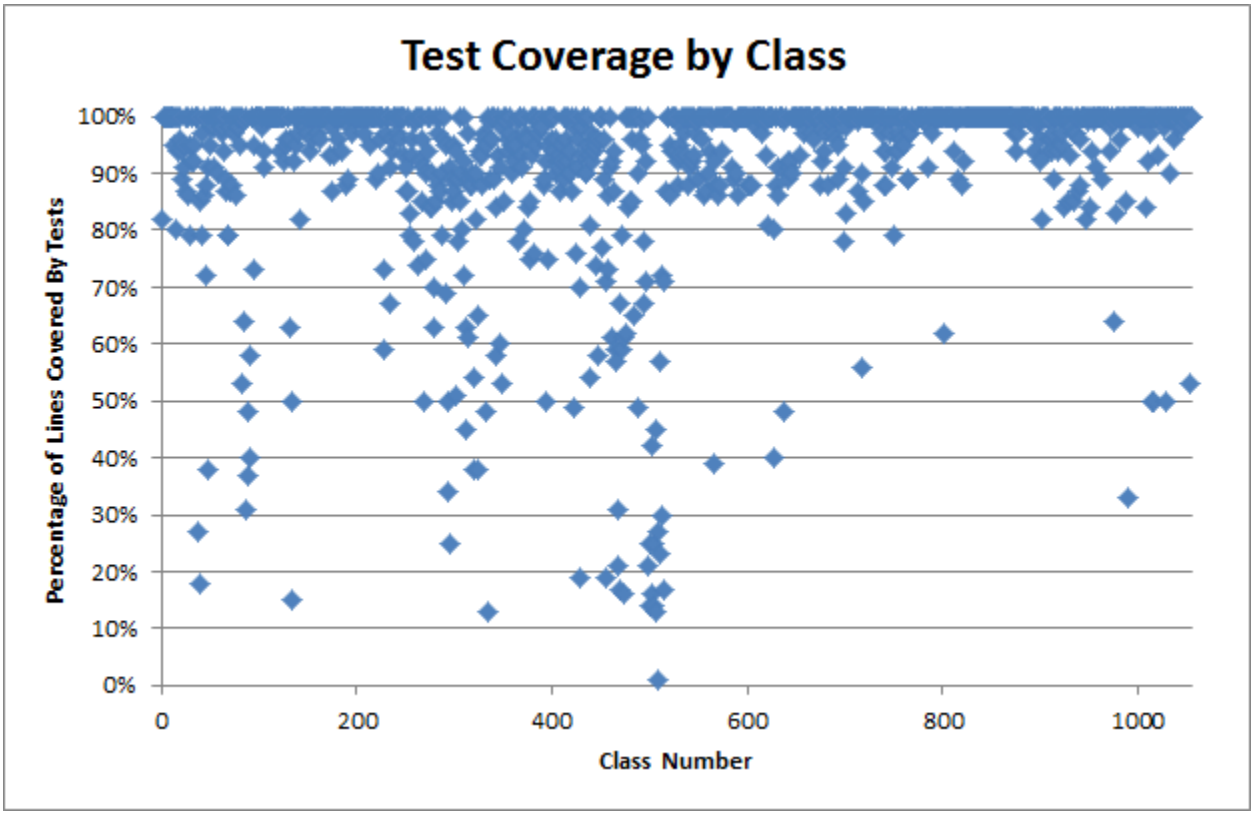
To wrap up our analysis of data mined from Django's ticket tracker, we provide a brief overview of the ticket database as a collective whole. Nearly 13% of all submitted tickets are duplicate tickets, which is a significant waste of developer resources. Luckily, less than half a percentage point of all tickets were submitted without enough information, so most ticket submitters were describing the problem in enough detail that a reader could understand the issue 99.57% of the time. However, a little more than 3% of the tickets could not be reproduced by the reader. Tickets are often submitted with a lack of information. About 62% of bug reports are submitted without a "ticket type," meaning the submitter did not specify whether it was a bug/defect, optimization, new feature, enhancement, etc. Of the tickets that did specify which type category they fell under, over 57% were bugs/defects, 16.5% were new feature requests, 14.6% were enhancements, and 7.5% were cleanups or optimizations.

Recommendation: developers should submit tickets with as much detail as possible, and idle developers should acknowledge said tickets to inspire each other to rapidly handle and close remaining tickets.

3.4 Unit Testing Coverage

To measure test coverage for Django, we first determined the overall test coverage metrics for Django. Coverage.py reported that 89% of the Django code is covered by the tests. The reports also listed what the coverage for each individual class is. The following graph shows the

coverage results by class.



Out of 1056 classes, 112 had less than 80% coverage, 41 less than 50%, and 21 less than 30%. There were 551 classes covered perfectly by the tests cases, so Django has great test coverage. Most of the files coverage.py reported to have less than 50% coverage contain definitions which are not executed, so it makes sense that test coverage is low for these files. While the Django project is well covered as a whole by the tests, some improvements can still be made.

Recommendation: Tests should be added for the classes which have low coverage particularly the classes with lower than 50% coverage.

The following list shows which classes have test coverage of less than 30% excluding the definitions files.

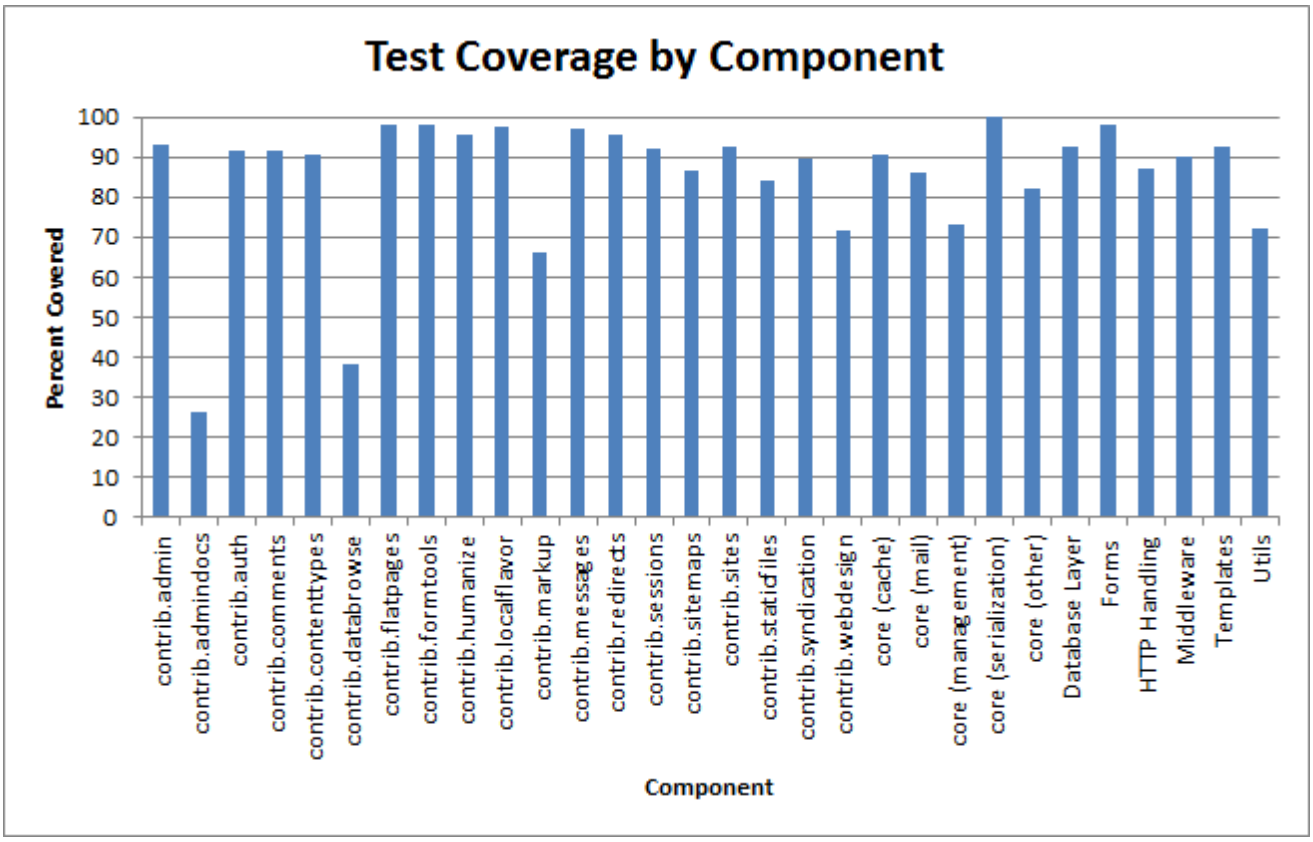
Classes with Coverage <30%	Coverage
/pyshared/pkgresources	1.00%
/core/servers/fastcgi	13.00%
/PIL/JpegImagePlugin	13.00%
/PIL/GifImagePlugin	14.00%
/PIL/ImagePalette	14.00%
/contrib/gis/tests/__init__	15.00%

/utils/unittest/util	16.00%
/PIL/ImageColor	16.00%
/utils/unittest/main	17.00%
/simplejson/scanner	17.00%
/contrib/admindocs/views	18.00%
/utils/autoreload	19.00%
/utils/simplejson/__init__ /utils/unittest/	19.00%
compatibility	21.00%
/PIL/BmpImagePlugin	21.00%
/pyshared/simplejson/decoder	23.00%
/core/files/temp	25.00%
/PIL/Image	25.00%
/PIL/ImageMode	25.00%
/contrib/admindocs/utils	27.00%
/usr/share/pyshared/PIL/PpmImagePlugin	27.00%

The files in the list should be focused on when improving code coverage if possible. Most of the classes with low coverage are related to processing and displaying image files of various types. These files may have low coverage because they may need tested manually to make sure images are visually correct when displayed, but if anything can be done to automate thise tests, it should be done.

Recommendation: Improve test coverage in the classes with less than 30% coverage in the above list.

When the coverage.py results are sorted by component, some interesting results show up. Sorting in this way allowed us to compare the test coverage results to the ticket data to reported in the prior section to see if there were any correlations between test coverage and tickets. The following graph shows the percentage of test coverage per component.



Most components are covered well. Out of 29 components, 23 have over 80% coverage and 17 have over 90%. Six components are covered less than 80%. These components are contrib.admindocs, contrib.databrowse, contrib.markup, contrib.webdesign, core (management), and utils. Contrib.admindocs and contrib.databrowse are by far the worst covered components with 26.35% and 38.46% percent respectively.

For the most part, the components with high coverage have a fewer number of tickets opened against them than the components with low coverage. Contrib.admindocs is the worst covered component and it is the component with the most defects opened against it. Improving test coverage may help find defects earlier and reduce the number of tickets opened against the contrib.admindocs component and the entire Django project.

Recommendation: Improve test coverage in the components with low test coverage particularly contrib.admindocs.

3.5 Coding Standards and Style Guidelines

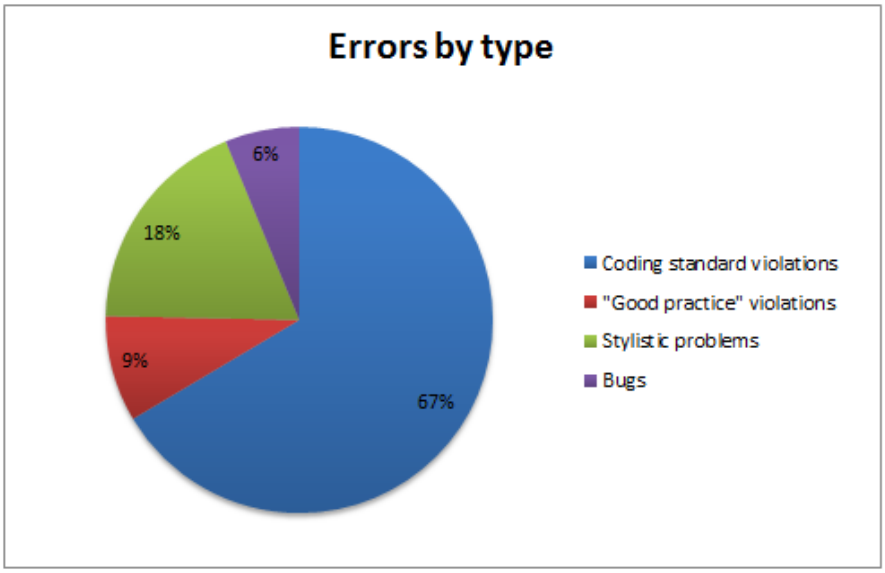
Analyzing the Django codebase using PyLint allowed us to gather interesting statistics about the whole project as a whole. One of the most useful uses of PyLint is the detailed analysis function that checks for errors in every line of code and displays messages describing these errors if present. A total of 72 unique messages were obtained from running PyLint on the codebase, the five most frequent messages are listed in the table below:

Message	Count	Description
C0103	3972	Used when the name doesn't match the regular expression associated to its type (constant, variable, class…). [PyLint uses Python standard naming convention for this analysis]
C0301	3775	Used when a line is longer than a given number of characters. The default limit is 80, the customary width of a terminal window.
C0111	3181	Used when a module, function, class or method has no docstring.
W0212	1009	Used when a protected member (i.e. class member with a name beginning with an underscore) is access outside the class or a descendant of the class where it's defined.
E1101	746	Used when a variable is accessed for a nonexistent member.

The produced messages fall under one of four error categories:

1. Coding standard violations
2. “Good practice” violations
3. Stylistic problems
4. Bugs

The majority of the the errors resulted from coding standard violations making over 60% of the total number of errors. That type of errors is expected in open source projects, especially those with large developer bases. That is mainly because different developers have different programming backgrounds and adopt different programming styles and conventions. These conventions do not necessarily match the Python coding standards that PyLint checks for. Following a set of standards help improve the understandability of the code. Understandable code is easier to debugged, reused, and modified by other developers. The chart below shows the distribution of errors among the four error categories listed above

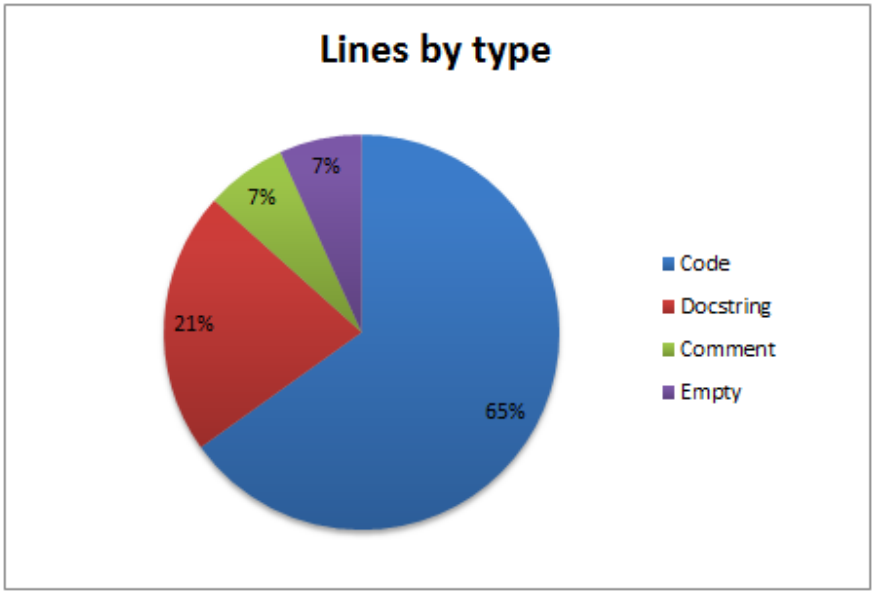


Recommendation: enforce coding standards for a more understandable code.

The other aspect PyLint allows us look at is the distribution of lines by type, each line is classified as one of the following types:

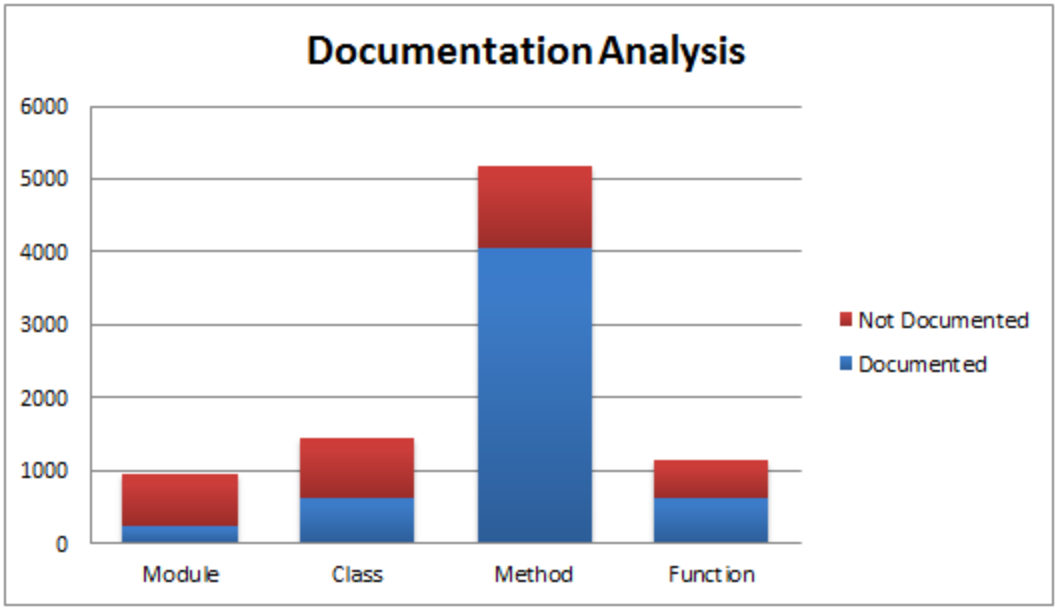
- Code
- Docstring
- Comment
- Empty Line

The chart below shows the percentages of each line in the Django project codebase:



It is important to point out that about one fourth (28%) of the total number of lines are dedicated to docstrings and comments. Only 7% were comment lines, a significantly small percentage of the codebase. The other 21% of the lines are used to document the code elements.

However, further analysis of the documentation of the codebase indicate that not all code elements are documented equally, the following chart shows that while almost 80% of the methods are documented, the percentages are much lower for the modules, classes, and functions (26%, 43%, and 55% respectively). But since the number of methods is very large (there are more methods than all the modules, classes, and functions combined), looking at documentation of the codebase as a whole can be misleading.



Recommendations: add more comments and document more modules and classes to improve understandability and maintainability.

4.0 CONCLUSION

This case study of the Django project, an open-source web development framework composed in Python, has presented a wide variety of statistically-significant data. To measure code size, rate of growth, complexity, and maintainability, we used Django's source control repository to analyze historical revisions since 2005. To analyze defect tracking, we looked at ticket records dating to the beginning of the Django project. For unit testing coverage, coding standards, and style guidelines analysis, we analyzed source code that was extracted from the Django subversion server on October 31, 2011.

We found that the size of the project has greatly increased since its start, but that the rate of increase has decreased over time. The largest component with approximately half of the code is the contrib component which could be broken up into smaller, more manageable components.

The overall cyclomatic complexity of Django is 2.49 which is respectable. Complexity started out nice and increased for the first few years, but has decreased since the initial increase. Some modules, though, have high complexities which should be reduced if possible.

The ticket data showed that the majority of the tickets opened were for defects. When looking at only the defect tickets, we found 4 components out of 48 had almost 60% of the defects opened against them. The core and database components have the most defects per SLOC, so improving code quality in these areas will greatly reduce the overall number of defects in the Django project.

As a whole, we found that the Django code is well covered by its test cases. The component with the lowest coverage, contrib.admin, was also found to be one of the components with the highest number of defects opened against it. Improving test coverage may reduce the number of tickets opened.

We found that python coding standards and style conventions are poorly followed by Django developers. Additionally, Django code is poorly commented, only 7% of the total lines of the code base were comment lines. We also found that the amount of documentation is not consistent throughout the code elements, while most of the methods are documented, less than half the modules and classes have documentations.