February 2013

Code Churn Dashboard

Joseph Daniel Rivera
Worcester Polytechnic Institute

Remy Jette
Worcester Polytechnic Institute

William Edward Caulfield
Worcester Polytechnic Institute

Project Number: MQP DXF MS12

CODE CHURN DASHBOARD

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the Bachelor of Science degree

by

_____	_____	_____
William Caulfield	Remy Jette	Joe Rivera

Date: December 14, 2012

Sponsor: Microsoft

Liaisons: Sandy Gotlib, Vinay Kumar

_____

Professor David Finkel, Advisor

WPI

Microsoft

Abstract

This project was conducted with Microsoft in Cambridge, Massachusetts. The purpose was to design and implement a tool that would allow users to easily identify which parts of a software project had high levels of code churn. The tool we developed interfaces with Microsoft's internal source control systems, calculates churn measures for each file, and makes the churn data available through a web interface. By analyzing this churn data, software developers can identify areas of code likely to contain defects.

Acknowledgements

We'd like to thank Microsoft for providing us with the opportunity and resources to complete this project. Special thanks go to Vinay Kumar and Sandy Gotlib for their valuable guidance and for making our project possible. We'd also like to thank Ben Fersenheim for providing us with help and assistance with some of Microsoft's internal tools. Finally, we'd like to thank Professor David Finkel for his assistance throughout the course of this project.

Table of Contents

Abstract ii

Acknowledgements iii

Table of Contents iv

1 Introduction 1

2 Background 2

2.1 Code Churn 2

2.2 Microsoft Internal Tools 3

2.2.1 Source Depot 3

2.2.2 Product Studio 5

2.2.3 BuildTracker 6

2.2.4 CodeFlow 7

2.2.5 Srctool API 8

3 Design 10

3.1 Database 10

3.2 Populate Job 12

3.3 Dashboard 14

3.3.1 Settings View 14

3.3.2 Analyze View 16

4 Data Acquisition 20

4.1 Desired Data 20

4.2 Collecting Branch Data 20

4.3 Collecting Source Code Churn Data 22

4.4 Collecting Binary Churn Data 23

5 Analytics 25

6 Future Work 27

6.1 Code Delta 27

6.2 Product Studio Integration and Risk Analysis 28

6.3 Additional Code Churn Metrics 28

6.4 Additional Dashboard Views 29

6.5 Dynamic Drop Folder 29
6.6 User-Friendly Directory/Files Selection 30
6.7 Integrate Test Data 31
6.8 More Filters 31
6.9 Seamless Transition between File Types 32
6.10 Applying Churn Coefficients to Specific Files and Folders 32
7 Conclusion 34
8 Works Cited 35

1 Introduction

The software development process is an iterative one, with each iteration reaching a milestone (also called sprints in the Agile methodology) before moving on to the next. Each milestone consists of the following series of repeated steps: designing the specification for the new feature, writing code to implement the feature, building the binary, testing and identifying bugs, and fixing bugs. These steps are repeated until a known quality bar is reached and the code is ready to be shipped to customers. The implementation and bug-fixing stages both involve modifying code, introducing code churn. Code churn measures the changes made to a codebase over a period of time and is a way to quantify the extent of these changes.

Code churn is important to a software development team as it can be used to predict defect density. A study by Nagappan et al. showed that certain code churn metrics correlate highly with flaws in the code. These metrics have a role in driving the development and test process, as highly churned code is more prone to defects and should be tested more carefully and more thoroughly.

Our ultimate goal is to answer the following questions:

- What is the best way to measure code churn?
- Which raw data measures and metrics should be used to express the degree to which a file is churned?
- How can we present this data in a useful fashion?

By answering these questions, we hope to analyze the available data for a project and provide a useful report to project managers that they can use to direct their teams.

2 Background

2.1 Code Churn

In order to find code churn metrics that we could trust, we looked to the literature for any research that had demonstrated metrics with a significant correlation to fault rate. This is very important, as studies have shown that using lines of code churned or simple size metrics alone is a poor way to measure defect density (Fenton, N. E., Ohlsson, N., 805). Fortunately, we found a technique that drastically improved the predictive power of these code churn metrics. The problem with many churn metrics is that and increase in that metric, or *absolute* measure, can be caused by one of many things. By computing the ratios of certain pairs of metrics, one can obtain much stronger indicators of defect density (Nagappan and Ball). These derived ratios of absolute measures are referred to as *relative* measures. There are also further indicators of defect density such as code complexity and organizational structure (Nagappan, Basilli, and Murphy), but obtaining this information temporally would prove too difficult and appears out of the scope of this project. We chose to use the relative measures described by Nagappan and Ball, which also determined which absolute measures we had to collect (see Figure 2-1).

Once a file's relative churn measures are calculated, they can be added together with different weights to predict the number of defects the file contains. Nagappan and Ball determined which measures have the strongest correlation to defect density, and assigned a set of coefficients to each measure that provided the best predictive model. However, their model was made specifically for a single software product (Windows Server 2003), and so the general applicability of their coefficients is unknown. It is possible that a different set of coefficients would be the best predictor for each different project one wishes to analyze. The researchers did not conjecture how much each coefficient might

differ, so it would probably be wise to re-run their analysis on whatever codebase one wishes to find defects in for best results.

Absolute Measure	Description
Total LOC	The total number of lines of code in the file
Churned LOC	The number of lines added or modified since a previous revision
Deleted LOC	The number of lines deleted since a previous revision
File Count	The total number of source files that contribute to a binary (for a source file this is always 1)
Weeks of Churn	The amount of time a file is open for editing
Churn Count	The number of intermediate revisions between two versions of a file
Files Churned	The number of contributing files that have been churned (for a source file this is always 1)

Relative Measure
Churned LOC / Total LOC
Deleted LOC / Total LOC
Files Churned / File Count
Churn Count / Files Churned
Weeks of Churn / File Count
Lines Worked On / Weeks of Churn
Churned LOC / Deleted LOC
Lines Worked On / Churn Count

Figure 2-1. The absolute and relative measures used by the Code Churn Dashboard
Note: "Lines Worked On" is the sum of Churned LOC and Deleted LOC

2.2 Microsoft Internal Tools

2.2.1 Source Depot

In order to facilitate collaboration on shared code among team members, many teams use a tool called a "version control" or "source control" system. Many popular source control systems exist, such as Subversion (also known as SVN), Git, and Microsoft Team Foundation Server. Although each system

has its differences in features and limitations, they all have the common functionality of storing current revisions of files, historical revisions, and some metadata about each revision.

Internally, many teams within Microsoft use a source control system called Source Depot. The Source Depot system is only used within Microsoft, and is not sold or licensed for use by other companies or individuals. As a result, all documentation about the Source Depot system, related tools used to access it, and the SD API are all stored on internal Microsoft sites.

Source Depot stores all data about files in the form of *changelists*. Each changelist has a numerical identifier, and although they are not sequential they are well-ordered chronologically. This means that a submitted changelist will always have a higher changelist number than changelists submitted earlier, but there may be gaps in the numbering. This is because before a changelist is submitted and synchronized with the rest of the source code, it is sent to Source Depot by the developer as a 'pending' changelist. This allows other developers to be able to refer to code that hasn't been submitted yet for code reviews and other tasks. When the status of a changelist is changed from 'pending' to 'submitted,' it may be given a new changelist number to ensure that its identifier is higher than those of the changelists that were submitted while it was in the pending state.

In addition to its numerical identifier, each changelist contains a description, the username of the developer who submitted it, a list of the files affected, a tag indicating what type of change it was ('add', 'edit', 'delete', 'branch', and 'integrate'), and a diff between the old and new versions for each file. The affected files are listed in the form of a network path on the Source Depot server, such as `//depot/dev/branch/file.txt`. With this collection of changelists, the entire history of a product's development can be stored.

In order for a development team to work on a large change without affecting other teams, Source Depot also implements the concept of branches. Branches are made using the 'branch' operation, which copies files from one location to another. An entire source code directory tree can be

cloned to a separate branch, and when development is finished all of the changes can then be merged back into the main tree at the same time using the 'integrate' operation. These branches simply take the form of another directory on the Source Depot server such as `//depot/se/softgrid/4.6/SE_46_Mainline/` for the branch called SE_46_Mainline.

Source Depot also provides an API, implemented through the Microsoft Component Object Model (COM), which allows a developer to access the data stored by Source Depot for use in other programs. This API is simply a wrapper for the Source Depot commands normally available to users via the sd.exe program, and for programmatic access to all of the data available in Source Depot such as changelist descriptions and affected files, the differences between revisions of files, etc. In standard mode, data is returned from Source Depot as either strings or arrays of ASCII bytes, depending on the information requested. The data is not divided up in any way based on its content and has to be parsed. The Source Depot API also has a structured mode. While the data is still returned as strings, the data is at least somewhat organized. For example, when requesting all of the changelists from a given branch, each changelist is given as just "`234530`" as opposed to "`Change 234530 on 2012/10/25 10:05:06 by <user> '<description>'`", making it easier to parse. As a result, we used the structured mode with all commands for which it was supported. This API allowed us to analyze Microsoft source code and identify source files with high levels of churn.

2.2.2 Product Studio

Another crucial part of maintaining large software projects is bug tracking. Bug tracking systems typically distinguish true bugs from orders for new work, allow for work to be assigned to certain people, and keep track of the current stage in the repair process for any given bug. Microsoft uses a proprietary bug tracking system called Product Studio, which does all of the above and more.

Each bug in Product Studio (PS) is assigned a severity and a priority, which respectively refer to how costly the bug is and how important it is to fix. Bugs can also be given a short description; assigned

a project area, which indicates which part of the codebase is affected; an assignee, who is charged with fixing the bug; and a reviewer, who inspects the proposed fix for quality. Bugs have other fields, too, but they aren't often set by less diligent developers. One such field with significance to us is the regression flag, which indicates whether this is a bug that was thought to be fixed but was broken again. Also valuable to us is the field that holds the changelist numbers of all changes made in the process of fixing the bug. When both the regression and changelist fields are set properly, they could hypothetically be used to identify segments of code that are especially risky to change.

2.2.3 BuildTracker

Microsoft tracks all of its product and branch information using a system called BuildTracker. When an employee wishes to enlist the code from a specific branch to his computer, he visits the BuildTracker website and chooses from a list of branches that the administrators have granted him access to. These branches are all associated with specific products, such as Office, AppV, or Application Virtualization. For example, our Code Churn Dashboard project was made part of the `se_46_mainline` branch, which is part of the Application Virtualization product. To enlist this branch to our computers, we had to select "Application Virtualization" from the list of products, and then select `se_46_mainline` from the list of branches.

Additionally, BuildTracker enables the management of build jobs. Build jobs instruct dedicated machines to synchronize to the latest version of the source code and compile it. Build jobs can be queued up and automatically run when the system is free to do so. After a build completes, all of the users who were specified during the creation of the build request are notified via email about the results of the build.

In addition to official builds which compile the code and produce the resulting binary files, a user can also perform test builds called "buddy builds". After a buddy build completes, the current build number of the branch does not change. The two purposes of running a buddy build are to make sure

that there are no pre-existing errors in the codebase and to make sure that no new errors would arise as a result of the introduction of the code to be committed. We frequently had to run buddy builds prior to committing code in order to ensure that our committed code would not interfere with the existing code (since our Code Churn Dashboard project was being stored in one of the Application Virtualization development branches).

There is an API for BuildTracker which includes many useful features. Among the most useful features to us were the ability to query BuildTracker for existing products and the related branches, and also the ability to retrieve the Source Depot source code locations for each of those branches.

2.2.4 CodeFlow

For the code review process, Microsoft uses an internal tool called CodeFlow. Before a developer commits any changes to Source Depot, those changes must first be reviewed by others. This is where CodeFlow comes into play. CodeFlow takes the changes that are contained within a pending changelist and visualizes them so that they can be reviewed before a final commit is made. This way, fellow developers and testers have a chance to suggest changes to the structure of the proposed code, as well as to catch any errors or unintended results that the original developer may have overlooked. As the developer makes the suggested changes to their code, they can introduce new *iterations* of the proposed changelist which address the suggestions made. CodeFlow keeps track of all these iterations and allows the users of CodeFlow to look at each successive iteration, thereby capturing the transformation of the developer's original code proposal as it has changed due to the suggestions of the reviewers.

Once a reviewer approves of everything in the most recent iteration of the changelist, the reviewer then "signs off" on the content of the changelist. Every team is different, but most teams require that a certain number of reviewers have signed off on the code review before the final changes can be submitted to Source Depot. Nothing actually restricts a developer from submitting code to

Source Depot immediately, but it is considered bad practice to do so before it has been reviewed in CodeFlow.

In order to use CodeFlow, a developer must first run "`mkreview <CL#>`," where CL# is the changelist number of the proposed changes. This creates an *sdpack* on a file share, which tracks all of the changes for the developer's changelist. Next, the developer must run "`cfreview <CL#>`." This takes the sdpack that was created in the previous step and uses it to generate a session in CodeFlow. At this point, CodeFlow can display a visual of all of the files that have been modified, added, or deleted in comparison to the current version of the codebase that currently resides in the Source Depot repository. After running `cfreview`, the developer can specify the list of reviewers who will be required to review the changes. After doing so, all the intended reviewers will receive emails that a new review has been submitted. In the email there will be a link to "Open in CodeFlow." Clicking on the link allows a reviewer to open the corresponding CodeFlow session and begin reviewing.

The use of CodeFlow was invaluable to us in the completion of our MQP. Our code review process was such that the reviewers for a CodeFlow session were the two remaining members of our three person team, in addition to our Microsoft mentor. Before committing code to Source Depot, we had to make sure that two out of the three reviewers had signed-off on the changelist. This ensured that our code was well-critiqued and as efficient as possible.

2.2.5 Srctool API

Srctool is a utility available within Microsoft that parses a binary's program debugging symbols (pdb's) to find out which source files were needed to build that binary. Srctool is accessible from the command line using Srctool.exe, and a subset of its functionality is exposed in the Srctool API.

When running srctool.exe, there are a number of command-line arguments that one can specify:

- `-u`: displays only source files that are not indexed
- `-r`: dumps raw source data from the pdb

- `-l <mask>`: limits to only source files that match the regular expression `<mask>`
- `-x`: extracts the files instead of simply listing them
- `-f`: extracts the files to a flat directory
- `-n`: shows version control commands and output while extracting
- `-d <dir>`: specifies which directory to extract to
- `-c`: displays only the count of indexed files – the names are not given

Srctool API, on the other hand, exposes a single method that is used to list raw source data. Its output is equivalent to running srctool.exe with the '`-r`' flag. The advantage to using Srctool API is that it doesn't require the starting of a new process to execute srctool.exe. The Srctool API is thus the means we chose to obtain information about which source files contribute to a given binary.

3 Design

The Code Churn Dashboard is divided into three functional parts. The first is the database, which stores all of the data mined from Source Depot. The second is the task which does the data mining through SDAPI and stores its results in the database. Lastly is the user-facing segment, which queries the database in order to present visualizations of the churn data in a customizable fashion to the user.

3.1 Database

One of the first decisions we made when designing the Code Churn Dashboard was to use a database to store raw data that had been mined from Source Depot. We chose to do so because we didn't want to flood the Source Depot server with connections whenever the dashboard was used, as this would likely be during working hours when other users would be trying to sync with or submit code to the Source Depot branches. By using a database, we could limit interaction with the Source Depot server to non-peak hours. In addition, we wanted to do calculations as early as possible (while the database was being populated) so that visualizing and presenting the data on the dashboard would require less computation and would therefore be more responsive. This became significant after working with the Source Depot API (see Section 2.2.1), as it prevented the long wait times involved with getting results from a Source Depot command from impacting the user experience.

For the database, we tried to split up our data into separate tables as much as possible and link those tables using foreign key constraints. This allowed us to avoid the wasted space that is caused by the same row being duplicated many times with only a small number of the columns changed. For example, with each change of a file, the churn data changes but the file path and associated branch do not. With the abstraction provided by LINQ to SQL, these foreign key links between tables made accessing necessary data much easier because data in linked tables was always available as a member in the LINQ object.

Figure 3-1 shows the tables in our database and the foreign keys that link them together. For example, our planning led us to have tables called `AllProducts` and `AllBranches`, with the `AllBranches ProductID` column being a foreign key to the `AllProducts ID` and linking the two together. These hold all of the product and branch data pulled from BuildTracker so it would be available for the user interface. When a branch is enlisted, a row is added to the `EnlistedBranches` table using data given at enlistment combined with the associated data in `AllBranches`. When the Populate Job described in Section 3.2 is run, the `SrcPath` and `BinPath` tables are populated to hold the paths of all source files and binaries, and the `Src` and `Bin` tables are then populated with the churn metrics of those files for each interval of time. Finally, the `MapChanges` table contains data on how binaries can be mapped to their component source files, which can be used to determine binary churn data (see Section 4.4).

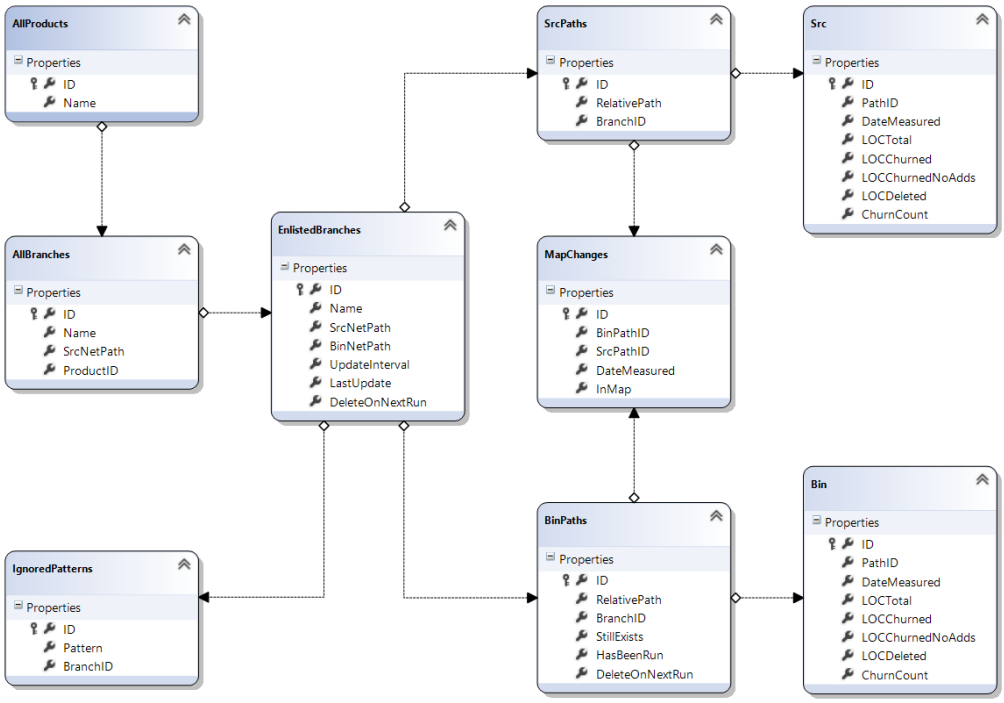


Figure 3-1. The organizational structure of the Code Churn Dashboard Database

3.2 Populate Job

Once the database design was in place, we needed a plan for collecting data and populating the database with it. We decided that the best method was to write a program that would run as a scheduled task to fetch data and store it into the database. That way, the data would be immediately available when requested, and the job could also be scheduled when the data sources weren't being used to avoid exhausting those servers' resources. The data we use draws from three different data sources, so he most logical approach we considered was to split the tables up based on which source they obtain their data from. The three sources are Build Tracker API, Source Depot API, and Srctool API, which were each discussed in Section 2.2.

The first source of data, the Build Tracker API, provides product and branch information. This is required to create the Settings View discussed in Section 3.3.1, and provides necessary information to enlist branches. As a result, it was separated to its own DLL library so that it could also be run on demand in the settings view if someone wishes to enlist a branch that is not yet in the database, in addition to being run with the rest of the Populate Job. This DLL has a single class, `PopulateProductBranchData`, which performs all of these tasks.

In order to populate the source churn data, there are two logical steps; the first is to request the necessary data about each file from Source Depot, and the second is to insert that data into the database. As a result, we decided to divide this section of the Populate Job into two classes based on this logical division. The first class, called `SrcChurnDataPopulator`, provided the overall flow and performed the insertions into the database. Before it starts, it pulls data from the `EnlistedBranches` and `IgnoredPatterns` tables to know which branches to collect data from and which source files should be excluded. The second, `SourceDepotExtractor`, handles all of the communication with Source Depot. To make our program more extensible, `SrcChurnDataPopulator` requests information from an `ISrcMetricExtractor` interface,

which the `SourceDepotExtractor` implements. As a result, if a development team wished to use the Code Churn Dashboard with another version control system, they would merely have to implement the methods from that interface.

Populating and updating the binary churn data requires several things to be done. We must aggregate any data that was just collected by the `SrcChurnDataPopulator` where the source files contribute to the binaries the dashboard is tracking. However, there is also the possibility that a binary has just been added to the dashboard, in which case we need to review the source data from the entire history of the branch to calculate that binary's churn retroactively. For this reason, we added a bit flag to the `BinPath` table called `HasBeenRun` that indicates whether a binary's churn has been calculated up to the current churn interval. There is also a flag called `DeleteOnNextRun` that indicates whether a binary is to be removed from the tracking next time the job runs. By checking these flags, the job is able to determine how to treat each of the tracked binary files. The next step of our design is to determine in what order we handle each class of binaries.

Choosing which category to process first seemed like a natural choice. It would be a waste of resources to collect data for a binary and then immediately delete it, so we perform all deletions first. The next choice we made regarding deletions was to use the `DeleteOnNextRun` flag in the first place. It would be possible for us to enable a dashboard user to delete a `BinPath` row directly from the dashboard and SQL Server would automatically delete the `Bin` rows associated with it, but replacing the data if the user changed their mind would be much more difficult. By controlling deletion through a flag in the database, we offer the rest of the day as a grace period in which users can change their mind without creating avoidable work for the dashboard's back end.

After purging the data that we no longer want, we can split the remaining tracked binaries into two groups – those that the job has already been tracking, and those which were added through the dashboard web UI. We choose to bring the already tracked binaries up to date before we begin enrolling

the new ones. This way, when we enroll the new ones, we only have to look up the list of sources that contributed to them once. We also get to set the `HasBeenRun` flag for each new binary only once that binary has been brought completely up to date. If we reversed the order, we would have to either only run the new binaries up to the same state as the old binaries (missing the most recent week), and then collect the newest week for all binaries at the same time, or run the new binaries all the way through, but hold off on setting the flag in order to distinguish them from the old binaries before we update the old binaries' churn data. In the first case, we look up the binary-source mapping for each of the new binaries twice when we could only do it once. In the second case, we increase the risk of leaving the database in an inconsistent state if the populate job is ever interrupted because we wait a longer-than-necessary time to set the flag after the new binaries have been partially run. Weighing these facts, we believe our ordering of the three steps to be the best one.

3.3 Dashboard

After the data was collected and the database was populated, the next step was to present it in a useful fashion so that actions could be taken on the data. To this end, we developed a dashboard frontend using HTML5 and Javascript that would both allow administrators to set up the dashboard and enlist branches as well as allow users to see aggregate churn data for a given branch of a period of time. The dashboard was developed using ASP.NET, C#, HTML5, and Javascript.

3.3.1 Settings View

In order to facilitate enrollment of new branches and simple modification of existing enlistments, we developed an administrative view of the dashboard that would allow these changes to be made (see Figure 3-2). When this view is loaded, a list of products is shown to the user. When clicked on, each product expands to show all of its available branches. If no branches are in the database, a link is provided which will force the collection of the most current data from Build Tracker (see Section 4.2).

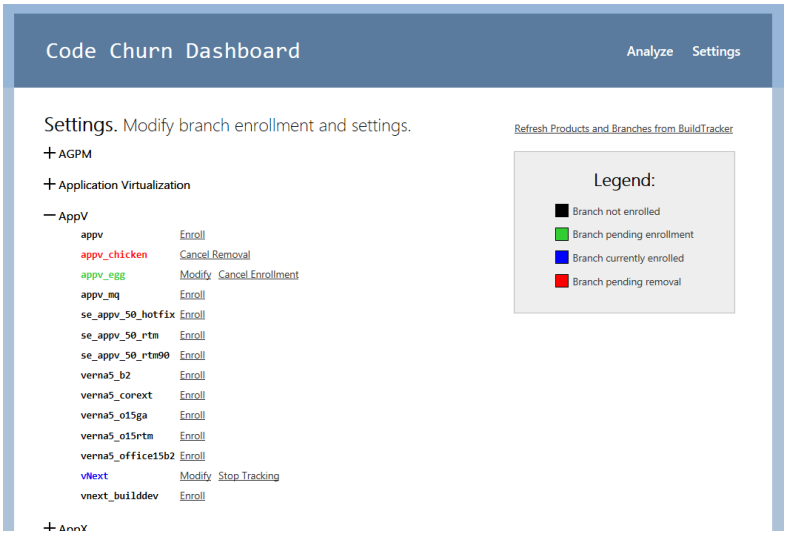


Figure 3-2. The Code Churn Dashboard Settings View

When branches are shown, they are color-coded to show their current state with respect to the Code Churn Dashboard; black indicates that a branch is not enlisted, green means that a branch has been enrolled but no data is available yet, blue indicates that a branch is currently being tracked, and red informs the user that the branch is pending removal. This pending state was added because populating a newly-enrolled branch takes a long time, so this allows one to undo an accidental removal.

Next to each branch, various options are given depending on the branch's current state. Branches that are currently pending removal or enrollment have the option to return them to enlisted or not enrolled, respectively. Branches that are not enlisted have the option to enroll them, and finally branches that are enrolled have options to either modify or remove them.

When a branch is selected to either be enrolled or modified, a modal dialog pops up providing additional options to the user (see Figure 3-3) – some options are given default values, or in the case of branch modification the current settings are shown. The 'update interval' determines the granularity of data for a given branch. The binary net path is the network location where binaries for a given branch are stored, which when provided allows the dashboard to calculate aggregate churn for those binaries

based on the churn data for their component source files. The included binaries list indicates which binaries should be tracked, and finally the 'ignore table' allows administrators to instruct the population job to completely ignore files matching a given pattern.

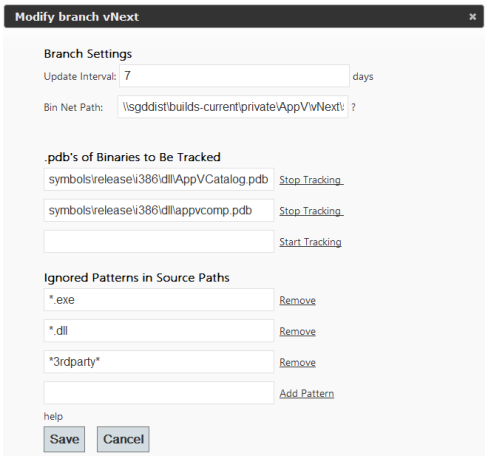


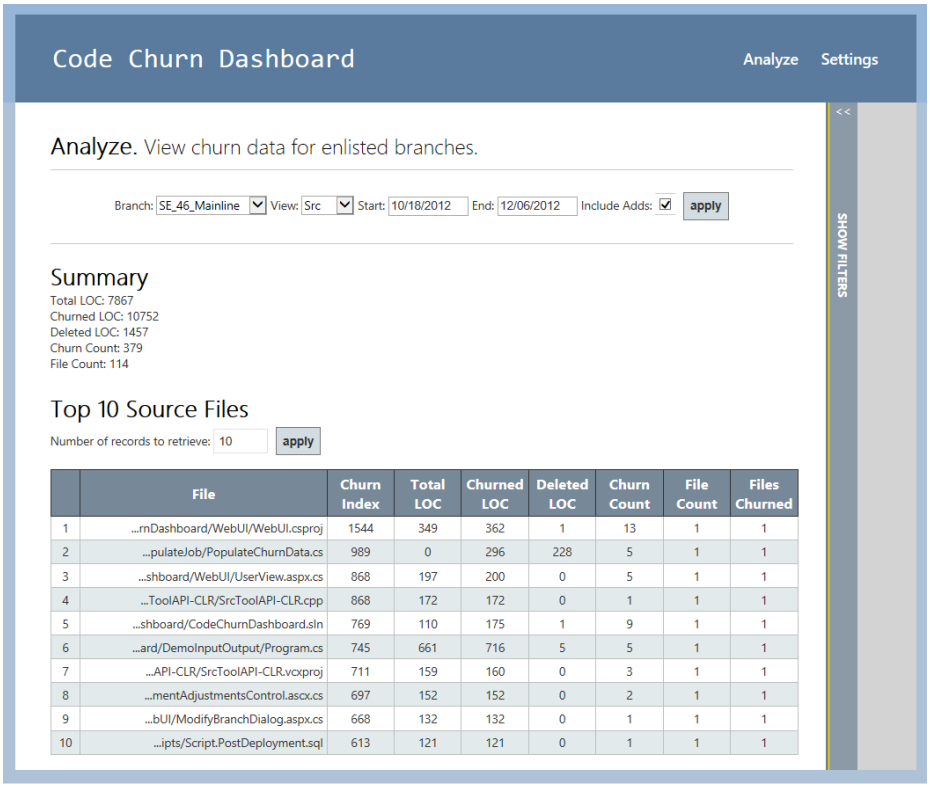
Figure 3-3. The Modify Branch modal dialog

When the settings are saved, the changes are stored in the database and will take effect the next time population job is run, allowing unwanted changes to be easily reversed. When the status of a branch changes, the page is refreshed to show each branch's current state.

3.3.2 Analyze View

The primary function of the Code Churn Dashboard is to view aggregate churn data for a selected branch. This data is accessible through the "Analyze" page of the dashboard. Along the top of the Analyze page, there is a panel where the user can specify the options for the desired churn data (see Figure 3-4). The user selects which one of the enlisted branches they wish to analyze, as well as other settings, like start date and end date. This top panel also gives the user the choice between viewing analytics for source files or binary files. The last option in the top panel is a checkbox which allows the user to specify whether they would like to include files that were newly added to Source Depot in the analysis. This can be useful because oftentimes added files will skew the results due to the fact that they

result in a very high churn count, despite the fact that there wasn't any pre-existing code that had changed.



	File	Churn Index	Total LOC	Churned LOC	Deleted LOC	Churn Count	File Count	Files Churned
1	...rnDashboard/WebUI/WebUI.csproj	1544	349	362	1	13	1	1
2	...pulateJob/PopulateChurnData.cs	989	0	296	228	5	1	1
3	...shboard/WebUI/UserView.aspx.cs	868	197	200	0	5	1	1
4	...ToolAPI-CLR/SrcToolAPI-CLR.cpp	868	172	172	0	1	1	1
5	...shboard/CodeChurnDashboard.sln	769	110	175	1	9	1	1
6	...ard/DemoInputOutput/Program.cs	745	661	716	5	5	1	1
7	...API-CLR/SrcToolAPI-CLR.vcxproj	711	159	160	0	3	1	1
8	...mentAdjustmentsControl.ascx.cs	697	152	152	0	2	1	1
9	...bUI/ModifyBranchDialog.aspx.cs	668	132	132	0	1	1	1
10	...ipts/Script.PostDeployment.sql	613	121	121	0	1	1	1

Figure 3-4. The Code Churn Dashboard Analyze View

On the right-hand side of the Analyze page there is a collapsible side panel that contains even more settings that the user can modify in order to further dictate how the data is processed (see Figure 3-5). The first section of the collapsible side panel allows the user to create filters that limit the data that is analyzed. An example of a filter that can be created here is one that excludes certain date ranges within the overall date range that was selected in the top panel of the Analyze page, as discussed in the preceding paragraph. Another example filter is a "path contains" filter. In this filter, the user specifies part of the relative path within the branch's binary or source code folder, and as a result, any file whose relative path does not contain the specified string will be excluded from the analytics.

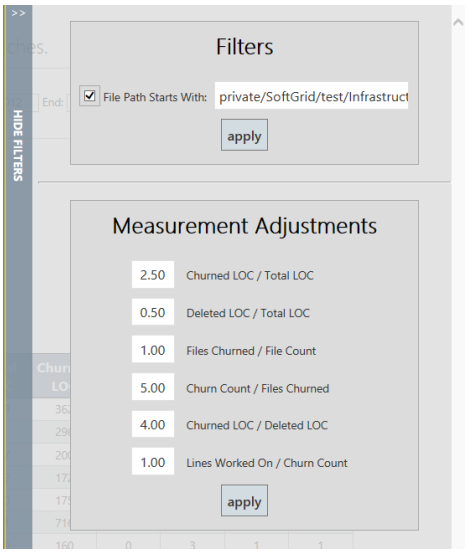


Figure 3-5. The Analyze View filters and measurement adjustments side panel

Below the filters is the Measurement Adjustments section. Here, the user can modify the default coefficients that are used when calculating the churn indices for the files within the search criteria. Churn index (as discussed in Section 5), is the sum of the six obtained relative measures, each multiplied by a coefficient, since the fine-tunings of the importance of each relative measure can vary from project to project. The default coefficients for the churn index calculations are all 1.0, but the user can change any of them to be non-negative floats in the Measurement Adjustments section.

Once all of the aforementioned settings have been specified, the user can then view the analytics results of their query in the main content area of the page. At the top of the main content area, there will be a Summary section where the churned measures for each of the selected files will be summed and displayed as totals for each of the absolute churn measures. Below this, there will be a "Top N" view, where the user can specify how many files they wish to see the absolute churn measures for. By default, N is 10, but can be changed to any positive integer. The "N" files with the highest churn indices will be displayed, including the paths to the files, their absolute churn measures, and their calculated churn indices. Due to the fact that the paths to a given file may be very long, we truncate it in

order to fit in the table cell that it is displayed in. The full path can be viewed by hovering over the shortened path to the file, at which point a tooltip will show up with the full path displayed in it.

When viewing the Top N source files, there will be an expandable dropdown beneath each file path which, when expanded, will show all of the binaries that the given source file contributes to. We made the decision not to implement this same functionality when viewing binary files due to the fact that the number of source files that make up a given binary will likely be very high and it wouldn't be feasible to have an expandable drop down with possibly hundreds of source files. Visually, this wouldn't look very appealing, and practically, this would take too long for the page to load.

4 Data Acquisition

4.1 Desired Data

In deciding what data we needed to collect, we began with the end in mind. Our eventual goal is to present code churn data for a given binary file to predict defect density in that file. Thus, we must be sure that the set of metrics we collect can be used to predict defect density effectively. We should also be able to display this data over a nearly arbitrary time period. The solution is to measure the churn over small date ranges, and construct a data model that allows us to obtain the code churn metrics for any given file and atomic date range pair. There is a conflict, however, between how fine-grained we want our data collection to be and how much data we can feasibly store. We would like to use smaller date ranges, if possible, but it turns out that making date ranges smaller is actually detrimental at a certain point. Measuring churn from day to day or from hour to hour would actually not provide better information than measuring from week to week, due to variations in working hours such as long meetings or days off, which both reduce the amount of actual coding that an employee is able to do. We leave it up to the project managers who enlist their projects in the code churn dashboard to specify this date range for the branches they own.

4.2 Collecting Branch Data

For our Code Churn Dashboard, we wished to show the user the same lists of products and branches that were available in BuildTracker. The user could then select from these lists and register the desired branches in our dashboard for code churn tracking. All of this branch and product data collection could have been done in real-time when a user of our Code Churn Dashboard was requesting the information, but we found that there was too much of a delay when waiting for the BuildTracker API to return the requested information. Therefore, we decided to run a daily job in order to populate the

branch and product information in our database, and then pull the data from there when a user requested that information. We had to choose whether we wanted to have faster page loads with potentially outdated data, or slower page loads with data that was pulled in real-time. We decided that we preferred to have faster load times. In the worst case, a user would have to wait 24 hours until the daily job ran again in order to see a new branch that had just been created in Source Depot, at which time they could choose to begin tracking it in our dashboard. The potential wait time didn't seem too problematic, especially since new branches aren't created very often (and old branches aren't deleted very often).

To achieve the necessary functionality, we pulled the product and branch information using the BuildTracker API. With the API, we were able to query BuildTracker for a list of all products, and then for each of these products, we were able to ask BuildTracker for a list of branches that belonged to each respective product. Then by traversing through each of these returned branch objects, we were able to analyze the enlistment specifications for each branch. If the branch object contained enlistment specification data pertaining to Source Depot, then this meant that the branch in question was being tracked in Source Depot. In this case, we pulled the necessary data pertaining to that branch from BuildTracker and stored it in our database. On the other hand, if the enlistment specification data didn't contain any Source Depot information, then we chose to ignore those branches because they were not being tracked in Source Depot, and therefore the code churn data related to those branches would not be obtainable by the Source Depot API.

For each product in BuildTracker, we stored the name of the product in our database. For each branch, we stored the name of the branch, the product to which it belonged, and the location of the branch within Source Depot (for example, `//depot/se/softgrid/4.6/SE_46_Mainline/`). The retrieval of the network path from the BuildTracker system required us to parse a string and separate the path from some other data that was included with it.

4.3 Collecting Source Code Churn Data

In order to collect the raw churn data needed to predict defect density, we needed to mine it from the Source Depot system described in Section 2.2.1. We used the Source Depot API to connect to the Source Depot server and collect data about the submitted changelists and their associated affected files.

In order to collect the raw churn data, the first step was to get a list of all changelists for each branch that was being tracked by the dashboard. Initially we collected all changelists for every branch, but this was changed when we discovered that Source Depot could filter its own results as we would no longer have to check if the changelist was for an enlisted branch. For each changelist, we then retrieved the time of the change and the list of affected files.

With the specific changes for each revision of every file, we could analyze those changes to calculate churn. Source Depot has the ability to provide 'diff summaries' between two revisions of a file, which gives the number of lines added, changed, and deleted. While it doesn't have a built-in way to calculate total lines of code, simply retrieving the number of lines added between the current revision and revision 0 (before the file existed) provided this metric. However, if the source depot operation was either an 'add' (a brand new file being added) or a 'branch' (a copy of a file from a different branch), we make a note of it in the database. This would allow us to implement the 'Include Adds' option on the dashboard discussed in Section 3.3.2. Finally, the Source Depot 'integrate' option could potentially result in many changes being applied at once. This can occur because the integrate operation indicates that changes from one branch are being merged to another. However, we opted to simply use Source Depot's aggregate of all these changes as it would otherwise skew the churn results to rank these operations much higher than others. We decided that if the individual churn for the files in the originating branch was desired, that branch could simply be enlisted as well. At this point, we had all of the data needed to analyze and calculate the relative churn metrics.

Finally, when a branch is enlisted the Dashboard administrator is allowed to select the interval that data should be collected. We stored the churn data by aggregating all data in periods of this interval. For example, with the default interval of one week each row would store the aggregate churn data for the preceding week. This allowed us to calculate the information a user desires when they access the dashboard page by analyzing the historical code churn data that had been stored in our database.

4.4 Collecting Binary Churn Data

To collect churn data on binary files, we need to determine which sources contribute to which binaries. This is done using the Srctool API (see Section 2.2.5). It is pretty trivial to check each line of API's output against the table containing all the source files we are tracking, and then insert a row to the `MapChanges` table with foreign keys to each of the binary and source files. It is also pretty straightforward to check where there was once a row in `MapChanges` but the mapped source file is no longer referenced in the pdb. In both cases we use LINQ statements to compare the `SrcPath` table to the srctool output and return the id's from the rows in `SrcPath` that had a match. However, many of the pdb's listed source files are not even human-written source code that gets checked into Source Depot. Since we know that none of those files will ever have a match in the `SrcPath` table, we would like to filter out such irrelevant file paths before we check them against every row of the `SrcPath` table. It turns out we can predict whether a file exists on source depot based on where it was in the build machine's directory tree. Anything from source depot goes into the C:\ drive, while other libraries are stored in separate locations. Each machine maps the root of the branch it is building to `C:\[aa]\[####]\`, where `[aa]` is two letters and `[####]` is a four-digit number. We use the regular expression `c:\\[A-Za-z]+\\\d+\\` in order to match, because the letter and number combinations could presumably move up to a greater number of characters as more and more branches are added. In some cases, we can ignore about half of the output of the .pdb by determining which files

can't be in source depot. Reducing the number of source files we have to look for in our `SrcPath` table gives a big performance bonus – especially as the `SrcPath` table gets larger and larger.

Once we have the map from binaries to sources established, we can compute the absolute churn measures for the binaries. We chose to measure binary and source churn data over exactly the same time intervals in order to make this step easier. Because the time segments always coincide, we can just do addition of all the source churn fields and arrive at the correct counts for the binary file. Even when a source maps to multiple binaries, we avoid having any conflicts between the source file's designated update interval and that of either binary file because the update interval is enforced across the entire branch.

5 Analytics

Once churn data is collected, we need a way to decide which files have been churned the most. We assign coefficients to each relative measure to compute a *code churn index* which indicates how much the file has been churned. Letting M_1 through M_6 be each of our 6 relative measures (see Figure 5-1) -- and C_1 through C_6 be their coefficients (see Section 2.1) -- we use the expression $C_1M_1 + C_2M_2 + C_3M_3 + C_4M_4 + C_5M_5 + C_6M_6$ as the indicator of how much a file has been churned (i.e. the churn index of the file). We leave the coefficients to be user-defined parameters in the dashboard, and a user must determine their coefficients based on how much interest they have in each type of churn. It should be noted that due to limitations of Source Depot, we were only able to calculate six out of the original eight relative code churn measures.

Absolute Measure	Description
Total LOC	The total number of lines of code in the file
Churned LOC	The number of lines added or modified since a previous revision
Deleted LOC	The number of lines deleted since a previous revision
File Count	The total number of source files that contribute to a binary (for a source file this is always 1)
Weeks of Churn*	The amount of time a file is open for editing
Churn Count	The number of intermediate revisions between two versions of a file
Files Churned	The number of contributing files that have been churned (for a source file this is always 1)

Relative Measure
Churned LOC / Total LOC
Deleted LOC / Total LOC
Files Churned / File Count
Churn Count / Files Churned
Weeks of Churn* / File Count
Lines Worked On / Weeks of Churn*
Churned LOC / Deleted LOC
Lines Worked On / Churn Count

Figure 5-1. The absolute and relative measures used by the Code Churn Dashboard.
** indicates metrics that could not be obtained due to limitations of Source Depot*

We leave the coefficients up to the user, but a user's choice of coefficients should be guided by what they are looking for in the data rather than made arbitrarily. For example, a tester looking to see where new features have been added might prioritize lines churned over lines deleted because deleted lines usually indicate that a change is actually a bug fix. A different set of coefficients might be used in order to predict which files are most likely to have had defects introduced into them. In theory, there should be an optimal list of coefficients for each interest a user could have. The researchers Nagappan and Ball derived a set to predict defect density in their research on relative code churn measures, but they warned that their data set was not very representative of code in general because they only analyzed data from a single software product. Therefore, we leave the churn index parameters up to user definition and leave the discovery of optimal sets of coefficients as a future task.

6 Future Work

The majority of the work done over the course of this project was devoted to retrieving and calculating code churn metrics. While the information displayed by the Dashboard's Analyze view discussed in Section 3.3.2 is extremely useful, there are many more ways one could use the data we found to gain insight about the code in a given project. In particular, risk analysis, integration with Product Studio, additional metrics, and addition analysis views are all areas where this product could be extended.

6.1 Code Delta

In order to increase the comprehensiveness of the analysis provided by the dashboard, it might be worth considering Hall and Munson's concept of code delta in "Software evolution: code delta and code churn." Whereas code churn is just a measure of how the code has physically changed over time, code delta measures how much the complexity of the code has changed over time. By including the measure of code delta, our system would be more perceptive of a small but very important change that would fly under the radar when only using raw churn metrics. The case when code delta can be a more accurate predictor of defect density than code churn is when very small but complicated code modifications are made to the code. For example, adding a call to a complex method (or changing a line within a widely used method) would appear as a single line of code added, but it could have larger effects throughout the entire code base. Because of the contrast between code churn and code delta, a much better picture of where potential defects may lie is given by analyzing the two measures in conjunction with each other. Therefore, it could be beneficial to explore this idea for further improvements to our Code Churn Dashboard system.

6.2 Product Studio Integration and Risk Analysis

In addition to Source Depot, the dashboard could also tap into data from Product Studio, Microsoft's bug-tracking system. Product Studio's database associates identified bugs with the changelist(s) that fixed them. Product Studio also keeps additional data about these bugs, such as their severity and priority. The severity and priority of these bugs can be used as an indicator of how fragile an individual code file is; a file that is associated with a lot of bugs -- or a few severe bugs – will pose a certain risk any time it is changed. Between this data and the code churn data we have already collected, the dashboard could expose a view that allows the user to enter a pending changelist and get an idea of how risky that change is. For example, if one of the proposed changes affects a file that causes frequent bugs or passes a certain threshold of code churn, it could be flagged to indicate that the change could be problematic and that additional testing is suggested. Depending on the diligence of the developer who entered the bug fix into Product Studio, the dashboard may even have further information, like what other areas of the codebase could be affected indirectly by the change.

6.3 Additional Code Churn Metrics

One of the absolute churn measures from Section 2.1 could not be obtained from Source Depot. This measure was "weeks of churn" (the total time a file was open for editing during a given time period), and was unobtainable because Source Depot does not keep track of this quantity. However, it would be possible for the dashboard to measure this quantity without requiring Source Depot to measure or store it. Currently, Source Depot keeps track of which files are currently checked out and by whom. The only problem is that once the file is checked back in, Source Depot keeps no record of the checkout. If the dashboard were to expose an API call that told it when a file was checked in or checked out, the dashboard's backend could be extended to keep track of weeks of churn and Source Depot would only have to be tweaked to call the API when a file is checked out. If this feature were to be

included in the future, the dashboard would gain two additional relative measures by which it could analyze churn.

6.4 Additional Dashboard Views

In Section 3.3.2, we discussed how we presented our churn data by showing the user the 'Top *N* most highly churned files.' Although this provides one way to view the data, the dashboard was developed in such a way as to allow alternative views to be added. For example, one possible view for source code would be a heat map of all source files, which shows all of the source files visually organized by their location in the branch, and would color-code them based on churn. This would allow a user to easily identify larger areas of a project that had high levels of churn, as opposed to only seeing data for one file at a time. In addition, this would allow the user to explore the available data more easily. For binaries, possible views that could allow a user to glean interesting information would be to show the relationships between binaries based on their component source files, and how churn in one affects others.

6.5 Dynamic Drop Folder

When pulling all of the branch data from BuildTracker (the name of the branch, the product which the branch belongs to, and the network path to the branch's source files), we were unable to retrieve the network path to each branch's most recent build drop location. Since the build drops are where the .pdb's are located, it would be very useful for the PopulateJob to automatically update each branch's `BinNetPath` to point to the most current build drop location. This way, the branch's administrator in the Code Churn Dashboard wouldn't have to manually change the `BinNetPath` every time there was a new build drop.

We were unable to conclude whether the BuildTracker API exposes the necessary functionality to achieve this. Each `Branch` object in BuildTracker can have `BuildShareGroup`s attached to it, which

then contain BuildDropCollections. In this case, it is easy to conclude that each of the BuildDrops in the BuildDropCollections belong to the given Branch, and then the DroppedTime of the BuildDrop can be looked at in order to determine which one is the most recent drop.

The uncertainty arises due to the fact that each Product object in BuildTracker can also have BuildShareGroups attached to them, each with BuildDropCollections belonging to those BuildShareGroups. We were unable to determine whether there is a link that goes from each of these BuildDrops back to a specific Branch object. For many Branches, all of the associated BuildDrops are associated with the Products, rather than the Branch itself, so this missing link from the Products' BuildDrops to the Branches they pertain to was what prevented us from being able to dynamically change the most recent build drop folder in the Code Churn Dashboard.

The BuildTracker website does successfully map the BuildDrops in question back to their individual Branches, so either we overlooked how it can be done with the BuildTracker API, or the necessary functionality was never exposed to the API. If it is not possible with the API, another possibility could be to look at the .log files that are generated in each build drop and parse them to find the names of the branches that they belong to. However, this would not be the most ideal scenario due to the time it would take to parse all of the files and the fact that not all .log files follow the same conventions.

6.6 User-Friendly Directory/Files Selection

From a user's perspective, it would be a lot easier to make file or folder selections via a clickable hierarchical display. This would make it a simpler experience for the user because it would no longer require them to type out full paths in order to get to the file or folder that they intended to input. This would also be beneficial because it would eliminate potential user errors where they incorrectly typed

the path that they were trying to target. Ideally, when implemented, this would be a popup that functioned similar to Windows Explorer.

6.7 Integrate Test Data

The dashboard would benefit from having access to data about testing. If the dashboard had a way to map code files to the test cases that cover them, then the dashboard could suggest a set of test cases to be run after a certain batch of changes. It would also be possible, once the map is created, for the dashboard to notify the user whenever it encounters a file which has no associated test cases. Untested code is a big vulnerability for the introduction of defects, and so reducing the amount of untested code would be a valuable service that the code churn dashboard could provide. Beyond alerting users when a file has no test cases, it could potentially be a research project to determine some heuristics by which the dashboard could determine when an already-tested file might need new test cases. Our research already suggests that certain relative measures will increase disproportionately when new features are being added versus when bug fixes are being applied. The dashboard might keep a counter of how many times each source file was churned in a way that indicates new features, and then alert the user after a certain number of such periods of churn in which no new test cases were added.

6.8 More Filters

As discussed in Section 3.3.2, the Analyze view of the dashboard allows a user to filter the results. Currently we have implemented one filter, which was the “Path Starts With” filter. (We also implemented the “No Adds” filter, but in our User Interface, this was presented as an option rather than a filter). Ideally, the Code Churn Dashboard should have a whole range of filters that can be set by the user. Some filter ideas that we discussed during the brainstorming phase were filters to “Exclude Date

Range," "Exclude Folder," and "Name Contains." These filters (along with any additional ones that might be necessary) would be very helpful to users as they tried to pinpoint the exact files that they were looking for. Furthermore, it would be useful if the user could add an arbitrary number of filters (rather than setting just the hard-coded ones that we have exposed to them). Ideally, this arbitrary number of filters could then be chained via the operations AND and OR, and negated via the NOT operator. In such a way, the user would be able to create a customized query that would help them select the exact files that they were interested in seeing churn data for.

6.9 Seamless Transition between File Types

Currently, the experience of viewing data for source files is fairly disjoint from the experience of viewing data for binary files. A very helpful improvement to the Code Churn Dashboard would be to come up with functionality so that the user could see more information about the binaries that a given source file contributes to (and vice versa). Currently, the only link between the two file types provided is the expandable drop down lists that are visible when a source file contributes to one or more of the binaries being tracked by the dashboard. However this link only displays the names of the binaries and does not any other associated information. Instead, churn metrics for those binaries could be displayed along with any other useful information that is known. In the same vein, when the user is looking at binary churn, all of the contributing source files could be displayed along with the metrics pertaining to them.

6.10 Applying Churn Coefficients to Specific Files and Folders

The Code Churn Dashboard implements the coefficients mentioned in Section 2.1, but currently it does so in a global manner. That is, the coefficient for a given metric is applied to all files equally. One area for future investigation would be the ability to weight different files or directories differently. This would allow a development team to put increased significance on changes made to a critical file that is

referenced in many places throughout the project over changes made to a less-important file. This would increase the churn index for those files, and in turn focusing effort and attention to them if many changes are made.

7 Conclusion

The goal of this project was to investigate code churn, define exactly what it is, and develop an application that would allow software development teams to easily see code churn measurements and take action on this data. If a piece of software ships with many noticeable defects, the software will not perform as expected which will result in a negative user experience. As a result, many software engineering teams shift focus toward finding and correcting bugs in their program, especially as the software nears release. The defect density prediction provided by the Code Churn Dashboard allows teams to target their testing on code likely to have flaws, and encourages developers to be careful when making changes that have a high probability of introducing errors.

From our research, we learned how to measure code churn, and what appropriate metrics would be. We then used these measures to implement the dashboard through a population program, a web-based user interface, and a SQL Server database. The dashboard we produced not only allows users to see the raw churn data, but was also written in an extensible manner and could have additional functionality implemented to make it even more useful to development teams. It will allow testing and sustained engineering teams to locate defects quickly and easily, resulting in a product that has fewer bugs and meets the team's goals.

8 Works Cited

Fenton, N. E. and N. Ohlsson. "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, pp. 797-814, 2000.

Hall, Gregory A. and John C. Munson. “Software evolution: code delta and code churn”, *Journal of Systems and Software*, Volume 54, Issue 2, 15 October 2000, Pages 111-118, ISSN 0164-1212, 10.1016/S0164-1212(00)00031-5.

Nagappan, Nachiappan, Brendan Murphy and Victor R. Basilli. “The Influence of Organizational Structure on Software Quality: An Empirical Case Study.”

Nagappan, Nachiappan and Thomas Ball. (2005). "Use of relative code churn measures to predict system defect density" in Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005, (p. 284).