Empirical Evaluation of Reliability Improvement in an Evolving Software Product Line

Sandeep Krishnan
Dept. of Computer Science
Iowa State University
Ames, IA 50014
sandeepk@iastate.edu

Robyn R. Lutz
Dept. of Computer Science
Iowa State University
& JPL/Caltech
rlutz@iastate.edu

Katerina Goševa-Popstojanova
Lane Dept. of CSEE
West Virginia University
Morgantown, WV 26506-6109
Katerina.Goseva@mail.wvu.edu

ABSTRACT

Reliability is important to software product-line developers since many product lines require reliable operation. It is typically assumed that as a software product line matures, its reliability improves. Since post-deployment failures impact reliability, we study this claim on an open-source software product line, Eclipse. We investigate the failure trend of common components (reused across all products), high-reuse variation components (reused in five or six products) and low-reuse variation components (reused in one or two products) as Eclipse evolves. We also study how much the common and variation components change over time both in terms of addition of new files and modification of existing files. Quantitative results from mining and analysis of the Eclipse bug and release repositories show that as the product line evolves, fewer serious failures occur in components implementing commonality, and that these components also exhibit less change over time. These results were roughly as expected. However, contrary to expectation, components implementing variations, even when reused in five or more products, continue to evolve fairly rapidly. Perhaps as a result, the number of severe failures in variation components shows no uniform pattern of decrease over time. The paper describes and discusses this and related results.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product metrics, Process metrics*

General Terms

Reliability

Keywords

Software product lines, reliability, failures, reuse, change

1. INTRODUCTION

Reliability is important to software product-line developers since many product lines require reliable operation. It is typically assumed that as a software product line matures, its reliability improves. The systematic reuse opportunities provided by software product line (SPL) techniques are considered to be an important factor in achieving increased quality and reliability across the product line. Empirically investigating the relationship between reuse and reliability in a gradually evolving software product line can help us understand the utility of SPL techniques and, perhaps, to improve existing SPL practice.

We follow Weiss and Lai in defining a product line as "a family of products designed to take advantage of their common aspects and predicted variabilities [26]." It has been reported that planned reuse of artifacts allows more rapid development of new products and lower-cost maintenance of existing products [4], [11], [24], [26]. Since the artifacts such as design, code, etc. are reused and maintained via a centralized, domain-engineered repository, there is reason to anticipate that the quality and reliability of both the existing products and the new products may improve over time.

In a product line, some requirements are shared by all the products in the product line. These are the common requirements and are called *commonalities*. The components implementing commonalities are called *common components*. The products also differ from each other based on a set of variation requirements called *variabilities*. The components implementing variations are called *variation components*. As the common and variation components are reused across products, they go through iterative cycles of testing, operation and maintenance that over time identify and remove many of the bugs that can lead to failures.

We define reliability as continuity of correct service [6]. A failure is a departure of the system or system component behavior from its required behavior, while a fault is an accidental condition which, if encountered, may cause the system or system component to fail to perform as required. Post-deployment failures are indicators of the reliability of the system or components.

In this paper we study how reliability, measured by the number of post-deployment failures, changes as Eclipse, a large, open-source software product line evolves over time. Eclipse documentation describes Eclipse as a platform for building integrated web and application development tooling. Eclipse provides a range of products based on the needs of different user groups. Since post-deployment failures impact reliability, we study the failure trend of common components and variation components as Eclipse evolves. We also study how much the common and variation components change over time. The relationships between reliability and change, and between quality and change, in a single system have been studied in several important and insightful papers, as described in Section 8. Most relevant to this study, Mohagheghi and Conradi have analyzed defect density in product lines [19] [20]. We are not aware, however, of other work that has quantitatively studied reliability in terms of failure occurrence and change within the context of a real-world, evolving product line.

We consider three types of SPL components: common components reused in all products; high-reuse variation components, used in many but not all products; and low-reuse variation components, used in only one or two products. The research questions that we investigate are described in detail in Section 4 and briefly here.

1. *Failure trends.* Do serious failures (both in terms of raw numbers and percentages of all failures) decrease over time as the common/high-reuse variation/low-reuse variation components evolve over releases?

2. *Change trends.* Does the percentage of new files and/or modifications to the source code decrease across releases for the common/high-reuse variation/low-reuse variation components?

3. *Failure/Change relationships.* Does the number of serious failures normalized for source-code changes decrease over time for the common/high-reuse variation/low-reuse variation components.

Several interesting findings result from the investigation and are described in the rest of the paper. The main contributions of the work are:

- As the product line evolves, fewer serious failures occur in components implementing commonality, and these components also exhibit less change over time.

- The occurrence of failures in variation components shows no uniform pattern of decrease as the product line evolves, even when normalized for the occurrence of change.

- Although the number of failures in some variation components decreases as the product line matures, the percentage of severe failures in those components holds steady or even increases.

- Components implementing variations, even when reused in five or more products, continue to evolve fairly rapidly.

- In common components, the percentage of new files shows a decreasing trend as the product line evolves.

- In variation components that are lightly reused, the percentage of new files generally shows a decreasing trend, comparable in values to one of the common components.

- Heavily reused variation components have a very low percentage of new files, much lower than either common components or lightly reused variation components.

This paper is organized as follows. Section 2 describes Eclipse and gives the reasons for considering it as a software product line. Section 3 presents the approach to data collection and analysis. Section 4 lists the research questions that are investigated. Section 5 presents the findings. Section 6 summarizes and discusses the results in the context of SPLs. Section 7 discusses threats to validity. Section 8 describes additional related work. Section 9 provides concluding remarks.

2. ECLIPSE PRODUCT LINE

In the commercial sector, multiple industries claim to have experienced the benefits of software product line engineering techniques. The SPL Hall of Fame [5], for example, lists leading industries that have successfully introduced SPL techniques into their production environments. It is accepted widely that product line engineering techniques improve the quality of the products [4], [16], [26]. However, empirically investigating such claims is difficult because it requires data that spans the evolution of the product lines. There is, in general, a lack of available product line data.

Eclipse is a notable exception. The evolution of Eclipse's products is documented in public failure reports, change reports and source code available across its component releases. The Eclipse web site describes Eclipse as an ecosystem due to its plugin-based architecture. Developers belonging to varied software communities can develop plugins which are integrated with other existing plugins. This makes Eclipse flexible enough to be used by different user communities.

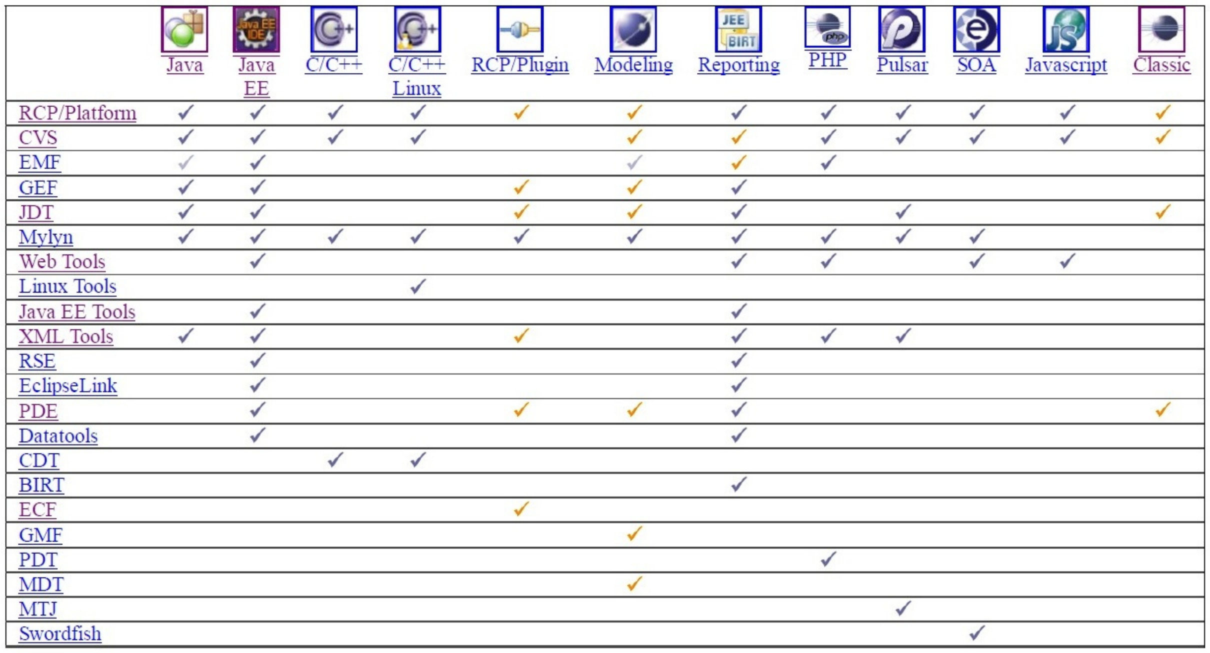
Another view is to look at Eclipse from a product line perspective. Following Chastek, McGregor and Northrop [9], we consider Eclipse as a software product line. Eclipse provides a set of different products to satisfy the needs of different user communities. Each product has a set of common features, yet each product differs from other products based on some variation features. The features are developed in a systematic manner with planned reuse for the future. The features are implemented in Eclipse as plugin components which are integrated to form the products. The products of the Eclipse product line are the multiple package distributions provided by Eclipse for different user communities.

2.1 Products

Each year Eclipse provides more products based on the needs of its user-communities. For Java developers, the Eclipse Java package is available; for C/C++ developers, Eclipse provides the C/C++ distribution package, etc. In 2007, five package distributions were available: Eclipse Java, Eclipse JEE,Eclipse C/C++, Eclipse RCP and Eclipse Classic. In 2008, two more products became available: Eclipse Modeling and Eclipse Reporting. 2009 saw the introduction of Eclipse PHP and Eclipse Pulsar. In 2010, Eclipse had twelve products, including three new ones: Eclipse C/C++ Linux, Eclipse SOA and Eclipse Javascript. Fig. 1's columns list the 2010 products. In each release, new products are introduced based on the needs of the user communities by reusing the common components and existing variation components, and by implementing any required new variabilities in new component files.

2.2 Components

The products are composed of components which are implemented as plugins. For the 2010 release, the components in the Eclipse product line are shown as rows in Fig. 1. The



	Java	Java EE	C/C++	C/C++ Linux	RCP/Plugin	Modeling	Reporting	PHP	Pulsar	SOA	Javascript	Classic
RCP/Platform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CVS	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
EMF	✓	✓				✓	✓	✓				
GEF	✓	✓			✓	✓	✓					
JDT	✓	✓			✓	✓	✓		✓			✓
Mylyn	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Web Tools		✓					✓	✓		✓	✓	
Linux Tools				✓								
Java EE Tools		✓					✓					
XML Tools	✓	✓			✓		✓	✓	✓			
RSE		✓					✓					
EclipseLink		✓					✓					
PDE		✓			✓	✓	✓					✓
Datatools		✓					✓					
CDT			✓	✓								
BIRT							✓					
ECF					✓							
GMF						✓						
PDT								✓				
MDT						✓						
MTJ									✓			
Swordfish										✓		

Figure 1: **Eclipse Product Line for the year 2010** [`http://www.eclipse.org/downloads/compare.php`]

individual cells indicate which components are used/reused in which products.

We observe three categories of components: common, high reuse variation and low-reuse variation.

Common components. The first category contains the common components reused in all products. The large component RCP/Platform is the only common component reused across all products. Henceforth in the paper, we abbreviate the RCP/Platform component to Platform. To consider a finer level of granularity we look at the failure trends for the five of the fourteen subcomponents of Platform with the largest number of severe failures: Resources, Runtime, SWT, UI and Debug. We choose these particular subcomponents because they contribute over 70% of the failures in Platform. We refer to each subcomponent as Platform-subcomponent_name. Failure numbers for the Platform component are the sum of the failures for each of its fourteen subcomponents and we refer to the sum as Platform-combined.

High-reuse variation components. The second category is the set of variation components with high reuse. This category consists of the components that are reused in some but not all products, and the number of products in which these components are reused increases with each subsequent release from 2007 to 2010. The components in this category are EMF, GEF, JDT, Mylyn, Webtools, XMLtools, and PDE.

Low-reuse variation components. The third category is the set of variation components with low reuse. This category includes components that are reused across some but not all products, and the number of products in which they are reused does not increase with each release. These components are reused in at most two products in the releases. The components in this category are CDT, Datatools and Java EE Tools (called JEEtools here).

For this study, we analyzed eleven components as well as five subcomponents of the common component, Platform. Table 1 lists the components studied.

Table 1: **List of components**

Category	**Component**
Common	Platform
High-reuse variation	EMF GEF JDT Mylyn Webtools XMLtools PDE
Low-reuse variation	CDT Datatools JEEtools

3. APPROACH

We observe the four most recent releases of the Eclipse Product Line, from 2007 till 2010. The individual components that form the products were available before 2007. However, the integration of these components into products began from 2007, leading us to select these four releases for our investigation.

In this work we focus on the effect of evolution for each component across these four releases on the post-release failures of the components. Post-release failures are those that occur after the software is operational. For a user-community, the number of post-release failures encountered strongly affects their opinion of the quality of the software. We also analyze the amount of change in the source code of these components across the four releases.

3.1 Failure Trends in an Evolving SPL

3.1.1 *Data Sources and Severity Categories*

The failure reports come from the public Eclipse Bugzilla database [2]. Users can query the database through a web interface and retrieve the results in graphical or textual format. We collect data for five of the six [3] severity categories: *blocker*, *critical*, *major*, *normal* and *minor*. We exclude the *trivial* failure category as these do not contribute significantly toward reliability.

We consider the failures in the top five severity categories to be the *total* failures for a given component. We aggregate the failures in the most serious three categories (blocker, critical and major) into a single category called *severe failures*. These failures all have serious consequences for the user, such as a major loss of functionality, crash, or blockage without a workaround.

A total of 9,266 failures are identified for the 11 components considered in this study across the four releases. Of these, approximately 1,542 are severe failures. The number of total failures and severe failures for each release is shown in Table 2.

Table 2: **Number of total and severe failures**

Year	**2007**	**2008**	**2009**	**2010**	**Total**
All Failures	2928	2781	2089	1468	**9266**
Severe Failures	496	497	303	246	**1542**

3.1.2 Data Collection, Integration and Analysis

For each of the components listed in Table 1, we query the Eclipse Bugzilla database and retrieve the number of total failures and the number of severe failures. In the Bugzilla database, data can be retrieved for each component or for each of the individual subcomponents of the component. We first map the plugins to the components for each distribution of every product. We collect the post-release failure data for only those subcomponents that have corresponding plugins in the product distributions. This is to ensure that we consider failures for only those subcomponents present in the distribution and not for other subcomponents that may not have been present during that particular release. The numbers for these subcomponents are aggregated to calculate the number of failures for each component. Although this approach is more time consuming than directly finding the numbers for the components from Bugzilla, it allows us to get more accurate numbers for the components based on the product distributions.

We analyze the collected data in two ways. First we calculate the raw number of severe failures for each of the three categories of components (common, high-reuse variation and low-reuse variation). This is to investigate whether there are interesting patterns such as decreasing or increasing trends in the number of severe failures. The second way that we look at the data is by the percentage of severe failures. To detect, when a failure occurs, how often its effects are judged to be severe and how this factor changes over time, we determine the percentage of the total failures that are severe. If the percentage of severe failures increases or remains stable over time, it may indicate that the impact of failure on users is not necessarily decreasing, even if the number of severe failures is decreasing.

3.2 Change Trends in an Evolving SPL

3.2.1 Data Sources and Type of Changes

In order to measure the amount of change to the source-code, we mine the CVS release repository of Eclipse, which is our source for change data. There are two ways in Eclipse to readily observe software change. This study uses both these types of change to try to characterize the SPL evolution. The first kind of observable change is changes to existing files. We measure change to existing files in terms of modifications to existing code. Since the number of modifications is large, we calculate the number of Kchanges to the source code in each release of a component. Kchanges is the number of modifications to existing files for that component, divided by 1000. The second kind of observable change is change via new files. Since the number of new files is not as large, we calculate the percentage of files that are new for each release of each component.

3.2.2 Data Collection, Integration, and Analysis

We use the tool *cvschangelogbuilder-2.5* [1] to query the CVS repository. The plugins for each component are available in the different Eclipse product distributions/packages. The plugins associated with these components are also annotated with the corresponding release numbers. Using this information of plugin name and associated release number, we query the CVS repository. The commit information also is annotated with whether files are changed or added. Using this information, we retrieve the number of changes and the number of additions made to the source-code per release using textual pattern matching. We match patterns like *changed* and *added* and find the number of times files are changed and number of new files added. We aggregate the number of changes for all plugins of a component to calculate the number of additions and modifications for each component. Since data is not available for all releases of some components, we exclude these components (PDE, Mylyn, EMF and Datatools) from the change analysis. However, we collect data for majority of components in each of the three categories and analyze them. Of the 11 components examined in this study, data is available and retrieved for 7 components (Platform, JDT, GEF, Webtools, XMLtools, CDT, and JEEtools).

4. RESEARCH QUESTIONS

To investigate whether the reliability of products (measured in terms of their components) in the Eclipse PL improved with reuse, we studied the following research questions.

1. ***Failure trends***

(a) Failure trends for common components

   (i) Do the number of severe failures *(blocker, critical and major)* decrease with time as the common component is being reused across multiple releases?

   (ii) Does the percentage of severe failures decrease with time as the common component is being reused across multiple releases?

(b) Failure trends for high-reuse variation components

   (i) Do the number of severe failures decrease with time as the high-reuse variation components are being reused across multiple releases?

   (ii) Does the percentage of severe failures decrease with time as the high-reuse variation components are being reused across multiple releases?

(c) Failure trends for low-reuse variation components

  (i) Same as b(i) but for low-reuse variation components.

  (ii) Same as b(ii) but for low-reuse variation components.

2. ***Change trends***

(a) Does the percentage of new files and/or modifications to the source code for the common components decrease across releases?

(b) Does the percentage of new files and/or modifications to the source code for the variation components decrease across releases?

3. ***Failures/Change relationship***

(a) Is there a decrease in the number of failures with respect to changes (new file creation/code modifications) for the common components across releases?

(b) Is there a decrease in the number of failures with respect to changes (new file creation/code modifications) for the variation components across releases?

5. OBSERVATIONS

There were several interesting observations from our study.

5.1 Failure Trends

5.1.1 Failure Trend for Common Components

These are the components that have been reused in all products.

(i) *Number of severe failures decreases over time, as expected.* Fig. 2 shows the decreasing number of severe failures for the five individual subcomponents of Platform and for the aggregate Platform-combined which is the sum of the failures for its fourteen subcomponents. Note that the 2007 Release is labeled as 1, 2008 Release as 2 and so on.

(ii) *Percentage of severe failures tends to stabilize and even shows a gradual increase over time, contrary to our expectations.* As shown in Fig. 3, the percentage of severe failures for Platform-combined tends to remain in the range of 14.5% to 17%, rather than continuing to drop.

In fact, for Platform-combined, the percentage of severe failures increases over the last three releases. SWT, UI, Resources and Platform-combined show an increase in the percentage of severe failures from release 3 to release 4. Of the remaining two subcomponents, Runtime and Debug, Debug shows a decrease of approximately 1% only from release 3 to 4. Only Runtime subcomponent shows a significantly decreasing trend over the last three releases. This shows that from release 3 to 4 the percentage of severe failures does not exhibit a significant decrease for the common subcomponents.

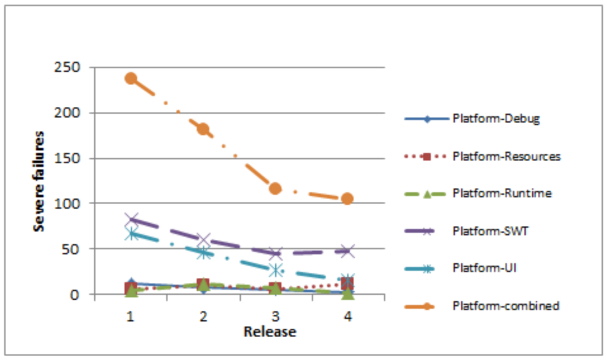


Figure 2: **Number of severe failures in common components**

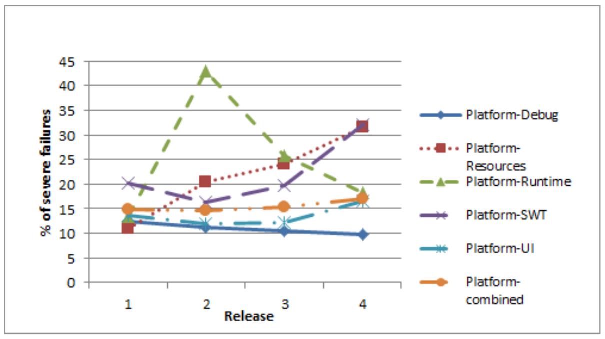


Figure 3: **Percentage of severe failures in common components**

5.1.2 Failure Trend for High-reuse Variation Components

These are the components which are reused increasingly in multiple products across the four releases.

(i) *Number of severe failures does not monotonically decrease over time and shows mixed behavior, contrary to our expectations.* Fig. 4 shows that this monotonic decrease occurs for some components (JDT), but not for others. For example, PDE, Mylyn and GEF show an increase in the number of severe failures from release 1 to 2. EMF shows an increase in the fourth release. Other components show uneven behavior. For example, Webtools and XMLtools show an increase in severe failures from release 1 to 2, then a decrease from release 2 to 3 and again an increase from release 3 to 4.

The data suggest that for variation components with high reuse, the trend for number of severe failures over time is highly mixed and dependent on the component. While we would expect that highly reused variation components tend to behave like common components, only a few of them do.

(ii) *Percentage of severe failures also shows a mixed trend contrary to our expectations.* Fig. 5 shows that the values for percentage of severe failures also show similar uneven trends. PDE and the last three releases of JDT show trends similar to the common Platform-combined component, with the percentage of severe failures tending to stabilize at 9.5% to 13%. However, the values for Webtools and XMLtools fluctuate in a large range of 4% to 27% with alternating increases and decreases in the percentage values. Six of the seven components have a higher percentage of severe failures in release 4 than in release 3. Interestingly, although the number of severe failures for JDT steadily decrease from release 2 to 4, the percentage of severe failures steadily increases for these releases.

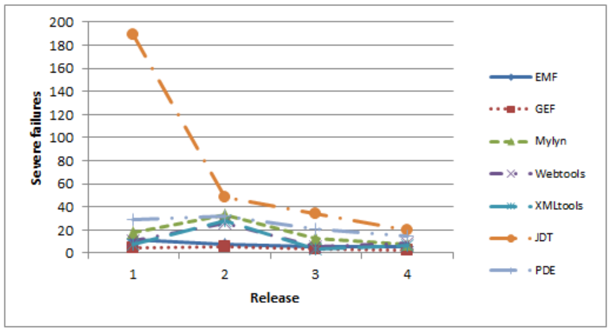


Figure 4: **Number of severe failures in variation components with high reuse**

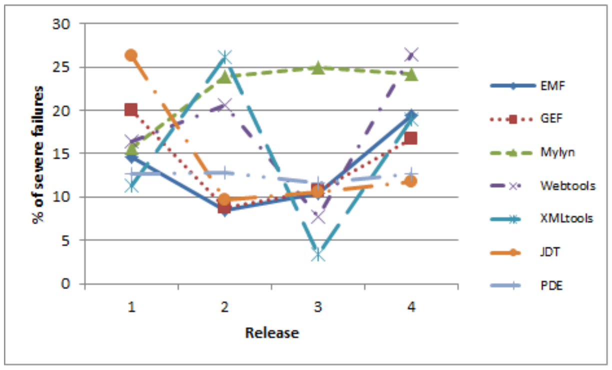


Figure 5: **Percentage of severe failures in variation components with high reuse**

5.1.3 Failure Trend for Low-reuse Variation Components

Variation components with low reuse display tendencies similar to variation components with high reuse.

(i) *Number of severe failures show mixed trends and do not monotonically decrease.* The low-reuse variation components have a higher number of severe failures than most of the high-reuse variation components, which matches our expectations. However, we also observe mixed trends. Fig. 6 shows an overall decrease in the number of severe failures from release 1 to 4 for CDT. The number of severe failures for JEEtools increases from release 1 to 2, then decreases from release 2 to 3 and then remains stable from release 3 to 4. Datatools shows an alternate rise and drop in the number of severe failures. The number of severe failures do not monotonically decrease for all components; rather there is a mixed behavior.

(ii) *Percentage of severe failures shows mixed trends and not a decreasing trend.* For low-reuse variation components, the percentage of severe failures fluctuates less than for high-reuse variation components. Only CDT shows a steady decrease in percentage values as seen in Fig. 7. JEEtools shows an increase from release 1 to 2, then a decrease from 2 to 3 and the remains stable from 3 to 4, whereas Datatools first decreases, then increases and remains stable. Thus, the percentage of severe failures shows mixed results similar to the trends for number of severe failures.

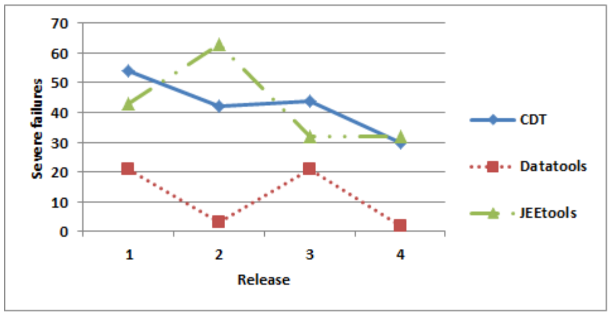


Figure 6: **Number of severe failures in variation components with low reuse**

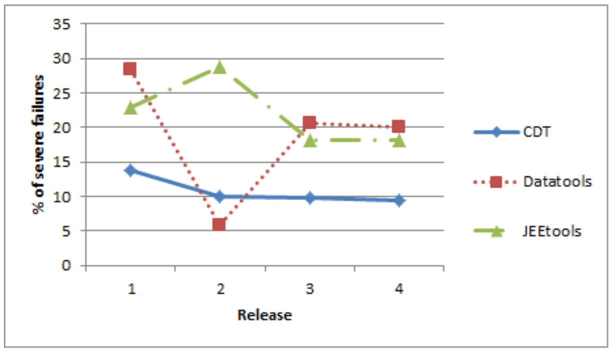


Figure 7: **Percentage of severe failures in variation components with low reuse**

5.2 Change Trends

Section 3 described two kinds of observable changes as Eclipse evolved: creation of new files and modifications to existing files. We first discuss trends related to new files and the occurrence of failures in the next subsection, and then trends related to modifications to existing files and the occurrence of failures in the following subsection.

5.2.1 SPL Evolution With Respect to New Files

We investigate the amount of change in the common components by calculating the percentage of new files for each component in each release. Table 3 gives the percentage of new files per release for each component we analyze. Cells with "No-info" indicate that the data for that particular time period were not available in the Eclipse repository. Cells with "-" indicate that there were no new files added in that time period. For the common components, the percentage of new files gradually decreases across the four releases. This is consistent with the SPL expectation that the common components, since they contain features shared by all products, will be relatively stable. We also see that in the initial release, the percentage of new files is very high.

For the high-reuse variation components, the percentages of new files are less than for the common components/ subcomponents. The percentages of the files that are new for a given component tend to be stable across releases, rather than showing a decreasing trend (as the common components do).

For the low-reuse variation components, the percentages of new files are comparable to the common components and subcomponents. These percentages are also much higher than for the high-reuse variation components. They show an overall decreasing trend as per our expectations with the last release of JEEtools being an exception.

5.2.2 SPL Modification of Existing Code

Modification to existing code is observed by calculating the number of code changes normalized to the number of files over the sequential releases of the components. There is no overarching trend, except that a single component, SWT, has a significantly larger modification rate over time than any other component. For space reasons we do not include this table. The two smallest common components in this study, in terms of number of files (Resource and Runtime), have the lowest modification rate among the common components. However, even the common components do not show a decreasing modification rate across releases. To summarize, existing as well as new files show significant change, and even the common components were not reused intact, but were modified on an ongoing basis.

Table 3: **Percentage of new files for commonalities and variabilities**

Category	Component	Percentage of new files			
		2007	2008	2009	2010
Common	Debug	14.87	4.32	4.26	3.16
	UI	9.10	6.57	4.80	No-info
	SWT	28.82	15.77	7.70	4.74
	Resources	42.76	1.32	-	5.15
	Runtime	-	-	-	-
High-reuse	JDT	2.28	6.43	1.91	1.03
	Webtools	1.44	13.30	1.26	1.23
	XMLtools	3.78	7.41	2.21	4.18
	GEF	0.53	1.90	0.68	2.18
Low-reuse	CDT	22.97	12.05	3.53	3.51
	JEEtools	29.92	9.44	2.13	7.31

Table 4: **Failures/new-file for commonalities and variabilities**

Category	Component	Failures/new-file			
		2007	2008	2009	2010
Common	Debug	0.09	0.21	0.15	0.10
	UI	0.27	0.25	0.19	No-info
	SWT	0.44	0.58	0.83	1.37
	Resources	0.05	2.50	-	0.86
	Runtime	-	-	-	-
High-reuse	JDT	1.95	0.17	0.40	0.43
	Webtools	0.16	0.03	0.07	0.11
	XMLtools	0.09	0.15	0.05	0.06
	GEF	1.33	0.55	0.75	0.15
Low-reuse	CDT	0.05	0.07	0.25	0.17
	JEEtools	0.03	0.11	0.24	0.06

5.3 Failure/Change Trends

One reason for increased failures might be large amounts of new/changed code that introduced faults. To analyze the failure/evolution relationship, for each component we extracted the number of new files added in each release and the number of times existing files were changed. Table 4 shows the number of severe failures per new file and Table 5 the number of severe failures per 1000 changes.

Table 5: **Failures/Kchanges for commonalities and variabilities**

Category	Component	Failures/Kchanges			
		2007	2008	2009	2010
Common	Debug	6.92	7.47	14.81	4.71
	UI	18.02	14.79	12.72	No-info
	SWT	26.64	6.90	7.83	10.86
	Resources	44.44	50.51	34.88	35.82
	Runtime	200.00	857.14	235.29	250.00
High-reuse	JDT	59.94	5.95	3.84	8.51
	Webtools	4.47	2.31	3.95	3.24
	XMLtools	3.02	17.29	2.65	5.91
	GEF	90.91	39.74	51.72	1.62
Low-reuse	CDT	6.65	5.83	10.98	9.69
	JEEtools	6.54	17.51	14.00	6.56

5.3.1 Failure/Evolution Relationship for New Files

Interestingly, the rise in the percentage of new files for three of the four high-reuse variation components (as seen in Table 3) is accompanied by an increase in the number of severe failures (as seen in Fig. 4). For the second release, Webtools, XMLtools and GEF show an increase in the percentage of new files and also a corresponding increase in the number of severe failures from the first release. Similarly in the fourth release, XMLtools shows an increase in the percentage of new files and also an increase in the number of severe failures. This may indicate a relationship between the failures and the number of new files.

Table 4 shows a non-uniform increase or decrease in the ratio of failures over new files. However, comparing the Failures/new file values of release 1 to release 4, there is one observation that distinguishes the common components from the variation components. With the exception of the Debug subcomponent, for the other two *common* subcomponents for which we have valid data (SWT and Resources), release 1 has a lower Failures/new-file value than release 4, with the difference being more than 0.8. This means that the addition of new files in later releases of the evolution led to more failures. For the *high-reuse variation* components, however, we observe that the values in release 1 are always higher than in release 4, which may mean that the addition of new files in later stages did not lead to as many failures as in the early stages. *Low-reuse variation* components also show trends similar to the *common* subcomponents.

5.3.2 Failure/Modification Relationship for Existing Files

As a reminder, Kchanges is the number of modifications to existing files divided by 1000. Table 5 shows that most components do not have a steadily decreasing rate for Failures/Kchanges. Even for *common* components, the Failures/Kchanges decreases over releases for only one of the five subcomponents (UI). For Debug, Failures/Kchanges increases in the first three releases, and for SWT it increases from release 2 to 4. Resources and Runtime first show an increase, then a decrease and then tend to remain stable. The *high-reuse* and *low-reuse* variation components show similarly mixed trends in the Failures/Kchanges. With respect to changes, failures fluctuate a great deal with no distinguishing upward or downward trend.

6. DISCUSSION OF THE RESULTS

The highlights of the empirical observations about post-deployment failures and stability of changes in the open-source, evolving product line Eclipse are summarized as follows:

1. Components/subcomponents implementing commonality reused in every product exhibit fewer serious post-deployment failures across releases.

2. Variable components, both heavily and lightly reused, do not show a monotonically decreasing trend for post-deployment failures across releases. No obvious trend is observed even when failures are normalized for the number of changes made to existing files or for the number of new files.

3. Although the number of failures in some variation components decreases as the product line matures, the percentage of severe failures in those components holds steady or even increases.

4. The percentages of new files in common components show a decreasing trend as the product line evolves through releases. The values of these percentages are roughly comparable to lightly reused variation components, but higher than for heavily reused variation components.

Briefly, as expected, common components experience fewer severe post-deployment failures and less change as the product line matures through releases. Conversely, contrary to typical expectations, variable components, even if reused in multiple products, do not show a decreasing pattern either in post-deployment failures or in the changes made/new files added across subsequent releases. These findings clearly indicate that the improvement of post-deployment quality and the stability of source code do not depend solely on how often components are reused.

None of the Eclipse components considered in our study was reused intact(“as-is”). “As-is” reuse without change to existing components might have led to more straightforward conclusions about the benefits of reuse in software product lines. The extent of enhancements/new features added with each release is one of the factors that may help explain the mixed results for the variation components and that may determine the benefit of reuse for software product lines such as Eclipse that undergo rapid evolution. This finding of on-going change in reused elements merits further investigation that should take into account the amount of change measured at a finer granularity (e.g., blocks or lines of source code), and possibly include metrics such as the size and complexity of the components.

The mixed results of this study also suggest that there may be other factors associated with these reuse components, such as the amount of pre-deployment testing and the extent of field usage, that are not accounted for in this analysis. Unfortunately, as pointed out in [10], this information is typically unavailable to allow more in-depth study in this direction.

7. THREATS TO VALIDITY

This section discusses the internal and external validity of the study.

A threat to the internal validity of the study is that we analyzed only one common component, namely Platform. To moderate this we investigated five subcomponents of Platform. Platform is a large component, and each of these subcomponents is comparable in size to other components in our study. Each of these Platform subcomponents provides a specific common functionality. Also, each of these subcomponents has a large number of severe failures.

Another threat to internal validity is the limited number of releases in the study. While analyzing more releases might give additional insight into the trends, the 2007-2010 releases provide a representative picture of the current product-line situation. We decided not to include the minor quarterly releases into our analysis because there were fewer users downloading them and because the entries in the bug database for these minor releases were missing data for several components. Some of the minor releases reported higher numbers of failures while others did not report any. We plan to observe future releases as they come and incorporate the new data for analysis.

A third threat to the internal validity is that the number and severity of failures may be affected by the expertise of the programmers who worked on the components. To alleviate this, we have normalized the failures with respect to the number of changes and additions to source code files, rather than normalizing it with the lines of code for each component. This is in accordance with Mockus, Fielding and Herbsleb who identify the programmer’s lack of expertise, leading to unnecessarily lengthy code, as one of the reasons for seemingly lower failure density [18]. In addition, because each component in Eclipse is typically developed by multiple programmers, this threat may be alleviated.

An external validity threat to this study is the extent to which these conclusions can be generalized to other product lines. Eclipse is a large product line with many developers in an open-source, geographically distributed effort. This means that the development of Eclipse product line is probably more varied in terms of the people involved and the development techniques used than in commercial product lines. The effect of the large number of contributors (e.g., the Platform component lists more than 100 developers) remains to be studied. Chastek, McGregor and Northrop consider the open-source development to be largely beneficial in terms of quality [9]. We hope to study other open-source SPLs and currently are studying a commercial SPL to learn more about reuse, change and reliability in SPLs.

8. RELATED WORK

There have been many studies observing the failure trends or profiles for commercial as well as open source systems. Work to date has been done to identify causes of failures, distribution of different types of failures, classification of the consequences of failures, comparison of the failure density of open-source systems to commercial systems, and defect/failure prediction for individual systems. However, studies investigating the effects of software product line engineering and their benefits by mining the failure databases are rare.

The work most closely related to ours is that of Mohagheghi and Conradi [19], [20], who reported on a system developed using a product family approach. They compared the fault density and the stability (amount of change) of the reused and non-reused components. They observed that reused components have lower fault density and less modified code as compared to non-reused components. Our work is similar to theirs in [19] in that both consider the effect of reuse on the quality of product lines, but we focus on post-deployment failures, rather than fault densities, since failures are experienced by end users and affect reliability more directly. Further, we consider the effects of code change specifically on the severe failures. Eclipse also involves components reused across more products than in [19].

For Eclipse, Zimmermann, Premraj and Zeller [28] performed defect prediction for different releases by using the data from the Bugzilla database. Like us, they used the Eclipse data, but their purpose was primarily to predict, based on static attributes, the defects in the next release. Mockus, Fielding and Herbsleb [18] investigate the effectiveness of open-source software development methods on Apache in terms of defect density, developer participation and other factors. They showed that for some measures of defects and changes, open-source systems appear to perform better while for other measures, the commercial systems perform better. In our study we use one of the measures they recommend.

Fenton and Ohlsson [10] analyzed the faults and failures in a commercial system and tested several hypotheses related to failure profiles. Their results related the distribution of faults to failures and the predictive accuracy of some widely used metrics. They found that pre-release faults are an order of magnitude greater than the operational failures in the first twelve months. Lutz and Mikulski [15] analyzed serious failures/anomalies in safety-critical spacecraft during operations. Hamill and Goševa-Popstojanova [13] conducted a study of two large systems to identify the distribution of different types of software faults and whether they are localized or distributed across the system. They analyzed different categories of faults and their contribution to the total number of faults in the system. Børretzen and Conradi [7] performed a study of four business-critical systems to investigate their fault profiles. They classified the faults into multiple categories and analyzed the distribution of different types of faults.

Paulson, Succi and Eberlein [23] investigated the growth pattern of open-source systems and compared them with that for commercial systems. They found no significant difference between the two in terms of software growth, simplicity and modularity of code. They found, however, that in terms of defect fixes, open-source systems have more frequent fixes to defects.

A comparative study of software reliability modeling for open source software was performed recently by Rahmani, Azadmanesh and Najjar [25]. They analyzed five open source software systems, collected the failure reports for them and compared the prediction capability of three reliability models on this data. One of their results was that the failure patterns for open-source softwares follow a Weibull distribution.

There has been a significant amount of work in the area of defect prediction for both commercial and open-source software. Studies reported in [28], [21], [22], [27], [12] have used bug reports and bug repositories such as Bugzilla for predicting defects and failures. Jiang, Menzies, Cukic and others [14], [17] have used machine learning algorithms to perform defect prediction.

Catal and Diri [8] recently found that 60% of the 74 papers they reviewed used non-public (private or commercial) data sets, making it difficult to reproduce the results. Only 31% of the papers used public datasets, and the rest were either partial or their source was unknown. Their findings confirm the need for public datasets such as those offered in the Eclipse repository.

9. CONCLUSION

The work reported here considers Eclipse as an evolving product line and distinguishes common components that implement commonalities and are shared across all products from high-reuse components that implement variations reused in many products, and from low-reuse components that implement variations used in only one or two products. We study the occurrence of severe failures, change to source code, and addition of new files over time across the four most recent Eclipse releases. The motivating question is whether the data from the Eclipse bug, change, and source code release repositories support the typical expectation that as a software product line matures, its reliability, as measured by serious post-deployment failures, improves.

The quantitative results are mixed. In support of the claim is that fewer serious failures occur in components/ sub-components implementing commonality, and that these components exhibit less change over time. Moreover, in common components the percentage of new files shows a decreasing trend as the product line evolves. However, the percentage of new files for the common components is higher than for variation components that are heavily reused. The occurrence of failures in variation components shows no uniform pattern of decrease, even when normalized for the occurrence of change. Components implementing variations, even when reused in five or more products, continue to evolve fairly rapidly. Although the number of failures in some variation components decreases as the product line matures, the percentage of severe failures in those components holds steady or even increases. Heavily reused variation components have a very low percentage of new files, much lower than either common components or lightly reused variation components. Contrary to our expectations, the number of failures in variation components shows no uniform pattern of decrease over time even if reused in multiple products.

The study reveals no simple answer to the question of whether reliability improves as a product line matures, but does suggest that on-going change may stay higher than commonly supposed, contributing to failure rates that persist in a lower-than-expected reliability range over time. The results of the current study point up the need for more detailed investigation of the relationship between failure and change in both open-source and proprietary software product lines.

10. ACKNOWLEDGMENTS

This work was supported by National Science Foundation grants 0916275 and 0916284 with funds from the American Recovery and Reinvestment Act of 2009.

REFERENCES

[1] Cvschangelogbuilder, tool for generating cvs log reports. `http://cvschangelogb.sourceforge.net/`.

[2] Eclipse bugzilla wiki homepage. `https://bugs.eclipse.org/bugs/`.

[3] Eclipse bugzilla wiki homepage. http://wiki.eclipse.org/Eclipse/Bug_Tracking.

[4] Software engineering institute, software product lines. http://www.sei.cmu.edu/productlines/.

[5] Software product line hall of fame. http://www.splc.net/fame.html.

[6] A. Avizienis, J. claude Laprie, and B. Randell. Fundamental concepts of dependability, 2001.

[7] J. A. Børretzen and R. Conradi. Results and experiences from an empirical study of fault reports in industrial projects. In *PROFES 2006. LNCS*, pages 389–394. Springer, 2006.

[8] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346–7354, 2009.

[9] G. Chastek, J. McGregor, and L. Northrop. Observations from viewing eclipse as a product line. In *Proceedings on the Third International Workshop on Open Source Software and Product Lines*, pages 1–6, 2007.

[10] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. on Software Engineering*, 26:797–814, 2000.

[11] H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[12] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE'10, pages 495–504, New York, NY, USA, 2010. ACM.

[13] M. Hamill and K. Goševa-Popstojanova. Common trends in software fault and failure data. *IEEE Trans. Softw. Eng.*, 35:484–496, July 2009.

[14] Y. Jiang, B. Cukic, and T. Menzies. Can data transformation help in the detection of fault-prone modules? In *Proc. of the 2008 workshop on Defects in large software systems*, DEFECTS '08, pages 16–20, New York, NY, USA, 2008. ACM.

[15] R. R. Lutz and I. C. Mikulski. Empirical analysis of safety-critical anomalies during operations. *IEEE Transactions on Software Engineering*, 30:172–180, 2004.

[16] R. R. Lutz, D. M. Weiss, S. Krishnan, and J. Yang. Software product line engineering for long-lived, sustainable systems. In J. Bosch and J. Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 430–434. Springer, 2010.

[17] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engg.*, 17:375–407, December 2010.

[18] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 263–272. ACM Press, 2000.

[19] P. Mohagheghi and R. Conradi. An empirical investigation of software reuse benefits in a large telecom product. *ACM Transactions on Software Engineering and Methodology*, 17:13:1–13:31, June 2008.

[20] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[21] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006.

[22] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *ISSRE*, pages 309–318, 2010.

[23] J. W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, 30:246–256, 2004.

[24] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[25] C. Rahmani, A. Azadmanesh, and L. Najjar. A comparative analysis of open source software reliability. *Journal of Software*, 5:1384–1394, December 2010.

[26] D. M. Weiss and C. T. R. Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[27] T. Zimmermann, N. Nagappan, and A. Zeller. *Predicting Bugs from History*, chapter Predicting Bugs from History, pages 69–88. Springer, February 2008.

[28] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, May 2007.