Predicting Component Failures at Design Time

Adrian Schröter
Saarland University
Saarbrücken, Germany
adrian@st.cs.uni-sb.de

Thomas Zimmermann
Saarland University
Saarbrücken, Germany
tz@acm.org

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@acm.org

ABSTRACT

How do design decisions impact the quality of the resulting software? In an empirical study of 52 ECLIPSE plug-ins, we found that the software design as well as past failure history, can be used to build models which accurately predict failure-prone components in new programs. Our prediction only requires usage relationships between components, which are typically defined in the design phase; thus, designers can easily explore and assess design alternatives in terms of predicted quality. In the ECLIPSE study, 90% of the 5% most failure-prone components, as predicted by our model from design data, turned out to actually produce failures later; a random guess would have predicted only 33%.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control*; D.2.8 [**Software Engineering**]: Metrics—*Complexity measures, Process metrics, Product metrics*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*

General Terms

Management, Measurement, Reliability

1. INTRODUCTION

During the design phase of new software, designers must make several decisions that will impact the cost and quality of the resulting software. Such decisions are typically guided by *experience*—that is, the accumulated knowledge of which designs worked and which designs did not. Think of a designer who has to choose between using graphics library A and graphics library B. Her choice will be influenced by previous experience with these libraries—her own experience as well as the experience reported from others. If she knows, for instance, that some other project has gotten into trouble because it was using A, she may opt for using B instead.

In this work, we aim to facilitate such decisions by making accurate *predictions* how failure-prone a component will be—by learning from history which design decisions correlated with failures in the past. More precisely, we focus on *usage relationships:* Does using the library A increase or decrease the risk of failure? It turns out that this usage information alone suffices to predict the failure-proneness of individual components. With such a prediction, our software designer can explore and assess the design alternatives and check which one has the lowest risk. She can also use the prediction to identify the components that are most likely to fail afterwards, and assign appropriate quality assurance efforts—and all this at design time, when decisions matter most.

Why does usage influence the risk of failure? As an example, consider two plug-ins from the ECLIPSE development framework. The package `org.eclipse.jdt.core.compiler` provides an interface to the JAVA compiler, such as invoking it or accessing compilation results. On the other hand, the `org.eclipse.ui` package gives access to the ECLIPSE user interface, providing GUI elements for user interaction.

For the average programmer, dealing with user interfaces is an everyday's job—in contrast to interacting with a compiler package, especially for an incremental compiler as used in ECLIPSE. Therefore, we would assume that code which uses the `compiler` package is more error-prone than code that uses the `ui` package. But can we actually quantify these differences? To this end, we have mined the ECLIPSE bug and version archives for usage data (which component uses which other components?) as well as for failure rates (which component had how many failures in the past?). Applied to the `compiler` and `ui` packages above, we found that 71% of the components using the `compiler` package needed to be fixed due to a failure after release. However, only 14% of the components using `ui` had to be fixed. Hence, we can confirm: compiler-related code is more failure-prone than GUI-related code.

In ECLIPSE, using the compiler and the GUI are just two extremes of how the usage of individual components impacts later failures. In this work, we therefore investigate whether *combinations* of usages make good failure predictors: If a component uses, say, the GUI and version control, but not the compiler, this specific *usage pattern* may indeed correlate with failures—and this is what we want to mine, as a model that makes accurate failure predictions.

In the remainder of this work, we start with an overview of related work (Section 2), followed by our research hypotheses (Section 3) Next, we describe how to collect usage and failure data from version archives (Section 4). From this data, we can build predictive models (Section 5), which we evaluate for 52 individual plug-ins in the ECLIPSE framework (Section 6). After a brief discussion of our results (Section 7), we close with conclusion and future work (Section 8).

2. STATE OF THE ART

Predicting which components are more failure-prone than others has been addressed by a number of researchers in the past. This work, discussed below, uses either complexity metrics or historical data to predict failures. In this paper, we go beyond this state of the art by showing that design data such as import relationships can predict failures.

2.1 Complexity Metrics

Typically, research on defect-proneness defines metrics to capture the software complexity and builds models that relate these metrics to failure-proneness [6]. Basili et al. [2] were among the first to validate that OO metrics are useful for predicting defect density. Subramanyam and Krishnan [23] presented a survey on eight more empirical studies, all showing that OO metrics are significantly associated with defects. Our models predict *post-release failures* since only such failures matter for the users of a program. Only few studies addressed post-release failures: Binkley and Schach [3] showed that their coupling dependency metric outperforms several other metrics; Ohlsson and Alberg [19] investigated a number of metrics to predict modules that fail during test or operation.

The work closest to ours in spirit is the MetriZone project at Microsoft Research. The goal of MetriZone is to make early estimated of software quality to predict post-releasure failures. Nagappan and Ball [17] showed that *relative code churn* predicts the software defect density; (absolute) code churn is the number of lines added or deleted between version. Additionally, Nagappan et al. [18] carried out the largest study of commercial software software so far. Within five Microsoft projects, they identified metrics that predict post-release failures and reported how to systematically build predictors for post-release failures from history.

2.2 Historical Data

In recent years, researcher exploited software repositories such as version archives and bug databases [20, 15, 17] for defect prediction. Our approach relies on the idea that one can map *problems* (in the bug database) to *fixes* (in the version archive) and thus to the location that caused the problem [22, 9, 10, 5, 4].

Several researchers used historical data without taking bug databases into account. Khoshgoftaar et al. [13] classified modules as defect-prone where the number of lines of code added or deleted exceeded a threshold. Graves et al. [11] used the sum of contributions to a module in its history to predict defect density. Ostrand et al. [20] used historical data from up to 17 releases to predict the files with the highest defect density in the next release. Hudepohl et al. [12] predicted whether a module would be defect-prone by combining metrics and historical data. From several software metrics, Denaro et al. [7] learned logistic regression models for Apache 1.3 and verified them against Apache 2.0.

3. HYPOTHESES

Our work is inspired by a simple observation: *Some problem domains are more failure-prone than others.* For instance, we found working on compiler internals to be much more difficult and error-prone than building user interfaces—regardless of developer, programming language, or the complexity of the resulting code. While we cannot say what it is that makes one problem domain more failure-prone than another one, we can at least attempt to *measure and predict failure proneness given a particular problem domain.*

Class C	N^C_{all}	$N^C_{\text{✘}}$	$p(\text{✘}\|C)$
org.eclipse.jdt.internal.compiler.util.CharOperation	111	87	0.7837
org.eclipse.team.internal.ccvs.core.Policy	48	34	0.7083
org.eclipse.core.resources.IWorkspaceRoot	50	34	0.6800
org.eclipse.jdt.core.IClasspathEntry	56	38	0.6785
org.eclipse.team.internal.ccvs.core.CVSProviderPlugin	60	40	0.6666
org.eclipse.jdt.core.Signature	73	48	0.6575
org.eclipse.jdt.internal.corext.util.JavaModelUtil	92	60	0.6521
org.eclipse.team.internal.ccvs.core.ICVSResource	63	41	0.6507
org.eclipse.swt.events.ModifyEvent	40	26	0.6500
org.eclipse.jdt.core.IPackageFragmentRoot	102	65	0.6372
org.eclipse.swt.events.ModifyListener	41	26	0.6341
org.eclipse.jface.text.ITextSelection	55	34	0.6181
...			
org.eclipse.swt.custom.BusyIndicator	101	18	0.1783
org.eclipse.swt.graphics.Point	122	21	0.1722
org.eclipse.core.runtime.Platform	103	17	0.1651
org.eclipse.jface.resource.ImageDescriptor	260	42	0.1616
org.eclipse.ui.PlatformUI	120	19	0.1584
java.util.ResourceBundle	160	24	0.1500
org.eclipse.jface.action.Action	136	16	0.1177

Table 1: Good and bad imports (classes) in ECLIPSE 2.0

Package C	N^C_{all}	$N^C_{\text{✘}}$	$p(\text{✘}\|C)$
org.eclipse.jdt.internal.compiler.lookup.*	197	170	0.8629
org.eclipse.jdt.internal.compiler.*	138	119	0.8623
org.eclipse.jdt.internal.compiler.ast.*	132	111	0.8409
org.eclipse.jdt.internal.compiler.util.*	148	121	0.8175
org.eclipse.jdt.internal.ui.preferences.*	63	48	0.7619
org.eclipse.jdt.core.compiler.*	106	76	0.7169
org.eclipse.jdt.internal.ui.actions.*	55	37	0.6727
org.eclipse.jdt.internal.ui.viewsupport.*	42	28	0.6666
org.eclipse.swt.internal.photon.*	50	33	0.6600
org.eclipse.jdt.internal.corext.util.*	117	70	0.5982
org.eclipse.swt.internal.motif.*	46	27	0.5869
org.eclipse.jdt.internal.ui.dialogs.*	60	34	0.5666
...			
org.eclipse.ui.model.*	128	23	0.1797
org.eclipse.swt.custom.*	233	41	0.1760
org.eclipse.pde.internal.ui.*	211	35	0.1659
org.eclipse.jface.resource.*	387	64	0.1654
org.eclipse.pde.core.*	112	18	0.1608
org.eclipse.jface.wizard.*	230	36	0.1566
org.eclipse.ui.*	948	141	0.1488

Table 2: Good and bad imports (packages) in ECLIPSE 2.0

How can one identify problem domains? We assume that the domain is implicitly described by the components that are used. When building an ECLIPSE plug-in that works on JAVA files, one has to import JDT classes; if the plug-in comes with a user interface, GUI classes are mandatory. In this paper, we will investigate whether the problem domain predicts future failures. Our research hypotheses H1 and H2 are as follows:

H1 *Importing certain components correlates with the failure-proneness of a file or package.*

H2 *Considering all imported components of a file or package, we can predict its future failure-proneness.*

In a first experiment, we tested hypothesis H1 and investigated whether the usage of certain components such as packages or classes correlates with failures. For every ECLIPSE component C, we computed the likelihood $p(\text{✘}|C)$ that a post-release failure occurred when this component was used in a file:

	N_{all}	$N_{✘}$	$p(✘)$
Eclipse 2.0			
– Files	6,751	1,649	0.244
– Packages	309	113	0.365
Eclipse 2.1			
– Files	7,909	1,021	0.129
– Packages	357	169	0.419

Table 3: Number of components in ECLIPSE

$$p(✘|C) = \frac{\text{number of files using } C \text{ with failures}}{\text{number of files using } C} = \frac{N_{✘}^{C}}{N_{\text{all}}^{C}}$$

For every likelihood, we tested with two *t-tests* ($\alpha = 0.05$) whether it is significantly higher (or lower) than the average likelihood $p(✘) = 1649/6751 = 0.244$ of a failure (see Table 3) and the likelihood $p(✘|\neg C)$ of a failure when *not* using component C.

Table 1 ranks the classes for which $p(✘|C)$ was tested as significant; Table 2 shows the same ranking for significant packages. On the top of the lists, we see components that come with a high risk when they are used. The class `CharOperation` was used 111 times, of which 87 times a post-release failure occurred. The packages `internal.compiler` are also risky: over 80% of the components using these packages failed after release.

There are also components whose usage implies low risk. Using the classes `ResourceBundle` and `Action` is rather safe compared to the average risk of a post-release failure $p(✘) = 0.244$. For packages, `ui` and `jface.wizard` can be used "safely". We also observed classes and packages that were exclusively used in files without post-release failures, i.e., $p(✘|c) = 0$; however, these components were not tested as significant by our t-tests.

An interesting observation in Table 2 is that many of the packages are *internal* (as indicated by the substring `internal` in the package name). Such packages are not part of the official ECLIPSE API and thus frequently changed; they should only be used by the ECLIPSE developer team [21]. Our results are consistent with this recommendation: using internal classes increases the likelihood of a post-release failure.

What does one do with such results? Of course, a component interacting with the user simply has to use one of the GUI-related packages; there is no design alternative available. However, suppose a programmer suggests a user interface design that relies on features of the internal `Motif` package. In the past, components that relied on `Motif` had a far higher risk of failure (58%) than components that relied on `ui` (14%)—and, when designing the component, this knowledge should be weighed against the potential benefit.

In ECLIPSE, the number of post-release failures is determined by the usage of specific components.

4. DATA COLLECTION

How do we know which components failed and which did not? For our analysis, we investigated two kinds of data: the *usage of import statements* within a single release and the *number of failures* for that release. We collect this data from *version archives* like CVS and *bug tracking systems* like BUGZILLA in three steps:

1. *Identify post-release failures.* From the bug tracking system, we get failures that were observed after a release (see Section 4.2).
2. *Classify files as failure-prone.* By mapping fixed failures to changes in the version archive, we can classify files as failure-prone or not (see Section 4.3).
3. *Determine imported classes for each file.* For each JAVA file, we collect the imported classes by a using simple syntactic analysis.

4.1 CVS and BUGZILLA

A CVS archive contains information about changes: Who changed what, when, why, and how? A *change* δ transforms a file from revision r_1 to a revision r_2 by inserting, deleting, or changing lines. Several file changes $\delta_1, \ldots, \delta_n$ form a *transaction* T if they were submitted to CVS by the same developer, at the same time, and with the same log message, i.e., they have been made with the same intention, e.g., to fix a bug or to introduce a new feature. As CVS records only individual changes to files, we group these to transactions with a *sliding time window* approach [25]. A CVS archive also lacks information about the *purpose* of a change: Did it introduce a new feature or did it fix a bug? Although it is possible to identify such reasons solely with log messages [14], we combine both CVS and BUGZILLA for this step because we need information from BUGZILLA for assigning bugs to releases.

A BUGZILLA database collects bug reports that are submitted by a *reporter* with a *short description*, *summary*, and *version* in which the bug was observed. After a bug has been confirmed, it is *assigned* to a developer who is responsible to fix the bug and finally commits her changes to the version control archive. BUGZILLA also captures the *status* of a bug, e.g., `UNCONFIRMED`, `NEW`, `ASSIGNED`, `RESOLVED`, or `CLOSED` and the *resolution*, e.g., `FIXED`, `DUPLICATE`, or `INVALID`. Additionally there is information about the *severity* of a bug that ranges from blocker ("application unusable") to trivial ("minor cosmetic issue"). Details on the life-cycle of a bug can be found in the BUGZILLA documentation [24, Sections 6.3 and 6.4].

For our analysis, we mirror both CVS and BUGZILLA in a local database. Our mirroring techniques for CVS are described in [25]; for BUGZILLA, we use its XML export feature.

4.2 Counting Post-Release Failures

We distinguish two different kinds of failures: *pre-release failures* are observed during development of a program, while *post-release failures* are observed after the program has been released and shipped to its customers. In our study, we focus on post-release failures, since these are the ones that matter for the users of a program.

Failures are typically documented in *bug reports.* For our study, we investigated all bug reports that were

- neither duplicates (=resolved as `DUPLICATE`) nor bugs of low severity such as `ENHANCEMENT`, `MINOR` or `TRIVIAL`,
- submitted within six months after a release, and
- fixed at some time after they were reported.

For mapping bug reports to releases, we used the *version* field of the BUGZILLA database. Since the values of this field may change during the life-cycle of a bug, we decided to take only the first reported version into account because this is the version in which the bug was introduced.

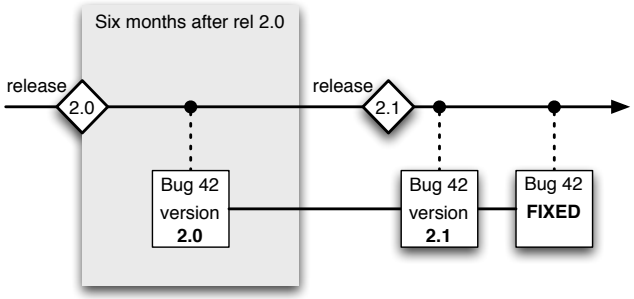

Figure 1: Locating post-release failures

Version		Total	Mapped	Ratio
1.0	2001-11-07	347	195	56.2%
2.0	2002-06-27	**1,437**	**1,018**	**70.8%**
2.0.1	2002-08-29	182	118	64.8%
2.0.2	2002-11-07	104	70	67.3%
2.1	2003-03-27	**1,051**	**714**	**67.9%**
2.1.1	2003-06-27	164	105	64.0%
2.1.2	2003-11-03	74	48	64.9%
2.1.3	2004-03-10	19	11	57.9%
3.0	2004-06-25	1,649	1,243	75.4%

Table 4: Number of post-release failures in ECLIPSE

Figure 1 illustrates post-release failures. Bug 42 is reported within six months after release 2.0 and marked with that version. However, it is not fixed within these six months, and still exists in release 2.1. After release 2.1, the version field is updated to 2.1 and finally the bug is fixed. Bug 42 counts as a post-release failure for release 2.0, but not as a post-release failure for 2.1 because it was already observed before that release.

Table 4 shows the releases of ECLIPSE in the first column and the number of identified post-release failures in the second column. We observed most post-release failures in major ECLIPSE releases such as 2.0, 2.1, and 3.0. In contrast, maintenance release such as 2.0.1 or 2.0.2 introduce less failures.

4.3 Mapping Post-Release Failures to Files

Let us now show how to map bug reports to corrections in the source code—and thus the location of the original bug. For this mapping, we use a bidirectional approach:

From code changes to bug reports. In version archives every change is annotated with a message that describes the reason for that change. We search this message for references to bug reports such as "Fixed 42233" or "bug #23444". Basically every number is a potential reference to a bug report, however such references have a low trust at first. We increase the trust level when the message contains keywords such as "fixed" or "bug" or matches patterns like "# and a number".

From bug reports to code changes. Not all bugs are referenced in log messages. We map such bugs to the change that (a) is closest to the time when a bug report was resolved, (b) but less than twelve hours away, and (c) was made by the developer to whom the bug was assigned. For the second condition, we manually created a mapping between the BUGZILLA and CVS user accounts.

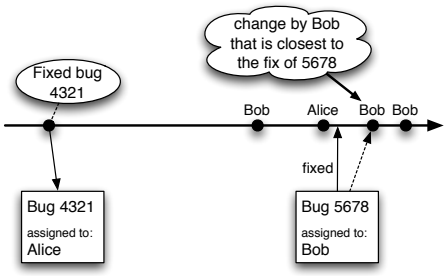

Figure 2: Mapping bug reports to changes

Figure 2 shows two examples: The change with the message "Fixed bug 4321" can be mapped directly to bug report 4321. For bug report 5678 that is assigned to Bob, there is no message; to identify the change that fixed 5678, we take the change by Bob that is closest to the time when 5678 was reported to be fixed, but not more than 12 hours apart.

Our approach for mapping code changes to bug reports is described in detail by Śliwerski et al. [22] and is similar to the approaches used by Fischer et al. [9, 10] and by Čubranić et al. [5]. The idea of mapping the other direction, from bug reports to changes, was proposed and implemented by Čubranić [4].

By mapping from code changes to bug reports, we identified the corrections of approximately 50% of all post-release-failures; the other direction, from bug reports to code changes, identified an additional 20%. Table 4 shows in the third column the number of post-release failures that we could map for each release. For major releases, we mapped approximately 70% of post-release failures.

We classify files as *failure-prone* by using the changes that corrected post-release failures: a file is failure-prone in a release R if it was changed in order to fix a post-release failure for R. Additionally, we can rank failure-prone files by the number of post-release failures that are assigned to it.

4.4 Assigning Imports to Files

In the previous section, we counted for every file the number of post-release failures. Next, we run a simple syntactic analysis in order to collect for every JAVA file the imported classes or packages.

As a result, we get for every JAVA file:

- the number $N_{\times}$ of post-release failures, and
- the set J of used resources, such as classes or packages.

In the remainder of the paper, we will build models that predict for JAVA files the number $N_{\times}$ of post-release failures from the set J of imported classes and packages.

5. PREDICTING FAILURES

Let us now describe how to build prediction models from the import relations which we collected in the previous section. Our models work on two different granularity levels that are described first. Next, we discuss the difference between classification and ranking and finally, we briefly describe the models that we used for our case study.

5.1 Granularity Levels

For our models, we can use two different granularity levels:

File/class level. For input features we directly used the classes that a file imported. Since in JAVA, import statements are per file, we can only predict failure-proneness for files and not for classes.

Package level. Packages describe a coarser granularity level than files or classes. For input features we mapped the imported classes to the surrounding package, e.g., the class `java.util.Vector` to the package `java.util`. When predicting for packages, we mapped post-release failures to directories instead of files, and counted them by package. This way we guaranteed that no post-release failure was counted twice.

Intuitively, predicting for a coarse granularity is easier while using fine-grained input features yields better results.

Note that we distinguish between the granularity for input features and for prediction; for instance we can predict the failure-proneness of files using their imported packages, or the failure-proneness of packages using imported classes.

In the case of the 52 ECLIPSE plug-ins, the granularity of the input features and for the prediction can be described in the same way. We start with the finest levels and will continue with the more coarse levels.

5.2 Classification vs. Ranking

With our models, we address two problems:

Classification. Can we tell whether a component will be failure-prone or not based on its design data? This information helps to mark risky parts of a software design.

Ranking. Can we tell which components will have the most failures? This information identifies the parts of a software design that require most attention when being implemented and tested.

Since classification yields strictly less information from the same training data than ranking, it is less error-prone, i.e., the likelihood of a wrong prediction is lower for classification than for ranking.

Typically, a software designer is interested in the components that are most failure-prone, and not in a complete ranking of all components. We take this into account by additionally assessing the top 5, 10, 15, and 20 percent of our rankings.

5.3 Prediction Models

We used four different prediction models for our case study.

Linear regression is a method of estimating the value of a dependent variable, which depends on the input features. We use the number of failures in a component as dependent variable and the imported resources used from this component as input features.

Ridge regression is a slight generalization of linear regression. The difference to linear regression is that we are able to penalize the log-likelihood.

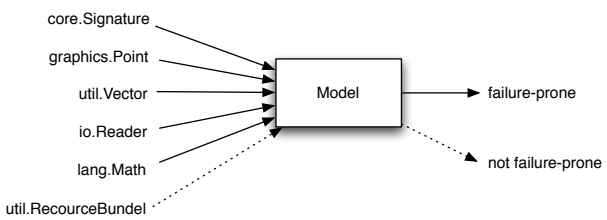

Figure 3: Classification of a component

Regression trees are mappings from observations about the dependent variable to conclusion about its value. Each level of such a tree corresponds to an input feature and an edge to a child represents its possible value. The leaves of the tree describe the predicted values.

Support vector machines create a hyper-plane separating the data into two classes with the maximum margin, i.e., distance between the hyper-plane and the closest point from both sides are maximized. We used support vector machines with a radial basis function as kernel which smoothens our data. This helps to reduce noise that was created during data collection.

All these models are regression models that can be used directly for ranking; for classification they are combined with a suitable decision boundary.

Figure 3 illustrates how our approach classifies files as failure-prone or not using classes as input features. For every possible import in ECLIPSE, we introduced a dummy variable which indicates whether a component uses a certain import. The example in Figure 3 imports `core.Signature`, `graphics.Point`, `util.Vector`, `io.Reader`, and `lang.Math` and is classified as failure-prone. If there is another component that imports the same classes but additionally `util.ResourceBundle`, our model may classify it as not failure-prone.

For ranking we predict for every component the number of failures and sort according to this. In Figure 4, we get a ranking `C.java`, `B.java`, `A.java`, and `D.java`. Next we compare this predicted ranking with the observed ranking using the Spearman rank correlation coefficient.

6. RESULTS: 52 ECLIPSE PLUG-INS

Let us now discuss the obtained results. After explaining how to assess the presented tables, we continue with a discussion of how the different levels of granularity impact the predictions. Subsequently, we take a look at the classification and ranking approaches, followed by a comparison of the different models. We close this section with a discussion of *robustness*, i.e., how the predictive power changes with the version for which we predict.

6.1 Assessing Results

We used different measurements to asses the quality of our prediction models.

Precision measures from the set of predicted failure-prone components the percentage of correctly classified components.

$$\text{precision} = \frac{\text{correct predicted failures}}{\text{all predicted failures}}$$

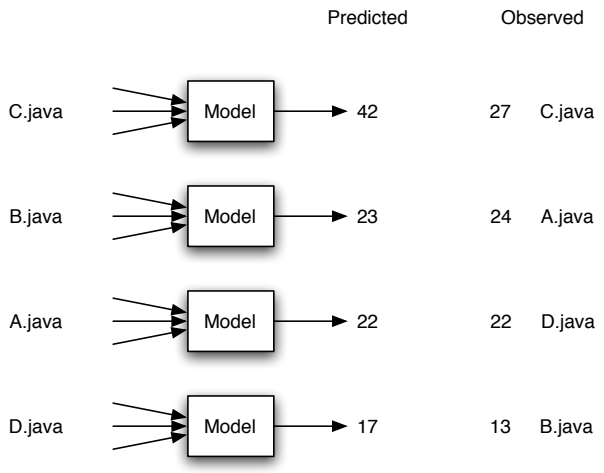

Figure 4: Comparison of predicted and observed ranking

In Table 6(d) we obtained a *precision* of 0.6671 for the test in version 2.0. That is, two of three components predicted as failure-prone turned out to produce failures. Had we used a random guess instead, the probability of a failure occurrence would have been $p(\text{✘}) = 113/309 = 0.365$ (see Table 3), and we would have obtained a precision of only 0.365.

Recall measures from the set of predicted failure-prone components the percentage of failure-prone components classified as such.

$$\text{recall} = \frac{\text{correct predicted failures}}{\text{all failures}}$$

As an example, the *recall* of the test in version 2.0 shown in Table 6(d) indicates that over two third of the failure-prone components are actually identified as failure-prone. Again, a random guess would have had a probability of 0.365 and therefore yielded a recall of only 0.365.

Spearman coefficient measures the correlation between the predicted and observed ranking. Hence, it indicates the quality of our overall ranking. High correlations are indicated by values close to 1 and -1, i.e., values of 1 indicate a identical ranking and values of -1 indicate an opposite ranking. Values close to 0 indicate no correlation. For example, the training result in Table 5(d) indicates an almost identical ranking with a value of 0.7641.

Top. We applied precision and spearman coefficents to the upper x% of our rankings. We did not compute the recall for these parts because the recall values are limited by the x percent to a maximum of $x/(100 \cdot p(\text{✘}))$. Additionally, the upper x% are a special case for which we can compute the recall from the precision:

$$\text{recall for top x\%} = \frac{\text{precision} \cdot x}{100 \cdot p(\text{✘})}$$

For our experiments, we used *random splits*. A random split is a technique to divide our data into training and test sets. To apply such a random split for our 52 ECLIPSE plug-ins, we randomly chose one third of all plug-ins contained in version 2.0 as our training set, the other two third as test set for 2.0 and the complement in 2.1 as test set for 2.1 (see Figure 5). We generated 40 independent training and test sets, trained and tested 40 models of each kind. To obtain our final results, we averaged over the 40 predictions made by the trained models.

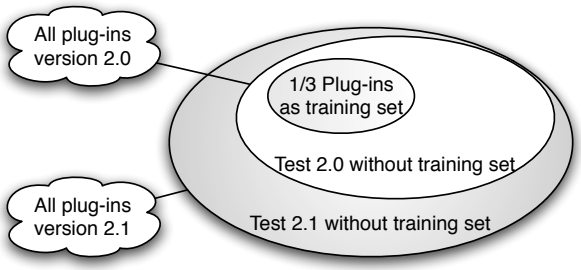

Figure 5: Random split on plug-in level

Note that when a model returned less then x% failure-prone components during the investigation of the top x%, we filled it up with false positives. As a result the precision was lowered while the recall remained unchanged. Thus top x% means "Return x% of all components that are most likely failure-prone". A consequence of this is that average top values may have lower precision as the overall prediction.

6.2 Packages vs. Files

As we see in Table 5 and 6, our prediction of failure-proneness for packages yields better results than predicting failure-proneness for files. For example, take the test in version 2.0 of Table 6(b) with a recall of about 0.1 and Table 6(d) with a recall of about 0.7.

Since packages represent a more coarse level of granularity, they provide some kind of *smoothness.* This smoothness is expressed through lowering the number of outliers we have to deal with predictions on file level. Usually more information on input features means better predictions. Since on package level we not only have binary input features, we are able to distinguish the packages on how intense they use a resources other than on file level where we only know the resources they use. Therefore, the prediction for packages with class imports as input feature yield better results than predicting with package imports as input features.

Note that every newly introduced feature increases the risk of making the data highly dependent on single features, rather than on combination of features. This leads on the one hand to good training results like a precision of 0.8331 and a recall of 0.8724 as shown in Table 5(b). On the other hand it results in bad test results like the test in version 2.0 shown in Table 5(b).

Prediction on package level yields the best results: Of the top 5% packages predicted as failure-prone, 90% are correctly classified.

6.3 Ranking vs. Classification

To obtain our results, we use several regression methods which provides us with rankings. We transform these rankings into a classification by applying a *decision boundary*. We label failure-prone components with the number of failures and components that did not fail with -1; thus we choose our decision boundary to be 0.

	Precision	Recall	Spear. Coef.
training	0.5246	0.6866	0.2937
test in v2.0	0.3800	0.2894	0.2738
→5%	0.4369		0.0717
→10%	0.4449		0.0503
→15%	0.4093		0.0320
→20%	0.3886		0.0734
test in v2.1	0.2219	0.3640	0.3201
→5%	0.2705		0.1136
→10%	0.2546		0.1527
→15%	0.2433		0.1539
→20%	0.2396		0.1486

(a) Predicting files with packages

	Precision	Recall	Spear. Coef.
training	0.8331	0.8724	0.1332
test in v2.0	0.2614	0.2958	-0.0313
→5%	0.3103		-0.1346
→10%	0.3092		-0.1257
→15%	0.2979		-0.1026
→20%	0.2826		-0.0883
test in v2.1	0.1967	0.4225	0.0979
→5%	0.2767		-0.0878
→10%	0.2543		-0.0586
→15%	0.2374		-0.0329
→20%	0.2274		-0.0276

(b) Predicting files with classes

	Precision	Recall	Spear. Coef.
training	0.7274	0.9973	0.1152
test in v2.0	0.5874	0.7955	-0.2622
→5%	0.4384		-0.0474
→10%	0.4522		-0.1056
→15%	0.4703		-0.1323
→20%	0.5054		-0.1250
test in v2.1	0.4511	0.7878	-0.0367
→5%	0.5214		0.0633
→10%	0.4704		0.0319
→15%	0.4711		-0.0310
→20%	0.4866		-0.0570

(c) Predicting packages with packages

	Precision	Recall	Spear. Coef.
training	1.0000	0.9996	0.7641
test in v2.0	0.5952	0.4512	0.0204
→5%	0.5747		0.0739
→10%	0.5776		-0.0086
→15%	0.5717		0.0170
→20%	0.5869		0.0097
test in v2.1	0.4754	0.4643	0.1224
→5%	0.5362		-0.0065
→10%	0.5466		-0.0568
→15%	0.5232		-0.0677
→20%	0.5168		-0.0377

(d) Predicting packages with classes

Table 5: Results from linear regression

	Precision	Recall	Spear. Coef.
training	0.6974	0.3183	0.3010
test in v2.0	0.2876	0.0387	0.2390
→5%	0.1063		0.1167
→10%	0.0771		0.1188
→15%	0.0535		0.1007
→20%	0.0411		0.1566
test in v2.1	0.2757	0.0969	0.3104
→5%	0.1616		0.0772
→10%	0.1043		0.1406
→15%	0.0754		0.1879
→20%	0.0573		0.1973

(a) Predicting files with packages

	Precision	Recall	Spear. Coef.
training	0.8944	0.2664	0.3366
test in v2.0	0.5523	0.0923	0.2253
→5%	0.2586		0.0566
→10%	0.1730		0.1001
→15%	0.1181		0.1123
→20%	0.0891		0.1667
test in v2.1	0.4646	0.1697	0.3181
→5%	0.2654		0.1866
→10%	0.1973		0.1969
→15%	0.1471		0.2337
→20%	0.1119		0.2836

(b) Predicting files with classes

	Precision	Recall	Spear. Coef.
training	0.7442	0.8765	0.0770
test in v2.0	0.5654	0.7611	-0.1263
→5%	0.6254		0.0091
→10%	0.5804		-0.0772
→15%	0.5824		-0.0886
→20%	0.5808		-0.0541
test in v2.1	0.4768	0.7714	0.0822
→5%	0.7329		0.3750
→10%	0.6510		0.1973
→15%	0.6174		0.0729
→20%	0.5981		-0.0076

(c) Predicting packages with packages

	Precision	Recall	Spear. Coef.
training	0.8770	0.8933	0.5961
test in v2.0	0.6671	0.6940	0.3002
→5%	0.7861		0.1369
→10%	0.7875		0.2032
→15%	0.7957		0.2648
→20%	0.8000		0.3190
test in v2.1	0.5917	0.7205	0.2842
→5%	0.8958		0.3416
→10%	0.8399		0.3702
→15%	0.7784		0.3675
→20%	0.7668		0.3615

(d) Predicting packages with classes

Table 6: Results from support vector machine

	Precision	Recall	Spear. Coef.
training	1	0.5914	0.7248
test in v2.0	0.7354	0.3107	0.1908
→5%	0.6958		-0.2812
→10%	0.7191		-0.1741
→15%	0.7419		-0.0810
→20%	0.7498		-0.3606
test in v2.1	0.6865	0.3864	0.2360
→5%	0.8455		0.1034
→10%	0.7757		0.1603
→15%	0.7218		0.1763
→20%	0.7140		0.1949

Table 7: Results of ridge regression for predicting packages with classes.

	Precision	Recall	Spear. Coef.
training	0.6090	0.9121	0.2550
test in v2.0	0.5862	0.7945	0.0774
→5%	0.8713		-0.0130
→10%	0.8014		-0.0610
→15%	0.7584		-0.0603
→20%	0.7122		-0.0518
test in v2.1	0.4830	0.8182	0.2359
→5%	0.7174		0.0674
→10%	0.6777		0.0041
→15%	0.6414		0.0468
→20%	0.6193		0.0893

Table 8: Results of regression trees for predicting packages with classes.

The low values of the Spearman rank correlation coefficients in Tables 5 and 6 indicates that our predicted rankings do not correlate with the observed rankings.

However, for almost all models the precision values for the top x% are higher than the overall values, e.g., in Table 6(d), the precision for the top 5% of version 2.1 is substantially higher than the overall precision (90% vs. 60%). This means that our classification works best for the parts that are highly ranked.

Classification works best for components that are ranked as most failure-prone.

6.4 Comparing Models

In this section, we compare the predictive power of the models.

Linear regression. Although *linear regression* models yield the best training results, the test results are not as good as expected. As an example, take a look at Table 5(b). Since *linear regression* is more sensitive to outliers, the recall is higher than the corresponding *support vector machine*, but in return the precision is substantially lower.

Ridge regression. Since *ridge regression* is a slight generalization of *linear regression* allowing us to penalize outliers, the results become more accurate. For instance the precision values presented in Table 7 are substantially higher than in Table 5(d), at the cost of a slightly decreased recall.

Regression tree. Table 8 shows that *regression tree* models achieve results similar to the *support vector machine* in Table 6(d). Compared to the *linear regression* the *regression tree* is a more accurate prediction model, e.g., the precision observed in the top of Table 8 is significantly larger than in the top of Table 5(d).

Support vector machine. The *support vector machine* achieved the best results for predicting the failure-proneness of packages with classes as input features. Since these results are better than any other results obtained either with packages or classes as input feature, we consider the *support vector machine* as the most effective model.

Note that we did not discuss the *ridge regression* and the *partition tree* models in detail because either the results or the idea of the model were similar to one of the discussed models. Since *ridge regression* is similar to *linear regression* we decided to present the *linear regression* models because the results are easier to interpret. Whereas, we skipped the detailed discussion of the *partition tree* models due to the similar results obtained by the *support vector machines*.

Of the four models researched, the support vector machine yields the best predictive power.

6.5 Applying Models across Versions

The idea behind the predictive models is to obtain models which we can use for predictions in future versions, e.g., we want to have a model which predicts failure-prone components for version 2.0 using the data of version 1.0.

In the case of the 52 ECLIPSE plug-ins the obtained results show a similar behavior between the versions 2.0 and 2.1 of ECLIPSE. This behavior results in a robustness of the models over time, i.e., the results from version 2.0 and 2.1 are similar with respect to classification.

Take for example Table 6(d), the recall and precision obtained from testing in version 2.0 are about 0.66 and 0.69. Comparing them to the results of the tests in version 2.1, the precision decreased by about 0.06 and the recall increased by about 0.03. Since these variance values present a precision recall trade-off, the results can be considered stable. Hence, we can use the models for later versions without losing predictive power.

Models trained in one version can be used to predict failure-prone components in later versions.

7. DISCUSSION

To our knowledge this is the first work that predicts post-release failures from design data such as import relationships. In contrast to previous work, our approach does not use any metrics. Our models rely solely on *import relationships* between components, an aspect that has not been investigated for failure-proneness so far. In addition, our approach takes advantage of bug databases to identify post-release failures, and only severe ones. However, we do not use historical data as input features which is another main difference to previous research conducted in this field.

7.1 Results

Our results support hypothesis H2 that one can predict future post-release failures by using imported components of a file or package.

File Level. The results on file level are not as good as on package level, but still substantially better than random guesses. This means that the problem domain of single files has an impact on post-release failures.

Package Level. Our precision and recall values show that the problem domain has a higher influence on the failure-proneness of packages than for files. This is not surprising as it gets more difficult to identify reliable predictors for fine-grained levels.

Although the results confirmed our hypotheses that problem domains correlate with and predict failure-proneness, one central question remains open: "What is it that make some domains more failure-prone?". This is the question on which our future work will concentrate.

7.2 Threats to Validity

We studied 52 plug-ins of the ECLIPSE development environment. Although the plug-ins themselves are very different and cover a wide spectrum of applications from compilers to user interfaces, we cannot claim that their version history would be *representative for all kinds of software projects.* Unfortunately, too few software projects combine their version archive with a well-kept bug database; ECLIPSE is one of them [1].

We assumed that fixes are in the same location as the corresponding defect. Although this is not always true, this assumption is frequently used in research [16, 8, 20, 18].

We approximated the design of a plug-in by its import relations at release time. These relations do not need to correspond to the actual initial design. We also used only static information about imports, thus neglecting dynamic binding. We believe that considering dynamic binding will improve the quality of our models.

We trained our models and examined their predictive power using data that was *computed by heuristics.* For instance, we could not map all post-release failures to source code and did not cross-check the fixes that were identified. Additionally, we missed failures that were not reported to bug databases. Eventually, usefulness for the programmer can only be determined by studies that include real developers, which we intend to accomplish in the future.

8. CONCLUSION AND FUTURE WORK

A component's likelihood to fail is significantly determined by the set of components that it uses. Why is this so? Our hypothesis is that the set of used components is *determined by the problem domain*—and some of these domains are harder to get right than others. If a component imports ECLIPSE internals, it is likely to work on the abstract syntax tree, which is failure-prone; a component importing GUI elements will interact with the user, which is easier to get right—at least, this is what the ECLIPSE failure history suggests.

Using our approach, managers and developers can leverage earlier failure history to predict future failure-prone components, and thus assign resources to those components which need it most. Since the set of used components is typically defined at design time, these decisions can be made very early in the software process.

Although our predictions are much better than random guesses, we still would like to increase their accuracy. Our future work will focus on the following topics:

More Design Features. Software design is characterized by far more than just import relationships. Features we would like to look into include *inheritance relations* (does sub-classing from a class C increase failure-proneness?), *part-of relations* (does including C as a part influence the likelihood of failure?), or general *design metrics* such as depth of inheritance or number of subclasses.

Indirect Assessments. Right now, when a component A imports a component B, we need to know the earlier failure history of other users of B to assess the failure-proneness of A. If our hypothesis that imports characterize the problem domain is true, though, it may be that the *likelihood of failure propagates along relationships.* In other words, if B *itself* is likely to fail, so may be A as an user of B—simply because their problem domains are related.

Tool Support. Right now, our approach is implemented as a series of scripts around the R statistics package. We would like to turn our approach into a tool that is immediately available to the software designer, allowing her to interactively explore design alternatives and assess the overall predicted quality.

At the dawn of the last century, the philosopher George Santayana remarked that those who do not learn from history would be condemned to repeat it. With our approach, we want to encourage developers to learn from failure history such that they get the chance to avoid these mistakes—and build better software in the future.

For ongoing information on the project, see

`http://www.st.cs.uni-sb.de/softevo/`

Acknowledgments. Our work on mining software repositories is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Thomas Zimmermann is additionally funded by the DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme". Special thanks go to Thomas Ball and Nachi Nagappan of Microsoft Research and Stephan Neuhaus and Rahul Premraj for their valuable comments on the approach as well as on earlier revisions of this paper.

9. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of eclipse Technology eXchange (eTX) Workshop at OOPSLA*, Oct. 2005.

[2] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[3] A. B. Binkley and S. R. Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the International Conference on Software Engineering*, pages 452–455, Apr. 1998.

[4] D. Čubranić. *Project History as a Group Memory: Learning From the Past*. PhD thesis, University of British Columbia, Canada, Dec. 2004.

[5] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005.

[6] G. Denaro, S. Morasca, and M. Pezzè. Deriving models of software fault-proneness. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pages 361–368, July 2002.

[7] G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Proceedings of the International Conference on Software Engineering (ICSE 2002)*, pages 241–251. ACM, May 2002.

[8] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Software Eng.*, 26(8):797–814, 2000.

[9] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, Nov. 2003. IEEE.

[10] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Trans. Software Eng.*, 26(7):653–661, 2000.

[12] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand. Emerald: Software metrics and models on the desktop. *IEEE Software*, 13(5):56–60, 1996.

[13] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, page 364, Washington, DC, USA, 1996. IEEE Computer Society.

[14] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proc. International Conference on Software Maintenance (ICSM 2000)*, pages 120–130, San Jose, California, USA, Oct. 2000. IEEE.

[15] A. Mockus, P. Zhang, and P. L. Li. Predictors of customer perceived software quality. In *Proceedings of the International Conference on Software Engineering (ICSE 2005)*, pages 225–233. ACM, May 2005.

[16] K.-H. Moller and D. Paulish. An empirical investigation of software fault distribution. In *Proc. IEEE First International Software Metrics Symposium*, pages 82–90, May 1993.

[17] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the International Conference on Software Engineering (ICSE 2005)*, pages 284–292. ACM, May 2005.

[18] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering (ICSE 2006)*. ACM, May 2006.

[19] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Software Eng.*, 22(12):886–894, 1996.

[20] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Software Eng.*, 31(4):340–355, 2005.

[21] J. Rivières. *How to use the Eclipse API*, May 2001. http://eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html.

[22] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? On Fridays. In *Proc. International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, U.S., May 2005.

[23] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Software Eng.*, 29(4):297–310, 2003.

[24] The Bugzilla Team. *The Bugzilla Guide - 2.18 Release*, Jan. 2005. http://www.bugzilla.org/docs/2.18/html/.

[25] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Edinburgh, Scotland, UK, May 2004.