Software Evolution

Bearbeitet von
Tom Mens, Serge Demeyer

1. Auflage 2008. Buch. xviii, 347 S. Hardcover
ISBN 978 3 540 76439 7
Format (B x L): 15,5 x 23,5 cm
Gewicht: 1530 g

4

Predicting Bugs from History

Thomas Zimmermann[1], Nachiappan Nagappan[2], and Andreas Zeller[1]

[1] University of Calgary, Alberta, Canada
[2] Microsoft Research, Redmond, Washington, USA
[3] Saarland University, Saarbrücken, Germany

Summary. Version and bug databases contain a wealth of information about software failures—how the failure occurred, who was affected, and how it was fixed. Such defect information can be automatically mined from software archives; and it frequently turns out that some modules are far more defect-prone than others. How do these differences come to be? We research how code properties like (a) code complexity, (b) the problem domain, (c) past history, or (d) process quality affect software defects, and how their correlation with defects in the past can be used to predict future software properties—where the defects are, how to fix them, as well as the associated cost.

4.1 Introduction

Suppose you are the manager of a large software project. Your product is almost ready to be released—where "almost ready" means that you suspect that a number of defects remain. To find these defects, you can spend some resources for quality assurance. Since these resources are limited, you want to spend them in the most effective way, getting the best quality and the lowest risk of failure. Therefore, you want to spend the most quality assurance resources on those modules that need it most—those modules which have the highest risk of producing failures.

Allocating quality assurance resources wisely is a risky task. If some non-defective module is tested for months and months, this is a waste of resources. If a defective module is not tested enough, a defect might escape into the field, causing a failure and potential subsequent damage. Therefore, identifying defect-prone modules is a crucial task for management.

During the lifetime of a project, developers remember failures as well as successes, and this experience tells them which modules in a system are most frequently associated with problems. A good manager will exploit this expertise and allocate resources appropriately. Unfortunately, human memory is limited, selective, and sometimes inaccurate. Therefore, it may be useful to complement it with findings from actual facts—facts on software defects as found after release.

In most quality-aware teams, accessing these facts requires no more than a simple database query. This is so because *bug databases* collect every problem reported

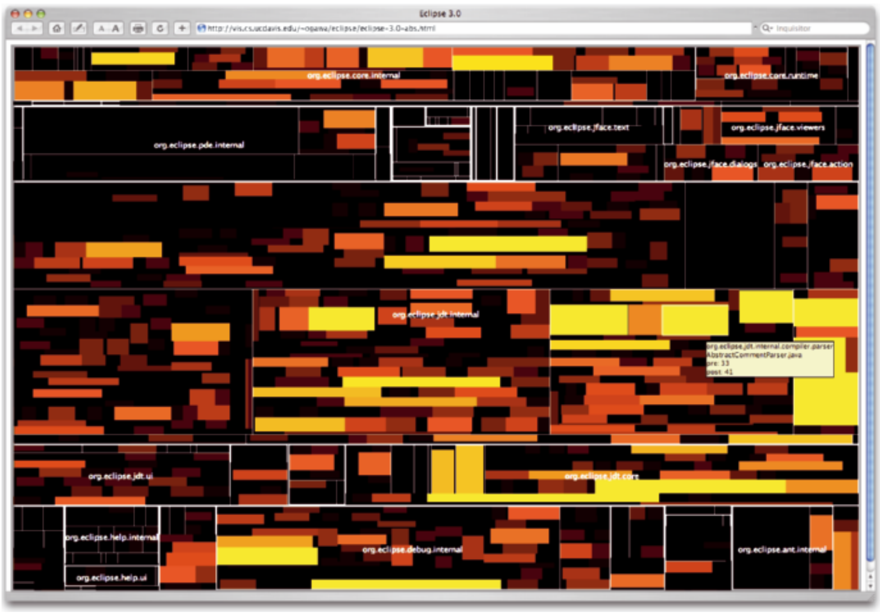

Fig. 4.1. Michael Ogawa's visualisation of the Eclipse Bug Data [401], inspired by Martin Wattenberg's "Map of the Market" [536]. Each rectangle stands for a package; the brighter the rectangle, the more post-release defects it has

about a software product. For a system that already has a large user community, bug databases are central to the development process: new bugs are entered into the system, unresolved issues are tracked, and tasks are assigned to individual developers. However, as a bug database grows, it can also be used to learn which modules are prone to defects and failures. This is so because as the problems are fixed, the fixes apply to individual modules—and therefore, one can actually compute how many defects (or reported failures) some module is associated with. (How to establish these mappings between bug reports, fixes, and locations is described in Chapter 3 of the present book.)

Figure 4.1 visualises defect data in the modules of the Eclipse programming environment. As the picture shows, the distribution of defects across modules is very uneven. For instance, compiler components in Eclipse have shown 4–5 times as many defects as user interface components [455]. Such an uneven distribution is not at all uncommon; Pareto's law [160, p. 132], for instance, already stated in 1975 that approximately 80 % of the defects are found in 20 % of the modules. Still, this uneven distribution calls for our first research question:

Why are some modules more defect-prone than others?

Answering this question helps in understanding the nature of defects—starting with the symptoms, and then following the cause-effect chain back to a root cause—and will help avoiding defects in the future.

Unfortunately, at the present time, we have no universal answer for this question. It appears that every project has its own individual factors that make specific modules prone to defects and others unlikely to fail. However, we can at least try to capture these *project-specific properties*—for instance, by examining and summarising the common symptoms of defect-prone modules. Coming back to our original setting, knowing these common properties will also help in allocating quality assurance resources. This will answer the second, more opportunistic, but no less important question:

Can we predict how defect-prone a component will be?

At first glance, this question may seem absurd. Did we not just discuss how to measure defect-proneness in the past? Unfortunately, the number of defects in the past may not be a good indicator of the future:

- Software *evolves,* with substantial refactorings and additions being made all the time. Over time, this invalidates earlier measurements about defects.
- The defects we measure from history can only be mapped to components because *they have been fixed.* Given that we consider defects that occur after release, by definition, any measurement of defect-prone components applies to an already obsolete revision.

For these reasons, we need to devise with predictive methods that can be applied to new as well as evolved components—predictions that rely on *invariants* in people, process, or structure that allow predicting defects although the code itself has changed. In the next section, we discuss how to find some of such invariants, and how they may impact the likelihood of defects.

4.2 What Makes a Module Defect-Prone?

If software needs to be fixed, it needs to be fixed because it does not satisfy a requirement. This means that between the initial requirement and the actual deployment of the software system, some mistake has been made. This mistake manifests itself as an error in some development artefact, be it requirements, specification, or a design document. Ultimately, it is the source code that matters most, since it is the realisation of the software system; errors in source code are called *defects*. If an error in some earlier stage does not become a defect, we are lucky; if it becomes a defect, it may cause a failure and needs to be fixed.

The defect likelihood of some module thus depends on its history—not only its code history, but the entire history of activities that led to its creation and maintenance. As the code evolves, this earlier history still stays unchanged, and the history may well apply to other modules as well. Therefore, when trying to predict the error-proneness of modules, we must look for *invariants* in their history, and how these invariants might manifest themselves in modules. In this chapter, we discuss the following invariants:

Complexity. In general, we assume that the likelihood of making mistakes in some artefact increases with

1. the number of details, and
2. the extent of how these details interact and interfere with each other.

This is generally known as *complexity,* and there are many ways complexity manifests itself in code. More specifically, *complexity metrics* attempt to measure the complexity of code, and therefore, it may be useful to check whether it is possible to predict defect-proneness based on complexity metrics. This is discussed in Section 4.3.

Problem domain. As stated initially, software fixes come to be because requirements are unsatisfied. This implies that the more likely it is to violate a requirement, the higher the chances of making a mistake. Of course, a large number of interfering requirements results in a higher problem complexity—and should thus manifest itself in complex code, as described above. However, we assume that specific *problem domains* imply their own set of requirements. Therefore, one should be able to predict defect-proneness based on the domain alone. How does one determine the problem domain? By examining the modules another module interacts with, as shown in Section 4.4.

Evolution. Requirements can be unsatisfied for a simple reason: They may be changing frequently. Modified requirements result in changed code—and therefore, *code that is frequently changed* indicates frequently changing requirements. The same applies for imprecise requirements or requirements that are not well understood, where trial-and-error approaches may also cause frequent fixes. Finally, since changes may introduce new defects [160], a high change rate implies a higher defect likelihood. Relying on the change rate to predict defects is discussed in Section 4.5.

Process. Every defect that escapes into the field implies a failure of quality assurance—the defect simply should have been caught during checking, testing, or reviewing. A good software process can compensate many of the risks described above; and therefore, the *quality of the development process* should also be considered when it comes to predicting defects. We discuss these open issues in Section 4.6.

The role of these invariants in software development has been analysed before, and a large body of knowledge is available that relates complexity, the problem domain, or evolution data to defects. What we see today, however, is the *automation* of these approaches. By having bug and change databases available for automated analysis, we can build tools that automatically relate defects to possible causes. Such tools allow for product- and project-specific approaches, which may be far more valuable than general (and therefore vague) recommendations found in software engineering textbooks. Our results, discussed in the following sections, all highlight the necessity of such goal-oriented approaches.

Related to this is the notion of *software reliability*. Software reliability is defined as the probability that the software will work without failure under specified conditions and for a specified period of time [388]. A number of software reliability models are available. They range from the simple Nelson model [392] to more sophisticated models using hyper-geometric coverage [248] and Markov chains [540].

4.3 Metrics

A common belief is that the more complex code is, the more defects it has. But is this really true? In order to investigate this hypothesis, we first must come up with a measure of complexity—or, in other words, a *complexity metric.* A metric is defined as *quantitative measure of the degree to which a system, component or process possesses a given attribute* [238]; the name stems from the Greek work *metron* (μέτρον), meaning "measure". Applied to software, a metric becomes a *software metric*.

Software metrics play an essential part in understanding and controlling the overall software engineering process. Unfortunately, metrics can be easily misinterpreted leading to making poor decisions. In this section, we investigate the relationships between metrics and quality, in particular defects:

Do complexity metrics correlate with defect density?

4.3.1 Background

We begin this section by quickly summarising some of the more commonly used complexity metrics in software engineering in Table 4.1. These metrics can be extracted from the source code information of projects.

Software engineering research on metrics has examined a wide variety of topics related to quality, productivity, communication, social aspects, etc. We briefly survey studies investigating the relationship between metrics and software quality.

Studies have been performed on the distribution of faults during development and their relationship with metrics like size, complexity metrics, etc. [172].

From a design metrics perspective, there have been studies involving the Chidamber/Kemerer (CK) metric suite [111]. These metrics can be a useful early internal indicator of externally-visible product quality [39, 479]. The CK metric suite consist of six metrics (designed primarily as object oriented design measures):

- weighted methods per class (WMC),
- coupling between objects (CBO),
- depth of inheritance (DIT),
- number of children (NOC),
- response for a class (RFC), and
- lack of cohesion among methods (LCOM).

The CK metrics have also been investigated in the context of fault-proneness. Basili et al. [39] studied the fault-proneness in software programs using eight student projects. They observed that the WMC, CBO, DIT, NOC and RFC metrics were

Table 4.1. Commonly used complexity metrics

Metric	Description
Lines of code	The number of non-commented lines of code
Global variables	The number of global variables in a module
Cyclomatic complexity	The Cyclomatic complexity metric [479] measures the number of linearly-independent paths through a program unit
Read coupling	The number of global variables read by a function. (The function is thus coupled to the global variable through the read operation)
Write coupling	The number of global variables written by a function. (The function is thus coupled to the global variable through the write operation)
Address coupling	The number of global variables whose address is taken by a function and is not read/write coupling. (The function is coupled to the global variable as it takes the address of the variable)
Fan-in	The number of other functions calling a given function in a module
Fan-out	The number of other functions being called from a given function in a module
Weighted methods per class	The number of methods in a class including public, private and protected methods
Depth of inheritance	For a given class the maximum class inheritance depth
Class coupling	Coupling to other classes through (a) class member variables, (b) function parameters, (c) classes defined locally in class member function bodies. (d) Coupling through immediate base classes. (e) Coupling through return type
Number of subclasses	The number of classes directly inheriting from a given parent class in a module

correlated with defects while the LCOM metric was not correlated with defects. Further, Briand et al. [82] performed an industrial case study and observed the CBO, RFC, and LCOM metrics to be associated with the fault-proneness of a class.

Structure metrics take into account the interactions between modules in a product or system and quantify such interactions. The information-flow metric defined by Henry and Kafura [230], uses *fan-in* (the number of modules that call a given module) and *fan-out* (the number of modules that are called by a given module) to calculate a complexity metric, $C_p = (\textit{fan-in} \times \textit{fan-out})^2$. Components with a large fan-in and large fan-out may indicate poor design. Such modules have to be decomposed correctly.

4.3.2 Case Study: Five Microsoft Projects

Together with Thomas Ball we performed a large scale study at Microsoft to investigate the relation between complexity metrics and defects [390]. We addressed the following questions:

1. Do metrics correlate with defect density?
2. Do metrics correlate *universally* with defect density, i.e., across projects?
3. Can we predict defect density with regression models?
4. Can we transfer (i.e., reuse) regression models across projects?

For our study, we collected code complexity metrics and post-release defect data for five components in the Windows operating system:

- the HTML rendering module of *Internet Explorer 6 (IE6)*
- the application loader for *Internet Information Services (IIS)*
- *Process Messaging Component*—a Microsoft technology that enables applications running at different times to communicate across heterogeneous networks and systems that may be temporarily offline.
- *Microsoft DirectX*—a Windows technology that enables higher performance in graphics and sound when users are playing games or watching video on their PC.
- *Microsoft NetMeeting*—a component for voice and messaging between different locations.

To protect proprietary information, we have anonymised the components and refer to them as projects A, B, C, D, and E.

In our study, we observed *significant correlations* between complexity metrics (both object oriented (OO) and non-OO metrics) and post-release defects. The metrics that had the highest correlations are listed in Table 4.2. The interesting part of this result is that across projects, different complexity metrics correlated significantly with post-release defects. This indicates that none of the metrics we researched would qualify as universal predictor, even in our closed context of only Windows operating system components.

When there is no universal metric, can we build one by combining existing metrics? We tried by building *regression models* [282] using complexity metrics. In order to avoid inter-correlations between the metrics, we applied *principal component*

Table 4.2. Metrics with high correlations

Project	Correlated Metrics
A	Number of classes and five derivations
B	Almost all metrics
C	All except depth of inheritance
D	Only lines of code
E	Number of functions, number of arcs, McCabe complexity

Table 4.3. Transferring predictors across projects

	Project predicted for				
Project learned from	A	B	C	D	E
A	–	no	no	no	no
B	no	–	yes	no	no
C	no	yes	–	no	yes
D	no	no	no	–	no
E	no	no	yes	no	–

analysis [246] first. For our regression models, we used the principal components as independent variables. As a result of this experiment, we obtained for every project a predictor that was capable of accurately predicting defect-prone components of the project [390].

Next, we applied these predictors across projects. The results in Table 4.3 show that in most cases, predictors could not be transferred. The only exceptions are between projects B and C and between C and E, where predictors are interchangeable. When comparing these projects, we observed that they had similar development processes.

A central consequence of this result would be to reuse only predictors that were generated in similar environments. Put another way: *Always evaluate with history before you use a metric to make decisions.* Or even shorter: *Never blindly trust a metric.*

Key Points

- ✏ Complexity metrics indeed correlate with defects.
- ✏ There is no universal metric and no universal prediction model.
- ✏ Before relying on a metric to make predictions, evaluate it with a true defect history.

4.4 Problem Domain

The chances of making mistakes depend strongly on the number and complexity of the *requirements* that some piece of code has to satisfy. As discussed in the Section 4.3, a large number of interfering requirements can result in a higher code complexity. However, this may not necessarily be so: an algorithm may have a very simple structure, but still may be difficult to get right. Therefore, we expect that specific requirements, or more generally, specific *problem domains,* to impact how defect-prone program code is going to be:

How does the problem domain impact defect likelihood?

Table 4.4. Good and bad imports (packages) in Eclipse 2.0 (taken from [455], ©ACM, 2006)

Packages imported into a component C	Defects	Total	$p(Defect \mid C)$
org.eclipse.jdt.internal.compiler.lookup.*	170	197	0.8629
org.eclipse.jdt.internal.compiler.*	119	138	0.8623
org.eclipse.jdt.internal.compiler.ast.*	111	132	0.8409
org.eclipse.jdt.internal.compiler.util.*	121	148	0.8175
org.eclipse.jdt.internal.ui.preferences.*	48	63	0.7619
org.eclipse.jdt.core.compiler.*	76	106	0.7169
org.eclipse.jdt.internal.ui.actions.*	37	55	0.6727
org.eclipse.jdt.internal.ui.viewsupport.*	28	42	0.6666
org.eclipse.swt.internal.photon.*	33	50	0.6600
...			
org.eclipse.ui.model.*	23	128	0.1797
org.eclipse.swt.custom.*	41	233	0.1760
org.eclipse.pde.internal.ui.*	35	211	0.1659
org.eclipse.jface.resource.*	64	387	0.1654
org.eclipse.pde.core.*	18	112	0.1608
org.eclipse.jface.wizard.*	36	230	0.1566
org.eclipse.ui.*	141	948	0.1488

4.4.1 Imports and Defects

To demonstrate the impact of the problem domain on defect likelihood, let us come back to the distribution of defects in Eclipse, as shown in Figure 4.1. Let us assume we want to extend the existing code by two new components from different problem domains: *user interfaces* and *compiler internals.* Which component is more likely to be defect-prone?

Adding a new dialog box to a GUI has rather simple requirements, in most cases you only have to extend a certain class. However, assembling the elements of the dialog box takes lots of additional complicated code. In contrast, using a parser to build an abstract syntax tree requires only a few lines of code, but has many rather complicated requirements such as picking the correct parameters. So which domain more likely leads to defects?

In the context of Eclipse, this question has been answered. Together with Adrian Schröter, we examined *the components that are used* as an implicit expression of the component's domain [455]. 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. Therefore, what a component imports determines its problem domain.

Once one knows what is imported, one can again relate this data to measured defect densities. Table 4.4 shows how the usage of specific packages in Eclipse impacts defect probability. A component which uses compiler internals has a 86% chance to have a defect that needs to be fixed in the first six months after release. However, a component using user interface packages has only a 15% defect chance.

This observation raises the question whether we can predict defect-proneness by just using the names of imported components? In other words: "Tell me what you import, and I'll tell you how many defects you will have."

4.4.2 Case Study: Eclipse Imports

Can one actually predict defect likelihood by considering imports alone? We built statistical models with linear regression, ridge regression, regression trees, and support vector machines. In particular, we addressed the following questions:

Classification. Can we predict whether a component will have defects?
Ranking. Can we predict which components will have the most defects?

For our experiments, we used *random splits*: we randomly chose one third of the 52 plug-ins of Eclipse version 2.0 as our training set, which we used to build our models. We validated our models in versions 2.0 and 2.1 of Eclipse. Both times we used the complement of the training set as the validation set. We generated a total of 40 random splits and averaged the results for computing three values:

- The *precision* measures how many of the components predicted as defect-prone actually have been shown to have defects. A high precision means a low number of false positives.
- The *recall* measures how many of the defect-prone components are actually predicted as such. A high recall means a low number of false negatives.
- The *Spearman correlation* measures the strength and direction of the relationship between predicted and observed ranking. A high correlation means a high predictive power.

In this section, we discuss our results for support vector machines [132] (which performed best in our evaluation).

Precision and Recall

For the validation sets in version 2.0 of Eclipse, the support vector machines obtained a *precision* of 0.67 (see Table 4.5). That is, two out of three components predicted as defect-prone were observed to have defects. For a random guess instead, the precision would be the percentage of defect-prone packages, which is only 0.37. The *recall* of 0.69 for the validation sets in version 2.0 indicates that two third of the observed defect-prone components were actually predicted as defect-prone. Again, a random guess yields only a recall of 0.37.

In practice, this means that import relationships provide a good predictor for defect-prone components. This is an important result since relationships between components are typically defined in the design phase. Thus, defect-prone components can be identified early, and designers can easily explore and assess design alternatives in terms of predicted defect risk.

Table 4.5. Predicting defect-proneness of Eclipse packages with Support Vector Machines and import relations (taken from [455], ©ACM, 2006)

	Precision	Recall	Spearman Correlation
training in Eclipse v2.0	0.8770	0.8933	0.5961
validation in Eclipse v2.0	0.6671	0.6940	0.3002
—top 5%	0.7861		
—top 10%	0.7875		
—top 15%	0.7957		
—top 20%	0.8000		
validation in Eclipse v2.1	0.5917	0.7205	0.2842
—top 5%	0.8958		
—top 10%	0.8399		
—top 15%	0.7784		
—top 20%	0.7668		

Ranking vs. Classification

The low values for the Spearman rank *correlation* coefficient in Table 4.5 indicate that the predicted rankings correlate only little with the observed rankings. However, the precision values for the top 5% are higher than the overall values. This means that the chances of finding defect-prone components increase for highly ranked components.

In practice, this means that quality assurance is best spent on those components ranked as the most defect-prone. It is therefore a good idea to analyse imports and bug history to establish appropriate rankings.

Applying Models Across Versions

The results for the validation sets of Eclipse version 2.1 are comparable to the ones of version 2.0; for the top 5% and 10% of the rankings, the precision values are even higher. This indicates that our models are robust over time.

For our dataset, this means that one can learn a model for one version and apply it to a later version without losing predictive power. In other words, the imports actually act as *invariants,* as discussed in Section 4.2.

4.4.3 Case Study: Windows Server 2003

At Microsoft, we repeated the study by Schröter et al. [455] on the defect data of Windows Server 2003. Instead of import relationships, we used *dependencies* to describe the problem domain between the 2252 binaries. A dependency is a directed relationship between two pieces of code such as expressions or methods. For our experiments, we use the MaX tool [473] that tracks dependency information at the function level and looks for calls, imports, exports, RPC, and Registry accesses. Again, the problem domain was a suitable predictor for defects and performed substantially better than random (precision 0.67, recall 0.69, Spearman correlations between 0.50 and 0.60).

In addition to the replication of the earlier study (and its results), we also observed a *domino effect* in Windows Server 2003. The *domino effect* was stated in 1975 by Randell [430]:

> *Given an arbitrary set of interacting processes, each with its own private recovery structure, a single error on the part of just one process could cause all the processes to use up many or even all of their recovery points, through a sort of uncontrolled domino effect.*

Applying the domino effect on dependencies, this would mean that defects in one component can increase the likelihood of defects in dependent components. Figure 4.2 illustrates this phenomenon on a defect-prone binary *B*. Out of the three binaries that directly depend on *B* (distance $d = 1$), two have defects, resulting in a *defect likelihood* of 0.67. When we increase the distance, say to $d = 2$, out of the four binaries depending on *B* (indirectly), only two have defects, thus the likelihood decreases to 0.50. In other words, the extent of the domino effect decreases with distance—just like with real domino pieces.

In Figure 4.3 we show the distribution of the defect likelihood, as introduced above, when the target of the dependency is defect-prone ($d = 1$). We also report results for indirect dependencies ($d = 2$, $d = 3$). The higher the likelihood, the more dependent binaries are defect-prone.

To protect proprietary information, we anonymised the *y*-axis which reports the frequencies; the *x*-axis is relative to the highest observed defect likelihood which was

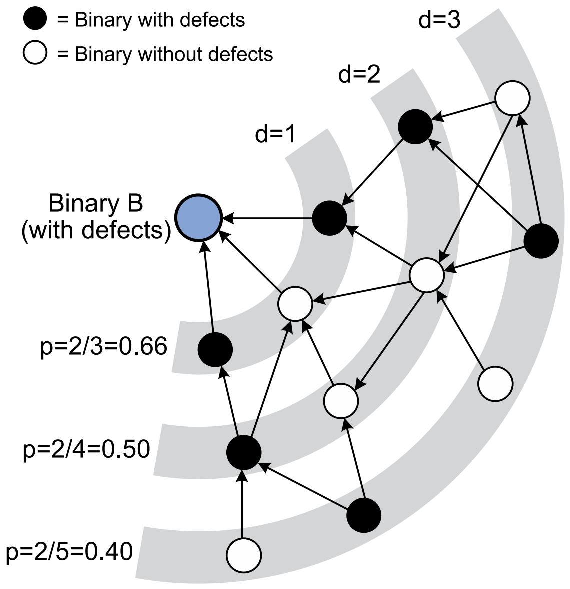

Fig. 4.2. Example of a domino effect for a binary B

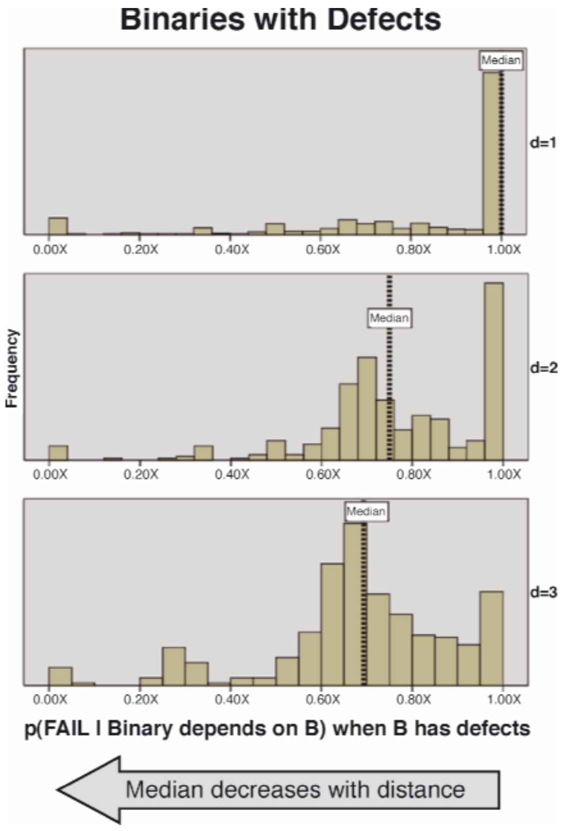

Fig. 4.3. The domino effect in Windows Server 2003

greater than 0.50 and is reported as X. The likelihood X and the scale of the x-axis are constant for all bar charts. Having the highest bar on the left (at 0.00), means that for most binaries the dependent binaries had no defects; the highest bar on the right (at X), shows that for most binaries, their dependent binaries had defects, too.

As Figure 4.3 shows, directly depending on binaries with defects, causes most binaries to have defects, too ($d = 1$). This effect decreases when the distance d increases (trend towards the left). In other words, the domino effect is present for most defect-prone binaries in Windows Server 2003. As the distance d increases, the impact of the domino effect decreases. This trend is demonstrated by the shifting of the median from right to left with respect to the defect likelihood.

Key Points

- ✏ The set of used components is a good predictor for defect proneness.
- ✏ The problem domain is a suitable predictor for future defects.
- ✏ Defect proneness is likely to propagate through software in a domino effect.

4.5 Code Churn

Code is not static; it evolves over time to meet new requirements. The way code evolved in the past can be used to predict its evolution in the future. In particular, there is an often accepted notion that code that changes a lot is of lower quality—and thus more defect-prone than unchanged code.

How does one measure the amount of change? Lehman and Belady [320] introduced the concept of *code churn* as the rate of growth of the size of the software. But measuring the changes in the size of the software does not necessarily capture all changes that have occurred during the software development, this is especially true if the software has been re-architected. More generally, code churn can be defined as a measure of the *amount of code change taking place within a software unit over time.* [389]. The primary question we address in this section is:

Does code churn correlate with defects?

4.5.1 Background

Several researchers have investigated how evolution relates to defects. Ostrand et al. [408] use information of file status such as new, changed, unchanged files along with other explanatory variables such as lines of code, age, prior faults etc. to predict the number of faults in multiple releases of an industrial software system. The predictions made using binomial regression model were of a high accuracy for faults found in both early and later stages of development.

Munson et al. [384] studied a 300 KLOC (thousand lines of code) embedded real time system with 3700 modules programmed in C. Code churn metrics were found to be among the most highly correlated with problem reports.

Graves et al. [209] predicted fault incidences using software change history. The most successful model they built computed the fault potential by summing contributions from changes to the module, where large and/or recent changes contribute the most to fault potential.

4.5.2 Case Study: Windows Server 2003

In addition to the above research results, we now summarise in detail the results of a case study performed on Windows Server 2003 [389]. We analysed the code churn between the release of Windows Server 2003 and the release of the Windows Server 2003 Service Pack 1 (Windows Server 2003-SP1) to predict the defect density in Windows Server 2003-SP1.

As discussed in Section 4.5.1, there are several measures that can be used to explain code churn. In our study, we used the following churn measures [389]:

- *Total LOC* is the number of lines of non-commented executable lines in the files comprising the new version of a binary.

- *Churned LOC* is the sum of the added and changed lines of code between a baseline version and a new version of the files comprising a binary.
- *Deleted LOC* is the number of lines of code deleted between the baseline version and the new version of a binary.
- *File count* is the number of files compiled to create a binary.
- *Weeks of churn* is the cumulative time that a file was opened for editing from the version control system.
- *Churn count* is the number of changes made to the files comprising a binary between the two versions (Windows Server 2003 and Windows Server 2003-SP1).
- *Files churned* is the number of files within the binary that churned.

The overall size of the analysed code base was around 40 million lines of code from more than 2000 binaries. Using the above extracted metrics from the version control system, we use a *relative approach* (as shown in Figure 4.4) to build our statistical regression models to predict system defect density. Our rationale for relative metrics is that in an evolving system, it is highly beneficial to use a relative approach to quantify the change in a system. A more detailed discussion of the experiment is available in [389].

Using random splitting techniques we used the above "relative" code churn measures as predictor variables in our statistical models. We selected two-thirds of the binaries to build our statistical prediction models (multiple regression, logistic regression) to predict overall system defect density/fault-proneness. Based on our sta-

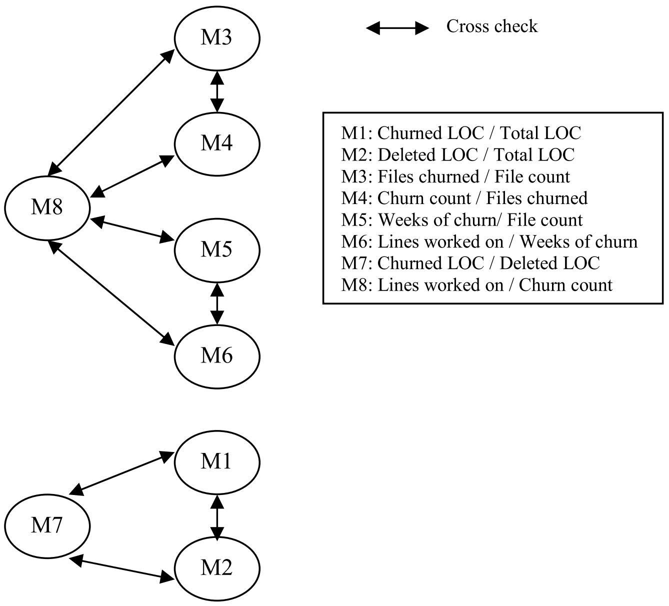
Cross check
M1
M2
M3
M4
M5
M6
M7
M8
M1: Churned LOC / Total LOC
M2: Deleted LOC / Total LOC
M3: Files churned / File count
M4: Churn count / Files churned
M5: Weeks of churn/ File count
M6: Lines worked on / Weeks of churn
M7: Churned LOC / Deleted LOC
M8: Lines worked on / Churn count

Fig. 4.4. Relative churn measures (taken from [389], ©ACM, 2005)

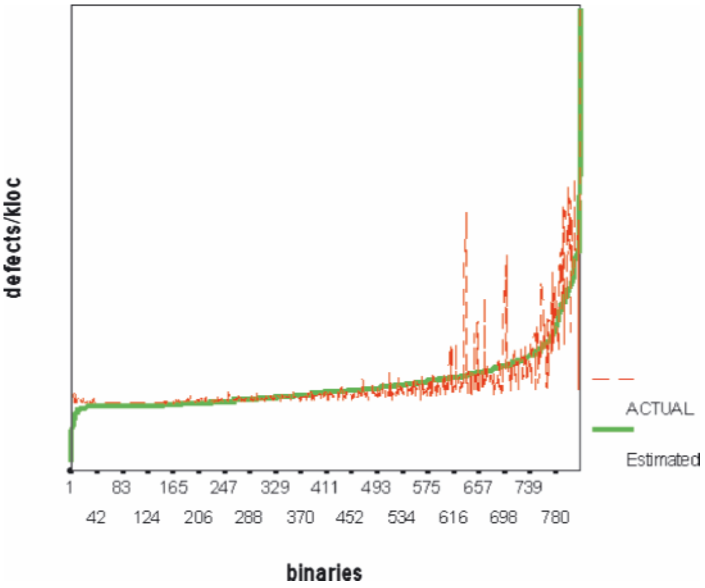

Fig. 4.5. Plot of actual versus estimated defect density (taken from [389], ©ACM, 2005)

tistical analysis we were able to predict system defect density/fault-proneness at high levels of statistical significance. Figure 4.5, for example, shows the results of the one of the random split experiments to predict the actual system defect density.

Key Points

✏ The more a component has changed (churned), the more likely it is to have defects.
✏ Code churn measures can be used to predict defect-prone components.

4.6 Open Issues

We have seen how complexity, the problem domain, or the change rate can be used to learn from history and to predict the defect density of new and evolved components. We can thus indeed predict bugs from history, and even do so in a fully automatic way. This is a clear benefit of having a well-kept track of earlier defects: avoiding future defects by directing quality assurance efforts.

The examples in this chapter all rely on *code features* to predict defects. By no means have we analysed all possible code features that might turn out to be good defect predictors. We expect future research to come up with much better predictors; Table 4.6 lists several data sets that are publicly available such that anyone can test her or his favourite idea.

Table 4.6. Datasets for empirical studies

Promise Data Repository	The *Promise Data Repository* contains data sets for effort estimation, defect prediction, and text mining. Currently, it comprises 23 datasets, but this number is constantly growing (*free*). `http://promisedata.org/`
NASA Metrics Data	The repository of the *NASA IV&V Facility Metrics Data Program* contains software metrics (such as McCabe and Halstead) and the associated error data at the function/method level for 13 projects (*free*). `http://mdp.ivv.nasa.gov/`
Eclipse Bug Data	This data contains the pre-release and post-release defects for three versions of the Eclipse IDE (*free*). `http://www.st.cs.uni-sb.de/softevo/`
ISBSG	The repository of ISBSG contains empirical data for software estimation, productivity, risk analysis, and cost information (*commercial*). `http://www.isbsg.org/`
Finnish Data Set	This dataset is collected by STTF to support benchmarks of software costs, development productivity, and software processes (*commercial*). `http://www.sttf.fi/`
FLOSSmole	FLOSSmole is a “collaborative collection and analysis of free/libre/open source project data.” `http://ossmole.sourceforge.net/`

Another common feature of the approaches discussed so far is that they all predict the number of defects. In general, though, managers not only want to minimise the number of defects, but minimise the overall *damage,* which involves the impact of each defect—that is, the number of actual *failures,* and the damage caused by each failure.

Finally, all the predictions discussed in this chapter require a *history of defects* to learn from. Over history, we can learn which features of the code or the development process are most likely to correlate with defects—and these very features can thus be used to predict defect-prone components. Of course, the more detailed this history of failures is, the more accurate our predictions will be. However, having a long history of failures is something we would like to avoid altogether. At least, we would like to learn enough from one project history to avoid repeating it in the next project:

- Is there a way to make predictions for new products with no known history yet?
- How can we leverage and abstract our knowledge about defects from one project to another one?
- Are there any *universal* properties of programs and processes that invariably result in a higher defect density?

We believe that such universal properties indeed do exist. However, it is very unlikely that these properties are code properties alone. Remember that we focus on defects that occur *after* release, that is, at a time where people already have taken care of quality assurance. It is reasonable to assume that the more (and better) quality assurance is applied, the fewer defects will remain. However, none of our predictor models takes the extent or effectiveness of quality assurance into account—simply because code does not tell how it has been tested, checked, reviewed, or verified.

The effectiveness of quality assurance is a feature of the software process, not the software itself. There are ways to characterise and evaluate quality assurance techniques—for instance, one can check the coverage of a test suite or its effectiveness in uncovering mutations. These are important features of the software process that may help predicting defects.

Besides quality assurance, there are further process features to look into. The qualification of the programmer, combined with the time taken to write a component; the quality of the specification; the quality of the design; the competence of management; continuity of work flow—all these, and many more, are factors which contribute to people making mistakes (or avoiding them). In some way, looking for universal properties that cause defects is like looking for a universal way to write software. As an intermediate goal, it may already be helpful to choose between multiple "good" ways.

Whatever features future predictors will be based upon—there is one invariant that remains: Any predictor will eventually turn out to be wrong. This is because if a predictor predicts a higher number of defects (or failures, or damage, for that matter), the component will be checked more carefully—which will, of course, reduce density. Any defect predictor will thus produce self-defeating prophecies—and this is a good thing.

Key Points

✏ So far, all defect predictors require a history of earlier defects.
✏ Automated predictors do not yet directly leverage process data.
✏ The quest for universal (e.g. history-less) defect predictors is still open.

4.7 Threats to Validity

As with all empirical studies drawing general conclusions from case studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables. For this reason, we cannot assume a priori that the results of a study generalise beyond the specific environment in which it was conducted [40]. Some of the threats to validity of our studies are discussed below.

- There could have been errors in measurement. In general, all the measurement of software data in our studies was done using automated tools. This alleviates to a certain degree errors in measurement. But it is possible that these tools could have had design errors that could have led to errors in measurement

- Our case studies were performed for two large systems namely Windows Server 2003 and Eclipse. It is possible that these results may not be observed for smaller or other systems. Further these systems have possibly several million users. Hence it is possible that other systems which do not have such an active usage profile may not have most of its field defects found equally.
- At least one of the three authors were part of each case study described in this chapter. Unknowingly it would have been possible to introduce experimenter bias into the case studies.
- All the analysis in our studies was done after-the-fact, i.e., all the field defects had been reported back and we had used it for our prediction models. It is difficult for us to gauge how our prediction made before hand would have influenced the development team behaviour effectively benefiting them to identify problem-prone components early.
- The statistical models built for the software systems may apply only for the particular family of systems for which they are built for [61]. For example it may not be useful to use a model built on Eclipse to predict bugs in small toy programs.
- Though our case studies predict defects significantly other information such as severity, impact of failure information etc. are missing from our predictions. This type of predictions would be part of future investigations in this research area.

Basili et al. state that researchers become more confident in a theory when similar findings emerge in different contexts [40]. Towards this end we hope that our case study contributes towards building the already existing empirical body of knowledge in this field [275, 274, 384, 402, 172, 209, 408, 326].

4.8 Conclusion and Consequences

Learning from history means learning from successes and failures—and how to make the right decisions in the future. In our case, the history of successes and failures is provided by the bug database: systematic mining uncovers which modules are most prone to defects and failures. Correlating defects with complexity metrics or the problem domain is useful in predicting problems for new or evolved components. Likewise, code that changes a lot is more prone to failures than code that is unchanged.

Learning from history has one big advantage: one can focus on the aspect of history that is most relevant for the current situation. In our case, this means that predictions will always be best if history is taken from the product or project at hand. But while we can come up with accurate predictors, we need more work in understanding the root causes for software defects—and this work should take into account the roles of quality assurance and the general software process.

In this light, we feel that our work has just scratched the surface of what is possible, and of what is needed. Our future work will concentrate on the following topics:

More process data. As stated above, code features are just one factor in producing software defects; process features may be helpful in finding out why defects are not caught. We plan to tap and mine further data sources, notably quality assurance information, to cover more process data.

Better metrics and models. Right now, the metrics we use as input for predictors are still very simple. We plan to leverage the failure data from several projects to evaluate more sophisticated metrics and models that again result in better predictors.

Combined approaches. So far, the approaches described in this chapter all have examined specific features in isolation. Combining them, as in "I have a complex module importing internal.compiler, which has had no defects so far" should yield even better predictions.

Increased granularity. Rather than just examining features at the component level, one may go for more fine-grained approaches, such as caller-callee relationships. Such fine-grained relationships may also allow predictions of defect density for individual classes or even individual methods or functions.

Overall, these steps should help us not only to predict where defects will be, but also to understand their causes, such that we can avoid them in the future. Version archives play a key role in telling which hypotheses apply, and which do not—for the project at hand, or universally.

Acknowledgement. Christian Lindig, Rahul Premraj, and Andrzej Wasylkowski provided helpful comments on earlier revisions of this chapter. The work of Andreas Zeller and Thomas Zimmmermann on mining software archives was supported by Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Thomas Zimmermann is additionally supported by the DFG Graduiertenkolleg "Leistungsgarantien für Rechnersysteme". We would like to acknowledge the Eclipse developers as well as the product groups at Microsoft for their cooperation in these studies.