CHAPTER

Learning from Mistakes 16

At the end of each debugging session, one wonders how the defect could have come to be in the first place. This chapter discusses techniques to collect, aggregate, and locate defect information; techniques to predict where the next defects will be; and what to do to prevent future errors.

16.1 WHERE THE DEFECTS ARE

In Figure 1.10, we have already seen a visualization of the defect distribution in ECLIPSE, showing for every single component how frequently it was involved in fixing a defect. As you can see, the distribution of defects across packages is very uneven. For instance, compiler components in ECLIPSE have a defect density that is four to five times higher than in user-interface components. This uneven distribution of defects is typical for software projects. It is commonly known as *Pareto's law*: 80 percent of the defects are found in 20 percent of the modules.

Pareto's law also holds if one focuses on specific *subsets* of defects. Figure 16.1 shows the distribution of *vulnerabilities* across the components of MOZILLA (commonly known as the FIREFOX Internet browser); here, the darker a component is, the more vulnerabilities it has. A vulnerability is a defect in one or more components that manifests itself as some violation of a security policy; again, the distribution was obtained by relating the MOZILLA vulnerability advisories to the MOZILLA change database. Again, the distribution of defects is uneven: While usual suspects, such as the JAVASCRIPT interpreter ("js," bottom left in figure), account for several vulnerabilities, 96 percent of all components never had a single vulnerability—or at least, none discovered yet.

Once we have such a defect distribution, we can use it to guide the debugging process toward the *usual suspects*. Suppose we are searching for a defect and have the choice between multiple components to examine. Then, the *previous defect history* becomes an important factor in prioritizing the search: If a component had several defects in the past, it is likely to have more waiting to be uncovered.

The main usage of such defect distributions, though, is to use them to *learn from the past*. By focusing on those components with the most defects, or otherwise

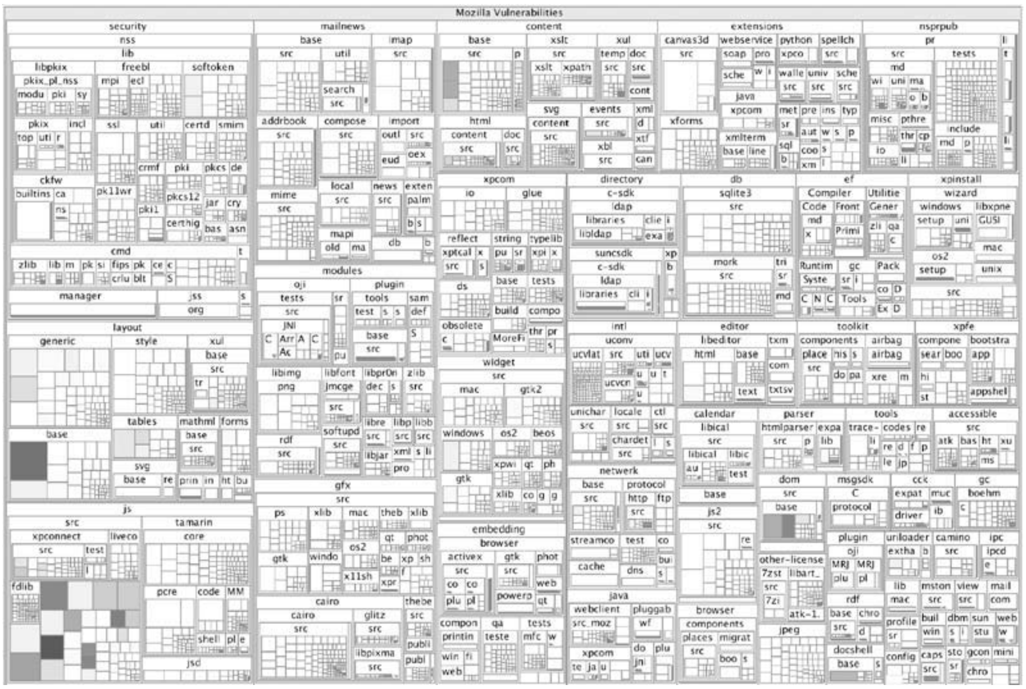


FIGURE 16.1

Vulnerability distribution in MOZILLA. Each rectangle stands for a component; the darker the rectangle, the more vulnerabilities were discovered (and fixed) in the component after release. *Source:* Neuhaus et al. (2007).

investigating the defect history, we can check whether we have multiple defects with common properties—and learn from such *defect patterns* to avoid them in the future. In the remainder of this chapter, we discuss the key question:

HOW CAN WE LEARN FROM PAST DEFECTS TO PREVENT FUTURE ONES?

16.2 MINING THE PAST

Before we discuss how to *interpret* data on earlier defects, let us first show how to *obtain* such data. Generally, past defect data are easily obtained from existing sources. The first source is the *version archive*, which records all changes to the system. Of course, we do not want to investigate these changes manually, but rather *automate* as much as we can. Unfortunately, such automation rarely comes as a turnkey solution—one has to set up or adapt appropriate *fact extractors* for the situation at hand. The first task is to distinguish the different types of changes—in general, we only want the subset that are *fixes.* There are a number of ways to extract just the fixes:

Relate changes to problems. Problem databases (see Chapter 2) store all problems ever encountered with the software. If a problem is listed as *fixed,* there should be a corresponding change in the version database. Such a corresponding change can be found by

- Checking the *log message* of the change against the PR number as stored in the problem database. Here is a typical log message that contains a PR number:

```
Fixed bug #3547.
```

- Check the *closing time* of the problem report. Typically, developers will commit their fix to the change database immediately before marking the problem report as *closed,* so simply checking for the developer's change activity in the few minutes before closing the report will likely point you to the related change.

The advantage of relating bug and change databases is that it provides additional information about the original defect, such as its *severity.* (You normally want to filter out problems classified as *trivial* or *request for enhancement.*) Also, you can check when and by whom the problem was discovered: in-house, during beta testing, or in production code. The disadvantage is that there will still be *noise:* In experiments with ECLIPSE and MOZILLA, Śliwerski et al. (2005a) could only relate about 50 percent of "closed" problem reports to a fix.

Check for changes in maintenance branches. Once a product is released, many organizations create a dedicated *maintenance branch* off the main versioning trunk—a branch in which only fixes to the released product are committed. By assessing only such maintenance branches, you obtain the subset of fixes. The advantage of this approach is that it provides precise information without requiring a problem database. The disadvantage is that it requires discipline and effort to set up and keep up the standards.

Check the message log for keywords. A simple way to identify fixes is to just check the log message for keywords such as "fix," "bug," "problem," "defect," "crash," and so on: If one of these keywords appears in the log message of a change, then the change is likely a fix:

```
Program crashed when encountering end-of-file on input.
```

The advantage of this approach is that it is very lightweight. The disadvantage is that it does not provide information about the original problem, and where it was discovered. If the problem was discovered by in-house testing, that is just fine—that is what in-house testing is there for. If the problem was discovered by one of your biggest customers, you definitively want to improve your in-house testing such that the problem does not occur again. In practice, such keyword information is thus typically used together with problem databases or maintenance branches.

None of these methods is perfect—there will always be problem reports that cannot be mapped to a fix and vice versa. If you set up a new project, consider tools and conventions that ensure a tight mapping between tasks and changes—and while you're doing that, track *effort* as well, such that you know how much effort went into individual tasks.

Even with an imperfect mapping, any of these methods (or even a combination thereof) will help you map the large majority of problems to fixes—and thus map defects to individual components. Once you have appropriate tools in place, defect distributions can be generated and updated at the touch of a button—by mining and remining the appropriate software archives.

16.3 WHERE DEFECTS COME FROM

Defects that escape into production are typically (and should be) pretty rare events. Each defect tells its own story, and by examining past defects and where they occurred, you will be able to determine common origins and cross-cutting concerns. Here are some first questions you should ask yourself:

- *Which modules have had the most defects?* If a module had several defects in the past, it is likely to have more waiting to be uncovered. Consider subjecting such a module to thorough quality assurance, or refactor it into smaller, less error-prone units.
- *When are most defects introduced?* Do they originate in the requirements/design/coding phase? If specific phases are more error prone than others, you may need to increase quality assurance in these phases, or rework the development process.
- *Which types of errors occur most often?* This can be extracted from *descriptions of the defect*—typically with categories such as "use of noninitialized variable," "bad control flow," "heap misuse," and so on. Consider using (or building) tools that check for these types of errors.
- *Who introduced the defects?* Some people create more defects than others—simply because they write more code, or because they address the most risky issues. If you find that some people or groups create more defects than normal, assign them to less risky tasks, or consider appropriate training.

  Note that this is a sensitive issue. If developers find that information in problem or version archives is used against them, they will no longer use these tools. Rather than blaming people, create an environment that focuses on finding errors.

All these measures have the same aim: to *fix the development process* rather than just its product. By fixing the process, we can address the ultimate cause of the problem—that is, *us* as humans and the decisions (and errors) we make. To fix the process, we must learn what the previous defects *have in common* such that we can either fix their common source, or otherwise prevent them in the future. This is where questions like the ones just noted can help.

Over time, though, practitioners and researchers alike have identified common sources of errors, as well as ways to avoid them and detect them early. In the next three sections, we discuss three important stages in the development process that all can introduce defects:

- *Specification:* What is it that makes a specification error prone?
- *Programming:* What is it that makes it hard to translate a specification into a correct program?
- *Quality assurance:* How can quality assurance fail to catch the defect?

For each of these stages, we present a short *rationale* on how the individual stage can introduce defects, a set of *consequences* that improve process quality, and some *ways to measure* which parts of the product one should focus on first.

16.4 ERRORS DURING SPECIFICATION

Why is specification hard? Complexity in specification stems from the sheer quantity of requirements to be fulfilled, as well as the interaction between these requirements. The more requirements and constraints there are to be met, the higher the chance of making a mistake—and the higher the probability of a defect in the resulting program.

Finally, the specification may be incomplete, inconsistent, frequently changing, or completely missing. All of these increase the chances of a defect—although in the absence of precise specification, *surprise* would be a more adequate term.

16.4.1 What You Can Do

When analyzing defects, assess whether they were introduced during the specification phase—that is, they could have been avoided by improving the specification. If you find that several defects are due to specification issues, following are measures you can take:

Improve quality assurance. Introduce systematic processes for eliciting requirements and software design. Improve better quality assurance on specification documents. Check specifications early—the earlier you check your artifacts against the requirements of your clients, the more precise the requirements will become—in particular, regarding what the product should *not* do.

Increase the precision. Introduce semiformal or formal methods that allow you to catch inconsistencies or incomplete specifications early.

Increase the degree of automation. The more development tasks you can automate, the less chances there are that the humans involved make mistakes. Consider contracts, assertions, and other forms of specifications that can be validated automatically (see Chapter 10). Generative techniques such as *model-driven development* also take humans out of the loop.

Note that these are *general recommendations* that have held for several projects in the past, but may not necessarily hold for yours. You should always go and validate these recommendations against the true defect distribution in your project, as obtained from history. Once you do that, though, you will have excellent facts in your hand when it comes to suggesting or implementing process improvements.

16.4.2 What You Should Focus On

Although it won't hurt to improve overall specification quality, you would normally prefer to focus on those problems that hurt most—that is, those components where the most defects related to specification issues have been found. This requires a manual classification of the *origin of the problem:* If the problem is due to a bad specification, it should be examined further.

However, you may not want to wait with your analysis until the product has been released—but rather check for early warning signs already during development. Unfortunately, checking whether a specification is adequate is hard to measure directly. If specification comes in natural language (as is mostly the case), human assessment will always be necessary. *Formal specifications* can be checked for internal consistency and completeness—but may still miss the client's requirements.

Problems with specifications frequently translate into problems during development, though—and these can be measured as follows:

- A *history of frequent changes* is a symptom of changing or incomplete specifications, as the component must constantly be adapted. A number of studies have found that a high number of changes in a component correlates with the number of defects. Therefore, components that are frequently changed should be checked whether their specification can be made more stable and precise.
- Some *problem domains* are more complex than others, resulting in more complex specifications, and thus a higher chance of defects. For instance, *compiler* components in ECLIPSE have shown a defect density that is four to five times higher than in *user interface* components. Be sure to identify these problem domains and focus on them to improve specification quality.

Again, what has been observed for other projects is likely to hold for yours, too—but it is much better to have facts in your hand before making decisions. If you can show that defects correlate with frequent changes in your program, too, you will know what to focus on.

16.5 ERRORS DURING PROGRAMMING

In a program, the functionality as originally specified is spread and detailed across several components. Again, the more requirements and constraints there are to be met by a single component, the higher the chance of a defect in this component.

Furthermore, programs show *structural complexity*:

- The more possible paths a computation can take (both controlwise and datawise),
- the longer these paths, and
- the more components involved in the computation,

the greater the chance of introducing an infection in one of these paths—and the higher the chance of a failure.

16.5.1 What You Can Do

A defect means that code does not match its specification—and if the specification (Section 16.4) is correct, then the code must be wrong. Here are some measures to take to avoid errors:

Improve programming as you improve specifying. A complex specification always translates into a complex program. (If there were a simple program, we could also come up with a simple specification.) Thus, anything you can do to make the specification more precise, complete, or better validated, as discussed in Section 16.4, will also improve the program as follows:

- Improve quality assurance of specifications.
- Increase the precision of specifications.
- Increase the degree of automation.

Reduce complexity. The higher the structural complexity of a program, the more ways there are for some part of the program to influence another—and the easier it is for infections to spread. Limiting the information flow and the amount of interaction between components

- Reduces the chance of an infection becoming a failure.
- Makes it easier to validate the possible paths and behaviors.
- Facilitates program understanding and reviewing.

All this is achieved by good software design, thereby reducing risk and reducing complexity.

Improve documentation. When writing software, write for humans just as well as you write for the machine. Misunderstanding how a piece of software works is a frequent cause of problems when using or changing it. Documentation should be thorough and precise; if it is too verbose, consider breaking a component into simpler pieces. Assertions (see following) combine documentation and validation in a single attractive package.

Set up assertions. If good software design limits the spread of infections, assertions catch them right away. If, while debugging, you have inserted assertions to narrow down the infection, *keep them in the code*. If some assertion would have helped catching the infection, go and write one. These assertions will catch similar infections in the future. At the very least, they will help during the debugging process. Consider keeping assertions active in production code (see Section 10.9 in Chapter 10).

Improve training. Many defects come to be because of simple mistakes. If you can characterize and summarize these mistakes to recurring patterns, go and make sure your programmers know about these patterns, and how to avoid them.

Change your programming language. Most features in new programming languages are designed to avoid the programming errors as experienced by the users or *older* programming languages. This is how great error-avoiding concepts like static typing, garbage collection, synchronized threads, or design by contract came along. If you can relate problems to features (or missing features) of a programming language, consider alternatives for your next project.

16.5.2 What You Should Focus On

Development is a stage in which problems with specifications manifest themselves.

- *Recent code changes before release* need a special focus, because code changed at the last minute may not be as thoroughly tested as older code. This risk comes in addition to the history of *frequent changes* as an early indicator of potential problems, as discussed in Section 16.4. Frequency and recency of changes can be mined from the version archive; focus on those components with the most and most recent changes.
- *Component imports*—that is the set of components interacted with—can characterize the *domain* of a component, and thus the proneness to errors, as discussed in Section 16.4.

  In a study of MOZILLA vulnerabilities, Neuhaus et al. (2007) found that vulnerable components shared similar sets of *imports.* In the case of MOZILLA, for instance, they found that of the 14 components importing `nsNodeUtils.h`, 13 components (93 percent) had to be patched because of security leaks. The situation is even worse for those 15 components that import `nsIContent.h`, `nsIInterfaceRequestorUtils.h`, and `nsContentUtils.h` together—they *all* had vulnerabilities. Identify those imports that correlated with defects in the past and focus on them.

The intrinsic problem of development, namely *structural complexity of code*, can be measured, too.

Software complexity metrics capture the complexity of programs by counting the number of classes, functions, variables, parameters, blocks, branches, or paths in a program: the more items, the higher the complexity. Although such metrics provide only a crude approximation to complexity, they are easy to compute, and have been shown multiple times to be related to effort in comprehension and in testing.

When it comes to defects, though, the picture is less clear. In a study of five Microsoft projects, Nagappan et al. (2006) found that correlation of metrics and defects was different in each project, and that complexity metrics should only be used when validated from past defect history.

Centrality measures pay special attention to *interactions between elements,* as most complexity metrics only focus on single components. The idea is to examine the relation between dependencies and defects. A recent study by Zimmermann and Nagappan (2008) on Windows Server 2003 has shown that:

- The more complex the dependencies of a component are, the more defects it will have.
- In addition, the presence of *cyclic dependencies* increases the number of defects.
- The more important (central) a component is in the dependency graph, the more defects it will have.
- Finally, they also observed a *domino effect* for binaries: If a component depends on defect-prone components, the likelihood of a defect is increased.

If you focus on components with such properties, you may also catch those that are most defect prone. As complexity metrics, centrality measures should also be validated from past defect history.

16.6 ERRORS DURING QUALITY ASSURANCE

Everyone knows that humans make mistakes. This is why there is *quality assurance*—to catch defects before they escape into production. Thus, any problem that escapes into production also implies a problem in quality assurance.

What are the reasons for quality assurance to fail to catch a defect? To start with, all quality-assurance techniques are limited. By its very nature, *testing* can only cover a limited set of executions, and there is always a chance to miss some specific aspect of the behavior (notably the aspects where the program fails). *Formal verification* and other symbolic techniques can cover all behavior, but only at a certain abstraction layer, and may therefore miss issues as they occur in real-life environments. No technique can be both concrete enough to capture all aspects of real life and abstract enough to capture all possible executions. In the end, all these problems can be reduced to the undecidability of the *halting problem,* which makes it generally impossible to predict what a given program will or will not do.

This is not to say that one cannot live within these limitations. But again, complexity in specification or programs directly translates into complexity in quality assurance, too—there simply is more to test, check, review, or prove. Even if some of these activities can be automated, humans will have to conduct several of them, and humans can err during quality assurance just as they can during development.

16.6.1 What You Can Do

Improve your test suite. Your test suite has failed to detect the problem. Therefore, you must extend the test suite such that the problem will not occur again. Your extension should not only test for the particular problem, but for related problems, too: If you find that the original problem was due to, say, a missing null pointer check, try to trigger similar errors by additional tests.

Test early, test often. The earlier you test, the more precise your specification will be, and the more you test, the more defects you will catch. An *automated test* (as discussed in Chapter 3) that reproduces the original problem is a good starting point.

Review your code. Software inspections have been shown to significantly increase productivity, quality, and project stability. We have already seen how being explicit helps in debugging (see Section 6.4 in Chapter 6). If you need to explain what your code does to others (or simply *know* that others will review it), you will also be explicit—and clearly formulate both expectations and contributions.

Improve your analysis tools. Verify whether common tools could have detected the defect early—in particular, tools that detect *code smells* (see Section 7.5 in Chapter 7) or tools that verify *system assertions* (see Section 10.8 in Chapter 10). *Checking programs* statically can help, too—for instance, using the techniques discussed in Chapter 7.

Calibrate coverage metrics. Many people believe in *coverage metrics* like "Our test covered 90 percent of all branches" to express whether a test suite is adequate. Unfortunately, even a high-coverage metric does not say much about the true quality of your code, unless you managed to cover those 20 percent of most of the defects and most of the risk. Therefore, it is better to *focus* on those components that need most of your attention rather than spending lots of effort on some piece of code that never showed any problems so far.

Consider mutation testing. Mutation testing is a technique to evaluate the adequateness of automated quality assurance. It automatically seeds artificial defects ("mutants") into the product and then checks whether quality assurance detects the mutant. If a mutant is not detected, this suggests that real defects may not be detected either. While being computationally expensive, mutation testing is a good simulation of a defect's life cycle—and its results are much more meaningful than artificial coverage metrics.

16.6.2 What You Should Focus On

In quality assurance, you should always focus on those components with the highest *risk*—that is, those with the highest likelihood and the highest severity of a failure, such as:

Components that have shown risk in the past are obviously the first suspects to focus on. A defect distribution obtained from version and problem archives, as discussed in Section 16.2, is a good starting point. To obtain a measure of risk, you need to also assess

- The *likelihood* of a potential defect causing a failure: Will the defect ever be executed at all? How far will the infection spread?
- The *severity* of the induced failure: How much damage will be caused by this failure? Will it lead to major loss of data, functionality, or valuable assets?

Components that are similar to risky ones are also suspect, as problems tend to have the same causes. Focus on

- Similarities in change history or problem domains (Section 16.4).
- Imports, complexity metrics, or centrality measures (Section 16.5).

Formally, this category also comprises all components that have shown risk in the past, because the defects reported in the past have been fixed, and thus, the "fixed" version of the component is similar to the earlier "risky" version.

Code that is not covered yet by quality assurance naturally is more likely to have undiscovered defects, and thus to fail. Therefore, try to achieve some basic coverage, such as statement coverage, across the product during quality assurance. Before moving to more advanced coverage criteria, first focus on the risky components, as described above.

Note that all this focusing applies as long as there is a Pareto effect—that is, as long as there is an uneven distribution of risk across the program. Once the risk is spread evenly, quality assurance should be spread evenly too.

By allocating your quality-assurance resources wisely, you will hopefully eliminate all of the most risky defects in your product—and, in the process, learn enough about past defects such that your quality assurance will also catch the most risky defects in the future.

16.7 PREDICTING PROBLEMS

In the previous sections, we have discussed how to relate features of specification, code, or quality assurance to defects. As all data from past defects is readily available, wouldn't it be possible to *automatically learn* which features correlate with defects, and thus *automatically predict* problems without the hassle of manual investigation? In the past years, a number of researchers have investigated this very question—and devised automatic predictors with surprising precision. In this section, we give four examples for such tools.

16.7.1 Predicting Errors from Imports

The VULTURE tool by Neuhaus et al. (2007) is a tool for *detecting* where in a product security vulnerabilities have occurred in the past. VULTURE mines a vulnerability database (a special kind of problem database) as well as a version database; using the techniques from Section 16.2, it then maps vulnerabilities to fixes and thus to components. The vulnerability distribution in MOZILLA (Figure 16.2) was obtained using VULTURE.

During the work with VULTURE, Neuhaus et al. discovered that vulnerabilities could be linked to specific *imports,* as discussed in Section 16.5.2: Components importing `nsNodeUtils.h` were all prone to vulnerabilities. They set up a *machine learner* (more specifically, a *support vector machine*) that was trained with the number of vulnerabilities and the set of imports for each component. The resulting support vector model would then also be used for *predicting* vulnerabilities by feeding in the imports; the model would then produce an estimate for the vulnerability. In an evaluation, Neuhaus et al. found that of all components flagged as vulnerable by the model, 70 percent turned out to be vulnerable; among those 30 components flagged as *most* vulnerable and 88 percent were vulnerable.

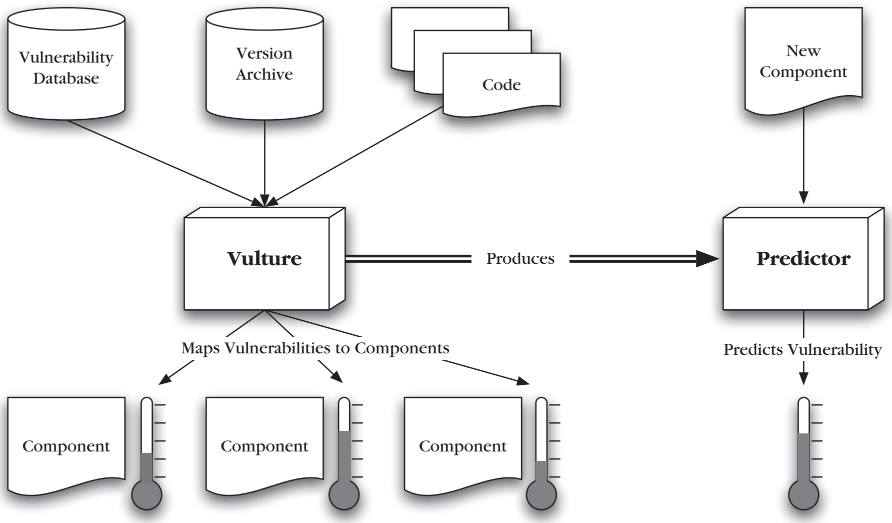


FIGURE 16.2

How VULTURE works. By mapping vulnerabilities to fixes and thus to components, VULTURE can assign an individual vulnerability for each component. By learning which *imports* correlate with vulnerabilities, VULTURE can predict the vulnerability of further components.

16.7.2 Predicting Errors from Change Frequency

In the past three years, Microsoft has conducted extensive research on how to predict defects in software. In 2005, Nagappan and Ball mined the change and problem databases of Windows Server 2003 to examine the relation between *code churn,* the extent of change made to a component over a period of time, and defect density. Using statistical regression models, they found that *relative* measures of code churn were high predictors of defect density—that is, those components that were changed most (in comparison to others) were also the most defect prone.

Their model is also able to discriminate between defect-prone and non-defect-prone components—that is, based on the change history, it would predict whether a component would be defect prone or not. In their experiment, the model reached an accuracy of 89 percent.

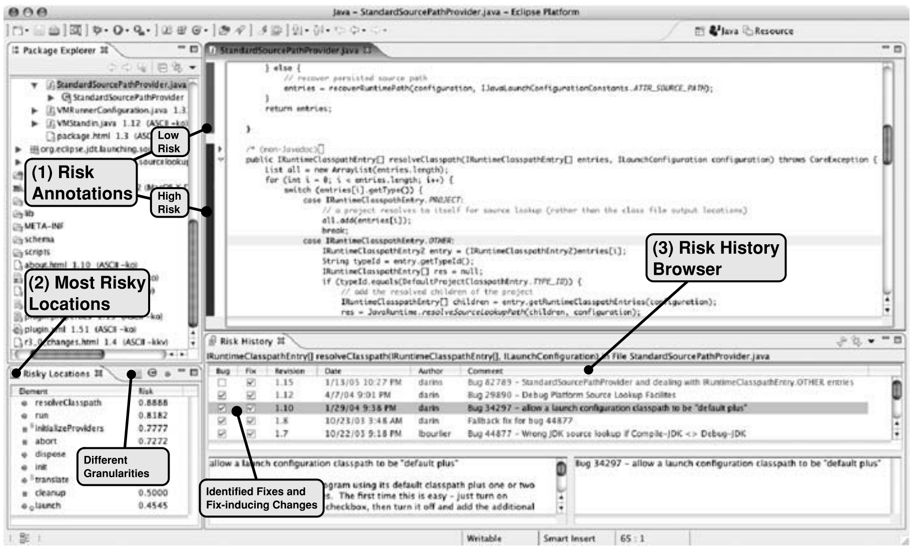
16.7.3 A Cache for Bugs

When a defect is found, it is likely that related locations also contain defects. But what does "related" mean? In a 2007 study of the defect and change history of seven open-source projects, Kim et al. (2007) formulated four hypotheses on how individual defects may be related:

1. *Changed components*. If a component was changed recently, it will tend to show defects soon.
2. *New components*. If a component has been added recently, it will tend to show defects soon.
3. *Related time*. If a component showed a defect recently, it will tend to show other defects soon.
4. *Related location*. If a component showed a defect recently, "nearby" components will also tend to show defects soon. A "nearby" component is a logically coupled component—that is, a component that is frequently changed at the same time.

Based on these hypotheses, Kim et al. built a model that would help in predicting defects. The model is called "bug cache" because it "caches" the most defect-prone components. The basic idea is as follows: Starting with an empty cache, the model processes the event history. Every time an event is detected to satisfy one of the hypotheses, the concerned components are loaded into the cache; if the component already is in the cache, its entry is refreshed.

The cache only holds up to 10 percent of all components; if the cache is full, the least recently refreshed components are discarded. Thus, the cache always holds those components that are new, most frequently changed, most recently found to have a defect, or logically coupled with another component that satisfies one of these criteria. At any time, the cache holds those components predicted to be most defect prone. In their evaluation, Kim et al. found that the 10 percent of cached components accounted for 73–95 percent of all defects, which places it among the most precise defect predictors.



```
        } else {
            // recover persisted source path
            entries = recoverRuntimePath(configuration, IJavaLaunchConfigurationConstants.ATTR_SOURCE_PATH);
        }
        return entries;

    }

    /* (non-Javadoc)
    public IRuntimeClasspathEntry[] resolveClasspath(IRuntimeClasspathEntry[] entries, ILaunchConfiguration configuration) throws CoreException {
        List all = new ArrayList(entries.length);
        for (int i = 0; i < entries.length; i++) {
            switch (entries[i].getType()) {
                case IRuntimeClasspathEntry.PROJECT:
                    // a project resolves to itself for source lookup (rather than the class file output locations)
                    all.add(entries[i]);
                    break;
                case IRuntimeClasspathEntry.OTHER:
                    IRuntimeClasspathEntry2 entry = (IRuntimeClasspathEntry2)entries[i];
                    String typeId = entry.getTypeId();
                    IRuntimeClasspathEntry[] res = null;
                    if (typeId.equals(DefaultProjectClasspathEntry.TYPE_ID)) {
                        // add the resolved children of the project
                        IRuntimeClasspathEntry[] children = entry.getRuntimeClasspathEntries(configuration);
                        res = JavaRuntime.resolveSourceLookupPath(children, configuration);
```

Element	Risk
resolveClasspath	0.8888
run	0.8182
initializeProviders	0.7777
abort	0.7272
dispose	
init	
translate	
cleanup	0.5000
launch	0.4545

IRuntimeClasspathEntry[] resolveClasspath(IRuntimeClasspathEntry[], ILaunchConfiguration) in File StandardSourcePathProvider.java

Bug	Fix	Revision	Date	Author	Comment
☐	✓	1.15	1/13/05 10:27 PM	darins	Bug 82789 – StandardSourcePathProvider and dealing with IRuntimeClasspathEntry.OTHER entries
✓	✓	1.12	4/7/04 9:01 PM	darin	Bug 29890 – Debug Platform Source Lookup Facilities
✓	✓	1.10	1/29/04 9:38 PM	darin	Bug 34297 – allow a launch configuration classpath to be "default plus"
✓	✓	1.8	10/23/03 3:48 AM	darin	Fallback fix for bug 44877
✓	✓	1.7	10/22/03 9:18 PM	lbourlier	Bug 44877 – Wrong JDK source lookup if Compile-JDK <> Debug-JDK

allow a la[illegible]nch configuration classpath to be "default plus"

[illegible]gram using its default classpath plus one or two [illegible]s. The first time this is easy – just turn on [illegible] checkbox, then turn it off and add the additional

Bug 34297 – allow a launch configuration classpath to be "default plus"

FIGURE 16.3

HATARI warns against risky changes. When a change to a location is likely to introduce a defect, HATARI annotates the code with a red bar. The redder the bar, the higher the risk. *Source:* Śliwerski et al. (2005).

16.7.4 Recommendation Systems

Since both mining change and defect data as well as predictions can be made automatically, one could go and integrate the appropriate prediction right into the development environment. This is what Śliwerski et al. (2005b) did: They explored whether predictions could be made *live* while the programmer is working. Their resulting HATARI tool does this for the *risk of change*—that is, the risk that some change introduces a defect.

Figure 16.3 shows HATARI in action, integrated into the ECLIPSE programming environment. Code that is risky to change is annotated with a red bar (1); the redder the bar, the higher the risk of introducing a defect. Additionally, HATARI also lists the locations that are most risky to change (2), as well as the risk history—the list of all fixes and other changes applied to the current location.

In Figure 16.3, HATARI is showing the risk of an ECLIPSE method, namely `resolveClassPath()`. It is shown in deep red, because out of nine previous fixes, eight introduced a new defect, which places it among the most risky methods to change within ECLIPSE. If you attempt to change it, HATARI will warn you against it; indeed, HATARI means "risk" in Swahili.

16.7.5 A Word of Warning

Although automatic predictions are compelling, we should remind ourselves that predictions can only rely on correlations, not on *causations.* If a specific feature of process or product is related to defects, this does not mean that removing the feature will make the defect go away. The feature may just be a symptom of some underlying, not yet recognized factor, which will still persist even if the symptom is eliminated or worked around. Such a symptom is called "fool's gold," as it is easily mistaken for true gold—that is, the actual cause. Thus, with every correlation you observe, you should have a good *theory* on why this correlation might be a causation—and how removing the supposed cause would change the effect, too.

As an example, consider one of the first mining experiments that I conducted with my then Ph.D. student, Thomas Zimmermann. We wanted to know which ECLIPSE developers had the highest defect density in the code they produced. To our surprise, Erich Gamma, the master developer of ECLIPSE (and probably one of the most influential programmers and program architects on this planet), on a list of more than 50 developers, showed up as number 2. In other words, the code produced by Erich Gamma was the second most defect prone.

From this fact, one could deduce that Erich Gamma is a lousy programmer, producing code that is littered with defects. However, this is "fool's gold." The true cause, as he confirmed to me later when we were sitting together on a plane, is simply that within ECLIPSE development, there are a number of tasks that nobody wants to do, because they are too risky. These tasks end up on the desk of the most experienced developers, and the most risky tasks end up with the boss—that is, Erich Gamma himself. With this in mind, it is no surprise if you find that the most experienced developers make the most mistakes—this happens because they get the riskiest jobs, and this again is because they are experienced. Assigning risky tasks to novices would *increase* the overall risk, and this is why the relationship between experience and defects is only a correlation, not a causation. Whatever correlation an automated mechanism discovers and recommends, always perform such a causality check before basing decisions on it.

16.8 FIXING THE PROCESS

There are so many ways a program can go wrong—and so many ways the production process can be wrong, too. It may turn out that the software erroneously was not tested before release. It may be that the wrong version was shipped to customers. It may be that some critical part was not reviewed at all. If the history of the problem indicates there is something wrong with the process, go and fix it. To achieve perfection, never stop thinking about how you could improve the process.

Of course, to *improve* the process, you need to have the means to *measure* the state of the art. In this chapter, I have given you some ideas on what could be measured, and how this can be leveraged to reduce the risk of defects. The more data you gather, and the better the data are organized, the more you will be able to discover about your code's strengths and weaknesses and of your development process. Gather data on problems, tasks, defects, and effort as much as you can, and use it to continuously review and improve.

This approach—gathering as much data as you can—is also used in places where the highest quality standards are to be met. The following excerpt from Fishman (1996) tells how the people who build the space shuttle software achieve quality:

> *The database records when the error was discovered; what set of commands revealed the error; who discovered it; what activity was going on when it was discovered—testing, training, or flight. It tracks how the error was introduced into the program; how the error managed to slip past the filters set up at every stage to catch errors—Why wasn't it caught during design? During development inspections? During verification? Finally, the database records how the error was corrected, and whether similar errors might have slipped through the same holes in the filters.*

All of this is being leveraged to find out how the error came to be—whether by a programmer or as the result of a flaw in the process. That is, the goal is not just to find errors in the code, but eventually *errors in the process.* This leads to a very disciplined way of building software. As Fishman (1996) explains:

> *The most important things the shuttle group does—carefully planning the software in advance, writing no code until the design is complete, making no changes without supporting blueprints, keeping a completely accurate record of the code—are not expensive. The process isn't even rocket science. It's standard practice in almost every engineering discipline except software engineering.*

BUG STORY 11

Typing "reboot" Reboots Your Phone

One of the first releases of Google's Android mobile phone operating system exhibited an unusual "root-console" problem. Google engineers had implemented a way to attach a remote device over the serial port, thus allowing to execute commands for diagnostic and debugging purposes. However, when there was no device attached, the phone would just use the built-in keyboard instead. Accidentally, this feature was left active in the production release.

As a consequence, any text that people typed (entering a text message, or composing email, for instance) was also executed as shell commands with superuser privileges. The defect became apparent when one user exchanged text messages with his girlfriend. She asked him: "What did you do in the last minutes?" He replied "reboot". Lo and behold, this command was interpreted and executed by the shell, effectively rebooting the phone, to the great astonishment of the user.

The user was lucky. If he had typed, say "rm -rf .", all of his data would have been erased. Fortunately, ordinary users rarely type this kind of thing. The Unix command "yes", though, puts the computer into a busy loop, eating battery and memory—and that is a word users indeed type from time to time.

16.9 CONCEPTS

How To

To learn from mistakes, use the *problem database* to check for frequently fixed code and frequent types of errors.

To map defects to components, relate problems from the problem database to changes in the version archive.

To reduce the risk of errors in specification, improve its quality assurance, increase its precision, and increase its automation.

To reduce the risk of errors in the code, do the same as for specifications; in addition, reduce complexity, improve documentation, set up assertions, and improve training.

To reduce the risk of errors in quality assurance, improve your test suite; test early, test often; review your code; improve analysis tools; and calibrate coverage metrics.

To allocate quality-assurance resources wisely, focus on the components determined as most defect prone in the past, and the components that are most similar to these.

The VULTURE tool predicts security vulnerabilities for a component, based on its set of *imports*.

The *bug cache* model retains components that are most defect prone because they are new, most frequently changed, most recently found to have a defect, or logically coupled with another component that satisfies one of these criteria.

Recommendation systems like HATARI integrate mining and prediction right into the programming environment.

When detecting a correlation between product and/or process features, make sure it is a causation, too.

16.10 FURTHER READING

Mining version and problem archives to uncover defect patterns is a subject that has recently received a lot of attention. Researchers are currently applying these techniques to open-source version and problem archives. For up-to-date information, see the working conference on *mining software repositories (MSR)*. Several of the papers and studies discussed in this chapter originated at this conference. For a lightweight introduction to the field, see the special edition of IEEE Software on Mining Software Repositories (Nagappan et al., 2009) as well as the overview in (Nagappan et al., 2008).

At its core, mining software repositories is a subject of *empirical software engineering,* an attempt to make software development more systematic by means of empirical observations, laws, and theories. The book by Endres and Rombach (2003) is an excellent introduction to the subject, listing the most important empirical findings.

Humphrey (1996) introduces the *personal software process,* a technique to measure and record what you do during software development—from lines of code produced per unit time to the time spent watching sports games. Of course, you also track any mistakes you make. By correlating these data, you find out how to improve your personal development process.

The Empirical Software Engineering and Measurement group at Microsoft Research is at the forefront of research when it comes to predicting software quality. Have a look at its homepage at *http://research.microsoft.com/en-us/projects/esm/*.

The article by Fishman (1996) on how the space shuttle software people write their software is a must-read for anyone who is interested in learning from mistakes. It is available online at *http://www.fastcompany.com/online/06/writestuff.html*.

EXERCISES

The IBUGS repository at *www.ibugs.org/* contains *bug repositories* extracted from project history. Each bug (enclosed in `<bug>···</bug>`), among other characteristics, lists the file(s) where the bug was fixed, enclosed in `<fixedFiles>···</fixedFiles>`. Each <file> has a name attribute listing the file that was fixed.

A `property name="severity"` attribute lists the severity of the bug, from enhancement over minor, normal, and major to critical.

16.1 *Find the most defect-prone files.* By parsing the ASPECTJ IBUGS repository, create a top-five list of the most defect-prone files in AspectJ—that is, those five files in which the most defects were fixed. Ignore all enhancement bugs. Does Pareto's law hold for AspectJ, too?

16.2 *Find the most defect-prone directories.* Every file is part of a directory, which represents a higher-level component of a system. By parsing the IBUGS data, create a top-five list of the most defect-prone directories in AspectJ—that is, those directories in which the most defects were fixed. Again, ignore all enhancement bugs.

Note: Be sure to count every bug only once per directory. That is, a single bug of which the fix affects `foo/a.c` and `foo/b.c` should be counted only once for `foo/`.

16.3 *Hypothesize.* In your own words, give an assessment of what makes these files or directories the most defect prone. Use the supplied source code and bug reports as additional information.

Resolve then, that on this very ground,
with small flags waving and tinny blasts on tiny trumpets,
we shall meet the enemy, and not only may he be ours,
he may be us.

— WALTER CRAWFORD KELLY
The Pogo Papers (1953)

Il faut imaginer Sysyphe heureux.

— ALBERT CAMUS
Le Mythe de Sysyphe (1942)