CHAPTER

Tracking Problems

2

This chapter deals with the issue of how to *manage* problems as reported by users: how to track and manage problem reports, how to organize the debugging process, and how to keep track of multiple versions. All of this constitutes the basic framework within which debugging takes place.

2.1 OH! ALL THESE PROBLEMS

Technically, a defect is created at the moment the programmer writes the appropriate code. However, its actual life cycle begins when some human spots the defect itself or one of its effects. Frequently, the human is a user, and the effect is a problem that needs to be solved.

Solving a user's problem is more than just debugging. At the start, we need to find out whether we can actually do something. Maybe the problem is a simple misunderstanding, or is caused by external components we cannot control. At the end, it does not suffice to fix the defect in our production code. To solve the user's problem, we also need to deploy the fix. All these steps involved in solving the problem need to be organized and managed. The life cycle of a software problem—from the first occurrence of a problem to its fix—can be summarized in the following five steps:

1. The user *informs* the vendor about the problem.
2. The vendor *reproduces* the problem.
3. The vendor *isolates* the problem circumstances.
4. The vendor *locates* and *fixes* the defect locally.
5. The vendor *delivers* the fix to the user.

As a programmer, you can be involved in each of these steps. First, you may always take the role of a user (for instance, when testing software or when working with third-party components). Later, you may be involved with reproducing, isolating, locating, and fixing the problem—the core of debugging. Finally, you may even be involved with delivering the fix.

Unless you are a one-person company with an elephant memory, this life cycle must be organized in some way. As a manager, you must be able to answer questions, such as:

- *Which problems are currently open?* An open problem indicates that there is probably some defect in the software that must be fixed.
- *Which are the most severe problems?* Typically, the most severe problems are the first to get fixed.
- *Did similar problems occur in the past?* If there were similar problems, there may be a solution at hand that need only to be delivered.

Furthermore, the *user* may want to know the state of his or her problem, and be quickly informed about any progress made. Our challenge is thus:

HOW CAN WE ORGANIZE THE LARGE-SCALE DEBUGGING PROCESS?

2.2 REPORTING PROBLEMS

To fix a problem, the developer must first be able to *reproduce* it. Otherwise, there would be no way of ascertaining further details about the problem. Worse, there would be no way of determining if the problem were fixed. The information required for this task is called a *problem report* (PR), also known as a *change request* (CR) or simply *bug report*. What goes into a problem report? The basic principle of problem reports is:

State all relevant facts.

Here, *relevant* means "relevant for reproducing the problem"—that is, *state the problem from the developer's perspective.* However, determining the relevant facts can impose problems. How can the user know what is relevant or not?

Real-life problem reports frequently include *too much*, such as gigantic core dumps, entire hard disk content, and even entire drives. However, they may not include *enough:* "Your program crashed. Just wanted to let you know" (A problem report about the GNU DDD debugger I received in 1999). Let us therefore explore which *specific facts* typically should be included in every problem report.

2.2.1 Problem Facts

In 2008, students of Saarland University conducted a survey across 156 experienced APACHE, ECLIPSE, and MOZILLA developers, trying to find out on what makes a good bug report. Their study found a frequent "mismatch between what developers consider most helpful and what users provide," and also found that for developers, *facts about the problem* are by far the most important.

The problem history

This is a description of what has to be done to reproduce the problem, as a minimal set of steps necessary. Typically, this also includes any accessed resources, such as input or configuration files:

```
   1. Start the  Preview application.
   2. Using “Open File”, open the attached document “Book.pdf”.
```

In the study by Bettenburg et al., the problem history was considered the most important fact: If the problem cannot be reproduced, it is unlikely to be fixed. If you can, have another user or tester repeat and review the steps described here.

If you can, *simplify* the problem as much as possible. If your database fails (say, on a 1,000-line SQL statement), the chances of getting the problem fixed are low—simply because such SQL statements do not occur this frequently in practice. But if you can simplify the SQL statement to three lines, such that the problem still persists, you're likely to get heard. Likewise, if certain circumstances are crucial for the problem to occur, be sure to include them. This gives developers a head start in debugging. (See Chapters 5 and 13 for more on this issue, especially on automating these steps.)

Diagnostic information as produced by the program

Some programs collect information for diagnostic purposes while executing. If the error occurs after a long series of events, it is often difficult for the user to retrace all steps from the program invocation to the failure. Therefore, the program can also be set up to *record* important events in a *log file*, which can later be forwarded to, and examined and reproduced by, the vendor. (In Chapter 8, we will learn more about log files. Section 11.4 in Chapter 11 has more ideas on information that can be collected and sampled from users.)

If a program *crashes,* the operating system is frequently able to produce a *stack trace,* a list of the functions active at the time of the program crash:

```
Thread 0 Crashed:
0  libSystem.B.dylib         0x95fef4a6 mach_msg_trap + 10
1  libSystem.B.dylib         0x95ff6c9c mach_msg + 72
2  com.apple.CoreFoundation  0x952990ce CFRunLoopRunSpecific + 1790
3  com.apple.CoreFoundation  0x95299cf8 CFRunLoopRunInMode + 88
4  com.apple.HIToolbox       0x92638480 RunCurrentEventLoopInMode
                               + 283
5  com.apple.HIToolbox       0x92638299 ReceiveNextEventCommon + 374
6  com.apple.HIToolbox       0x9263810d BlockUntilNextEventMatching
                               ListInMode + 106
7  com.apple.AppKit          0x957473ed _DPSNextEvent + 657
8  com.apple.AppKit          0x95746ca0 - [NSApplication nextEvent
                               MatchingMask:untilDate:inMode
                               :dequeue:] + 128
9  com.apple.AppKit          0x9573fcdb -[NSApplication run] + 795
10 com.apple.AppKit          0x9570cf14 NSApplicationMain + 574
11 com.apple.Preview         0x000024ea start + 54
```

Not very surprising, the study of Bettenburg et al. found *diagnostic information* like logs and stack traces to be the second most important information in a bug report.

A description of the experienced *behavior*

These are the *symptoms* of the problem—that is, what has happened in contrast to the expected behavior. For example:

```
The program crashed.
```

As you are the bearer of bad news, it is important to remain *neutral.* Humor, sarcasm, or attacks will divert developers from their goal, which is increasing the product quality. Just stay with the facts.

In the survey by Bettenburg et al., the experienced behavior was also rated important by developers, because it frequently "mimics steps to reproduce the bug."

A description of the expected *behavior*

This describes what should have happened according to the user. For example:

```
The program should have opened the file, or reported an error.
```

In the study, developers considered the expected behavior to be of less importance than steps to reproduce and diagnose information, possibly because the expected behavior is generally just the opposite of the (reported) experienced behavior.

A one-line summary

The one-line summary captures the essence of the problem. It is typically the basis for deciding the severity of the problem—that is, its impact on customers—and, consequently, the priority by which the problem will get fixed. For example:

```
Preview crashes when opening PDF document
```

2.2.2 Product Facts

All the facts discussed so far refer directly refer to the problem. The second category of facts refers to the *product* as well as the *environment* in which the problem occurred.

The product release

This is the version number or some other unique identifier of the product. Using this information, the developer can recreate a local copy of the program in the version as installed in the user's environment. For example:

```
Preview  4.1 (469.2.2), Build Info Preview-4690202~2
```

The operating environment

Typically, this is version information about the operating system. As problems may occur because of interactions between third-party components, information about such components may be useful as well. For example:

```
Mac OS X 10.5.5 (9F33)
```

Again, if you can, try to generalize. Does the problem occur under different operating environments, too? In our case, for instance, you might wish to check alternate operating systems or alternate printers.

The system resources

Some problems occur only under limited resources. Therefore, it is advisable to include information about your system's memory, disk space, or other hardware resources.

```
Model: MacBook1,1, BootROM MB11.0061.B03, 2 processors, Intel Core Duo, 2 GHz, 2 GB
Graphics:kHW_IntelGMA950Item, GMA 950, spdisplays_builtin, spdisplays_integrated_vram
Memory Module: BANK 0/DIMM0, 1 GB, DDR2 SDRAM, 667 MHz
Memory Module: BANK 1/DIMM1, 1 GB, DDR2 SDRAM, 667 MHz
AirPort: spairport_wireless_card_type_airport_extreme (0x168C, 0x86), 1.4.8.0
Bluetooth: Version 2.1.0f17, 2 service, 1 devices, 1 incoming serial ports
Serial ATA Device: Hitachi HTS722020K9SA00, 186,31 GB
Parallel ATA Device: MATSHITADVD-R   UJ-857
USB Device: Built-in iSight, Micron, high_speed, 500 mA
USB Device: HUAWEI Mobile, HUAWEI Technologies, full_speed, 500 mA
USB Device: Apple Internal Keyboard / Trackpad, Apple Computer, full_speed, 500 mA
USB Device: Bluetooth USB Host Controller, Apple, Inc., full_speed, 500 mA
USB Device: IR Receiver, Apple Computer, Inc., full_speed, 500 mA
```

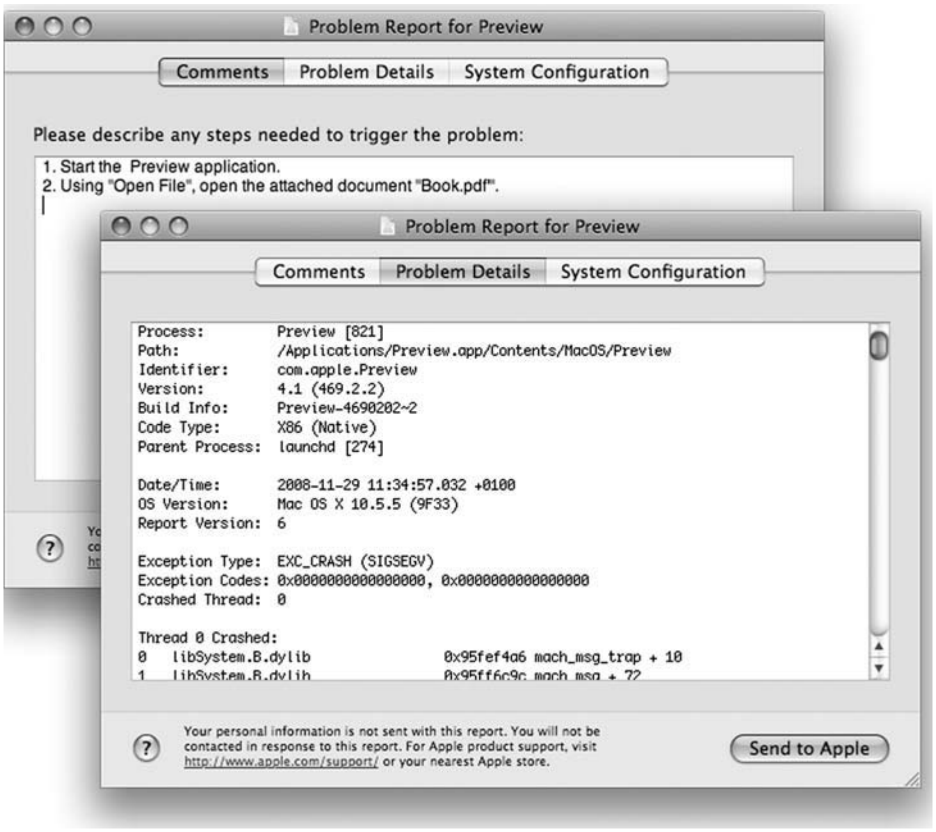
Looking on how system information impacts the failure can be helpful. If you can (for instance, if you are a tester), try to *generalize*. Does the problem occur under alternate releases, too? Does it occur in the most recent version?

Knowing details about the product and its environment can be crucial if the problem indeed depends on specific product or environment features. In the study by Bettenburg et al., though, developers considered these facts secondary:

Hardware, and to some degree, [OS] fields are rarely needed as most our bugs are usually found in all platforms.

2.2.3 Querying Facts Automatically

Facts on the product and the environment can easily be queried *automatically*, as products or systems include specific functionality or stand-alone tools to produce



Please describe any steps needed to trigger the problem:

1. Start the Preview application.
2. Using "Open File", open the attached document "Book.pdf".

```
Process:         Preview [821]
Path:            /Applications/Preview.app/Contents/MacOS/Preview
Identifier:      com.apple.Preview
Version:         4.1 (469.2.2)
Build Info:      Preview-4690202~2
Code Type:       X86 (Native)
Parent Process:  launchd [274]

Date/Time:       2008-11-29 11:34:57.032 +0100
OS Version:      Mac OS X 10.5.5 (9F33)
Report Version:  6

Exception Type:  EXC_CRASH (SIGSEGV)
Exception Codes: 0x0000000000000000, 0x0000000000000000
Crashed Thread:  0

Thread 0 Crashed:
0   libSystem.B.dylib               0x95fef4a6 mach_msg_trap + 10
1   libSystem.B.dylib               0x95ff6c9c mach_msg + 72
```

Your personal information is not sent with this report. You will not be contacted in response to this report. For Apple product support, visit http://www.apple.com/support/ or your nearest Apple store.

FIGURE 2.1

Mac OS talkback dialog. When a program fails, users can send relevant information to Apple.

standardized problem reports. Figure 2.1 shows the talkback dialog of Mac OS, which appears whenever a program crashes. Clicking on "Send to Apple" forwards all relevant information (shown in the Comments, Problem Details, and System Configuration fields) to the operating system vendor, who will in turn contact the developers.

Talkback-like tools typically also forward *internal* information to the vendor—for instance, a full *core dump*, which records the state of the program at the moment it failed. Such core dumps can be read in and examined by developers (see Section 8.3.3 in Chapter 8 for details on how to perform such postmortem debugging).

In all of these cases, the *privacy* of the user is an important issue. It is very advisable that the user be aware of whatever information is being sent to third parties (such as the vendor). This is not as much of a risk with manually written problem reports, but it becomes an increasing risk with information collected automatically.

Internal information (such as a core dump) cannot be interpreted by the user at all, and thus brings the greatest risk of an unwanted breach of privacy. In addition, log files about user interactions can be misused for all types of purposes, including third-party attacks. For these reasons, users should be made aware of any information your product may be collecting and forwarding. In addition, users should be able to turn off all recording features.

Privacy issues are less of a concern if a problem is discovered during testing. If an in-house tester finds a problem, she should make every effort to fix the problem. This includes recording and providing as much information as possible.

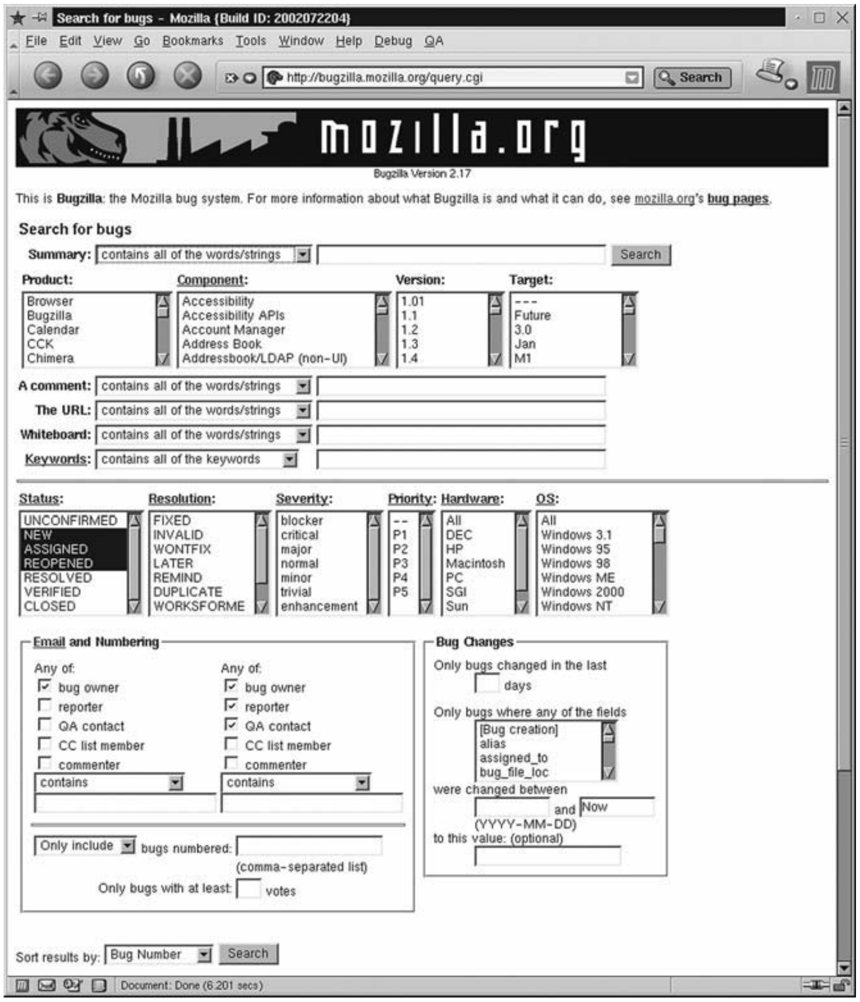
2.3 MANAGING PROBLEMS

Most developer teams keep track of the current problems in their system using a single "problem list" document that lists all open or unresolved problems to date. Such a document is easy to set up and easy to maintain. However, associated problems include the following:

- *Only one person at a time can work on the document.* Exception: The document is in a *version control system* that allows parallel editing and later merging.
- *History of earlier (and fixed) problems is lost.* Exception: The document is in a version control system and evolves together with the product.
- *Does not scale.* You cannot track hundreds of different issues in a simple text document.

The alternative to keeping a document is to use a *problem database*, which stores all problem reports. Problem databases scale up to a large number of developers, users, and problems.

Figure 2.2 shows an example of such a problem-tracking system. This is *BUGZILLA*, the problem-tracking system for the MOZILLA Web browser. BUGZILLA employs a Web browser as a user interface, which means that it can be accessed from anywhere (and by anyone, as MOZILLA is an open-source project). You can even install and adapt BUGZILLA for your own project. Note, though, that BUGZILLA (and other problem-tracking systems) is meant for developers, not for end users. Information provided from end users must be distilled and classified before it can be entered into the database.


Search for bugs - Mozilla {Build ID: 2002072204}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://bugzilla.mozilla.org/query.cgi
Search
mozilla.org
Bugzilla Version 2.17

This is **Bugzilla**: the Mozilla bug system. For more information about what Bugzilla is and what it can do, see mozilla.org's **bug pages**.

Search for bugs

Summary: contains all of the words/strings

Product:	Component:	Version:	Target:
Browser	Accessibility	1.01	---
Bugzilla	Accessibility APIs	1.1	Future
Calendar	Account Manager	1.2	3.0
CCK	Address Book	1.3	Jan
Chimera	Addressbook/LDAP (non-UI)	1.4	M1

A comment: contains all of the words/strings
The URL: contains all of the words/strings
Whiteboard: contains all of the words/strings
Keywords: contains all of the keywords

Status:	Resolution:	Severity:	Priority:	Hardware:	OS:
UNCONFIRMED	FIXED	blocker	--	All	All
NEW	INVALID	critical	P1	DEC	Windows 3.1
ASSIGNED	WONTFIX	major	P2	HP	Windows 95
REOPENED	LATER	normal	P3	Macintosh	Windows 98
RESOLVED	REMIND	minor	P4	PC	Windows ME
VERIFIED	DUPLICATE	trivial	P5	SGI	Windows 2000
CLOSED	WORKSFORME	enhancement		Sun	Windows NT

Email and Numbering

Any of:	Any of:
[✓] bug owner	[✓] bug owner
[ ] reporter	[✓] reporter
[ ] QA contact	[✓] QA contact
[ ] CC list member	[ ] CC list member
[ ] commenter	[ ] commenter
contains	contains

Only include bugs numbered: (comma-separated list)
Only bugs with at least votes

Bug Changes

Only bugs changed in the last days
Only bugs where any of the fields
[Bug creation]
alias
assigned_to
bug_file_loc
were changed between and Now
(YYYY-MM-DD)
to this value: (optional)

Sort results by: Bug Number Search

Document: Done (6.201 secs)

FIGURE 2.2

The BUGZILLA problem database. The database organizes all problem reports for MOZILLA.

2.4 CLASSIFYING PROBLEMS

Assume we want to report a problem in BUGZILLA (either because we are expert users and know how to enter a problem on a web site or because we are in charge of processing a user problem report). To report a problem, we must supply the required information (Section 2.2 has details on how to report problems) and *classify* the problem. The attributes BUGZILLA uses to classify problems, discussed in the following, are typical for problem-tracking systems.

2.4.1 Severity

Each problem is assigned a *severity* that describes the impact of the problem on the development or release process. BUGZILLA knows the following severity levels, from those with the greatest impact to those with the least.

- *Blocker:* Blocks development and/or testing work. This highest level of severity is also known as a *showstopper.*
- *Critical:* Crashes, loss of data, and severe memory leak.
- *Major:* Major loss of function.
- *Normal:* This is the "standard" problem.
- *Minor:* Minor loss of function, or other problem for which an easy workaround is present.
- *Trivial:* Cosmetic problem such as misspelled words or misaligned text.
- *Enhancement:* Request for enhancement. This means that the problem is not a failure at all, but rather a desired feature. Do not confuse this with missing functionality, though: If the product does not meet a requirement, this should be treated as a major problem.

Ideally, a product is not shipped unless all "severe" problems have been fixed—that is, major, critical, or blocker problems have been resolved, and all requirements are met. If a product is to be released at a fixed date, optional functions that still cause problems can be disabled.

The severity also determines our wording. In general, the word *problem* is just as a general term for a questionable property of the program run. A problem becomes a *failure* as soon as it is considered an incorrect behavior of the system. It is a *feature* if it is considered normal behavior ("It's not a bug, it's a feature!"). However, a *missing* or *incomplete* feature can also be a problem, as indicated by the *enhancement* category.

2.4.2 Priority

Each problem is assigned a specific *priority*. The higher the priority, the sooner the problem is going to be addressed. The priority is typically defined by the management. In fact, it is the *main means* for management to express what should be done first and what later. The importance of the priority attribute when it comes to control the process of development and problem solving cannot be overemphasized.

2.4.3 Identifier

Each problem gets a unique *identifier* (a *PR number*; in BUGZILLA, *bug number*) such that developers can refer to it within the debugging process—in emails, change logs, status reports, and attachments.

2.4.4 Comments

Every user and developer can attach *comments* to a problem report—for instance, to add information about the circumstances of the problem, to speculate about possible problem causes, to add first findings, or to discuss how the problem should be fixed.

2.4.5 Notification

Developers and users can attach their email address to a problem report. They will get notified automatically every time the problem report changes.

2.5 PROCESSING PROBLEMS

Assume that someone has entered a new problem report into the problem database. This problem report must now be *processed*. During this process, the problem report runs through a *life cycle* (Figure 2.3)—from `UNCONFIRMED` to `CLOSED`. The position in the life cycle is determined by the *state* of the problem report. These states are described in the following.

`UNCONFIRMED`: This is the state of any new problem report, as entered into the database. Let us introduce Olaf, who is a happy user of the Perfect Publishing Program—until it suddenly crashes. Thus:

1. Olaf reports the failure to Sunny at customer support.
2. Sunny enters the failure details as a new problem report into the problem database. She reports how Olaf can reproduce the failure, ascertains the relevant facts about Olaf's configuration, and sets the severity to "normal." She also reports Olaf's contact address.

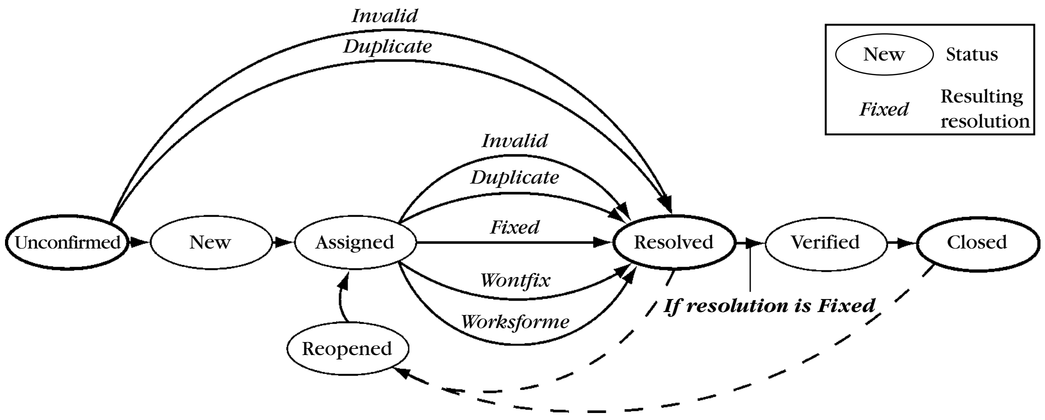


FIGURE 2.3

The life cycle of a problem in BUGZILLA. A problem starts `UNCONFIRMED`, is later `ASSIGNED` to a specific developer, and finally `CLOSED` with a specific resolution.

3. The problem gets a PR number (say, PR 2074). Its initial state is `UNCONFIRMED`. Nobody has yet tried to reproduce it.

`NEW`: The problem report is *valid*.

- It contains the relevant facts. (Otherwise, its *resolution* becomes `INVALID`. See material following.)
- It is not an obvious *duplicate* of a known problem. (Otherwise, its resolution becomes `DUPLICATE`. See material following.)

A `NEW` problem need not necessarily be reproducible. This is being taken care of in the remainder of the life cycle. In our example, programmer Violet may be in charge of checking problem reports. Finding that PR 2074 was not known before, she sets its state to `NEW`.

`ASSIGNED`: The problem is not yet resolved, but is already assigned to a developer (in BUGZILLA, to the *bug owner*). Mr. Poe, the manager, asks Violet to solve PR 2074. The state is now `ASSIGNED`.

`RESOLVED`: The problem is resolved. The *resolution* tells what has become (for now) of the problem report.

- `FIXED`: The problem is fixed.
- `INVALID`: The problem is not a problem, or does not contain the relevant facts.
- `DUPLICATE`: The problem is a duplicate of an existing problem.
- `WONTFIX`: The problem described is a problem that will never be fixed. This may also be the case for problems that turn out to be features rather than failures. The following is a `WONTFIX` example. The MOZILLA browser does not display ALT texts for images as tooltips, which many consider a problem. However, the MOZILLA developers say this behavior is mandated by Web standards and thus will not fix the "problem." (See bug #25537 at *www.bugzilla.mozilla.org*.)
- `WORKSFORME`: All attempts at reproducing this problem were futile. If more information appears later, the problem may be reopened. If the resolution is `FIXED`, the fix must be verified (state `VERIFIED`) and finally delivered (state `CLOSED`). In our example, let's assume Violet is unable to reproduce PR 2074 in her environment. In this case, the following happens:
  1. Violet documents her attempts in additional comments to the problem report, and sets the resolution to `WORKSFORME` and the state to `RESOLVED`. However, could it be that Olaf has the product configured to use the metric system? She asks Sunny whether she could get further data.
  2. Sunny requests further information from Olaf and sets the state of PR 2074 to `REOPENED`.
  3. Violet is still responsible for PR 2074 (state `ASSIGNED`). With the new data, she can finally reproduce and fix the problem. The state becomes `RESOLVED`; the resolution is `FIXED`.

`VERIFIED`: The problem is fixed. The fix has been verified as successful. The problem remains `VERIFIED` until the fix has been delivered to the user (for instance, by shipping a new release). Tester Klaus reviews Violet's fix. He gives his okay to integrate the fix in the current production release. The state of PR 2074 becomes `VERIFIED`.

`CLOSED`: A new release (or patch) of the product was shipped. In this release, the problem no longer occurs. As soon as the fix is delivered to Olaf, the PR 2074 state is `CLOSED`.

`REOPENED`: If a problem occurs again, it is assigned a state of `REOPENED` rather than `NEW`. It must be assigned again. In our example, if further users contact customer support about the problem, Sunny can look up the problem in the problem-tracking system and point them to the new release.

All of these states and resolutions can (and should) be adapted to the process at hand. If there is no independent verification, for example, reports skip the `VERIFIED` state. If problems are fixed at the user's site (skipping shipment), the `RESOLVED` and `CLOSED` states become synonyms. On the other hand, if additional clearance is required before a fix gets accepted this can be expressed by additional states and resolutions.

2.6 MANAGING PROBLEM TRACKING

A good problem-tracking system is the basis for all daily work on problems and failures. If nobody files problem reports, it is useless. If nobody marks problems as resolved, it will quickly be filled with outdated information. Therefore, the following issues should be resolved:

- *Who files problem reports?* This could be support personnel only. In general, though, it is probably useful if any developer can add new entries. Advanced users and beta testers may also be enabled to file problem reports.
- *Who classifies problem reports?* The *severity* of a problem can be extracted from the initial report. Sometimes, the severity is determined only after the problem could be reproduced.
- *Who sets priorities?* To determine the priority of a problem, management must assess the *impact* of a problem—that is, not only its severity but the following:
  - Its likelihood
  - The number of users affected
  - The potential damage

  Thus, the priority need not be correlated with the severity of a problem. A "blocker" problem in an alpha release may have lower priority than a "major" problem in a widely distributed product.

Many organizations use a *software change control board* (SCCB) to set priorities. This is a group of people who track problem reports and take care of their handling. Such a group typically consists of developers, testers, and configuration managers.

- *Who takes care of the problem?* All problem-tracking systems allow *assigning* problems to individual developers. This is also the task of an SCCB or like group.
- *Who closes issues?* This can be the SCCB or like group, the individual tester, or some quality assurance instance that verifies fixes (as in the scenario described previously).
- *What's the life cycle of a problem?* The BUGZILLA model, shown in Figure 2.3, is typical of problem databases but is by no means the only possible one. Depending on your individual needs, one can design alternate states and transitions. Problem-tracking systems may be configured to incorporate such processes.

Once a problem-tracking system is well integrated into your development process, you can use it to organize all aspects of software development. As it always gives you insights about the current state of the product, a problem-tracking system is a great management asset when it comes to decide whether a system is ready for release. A problem-tracking system can also be used for *historical* insights—that is, in hindsight, finding out what the most important problems were and how they could have been avoided. We will talk more about this role of problem tracking in Chapter 16.

2.7 REQUIREMENTS AS PROBLEMS

Problem-tracking systems are frequently used during the entire development of the product. In fact, they can be used from the very start, even when the product does not exist. In this setting, one enters *requirements* into the problem-tracking database, implying that each requirement not yet met is a problem. The severity of the problem indicates the importance of the requirement. Central requirements not yet met are marked as "major" problems. Minor or optional requirements could be marked as "requests for enhancement."

As requirements are typically broken down into subrequirements, the problem-tracking system should have a way of organizing problems *hierarchically.* That is, there should be a way of decomposing problems into subproblems, and of marking the problems as `FIXED` as soon as all subproblems are `FIXED`. In this fashion, requirement 1 is the product itself, and problem 1 thus becomes "the product is missing." As requirement 1 is broken down into a large number of individual features, so is the problem—with one subproblem for every feature. The product is ready for shipping when all problems are resolved—indicated by problem 1 being `FIXED`, which implies that all features are now implemented.

BUG STORY 3

Tracking Milk Issues at Microsoft

The following bug report is purported to originate from Microsoft's Excel group from 1994. Aliases have been removed. The *T:* indicates that the person was a tester, whereas *D:* stands for developer and *P:* for program manager.

```
------------------- ACTIVE - 05/12/94 - T:XXXXXX ---------------------
: Go to the kitchen
: Grab a Darigold chocolate milk carton
: Read the ingredients list

--! Either Darigold has discovered a chocolate cow, or something's
  missing from the ingredients list. It only lists milk, vitamin A,
  and vitamin D.  So where does the chocolate/sugar flavor come from?
------------------- ACTIVE - 05/12/94 - T:XXXXXX ---------------------
Moo info:
: Grab a Darigold 2% milk carton (NOT chocolate)
: Read the ingredients
--! Says it contains Cocoa, Sugar, Guar gum ...
Looks like the Chocolate
  and 2% ingredient lists have been swapped.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXXXXX -----------
looks like an internals problem?
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX --------------
UI Problem. I'll take it.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX --------------
They don't make milk at the Issaquah Darigold. Calling Ranier Ave.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX --------------
I can't repro. Do you have the wrong MILKINTL.DLL?
-------------- ASSIGNED to D:XXXXX - 05/12/94 - T:XXXXXXXX -----------
By design? I think new US health labeling went into effect this month.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX --------------
Wrong Department. Transferred from Distribution to Production.
Left voice mail for "Frank".
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX --------------
Reproduces in the Development Kitchen. Need a native
build of the Kitchen ...
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXXX --------------
This is a feature. IntelliSense labeling knew that you didn't want to feel
guilty about the chocolate in the milk, so it didn't list it on the box.
-------------- ASSIGNED to D:XXXXX - 05/12/94 - D:XXXX --------------
Recommend postpone. Reading the ingredients is not a common user
scenario ...
-------------- RESOLVED - WON'T FIX - 05/12/94 - P:XXXXX -------------
Fixing the package is just a band-aid. We need to come up with a solution
that addresses the real problem in 96. My recommendation is
chocolate cows.

Please close and assign to DARIGOLD.
```

A good problem-tracking system can *summarize* the problem database in the form of statistics (how many problems are still open, how many problems are being assigned to whom, and so on). A problem-tracking system may thus become the *primary tool* for organizing the debugging process, or the development process in general. The key is that managers check the problem-tracking system for outstanding issues, and assign appropriate priorities for developers. Eventually, your problem-tracking database will be a universal tool for resolving all types of problems—even those only remotely related to the product at hand (see Bug Story 3).

2.8 MANAGING DUPLICATES

If your problem-tracking system is well used, you may experience a metaproblem: *a lot of problem reports.* In September 2003 the MOZILLA problem database listed roughly 8,300 `UNCONFIRMED` problems waiting to be assigned and resolved.

One reason for such a large number of problem reports is *duplicates.* If your program has several users and a defect, chances are that several users will experience similar failures. If all of these users send in problem reports, your problem-tracking system will quickly fill up with similar reports, all related to the same defect.

For instance, if your Web browser crashes whenever it tries to render a dropdown list users will send in problem reports: "I opened Web page X, and it crashed," "I opened Web page Y, and it crashed," and so on. Processing all of these reports and finding out that each of the mentioned Web pages includes a dropdown list takes a lot of time.

A partial solution to this problem is to *identify duplicates.* The idea is that within the problem-tracking system one can mark problems as a *duplicate* of another problem. Submitters of new problem reports can then be encouraged to search the problem database for similar problems first. If a similar problem is found, the new problem can be marked as a duplicate. When the initial problem is fixed, the developer can close the duplicates where the same failure cause occurs. Unfortunately, it is not always easy to spot duplicates. This is due to two conflicting goals:

- A problem report includes *as many facts* as possible, in that any of them may be relevant for reproducing the problem.
- Identifying duplicates requires *as few facts* as possible, because this makes it easier to spot the similarity.

The solution here is *simplification*—that is, to simplify a problem report such that only the relevant facts remain. We will come back to this issue in Section 5.1, on simplifying problems. Automated methods are presented in Chapter 5.

Even if all duplicates have been resolved, however, your database will eventually fill up with unresolved problem reports—problems that could not be reproduced, problems that may be fixed in a later version, and low-priority problems. This is less a problem of space or performance (modern databases can easily cope with millions of problem reports) than of *maintenance*, as your developers wade again and again through this swamp of unresolved bugs. Having thousands of unresolved problems is also bad for morale. The solution is to clean up the database by searching for *obsolete* problems. A problem report could be declared obsolete if, for instance:

- The problem will never be fixed—for instance, because the program is no longer supported.
- The problem is old and has occurred only once.
- The problem is old and has occurred only internally.

Obsolete problem reports should show up only if explicitly searched for. If they ever show up again, you can recycle them by making them nonobsolete. In BUGZILLA, problems that will never be fixed are assigned a `WONTFIX` resolution.

2.9 RELATING PROBLEMS AND FIXES

Few products ever came out as a single version. Thus, whenever users report a problem they must state the version of the product in which the problem occurred. Without this, developers will not know which version to use when trying to reproduce the problem.

But even *with* a version identifier, are you prepared to access the specific version as released to your user? This means not only the binaries as shipped, but every source that went into this specific release and all tools in their specific versions that were used to produce the binary. This is one of the key issues of *software configuration management*: to be able to recreate any given configuration any time.

To keep track of versions and configurations is something far too error prone to be done manually. Instead, use a *version control system* to support the task. Using version control has few costs and many benefits. *Not* using version control, though, makes your development process risky, chaotic, and generally unmanageable.

So, how do you manage your versions in order to track bugs? Simple: Whenever a new version is shipped, mark its source base with an appropriate *tag*. Use this tag to recreate the source base, and together with the source base, the product itself.

Another good thing about version control systems is the management of *branches* for storing *fixes*. The basic idea is to separate the evolution into two parts:

- A *main trunk* in which new features are tested and added.
- *Branches* in which fixes (and only fixes) are added to stable versions.

This separation allows vendors to ship out new fixed versions of stable products while developing new features for the next major release. The use of tags and branches is illustrated in Figure 2.4.

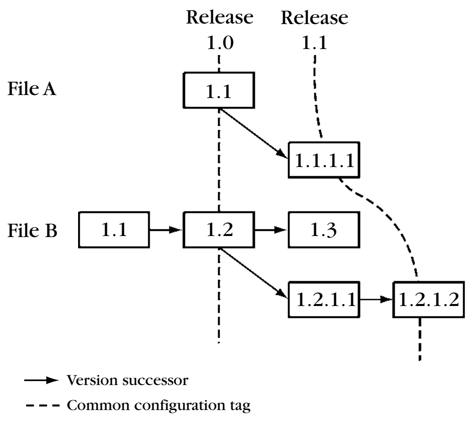


FIGURE 2.4

Tags and branches in version control. As a system progresses, individual releases are tagged such that they can be reconstructed on demand.

A product consists of two files, which have evolved independently into multiple revisions. Consider Figure 2.4. The initial release of the system, indicated by the dotted line, consisted of revision 1.1 of file A and revision 1.2 of file B. In the version control system, this is reflected by an appropriate tag of these revisions. Since then, file B has evolved into revision 1.3 with new features, but is as yet untested.

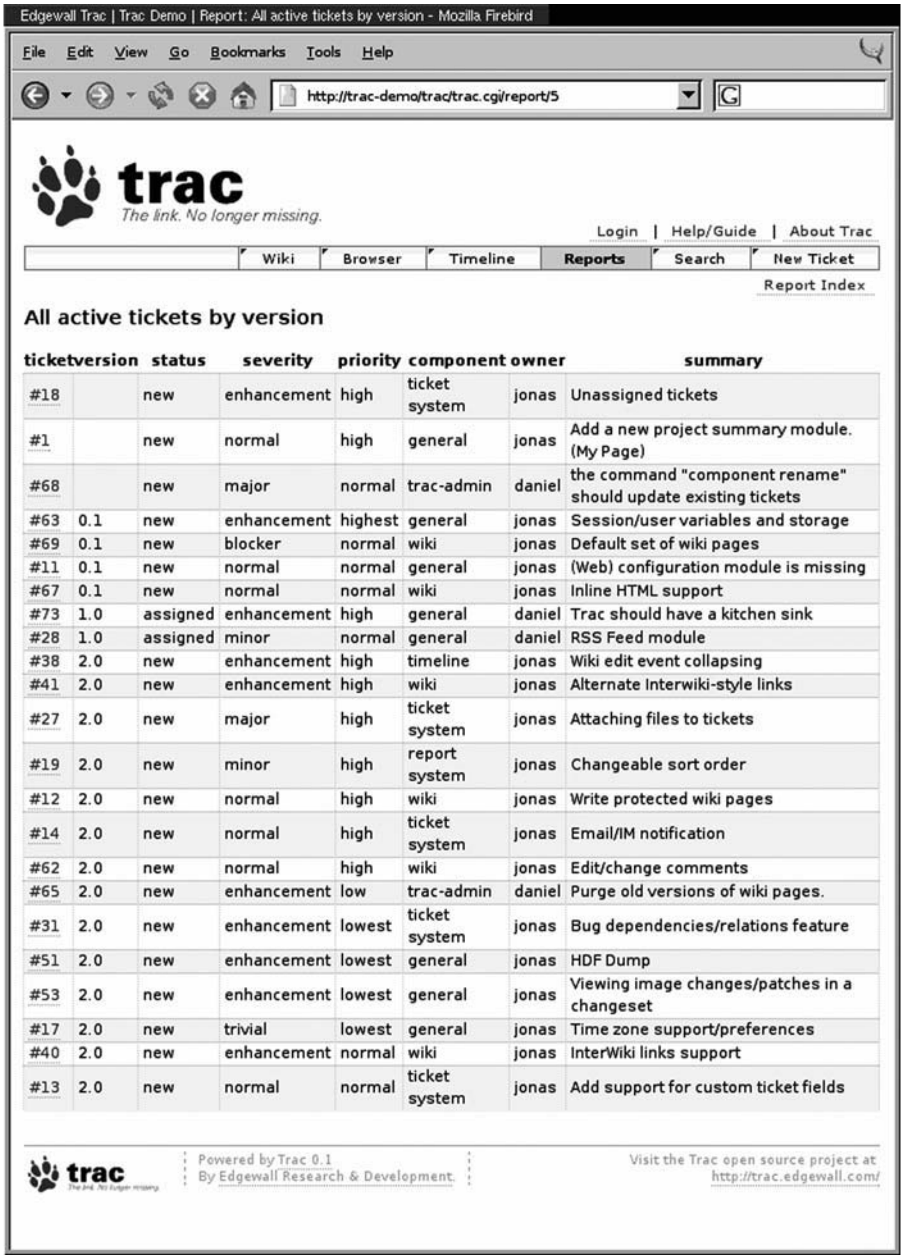
When a user now reports a problem that calls for a fix, what do we do? Do we ship the new, untested configuration with file B included? Usually not. Instead, based on the initial release, we create a *branch* for files A and B, containing the new versions 1.1.1.1 and 1.2.1.1, respectively. This branch holds only the indispensable *fixes* for the initial release, which can then be released as minor service updates (shown as release 1.1 in the figure). New, risky features still go to the main trunk. Eventually, all fixes from the branch can be *merged back* into the main trunk such that they will also be present in the next major releases. As maintainers routinely want to check whether a certain problem has been fixed in a specific release, be sure to relate problem reports to changes as follows:

- Within the problem-tracking system, identify the change in version control that fixed the problem. For instance, attach the branch identifier as a comment to the problem report:

```
Problem fixed in RELEASE_1_1_BRANCH
```

- Within the version control system, identify the problem report number that is addressed by a specific change. This can be done in the log message given when the change is committed:

```
Fix: null pointer could cause crash (PR 2074)
```


Edgewall Trac | Trac Demo | Report: All active tickets by version - Mozilla Firebird
File Edit View Go Bookmarks Tools Help
http://trac-demo/trac/trac.cgi/report/5
trac
The link. No longer missing.
Login | Help/Guide | About Trac
Wiki
Browser
Timeline
Reports
Search
New Ticket
Report Index
Powered by Trac 0.1
By Edgewall Research & Development.
Visit the Trac open source project at http://trac.edgewall.com/

All active tickets by version

ticket	version	status	severity	priority	component	owner	summary
#18		new	enhancement	high	ticket system	jonas	Unassigned tickets
#1		new	normal	high	general	jonas	Add a new project summary module. (My Page)
#68		new	major	normal	trac-admin	daniel	the command "component rename" should update existing tickets
#63	0.1	new	enhancement	highest	general	jonas	Session/user variables and storage
#69	0.1	new	blocker	normal	wiki	jonas	Default set of wiki pages
#11	0.1	new	normal	normal	general	jonas	(Web) configuration module is missing
#67	0.1	new	normal	normal	wiki	jonas	Inline HTML support
#73	1.0	assigned	enhancement	high	general	daniel	Trac should have a kitchen sink
#28	1.0	assigned	minor	normal	general	daniel	RSS Feed module
#38	2.0	new	enhancement	high	timeline	jonas	Wiki edit event collapsing
#41	2.0	new	enhancement	high	wiki	jonas	Alternate Interwiki-style links
#27	2.0	new	major	high	ticket system	jonas	Attaching files to tickets
#19	2.0	new	minor	high	report system	jonas	Changeable sort order
#12	2.0	new	normal	high	wiki	jonas	Write protected wiki pages
#14	2.0	new	normal	high	ticket system	jonas	Email/IM notification
#62	2.0	new	normal	high	wiki	jonas	Edit/change comments
#65	2.0	new	enhancement	low	trac-admin	daniel	Purge old versions of wiki pages.
#31	2.0	new	enhancement	lowest	ticket system	jonas	Bug dependencies/relations feature
#51	2.0	new	enhancement	lowest	general	jonas	HDF Dump
#53	2.0	new	enhancement	lowest	general	jonas	Viewing image changes/patches in a changeset
#17	2.0	new	trivial	lowest	general	jonas	Time zone support/preferences
#40	2.0	new	enhancement	normal	wiki	jonas	InterWiki links support
#13	2.0	new	normal	normal	ticket system	jonas	Add support for custom ticket fields

FIGURE 2.5

Open issues as reported by the TRAC system. The report shows which problems persist for which version of the product.

Such a relationship between problem tracking and version control works best when it is established *automatically.* Some version control systems integrate with problem-tracking systems such that the relationship between problems and fixes is maintained automatically. This allows for queries to determine which problem reports have occurred or fixed in which release.

As an example, consider the report of the TRAC system, shown in Figure 2.5. TRAC is a lightweight system that integrates version control and problem tracking. Its report shows which problems still persist in which version of the product.

2.10 RELATING PROBLEMS AND TESTS

Many developers use problem-tracking systems not only for problems as reported by end users but also for problems encountered in-house. That is, as soon as a developer stumbles across a problem, he or she reports it just as if an end user had told him or her about the problem.

In principle, this is a good thing, in that no problem remains unreported. However, the main way in which developers find problems is by *testing* the program, and this induces a conflict between test outcomes and problem reports. Should a failing test be tracked in the problem database? And if so, how are we going to *synchronize* the problem database with the test outcomes?

Unless you have a problem-tracking system that neatly integrates with your test suite, I recommend keeping test outcomes separate from problem reports. There are several reasons for doing so:

- Test outcomes occur frequently—possibly (and hopefully) far more frequently than user's problem reports. Storing these test outcomes in the problem database would quickly flood the database—and divert from the actual problems in the field.
- If you use automated testing (see Chapter 3), you can at any time check test outcomes for any version, at the press of a button. Thus, there is no need for storing that information.
- Suppose a test fails. If you can find the defect and fix it right away, there is no need to enter a record in the problem-tracking system.

All of this boils down to one point: *Test cases make problem reports obsolete.* If a problem occurs during development, do not enter it into the problem-tracking system. Instead, write a *test case* that exposes the problem. This way, the test case will show that the problem is still present, and you can always check for the problem by running the test.

You *can* always use the problem-tracking system, though, for storing *ideas* and *feature requests*—that is, for anything that does not immediately translate into code or a test case. As soon as you start to implement the feature request, start writing a test case that exposes the lack of the feature, and close the entry in the problem-tracking system. Once the feature is implemented, the successive test case can be used to determine whether it meets expectations.

2.11 CONCEPTS

Reports about problems encountered in the field are stored in a *problem database* and are classified with respect to status and severity.

A problem report must contain all information that is *relevant to reproduce the problem.*

How To

To obtain the relevant problem information, set up a standard set of items that users must provide (see also Section 2.2, on reporting problems). This includes:

- Problem history
- Diagnostic information
- Expected behavior
- Experienced behavior
- Product release
- Operating environment

To write an effective problem report, make sure the report:

- Is *well structured*
- Is *reproducible*
- Has a descriptive *one-line summary*
- Is as *simple* as possible
- Is as *general* as possible
- Is *neutral* and stays with the facts

For details, see Section 2.2.

Products can be set up to *collect and forward information* that may be relevant to reproduce the problem. Be aware of privacy issues, though.

A typical life cycle of a problem starts with a status of `UNCONFIRMED`. It ends with `CLOSED` and a specific *resolution* such as `FIXED` or `WORKSFORME` (Figure 2.3).

To organize the debugging process, have a *software change control board* that uses the problem database to:

- Keep track of resolved and unresolved problems
- Assign a priority to individual problems
- Assign problems to individual developers

To track requirements, one can also use a problem-tracking system. Each requirement not yet met is a problem.

Keep problem tracking simple. If it gets in the way, people won't use it.

To restore released versions, use a version control system to *tag* all configurations as they are released to users.

To separate fixes and features, use a version control system to keep fixes in branches and features in the main trunk.

To relate problems and fixes, establish conventions to *relate problem reports to changes*, and vice versa (Section 2.9). Advanced version control systems integrate with problem-tracking systems to maintain this relationship automatically.

To relate problems and tests, *make a problem report obsolete* as soon as a test case exists. When a problem occurs, prefer writing test cases to entering problem reports.

2.12 TOOLS

BUGZILLA. The BUGZILLA problem-tracking system can be found at *http://www.bugzilla.org/*.

Its specific incarnation for MOZILLA is available for browsing at *http://bugzilla.mozilla.org/*.

PHPBUGTRACKER. PHPBUGTRACKER is a lightweight problem-tracking system that is simpler to install and manage than BUGZILLA. It can be found at *http://phpbt.sourceforge.net/*.

ISSUETRACKER. Like PHPBUGTRACKER, ISSUETRACKER aims to be a lightweight problem tracker, with a special focus on being user friendly. If you want to toy with a problem-tracking system at your site, PHPBUGTRACKER or ISSUETRACKER might be your choice. ISSUETRACKER can be found at *http://www.issue-tracker.com/*.

TRAC. TRAC is another lightweight problem-tracking system. Its special feature is that it integrates with version control. This makes it easier to track problems across multiple versions. Just as PHPBUGTRACKER and ISSUETRACKER, TRAC is open-source software, but with optional commercial support. TRAC can be found at *http://trac.edgewall.org/*.

SOURCEFORGE. The SOURCEFORGE project provides automated project organization beyond simple problem tracking. It includes facilities such as discussion forums, public version archives, user management, and mailing lists. It is available to open-source projects. A simple registration suffices, and there is no installation hassle. SOURCEFORGE is also available as a commercial version to be installed at other sites and to manage commercial projects. SOURCEFORGE can be found at *http://www.sourceforge.net/*.

GFORGE. GFORGE is a fork of the original SOURCEFORGE code. Just like SOURCEFORGE, it provides problem tracking, discussion forums, public version archives, user management, mailing lists, and much more. In contrast to SOURCEFORGE, the GFORGE people do not host projects. Instead, you install the

GFORGE software at your site. (Commercial support is available.) If you want a single open-source package that manages the entire development life cycle, GFORGE delivers. GFORGE can be found at *http://www.gforge.org/*.

2.13 FURTHER READING

Regarding problem-tracking systems, there is not too much information available except from those provided by vendors. Mirrer (2000) addresses the issue of obsolete test cases. For him, organizing a problem-tracking system is like "organizing your socks": Once in a while, an overflowing sock drawer has to be cleaned up.

Kolawa (2002) comments on the relationship between problem-tracking systems and testing. He states that problem-tracking systems "should be used exclusively as a place to store feedback when you cannot immediately modify the code." Otherwise, you should create a reproducible test case.

Advanced problem-tracking systems can do an even better job of integrating with version control systems. The *Software Configuration Management FAQ* posting of the `comp.software.config-mgmt` newsgroup contains a large list of problem-tracking systems and their integration within software configuration management. The newsgroup can be found at *http://www.daveeaton.com/scm/*.

The award-winning study of Bettenburg (2008) was one of the first to survey what developers need in a bug report; it is worth a read. It is available at the ACM digital library at *http://doi.acm.org/10.1145/1453101.1453146*.

EXERCISES

2.1 Write a bug report for the `sample` problem (Section 1.1). Justify the amount of information you gave.

2.2 Visit the MOZILLA problem-tracking site at *http://bugzilla.mozilla.org/*, and answer the following questions:

(a) How many problems have been entered as `NEW` into BUGZILLA in the past three days?
(b) How many of these are critical (or even blocking)?
(c) How many of these are invalid? Why?
(d) How many unresolved or unconfirmed problems are there in the currently released version?
(e) Which is the worst currently unresolved problem?
(f) According to problem priority, which problem should you address first as a programmer?

2.3 What are the major differences between a dedicated problem-tracking system such as BUGZILLA and a general organizing and messaging tool such as Microsoft Outlook?

2.4 Which other problems (in addition to software) could be managed using a problem-tracking system?

Six Stages of Debugging

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?