CHAPTER 5

Software Metrics Data Collection

We saw in Chapter 3 that measurements should be tied to an organization, project, product and process goals, so that we know what questions we are trying to answer with our measurement program. Chapter 4 showed us how to frame questions as hypotheses, so that we can perform measurement-based investigations to help us understand the answers. But having the right measures is only part of a measurement program. Software measurement is only as good as the data that are collected and analyzed. In other words, we cannot make good decisions with bad data.

> Data should be collected with a clear purpose in mind. Not only a clear purpose but also a clear idea as to the precise way in which they will be analysed so as to yield the desired information. … It is astonishing that men, who in other respects are clear-sighted, will collect absolute hotchpotches of data in the blithe and uncritical belief that analysis can get something out of it.
>
> MORONEY 1962, P. 120

In this chapter, we consider what constitutes good data, and we present guidelines and examples to show how data collection supports decision making. In particular, we focus on the terminology and organization of data related to software quality, including faults and failures. We also discuss issues related to collecting data on software changes, effort, and productivity.

5.1 DEFINING GOOD DATA

It is very important to assess the quality of data and data collection before data collection begins. Your measurement program must specify not only what metrics to use, but what precision is required, what activities and time periods are to be associated with data collection, and what rules govern the data collection (such as whether a particular tool will be used to capture the data). Of critical importance is the definition of the metric. Terminology must be clear and detailed, so that all involved understand what the metric is and how to collect it.

There are two kinds of data with which we are concerned. As illustrated by Figure 5.1, there is raw data that results from the initial measurement of process, product, or resource. But there is also a refinement process, extracting essential data elements from the raw data so that analysts can derive values about attributes.

To see the difference, consider the measurement of developer effort. The raw effort data may consist of weekly time sheets for each staff member working on a project. To measure the effort expended on the design so far, we must select all relevant time sheets and add up the figures. This refined data are a direct measurement of effort. But we may derive other measures as part of our analysis: for example, average effort per staff member, or effort per design component.

Deciding what to measure is the first step. We must specify which direct measures are needed, and also measures that may be derived from the direct ones. Sometimes, we begin with the derived measures. From a goal, question, metric (GQM) analysis, we understand which derived measures we want to know; from those, we must determine which direct measures are required to calculate them.

Most organizations are different, not only in terms of their business goals but also in terms of their corporate cultures, development preferences, staff skills, and more. So, a GQM analysis of apparently similar projects may result in different metrics in different companies. That is exactly why GQM is preferable to adopting a one-size-fits-all standard measurement set. Nevertheless, most organizations share similar problems. Each

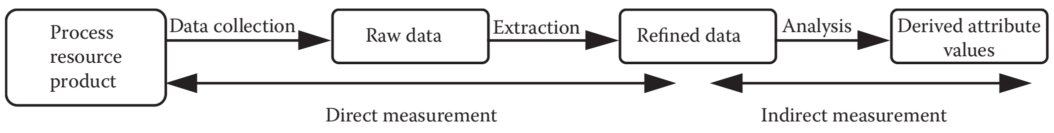


FIGURE 5.1 The role of data collection in software measurement.

is interested in software quality, cost, and schedule. As a result, most developers collect metrics information about quality, cost, and duration. In the next section, we shall learn about the importance of data definition by examining possible definitions for attributes and measures related to the reporting of problems that occur in software development.

5.2 DATA COLLECTION FOR INCIDENT REPORTS

No software developer consistently produces perfect software the first time. Thus, it is important for developers to measure aspects of software quality. Such information can be useful for determining

- How many problems have been found with a product?
- How effective are the prevention, detection, and removal processes?
- Whether the product is ready for release to the next development stage or to the customer?
- How the current version of a product compares in quality with previous or competing versions?

The terminology used to support this investigation and analysis must be precise, allowing us to understand the causes as well as the effects of quality assessment and improvement efforts. However, the use of terms varies widely among software professionals, and terms such as "error," "fault," "failure," and so forth are used inconsistently. For example, there is disagreement with the definition of a software *failure*.

From a formal computer science perspective, a failure is any deviation from *specified behavior*. That is, a failure occurs whenever the output does not match the specified behavior. Using the formal perspective, there is no failure when program behavior matches an incorrect specification even if the behavior does not make sense.

From an engineering perspective, a failure is any deviation from the required or *expected* behavior. Thus, if a program receives an unspecified input it should produce a "sensible" output appropriate for the circumstances.

EXAMPLE 5.1

Consider the following specification for a program:

> When a non-negative integer *n* is input into the factorial program, it shall produce *n*! as an integer string for values of *n* up to and including 149.

Table 5.1 gives interpretations of failure events caused by running the program. From a formal perspective, if the program input is an unspecified value (151 or "fred"), it is not a failure for the system to crash or wipe the file system. From an engineering perspective, these behaviors are deemed failures since such output is not sensible and can have catastrophic consequences.

5.2.1 The Problem with Problems

Figure 5.2 depicts some of the components of a problem's cause and symptoms, expressed in terms consistent with Institute of Electrical and Electronic Engineers (IEEE) standard 610.12 (IEEE 610.12-1990) and IEEE standard 1044-2009 (IEEE 1044-2009). A *fault* in a software product occurs due to a human error or mistake. That is, the fault is the encoding of the human error. For example, a developer might misunderstand a user-interface requirement, and therefore create a design that includes the misunderstanding. The design fault can also result in an incorrect code, as well as incorrect instructions in the user manual. Thus, a single error can result in one or more faults, and a fault can reside in any of the products of development.

However, a *failure* is the departure of a system from its required behavior. Failures can be discovered both before and after system

TABLE 5.1 Formal versus Engineering Interpretations of Failure Events for a Factorial Program

Input	Output	Formal Failure?	Engineering Failure?
150	"Too large a number"	No	No
151	System crash	No	Yes
"fred"	Delete all files	No	Yes

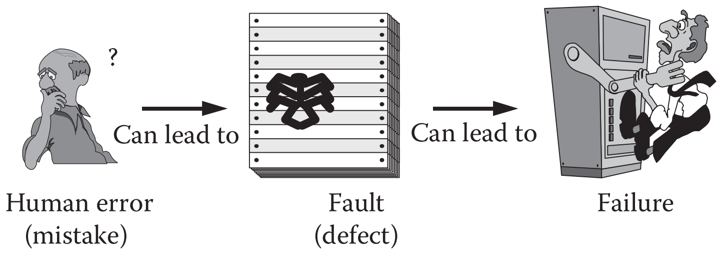


FIGURE 5.2 Software quality terminology. Terminology from the IEEE standard 1044-2009 is in parentheses when it differs from ours. (IEEE Standard 1044-2009: *Standard Classification for Software Anomalies*, IEEE Computer Society Press, 2009.)

delivery, as they can occur in testing as well as in operation. Since we take an engineering perspective to identify failures, we compare actual system behavior, rather than with specified behavior. Thus, faults in the requirements documents can result in failures, too.

In some sense, you can think of faults and failures as inside and outside views of the system. Faults represent problems that the developer sees, while failures are problems that the user sees. Not every fault corresponds to a failure, since the conditions under which a fault results in system failure may never be met.

EXAMPLE 5.2

The software in an actual nuclear reactor failed due to a fault that was caused by the process described below:

1. *Human error:* Failure to distinguish signed and absolute value numbers in an algorithm resulted in the following fault.
2. *Fault:* '$X := Y$' is coded instead of '$X := ABS(Y)$' which in turn led to the following failure.
3. *Failure:* Nuclear reactor shut down because it was wrongly determined that a meltdown was likely.

A failure may or may not occur depending on the sequence of inputs processed by the system—a fault may reside in a program segment that is never executed, thus preventing the processing error from occurring.

We need to introduce another stage to this failure process that of the software error. Ammann and Offutt define the term *software error* as "an incorrect internal state that is the manifestation of some fault" (Ammann and Offutt, 2008). This notion of error is commonly used in the software testing community, which is concerned with identifying software errors. In general, the triggering of a fault may not lead to instantaneous failure; rather, an erroneous state arises within the system which at some later time leads to failure. We can think of software errors as intermediate error states which when propagated through the system ultimately lead to failure. One example is an operating system fault that is triggered by a particular input that results in a state error that causes another state error in the word processor which then crashes. Thus, the fault starts a chain of software errors that ultimately result in a failure.

This notion of software error is also highly relevant for software *fault tolerance*, which is concerned with how to prevent failures in the presence of software errors. If an error state is detected and countered before it propagates to the output, the failure can be prevented. Preventing error states from propagating is the goal of software fault tolerance. Researchers continue to search for methods that enable software to operate dependably even when it contains faults that have been triggered. Software errors do not necessarily cause failures in fault-tolerant software.

Unfortunately, the terminology used to describe software problems is not uniform. If an organization measures software quality in terms of faults per thousand lines of code, it may be impossible to compare the result with the competition if the meaning of “fault” is not the same. The software engineering literature is rife with differing meanings for the same terms. Below are just a few examples of how researchers and practitioners differ in their usage of terminology.

- In many organizations, *errors* can mean faults. This meaning contrasts with the notion of *software error* as used in the software testing community—an invalid system state that results when a fault is triggered and may or may not lead to a failure.
- *Anomalies* usually mean a class of faults that are unlikely to cause failures in themselves but may nevertheless eventually lead to failures indirectly. In this sense, an anomaly is a deviation from the usual, but it is not necessarily wrong. For example, deviations from accepted standards of good programming practice (such as use of nonmeaningful names) are often regarded as anomalies. Note that the *IEEE Standard Glossary of Software Engineering Terminology* defines an anomaly as “anything observed in the documentation or operation of software that deviates from expectations” (IEEE 610.12-1990). Also the IEEE Standard Classification for Software Anomalies addresses both faults (defects) and failures (IEEE 1044-2009).
- *Defects* often refer collectively to faults and failures. However, the IEEE Standard 1044-2009 uses the term “defect” to refer only to faults (IEEE 1044-2009).
- *Bugs* refer to faults occurring in the code, but in some cases are used to describe failures.

- *Crashes* are a special type of failure, where the system ceases to function.

Since the terminology used varies widely, it is important for you to define your terms clearly, so that they are understood by all who must supply, collect, analyze, and use the data. Often, differences in meaning are acceptable, as long as the data can be translated from one framework to another.

We also need a good, clear way of describing what we do in reaction to problems. For example, if an investigation of a failure results in the detection of a fault, then we make a *change* to the product to remove it. A change can also be made if a fault is detected during a review or inspection process. In fact, one fault can result in multiple changes to one product (such as changing several sections of a piece of code) or multiple changes to multiple products (such as a change to requirements, design, code, and test plans).

In this book, we describe the observations of development, testing, system operation, and maintenance problems in terms of failures, faults, and changes. Whenever a problem is observed, we want to record its key elements, so that we can investigate the causes and cures. In particular, we want to know the following:

1. *Location*: Where did the problem occur?
2. *Timing*: When did it occur?
3. *Symptom*: What was observed?
4. *End result*: Which consequences resulted?
5. *Mechanism*: How did it occur?
6. *Cause*: Why did it occur?
7. *Severity:* How much was the user affected?
8. *Cost:* How much did it cost?

These eight attributes are similar to the failure attributes and defect (fault) attributes in the IEEE Standard Classification for Software Anomalies (IEEE 1044-2009). However, the IEEE standard for failure attributes does not separate *symptom* from *end result*; both are included in the general attribute called "description." The IEEE standard for defect (fault) attributes includes *symptom* under the "description" attribute, and *end result* under the "effect" attribute.

We use the above eight problem attributes because they are (as far as possible) mutually independent, so that proposed measurement of one does not affect measurement of another*; this characteristic of the attributes is called *orthogonality*. Orthogonality can also refer to a classification scheme within a particular category. For example, cost can be recorded as one of the several predefined categories, such as *low* (under $100,000), *medium* (between $100,000 and $500,000), and *high* (more than $500,000). However, in practice, attempts to over-simplify the set of attributes sometimes result in non-orthogonal classifications. When this happens, the integrity of the data collection and metrics program can be undermined, because the observer does not know in which category to record a given piece of information.

EXAMPLE 5.3

Some organizations try to provide a single classification for software faults, rather than using the eight components suggested above. Here, one attribute is sometimes regarded as most important than the others, and the classification is a proposed ordinal scale measure for this single attribute. Consider the following classification for faults, based on severity:

- Major
- Minor
- Negligible
- Documentation
- Unknown

This classification is not orthogonal; for example, a documentation fault could also be a major problem, so there is more than one category in which the fault can be placed. The nonorthogonality results from confusing severity with location, since the described fault is located in the documentation but has a severe effect.

When the scheme is not orthogonal, the developer must choose which category is most appropriate. It is easy to see how valuable information is lost or misrepresented in the recording process.

* This mutual independence applies only to the initial measurement. Data analysis may reveal, for example, that severity of effects correlates with the location of faults within the product. However, this correlation is discovered after classification, and cannot be assumed when faults are classified.

EXAMPLE 5.4

Riley described the data collection used in the analysis of the control system software for the Eurostar train (the high-speed train used to travel from Britain to France and Belgium via the Channel tunnel) (Riley 1995). In the Eurostar software problem-reporting scheme, faults are classified according to only two attributes, cause and category, as shown in Table 5.2. Note that "cause" includes notions of timing and location. For example, an error in software implementation could also be a deviation from functional specification, while an error in test procedure could also be a clerical error. Hence, Eurostar's scheme is not orthogonal and can lead to data loss or corruption.

On the surface, our eight-category report template should suffice for all types of problems. However, as we shall see, the questions are answered very differently, depending on whether you are interested in faults, failures, or changes.

5.2.2 Failures

A failure report focuses on the external problems of the system: the installation, the chain of events leading up to the failure, the effect on the user or other systems, and the cost to the user as well as the developer. Thus, a typical failure report addresses each of the eight attributes in the following way:

TABLE 5.2 Fault Classifications Used in Eurostar Control System

Cause	**Category**
Error in software design	Category not applicable
Error in software implementation	Initialization
Error in test procedure	Logic/control structure
Deviation from functional specification	Interface (external)
Hardware not configured as specified	Interface (internal)
Change or correction-induced error	Data definition
Clerical error	Data handling
Other (specify)	Computation
	Timing
	Other (specify)

Source: IEEE Standard 1044-2009: *Standard Classification for Software Anomolies*, IEEE Computer Society Press, 2009.

Failure Report

Location: Such as installation where failure was observed

Timing: CPU time, clock time, or some temporal measure

Symptom: Type of error message or indication of failure

End result: Description of failure, such as "operating system crash," "services degraded," "loss of data," "wrong output," and "no output"

Mechanism: Chain of events, including keyboard commands and state data, leading to failure

Cause: Reference to possible fault(s) leading to failure

Severity: Reference to a well-defined scale, such as "critical," "major," and "minor"

Cost: Cost to fix plus cost of lost potential business

Let us examine each of these categories more closely.

Location is usually a code (e.g., hardware model and serial number, or site and hardware platform) that uniquely identifies the installation and platform on which the failure was observed. The installation description must be interpreted according to the type of system involved. For example, if software is embedded in an automatic braking system, the "installation" may move about. Similarly, if the system is distributed, then the terminal at which a failure is observed must be identified, as well as the server to which it was online.

Timing has two, equally important aspects: real time of occurrence (measured on an interval scale), and execution time, up to the occurrence of failure (measured on a ratio scale).

The *symptom* category explains what was observed as distinct from the *end result*, which is a measure of the consequences. For example, the symptom of a failure may record that, the screen displayed a number that was one greater than the number entered by the operator; if the larger number resulted in an item's being listed as "unavailable" in the inventory (even though one was still left), that symptom belongs to the "end result" category.

End result refers to the consequence of the failure. Generally, the "end result" requires a (nominal scale) classification that depends on the type of system and application. For instance, the end result of a failure may be any of the following:

- Operating system crash
- Application program aborted
- Service degraded
- Loss of data
- Wrong output
- No output

Mechanism describes how the failure came about. This application-dependent classification details the causal sequence leading from the activation of the source to the symptoms eventually observed. This category may also characterize what function the system was performing or how heavy the workload was when the failure occurred. Unraveling the chain of events is a part of diagnosis, so often this category is not completed at the time the failure is observed.

Cause is also part of the diagnosis (and as such is more important for the fault form associated with the failure). Cause involves two aspects: the type of trigger and the type of source (i.e., the fault that caused the problem). The trigger can be one of several things, such as

- Physical hardware failure
- Operating conditions
- Malicious action
- User error
- Erroneous report

While the actual source can be faults such as these:

- Physical hardware fault
- Unintentional design fault
- Intentional design fault
- Usability problem

For instance, the cause can involve a *switch* or *case* statement with no code to handle the drop-through condition (the source), plus a situation (the trigger) where the listed cases were not satisfied.

The cause category is often cross-referenced to fault and change reports, so that the collection of reports paints a complete picture of what happened (the failure report), what caused it (the fault reports), and what was done to correct it (the change reports). Then, post-mortem analysis can identify the *root cause* of a fault. The analysis will focus on how to prevent such problems in the future, or at least catch them earlier in the development process.

Severity describes how serious the failure's end result was for the service required from the system. For example, failures in safety-critical systems are often classified for severity as follows (Riley 1995):

- *Catastrophic* failures involve the loss of one or more lives, or injuries causing serious and permanent incapacity.
- *Critical* failures cause serious and permanent injury to a single person but would not normally result in loss of life to a person of good health. This category also includes failures causing environmental damage.
- *Significant* failures cause light injuries with no permanent or long-term effects.
- *Minor* failures result neither in personal injury nor a reduction in the level of safety provided by the system.

Severity may also be measured in terms of cost to the user.

Cost to the system provider is recorded in terms of how much effort and other resources were needed to diagnose and respond to the failure. This information may be a part of diagnosis and therefore supplied after the failure occurs.

Sometimes, a failure occurs several times before it is recognized and recorded. In this case, an optional ninth category, *count*, can be useful. *Count* captures the number of failures that occurred within a stated time interval.

EXAMPLE 5.5

Early on December 31, 2008, Microsoft's first-generation Zune portable media players hung. This event was widely reported in the media since there were many thousands of affected users (Robertson 2009). The failure was traced to a fault in the program code related to date calculations—2008 is a leap year with 366 days, and the code used 365 days as the length of all years. The workaround was to wait until January 1, let the battery drain completely, and then restart the device (this workaround left the fault in place). Using our failure attributes, a possible failure report might look like the following:

- *Location*: Many first-generation Zune 30 media players in use around the world.
- *Timing*: December 31, 2008, starting early in the morning.
- *Symptom*: The device froze.
- *End result*: The device became unusable, even after a restart.
- *Mechanism*: Upon startup, the loading bar indicates "full," and then the device hangs.
- *Cause* (1): (Trigger) Starting the device on December 31, 2011.
- *Cause* (2): (Source type) Coding fault related to date calculation.
- *Severity*: Serious, as it made the device unusable until the inconvenient workaround was communicated to the large and diverse user community.
- *Cost*: Effort to diagnose the problem, develop and publicize a workaround, and repair the fault. Perhaps, the greatest cost was damage to the company reputation.

EXAMPLE 5.6

In the 1980s, problems with a radiation therapy machine were discovered, and the description of the problems and their software causes are detailed in an article by Leveson and Turner (1993). Mellor used the above framework to analyze the first of these failures, in which patients died as a result of a critical design failure of the Therac 25 radiation therapy machine (Mellor 1992). The Therac administered two types of radiation therapy: x-ray and electron. In x-ray mode, a high-intensity beam struck a tungsten target, which absorbed much of the beam's energy and produced x-rays. In electron mode, the Therac's computer retracted the metal target from the beam path while reducing the intensity of the radiation by a factor of 100. The sequence of a treatment session was programmed by an operator using a screen and keyboard; the Therac software then performed the treatment automatically. In each of the accidents that occurred, the electron beam was supposed to be at a reduced level but instead was applied full strength, without the tungsten target in place. At the same time, the message "Malfunction 54" appeared on the monitor screen. Diagnosis revealed that the use of the up-arrow key to correct a typing mistake by the operator activated a fault in the software, leading to an error in the system which scrambled the two modes of operation.

Applying our failure framework to the 1986 accident which caused the death, six months later, of Mr V. Cox, the failure report might look like this:

- *Location*: East Texas Cancer Center in Tyler, Texas, USA.
- *Timing* (1): March 21, 1986, at whatever the precise time that "Malfunction 54" appeared on the screen.
- *Timing* (2): Total number of treatment hours on all Therac 25 machines up to that particular time.
- *Symptom* (1) "Malfunction 54" appeared on the screen.
- *Symptom* (2) Classification of the particular program of treatment being administered, type of tumor, etc.
- *End result*: Strength of the beam too great by a factor of 100.
- *Mechanism*: Use of the up-arrow key while setting up the machine led to the corruption of a particular internal variable in the software.
- *Cause* (1) (Trigger) Unintentional operator action.
- *Cause* (2) (Source type) Unintentional design fault.
- *Severity*: Critical, as injury to Mr Cox was fatal.*
- *Cost*: Effort or actual expenditure by accident investigators.

In Leveson and Turner's post-mortem analysis, they identified several root causes of the failures including the following:

- "Overconfidence in the software" especially "among nonsoftware professionals."
- "Confusing reliability with safety." The software rarely failed. However, the consequences of failure were tragic.
- "Lack of defensive design."
- "Failure to eliminate root causes" of prior failures.
- "Complacency."
- "Unrealistic risk assessments." Software risk was not treated as seriously as hardware risks.
- "Inadequate investigation or follow-up on accident reports."
- "Inadequate software engineering practices."

Remember that only some of the eight attributes can usually be recorded at the time the failure occurs. These are:

- Location
- Timing
- Symptom
- End result
- Severity

* The failure is classed as "critical," rather than "catastrophic," in accordance with several safety guidelines that (unfortunately) require the loss of *several* lives for catastrophic failure.

The others can be completed only after diagnosis, including root-cause analysis. Thus, a data collection form for failures should include at least these five categories.

When a failure is closed, the precipitating fault in the product has usually been identified and recorded. However, sometimes there is no associated fault. Here, great care should be exercised when closing the failure report, so that readers of the report will understand the resolution of the problem. For example, a failure caused by user error might actually be due to a usability problem, requiring no immediate software fix (but perhaps changes to the user manual, or recommendations for enhancement or upgrade). Similarly, a hardware-related failure might reveal that the system is not resilient to hardware failure, but no specific software repair is needed.

Sometimes, a problem is known but not yet fixed when another, similar failure occurs. It is tempting to include a failure category called "known software fault," but such classification is not recommended because it affects the orthogonality of the classification. In particular, it is difficult to establish the correct timing of a failure if one report reflects multiple, independent events; moreover, it is difficult to trace the sequence of events causing the failures. However, it is perfectly acceptable to cross-reference the failures, so the relationships among them are clear.

The need for cross-references highlights the need for forms to be stored in a way that allows pointers from one form to another. The storage system must support changes to stored data. For example, a failure may initially be thought to have one fault as its cause, but subsequent analysis reveals otherwise. In this case, the failure's "type" may require change, as well as the cross-reference to other failures.

The form storage scheme must also permit searching and organizing. For example, we may need to determine the first failure due to each fault for several different samples of trial installations. Because a failure may be a first manifestation in one sample, but a repeat manifestation in another, the storage scheme must be flexible enough to handle this.

5.2.3 Faults

A failure reflects the user's view of the system, but a fault is seen only by the developer. Thus, a fault report is organized much like a failure report but has very different answers to the same questions. It focuses on the internals of the system, looking at the particular module where the fault occurred and the cost to locate and fix it. A typical fault report interprets the eight attributes in the following way:

Fault Report

Location: Within system identifier, such as module or document name

Timing: Phases of development during which fault was created, detected and corrected

Symptom: Type of error message reported, or activity which revealed fault (such as review)

End result: Failure caused by the fault

Mechanism: How source was created, detected, and corrected

Cause: Type of human error that led to fault

Severity: Refer to severity of resulting or potential failure

Cost: Time or effort to locate and correct; can include analysis of cost had fault been identified during an earlier activity

Again, we investigate each of these categories in more detail.

In a fault report, *Location* tells us which product (including both identifier and version) or part of the product (subsystem, module, interface, document) contains the fault. The IEEE Standard Classification for Software Anomalies (IEEE 1044-2009) provides location attributes in four levels:

1. *Asset*, which identifies the application with the fault.
2. *Artifact*, for example, a specific code file, requirements document, design specification, test plan, test case.
3. *Version detected*, which identifies the version where the fault was found.
4. Version corrected.

Timing relates to the three events that define the life of a fault:

1. When the fault is created
2. When the fault is detected
3. When the fault is corrected

Clearly, this part of a fault report will need revision as a causal analysis is performed. It is also useful to record the time taken to detect and correct the fault, so that product maintainability can be assessed.

The *Symptom* classifies what is observed during diagnosis or inspection. The 1993 draft version of the IEEE standard on software anomalies (IEEE 1044-2009) provides a useful and extensive classification that we can use for reporting the symptom. We list the categories in the following table:

Classification of Fault Types

- Logic problem
  - Forgotten cases or steps
  - Duplicate logic
  - Extreme conditions neglected
  - Unnecessary function
  - Misinterpretation
  - Missing condition test
  - Checking wrong variable
  - Iterating loop incorrectly
- Computational problem
  - Equation insufficient or incorrect
    - Missing computation
    - Operand in equation incorrect
    - Operator in equation incorrect
    - Parentheses used incorrectly
  - Precision loss
    - Rounding or truncation fault
    - Mixed modes
  - Sign convention fault
- Interface/timing problem
  - Interrupts handled incorrectly
  - I/O timing incorrect
    - Timing fault causes data loss
  - Subroutine/module mismatch
    - Wrong subroutine called
    - Incorrectly located subroutine call
    - Nonexistent subroutine called
    - Inconsistent subroutine arguments
- Data-handling problem
  - Initialized data incorrectly
  - Accessed or stored data incorrectly
    - Flag or index set incorrectly
    - Packed/unpacked data incorrectly

continued

(continued) Classification of Fault Types

- Referenced wrong data variable
- Data referenced out of bounds

Scaling or units of data incorrect

Dimensioned data incorrectly

- Variable-type incorrect
- Subscripted variable incorrectly

Scope of data incorrect

Data problem

- Sensor data incorrect or missing
- Operator data incorrect or missing
- Embedded data in tables incorrect or missing
- External data incorrect or missing
- Output data incorrect or missing
- Input data incorrect or missing

Documentation problem

- Ambiguous statement
- Incomplete item
- Incorrect item
- Missing item
- Conflicting items
- Redundant items
- Confusing item
- Illogical item
- Nonverifiable item
- Unachievable item

Document quality problem

- Applicable standards not met
- Not traceable
- Not current
- Inconsistencies
- Incomplete
- No identification

Enhancement

- Change in program requirements
  - Add new capability
  - Remove unnecessary capability
  - Update current capability
- Improve comments
- Improve code efficiency
- Implement editorial changes
- Improve usability
- Software fix of a hardware problem
- Other enhancement

(continued) Classification of Fault Types
Failure caused by a previous fix
Other problems

Source: IEEE Standard 1044-2009, *Standard Classification for Software Anomolies*, IEEE Computer Society Press, 2009.

The *End result* is the actual failure caused by the fault. If separate failure or incident reports are maintained, then this entry should contain a cross-reference to the appropriate failure or incident reports.

Mechanism describes how the fault was created, detected, and corrected. Creation explains the type of activity that was being carried out when the fault was created (e.g., specification, coding, design, maintenance). Detection classifies the means by which the fault was found (e.g., inspection, unit testing, system testing, integration testing), and correction refers to the steps taken to remove the fault or prevent the fault from causing failures.

Cause explains the human error (mistake) that led to the fault. Although difficult to determine in practice, the cause is classified in terms of the suspected causes such as lost information, requirements misunderstanding, management not taking engineering concerns seriously, and so forth.

Severity assesses the impact of the fault on the user. That is, severity examines whether the fault can actually be evidenced as a failure, and the degree to which that failure would affect the user.

The *Cost* explains the total cost of the fault to the system provider. Much of the time, this entry can be computed only by considering other information about the system and its impact.

The optional *Count* field can include several counts, depending on the purpose of the field. For example, count can report the number of faults found in a given product or subsystem (to gauge inspection efficiency), or the number of faults found during a given period of operation (to assist in reliability modeling).

EXAMPLE 5.7

We reexamine the problem with the Zune media player described in Example 5.5. The fault causing the failure was quickly identified as a coding error (Zuneboards, 2008), and was used as an example by Weimer, Forrest, Le Goues, and Nguyen in a study of automated program repair (Weimer et al. 2010).

- *Location*: Module rtc.c, Convert Days function, lines 249–275.
- *Timing*: Created during coding, detected during operational use.
- *Symptom*: Missing condition test causing a loop to iterate incorrectly (nontermination).
- *End result*: The device froze.
- *Mechanism*: Creation: during code development; Detection: diagnosis of operational failure; Correction: workaround provided, code correction probably done.
- *Cause*: Human mistake in dealing with a special case—leap years.
- *Severity*: Serious, as all of the first-generation Zune devices froze.
- *Cost*: Minimal cost to diagnose, prepare workaround, and repair; however, there was significant cost with respect to the reputation of the company.

EXAMPLE 5.8

We return to the Therac problem described in Example 5.6 (Leveson and Turner 1993). The fault causing the failure that we discussed may be reported in the following way:

- *Location*: Product is the Therac 25; the subsystem is the control software. The version number is an essential part of the location. The particular module within the software is also essential.
- *Timing*: The fault was created at some time during the control software's development cycle.
- *Symptom*: The category and type of software fault that were the root cause of the failure.
- *End result*: "Malfunction 54," together with a radiation beam that was very strong.
- *Mechanism*: Creation: during code development; Detection: diagnosis of operational failure; Correction: the immediate response was to remove the up-arrow key from all machines and tape over the hole!
- *Cause*: This is discussed at length by Leveson and Turner (1993).
- *Severity*: Critical,[*] in spite of the fact that, not all of the "Malfunction 54" failures led to injury.
- *Cost*: The cost to the Therac manufacturers of all investigations of all the failures, plus corrections.

---

[*] Again, the failure is not catastrophic, because only one life was lost.

EXAMPLE 5.9

The popular open source problem tracking tool Bugzilla, which describes both failures and faults as "bugs," supports recording the eight problem attributes either directly or indirectly as follows:

- *Location*: Location attributes include product, component, version, and platform.
- *Timing*: Timing attributes include dates for when a bug is reported and when an artifact is modified.
- *Symptom, end result, and mechanism*: These attributes are not separated. They can be recorded under the category "steps to reproduce."
- *Cause*: Cause can be recorded as "additional information."
- *Severity*: Severity is directly supported.
- *Cost*: Cost of fault diagnosis and repair is recorded under "time tracking."

5.2.4 Changes

Once a failure is experienced and its cause determined, the problem is fixed through one or more changes. These changes may include modifications to any or all of the development products, including the specification, design, code, test plans, test data, and documentation. Change reports are used to record the changes and track the products most affected by them. For this reason, change reports are very useful for evaluating the most fault-prone modules, as well as other development products with unusual numbers of defects. A typical change report may look like this:

Change Report

Location: Identifier of document or module changed

Timing: When the change was made

Symptom: Type of change

End result: Success of change, as evidenced by regression or other testing

Mechanism: How and by whom change was performed

Cause: Corrective, adaptive, preventive, or perfective

Severity: Impact on the rest of the system, sometimes as indicated by an ordinal scale

Cost: Time and effort for change implementation and test

The *Location* identifies the product, subsystem, component, module, or subroutine affected by a given change. *Timing* captures when the change was made, while *End result* describes whether the change was successful or not. (Sometimes changes have unexpected effects and have to be redone; these problems are discovered during regression or specialized testing.)

The *IEEE Standard Glossary of Software Engineering Terminology* defines four types of maintenance activities, which are classified by the reasons for the changes (IEEE 1990). A change may be *corrective maintenance*, in that the change corrects a fault that was discovered in one of the software products. It may be *adaptive maintenance*: the underlying system changes in some way (the computing platform, network configuration, or some part of the software is upgraded), and a given product must be adapted to preserve functionality and performance. Developers sometimes make *perfective* changes, refactoring code to make it more adaptable by removing "bad smells" (Fowler 1999), rewriting documentation or comments, and/or renaming a variable or routine to clarify the system structure so that new faults are not likely to be introduced as part of other maintenance activities. Finally, *preventive maintenance* involves removing anomalies that represent potential faults. The *Cause* entry in the change report is used to capture one of these reasons for change: corrective, adaptive, perfective, or preventive or perfective.

The *Cost* entry explains the cost to the system developer of implementing a change. The expense includes not only the time for the developer to find and fix the system but also the cost of doing regression tests, update documentation, and return the system to its normal working state.

A *Count* field may be used to capture the number of changes made in a given time interval, or to a given system component.

5.3 HOW TO COLLECT DATA

Since the production of software is an intellectual activity, the collection of data requires human observation and reporting. Managers, systems analysts, programmers, testers, and users must record raw data on forms. This manual recording is subject to bias (deliberate or unconscious), error, omission, and delay. Automatic data capture is therefore desirable, and sometimes essential, such as in recording the execution time of real-time software.

Unfortunately, in many instances, there is no alternative to manual data collection. To ensure that the data are accurate and complete, we must plan our collection effort before we begin to measure and capture data. Ideally, we should do the following:

- Keep the procedures simple.
- Avoid unnecessary recording.
- Train staff in the need to record data and in the procedures to be used.
- Provide the results of data capture and analysis promptly to the original providers and in a useful form that will assist them in their work.
- Validate all data collected at a central collection point.

The quality of collected fault and failure data can vary. Often key data are missing.

EXAMPLE 5.10

Open source projects are potentially a rich source of data for empirical studies. They often include source code, test cases, and change log files. The GNU project coding standards say that you should "keep a change log to describe all the changes made to program source code" (GNU 2013). Yet, a study of three open-source projects—GNUJSP, GCC-g++, and Jikes—by Chen, Schach, Yu, Offutt, and Heller found that overall 22% of the changes in project files were not recorded in the change logs. In one of the versions of Jikes, 62% of the changes were not recorded (Chen et al. 2004).

In a similar study, Bachmann and Bernstein examined data from five open-source and one closed-source project (Bachmann and Bernstein 2009). They found that most of the fault reports are not linked to an entry in the change logs.

Clearly, we must check the integrity of the data collected when analyzing project data whether it is from an open-source or proprietary projects. This requires careful planning.

Planning for data collection involves several steps. First, you must decide which products to measure, based on your GQM analysis. You may need to measure several products that are used together, or you may measure one part or subsystem of a larger system.

EXAMPLE 5.11

Ultimately, failures of all types have to be traced to some system component, such as a program, function, unit, module, subsystem, or the system itself. If the measurement program is to enable management to take action to prevent problems, rather than waiting for problems to happen, it is vital that these components be identified at the right level of granularity. This ability to focus on the locus of a problem is a critical factor for your measurement program's success. For example, software development work at a large British computer manufacturer was hampered by not having data amalgamated at high levels. The software system had a large number of very small modules, each of which had no faults or very few faults. At this level of granularity, it was impossible to identify trends in the fault locations. However, by combining modules according to some rule of commonality (e.g., similar function, same programmer, or linkage by calling routines), it may have been possible to see patterns not evident at lower levels. In other words, it was of little use to know that each program contained either 0 or 1 as known fault, but it was of great interest to see, for example, a set of 25 programs implementing a single function that had 15 faults, whereas a similar set of programs implementing another function had no faults. Such information suggested to management that the first, more fault-prone function, be subject to greater scrutiny.

In Example 5.11, the level of the granularity of collection was too fine. Of course, lowering the granularity can be useful, too. Suppose, your objective is to monitor the fault density of individual modules. That is, you want to examine the number of faults per thousand lines of code for each of a given set of modules. You will be unable to do this if your data collection forms associate faults only with subsystems, not with individual modules. In this case, the level of granularity in your data collection is too *coarse* for your measurement objectives. Thus, determining the level of granularity is essential to planning your data collection activities.

The next step in planning data collection is making sure that the product is under configuration control. We must know which version(s) of each product we are measuring.

EXAMPLE 5.12

In measuring reliability growth, we must decide what constitutes a "baseline" version of the system. This baseline will be the system to which all others will be compared. To control the measurement and evaluate reliability over time, the changed versions must have a multi-level version numbering scheme, including a "mark" number that changes only when there is a major functional enhancement. Minor version numbers track lesser changes, such as the correction of individual faults.

The GQM analysis suggests which attributes and measures you would like to evaluate. Once you are committed to a measurement program, you must decide exactly which attributes to measure and how derived measures will be derived. This will determine what raw data will be collected, and when.

EXAMPLE 5.13

We saw in Chapter 3 that Barnard and Price used GQM to determine what measures they wanted to investigate in evaluating inspection effectiveness (Barnard and Price 1994). They may have had limited resources, so they may have captured only a subset of the metrics initially, focusing first on the ones supporting the highest-priority goals. Once their metrics were chosen, they defined carefully exactly which direct and derived measures were needed. For example, they decided that defect-removal efficiency is the percentage of coding faults found by code inspections. To calculate this derived metric, they needed to capture two direct metrics: total faults detected at each inspection, and the total coding faults detected overall. Then, they defined an equation to relate the direct measures to the derived one:

$$Defect\_removal\_efficiency = 100 \times \frac{\sum_{i=1}^{N} total\_faults\_detected_i}{total\_coding\_faults\_detected}$$

The direct and derived measures may be related by a measurement model, defining how the metrics relate to one another, equations such as the one in Example 5.13 are useful models. Sometimes, graphs or relationship diagrams are used to depict the ways in which metrics are calculated or related.

Once the set of metrics is clear, and the set of components to be measured has been identified, you must devise a scheme for identifying each entity involved in the measurement process. That is, you must make clear how you will denote products, versions, installations, failures, faults, and more on your data collection forms. This step enables you to proceed to form design, including only the necessary and relevant information on each form. We shall look at forms more closely in the next section.

Finally, you must establish procedures for handling the forms, analyzing the data, and reporting the results. Define who fills in what, when, and where, and describe clearly how the completed forms are to be processed. In particular, set up a central collection point for data forms, and determine who is responsible for the data, each step of the way. If no one person or group has responsibility for a given step, the data collection and analysis process will stop, as each developer assumes that another is handling the data. Analysis and feedback will ensure that the data are used, and useful results will motivate staff to record information.

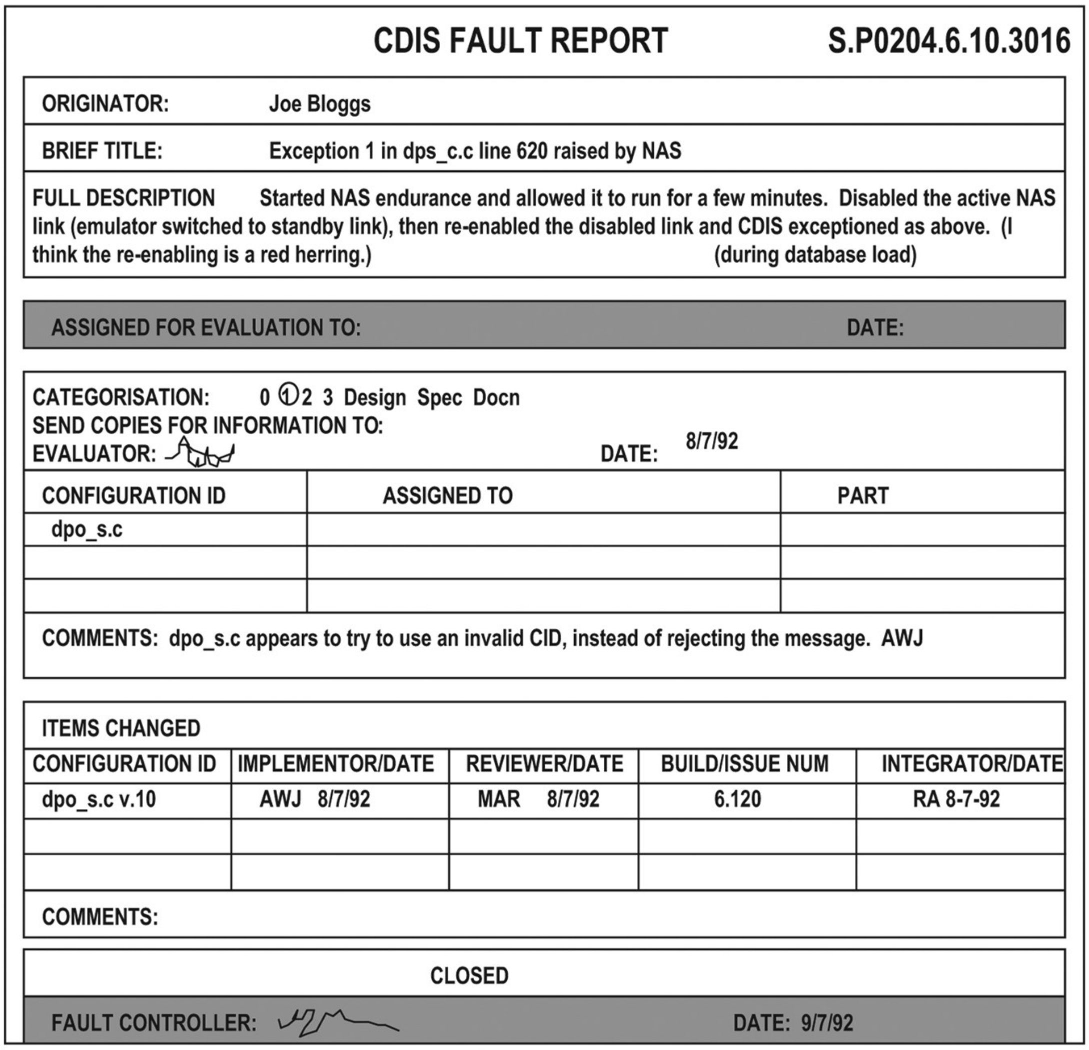
5.3.1 Data Collection Forms

A data collection form encourages collecting good, useful data. The form should be self-explanatory, and include the data required for analysis and feedback. Regardless of whether the form is to be supplied on paper or computer, the form design should allow the developer to record both fixed-format data and free-format comments and descriptions. Boxes and separators should be used to enforce formats of dates, identifiers and other standard values. By pre-printing data, that is the same on all forms (e.g., a project identifier) will save effort and avoid mistakes. Figure 5.3 shows an actual form used by a British company in reporting problems for air traffic control support system. (Notice that it is clearly designed to capture failure information, but it is labeled as a fault report!)

As we have seen, many measurement programs have objectives that require information to be collected on failures, faults, and changes. The remainder of this section describes the forms used in case study of a project whose specific objective was to monitor software reliability. The project developed several data collection forms, including separate ones for each failure, fault, and change. We leave it as an exercise for you to determine whether the data collection forms have all the attributes we have suggested in this chapter.

Table 5.3 depicts an index for a suggested comprehensive set of forms for collecting data to measure reliability (and other external product attributes). Each form has a three-character mnemonic identifier. For example, "FLT" refers to the fault record, while "CHR" is a change record. These forms are derived from actual usage by a large software development organization. As we describe each set of forms, you can decide if the forms are sufficient for capturing data on a large project.

Some fields are present on all or most of the forms. A coded Project Identifier is used so that all forms relevant to one project can be collected and filed together. The name and organization of the person who completes each form must be identified so that queries can be referred back to the author in case of question or comment. The date of form completion must be recorded and distinguished from the date of any failures or other significant events. The identifier and product version must always be recorded,



CDIS FAULT REPORT **S.P0204.6.10.3016**

ORIGINATOR:	Joe Bloggs
BRIEF TITLE:	Exception 1 in dps_c.c line 620 raised by NAS

FULL DESCRIPTION Started NAS endurance and allowed it to run for a few minutes. Disabled the active NAS link (emulator switched to standby link), then re-enabled the disabled link and CDIS exceptioned as above. (I think the re-enabling is a red herring.) (during database load)

ASSIGNED FOR EVALUATION TO: DATE:

CATEGORISATION: 0 [✓ 1] 2 3 Design Spec Docn
SEND COPIES FOR INFORMATION TO:
EVALUATOR: [illegible] DATE: 8/7/92

CONFIGURATION ID	ASSIGNED TO	PART
dpo_s.c		

COMMENTS: dpo_s.c appears to try to use an invalid CID, instead of rejecting the message. AWJ

ITEMS CHANGED

CONFIGURATION ID	IMPLEMENTOR/DATE	REVIEWER/DATE	BUILD/ISSUE NUM	INTEGRATOR/DATE
dpo_s.c v.10	AWJ 8/7/92	MAR 8/7/92	6.120	RA 8-7-92

COMMENTS:

CLOSED

FAULT CONTROLLER: [illegible] DATE: 9/7/92

FIGURE 5.3 Problem report form used for air traffic control support system.

TABLE 5.3 Data Collection Forms for Software Reliability Evaluation

Identifier	Title
PVD	Product version
MOD	Module version
IND	Installation description
IRP	Incident report
FLT	Fault record
SSD	Subsystem version
DOD	Document issue
LGU	Log of product use
IRS	Incident response
CHR	Change record

and a single project may monitor several products over several successive versions. For example, the Installation Description form may record the delivery and withdrawal of several successive versions of a product at a given installation, while Log of Product Use may be used to record the use of several different products at the same terminal or workstation.

The Product Version, Subsystem Version, Module Version, and Document Issue forms identify a product version to be measured: all its component subsystems, modules, and documents, together with their version or issue numbers. They are used to record when the product or component enters various test phases, trial, and service. Previous versions, if any, may be cross-referenced. Some direct product measures (such as size) are recorded; sometimes, the scales used for some associated process measures are also captured. Each Subsystem Version and Module Version Description form should cross-reference the product version or higher-level subsystem version, to define a hierarchical product structure. Similarly, the Document Issue Description should identify the product version of which the particular issue of the document is part. Together, this set of forms provides a basis for configuration control.

The Installation Document identifies a particular installation on which the product is being used and measured. It records the hardware type and configuration, and the delivery date for each product version.

The Log of Product Use records the amount the product is used on a given installation. Separate records must be kept for each product version, and each record must refer to a particular period of calendar time, identified by a date; for example, the period can be described by the date of the end of the week. Total product use in successive periods on all installations or terminals can then be extracted as discussed above. This form may be used to record product use on a single central installation, or adapted for use at an individual terminal on a distributed system.

The Incident Report form has fields corresponding to the attributes listed in the sample failure report at the beginning of this chapter. Each incident must be uniquely identified. The person who completes the form will usually identify it uniquely among those from a particular installation. When the report is passed to the Central Collection Point, a second identifier may be assigned, uniquely identifying it within the whole project.

The Incident Response report is returned to the installation from the central collection point following investigation of the incident's cause. If the wrong diagnosis is made, there may be a request for further diagnostic information, and several responses may be combined into one report. This report includes the date of response, plus other administrative information. After a response, the incident may be still open (under investigation) or closed (when the investigation has reached a conclusion). A response that closes an incident records the conclusion and refers to the appropriate fault record.

The Fault Record records a fault found when inspecting a product or while investigating an incident report. Each fault has a unique identifier. Note that a given fault may be present in one or more versions of the product, and may cause several incidents on one or more installations. The fields in this report correspond to those in the fault report recommended in Section 5.2.3. Finally, the Change Record captures all the fields described in the recommended change report in Section 5.2.4.

Thus, the collection of 10 forms includes all aspects of product fault, failure, and change information. Most organizations do not implement such an elaborate scheme for recording quality information. Scrutinize these descriptions to determine if any data elements can be removed. At the same time, determine if there are data elements missing. We will return to the need for these measures in Chapter 11, where we shall discuss software reliability in depth.

5.3.2 Data Collection Tools

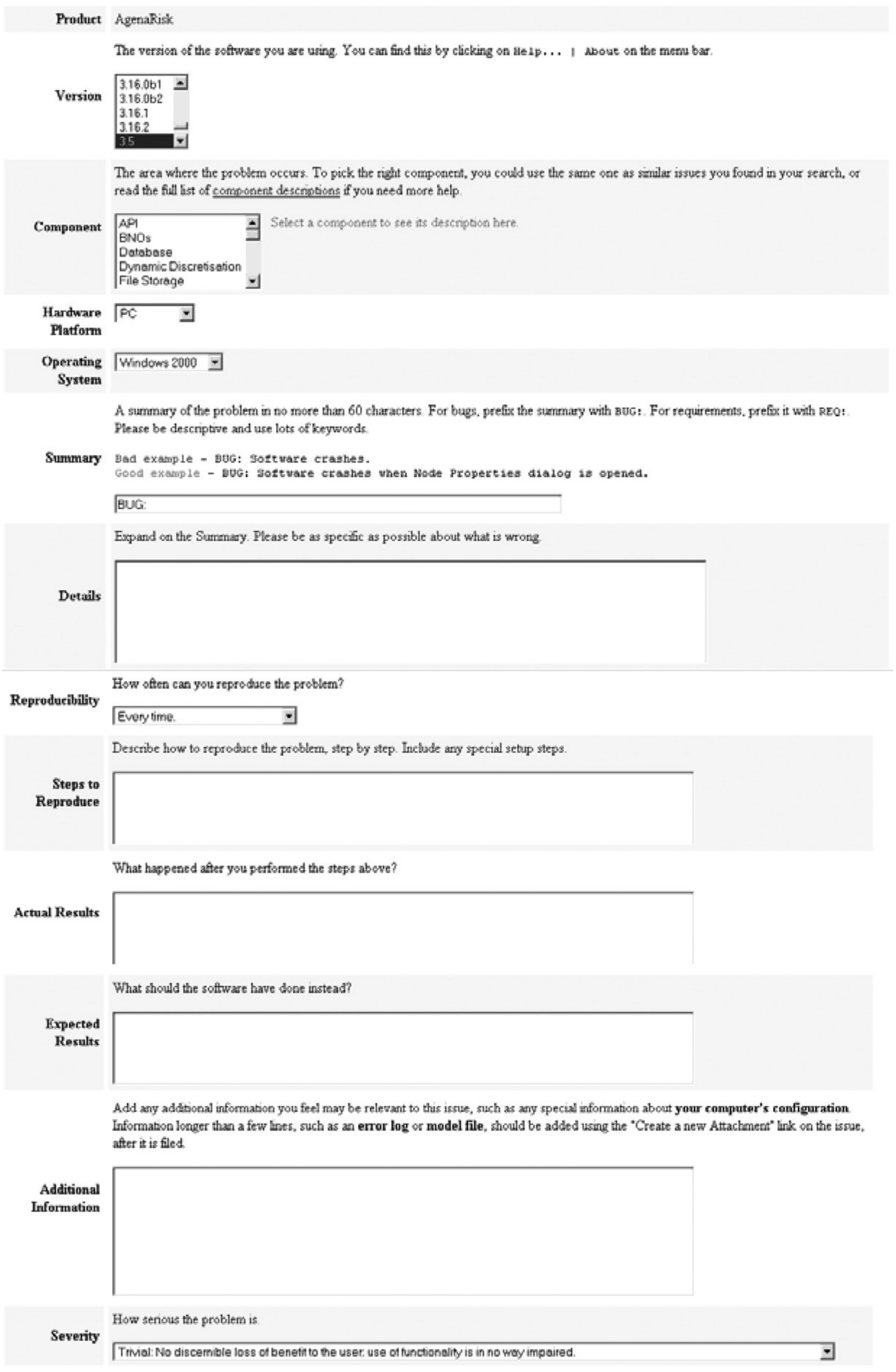
There are many software tools available that support the recording and tracking of software faults and their attributes. These tools provide data collection forms or frameworks for designing your own forms. We found 98 different commercial and freeware fault-tracking tools listed online.*
These tools can make it much easier for developers to monitor faults from their discovery to their resolution. Figure 5.4 gives a screenshot of a tailored form using the Bugzilla open-source freeware tool.

In addition to tools to support tracking of faults, tools are also available to support the collection and analysis of source code, analysis of the social networks involved in developing software, extraction, and analysis of a variety of information from CVS and subversion repository logs including the analysis of the evolution of related components and the overall architecture of a system.†

Tools to support data collection are constantly changing. One source for up-to-date information on data collection tools is the International Software Benchmarking Standards Group (ISBSG). The ISBSG web site

* Conduct a web search on terms such as "free software testing tools" or "software test automation."

† Sites with information about software project data collection and analysis tools include http://www.swag.uwaterloo.ca/tools.html and http://tools.libresoft.es/.



Product	AgenaRisk
Version	The version of the software you are using. You can find this by clicking on `Help...` | `About` on the menu bar. 3.16.0b1 3.16.0b2 3.16.1 3.16.2 3.5
Component	The area where the problem occurs. To pick the right component, you could use the same one as similar issues you found in your search, or read the full list of component descriptions if you need more help. API BNOs Database Dynamic Discretisation File Storage Select a component to see its description here.
Hardware Platform	PC
Operating System	Windows 2000

Summary	A summary of the problem in no more than 60 characters. For bugs, prefix the summary with `BUG:`. For requirements, prefix it with `REQ:`. Please be descriptive and use lots of keywords.
	 `Bad example - BUG: Software crashes.`
	 `Good example - BUG: Software crashes when Node Properties dialog is opened.`
	 BUG:

Details	Expand on the Summary. Please be as specific as possible about what is wrong.
Reproducibility	How often can you reproduce the problem? Every time.
Steps to Reproduce	Describe how to reproduce the problem, step by step. Include any special setup steps.
Actual Results	What happened after you performed the steps above?
Expected Results	What should the software have done instead?
Additional Information	Add any additional information you feel may be relevant to this issue, such as any special information about **your computer's configuration**. Information longer than a few lines, such as an **error log** or **model file**, should be added using the "Create a new Attachment" link on the issue, after it is filed.
Severity	How serious the problem is. Trivial: No discernible loss of benefit to the user: use of functionality is in no way impaired.

FIGURE 5.4 Bugzilla screenshot.

(http://www.isbsg.org) includes repositories of software development data from thousands of software projects. Thus it is a great source for industry software project data, and has links to several data collection and analysis tools. These tools focus on project size and cost estimation. Of particular interest are the ISBSG guidelines on what data the group finds to be most effective for improving software processes. ISBSG data are available for both academic and commercial use.

5.4 RELIABILITY OF DATA COLLECTION PROCEDURES

For data collection to be reliable and predictable, it needs to be automated. Data collection technology must also be adaptable so that you can continue to collect data while development environments (both development languages and tools) and the measurement tools evolve (Sillitti et al. 2004). Silliti et al. found that it takes a significant and flexible infrastructure to collect data from projects involving many developers using multiple clients employing different languages and tools. Application domains tend to be unique and require varied data collection support environments and tools. Reports generated from the data need to be customized for an application domain and the context of use.

Data that is not reliable—data that is not appropriate for the application domain or the context of use—represents a threat to the construct validity of results gleaned from the data (as described in Chapter 4).

EXAMPLE 5.14

Lethbridge, Sim, and Singer identify many factors that can affect the reliability of collected data (Lethbridge et al. 2005). One factor to consider is the closeness of the connection between the evaluators or researchers and the software developers that are part of a project being studied. The relative closeness of the connection is classified as *first degree*, *second degree*, or *third degree*:

- *First degree:* Evaluators or researchers directly interact with developers. They may interview developers, observe them working, or otherwise interact directly with developers in their daily activities.
- *Second degree:* Evaluators or researchers indirectly interact with developers. They may instrument software development tools to collect information or may record meetings or other development activities.
- *Third degree:* Evaluators have no interactions with developers. Rather, they study artifacts such as revision control system records, fault reports and responses, testing records, etc. Third-degree studies can be performed retrospectively.

First-degree studies offer researchers the greatest level of control over the quality of the data, since they directly observe and control the data collection. However, first-degree studies generally require the greatest level of resources, as researchers must be present to collect data. Also, there is a potential for the presence of researchers to affect the behavior of developers. Second-degree studies support the direct collection of data, without the presence of researchers. However, second-degree studies require the availability (or development) of appropriate data collection tools. Also, with second-degree studies, it is more difficult to collect data concerning the rationale that developers use to make particular decisions. Third-degree studies do not support the direct collection of data from developers. However, they do allow researchers to mine available software repositories for their studies.

EXAMPLE 5.15

Lincke, Lundberg, and Löwe find different data collection tools can produce measurement values that vary widely (Lincke et al. 2008). They examined the values of nine common measures generated by 10 different measurement tools, and found discrepancies. In particular, they found large discrepancies in the measured values of coupling between objects (CBO), and lack of cohesion between object classes (LCOM). The discrepancies were large enough to affect the ranking of classes in terms of these measures.

The usability of collected data depends on how the data are stored. To be useful over an extended period, data must be stored in a manner that supports future needs. Unfortunately, we may not know now how we will need to use the data in the future. Harrison proposed a flexible design for a metrics repository based on a transformational view of software development (Harrison 2004). The design uses meta-data and their relations and defers the choice of specific metrics to quantify attributes. Such flexible designs promise to support repositories that can remain useful over the long term.

5.5 SUMMARY

We have seen in this chapter that the success or failure of any metrics program depends on its underlying data collection scheme. Chapter 3 showed us how to use measurement goals to precisely specify the data to be collected; there is no single type of data that can be useful for all studies. Thus, knowing if your goals are met requires careful collection of valid, complete, and appropriate data.