Chapter

9

Orthogonal Defect Classification

Ram Chillarege
IBM Watson Research

9.1 Introduction

One of the perpetual challenges for measurement in the software development process is to provide fast and effective feedback to developers. Traditional techniques, although they provide good report cards, seldom have the fine insight into the process or product to truly guide decision making. Thus it is not uncommon to witness decisions guided more by intuition than by true measurement, analysis, and engineering.

Orthogonal defect classification (ODC) makes a fundamental improvement in the level of technology available to assist software engineering decisions via measurement and analysis. This is achieved by exploiting software defects that occur all the way through development and field use, capturing a significant amount of information. ODC extracts the information from defects cleverly, converting what is semantically very rich into a few vital measurements on the product and the process. These measurements provide a firm footing upon which sophisticated analysis and methodologies are developed. Thus it is possible to emulate some of the detail and expansiveness of qualitative analysis with the low cost and mathematical tractability of quantitative methods.

At IBM Watson Research we began a program to place ODC in the real world of software development. Over the past few years, it has been deployed at a dozen IBM locations, worldwide, in over 50 projects. It has also been extended to apply at different parts of the development cycle and different layers of decision making. Over time there have been analytical methods, tools, methodologies, feedback techniques, diagnosis procedures, and support processes developed to fit different needs. When applied to reduce the cost of classical root-cause analysis, it achieves cost reductions by a factor of 10. An organization that fully

employed ODC, coupled with the Butterfly model [Bass93], has seen a cycle-time reduction by a factor of 3, and achieved a defect reduction by a factor of 80 in 5 years.

This chapter is designed to provide an overview of ODC, complete with examples illustrating its real-world application. It begins with an overview of the domain of defects in software development, followed by examples which motivate the ODC concept. Since the concepts are fairly new, we have chosen to describe two attributes—the defect type and trigger—in detail, which provide measurement instruments on the development process and the verification process. The latter part of the trigger section illustrates how the two together provide measures of effectiveness. These sections are complete with definitions so that it is possible to start some sample prototypes. ODC is primarily a multidimensional measurement system, which allows for very creative analysis. The end of this chapter hints at this and briefly sketches the current implementation at IBM, which includes two additional attributes: source and impact. We have, of course, assumed that classical measures such as severity and process-phase identifiers exist, since most defect-tracking systems provide them. The key in all such endeavors is to set up a measurement system (not merely a classification taxonomy) that is clean, expandable, and broadly applicable. We believe ODC makes a significant stride in accomplishing that.

9.2 Measurement and Software

Software, as most of us know, is a very difficult area in which to develop crisp in-process measurements and analysis techniques. Nevertheless, the need to get a grasp on the software development processes is vital to manage it as a business. This need becomes ever more critical as the software industry grows, reaching almost $100 billion in 1995,* and becomes more competitive. The task has enchanted software engineering for decades, but the solutions available were much too primitive to significantly impact the level of technology in practice. The best practices in the software industry have therefore remained largely human-intensive processes, which are qualitative, suffer poor repeatability, and have difficult introduction barriers.

The software development process is amorphous by nature—be it waterfall, iterative development, or an undefined chaotic process. The definitions of activities within a process and transitions between them are dependent largely on human behavior as opposed to being enforced by physical restrictions or conceptual roadblocks. The few crisp separa-

* *Business Week* summary of the computer industry, 1995.

tions between activities are primarily due to the rigidity of tools and their limitations. Thus, measurement in the software development environment is a tough challenge until tools exist to automatically extract them. As technology evolves and new tools are invented, they directly compete with some of the activities defined within a process. A technology might render an existing process-based activity ineffective or even unnecessary. Thus, processes have to be flexible and sensitive to accommodate such advances. At the same time, the measurement system needs to make the corresponding compensations. For instance, code inspection used to include a large number of checks to guard against errors due to the operational semantics of languages. As syntactic checkers and tools such as Lint became more powerful, they rendered some of these checks unnecessary. However, when the code inspection process and related measurements did not quickly adapt to these advances, some development communities reacted by doing away with code inspections altogether. In this particular case, the resultant loss was significant, since inspections addressed a much broader set of issues.

Measures related to software defects have survived the evolution of software development processes and are still widely used despite large variability in processes. The popular belief is that this is because defect counts provide a measure of product quality and, indirectly, productivity. However, the real reason, we believe, is far more subtle. A software defect signifies stoppage in the product development process. By the time the defect is resolved several activities and pieces of work occur. Thus, the defect has the potential to capture a variety of information on the product and the process—thereby providing an excellent measurement opportunity. Thus, defect-related information is actively used for a variety of purposes ranging from work completion to resource management to schedule estimation to risk analysis, and even continuous improvement.

This chapter is an exposition on the use of defects to measure, manage, and understand the software development process. Although the use of defect data has existed for years, its exploitation has been limited due to the lack of some fundamental insights. Orthogonal defect classification provides this insight and a framework to achieve the next order of technology.

9.2.1 Software defects

The word *defect* tends to include in its meaning more than a mathematician would like to attribute to it. From the perspective of strict definition it often captures the fault, sometimes the error, and often the failure. Given the wide range of meanings that could be attributed to a defect we need to clarify how defect data are used, what aspect is being used, and for what purpose. Before we delve fur-

ther we must first understand the kind of defects that appear in software development, and arrive at a reasonably unambiguous definition for the word *defect.*

Software defects occur all the way through the life cycle of software development—from conception of product to end of life. The vernacular of development organizations tends to name and treat defects as different objects depending on when or where they are found. Some of the more common names are *bug, error, comment, program trouble memoranda, problem,* and *authorized program analysis report* (APAR). For starters, let us describe a couple situations where defects are identified, and then develop a more unifying definition.

When design documents and requirements are written, they are usually captured in plain text. Although there is increasing emphasis on documenting them in design tools, that is yet an evolving technology. When these documents are reviewed among peers, they can result in comments that question or critique the design. These comments may eventually result in changes. The concept of *failure* here is broader than its strict definition. A potential failure is identified by the human recognizing a departure, exception, or a gaping hole in the design. The corrective action results in changes that fix a likely fault. The comments and the resultant changes are essentially defects. Unfortunately, from a measurement perspective, quite often these defects are not formally captured by an organization and can be lost in desk drawers.

When a customer calls with a *problem* experienced with a product, it might be due to a software failure caused by a fault. On the other hand, not all problems experienced are due to the classical software programming bug causing a failure. More often than not (sometimes 80 to 90 percent), a customer calls experiencing difficulties due to poor procedures, unclear documentation, poor user interfaces, etc. The product may actually be working as designed, but poorly and much to the dissatisfaction of the customer. In some of these cases, the vendor may admit to the poor workings of the product and open a defect against it. This could eventually result in a fix being developed and distributed.

One of the primary difficulties with the notion of failure in software is that it is more amorphous than in hardware. The scientific definition accepted for *failure* is "deviation of the delivered service from compliance with the specification" [Lapr92a]. However, there is rarely a well-documented specification for most software in the industry. It is common practice to recognize a failure when a piece of software does not meet customer expectations. The corresponding fault can be equally obscure to identify, particularly in areas such as usability. The terms *fault, error,* and *failure* provide a good conceptualization model that we can draw from. However, for the practicality of implementation, ease of understanding, and a tangible connection to the real

world, we develop the concept of a *defect:* simply put, it is a necessary change to the software.

There is a subtlety in saying that the *necessary* change is the defect. This is because it could be the case that a change was necessary but was not executed. It may have been forgotten, or remembered but without the resources to fix it. It is still a defect, since it was deemed a necessary change. Treating the necessary change as a defect ties the metaphysical concept of a fault, error, and sometimes failure to a physical action on the product that is more traceable. No matter what the development environment, the activity is specific and can become the anchor point from which other measurements can follow.

A software product usually begins with some set of requirements, either explicitly stated by a customer or identified through a vision of the market. Between the requirements and the availability of the product, there is design, code development, and test. The words *design, code,* and *test* do not automatically imply a pure waterfall process. On the other hand, it is hard to imagine a successful product developed without requirements, or code written without a design, or tests conducted without code. A pure waterfall process tends to have these phases well identified and separated with documents that capture the essence of these phases, providing well-defined handoffs between them. In more recent times, the small team with iterative development is popular, where the reliance on documentation and distinct handoffs are less emphasized. However, having some record of completion and milestones within a process is not only useful for project management but necessary for internal communication.

No matter what the implementation of a software development process, there are artifacts that are developed and storage mediums where the artifacts are recorded. Depending on the tools and methods, these artifacts may be on paper, images on tape, code in libraries, meeting minutes, discussions yielding decisions, etc. Furthermore, there is a time after which a work item is considered committed. It may not yet be completed, but being committed implies a clear record of intent. Any further necessary changes to the work item that becomes a product is considered a defect. Depending on the tools used in a development process, the identification and tracking of defects may be easy or difficult.

A process usually has several checks interspersed to ensure that the product being developed meets the record of intent. These checks could be reviews, inspections, informal tests, formal functional tests, stress tests, etc. Each of these verification efforts can identify deviations to the intent, which warrant change. Anything that requires change is a defect. Thus, there are defects that arise from requirements inspection, design inspection (usually, a high level and a low level), a code inspection (of which there might be one or two), a unit test, a functional veri-

fication test, and a system test. All these verification efforts can yield defects. Depending on what is included in a count, we find the number of defects per thousand lines of code varies, from as low as 10 to as high as 120.

9.2.2 The spectrum of defect analysis

Analysis of software defects has existed for a long time in the industry. One of the more common forms is bean counting for purposes such as estimating completion time and warranty cost. Another popular form of defect analysis is qualitative—to recognize the types of errors committed and guide defect prevention. Stepping back from the variety of methods of defect analysis, we can recognize a broad spectrum in its usage. The two extremes of this spectrum are distinctly defined, and Fig. 9.1 captures some of its elements. The left-hand extreme signifies purely quantitative methods such as mathematical and statistical methods. The right-hand extreme signifies purely qualitative analyses such as those employed in quality circles and defect prevention programs. There is a large white space between these extremes, which is the subject of this chapter. Before we delve into this white space, let us examine the extremes first.

An example of the left-hand extreme is statistical models of defect discovery. One of the well-known examples is the Raleigh model. This is an abstraction that is simple and intuitive, and has found extensive use. Another example is the software reliability models, several of which are discussed in this handbook (see Chaps. 3, 4, 6, and 7). As useful as these are, one has to recognize that they are the mathematicians' approximation of the defect process. Defect counts measure just one of the many effects of the complex human process of software development. In and of itself it represents a small part of the information from

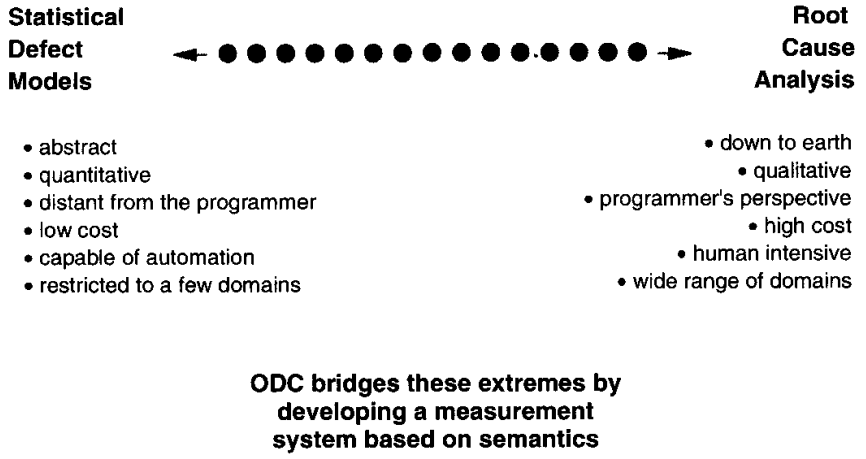

Figure 9.1 The spectrum of process measurement and analysis.

the software development process. We do not understate its implication or use, but recognize up front the overall scope and opportunity that is exploited or left unexploited.

A good example of the right-hand extreme is the classical approach toward defect prevention using causal analysis of defects. The origins are traceable to Demming and Juran and a more recent implementation of the process described in [Mays90, Hump89]. The methods employed are detailed qualitative analysis of defects, often conducted by a small team of people. The goal is to identify practices or oversights that could be prevented by implementing corrective or educational actions in the development organization. This is not a quick process, either for the analysis or the implementation of the program. For example, a team of four people discussing defects could reasonably work through three or four in an hour, yielding a cost of one person-hour per defect for the analysis phase. This is usually followed by action teams that execute the plans and directions developed from such analysis. It is not uncommon to see organizations that have developed ambitious action lists without the resources or commitment to execute them. The challenge is mostly one of prioritization and knowing the cost/benefit of such actions. This fact does not detract from the success of defect prevention. There have been significant improvements in quality and productivity reported. The challenge is how to do it cleverly. The traditional methods of laborious qualitative analysis followed by poor prioritization do not make it attractive in a world of short cycle time and reduced budgets.

Between the two extremes of the spectrum (quantitative statistical defect models and qualitative causal analysis) is a wide gap. This gap is characterized by a lack of good measurement methods that are meaningful to the developer and that exploit good engineering methods. At one extreme, the traditional mathematical modeling efforts tend to step back from the details of the process and to approximate defect detection by a statistical process [Litt73, Ohba84]. When calibrated, some of these methods are shown to be quite repeatable. However, they do not provide timely feedback to the developer in terms of available process controls. At the other extreme, the causal analysis mechanism is qualitative and labor-intensive; it can provide feedback on each individual defect. However, in a large development effort it is akin to studying the ocean floor with a microscope. It does not naturally evolve into abstractions and aggregations that can feed into engineering mechanisms to be used for overall process control.

It is not as though there has been no work done between these extremes. There is a myriad of reported research and industry attempts to quantify the parameters of the software development process with "metrics" [IEEE90a, IEEE90b]. Some efforts have been more successful

than others. For example, the relationship between the defects that occur during software development and the complexity of a software product has been discussed in [Basi84a]. Such information, when compiled over the history of an organization [Basi88], will be useful for planners and designers. On the other hand, the focus of empirical studies has largely been to validate hypotheses that could be turned into a set of general guidelines. Unfortunately, the variability in productivity, skill, and process makes such findings difficult to establish. Each develolpment effort appears unique, making the task of treating them as repeatable experiments hard. Over the years, this has led to disillusionment of software measurement and in some extremes a distaste for the effort, particularly when developers are both skeptical about process change and resistant to the extra burden. Yet, it remains one of the most important technological areas in which to make progress, given where the software industry is headed.

As we shall see, this cause is not without hope. Only recently have there been studies to examine the feasibility of serious analytical software engineering. The work explored the capability of taking the semantically rich information from the defect stream and turning it into measurements. At the same time it tried to establish the existence of information that could predict the behavior of a software development effort via such measurements. The successes illustrated that a new class of methods can be developed that rely on semantic extraction of information linking the qualitative aspects from the right extreme of the spectrum to measurable computable aspects from the left extreme. The semantic extraction is done via classification. The link with the right extreme occurs when the classification has properties that make it a measurement. It should not be confused with a mere taxonomy of defects such as [IEEE87b], which serves a descriptive purpose. ODC makes the classification into a measurement that helps bridge the gap.

In summary, although measurements had been extensively used in software engineering, it still remained a challenge to turn software development into a measurable and controllable process. Why is this so? Primarily because no process can be modeled as an observable and controllable system unless explicit input-output or cause-and-effect relationships are established. Furthermore, such causes and effects should be easily measurable. It is inadequate to propose that a collection of measures be used to track a process, with the hope that some subset of these measures will actually explain the process. There should at least be a small subset that is carefully designed based on a good understanding of the mechanisms within the process.

Looking at the history of software modeling, it is evident that little heed has been paid to the actual cause-effect mechanism, let alone

investigations to establish them. At the other extreme, when cause and effect was recognized, though qualitatively, it was not abstracted to a level from which it could graduate to engineering models. To the best of our knowledge, in the world of in-process measurements, until recently there has been no systematic study to establish the existence of measurable cause-and-effect relationships in a software development process. Without that insight and a rational basis, it is difficult to argue that any one measurement scheme or model is better than another.

9.3 Principles of ODC

Orthogonal defect classification (ODC) is a technique that can bridge the gap between quantitative methods and qualitative analysis. It does so by bringing in scientific methods by defining a measurement system in an area that has been historically ad hoc. This naturally provides a firm footing upon which sophisticated and detailed analysis can be developed. Fundamentally, we extract semantic information in defects via classification. ODC properties ensure that the classified data become measurements. As these measurements are related to the process and environment they provide the instrumentation for very detailed and precise feedback. Thus defects which are rich in information can be scientifically exploited to assist the decision-making process in software engineering.

9.3.1 The intuition

ODC is best described with an example that captures some of the motivations and provides an intuitive understanding of the concept. The detailed technical description can then be built on this base. We do so by illustrating, in Example 9.1, a situation from the real world. The objective is to contrast the classical methods of growth modeling with what can be achieved via semantics extraction from defects.

Example 9.1 The example is from the development of a component of 80,000 to 100,000 lines of code in an operating systems product. During the last few drivers (and months) of system test it became evident that there was a crisis in stability and completion. The top part of Fig. 9.2 shows the cumulative number of defects over time. Ideally, it should plateau, signifying a decreasing number of defects being detected and promising fewer defects in the field. Although the figure shows a slight plateau, it is artificial since these data did not include defects found in the field. It is the steep increase in the defect rate during period 3 leading to the near doubling of development defects, which identifies the criticality of the situation. Classical growth curve modeling techniques would recognize this trend and identify the problem. The issue is that by the time growth modeling can identify the problem, it is too late to take all but some desperate reactive measures.

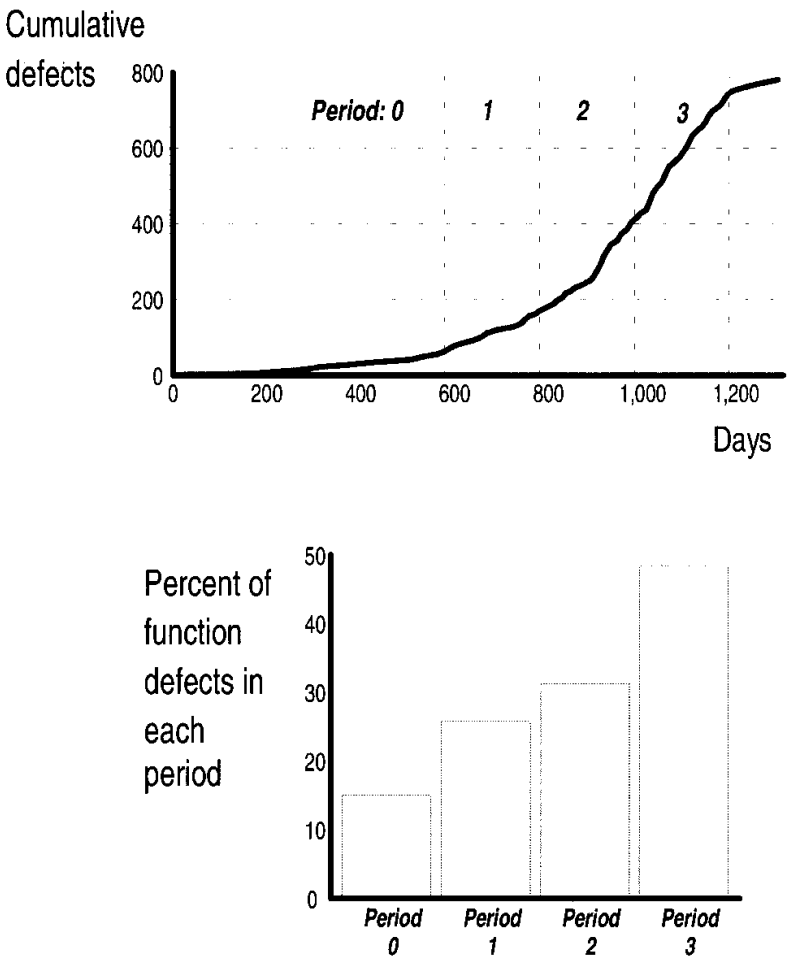

Figure 9.2 Total defects and proportion of function-type defects.

We attempt to do better than raw counts of defects by using the semantic information in them. To do so, we categorized defects into these classes: *assignment, checking, function, interface,* and *timing.* In the paradigm of programming, there are only certain ways to fix a programming problem. These categories capture the essence of what was fixed, thereby identifying the nature of work necessary to fix the defect. This we call the *defect type.* Samples of defects from the three periods are categorized into this defect type attribute. We then examine how the distribution contributed by defects changes as a function of time. For the purposes of illustration, the lower part of Fig. 9.2 shows just the proportion of defects that have the type function during each period.

The early periods of development are characterized by larger amounts of design, whereas the latter parts are characterized by greater amounts of system test and less of function test. Thus, the expectation would be that the proportion of defects of type function would be larger initially and smaller later. However, the data show exactly the opposite trend. Period 3, which was largely system test, shows close to 50 percent of the defects being of type function. The fact that the proportion of functional defects kept rising is discernible even in period 2. This identifies the crises that occurred by using semantic information almost six months earlier than by using raw defect counts. Furthermore, it also points to the potential cause of the problem and motivates corrective measures. In this case, it will

require the appropriate skills (possibly design) to start examining the code and design. Reactive measures such as redoubling the test effort are unlikely to be as effective.

Example 9.1 illustrates the use of qualitative information in defects converted to a quantitative measure to make earlier predictions than more traditional quantitative methods. In addition, it provides clues to the reasons, which are translatable to recommendations for action. In Example 9.1, a causal analysis team could be directed to focus on a few of the function defects to provide further guidance to the development team. This approach avoids "boiling the ocean" to determine causality, since the problem and scope are better understood.

We need to extend this idea further toward a general case. Figure 9.3 shows a hypothetical distribution of defect types across the phases of design, unit test, integration, and system test. For the purposes of illustration we show just four defect types: function, assignment, interface, and timing. The fact that the process phases are shown in sequence does not imply a strict waterfall development process. As long as these activities exist (in any order or repeatedly), defects from these activities are used to construct the four defect-type distributions. The concept is that we examine the normalized distribution of all defects found during a phase against what the process should achieve. The bar representing function defects is steadily decreasing from design all the way to system test. This would make sense given that the intent of a design phase is to deal with functional issues and the expectation is that fewer of them have to be dealt with at system test. It is reasonable to anticipate that the proportion of timing defects would increase quite the opposite way to functional defects. System test occurs when the product is on the real hardware and is more likely to be stressed by timing conditions. The assignment and checking kinds of defects are likely to peak during unit test and fall either before or after that phase, whereas interfaces defects would possibly peak during the integration test and fall off on either side of those process stages. By defining a distribution that changes with the process activity we have created a very powerful instrument. *This instrument will allow us to measure the progress of a product through the process.* Changes in the distribution are easy to measure and are invariant to the total number of defects injected by a development community. A departure from the expected distribution or a change from the expected trend identifies problems and recommends possible corrective action. For instance, had the defect-type distribution at system test looked like that of unit test, we could argue that although the product met physical schedule dates it probably hadn't progressed logically. The offending defect type would be in a large number of function

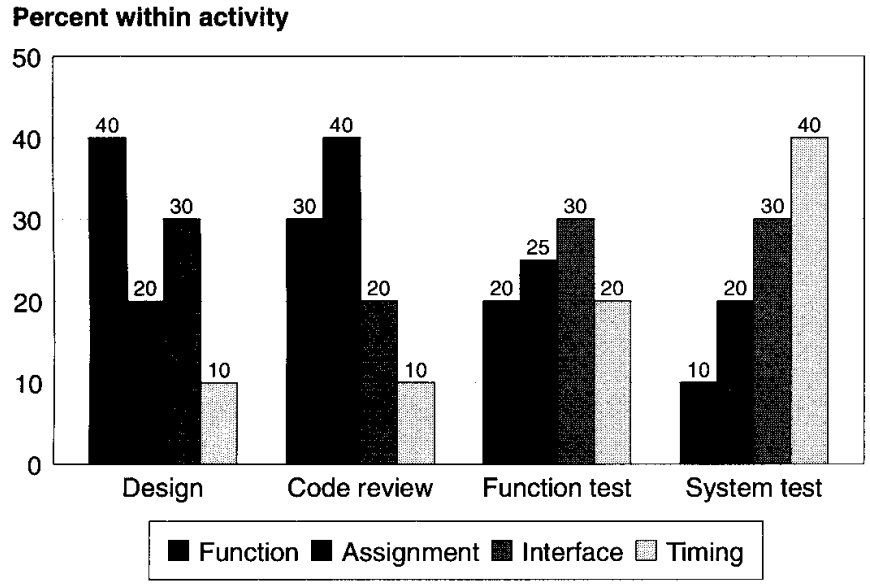

Figure 9.3 Change in the defect-type distribution with phase.

and assignment defects relative to the expected numbers, requiring that the corrective actions address those departures.

9.3.2 The design of orthogonal defect classification

The preceding was a hypothetical discussion of defect-type categories and what that could mean in the process space. In actuality, we use more than four defect types, which we illustrate with some real-life examples. It is useful at this point to conceptualize what it is that we are doing differently and why it works for us. Notice that we carefully extract semantics of defects using a classification scheme. The classification scheme is able to extract information that tells us how well the product is progressing on a scale that is defined within the process. The categories themselves have been carefully chosen. Otherwise, we would not have the kind of effects that we would desire from such measurement. So, clearly, there is an issue of defining the value set into which an attribute is categorized. The value set should provide an adequate range to provide the differentiation and resolution desired within the process space. These requirements boil down to a set of necessary and sufficient conditions that make it orthogonal defect classification. The orthogonality here is in the value set associated with the particular attributes such as defect type. When these values are independent of each other (at least reasonably, since it is a semantic translation) it provides for easy categorization.

ODC essentially means that we categorize a defect into classes that collectively point to the part of the process that needs attention, much like characterizing a point in a cartesian system of orthogonal axes by

its x, y, and z coordinates. In the software development process, although activities are broadly divided into design, code, test, and service, each organization can have its variations. These process stages often overlap, with consecutive releases being developed in parallel. With a large product these process stages can be carried out by different organizations, while for small products people may be shared across products. Therefore, for ODC to become a measurement system that allows for the analysis of such complexity, it should be invariant to these boundaries. The classification should be consistent across process stages; otherwise it is almost impossible to study trends across them. Ideally, the classification should also be quite independent of the specifics of a product or organization. If the classification is both consistent across phases and independent of the product, it tends to be fairly process-invariant and can eventually yield relationships and models that are very useful. Thus, a good measurement system that allows learning from experience and provides a means of communicating experiences between projects has at least three requirements:

- Orthogonality
- Consistency across phases
- Uniformity across products

One of the pitfalls in classifying defects is that it is a human process and subject to the usual problems of human error, confusion, and a general distaste if the use of the data is not well understood. However, each of these concerns is resolved if the classification process is simple, with little room for confusion or possibility of mistakes, and if the data can be easily interpreted. If the number of classes is small, there is a greater chance that the human mind can accurately resolve them [Mill90]. Having a small set to choose from makes classification easier and less error-prone. When orthogonal, the choices should also be uniquely identified and easily classified.

9.3.3 Necessary condition

There exists a semantic classification of defects such that its distribution, as a function of process activities, changes as the product advances through the process.

The earlier examples help explain this necessary condition, since they discussed not only what it means, but how it could be used. There are, however, a few subtleties and details that merit discussion. It also helps to compare ODC with more traditional approaches to clarify this scheme and illustrate the differentiation.

Fundamental to the use of the change in distribution of defects to meter the advancement of a product in a process is the existence of such changes. All these years, while there has been significant research in closely related areas such as metrics, software reliability, taxonomy for defect classification, and root-cause analysis, etc., there were almost no studies that tried to quantify the existence of properties that allowed us to extract semantics and turn them into measurements. Thus, while studies reported statistics on causes of defects and the proportion of escapes from various stages of development, they stopped short of developing a comprehensive measurement methodology. This was not a simple oversight. It is a difficult problem, which requires the demonstration of properties that make a connection between semantics and measurement possible. A breakthrough study that attempted to uncover the existence of such properties relating the semantics in defect content to the overall response from a development effort is discussed in [Chil91]. This became the stepping-stone to help formulate the concepts in ODC.

However, it is easy to confuse semantic classification of defects with direct process-based measurement. This difference is subtle but important. Take, for example, the defect type attribute. It is primarily defined on the paradigm of programming as opposed to the process of how to do it. So the value set is about the meaning of what is done as opposed to how it is done. This has the advantage that even though the underlying process may change or be undefinable, the measurements are possible. Thus, they allow the process to be changed when measurements from the very process are used for the analysis that motivates the change. This would not be possible if the measurements are directly tied to the process, since then the very first adjustment to the process destroys the measurement system.

This point is best illustrated with an example that is commonplace in the industry. A popular classification of defects is the value set of where it is believed to have been inserted. So, for instance, the values would comprise the process activities such as design, coding, unit test, function test, and system test. This can be used to estimate escapes from the various stages of development, yielding a causality measure that can be quite useful. The distribution of this value set may also change as a function of activity (especially since the value sets themselves are the activities). Unfortunately, as simple and straightforward as this may seem, it is plagued with several problems. First, practitioners are quick to point out that the answer to the above question tends to be error-prone. Programmers, while fixing a defect, are too closely focused on the product to step back and reflect on the process. Thus the most common answers to "where the defect is injected" are "the earlier stage" or "requirements," which do not help much. Second,

a causality mapped directly on the existing process has limited longevity. If the process is changed or altered (which is sometimes the goal of the exercise), then the measurements so far are subsequently invalid. Third, use of these data, inferences, and learning are limited to this project or to processes that are identical to this one. Finally, it cannot work where the process is not well defined or the process is being changed dynamically to suit the pressures of changing requirements. Such direct classification schemes, by the nature of their assumptions, qualify as good opinion surveys but do not constitute a measurement on the process.

Semantic classification that is based on the meaning of what is done is more likely to be accurate, since it is tied to the work just completed. It is akin to measurements of events in the process, as opposed to opinions of the process. There is an important advantage in the semantic classification of a defect, such as *defect type,* over an opinion-based classification, such as *where injected.* This semantic classification is invariant to process and product, but requires a mapping to process stages. This mapping, such as associating function and algorithm defects to a process activity (e.g., design or low-level design), provides the flexibility to keep the measurement system stable. Furthermore, this neutrality with products, processes, and even implementation methodologies offers the opportunity that these measurements could be benchmarked across the industry.

9.3.4 Sufficient conditions

The set of all values of defect attributes must form a spanning set over the process subspace.

The sufficient conditions are based on the set of elements that make up an attribute, such as *defect type.* Based on the necessary conditions, the elements need to be orthogonal and associated with the process on which measurements are inferred. The sufficient conditions ensure that the number of classes is adequate to make the necessary inference. Ideally, the classes should span the space of all possibilities that they describe. The classes would then form a spanning set, with the capability that everything in that space can be described by these classes. If they do not form a spanning set then there is some part of the space on which we want to make inferences that cannot be described with the existing data. Making sure that we have the sufficiency condition satisfied implies that we know and can accurately describe the space into which we want to project the data.

Given the experimental nature of the work, it is hard to a priori guarantee that sufficiency is met with any one classification. Given that we are trying to observe the world of the development process and

make inferences about it from the defects coming out, there are the tasks of (1) coming up with the right measurement, (2) validating the inferences from the measurements with reference to the experiences shared, and (3) improving the measurement system as we learn more from the pilot experiences. However, this is the nature of the experimental method [Chil90]. For example, in the first pilot [Chil91], the following defect types evolved after few classification attempts: function, initialization, checking, assignment, and documentation. This set, as indicated earlier in this section, provided adequate resolution to explain why the development process had trouble and what could be done about it. However, in subsequent discussions [IBM90] and pilots it was refined to the current eight. Given the orthogonality, in spite of these changes several classes such as *function* and *assignment* and the dimension they spanned (associations) remained unchanged.

9.4 The Defect-Type Attribute

A programmer making the changes for a defect is best suited to pick the defect type. The selection of defect type captures the nature of the change. These types are simple in that they should be obvious to a programmer without much room for confusion. In each case a distinction is made between something *missing* or something *incorrect.*

Function. A function defect is one that affects significant capability, end-user features, product application programming interface (API), interface with hardware architecture, or global structure(s). It would require a formal design change.

Assignment. Conversely, an assignment defect indicates a few lines of code, such as the initialization of control blocks or data structure.

Interface. Corresponds to errors in interacting with other components, modules, device drivers via macros, call statements, control blocks, or parameter lists.

Checking. Addresses program logic that has failed to properly validate data and values before they are used, loop conditions, etc.

Timing / serialization. Timing/serialization errors are those that are corrected by improved management of shared and real-time resources.

Build / package / merge. These terms describe errors that occur due to mistakes in library systems, management of changes, or version control.

Documentation. Errors can affect both publications and maintenance notes.

Algorithm. Errors include efficiency or correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need for requesting a design change.

The defect types are chosen so as to be general enough to apply to any software development, independent of a specific product. Their granularity is such that the classifications apply to a defect found in any phase of the development process, yet can be associated with a few specific activities in a particular process. The defect types should also span the space of these phases to satisfy the sufficient condition. For instance, a typical association that occurs is to tie the functional defect to the design aspects of the process. Thus, no matter where the defect is found, if the distribution peaks with function dominating the distribution, it is indicative of activity that escaped the design phases. It is not as though function defects may only be found at a design review; they probably will be found throughout the development process. The issue is how many, and whether they dominate the overall mix of work that needs to take place during an activity. Similarly, an assignment and checking defect may be primarily associated with the coding phase and expected to be weeded out with code review and unit-test-type activity. These could peak repeatedly in the case of iterative development in those phases. However, when the function tests tend to be dominated by those types of defects, it is reflective of escapes from earlier verification phases, and impacts the overall productivity of the function test activity. The choice of the defect types are different enough so that they span the development process through all its activities. This allows us to use the distributions to provide feedback on the process, especially in the language that programmers can relate to.

Example 9.2 Figure 9.4 shows you the defect-type distribution of a product moving through the stages of high-level design, low-level design, code, and unit test. These data reflect a real product version of the hypothetical picture that we described earlier. We have chosen to show the total number of defects within each of these activities, as opposed to the normalized distribution, to also reflect a sense of volume that occurs in these phases. Notice that the bar corresponding to function defects has been steadily decreasing from high-level design all the way to unit test and is also decreasing in the rank order among the bars within each of these activities. At the same time notice that the assignment and checking types of defects are increasing both in volume and rank order among the other defect types as the product moves from high-level design into code review. Similarly, the algorithm types of defects tend to be peaking at low-level design and then trailing off into code review. This change in defect-type distribution is reflective of all the various discussions that we have had so far regarding the semantic extraction and explanation of how ODC works.

Example 9.3 There is another presentation of a defect-type distribution in Fig. 9.5. These curves provide a graphical display for a categorical data series so that an ordinate through the curves has intercepts that add up to 100 percent. Thus,

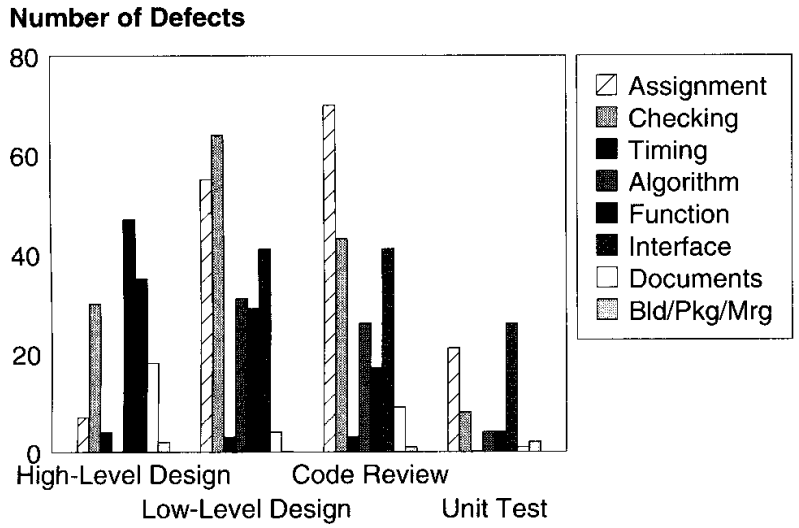

Figure 9.4 The defect type and process associations.

these curves show an instantaneous change in the distribution of defect data as a continuous curve created by smoothing the distribution over a moving window. The curve is a combination of a simple moving average and a cosine arch moving average applied to the binary representation of each category [Biya95]. The figure shows four groups of defect types: (1) assignment and checking, (2) function and algorithm, (3) timing and interface, and (4) the rest of them. The idea being that we would like to examine how the distribution of these groups of defect types changes as the product advances through development. These curves represent the instantaneous change in its distribution. Note that the early part of development has a higher fraction of functional defects, which decreases through time, while the timing component, which was originally low, is increasing. The assignment and checking type of defects change as the process phases change. This representation is intuitive, providing a concise representation of the overall data.

9.5 Relative Risk Assessment Using Defect Types

ODC data provide a foundation of good measurements that can be exploited for a variety of new analysis methods. One of the natural extensions is illustrated by using the defect-type data to significantly enhance the methodology of growth models. This section illustrates the development of such an extension to yield relative risk assessments, which are useful during the shutdown of a product release.

Chapter 3 provided a detailed discussion on growth models. Our experience in the practical use of these methods is that they are usually more successful with data from late system test or early field test. Their application to function/component test is limited in their current form. This criticism is what troubles the development manager who finds that current growth-curve-based analysis works only too late in the development cycle to make a difference.

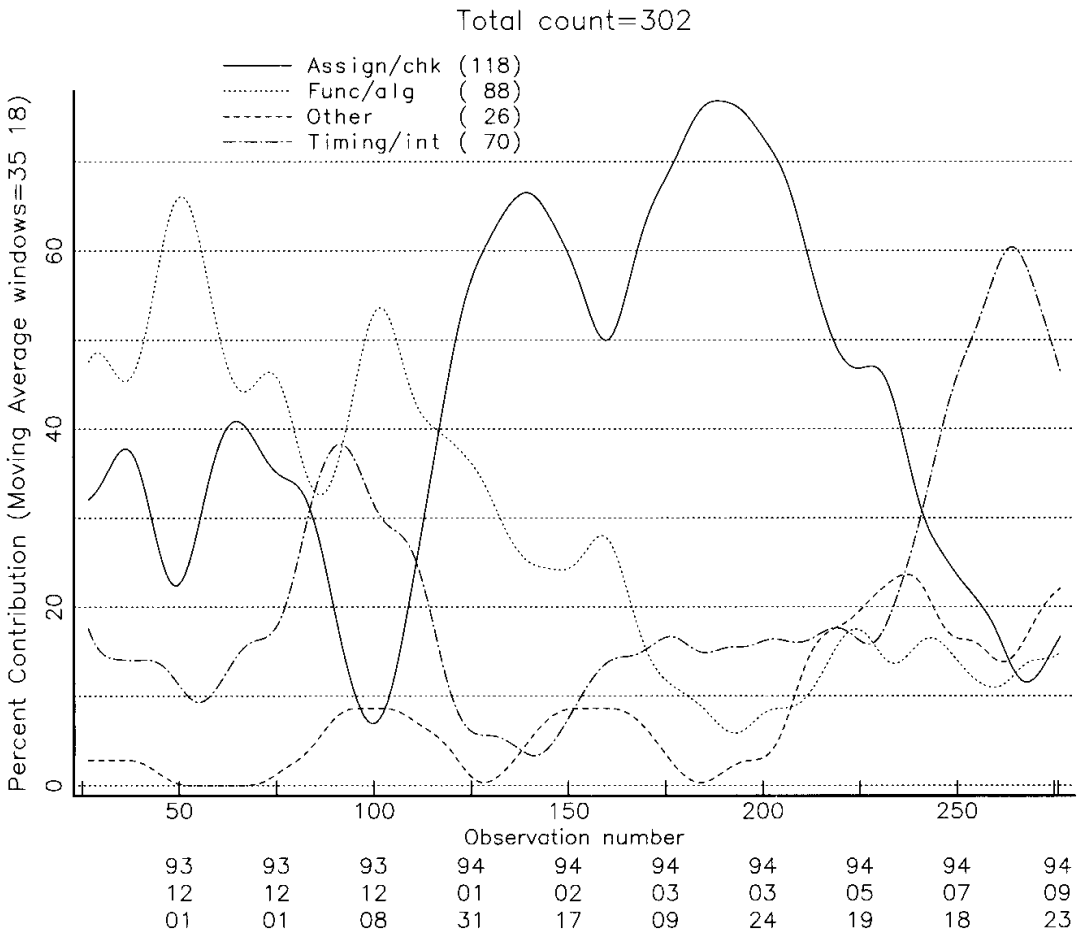

Figure 9.5 A doubly smoothed presentation of defect-type distribution.

We combine ODC with growth modeling to provide far greater insight than is commonly available with the typical growth models. There are two elements to this insight: one contributed directly from ODC and the other from assessing the relative growth of different groups of defects. Examining them against a common abscissa is what provides the additional insight to help make key qualitative inferences that drive decisions. To illustrate these ideas, we first need to step back and reflect on the qualitative aspects of a typical growth curve. Next we factor in ODC.

9.5.1 Subjective aspects of growth curves

Figure 9.6 shows a typical reliability growth curve with the cumulative number of defects on the ordinate and calendar time on the abscissa. This is one of the fairly standard representations that we use for this discussion. There are, however, several variations of these. The ordinate may represent failure rate or failure density. The abscissa may represent calendar time, execution time, test cases run, percent of test cases, etc. For this illustration, we restrict our attention to growth

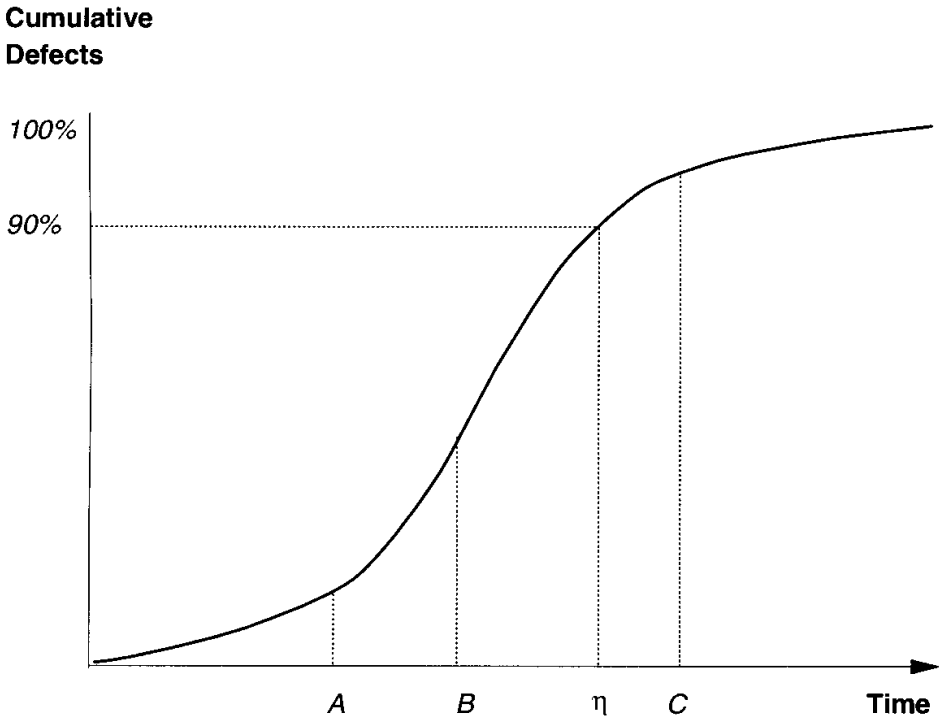

Figure 9.6 Typical growth curve.

curves with the cumulative number of defects on the ordinate and calendar time on the abscissa. The ideas that are proposed may map into other representations as well.

The growth curve in Fig. 9.6 is meant to show the span of development periods beginning with function/component test, continued into system test and possibly the early part of field life. For this discussion, we identify a few key points on the growth curve that are projected down on the time line. These are annotated as A, B, C, and η. Point A is when the test effort is ramped up, which is often visible when test takes awhile to be ramped up. Point B is computable when there is an inflection in the growth curve. In our experience, its existence is a good indicator to the modeler that the subsequent predictions become more stable. Between points A and B the curve can be accelerated depending on which fraction of function test is executable. Often there is parallel development of code and execution of test until a large part of the product can be integrated. Point C, commonly called the *knee* of the curve, is a classic indication of the begining of the end of the test effort. An ideal time to release a product is after the knee in the curve is observed, which reduces the exposure of defects found in the field. In practice, several variances occur: products are not shipped after the knee, or the knee can occur in the field. There are cases where two knees occur: one before shipping and another a few months out into the field.

It is not necessary that these three points be identifiable on each growth curve—for instance, some of them do not have an inflection point

at all. When they are identifiable, they provide important points on the time line to identify progress of the development effort. Another important point that we find useful, from practice, is to identify the point at which 90 percent of the predicted total number of defects are found. In Fig. 9.6 this is projected to the time line and annotated as η. It will be referred to as the η point. These points are sometimes visible and identifiable in a growth curve and sometimes not. This is dependent on the data and the nature of the growth curve. The η point provides a milestone in the development cycle from which to make comparative assessments.

9.5.2 Combining ODC and growth modeling

Separate growth curves can be generated for each of the defect-type categories, demonstrating their relative growth. Since several of these categories could be sparse (such as build/package/merge) it is more meaningful to collapse categories to reflect broader aspects of the development process. Function and algorithm defects capture the high-level-design and low-level-design aspects of the product. Similarly, assignment and checking tend to be related to coding quality. Thus a reasonable collapsing of the categories provides useful subdivisions of the data, making the relative comparison far more comprehensible.

To illustrate the ideas, Fig. 9.7 shows an exaggerated version of possible growth curves with collapsed categories. The curve on top shows the overall growth curve, and at time T it is hard to predict the end of development. However, when split by three groups of defect types (shown in the lower graph), there is greater insight on the dynamics of the development. It is obvious that function and algorithm defects have stabilized, given that the growth curve has reached its knee, implying that the design aspects of the product are possibly stable. On the other hand, the code quality represented by assignment and checking defects has not stabilized anywhere close to the degree that function defects did. Yet, the growth curve has well passed the inflection point, and the knee of the curve is predictable within reason. Between the growth curves of function+algorithm and assignment+checking it appears that the code parts of the product will stabilize shortly after T. Thus if the testing efforts are continued at the current rate and pace, the product should stabilize as far as these elements. On the other hand, this is not the case with the third growth curve, which represents user interface and messages defects. This curve is clearly rising very rapidly and the prediction of when it will stabilize is much further out. Since these defects are driving the volume, a separate decision has to be made regarding the risk due to these defects.

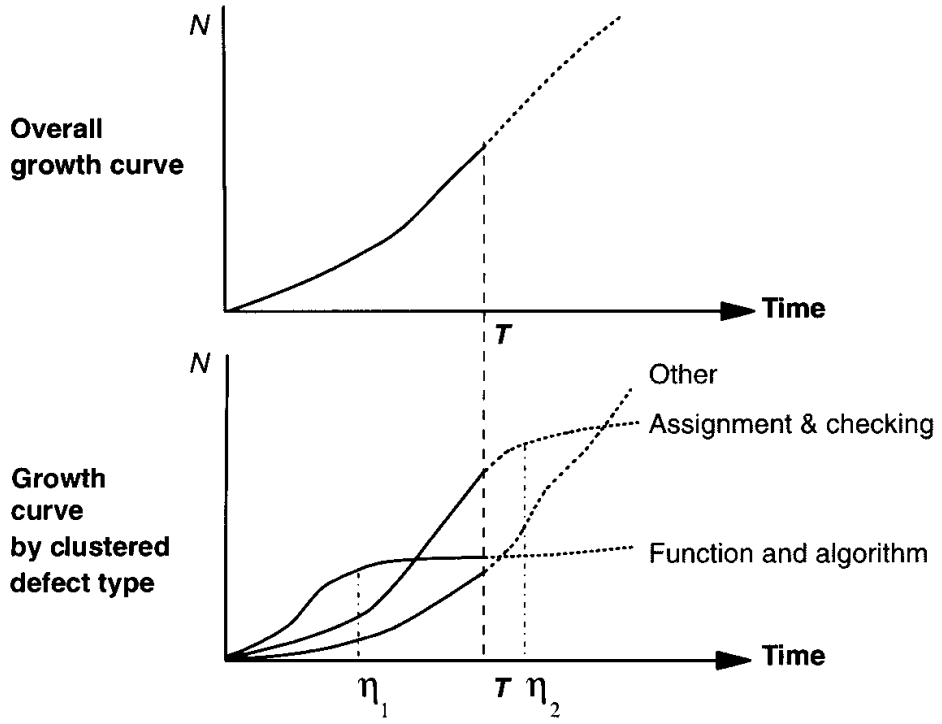

Figure 9.7 ODC for risk assessment (illustrating relative growth using ODC).

Looking at the relative growth curves, a development manager could respond to what is occurring in the product by carefully choosing the right skill mix and staffing levels during the later parts of testing. It is evident that the lead developers with the design skills are not critical at time T and could be working on the next release. Defects of type assignment+checking, representing coding issues, will continue being opened at current rates, which will need the appropriate staffing to handle the volume. However, the volume is dominated by defects of type miscellaneous that contain messages, panels, and interfaces. There is a major exposure here, since the end is not in sight, and the management has to deal with stabilizing this aspect of the product. Since the type of problems are known, management has the opportunity to respond to it by process changes and by bringing the right type of skills and experience to bear. Also, the severity of the defects can be examined to understand the risk of shipment without complete closure of the open problems.

Example 9.4 Data for this example are taken from a large project with several tens of thousands of lines of code (see ODC1.DAT on the Data Disk). The time frame includes function test and systems test, and the current date of the analysis is around two months from the desired ship date. Figure 9.8 shows the trend of the cumulative number of defects, which appears to be steadily growing, and stabilization of the product is not in sight. The developers' perception was that the product is not yet stable, and the growth model reaffirms that belief. The defect discovery rate is high, and the volume is not likely to be handled despite

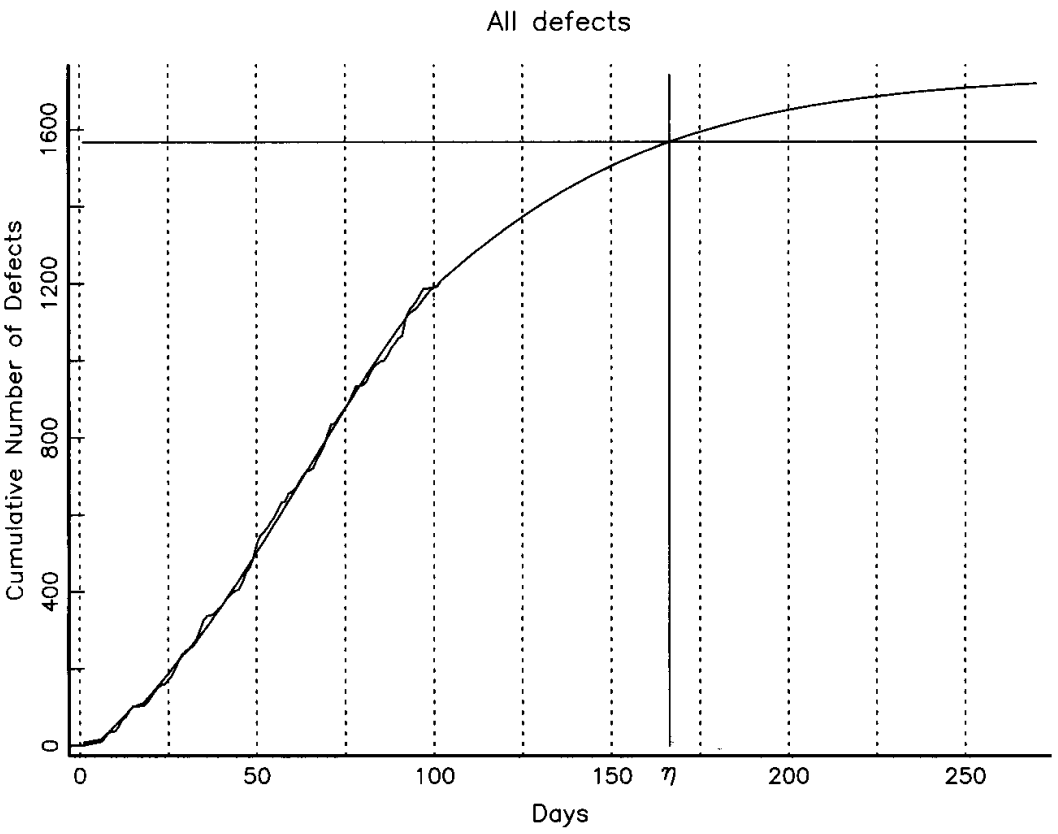

Figure 9.8 Overall cumulative defects.

increased staffing. The question of why and what to do to meet schedule while reducing the risk in the field is not evident from this level of analysis.

To gain more insight into the data, we use the ODC defect type to split the defects into three categories: *function* defects, *assignment+checking,* and all others termed *miscellaneous.* The last category predominantly consists of *documentation* and *panel+message* defects. Figure 9.9 shows the separate growth curves for the three categories superimposed on each other. Observe that the defect growth for *function* and *assignment+checking* defects is slowing down, and both of these categories are expected to stabilize soon. The growth of defects in the *miscellaneous* category, however, shows no signs of stabilization.

We decided to predict the future course of the growth curve using the inflection S-shaped model (Sec. 3.3.6),

$$N(t) = n\frac{1 - e^{-\phi t}}{1 + \psi e^{-\phi t}}$$

where $N(t)$ is the cumulative number of defects found by time t, n is the total number of defects originally present, and ϕ and ψ are model parameters related to the defect detection rate and the ratio of masked/detectable defects. We were able to fit the inflection S-curve to each of the first two categories (function and assignment/checking). The growth curve for the miscellaneous category, however, had not yet reached its knee, and it was not far enough advanced to fit an S-curve to it in the normal manner. We were, however, able to fit an S-curve by assigning a fixed value for one of the parameters ψ, using a guess, based on fitting the model to the entire data. Figures 9.10, 9.11, and 9.12 show the growth curves for

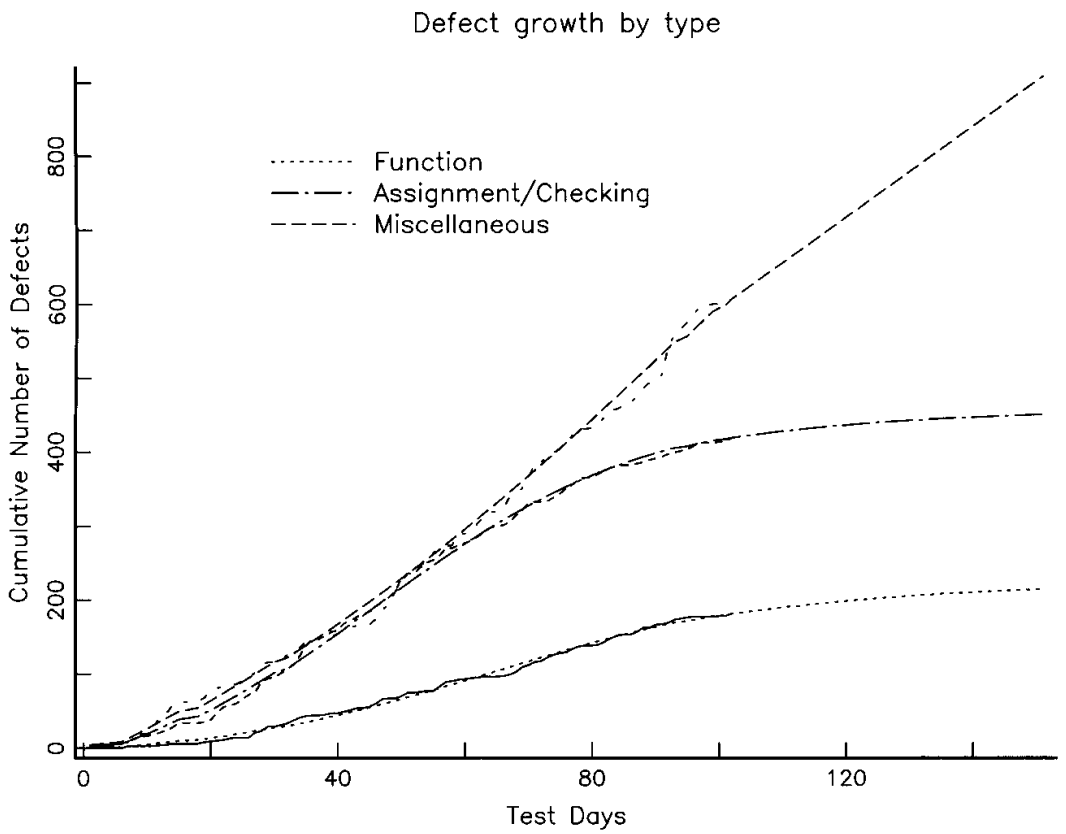

Figure 9.9 Cumulative defects by defect type.

the function defects, the assignment and checking defects, and the miscellaneous defects, respectively. In each of these curves, there is a horizontal line corresponding to 90 percent of the estimated number of defects in this category and the projection to the abscissa showing the η point. Note that the η point for function defects is around day 125, whereas that for the assignment and checking is at day 100, and the η point for the miscellaneous defects doesn't intercept the projection until all the way past day 250.

We are near day 100, and need to make decisions regarding the actions to follow in the next 60 days to meet the desired release date. From the curves it is evident that the code quality, represented by assignment+checking defects, is likely to stabilize, since we are already at the η point. On the other hand, the function defects would stabilize in the next month or so given that the η point is predicted to occur in 25 days. On the other hand, the miscellaneous defects are unlikely to stabilize given the current testing and development activity. As a result of the relative growth comparison we could say that a slight exposure exists regarding design, and the critical issue may be the miscellaneous defects. Therefore, maintaining the current test and development team will probably address the code quality aspects, but would need additional skills and resources on the other two issues. As we found out, in this project some key design skills were removed from the development team to work on the next release several weeks ago. Given this analysis and inference these skills were redirected back into this release, whereas the miscellaneous defects were a larger problem given their volume. A further analysis of those defects showed a large number of severity 3 and 4 (as opposed to 1 and 2, the higher-severity defects). Thus, releasing with several of them open was not deemed a major exposure. As it worked out, this product had

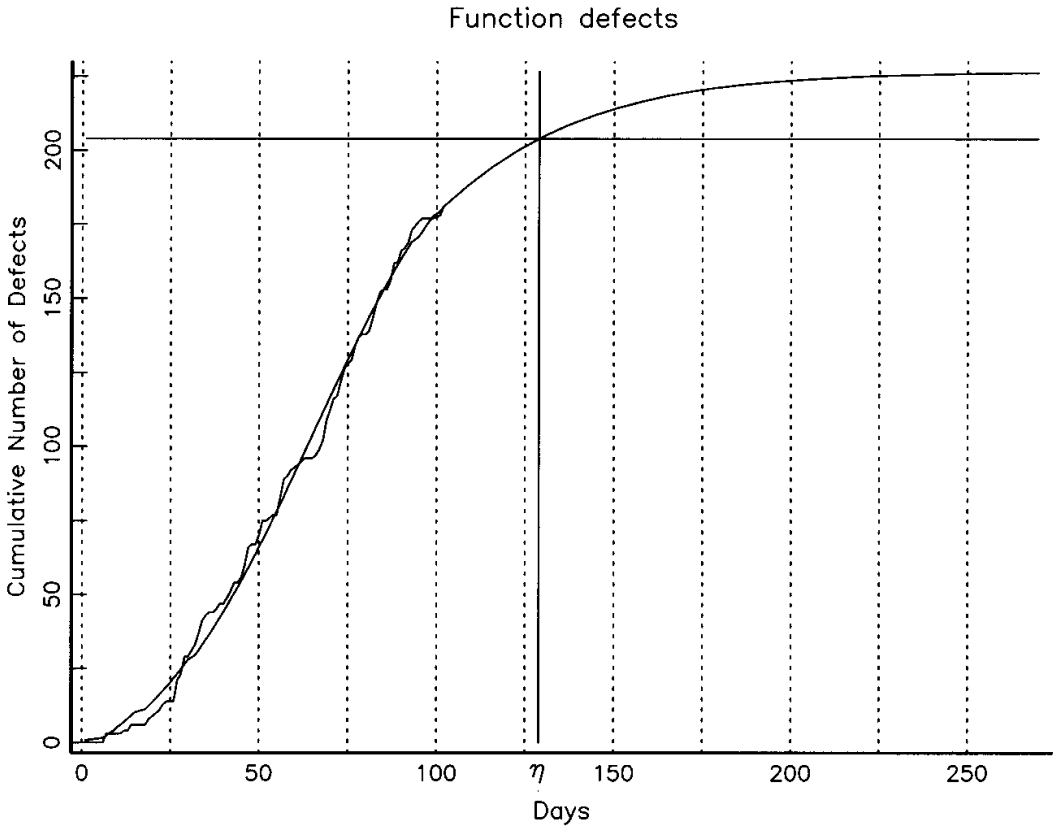

Figure 9.10 Cumulative defects: function.

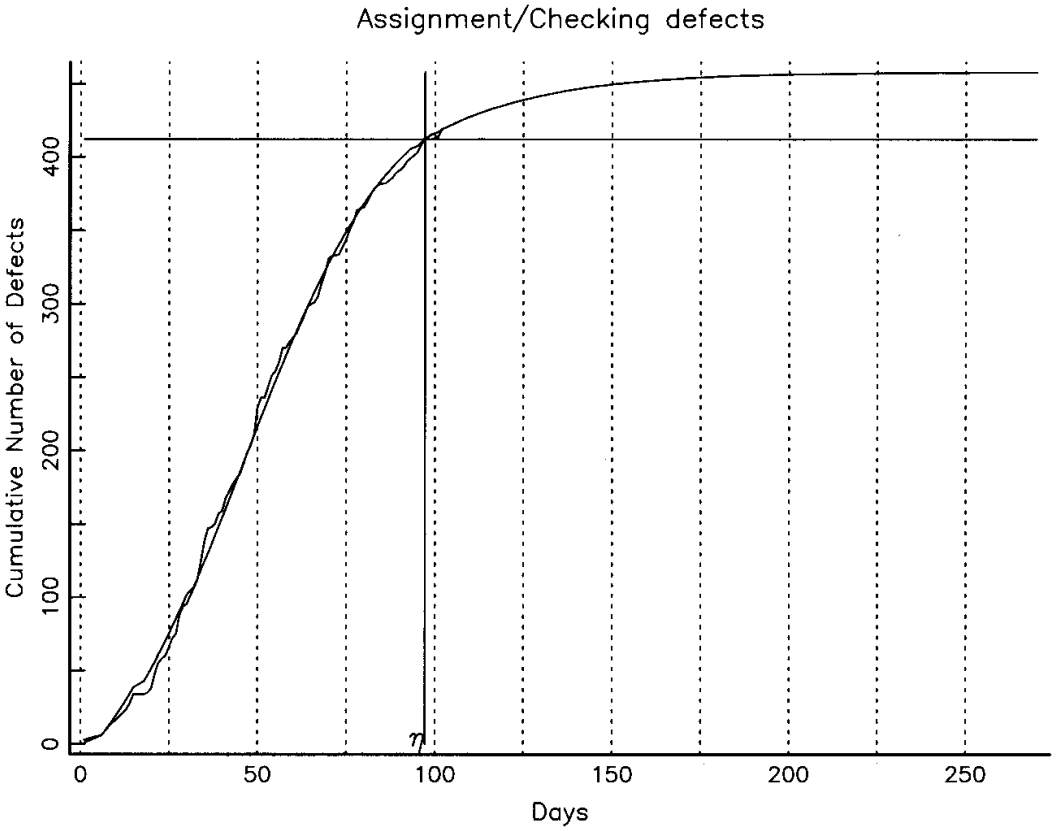

Figure 9.11 Cumulative defects: assignment/checking.

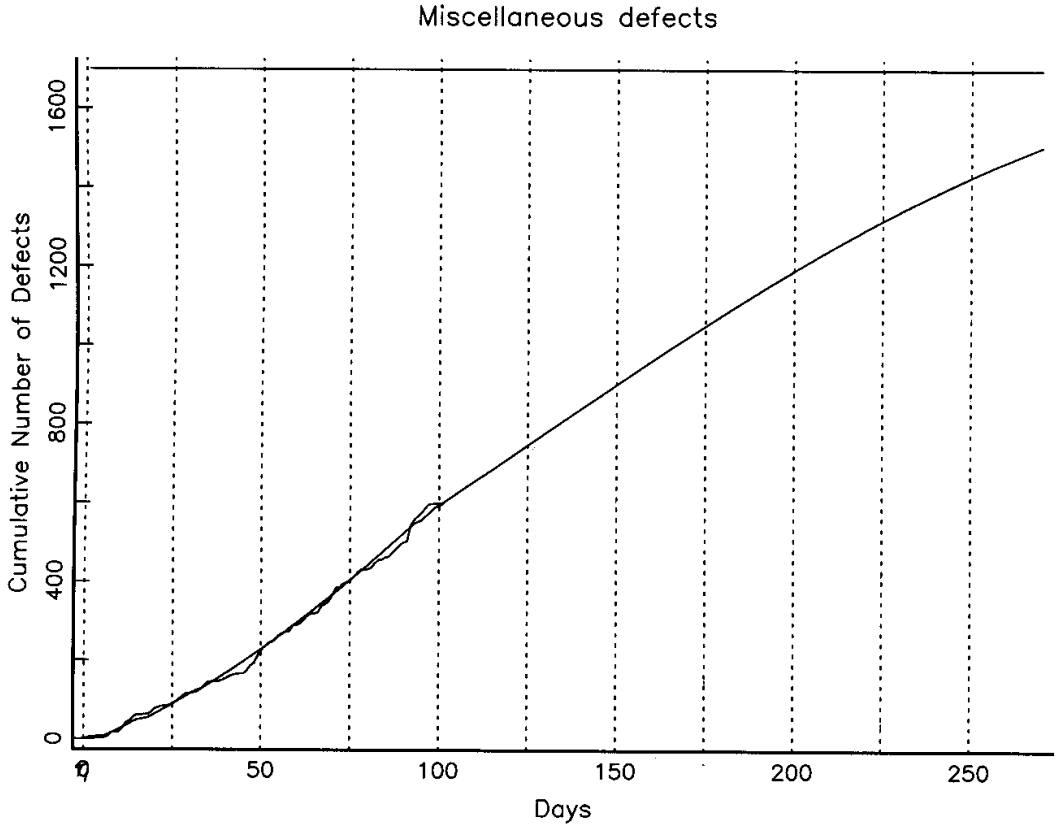

Figure 9.12 Cumulative defects: miscellaneous.

one of the higher quality ratings in the field and these decisions made during the shutdown helped not only to reduce risk but also to meet schedule and assure good field reliability.

9.6 The *Defect Trigger* Attribute

The concept of the software trigger is relatively new to the area of defect classification as opposed to defect type, which has parallels in earlier literature (although they may not have provided the measurement capability defined by ODC). The defect trigger is a new idea and therefore needs to be carefully understood.

9.6.1 The Trigger Concept

The trigger, as the name suggests, is what facilitated the fault to surface and result in a failure. It is therefore used to provide a measurement on the verification aspects of the development process. Just as the defect type provides an instrument on the progress of development, the trigger will provide a corresponding instrument on verification of testing.

We need to tear apart this activation process of faults in greater detail to appreciate the nuances of the triggering process. Before we do

that, let us reflect on the nature of software faults for a moment. Software faults are dormant by nature. They can lie undetected for a long period of time—commonly, the entire development cycle—and then be detected when a product is introduced in the field. The various verification and testing activities provide mechanisms to detect these faults. Often they verify what should work. Sometimes they are designed to create conditions that would emulate customer usage and thereby identify the faults that escaped the earlier test stages. The question to ask is: what activates the otherwise dormant faults, resulting in failures? This catalyst is what we call the *trigger.*

There are really three different classes of software triggers. These come about by the three substantially different activities of verification that are commonly employed in software—review, unit/function test, and system test. The distinction arises from how they attempt to detect faults. Review is a passive process, since there is nothing to execute. The unit/function test actively checks the implementation by execution of the code, driven by triggers that are structural and compositional. System test emulates usage under customer environmental conditions.

The review process (which includes inspection) is geared to identify pitfalls in a product using the design documents and code. The triggering mechanisms appear when someone thinks about the product, examines a design, discusses an implementation, etc. These are human triggers that result in the identification of faults by thinking about factors such as design conformance, compatibility with other releases, rare situations, and operational semantics. When a product is being tested, either unit test or function test faults are identified because a test failed or did not complete. The trigger really underlies the test case. That is the reason why a test case was written in the first place—to check for coverage, functional completeness, etc. In another sense, it is the thought behind the design of a test that is the trigger. In the case of systems test, the product is usually stressed and taken through the scenarios to which most customers would subject it. The mechanisms that identify faults are those that would also identify faults in the field, such as stress, workload, and software configurations. Contrasted with the unit/function test triggers, these are the set of things that happen to the product as opposed to what is done to it.

To put triggers in perspective, let us for a moment digress and discuss the more commonly known attributes of failures. This will help differentiate what triggers are and clarify any potential confusion. Some of the more commonly discussed attributes of failures are their *failure modes* and characteristics such as *symptom, impact,* and *severity.* The *symptom,* a visible attribute, is the characteristic displayed as a result of the failure and the net effect on the customer. For instance, the symptom attributes reported in the IBM service process have a

value set such as hang, wait, loop, incorrect output, message, and abnormal termination (abend). Fault injection experiments also use a similar attribute (often called *failure* or *failure mode*) with a value set such as no error, checksum, program exit, timeout, crash, reboot, and hang [Kana95, Kao93, Huda93]. The *impact* is an attribute that characterizes the magnitude of outage caused (severity) such as timing, crash, omission, abort fail, lucky, and pass [Siew93]. At first glance, it is not uncommon to confuse the symptom with the trigger. However, they are very different and orthogonal to each other. In simple terms, the trigger is a condition that activated a fault to precipitate a reaction, or series of reactions, resulting in a failure. The symptom is a sign or indication that something has occurred. In other words, the trigger refers to the environment or condition that helps force a fault to surface as a failure. A symptom describes the indicators that show a failure has occurred, such as a message, an abend, or a "softwait." Thus, a single trigger could precipitate a failure with any of the above symptoms or severities, and, conversely, a symptom or severity could be associated with a variety of trigger mechanisms.

The intent of capturing the triggers is to provide a measurement of the verification aspects of software development. Essentially, triggers need to conform to the same rules of ODC as did the defect types. Then they would provide a measurement on the verification process. To briefly summarize, it requires that the distribution of an attribute (such as trigger) changes as a function of the activity (process phase or time) to characterize the process. In addition, the set of triggers should form a spanning set over the process space for completeness. Changes in the distribution as a function of activity then become the instrument, yielding signatures, which characterizes the product through the process. This is the point at which the trigger value set is elevated from a mere classification system to a measurement of the process and qualifies as an ODC. The value set has to be experimentally verified to satisfy the stated necessary and sufficient conditions. Unfortunately, there is no shortcut to determine the right value set. It takes several years of systematic data collection, experimentation, and experience with test pilots to establish them. However, once established and calibrated, they are easy to roll out and "productionize." We have the benefit of having executed ODC in around 50 projects across IBM, providing the base to understand and establish these patterns. We would have liked to have one set of triggers that apply across the entire life cycle of development. Realistically, given the nature of the verification technology today, there are at least three distinctly different activities that need to be captured. Thus the triggers are in three sets, each of which span the process on which they are defined. In the following subsections we will define the triggers and illustrate their use with data from real examples.

9.6.2 System test triggers

System test usually implies the testing that is done when all the code in a release is mostly available, and workload similar to what a user might generate is used to test the product. These triggers characterize that which regular use of the product in the field would generate. They therefore apply to system test in the field.

Recovery/exception handling. Exception handling or recovery of the code is initiated due to conditions in the workload. The defect would not have surfaced had the exception handling process or the recovery process not been called.

System start-up and restart. This has to do with a product being initialized or being shut down from regular operation. These procedures can become significantly involved in applications such as database. Although this would be considered normal use of the product, it reflects the operations that are more akin to maintenance rather than regular operations.

Workload volume/stress. This indicates that the product has been stressed by reaching some of the resource limits or capability limits. The types of stresses will change depending on the product, but this is meant to capture the actions of pushing the product beyond its natural limits.

Hardware configuration and software configuration. These triggers are those that are caused by changes in the environment of either hardware or software. It also includes the kinds of problems that occur due to various interactions between different levels of software precipitating problems that otherwise would not be found.

Normal mode. This category is meant to capture those triggers where nothing unusual has necessarily occurred. The product fails when it was supposed to work normally. This implies that it is well within resource limits or standard environmental conditions. It is worthwhile noting that whenever normal mode triggers occur in the field it is very likely that there is an additional trigger attributable to either review or function test that became active. This additional level of classification by a function test or review trigger is recommended for field defects.

9.6.3 Review and inspection triggers

When a design document or code is being reviewed, the triggers that help find defects are mostly human triggers. These triggers are easily mapped to the skills that an individual has, providing an additional level of insight.

Backward compatibility. This has to do with understanding how the current version of the product would work with earlier versions or maintain n to $n + 1$ (subsequent release) compatability. This usually requires skill beyond just the existing release of the product.

Lateral compatability. As the name suggests, this trigger has to do with how this product would work with the other products within the same software configuration. The experience required by the individual should span the subsystems of the product and also the application program interface of the product with which it interacts.

Design conformance. These faults are largely related to the completeness of the product being designed with respect to the requirements and overall goals set forth for the product. The skills required for finding these kinds of triggers has more to do with an understanding of the overall design than with the kinds of skills required to ensure compatibility with other products.

Concurrency. This has to do with understanding the serialization and timing issues related to the implementation of the product. Specific examples are locking mechanisms, shared regions, and critical sections.

Operational semantics. This has to do largely with understanding the logic flow within the implementation of a design. It is a trigger that can be found by people who are reasonably new but well trained in software development and the language being used.

Document consistency/completeness. This has to do with the overall completeness of a design and ensures that there is consistency between the different parts of the proposed design or implementation. The skill is clearly one that requires good training and implementation skills, but may not require significant in-depth understanding of the products, dependencies, etc.

Rare situation. These triggers require extensive experience of product knowledge on the part of the inspector or reviewer. This category also recognizes the fact that there are conditions peculiar to a product that the casual observer would not immediately recognize. These may have to do with unusual implementations, idiosyncrasies, or domain specific information that is not commonplace.

9.6.4 Function test triggers

One of the primary differences between function test and the other set of triggers is that the meaning of the trigger needs to be more complex. Since the defining question for the trigger "why did the fault surface?" would result in the answer "test case," the definition of trigger involves a finer refinement.

The question becomes "why did you write the test case?" Thus the triggers that are identified reflect the different motivations that drive the test case generation. Therefore, it is feasible to actually identify the triggers for each test case as it is written. It is not necessary that the triggers be classified only after a fault is found. Each test case would then have a trigger associated with it. The trigger distribution is then reflective of the different methods and coverages intended through the test plan. These test cases should also be mapped into the white box and black box of testing.

Test coverage. This refers to exercising a function through the various inputs to maximize the coverage that is possible of the parameter space. This would be classified as a black-box test trigger.

Test sequencing. These are test cases that attempt to sequence multiple bodies of code with different sequences. It is a fairly common method of examining dependencies which exist that should not exist. This is also a black-box test.

Test interaction. These are tests that explore more complicated interactions between multiple bodies of code usually not covered by simple sequences.

Test variation. This is a straightforward attempt to exercise a single function using multiple inputs.

Simple path coverage. A white-box test that attempts to execute the different code paths, increasing statement coverage.

Combination path coverage. Another white-box test that pursues a more complete signal of code paths, exercising branches and different sequences.

9.6.5 The Use of Triggers

Once we understand what the trigger data capture we can begin to appreciate the different potential uses of the trigger concept. Triggers can be used on their own to measure the effectiveness of the verification phase, and they can also be used in conjunction with the other categories in ODC to provide further insight into the cause of a process situation. In the following we will illustrate the use of triggers by showing examples from pilot studies conducted at IBM. Some of these examples will use the trigger category in conjunction with the defect type to illustrate the dynamics occurring in a particular process phase. The type of trigger that is used will change depending on which process phase we are looking at. Triggers can also be used with field data to recognize the different environmental stresses that are placed on a product.

The concept of the trigger provides insight not on the development process directly, but on the verification process. Discussing a few examples that use triggers gives us a much better understanding of how this works. We will do so by examining a few specific situations and illustrating the kind of inferences that can be drawn from the trigger data. We begin by first examining a project in the high-level-design phase in Example 9.5 and then follow it up with triggers in the field in Example 9.6.

Example 9.5 This example is from the high-level design inspection of a middleware software component (see ODC2.DAT on the Data Disk). The component's API was intended for use with several other products and vendor applications. The example uses the idea of triggers combined with the defect type to illustrate measuring the effectiveness of the inspection. Figures 9.13 and 9.14 show the distribution of defect types and triggers subgrouped by defect types. Let us factor into these distributions our expectations and critique the situation. Studying the defect-type distribution, we notice that it is typical of what a high-level inspection should produce: the number of function defects is fairly large—the mode being documents, which is understandable given that it was a document review. The trigger distribution shows that the largest number of defects were found by the operational semantics trigger. This again is explainable given that completeness and correctness are major issues considered in a high-level-design inspection. Given that this is a middleware component that will be used by several other products, interfaces are key. Furthermore, lateral compatibility is important since it is middleware. What is surprising is that there are a small number of interface defects found using the lateral compatibility trigger. Defects found using the lateral compatibility trigger are mostly function defects. Given the nature of this component, this raises serious questions regarding the skills of the inspection team, particularly from a cross-product perspective. A check of the team's membership found few people with that particular skill.

In this case, an additional review was requested to fill the gaps, and two experts (including an IBM fellow) found the defects shown in Figs. 9.15 and 9.16. It was

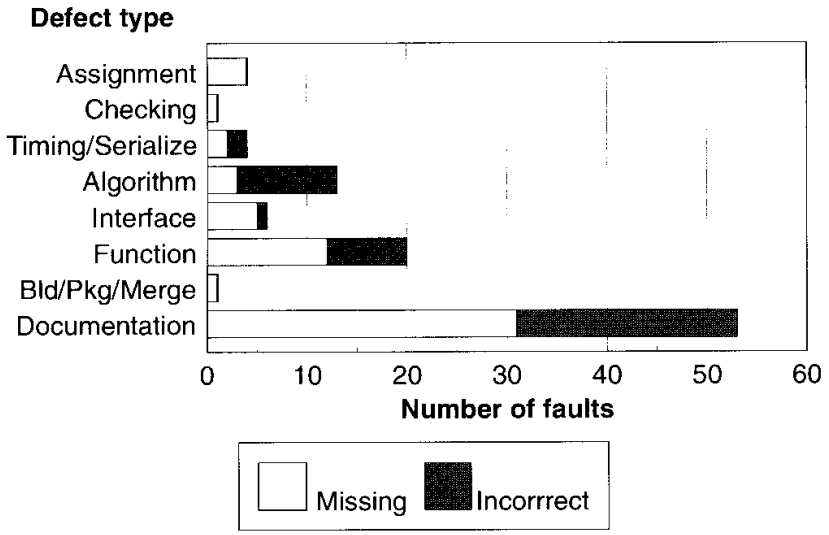

Figure 9.13 Defect-type distributions at first high-level design.

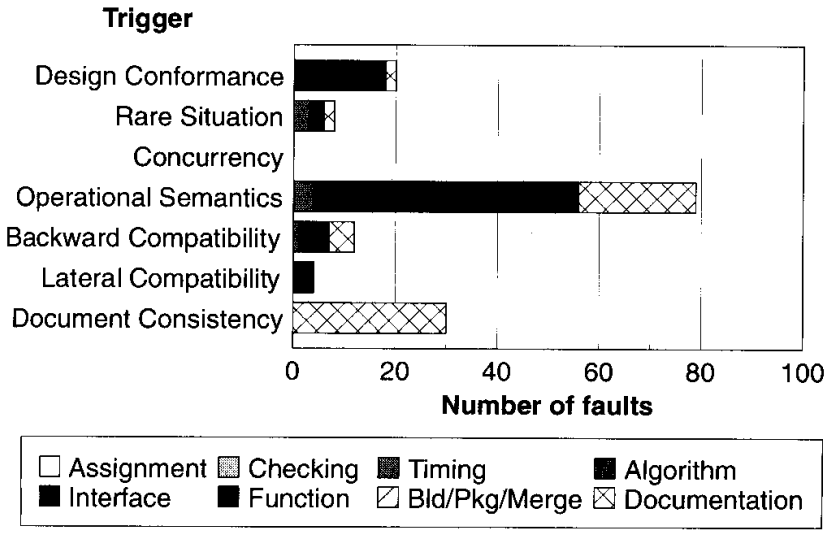

Figure 9.14 Defect-trigger distributions at first high-level design.

possible to advise these experts to focus their efforts along the dimension that we suspected was weak rather than redo the entire inspection. The second defect-type distribution, like the former, is characteristic of a high-level design inspection—namely, large numbers of document and function defects. From the trigger distribution in Fig. 9.16, it is clear that a substantial number of lateral compatibility triggers were being used to identify defects. In addition, the lateral compatibility trigger found many different defect types, indicating a more detailed review. In this case, 102 defects were found due to the additional review—in precisely the areas where deficiencies were suspected. The savings due to this early detection exceeds more than $1 million in development and service costs.

Note that this is identified using the cross product of the defect type and the trigger. The feedback was available right after the inspection—extremely fast and

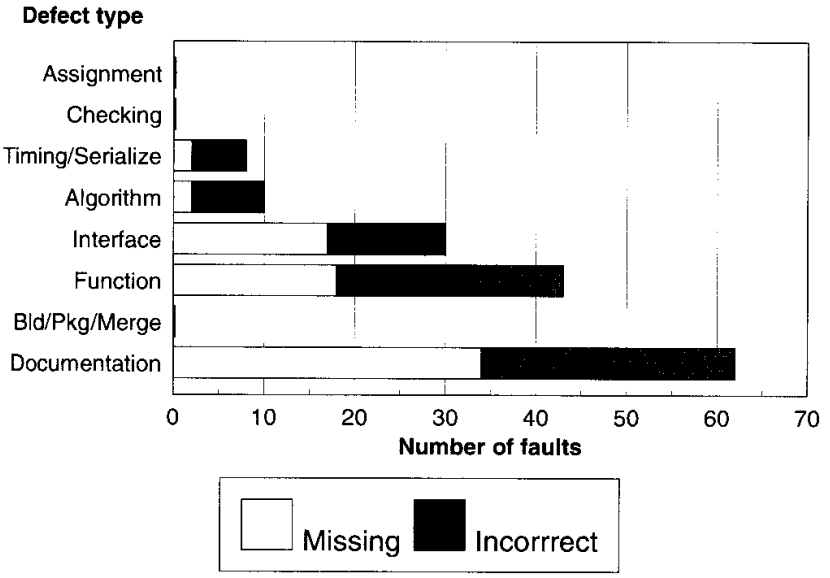

Figure 9.15 Defect-type distributions at second high-level design.

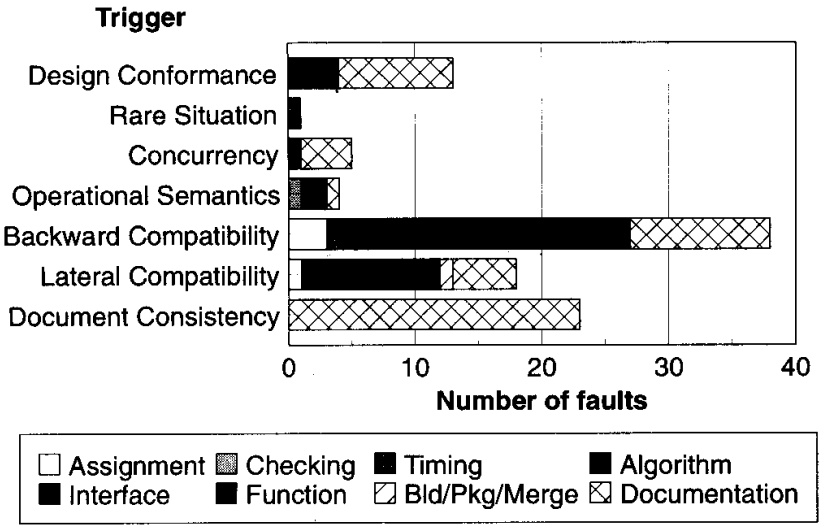

Figure 9.16 Defect-trigger distributions at second high-level design.

actionable feedback quite unusual for the software industry. Traditional methods rely on the volume of defects being found compared with earlier statistics, which only make broad inferences. Given the wide variances of defect injection rates, the credibility of such inferences is often questioned. Classical root-cause analysis, after considerable expense, would reveal the nature of defects found, but could not react to the omissions. Furthermore, two-dimensional cross products are hard to correlate mentally in qualitative analysis.

Example 9.6 This example shows trigger distribution of defects found in the field. The data are from a two-year period of a product in the field. Two years is a reasonable length of time, since the bulk of the defects are usually found during this period. There are several inferences to be drawn from Fig. 9.17. First, the trigger distribution changes as a function of time for the different triggers. Recall the ODC necessary and sufficient conditions, which need to be empirically established. The change in distribution as a function of time signifies meeting the necessary condition. Second, the trigger distribution indicates to us when a particular product is likely to have the maximum defects from a particular trigger. This information is vital to develop testing and service strategies. For instance, the fact that hardware configuration and stress-based triggers tend to peak in later years would allow for a testing strategy that focuses on weeding out the triggers that peak earlier. This could be exploited if there is an opportunity to refresh the release in the field with a better-tested upgrade within the next year or so. Third, knowing the trigger profiles allows us to project the number of defects that appear in the field better. There are several techniques that could be developed in the future. However, for a simple intuitive understanding, consider comparing trigger distributions of the past release's field to this release's system test. A better coverage in system test of areas that the customer hits hard would result in a smaller field fallout. This assessment, combined with the detection profile in the field, allows us to make more accurate projections of the number of defects to be found in the first, second, and third quarters following a release. Having a better projection based on triggers allows us to do service planning and staff the functions better by skill group.

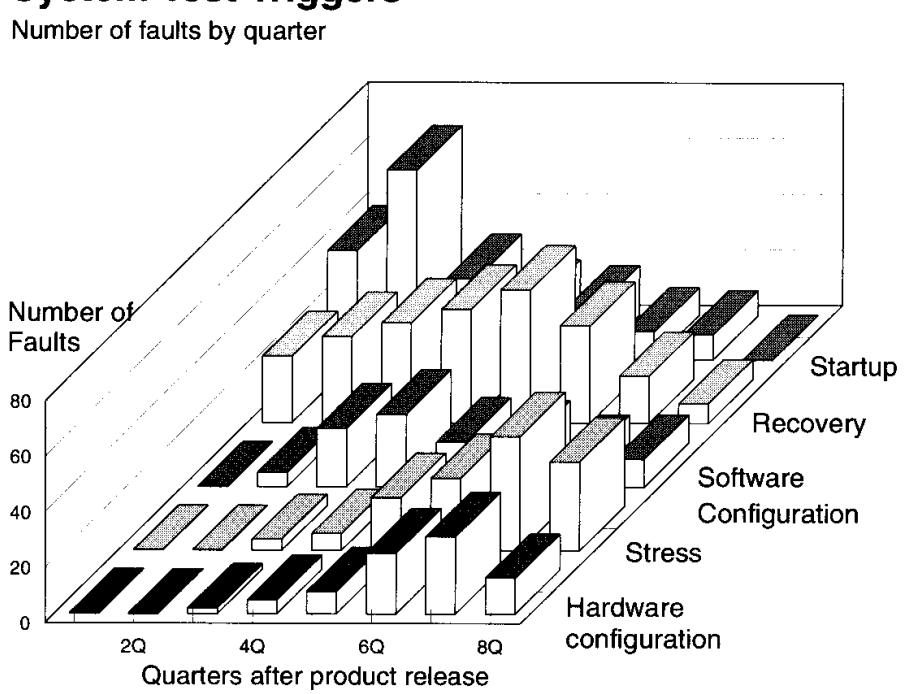

Figure 9.17 Triggers of defects found in the field. Subsets of system test triggers showing number of faults by quarter.

Although this section does not show data on trigger distribution through function tests, the discussion on the review triggers and the system test triggers should provide good initial understanding of how triggers work. For a further discussion of triggers there are several articles that are recommended. The ODC paper provides an overview of the concept of the trigger and illustrates some data on both the review and test process [Chil92]. A more detailed discussion on the test process exists in [Chaa93], where the trigger concept is expounded and examples from several pilots are included. For an understanding of how the trigger concept can be levered to guide fault injection and characterize the field performance of the system, [Chil95] provides a lot of information.

9.7 Multidimensional Analysis

This chapter has so far primarily focused on two attributes: namely, the defect type and the trigger. Each of these attributes has a specific purpose from a measurement perspective. The defect type is a measure related to the development process, and the trigger to the verification process. Both attributes are of a causal nature: the defect type is "what is the meaning of the fix," and the trigger is "what was the catalyst to find the defect." The fact that the value set conformed to the properties of ODC is what made these classifications into measurements.

ODC does not imply only these two classification attributes. In fact, any classification can be considered ODC if the attribute-value set conformed to the necessary and sufficient conditions. However, to build a measurement system with a firm footing, a substantial amount of piloting and experimentation needs to be conducted to verify the properties. In theory, there could be any number of attribute-value sets, each of which meets ODC requirements—providing a large space of measurements. The attribute-value sets could address different issues about the product or the process. Each attribute does not have to be independent of the other, but the values within an attribute need to be. However, having attributes that are highly dependent on each other, with highly correlated value mappings, does not make for useful measurement.

Having multiple attribute-value sets that conform to ODC properties provides a rich measurement opportunity. It allows for the analysis using multiple attributes to be easily interpreted, be they correlations, subsets, functional relationships, etc. In fact, the multidimensional nature of this measurement provides fertile ground for data mining. Having a large body of such data from several products or releases makes it particularly amenable for developing models and examining properties, trends, and characteristics. This is useful to develop an understanding of a development experience, as well as to characterize teams, environments, practices, etc.

At first glance it seems relatively easy to come up with attributes that potentially could be useful. However, it takes considerable experience to find the truly useful attribute-value sets. Thus, it is not uncommon to see teams coming up with recommendations to improve the classification or add new attributes. The danger in arbitrarily adding to or changing the measurement system is that before one is verified to be ODC, widely deploying it could be an unproductive exercise. Furthermore, it is important not to thrust too many attributes on a development team and scare them away. We have chosen to introduce just four new attributes over and above what is traditionally in place.

Figure 9.18 shows six attributes and their purposes. The italicized attributes are recommended as starters for ODC. Defect type, trigger, source, and impact are four attributes that are introduced as new additions on top of the standard measurements that most defect-tracking mechanisms provide. To help put this in perspective, two commonly used attributes—phase found and severity—are also shown. Most defect-tracking tools commonly have between five and fifteen such attributes to categorize defects. Many of them never get used, or when used, it is not uncommon for organizations to never use the data! In sharp contrast, the ODC approach uses few attributes, but very carefully designs them.

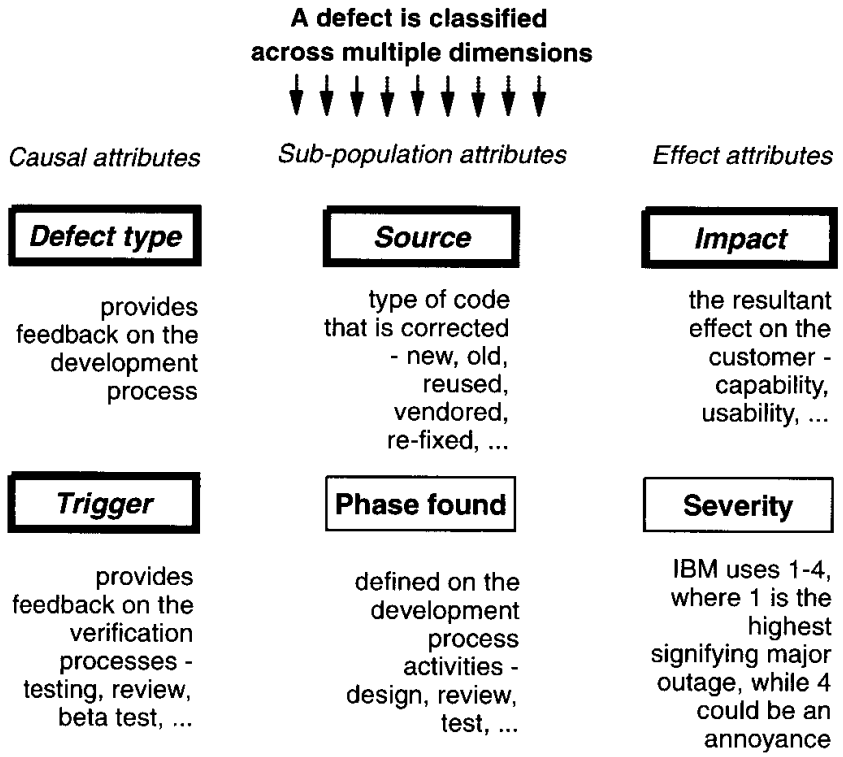

Figure 9.18 Dicing a defect, generating multidimension data.

These six attributes are further grouped under the broad areas of cause, effect, and subpopulation identifiers. This helps us visualize a framework on which these attributes could be used. Since every defect will have each of these classifications, analyzing a collection of defects allows us to statistically develop models that relate cause with effect. Furthermore, the subpopulation identifiers allow us to slice and dice the data and provide greater visibility of the underlying issues within a software development process.

One of the key advantages of having the data along the dimensions shown in the figure is that this provides a good minimal set for a variety of purposes. For example, root-cause analysis usually requires a few cause-and-effect attributes, particularly if we want to diagnose an issue from a customer's perspective. Thus, if we want to understand how to best improve the serviceability of a product, the ODC data sets provide a clear and efficient diagnosis search path. The impact attribute would be used to create a subset including reliability, maintainability, and serviceability defects. The subset is then examined using the defect type and trigger to understand the reason such defects escape into the field and the kind of faults that are being committed to cause the impact. This understanding is then translated into recommendations for actions. To further reduce the search space to apply such actions, the source category can be used to explore whether certain kinds of code—new code, old code, vendored code, etc.—dominate the volume of such problems.

The data requirements document [IBM95] provides the details of the attributes and values of the latest version of ODC. This includes values and definitions for the attributes that are currently tested to meet the ODC requirements. The document also includes any enhancements made in ODC, applied across different types of development—software, hardware, information, etc. It also includes new attributes as they are invented and the uses developed for them.

Having multidimensional ODC-based measurements, not just classification, opens up a new world for analysis. Fairly systematic methods for diagnosis can be established for the commonly arising problems. At the same time it keeps open opportunities to explore via data mining methods.

9.8 Deploying ODC

The deployment of a technology such as ODC requires careful thought and considerable insight into the means of process insertion. Most practitioners would recognize that technology transfer is a difficult business. Process transfer is yet another order of magnitude harder and especially so in software. This is because, unlike a technology that can be captured in a tool or a design which impacts a product group, process transfer in software requires that every programmer change a little. Although the change may be minor in terms of the actual work a programmer does, getting acceptance of the concepts and buy in through the organization is a major undertaking.

Unless programmers quickly see the value of ODC, it is hard to sustain their interest and commitment to provide good data. This quickly becomes an exercise in managing all the processes to execute ODC. Knowing the processes and having the necessary skills are the minimum requirements. Being able to quickly recognize when they are not working and reacting to them effectively is the difference between success and failure. At Watson Research, we started as technologists and teamed with our divisional partners to do deployment. It quickly became evident that the split was artificial. Both teams needed to understand the technology and the nuisances of the real world to be effective. We learned to create processes that were necessary and developed schemes to maintain and troubleshoot them.

Figure 9.19 identifies some of the key processes. It also divides the range and scope of deployment into *pilot, staged production,* and *production,* indicating the growth of deployment in an IBM lab. The idea is that initially an organization would usually start off with a pilot project, almost as a trial. These usually last between three months and one year, and become the proving ground for ODC at the pilot stage. Most of the processes were owned by Watson Research, and the

responsibilities of the participating lab were limited. As we made progress, we would develop the skills in the organization to own more of the processes, reducing the responsibility and involvement of the research team. In the best cases, we were able to obtain ownership of more than 70 percent of the processes in around 18 months. Results from the use of ODC in projects across the company are described in [Bhan94]. At the end of 1995, we had close to 50 projects in about a dozen labs. Two labs could be considered to be well into staged-production.

The cost of ODC is quite low. There is an up-front cost in terms of modifying the tool set and providing education. The execution cost is dramatically lower if a developer is already using a change control system. This is because most change control systems require that the programmers update a panel, and ODC requires only four additional fields (in most cases), which is a small delta cost. The subsequent costs are incurred in analyzing the data, which is mostly tool cost and either management or technical review which runs approximately two hours a month (on average). This can be rolled into existing quality programs or quality circle efforts, thereby not requiring additional effort.

When ODC is used to enhance the quality circle of the defect prevention process (DPP) [Mays90] significant savings can be accrued in analysis costs. Typically, DPP-related efforts cost in the range of one person hour per defect. Imagine four people in a room analyzing defects. They usually do a detailed root-cause analysis of around four or five defects in an hour. This one hour usually includes not only qualitative analysis but also identifying a potential solution and writing it down as an action item for the organization to execute. Given such high costs, it is again not common for organizations to be able to do

Pilot Projects	Staged Production	Regular Production
Lab Ownership	**Lab Ownership**	**Lab Ownership**
Classification	Classification	Classification
Decisions	Decisions	Decisions
Actions	Actions	Actions
Watson Ownership	ODC Education	ODC Education
ODC Education	Advocacy	Advocacy
Advocacy	Data collection tools	Data collection tools
Data collection tools	Process definition	Process definition
Process definition	**Watson Ownership**	Analysis
Analysis	Analysis	Feedback
Feedback	Feedback	Databases
Databases	Databases	**Watson Ownership**
Consultation	Consultation	Consultation

Figure 9.19 Deployment of ODC.

DPP on every defect, since they usually run into thousands. The ODC classification, which extracts cause and effect, usually takes only two minutes when done retrospectively. Granted that the granularity of the measurement is very coarse, its low cost allows full coverage over the defect population. The analysis of these data provide a statistical means to do causal analysis by associating cause and effect. This now occurs not on each defect, but on a collection of them, and is appropriately timed at the exit of a development phase. Since the analysis of the data (which may even be qualitative) is amortized over several of them, the overall cost is reduced by about an order of magnitude (according to our estimates) when including all the time costs involved.

9.9 Summary

Orthogonal defect classification fundamentally improves the technology for in-process measurement for the software development process. This opens up new opportunities for developing models and techniques for fast feedback to the developer, thus addressing a key challenge that has been nagging the community for years. At one end of the spectrum, research in defect modeling focused on reliability prediction, treating all defects as homogeneous. At the other end of the spectrum, causal analysis provided qualitative feedback on the process. The middle ground did not develop, primarily because the basic discoveries establishing the feasibility were not yet there. This work is built on some fundamental breakthroughs, which show that certain cause-effect relationships are measurable. Furthermore, the measurement system is definable on the semantic information contained in the defect stream. ODC provides the basic capability to extract signatures from defects and infer the health of the development process.

Our experience with ODC indicates that it can provide fast feedback to developers. Developers find this a useful method to gain insight they did not have before. It also provides a reasonable level of quantification to help make better management decisions to significantly impact cost and opportunity. There are several levels of analysis and feedback that can be built on ODC. The published literature discusses trend analysis, relative risk reduction, data mining, prediction methods, and assisting root-cause analysis. When used to assist root-cause analysis, it can cut the cost by a factor of 10 compared to traditional methods. It can be used as a general diagnostic tool retroactively to assess problem situations in development organizations. ODC has since been extended into information development and non-defect-oriented issues, and has been applied to hardware (microcode) development.

Problems

9.1 Define a process with three stages. Take the eight defect types from Sec. 9.4 and draw the expected defect type distribution to signify the ideal signature. Explain why the mode occurs in each of the phases and its relative size compared to the category that is second to the mode.

9.2 For the process described in Prob. 9.1, increase the process to five stages. Now modify the distributions to accommodate the addition of process stages. Redraw the distributions with the abscissa to represent number of defects, instead of percent of defects.

9.3 Consider release 2 of a product made up of 80 percent of old code (from release 1) and 20 percent of new code developed in release 2. Defects are found during testing from the old code and the new code. Write down an expectation of the defect-type distribution of the new code and the old code. Defend your position. If we had information on the defects found in the field use of release 1, how would that influence your expectations? Develop a hypothetical example.

9.4 Take data from `ODC3.DAT` on the Data Disk. Develop defect-type distributions for the design phase and the code/unit-test phase. Compare the distributions and assess the trends in the changes. Next compute the proportion of missing to incorrect defects from the two phases and explain the difference between the two phases. Develop feedback to the development team based on your analysis.

9.5 Suppose we have triggers of defects found during the system test and the first six months' usage in the field after product release. Argue what the differences or similarities should be between the trigger distribution of the system-test defects and the field defects. Would they be (*a*) identical, (*b*) complementary, (*c*) unrelated? Explain why.

9.6 Study the trigger distributions of defects reported in reference [Chil95]. What would you recommend to the development team that is developing the next release of this product?

9.7 How would your recommendations change if you know that the triggers from the defects in Prob. 9.6 come from three different releases? For simplicity let us assume that all the system-test-triggered defects are from release 1, the function-test-triggered defects from release 2, and the others from release 3. Now, if we were to make a recommendation for the development of release 4, how would it be different from those of Prob. 9.6?

9.8 Develop defect types for the art of writing a paper. To do so, use the experience of writing a technical paper to identify defects.

a. Define what a defect means in this activity.
b. Define the parallel to the defect type and trigger for these defects.
c. Collect defects from a paper-writing project and classify them by defect types and triggers.

d. Develop the distributions as a function of the phases of writing a paper. If the phases cannot be clearly identified, then develop them as a function of time.

9.9 Take the data sets of `ODC3.DAT` to `ODC6.DAT` from the Data Disk and analyze the defect-type distribution by doing simple trend analysis to raise issues that should be of concern. To do so, develop distributions (such as in Figs. 9.13 to 9.16) to obtain insight. Specifically, generate defect-type distribution as a function of phase, and the trigger versus type distributions. Additional insight can be gained by looking at additional attributes in the data sets. More sample data sets are available from the Web site: `http://research.ibm.com/softeng`.

9.10 Take the data for Example 9.4 and try to do a similar analysis on relative risk using a different grouping of defect types. What are your conclusions? Are they different from the stated example?

9.11 What is the parallel to the risk assessment (Sec. 9.5) using triggers instead of using defect types? Discuss applications for this new approach.