Contents

12

The Cost Effectiveness of Telecommunication Service Dependability

Y. LEVENDEL
AT&T Bell Laboratories

ABSTRACT

In switching software applications, service quality has traditionally been achieved by the combination of two strategies: high reliance on defect elimination and fault recovery/tolerance. In turn, defect elimination has proven to be costly in staffing, and the implementation of fault tolerance has resulted in high hardware costs, by exclusively relying on proprietary hardware and by using monolithic recovery techniques external to the applications to achieve high quality service. The driving force for this strategy were: no unscheduled downtime, deferred maintenance, easy restart when needed, and easy growth and de-growth. While these objectives are still attractive, they can today be achieved in a more cost-effective way by increased reliance on standard software fault recovery components distributed closer to and inside the applications, and by using hardware sparing recovery at the system level. A recent trend toward rapid software customization will limit traditional software recovery techniques where absolutely necessary to satisfy performance requirements.

12.1 INTRODUCTION

Electronic telephone switching systems are expected by telephone customers to satisfy high reliability requirements in order to assure high quality service. Ideally, the system is expected to perform continuous, uninterrupted operation, and in principle, switching systems are designed for long range continuous operation. In practice, however, target thresholds are set in the form of maximum allowable failure rate, and different sets of requirements are applied

Software Fault Tolerance, Edited by Lyu

for different types of situations. Under these conditions, the products are subjected to the following requirements.

1. Operation without Catastrophic Failures
   In the USA, the system downtime is expected to be less than 1 hour for 20 years of continuous operation.
2. Operation without Non-catastrophic Failures
   The system is expected to provide call processing from beginning to end, but is allowed to interrupt at most 1 call in 10,000. Dial tone must be provided with a delay not to exceed 1 sec.
3. Non-intrusive Office Administration and Maintenance
   The data base, the software, and the system configuration can be changed under specific sets of circumstances without affecting call processing operation. These include: customer data modification, minor software updates, hardware growth, and hardware preventive maintenance and repairs.
4. Intrusive Office Provisioning
   During intrusive Office Provisioning, such as major software or hardware upgrades, it is acceptable for a customer not to be able to initiate new calls after the beginning of the provisioning operation. However, a strict requirement applies to calls in progress: they must be maintained as long as the customers desire during the entire provisioning operation.

A large number of additional requirements further specifies internal operating conditions, such as overload conditions, performance under limited hardware availability, fault propagation restrictions, etc.

Large systems, such as electronic telephone systems, are composed of a collection of hardware and software elements, each one with a certain degree of imperfection. A continuous analysis of field failures is essential to guide the effort of improving system performance. For the 5ESS®switch, the typical partition of field failures is given in Table 12.1 [1] [Cle86, Lev90a]. The proportions may vary from product to product and depend on product maturity. Also, the data in Table 12.1 must be first qualified by the fact that, with the exception of undesirable multiple counting, the data tend to represent the first occurrence of a failure, which is closer to fault enumeration.

Table 12.1 System failure attribution

Failure Type	Usual Rate	Mature Product
Hardware	30%-60%	22%
Software	20%-25%	1%
Procedure	30%	62%
Other	-	15%

Procedural errors which dominate mature systems may be due to personnel's inexperience, inadequate operations manuals or inconsistent behavior of multi-vendor equipment. Hardware failures dominate the other failure types. However, this does not reflect the current industry

[1] The rightmost column is the Federal Communication Commission Data for the 5ESS®Switch (Fourth Quarter 1992).

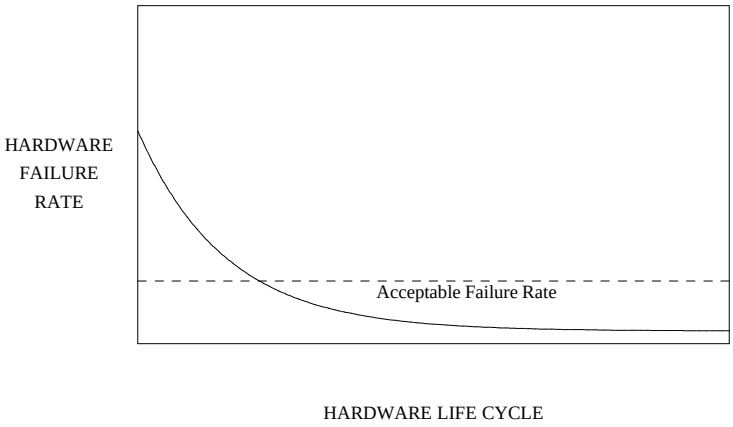

Figure 12.1 Hardware failure rate improvements

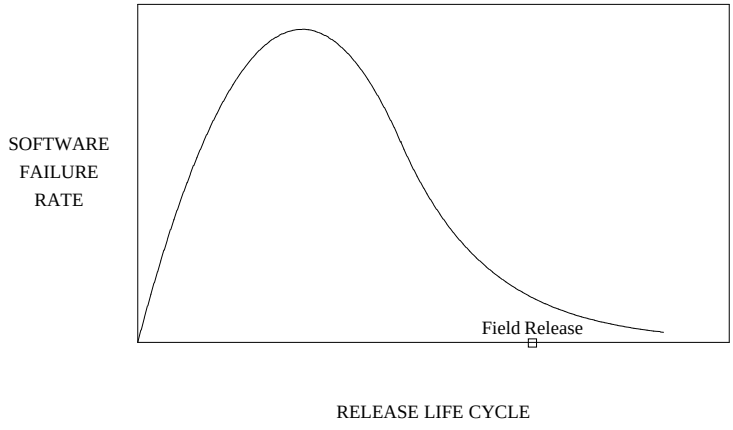

Figure 12.2 Software failure rate improvements

experience in commercial hardware where the hardware failure rate is much lower [Har93]. This is mainly due to the fact that large segments of telephone switching hardware are highly proprietary and its reliability improvement process is slower than commercial hardware which is produced in larger quantities. Commercial hardware may reach higher dependability earlier than proprietary hardware. Over the product life cycle software defects represent a significant overhead.

Over the years, however, 5ESS® hardware failure rates have significantly dropped because of design improvements (Figure 12.1), and software quality improves even more rapidly (Figure 12.2). We use here the classic distinction between faults and failures [Lap92]. Actual field data can be found in a separate publication [Lev90a]. New releases of hardware or software may cause a resurgence of failures.

Recently, the Federal Communication Commission released outage statistics for the major telecommunication equipment suppliers in the US. As shown in Figure 12.3, the AT&T products surpass the products of its competitors by a significant margin. The outage data covers all

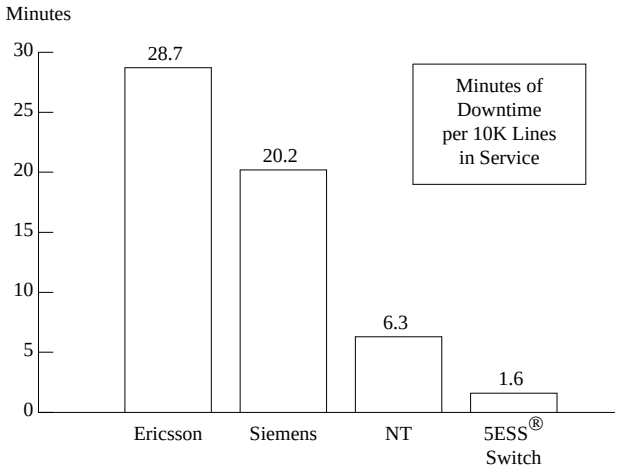

Figure 12.3 US outage data for major telecommunication suppliers (1992)

causes (hardware, software, operator errors,...) is normalized to 10,000 telephone lines to provide a common scope of customer impact of the outages. In fact, it took a decade to achieve this result, and Figure 12.4 can testify that the road to success was not a simple one. 5E4, 5E5, 5E6, 5E7, and 5E8 are major successive releases whose development spanned most of the recent decade and which have had enough field exposure to justify the significance of field statistics.

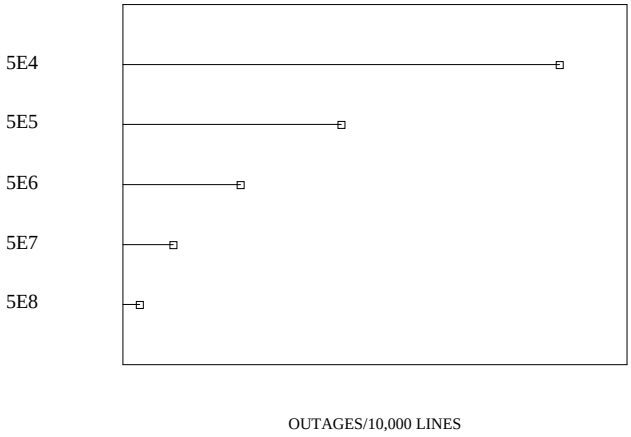

Figure 12.4 A decade of system dependability improvements

The remainder of this chapter details how the improvement was achieved and outlines directions to a new step function in improving large system dependability. Section 12.2 studies the fault removal process in achieving the quality objectives in the past decade. Section 12.3 examines fault recovery mechanisms in improving system dependability. Remaining shortcomings are also discussed. New promising avenues are indicated in Section 12.4. Finally, Section 12.5 provides some conclusions.

12.2 TEN YEARS OF SOFTWARE DEPENDABILITY IMPROVEMENTS THROUGH FAULT REMOVAL

Traditional software quality improvement processes rely heavily on test-based debugging which has required immense effort and resources. A better understanding of the defect removal process is essential to reduce software development cost and increase its quality [Lyu95].

12.2.1 Modeling of the Defect Removal Process

In most software projects, the fault detection rate follows the profiles in Figure 12.5, where curves a, b and c represent a *successful* project, a *failing* project and a *bankrupt* project, respectively [Abe79]. The bankrupt nature of curve c originates from the fact that the fault detection rate decreases very slowly, indicating that the number of residual faults does not substantially change. The failing project of curve b is characterized by a high rate of field failures. All projects in Figure 12.5 are managed in such a way that the number of residual faults first increases ("destruction" phase) and then decreases ("reconstruction" phases). In the context of such a management strategy, project success (curve a) is an expensive task.

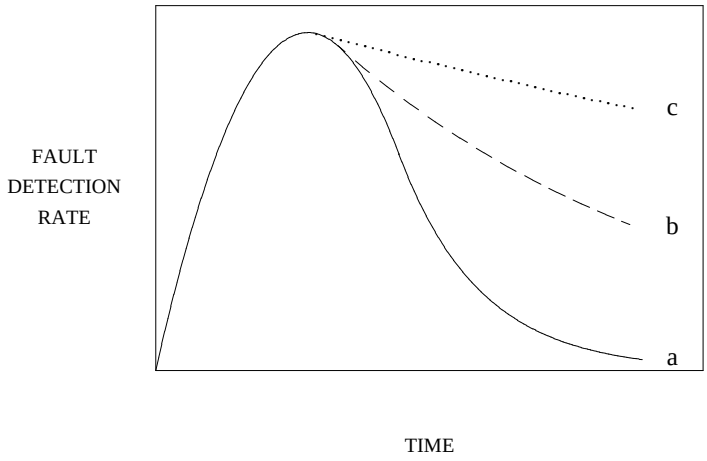

Figure 12.5 Software fault detection rates

Evidently, the ability to forecast the behaviors of the fault finding rate is very attractive for project managers. However, software reliability growth models have been used and abused in this endeavor, and their use has not always led to satisfactory results. Good studies of reliability models and modelling can be found in the technical literature [Goe85, Mus87]. The industry has often been tempted to use software reliability growth models to track software quality improvement during the development process (cumulative detected faults, fault detection rate). While they were not intended as such, they may prove of some use after they are better understood.

12.2.1.1 SOFTWARE RELIABILITY GROWTH MODELS EVALUATION

A fundamental observation was made by B. Littlewood which significantly puts in question the validity of these models [Lit89]. He compared the forecasts of 7 models and discovered that 4 of them were consistently pessimistic (curve a in Figure 12.6) while 3 of them were consistently optimistic (curve b). The true field behavior being approximately in the middle between the two classes, a pragmatic solution was proposed to interpolate between the two groups (curve c).

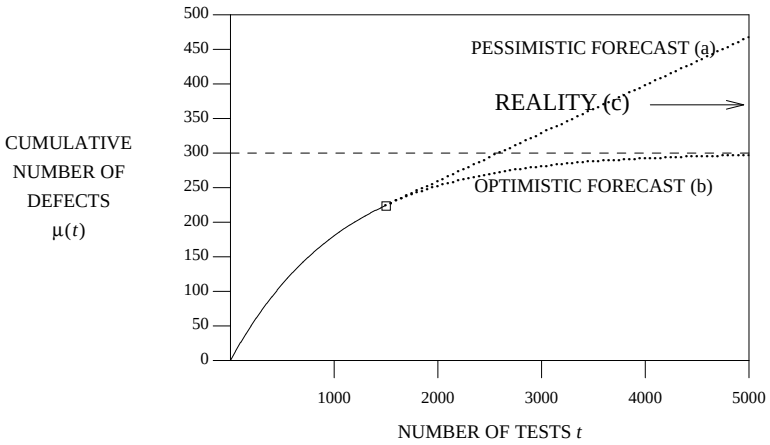

Figure 12.6 Two inadequate forecasts

While the interpolation is pragmatically attractive, it does not stand scientific rigor and may at best expose the limitations of the models.

In fact, the difficulty brought forth by Littlewood is not different from what practical field experience had already taught us about software reliability growth models. The optimistic models share their excessive reliance on preset fault density, namely these models assume a constant number of software defects known in advance. The pessimistic class is excessively driven by the derivative (fault finding rate) and will always overestimate the number of defects to be found.

The fault density driven models naturally tend to be more popular because the initial fault density is a parameter which can easily be derived from historical product data, and intuitively, it is very attractive to embed in a model readily available historical knowledge. A major drawback of this class of models is its optimism, namely the actual defect finding rate does not taper down when expected.

A possible practical instrument to model this effect is to introduce the *software revisiting rate*, β, which reflects the fact that for every defect fixed there will be a probability β to revisit the fix, because it which was either partial or incorrect. Curves a, b, and c in Figures 12.5 represent respectively a low, high and catastrophic effect of β. Practical measured values of the software revisiting rate range between 0.25 and 0.35. This seriously puts in question the assumption that, in most models, the revisiting rate is 0 [Goe85].

12.2.1.2 MODEL WITH SOFTWARE REVISIT RATE

As a result of the difficulties mentioned above and for benefits discussed below, a new model was developed [Lev87, Lev89a, Lev89b]. In addition to the initial fault density α and contrary to most models in the literature, the model presented here assumes a time-to-detection s, a non-zero defect repair time t, and a defect revisiting rate β, and is tightly coupled with the software production rate through the fault density and the defect detection time. The model with revisiting rate belongs to the "birth-death" family of models [Bai64, Kre83]. A good understanding of these parameters can provide the answers to the following questions:

1. How many defects are expected in the system at the time the system is delivered to the field? Is the system ready?
2. How many defects are expected in the future as a function of time after the system has been put in service? What is the expected failure rate as a function of time?
3. What is the required staffing profile needed to support the field deployment?

A — Defect Introduction, Detection and Removal during Software Development

During the development process, defects are introduced as a function of the size of the software being introduced or modified [Gaf84, Lip82]. The defects introduced at some point in time are discovered at later times and recorded as trouble reports. Critical defects and "ultra-visible" defects are discovered immediately, whereas latent defects remain in the system until they are exposed by the appropriate set of tests. The less critical and the less visible a defect is, the longer the defect will remain in the system undetected.

Without lowering the importance of achieving quality up front by better design practices [ATT86], it is important to consider the existence of software defects as a reality. As a matter of fact, the success of a development is largely determined by the programmers ability to eliminate defects in the speediest way [Abe79, Mon82]. Therefore, the software development process may be modeled as a defect removal process. Defect removal becomes the bottleneck of the design process. An important question becomes: when is system dependability high enough to release the system? [Dal88, Dal94, Lev89c, Lev90b, Lev91a, Mus89]

B — A Time Dependent Model

Calendar and execution time models [Goe85, Mus75] are common models for describing the dynamics of defect detection in large software projects.

Here, a calendar time model is developed. The model is based upon an incremental software introduction in the public domain. Every software increment is assumed to bring into the system a corresponding increment of defects. Tools have be developed to measure the incremental software deliveries (number of lines of code per incremental release).

The model developed here provides an analytical link between code introduction, fault detection and defect removal (Figure 12.7).

Given n lines of code, the number of defects appearing during time period i is described by

$$d_i = n\alpha e^{-s}\frac{s^i}{i!} \tag{12.1}$$

where α is the defect density per line of code. d_i is proportional to a Poisson distribution with mean time s. If we assume that the defect repair rate is described by a Poisson distribution with mean t, the removal rate of the $n\alpha$ defects introduced with n lines of code is described by the new Poisson distribution:

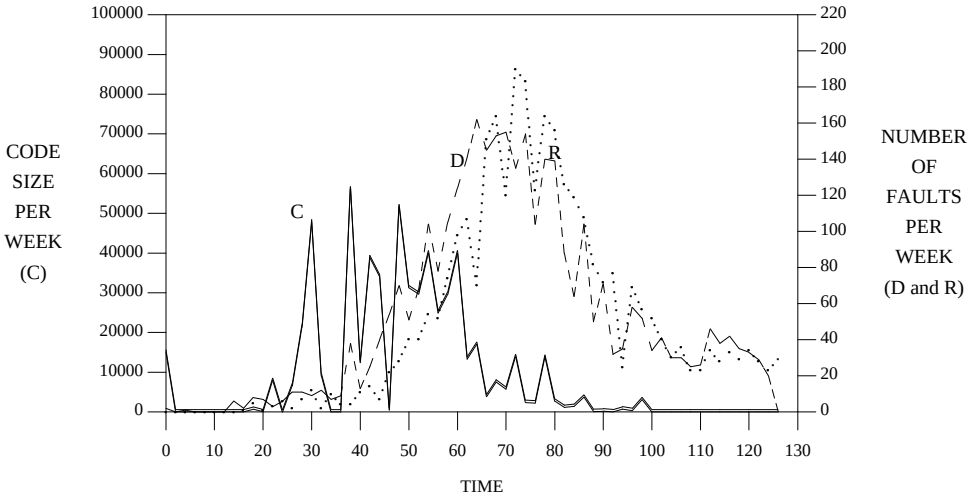

Figure 12.7 Code introduced (C), faults detected (D) and faults repaired (R)

$$r_i \;=\; n\alpha e^{-(s+t)}\,\frac{(s+t)^i}{i!} \tag{12.2}$$

which is the composition of two Poisson distributions, one with mean s and the other with mean t. In addition to the introduction of defects resulting from successive code submissions, each defect repair will require a revisiting of the software at the rate β, because of new defects reintroduction or incomplete repair, and the defect detection rate D_k for time period k becomes:

$$D_k \;=\; \alpha \sum_{i=0}^{k} n_{k-i}\gamma_i \tag{12.3}$$

for $0 \leq k \leq C$ and

$$D_k \;=\; \alpha \sum_{i=k-C}^{k} n_{k-i}\gamma_i \tag{12.4}$$

for $k > C$, where

$$\gamma_i \;=\; \sum_{j=0}^{\infty} \beta^j e^{-((j+1)s+jt)}\,\frac{((j+1)s+jt)^i}{i!} \tag{12.5}$$

Similarly, the defect removal is given by:

$$R_k \;=\; \alpha \sum_{i=0}^{k} n_{k-i}\phi_i \tag{12.6}$$

for $0 \leq k \leq C$ and

$$R_k \;=\; \alpha \sum_{i=k-C}^{k} n_{k-i}\phi_i \tag{12.7}$$

for $k > C$, where

$$\phi_i \;=\; \sum_{j=0}^{\infty} \beta^j e^{-(j+1)(s+t)} \frac{((j+1)(s+t))^i}{i!} \tag{12.8}$$

The median time m_D for D_k is given by the expression:

$$m_D \;=\; \frac{s+\beta t}{1-\beta} \;+\; \frac{\sum_{i=0}^{C} i n_i}{\sum_{i=0}^{C} n_i} \tag{12.9}$$

Similarly, it can be shown that:

$$m_R \;=\; \frac{s+t+\beta t}{1-\beta} \;+\; \frac{\sum_{i=0}^{C} i n_i}{\sum_{i=0}^{C} n_i} \tag{12.10}$$

Also, it appears that:

$$m_R \;-\; m_D \;=\; \frac{t}{1-\beta} \tag{12.11}$$

Another important quantity is the predicted number of defects remaining to be fixed after time period T:

$$ERD_T \;=\; \alpha \frac{1}{1-\beta} \sum_{i=0}^{C} n_i \;-\; \sum_{i=0}^{T} R_i \tag{12.12}$$

The estimated current defect number at time T , ECD_T, is an estimate of the level of defects currently in the system inclusive of the net open defect profile, NOD_T. The values of ECD_T make sense for $T \;\leq\; C$.

$$ECD_T \;=\; (1-\beta) ERD_T \tag{12.13}$$

The testing process effectiveness, TPE_T, is the ratio:

$$TPE_T \;=\; \frac{number\ of\ filtered\ defects\ found\ so\ far}{number\ of\ raw\ defects\ found\ so\ far} \tag{12.14}$$

This ratio is a measure of the time spent on fixing real defects versus the time spent eliminating "false" defects. The testing process quality at time T (TPQ_T) is a measure of the success of the defect identification process so far. It is defined as:

$$TPQ_T \;=\; \frac{number\ of\ defects\ found\ so\ far}{number\ of\ defects\ introduced\ so\ far} \tag{12.15}$$

The previous formula becomes:

$$TPQ_T \;=\; \frac{\sum_{i=0}^{T} D_i}{\alpha \sum_{i=0}^{C} n_i \;+\; \beta \sum_{i=0}^{T} R_i} \tag{12.16}$$

The analytical model is fitted with the actual project data in Figures 12.7, and the resulting estimates are summarized in Table 12.2.

Table 12.2 Key parameters and variables (with defect reintroduction)

Defect Detection Time Constant s	17.2 Weeks
Defect Repair Time Constant t	4.7 Weeks
Code Delivery	589810 Lines
Initial Error Density α	0.00387 Defects per Line
Defect Reintroduction Rate β	33 Percent
Deployment Time T	Week 100
Estimated Remaining Defects ERD_T	664 Defects
Estimated Current Defects ECD_T	445 Defects
Testing Process Quality TPQ_T	90 Percent
Testing Process Efficiency TPE_T	60 Percent

The results above indicate that the time constants of the process for reliability improvement are very large, and therefore, it must be a very expensive process.

12.2.2 Process Control Deficiencies

An important question is that of establishing a more cost effective process control. However, two important deficiencies of the current software dependability improvement process may stand in the way:

A — Excessive Reliance on Back-end Quality Monitoring

One of the traditional software development process deficiencies is a tendency to be driven by emergency which in turn results in a back-end loaded process, namely heavy resources are invested late. The main disadvantages of a back-end loaded process are:

1. Large staff results in major inefficiencies.
2. There is no room for additional course corrections.

The solution to this problem resides in the institution of on-going quality gates to bring attention on difficulties as they occur instead of postponing them to the end of the process. The quality gates are distributed all along the development process.

B — Inadequate Operational Profile

A correct operational profile is a necessary condition for the validity of test data as a measure of quality and reliability. The operational profile [Mus87] represents the set of environment conditions which will guarantee that the test program is executed under realistic field conditions. Although the concept of operational profile is an elegant one, it may prove to be unusually difficult and often impossible to implement. A study of the telephone central offices was done in the US [Sys90] to determine the profile of the end-customers (features used) in each telephone central office. It became obvious that it would be impossible to accurately reflect all realistic situations. Instead, 9 canonical operational profiles were constructed (large metropolitan center, hospital, university, large business customer, etc.). A lab model was constructed for each model, but it proved extremely expensive to continuously maintain because of the need to assure consistency between large collections of data. The models proved equally prohibitively expensive to exercise in a meaningful way because they required large sets of automated tests. In practice, the automated test sets were too small, and the metrics derived

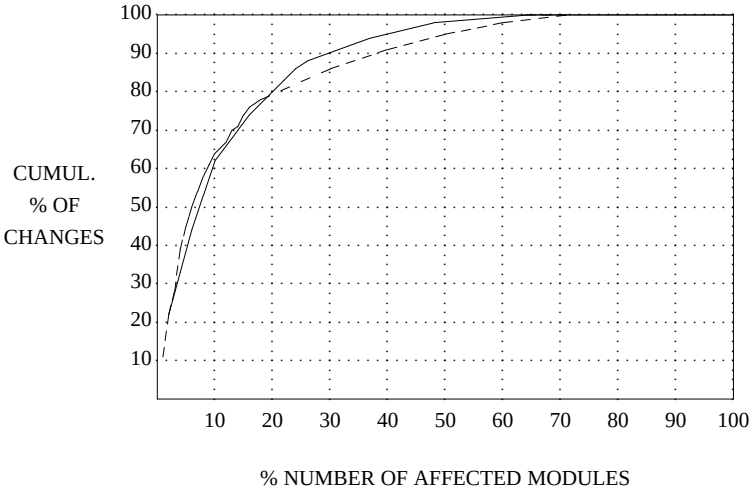

Figure 12.8 80 percent of the changes affect 20 percent of the modules

from these tests tended to converge much before the product was ready for field, namely the fault finding rate of the automated tests became very low while the fault finding rate through manual testing was still too high for field release. In other words, the test metrics obtained by using the operational profile were *optimistic*. The reason for this result is the difficulty to execute an ever changing set of automated tests to avoid ”tuning” the software for a limited experiment.

12.2.3 The Design Hole Conjecture

12.2.3.1 SOFTWARE ERROR CLUSTERING

Data collected under various circumstances demonstrate that software defects tend to concentrate in specific parts of the system. A typical distribution of software modifications against the number of software modules is given in Figure 12.8. Two typical subsystems in the 5ESS® system were selected (solid and dashed graphs). However, all the subsystems display a similar behavior, namely a large amount (80%) of the changes are concentrated in a small number (20%) of modules. These highly active modules during development will likely be the lingering cause of residual defects. This is in line with other studies [Gra92]. Although operational profiles could theoretically give more weights to active modules, deriving operational profiles for new additions to software is largely impractical.

There is a dual importance to the clustering of defects:

1. Eliminating clusters of defects is more economical than eliminating isolated defects.
2. The knowledge of the recent clustering information may be used as a guide to better position a test program in areas that are the likely sites of residual defects.

12.2.3.2 THE PRINCIPLE OF INCOMPLETENESS

Many quantitative theories of software reliability are based on defect enumeration, and they tend to consider bugs as discrete objects statistically distributed over the entire software space [Gaf84, Goe85, Lev87, Lev89a, Lev89b, Lip82]. In the classical prediction of residual defects, known bugs are subtracted from a preestimated quota (predicted total number of defects) regardless of the cause and nature of software defects. A different and alternative point of view, which is presented here, was developed by the author to alleviate difficulties encountered with traditional reliability models.

The mechanism that leads to software design errors is tightly related to the human factor, and must be analyzed from the point of view of human comprehension of complexity. During the software development process, corrections and improvements are performed, and new features are incrementally added. Each of these human interventions perturbs numerous system interactions. Since it is practically impossible for a single individual to comprehend the entire scope of possible interactions between modules that implement a feature, the programmers address a small fraction of the necessary software scope, and entire segments may be overlooked. At the beginning of the design process, the design is *incomplete*, and the debugging process is used to construct the missing areas. More obvious design segments and interactions are implemented earlier in the design process. Each large deficiency breaks down into many smaller ones. The design process stops when a compromise is reached between quality and cost. The software development process obeys two important rules:

The Impact of Complexity: *One large class of software bugs is caused by human inability to comprehend the design complexity within a finite time and cost.*

The Principle of Incompleteness: *Many Software bugs originate from design incompleteness. As the design matures, the missing design areas are progressively filled in, thus creating numerous smaller design deficiencies.*

The need for revisiting the same software for iterative repairs has been recognized in the industry as ”fix-on-fix” or defect reintroduction [Lev90a]. The classical defect theories assume that the subsequent repairs are done to correct errors introduced in earlier repairs. The design hole hypothesis can offer one plausible cause of iterative code modification: the iterative code repairs can also be linked to incomplete designs in addition to erroneous repairs. A few definitions are now in order.

Definition 1: *Design Space or Scope*
The design scope is the complete span of a design with its full intended functionality. The design scope corresponds to what is called *input space* in other contexts.

Definition 2: *Design Pass-Space*
The design pass-space is the actual span of the correct part of the design. The design pass-space will generally be smaller than the design space.

Definition 3: *Design Fail-Space*
The design fail-space is the complement of the design pass-space. It is the part of the design space that causes incorrect responses to legitimate inputs. It corresponds to the *fail space* in other contexts.

Definition 4: *Design Hole*
A design hole is a subset of the design fail-space which is composed of closely related defects.

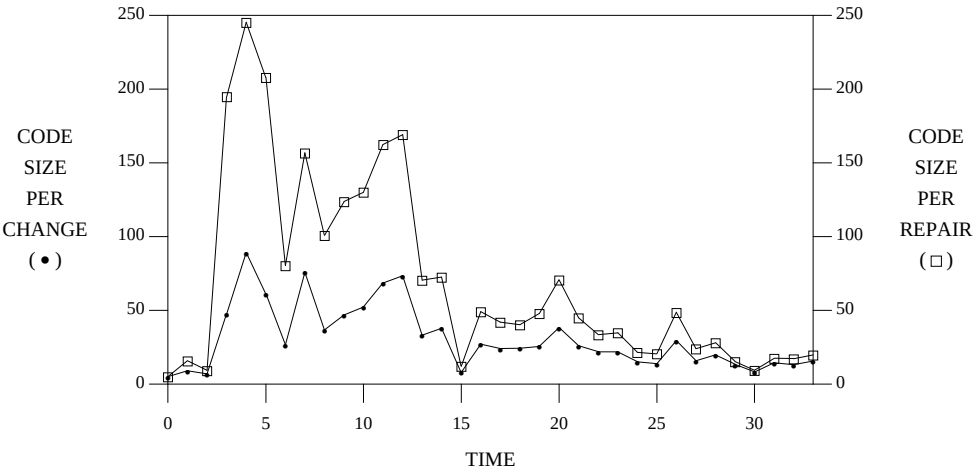

Figure 12.9 Code change size (in lines) per repair during development

12.2.3.3 DESIGN HOLE DISINTEGRATION

Statistical analysis of the way software development is done tends to confirm the theory of design holes and their disintegration. A repair generally contains multiple software changes due to the distributed nature of the system functionality. As can be seen in Figures 12.9, the size and the scope of changes progressively decreases as the design ”converges,” indicating a slow reduction of the design holes. Figure 12.9 shows that the number of changes per repair decreases with time from 4 to 1.2. This behavior is characteristic of all 5ESS® field releases.

12.2.4 The Manufacturization of the Testing Process

The size of large software systems often reaches millions of lines of source code that are developed by hundreds of programmers. By and large, the manufacturing of software does not adhere to robust disciplines, and recent advances in the software development methodology have not yet been widely accepted. Contrary to hardware manufacturing, software manufacturing is lacking the essential elements of statistical process control that have for decades been the attributes of other forms of manufacturing. Reduction of the variance is a key element in our ability to improve the quality of the testing process [Tag90].

Neither the theory of statistical manufacturing process control [Rya88] nor the statistical analysis of the software development [Goe85] are new. We show here a use of these techniques in the software development process [Lev91b].

12.2.4.1 THE ROLE OF TESTING: DEBUGGING OR PROCESS CONTROL

When testing is used strictly as a debugging mechanism, the execution of an exhaustive set of tests is required, and quality becomes dependent on mass inspection. Every possible situation needs to be explored, since testing will continue as long as there still exist bugs. *From the point of view of debugging, testing is aimed at proving that there are no more defects.* Projects

that adopt this testing philosophy are bound to release defective products to the customer or incur extraordinary costs in a never ending quest for full test coverage. In fact, it is impossible to implement a test program that spans all possible situations.

When used for process control, testing helps identify the most defective areas first. As soon as an important defective area is found, testing is suspended and a corrective action is undertaken. *In the context of process control, testing is aimed at proving that there still are remaining defects.* This approach makes a more cost effective use of testing.

In practice, it is impossible to completely separate the two roles of testing and exclusively operate from one point of view or from the other, but, in reality, the tendency is to exclusively use testing for debugging purposes. Therefore, the question at hand is: ”how does one refocus testing to serve as a process control mechanism?”.

12.2.4.2 WHEN TO STOP TESTING?

The question: ”when to stop testing?” can be asked in several contexts, and the answer is different depending on the situation.

Testing Purpose #1: *Assessing Software Quality by Fault Detection*
Question: *How long does one need to test for assessing software quality?*
Answer: *One can stop testing when enough tests have been executed to allow a determination of the software quality with high enough confidence.*
Testing Purpose #2: *Improving Software Quality by Fault Identification*
Question: *How long does one need to test for finding the software deficiencies?*
Answer: *One can stop testing when enough failure points have been generated to determine the likely source of the most important software deficiencies with high enough confidence.*
Testing Purpose #3: *Reaching Acceptable Software Quality*
Question: *How long does one need to test for reaching field quality?*
Answer: *One can stop testing when it can be determined with high enough confidence that the desired software quality is achieved.*

The remainder of this section focuses on developing a testing methodology that can serve as a process control mechanism, and help alleviate the difficulties addressed earlier. This testing methodology is used as a homing mechanism onto design holes, and it is combined with off-line debugging which is a cheaper alternative than testing defects one by one out of a system.

12.2.4.3 THE USE OF UNTAMPERED TEST METRICS FOR PROCESS CONTROL

The word ”tampering” is often used in an ethical sense. Modifying information with the intent of defrauding represents an unethical form of tampering. In the context of this paper, tampering will be used in a systemic sense. In other words, systems that induce employees to distort failure data by providing them with unconscious incentives to do so practice ”systemic tampering”.

A — Unrepaired and Repaired Test Pass Rate

Commonly, software test programs are preset and run over several months. In a debugging-oriented test philosophy, the failing tests are repaired, rerun, and the testing operation ceases when ”100% cumulative pass rate is reached”. Unfortunately, the backlog of unrepaired known defects is often used as the dominant measure of product quality. *Cumulative (re-*

paired) pass rate is computed as the sum of the initial *first pass (unrepaired)* pass rate and any correction due to repair of the faults that cause system failures during the initial execution. A 100% repaired pass rate can always be reached, no matter how low the initial pass rate was.

Our experience in the telecommunication industry has repeatedly shown that field quality does not correlate to the high level of system test cumulative pass rate (usually close to 100%) or, equivalently, to the small size of the backlog of unrepaired defects. This can be explained by the fact that system test programs represent at best a well-distributed small percentage of the universe of all reasonable system exercises. Obviously, improving the quality of a small sample will marginally affect the overall product quality.

In the hardware manufacturing process, mostly untampered failure data is used to control the process. The first pass pass rate (first pass yield) generated at a given test station is not recomputed to reflect the failure repairs. This fundamental ingredient allows the hardware manufacturing process managers to control the hardware production quality.

Giving to much importance to the cumulative pass rate is a typical example of systemic data tampering, since the project creates strong incentives for employees to speedily ”improve” failure data by overemphasizing the measurement of residual unrepaired known defects. As a matter of fact, hasty repairs may in the long run adversely affect the product quality. The size of the backlog of unrepaired defects reflects the *project ability to fix known problems*, whereas the first pass pass rate reflects the *product quality*.

B — Cumulative Test Failures and Unrepaired Test Fail Rate

Often, the reliability of a software system is characterized by the instantaneous rate of failures and the cumulative number of failures. The *cumulative number of test failures* $\mu(t)$ can be modelled by the function

$$\mu(t) \ = \ \nu_0(1 \ - \ e^{-\frac{\lambda_0}{\nu_0}t}) \tag{12.17}$$

where t is the testing time measured in testing hours [Kru89, Mus89] or in number of tests (Figure 12.10).

The derivative, $\lambda(t)$, of $\mu(t)$ is the *instantaneous rate of failure*

$$\lambda(t) \ = \ \lambda_0 e^{-\frac{\lambda_0}{\nu_0}t} \tag{12.18}$$

A special case of interest occurs when t is the number of tests executed. $\mu(t)$ becomes the *Cumulative Test Failures* and $\lambda(t)$ becomes the *Unrepaired Test Fail Rate*. Then, the first pass pass rate $p(t)$ is the complement of the derivative $\lambda(t)$, namely

$$p(t) \ = \ 1 - \lambda(t) \tag{12.19}$$

In other words, the derivative (slope) of $\mu(t)$ is a good indicator of product quality. A product will be acceptable if the derivative of $\mu(t)$ has a value below a given threshold. The model of Equation 12.17 is the simplest of a collection of published models. However, this will not affect the generality of our conclusions, since we are only interested in the slope of the curve regardless of its exact nature.

C — System Partitioning

Although many projects use uniform test metrics, practical experience has shown that software is not uniform in a given project. This observation yields a decomposition of software into various functional areas for the purpose of testing and evaluation. First, a hierarchical decomposition may be considered (Table 12.3).

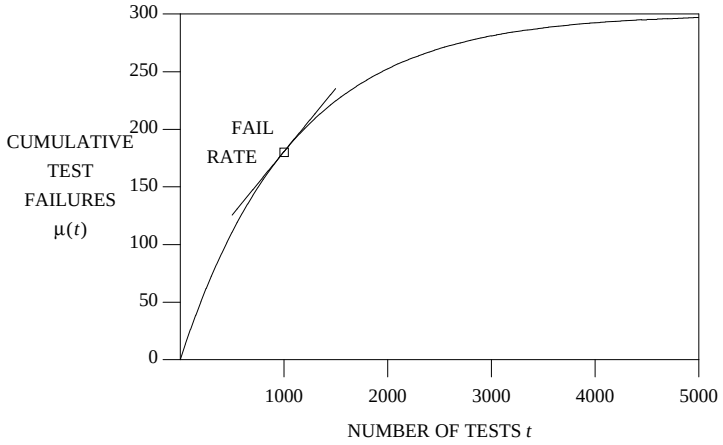

Figure 12.10 Cumulative test failures $\mu(t)$

Table 12.3 Functional and testing hierarchy

FUNCTIONAL HIERARCHY	TEST HIERARCHY
Functional Area	*Evaluation Segment*
Complex Functionality	*Test Suite*
Functional Component	*Test Component*

Testing levels correspond to functional levels. A *Test Component* is an exercise that spans the smallest element of functionality, the functional component. For instance in telecommunication, the action of forwarding a call to another line is a test component. Generally, a test is a succession of steps that correspond to the execution of several functional components. A *Test Suite* is an ordered set of test components. An *Evaluation Segment* is a collection of test suites. Conceptually, a test suite can be considered as a traversal of functional modules.

D — Number of Test Suites for Detection Purposes: an Adaptive Test Program Size

The validity of the test results as a process control element depends on the size of the test sample which is determined by the desired level of confidence. The theory of sampling [Rya88] and random testing [Yar88] are well-established. For a given desired level of confidence, sample sizes can be determined using the variance of the test results.

A sample size between a few tens and a few hundreds is enough to approximate the incoming quality of a measured entity [Dem82, Mas89]. This test sample should be applied to a relatively homogeneous software functional area (evaluation segment). An example of using test sampling results for quality evaluation is given in Figure 12.11. The actual first pass pass rate p_s is computed as the sample size grows. p_{st} is the desired test suite pass rate target, and p_{st}^+ and p_{st}^- are the upper and lower 95% confidence levels, respectively. For a confidence level of 95%, p_{st}^+ and p_{st}^- are approximated by the following formulas:

$$p_{st}^+ = p_{st} + \frac{1.96\sigma}{{n_s}^{1/2}} \tag{12.20}$$

and

$$\bar{p_{st}} = p_{st} - \frac{1.96\sigma}{{n_s}^{1/2}} \tag{12.21}$$

where n_s is the number of test suites and $L = \frac{1.96\sigma}{{n_s}^{1/2}}$ is half the *ambiguity* interval. The level of confidence is inversely related to $2L$ (or the size of the variance). The variance σ is estimated by

$$\sigma = [p_{st}(1 - p_{st})]^{1/2} \tag{12.22}$$

with a maximum value of 50%.

The upper and the lower confidence limits partition the graph of Figure 12.11 into three regions: the *acceptance region*, the *rejection region* and the *ambiguity region*. As soon as the number of tests drives the pass rate into the acceptance region or the rejection region, it is possible to stop testing with a definite conclusion about the product quality. A pass rate in the ambiguity region requires to continue testing until the pass rate reaches one of the two definite regions or until testing is no more economical. For a sample size of T or larger, the test results in Figure 12.11 probably indicate poor software quality.

It is important to notice that less tests are needed to assess both high and poor quality product than to assess marginal results (close to the target). This has a serious economic consequences towards the end of the development process: *more and more resources (tests) are needed to provide a reliable assessment of the product quality as quality increases unless the quality becomes very high* [Lev89c].

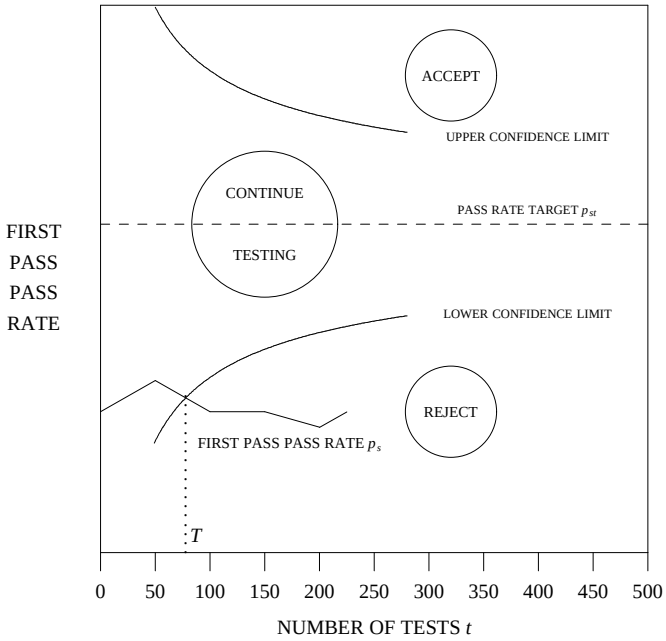

Figure 12.11 Sample size and confidence limits

The correctness of Equations 12.20 and 12.21 is predicated on the validity of the statistical distribution underlying the testing results (we have assumed a binomial distribution).

However, our experience confirms the validity of statistical sample data analysis, even if the statistical distribution underlying the data is not well characterized. Specifically, actual values of the variance of the first pass pass rates were found to be very small for sample sizes of a few tens of tests. Alternate methods for adaptive (sequential) sampling have been documented for statistical control of manufacturing [Gra88].

E — System Functional Decomposition to Aid Fault Identification

The statistical method explained above can be applied to any evaluation segment. The test suite pass rate produces a "GO/NO GO" decision for the evaluation segment (fault detection). In case of a "NO GO" decision, an additional mechanism is needed to narrow down the problem to a lower level root cause, namely the functional component (fault identification). During test suite execution, a count is kept for both component test execution and fail rate. Equations 12.20 and 21 can be extended to component pass rates as well, yielding the following results.

$$p_{ct}{}^{+} = p_{ct} + \frac{1.96\sigma}{n_c{}^{1/2}} \tag{12.23}$$

and

$$p_{ct}{}^{-} = p_{ct} - \frac{1.96\sigma}{n_c{}^{1/2}} \tag{12.24}$$

where p_{ct} and n_c are the component pass rate target and the number of component tests executed, respectively.

The fail rates for the various modules of a typical project are provided in Figure 12.12. The relative concentration of failures indicates the relative quality of the software segments represented by test components (functional modules). A functional module with a high fail rate is likely to be affected by a "megabug". Isolating modules with potential megabugs is a cost effective way of improving quality.

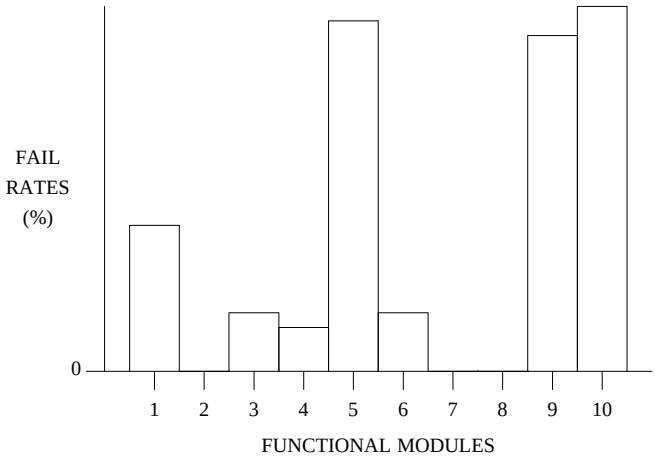

Figure 12.12 Failure data distribution against functional modules

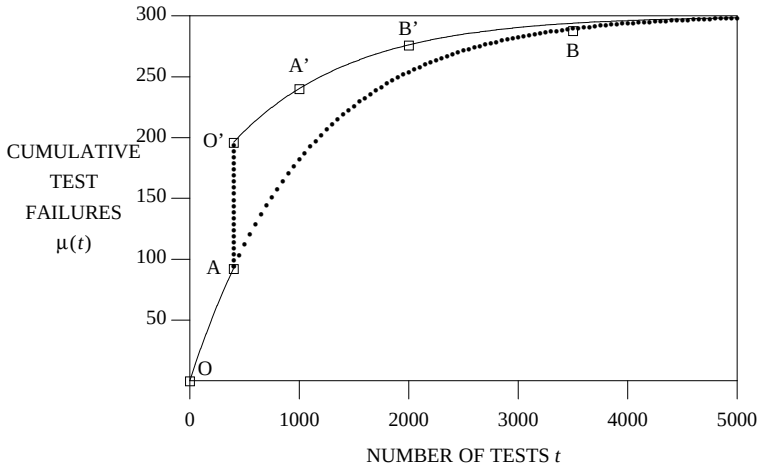

Figure 12.13 Cumulative test failures for a functional area

12.2.4.4 USE OF UNREPAIRED PASS RATE AS A PROCESS CONTROL METRIC

As discussed earlier in Section 12.2, error prevention methods are essential to lower the risk of field product quality to be too low. However, it is necessary to accept the reality of error correction and optimize its efficiency, since it is the last resort for protecting the customers interest. Also, appropriate analysis of the corrected errors can in the long run lead to error prevention.

A quality improvement methodology based on "megabug" identification and elimination is discussed next. Assume that a testing program has reached point A in the process of evaluating a given software functional area (Figure 12.13).

It was shown [Lev91a] that the instantaneous rate of failure $\lambda(t)$ (derivative of $\mu(t)$) is a good measure of quality. If the derivative of $\mu(t)$ at A is too high (the first pass pass rate is too low), then the software quality is too low. A possible solution consists of continued testing until the pass rate (derivative) is acceptable (point B on the dotted curve). This alternative requires the execution of a large number of new tests, $t_B - t_A$, and may be impractical. At this point, it is more appropriate to stop and examine a functional decomposition of the defect data. The results will guide the designer to reexamine the software design and repair large design holes. Redesigning defective areas results in jumping to point O' on a higher curve with a lower failure rate. The derivative at O' is again approximated by performing additional testing until point A'. This either yields an acceptable derivative or the distance to point B' with an acceptable derivative is smaller than the distance to an acceptable derivative on the original curve, namely

$$t'_B - t'_A < t_B - t_A \tag{12.25}$$

If the derivative is not acceptable but the point with an acceptable one is close enough, then it is appropriate to reach the point by an additional testing program. This procedure may have to be repeated if $t'_B - t'_A$ is too large. The choice between the various alternatives should be based on an economic decision.

12.2.4.5 MANUFACTURIZATION OF THE SYSTEM TEST PHASE

This section describes the result of an experiment that was recently conducted in a software project for a large distributed system (5E6 release of the 5ESS®system). During this experiment, a ”manufacturization” of the back end of the development process was implemented using most of the techniques described earlier. The software release included a substantial hardware architectural modification and spanned approximately one million of new and modified source code built on an existing base of several millions of source code. The data collected during the experiment [Sys90] provides the base for the analysis of the remainder of this section. Subsets of the data are provided in the sequence (Tables 12.4 to 12.7).

The early field failure rate for the recent release is 60% lower than the early field failure rate of previous releases and is 10% lower than the current field failure rate of the previous release after 15 months of field exposure. In addition, the frequent abortions of new software installations have practically disappeared.

A — Repetitions in a Manufacturing Process

In any manufacturing process, repetition is the key element that allows the application of statistical process control. Usually, the repetition happens along the time dimension, namely a new copy of the product is produced at regular intervals. However, space repetition is also used for process control. For instance, n production stations working in parallel constitute a space repetition, and the failure data produced by all the stations can be compared in order to identify and correct abnormal station behavior. In the case of time or space repetition, the repetition must be large enough to justify the use of statistics. *Conversely, the existence of space or time repetitions in a process allows its modelling as a ”manufacturing” process*.

In software, there are several possible dimensions of meaningful repetition:

1. The various modules in a given software delivery represent a space repetition.
2. The implementation of all evaluation segments in a software delivery represents another type of space repetition.
3. The successive execution of test suites within a given evaluation segment represents a time repetition that maps directly to the second space repetition mentioned above.

The various forms of repetition happening at every phase of the software manufacturing process provide the base for the use of statistical methods to control this process and the transitions between its phases. The use of time and space repetitions is illustrated next.

B — The Manufacturization of System Test: a Case Study

The process used is described in Figure 12.14. Partitioning the features into 54 groups of features (evaluation segments ES_1 to ES_n) constituted space repetition. Grouping features into feature groups is essential for providing comparisons between groups of features and between features in the same group. Four successive evaluation intervals realized a time repetition. Time repetition is essential for measuring the rate at which the fail rate decreases and the software approaches field quality.

The implementation of the system test phase using the four subphases described in Figure 12.14 constitutes a manufacturing process within the system test phase.

C — The Economics of Quality Assessment

As discussed earlier, less tests are needed to assess the quality of software when the quality is far below or far above the acceptability threshold than close to it. This can be seen in Figure 12.11.

If the pass rate remains in the ambiguity region, more tests are executed, and several alternatives are available afterwards:

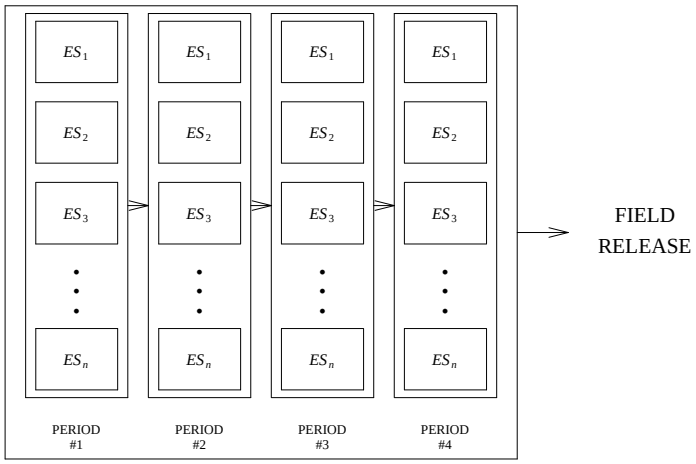

Figure 12.14 Manufacturization of system test

1. If the results are definitely in the acceptance region, stop testing
2. If the results are definitely in the rejection region, repair the software
3. If the results are still ambiguous, two possibilities are available:

   (a) Continue the test/repair cycle until the ambiguity is lifted
   (b) Stop testing and release the product anyway

Under alternative 3, the choice between pursuing and stopping testing depends on a cost/benefit tradeoff and a risk analysis. The pass rate is likely to remain around the ambiguity region until many tests are executed and the software repaired. At this point, the risk of releasing an imperfect product must be weighed against lost market opportunities before a decision is made. The difficulty originates from the fact that, in the ambiguity region, a few failure points are available to accelerate the product quality into the acceptance region.

D — System Evaluation using Evaluation Segments

It is important to note that the test suites in the four evaluation intervals must be distinctly different in order to provide an untampered view of the product quality. Along the four evaluation intervals, the testing results showed substantial improvements due to the identification and repair of design holes. The resulting first pass pass rates (FPPR) and failures per test (FPT) are given in Table 12.4.

Table 12.4 Quality improvements during the software manufacturing process

Period	#1	#2	#3	#4
FPPR	75%	80%	84%	90%
FPT	0.115	0.108	0.072	0.080

A closer look is now given to evaluation segments. As mentioned earlier, a test suite is composed of test components. The components that are critical to an evaluation segment are

key to the evaluation segment. The non-critical ones may provide additional information for the evaluation of other evaluation segments for which they are critical. First pass pass rates (FPPR) for Period #1 are given in Table 12.5. The variance of the FPPR is much smaller for the component test pass rates than for the test suite pass rates.

Table 12.5 Test suite and test component pass rates

Feature Subset	Test Suite FPPR	Test. Comp. FPPR	Crit. Comp. FPPR
ES_1	86%	98%	94%
ES_2	96%	99%	99%
ES_3	67%	90%	89%
ES_4	75%	94%	97%
ES_5	72%	95%	94%
ES_6	92%	97%	98%
ES_7	90%	96%	96%
ES_8	71%	95%	90%
ES_9	97%	99%	98%
ES_{10}	86%	96%	98%
ES_{11}	76%	97%	87%
ES_{12}	91%	96%	88%
ES_{13}	67%	87%	89%
ES_{14}	92%	97%	97%

A further analysis of the FPPR of the critical components of an evaluation segment will help identify which functional module is most likely to be the site of a ”megabug”. In Table 12.5, the test suite FPPR for ES_{11} indicates a quality problem. The FPPR distribution for four critical components of feature set ES_{11} is given in Table 12.6. Critical components c_{i1} and c_{i2} definitely point at the cause of the quality problem. Note that critical component c_{i4} has a low FPPR. However, there were too few tests, and more tests were required for a good fault identification.

Table 12.6 Critical component FPPR for ES_{11}

Component	c_{i1}	c_{i2}	c_{i3}	c_{i4}
Tests	55	44	71	18
FPPR	87%	80%	100%	61%

The example of Table 12.6 is consistent with the observation [Dem82, Lev91a] that only a few tens of samples are sufficient to produce a good estimate of the product quality.

12.2.5 Are Testing Metrics Really Untamperable?

12.2.5.1 VARIANCE DUE TO HUMAN FACTOR

An experiment was conducted over a four-week period among 51 testers to assess the importance of human factor variance. Each tester was responsible for a different functionality, but

all were told that the experiment would focus on individual fault finding rate. As expected fault finding rates started diverging significantly (Table 12.7), and a closer examination of the techniques used by individuals show an important pattern: all the individuals with the highest fault discovery rates had found a way to home onto software vulnerabilities by testing areas where the code had been recently significantly modified because of new features or significant recent deficiencies. By using a combination of ”street wisdom ” and actual software knowledge the most productive testers had been able to home in on design vulnerabilities (”design holes”). Operationally, the testers used a combination of black box (functional) testing and white box (code structure) testing.

Table 12.7 Testing results for a four-week period

TESTER IDENTIFICATION	PROBLEMS OPENED	REAL FAULTS
1	55	48
2	50	46
3	49	45
4	27	25
5	26	24
6	21	21
7	20	19
.	.	.
.	.	.
37	1	1
.	.	.
44	1	0
45	0	0
.	.	
51	0	0

This experiment indicates the plausibility of the ”design hole” hypothesis, namely an area of the software where a concentration of defects exists. In the neighborhood of a design hole, the probability of failing a test is the highest. In different terms, the test program is approaching an area of *software instability,* which acts like a *pole* in linear systems. Further research is required to better define and locate software instability. Testers 1-7 seem to have empirically homed on to design holes.

12.2.5.2 A NEW CHALLENGE FOR AN OLD TECHNOLOGY: HARNESS THE VARIANCE

It would be attractive to develop a theory of software stability (and instability) similar to the theory of linear systems stability. However, such a theory is not at hand. In the mean time, it is possible to develop testing instrumentation which enables ”homing” on design holes [Cla92]. This instrumentation is a tool which emulates the experiment described in the previous section, namely directs the testing towards areas which detect more software errors by measuring the changes in fault finding rate: the larger the rate (faults/tests), the closer testing is to a design hole or instability. The purpose of testing then becomes that of maximizing the fault finding rate.

12.3 RECOVERY: A SUCCESSFUL BUT EXPENSIVE STRATEGY

The existence of recovery software is predicated on the assumption that it is impossible to deliver high quality hardware and software. It is the need for a system to operate correctly in the presence of these hardware or software faults that led to the development of large recovery software. In the 5ESS® switch, software development costs are an order of magnitude larger than hardware development costs, and 38% of this software deals with recovery. This represents a significant overhead cost directly related to the imperfections of hardware and software.

5ESS® recovery software has traditionally been partitioned into *Hardware Failure Recovery Software*, or in short *Fault Recovery Software*, and *Software Fault Recovery Software*, or *Software Integrity*. Of course, the boundary between the two is not totally unambiguous because some hardware failure may cause software corruptions without being clearly identifiable as hardware failures. For all AT&T switching products, the two segments of recovery software are regulated by recovery strategies and escalation levels [Cle86].

The recovery from hardware failures requires fault detection implements which are capable of rapidly identifying the presence of these failures and performing some degree of correction. At detection time, error data is automatically collected and stored so that it can be later analyzed. After error detection, a multi-level recovery strategy is used with an escalation policy, ultimately leading to resource swapping and system reconfiguration decisions based on alternate resource availability. In the case of the 5ESS® switch, several forms of redundancy are used in different parts of the system depending on the criticality of the function of each one of these parts (duplex, $m + k$,...). At the end of reconfiguration and after the system resumes operation, automatic troubleshooting is initiated in the suspected failing unit.

Two basic strategies are utilized for software fault recovery: dynamic error checking for very critical software, and data check for the remainder of the software. The data checks assume that a software error will ultimately manifest itself as a data error or inconsistency. Here too, an escalation strategy has been commonly used.

12.3.1 External Recovery Software Versus Embedded Hardware Recovery

Traditionally, hardware fault recovery has partially been embedded in the hardware while its software component was external to the application. The software fault recovery has essentially been external. Embedding recovery in the hardware has resulted in extremely high cost of hardware design and manufacturing goods. This is aggravated by the high proprietary nature of the hardware which prevents an economy of numbers and also affects software portability. The external nature of the recovery software makes it difficult to tune to specific applications and makes it vulnerable to inadequacies.

12.3.2 The Nature and Implications of Software Errors

As reported earlier [Cle86], several observations about software development are essential. Two software classes can be identified:

1. *High usage software*
   This is software which deals with delivering to the customer essential system functions which will be triggered by end customer requests. These include basic call processing, billing, system provisioning, customer data and office administration, etc.

2. *Low usage software*
   This category includes software which is run infrequently and software triggered by unexpected events. Recovery software belongs to that category. Some parts of call processing may belong to low usage software depending on the selling patterns and service cost policies of the service providers.

High usage software is easier to test and will ultimately receive the highest exposure. Therefore, its quality improvement curve will be the steepest. Conversely, recovery software will be the most difficult to "train." Testing it thoroughly in laboratory is impossible, and its field exposure will be the lowest, making its reliability improvement curve the flattest.

Recovery software can be triggered by errors originating from hardware or software or can be run on a routine basis. For instance, defensive diagnostic software can be routinely run in sensitive hardware areas on a low priority basis when processing cycles are available. The same can be done for routine audits which examine the consistency of software and data.

Software failures may be primary or secondary. A software *primary* failure is a failure that originated from a single error in a segment of code. A software *secondary* failure is a failure that originated from an erroneous reaction of a segment of software to another imperfection in software or in hardware.

Example 1: A software bug in call processing causes a call to be interrupted. The system routines that are in charge of cleaning up the data structures associated with interrupted calls does its job correctly. The software bug is primary.

Example 2: As a result of a hardware failure, a segment of recovery software takes the wrong hardware unit out of service. This software error is a secondary failure to the original hardware failure.

Example 3: Recovery software exhibits a secondary failure as a result of a primary call routing error.

Example 4: As a result of a routine audit of system data, recovery software erroneously tears down all call processing in a community of 256 customers. This failure is primary in recovery software. Although such cases are possible, most of the field failures in recovery software are secondary failures.

As the product improves in the field, primary failures in hardware or in software outside the recovery software will become rarer and rarer, thus decreasing the exposure of the recovery software and slowing down its debugging. In conclusion, the recovery software is likely to remain with the highest fault density, because its improvement depends on field exposure, as for any other software.

12.3.3 Software Which Does Not Fail Does Not Exist

Table 12.8 summarizes the field fault densities (x) and the fault densities during development (y) for a given product release. This release was composed of a significant hardware configuration change and a large increment of software (850,000 lines of code). The results were measured over 4 years.

The development and field fault density [Lip82] are obtained by dividing the number of development and field errors by the size of the code. x/y is the *training ratio.*

The data in Table 12.8 demonstrate the high "training" experienced by the high usage software (upper segment of Table 12.8). It is therefore likely that the high usage software has significantly improved as a result of this "training." Conversely, the low usage software (lower segment of Table 12.8) receives poor field training and displays a significantly lower train-

Table 12.8 Development and field fault density comparison

Software Increment Size	Development Fault Density (y)	Field Fault Density (x)	Training Ratio x/y	Software Type
42069	0.0285	0.0072	0.2543	Preventive Maintenance
5422	0.0673	0.0210	0.3123	Billing
9313	0.0793	0.0277	0.3491	Field Update Software
14467	0.0265	0.0072	0.2741	System Growth
165042	0.1016	0.0053	0.0534	Hardware Fault Recovery
16504	0.0841	0.0020	0.0237	Software Fault Recovery
38737	0.1494	0.0058	0.0393	Hardware and Software System Integrity

ing ratio. These measures clearly indicate serious problems with operational profile during software development.

12.3.4 Software Exposure and Software Improvement

The improvements of hardware and software (Figures 1 and 2) due to field exposure cause a rapid irrelevance of large amounts of recovery software. After a short interval in the field, the improvement of recovery software becomes very slow, postponing its maturation.

Most of the impact of fault recovery software occurs during system development, and helps maintain reasonable system cycling when the system has not yet reached field grade. In a laboratory experiment, system integrity software was turned off at various points during the development cycle to measure a global software quality metric. At the worse point in development (highest software failure rate in Figure 2), the system could not stay up beyond ten minutes without system recovery software. A few days before release the system remained up for one week before the experiment was stopped. In a sense, this indicates that a large part of the role of recovery software is aimed at maintaining a satisfactory design environment by allowing the system to run in the lab in spite of a relatively high number of hardware failures and software errors. It also allows to deliver to the field a product that is still imperfect. But, over a short period of time, the hardware and software reliability improvements make the recovery software somewhat irrelevant. However, when invoked, recovery software may exhibit secondary failures. Therefore, excessive reliance in the field on imperfect recovery software may prove to be more detrimental than it is beneficial since it is based on partially immature software, as exhibited in recent noted field incidents.

Given the decreasing number of field failure triggers, the time to significantly improve this software may go beyond the product useful life. Certainly, because of its sheer size (38% of the total software), large recovery software represents a costly strategy for achieving system dependability. On the other hand, since the cost of injecting failures in the laboratory prior to field release proves equally prohibitive, one cannot realistically avoid relying on field training of recovery software.

There also is a hidden cost of using ”untrained” recovery software. Non-catastrophic, unattended errors will tend to absorb excessive hardware resources further increasing the cost to performance ratio of the hardware.

In conclusion, recovery software may end up serving as a placebo, in the best case, and being detrimental, in the worst case.

12.4 A MODERN STRATEGY FOR DESIGNING DEPENDABLE SYSTEMS

12.4.1 Low Cost Upfront Hardware Dependability

Some of the techniques explained below are consistent with Bernstein and Yuhas [Ber93]. An essential step of a better strategy for system dependability is to increase up front dependability of its components.

1. Increased Reliance on Off-the-shelf Components
   Excessive utilization of specialized designs causes excessive failure rates and excessive reliance on recovery software. Conversely, using commercial components will allow the system designers to "ride" on commercial quality and performance curves, achieving earlier hardware maturation. This in turn will allow lowering the investment in recovery software. The use of commercial hardware components brings with it the potential of increasing the usage of commercial processors and their software, furthering the dependability of the system components.
2. Decrease Reliance on Duplex Configurations
   With an increase in commercial computers dependability, it may be feasible to rely less on duplex recovery mode in favor of $m + k$ techniques which may be more economical.
3. System and Functionality Distribution
   Distribution of functionality allows to reduce system complexity and therefore decreases the cost of achieving reliability targets. This technique enhances the usability of off-the-shelf system components, and it also allows to better choose hardware quality (and cost) in a range of possibilities and limit the use of proprietary hardware where absolutely necessary:

   (a) Fault tolerant hardware for high performance application segments.
   (b) High availability commercial hardware for mid-range applications.
   (c) Regular commercial hardware for the remainder.

   Interestingly enough, this trend may already be emerging in telecommunication and will affect its economy in the same way computer distribution to the end-user's desk has affected computing.
4. Use of Client-server Model
   This model strengthens our ability to implement a choice of hardware attributes and makes it a better fit for a simplified recovery software because distribution creates smaller software segments with more robust boundaries between them, yielding a better software architecture [Bor93].

12.4.2 Low Cost Upfront Software Dependability

In the past, significant effort has been invested in formally constructing software [Bac78, Pry77]. However, these techniques have not yielded a lowering of software development costs

and time-to-market, because they have transferred the complexity from software programming to software specifications.

Some progress has been achieved in the area of empirical software robustness methods [Bro95]. This work provided a prototype software development environment using object orientation which supports the concept of *Software Physical Design*, namely the boundaries of software components are given a certain amount of rigidity (physical design) similar to hardware components. This concept is implemented by requiring mapping of software components (objects) on processors during execution and communication exclusively performed by messaging. It is important to mention that robust recovery strategies can be implemented in a similar way by taking advantage of the partitioning provided by the software "physical design" [Bro95].

The resulting productivity increased by an order of magnitude, but the method fell short of a complete hardware analogy by not focusing on the software *Logic Design*, which can provide another dimension of software reuse. Indeed, as long as software technologies will remain *generic* in nature, software reuse will not be feasible on a large scale.

12.4.2.1 SOFTWARE ASSEMBLY SPEED AND CUSTOMIZATION

Without minimizing the need to improve current software production processes, it must be noted that today software production is largely a manual process. This is why the necessary additional breakthroughs can only be achieved by technological solutions which can in turn lead to the industrialization of the software production process. The essential ingredients to achieve upfront software dependability are:

1. Reusability of software components
2. Customization by software layering
3. Ease of software assembly and portability by using platforms

12.4.2.2 DOMAIN ANALYSIS AND SOFTWARE REUSE

It is interesting to analyze software technologies along two dimensions, expansion factor and degree of specialization. The expansion factor is a measure of the amount of object code generated by the invocation of a language high level construct. For instance, C++ may have an expansion factor 3 times larger than C. Larry Bernstein observed (private communication) that the expansion factor grows by an order of magnitude every 20 years, yielding commensurate productivity improvements. While recovery software principles have remained practically constant over the years, improvements in software technologies have continued to happen. Fundamentally, software engineers can now write software with a large "expansion factor" by using better languages and technologies. This improvement is based on an increase of the elementary language constructs which become larger reusable units. These techniques can improve the initial quality of software deliveries making the reliance on recovery software less necessary. Continued emphasis on initial software quality may prove to be a more economical route.

The second dimension, specialization, represents the degree to which a software technology is specially tailored for a given application. For instance, Finite State Machine (FSM) definition languages have expansion factors similar to C but a higher degree of specialization.

Figure 12.15 is a representative map of a few software technologies. The technologies at

the bottom of the map are the most popular ones, and by and large, the upper region of the map is less popular. This probably illustrates one of the major differences between software applications and other engineering applications. Indeed, specialized technologies have been critical in bringing many application domains to the industrial age. For instance, manufacturing could not have become effective without specialized technologies. In a similar fashion, it is expected that specialization of software technologies will bring software applications to the industrial age, namely will provide high velocity and low cost. The *Domain of Opportunities* is the area of the map of Figure 12.15 which represents technologies with high expansion factor and high specialization. Although this area of the map is not heavily populated, it is expected that this area will become denser as software becomes less of an art and more of a technology.

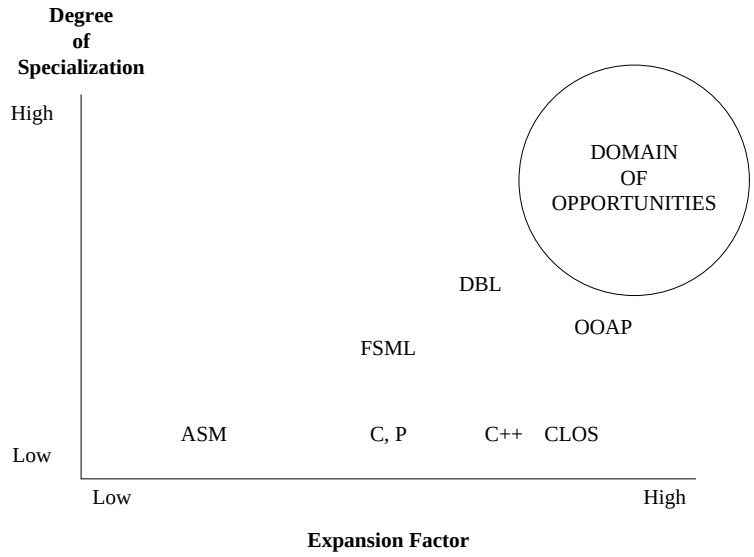

Figure 12.15 Software expansion factor and software specialization

The acronyms used in Figure 12.15 are given in Table 12.9.

Table 12.9 Sample of software technologies

ASM	Assembler
C	C Programming Language
P	Pascal Programming Language
C++	C++ Programming Language
CLOS	Common Lisp
FSML	Finite State Machine Programming Languages
DBL	Data Base Applications Languages
OOAP	Object Oriented Applications

As in any other industrial domain, specialization can occur only after thorough *Domain Analysis.* Much of the on-going work is predicated on the assumption that each application

domain ought to yield its own flavor of specialization. Applications have ranged from telecommunication applications [Gac94, Kol92, Lev95] to general computing applications [Huy93].

12.4.2.3 RAPID CUSTOMIZATION AND SOFTWARE LAYERING

Our approach is to provide various degrees of software customization by creating three software layers defined in Table 12.10. The programming in each layer uses programming elements from the adjacent layer below and produces products which are used as elements in the adjacent layer above. Primitive capabilities are used to produce building blocks, and building blocks are used to produce services.

While the service layer allows service assembly and customization, additional customization can be achieved by providing programmability of the building blocks using lower level specialized primitive capabilities. This is expected to provide a simplification of the existing programming paradigms (C, C++,...) by harnessing specialization. As a result of higher specialization of all the layers, our approach moves our work to the domain of opportunities on the map of Figure 12.15. As a result of developing specialized software technologies, an advantageous relationship ought to appear between programming ease (velocity) and service revenue opportunity (Figure 12.16). The upper layer ought to be the easiest to program and produce the largest revenue. Conversely, the lower layer should be the stablest and the least rewarding to modify.

Table 12.10 Software assembly programming layers

PROGRAMMING LAYER	ELEMENTS USED	PRODUCTS LOCATION
SERVICE (GUI*)	BUILDING BLOCKS	SERVICE SOFTWARE
INTERMEDIATE (AOL**)	PRIMITIVE CAPABILITIES	BUILDING BLOCKS
PLATFORM	GENERIC SOFTWARE	PRIMITIVE CAPABILITIES

*GUI: Graphical User Interface
*AOL: Application Oriented Language

12.4.2.4 NETWORK EXECUTION PLATFORMS AND PRIMITIVE CAPABILITIES

The lower software layer provides primitive capability access for the software layer above it. In a sense, it acts as a specialized *Operating System* whose role is to interact both with the layer above it and with various network resources. The lower layer acts as a platform and its role is to facilitate the programming of the building blocks in the intermediate layer above it.

The Network Execution Platform is composed of a number of *Genies*, each one capable of providing support to the application software above it through a collection of primitive capabilities, and of communicating with the network through a hardware platform. Each genie is specialized to a specific domain of support. Examples are given next.

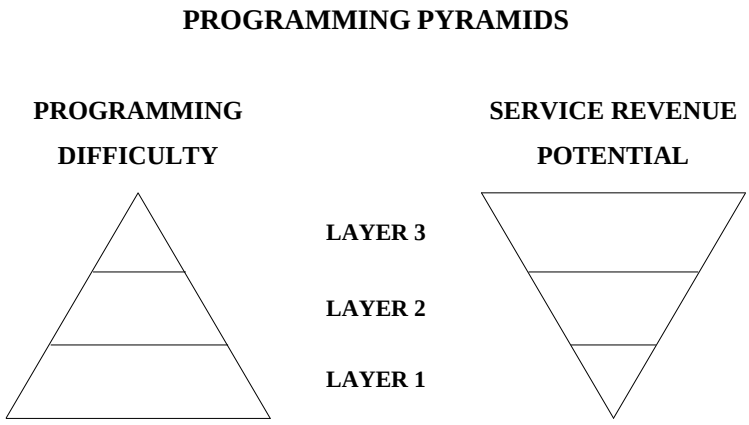

Figure 12.16 Programming difficulty and service revenue potential in software layers

1. Signaling Genie: takes care of tracking and interpreting telecommunication signaling messages.
2. Network Functions Genie: invokes network resources and functions (for instance, executes I/O commands such as ASCII translation to voice or voice collection and storage).
3. Service Data Genie: accesses service description data bases.
4. System Integrity Genie: takes care of error recovery using hardware independent Recovery Software.

There are two merits of defining a Network Execution Platform: a) the hardware can be upgraded to track industry performance and cost curves, and b) different service domains can use the same platform as long as the necessary primitive capabilities are provided by the platform (Figure 12.17). The network execution platform is a client which can invoke resources, R_1,..., R_n, distributed on network servers.

12.4.2.5 INTERMEDIATE PROGRAMMING: CONSTRUCTION OF REUSABLE BUILDING BLOCK

In general, a robust service software assembly environment ought to provide simple technologies to program building blocks based on the availability of powerful platform functionalities as described in Section 12.4.2.4. The benefit of a simple programming paradigm over existing paradigms (i.e. finite state machines) is velocity of new building block construction. The power of the primitive capabilities provided by the network execution platform is the key to this simplicity.

In a typical telecommunication context, reusable robust software building blocks can be designed and reused to assemble more complex software which implements complete services. Typical building blocks represent often used capabilities such as Voice Message Storing, Synthetic Voice Prompting, Dial Tone Message Collection, Call Transfer, etc. The existence of these executable software building blocks allows to assemble and execute more complex functionality.

12.4.2.6 SERVICE SOFTWARE ASSEMBLY ENVIRONMENT AND NETWORK EXECUTION PLATFORM

Figure 12.17 represents a simplified system architecture capable of supporting the approach described in Section 12.4.

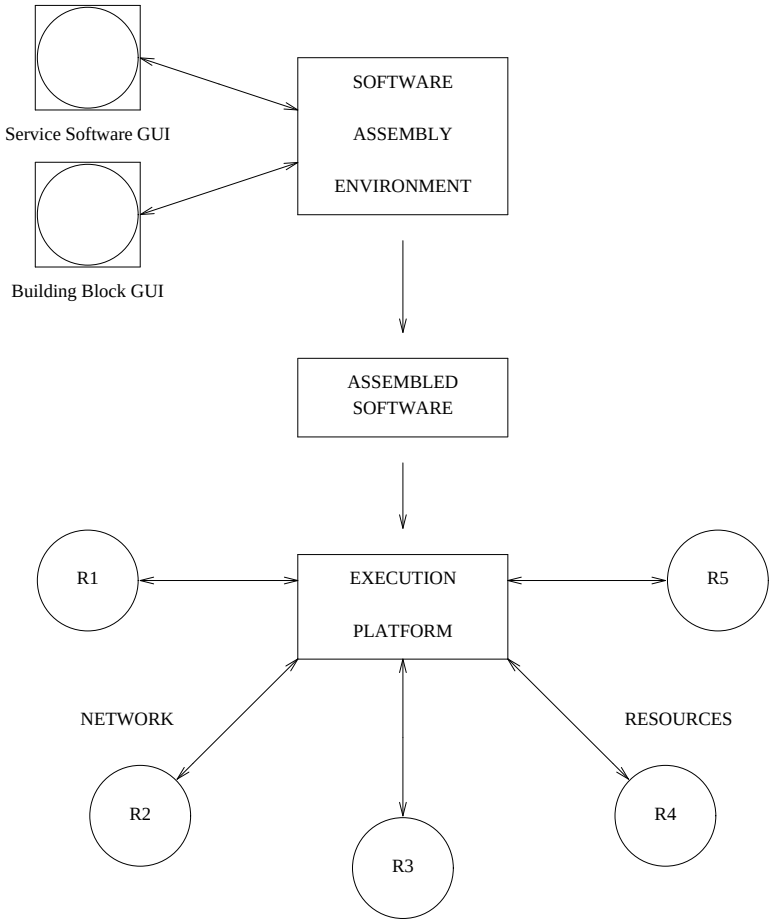

Figure 12.17 System architecture

The service Software Assembly Environment (SAE) is a two-layered programming environment including the service layer and the building block layer, each layer with its programming paradigm, aided by Graphical User Interfaces (GUI). The software generated in the SAE is shipped to the Network Execution Platform after having been adequately verified. The service software is executed by the service manager which can access the network resources through the genies in the network execution platform.

In addition, the language constructs drive both a *simulation* of the service under assembly and the *production* of the software to be executed on the network execution platform.

12.4.2.7 MORE RELIANCE ON RECOVERY SOFTWARE METHODOLOGIES

In rethinking the recovery strategy, it is important to distribute it when appropriate and move it from hardware to software as much as possible. Coupled with an appropriate choice of "fault-tolerance quality," this allows the appropriate hardware cost and performance overhead choices.

As discussed in Chapter 10, Huang and Kintala [Hua93] recognize 4 levels of recovery:

- *Level 1*: detection and restart; data may still be inconsistent.
- *Level 2*: Level 1 plus periodic check of dynamic data and recovery of internal states.
- *Level 3*: Level 2 plus check and recovery of static data.
- *Level 4*: continuous operation by duplication (hot spares, etc.).

These authors experimented with fault tolerant software that can address levels 1,2 and 3, and recover from faults undetected by hardware. The method incurred a performance overhead of 0.1% to 14%. The strategy is composed of the following three components:

1. *A watchdog daemon process (watchd)* which watches the cycle of local application process, recovers to the last checkpoint or to a standard state, can watch other watchdogs and reconfigure the network, and can watch itself to a certain extent.
2. *A library of C-functions for fault tolerance (libft)* which can be used in application programming to specify checkpoints and establish the recovery strategy. They allow tuning of the performance/resilience ratio of the recovery.
3. *A multi-dimensional file system (nDFS)* which allows the replication of critical data and is built on top of UNIX, which lowers design costs.

As opposed to hardware recovery, these techniques allow transferring the design effort from designing a complex mechanism to that of designing a strategy that is easily implementable by "off-the-shelf" software components (libraries). Since these techniques lend themselves to some degree of "mechanization" in implementing recovery software, they can help tune the strategy with respect to the frequency and amount of dynamic data (as well as static data) to be recovered. This can also facilitate software portability, making hardware upgrades easier.

A more detailed description is provided in Chapter 10.

12.4.2.8 RECOVERY SOFTWARE BUILT IN REUSABLE COMPONENTS

The aforementioned strategy for rapid assembly of reusable software components requires to rethink and modernize the strategy for recovery from hardware failures and software errors. Indeed, it would make no sense to rapidly assemble service software if it had to take a considerable amount of time to provide the software dependability. Therefore, it is necessary to develop modular recovery software which can be built automatically during the software assembly process. Using more dependable components will release the fault tolerance designer from the need to worry about the internal behavior of the components as a result of faults, and focus instead on the system as a whole. In addition to the standard reliability techniques used in commercial products, it may be feasible to implement "boundary" hardware reliability techniques to aid system recovery.

In addition to using the aforementioned software recovery techniques, variable degrees of service dependability can be achieved by replicating on multiple platforms services and/or service resources, depending on their criticality [Bar93] and the economic value of the de-

sired level of service dependability. Evidently, a successful application of service component replication depends on more rigorous application programming technologies.

12.5 CONCLUSIONS

Although traditional methodologies for designing large dependable software have been feasible in many industries (telecommunications, space, aeronautics, etc,), these methodologies have proven expensive, lengthy, and inflexible. In order to maintain leadership and competitiveness, software manufacturing will have to move toward industrialization which can be enabled by reusability of specialized software components for both operational and recovery softwares.

This chapter is based on the work of many engineers who, over the years, have made the 5ESS® switch a success. They have met the challenges of producing a very large system which has exceeded customers quality and availability expectations. The experience gained and the dependability improvements achieved over the years can now be embodied in a "leaner" recovery strategy more in line with the economic trends of the nineties.

ACKNOWLEDGEMENTS

I am deeply indebted to L.Bernstein whose challenging questions and insights have contributed to shape the viewpoints expressed here. My thanks also go to the team of Steel Huang which has produced some of the data used here.

REFERENCES

[Abe79] J. Abe, K. Sakurama, and H. Aiso. An analysis of software project failures. In *Proc. of the 4th. International Conference on Software Engineering*, Munich, September 17-19, 1979.

[ATT86] *Quality: Theory and Practice*, AT&T Technical Journal, 65(2), March 1986.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[Bai64] N. T. J. Bailey. *The Elements of Stochastic Processes*. John Wiley & sons, 1964.

[Bar93] Rod Bark. Fault-tolerant platforms for emerging telecommunications markets. In *Proc. of the Workshop on Hardware and Software Architectures For Fault Tolerance: Perspectives and Towards a Synthesis*, Le Mont Saint Michel, France, June 14–16 1993.

[Ber93] L. Bernstein and C. M. Yuhas. To err is human; to forgive, fault tolerance. *SuperUser Newsletter*, International Data Group, July 1993.

[Bor93] Andrea Borr and Carol Wilhelmy. Highly-available data services for UNIX client-server networks: why fault-tolerant hardware isn't the answer. In *Proc. of the Workshop on Hardware and Software Architectures For Fault Tolerance: Perspectives and Towards a Synthesis*, Le Mont Saint Michel, France, June 14-16 1993.

[Bro95] D. Brown. Design of a highly available switching platform employing commercial components. In *Proc. 1995 International Communication Conference (ICC)*, Hamburg, Germany, 1995.

[Cla92] K. C. Clapp, R. K. Iyer, and Y. Levendel. Analysis of large system black-box test data. Third International Symposium on Software Reliability Engineering, pages 94–103, October 1992. IEEE Computer Society Press.

[Cle86] George F. Clement. Evolution of fault tolerant computing at AT&T. In *Proc. of the One-*

day Symposium on the Evolution of Fault Tolerant Computing, pages 27–37, Baden, Austria, 1986.

[Dal88] Siddhartha R. Dalal and C. L. Mellows. When should one stop testing software? *Journal of American Statistician Association*, 83:872–879, 1988.

[Dal94] Siddhartha R. Dalal and Allen A. McIntosh. When to stop testing for large software systems with changing code. *IEEE Transactions on Software Engineering*, SE-20(4):318–323, April 1994.

[Dem82] W. Edwards Deming. *Out of the Crisis*. Massachusetts Institute of Technology Center for Advanced Engineering Studies, 1982.

[Gac94] Raymond Ga Coté. Desktop telephony. *Byte*, pages 151–154, March 1994.

[Gaf84] J.E. Gaffney. Estimating the number of faults in code. *IEEE Transactions on Software Engineering*, SE-10(4):459–464, July 1984.

[Goe85] Amrit L. Goel. Software reliability models: assumptions, limitations, and applications. *IEEE Transactions on Software Engineering*, SE-11(12):1411-1423, December 1985.

[Gra92] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*, pages 60–61. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Gra88] E. L. Grant and R. S. Levenworth. *Statistical Process Control*. McGraw-Hill Book Company, 1988.

[Har93] *Hardware and Software Architectures For Fault Tolerance: Perspectives and Towards a Synthesis*, Le Mont Saint Michel, France, June 14–16 1993.

[Hua93] Yennun Huang and Chandra M. R. Kintala. Software implemented fault tolerance: technologies and experience. In *Proc. of the 23rd International Symposium on Fault Tolerant Computing (FTCS-23)*, Toulouse, France, June 22-23 1993.

[Huy93] T. Huynh, C. Jutla, A. Lowry, R. Strom, and D. Yellin. The global desktop: a graphical composition environment for local and distributed applications. *IBM TJ Watson Technical Report 93-227*, November 1993.

[Kol92] M. V. Kolipakam, G. Y. Wyatt, and S. Y. Yeh. Distributed telecommunication service architectures: design principles and evolution. In *Proc. of the 2nd International Conference on Intelligence in Networks*, pages 1–5, Bordeaux, 1992.

[Kre83] W. Kremer. Birth-death and bug counting. *IEEE Transactions on Reliability*, R-32(1):37–46, April 1983.

[Kru89] G. A. Kruger. Validation and further application of software reliability growth models. *Hewlett-Packard Journal*, pages 75–79, April 1989.

[Lap92] J.C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, Wien-New York, 1992.

[Lev87] Y. Levendel. Quality and reliability estimation for large software projects using a time-dependent model. In *Proc. of COMPSAC87*, pages 340–346, Tokyo, Japan, October 1987.

[Lev89a] Y. Levendel. Defects and reliability analysis of large software systems: field experience. In *Proc. of the Nineteenth International Symposium on Fault-Tolerant Computing*, pages 238–244, Chicago, June 1989.

[Lev89b] Y. Levendel. Quality and reliability estimation: a time-dependent model with controllable testing coverage and repair intensity. In *Proc. of the Fourth Israel Conference on Computer Systems and Software Engineering*, pages 175–181, Israel, June 1989.

[Lev89c] Y. Levendel. The manufacturing process of large software systems: the use of untampered metrics for quality control. *National Communication Forum*, Chicago, October 1989.

[Lev90a] Y. Levendel. Reliability analysis of large software systems: defect data modeling. *IEEE Transactions on Software Engineering*, February 1990.

[Lev90b] Y. Levendel. Large software systems development: use of test data to accelerate reliability growth. *First IEEE International Sympoium on Software Reliability Engineering*, Washington, DC, April 12-13 1990.

[Lev91a] Y. Levendel. Using untampered metrics to decide when to stop testing. In *TENCON91 Preconference Proc.*, volume II, pages 352–356, Delhi, August 1991.

[Lev91b] Y. Levendel. The manufacturization of the software quality improvement process. In *TENCON91 Pre-conference Proc.*, volume II, pages 362–366, Delhi, August 1991.

[Lev95] Y. Levendel, and J. Lumsden. One-step service creation for intelligent networks applications. In preparation for *Proc. ISS95*.

[Lip82] M. Lipow. Number of faults per line of code. *IEEE Transactions on Software Engineering*, SE-8(5):437–439, July 1982.

[Lit89] B. Littlewood. *Forecasting Software Reliability*, Center for Software Reliability Report, The City University, London, February 1989.

[Lyu95] Michael R. Lyu, editor. *McGraw-Hill Software Reliability Engineering Handbook*. McGraw-Hill Book Company, New York, New York, 1995.

[Mas89] R. L. Mason, R. F. Gunst and J. L. Hess. *Statistical Design & Analysis of Experiments*. Wiley-Interscience, John Wiley and Sons, 1989.

[Mon82] M. Monachino. Design verification system for large-scale LSI designs. *IBM Journal of Research and Development*, 26(1):89–99, January 1982.

[Mus75] J. D. Musa. A theory of software reliability and its application. *IEEE Transactions on Software Engineering*, SE-1(3):312–327, September 1975.

[Mus87] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.

[Mus89] J. D. Musa and A. F. Ackerman. Quantifying software validation: when to stop testing? *IEEE Software*, pages 19–27, May 1989.

[Pry77] N. S. Prywes. Automatic generation of computer programs. *Advances in Computers*, volume 16, M. Rubinoff and M. Yovits, editors, pages 57–125, Academic Press, 1977.

[Rya88] T. Ryan. *Statistical Methods for Quality Improvement*. Wiley-Interscience, John Wiley and Sons, 1988.

[Sys90] *System Verification of the 5ESS® Switch (5E6 Generic)*. AT&T Internal Reports Nos. 1-4, January to June 1990.

[Tag90] G. Taguchi and D. Clausing. Robust quality. *Harvard Business Review*, 90(1):65–75, January-February 1990.

[Yar88] V. N. Yarmolik and S. N. Demidenko. *Generation and Application of Pseudorandom Sequences for Random Testing*. John Wiley and Sons, 1988.