

  
  
CHAPTER 22

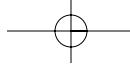
# CASE STUDY: MANAGEABILITY

---

## 22.1 Background

A Fortune 100 company had developed a complex set of database applications some years before. These applications provided a set of extremely mission-critical business functions that were also time-critical. Initially, the applications worked well, but over time the applications grew in complexity with different parts of the application being maintained by different groups. The applications became increasingly unreliable, and the failures did not seem to fit a consistent pattern and often were undiagnosed.

---



## 584 HIGH-ASSURANCE DESIGN

The failures affected multiple prominent parts of the organization, resulting in lots of finger-pointing and confusion as to what the source of the problem was. As a result, an independent assessment was undertaken to find the cause of the problems. The scope of the assessment included organization structure and responsibility, efficacious use of technology, development and testing processes, configuration management, operational processes, user support processes, and application performance. The discussion here will focus on the use of technology, the development and testing processes, and operational and user support processes.

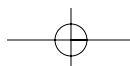
By the time the assessment was undertaken, the applications included PERL scripts, BASH scripts, Java applications, C++ applications, and CORBA—all accessing Sybases databases and running on shared Solaris E6000 or similar systems.

## 22.2 Analysis Approach

The assessment began with interviews of managers, followed by interviews of their staff for detailed information. Code snapshots were requested, as were problem reports for the preceding six-month period. A software process flow and data flow was constructed incrementally, and follow-up interviews were conducted to address various technical questions. Eventually, all pertinent artifacts were identified, and most of the requested code modules had been obtained—although not without considerable stonewalling by the technical teams.

It was important to obtain the support of the technical parts of the organization because they would eventually play an important role in remediation, so an effort was made to establish trust by showing a willingness to point out the challenges that they faced in their work while at the same time identifying the problems that they had created.

The final assessment results were documented in a report, with recommendations at both tactical and strategic levels, and meetings were conducted to discuss the results with all of the stakeholders.



## 22.3 Problems Found

The problems that were found were at all levels. There were major systemic problems that resulted from the organization's structure and division of responsibility, as well as significant process problems and particular staff-related problems—in particular, very substantial key-person dependencies. The technical problems were, of course, a result of these higher-level problems, but the following discussion focuses on the technical and process problems.

### 22.3.1 Too Much Concentration of Knowledge

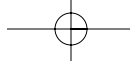
A very small group of people who had developed the code knew how things actually worked. This small group was no longer able to support the applications because their success in building the applications caused them to be drafted into other projects that required saving. Most of the maintenance and support team was effectively in the dark due to staff turnover, the lack of design documentation, and the complexity and obscurity of the implementation.

### 22.3.2 Insufficiently Robust Development Processes for a Mission-Critical System

The processes for configuration management and migration across testing and live environments were highly inadequate. Almost everything was done ad-hoc by technical “gurus,” with email as the only traceability mechanism.

### 22.3.3 Too Much Focus on Productivity at the Expense of Reliability

The business side applied immense pressure to add functionality while expecting high reliability, yet expected that reliability was a natural byproduct of development and did not allocate sufficient resources or priorities to address reliability.



#### **22.3.4 Poorly Defined External Interfaces**

The applications relied on large amounts of data from many external systems that were operated by outside companies. The interfaces to these external systems did not have definitive specifications: Specifications were either incomplete, non-existent, or empirically derived.

#### **22.3.5 Many Single Points of Failure**

The primary application process was designed as a pipeline, with no failure resiliency built in. If anything went wrong, the entire process had to stop. Restarting from specific checkpoints was possible, but no processing could continue unless all problems were resolved. This was a result of the implementation and not due to the nature of the calculations.

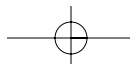
#### **22.3.6 Poor Encapsulation of Function**

The code exhibited grossly inadequate encapsulation of business functionality. The use of scripting languages such as PERL exacerbated this problem, making it impossible to have unit tests.

#### **22.3.7 Inability to Perform Effective Problem Diagnosis**

Support staff were unable to diagnose problems in an efficient and effective manner due to these factors:

- Inadequate or inappropriate error handling by applications and a failure to save forensic information.
- Meaningless or misleading error messages, such as “connection failure” when the problem was invalid data.
- Massive amounts of data written to logs, with no prior consideration of the organization of this information.
- Every error resulted in a telecommunications page (via a paging gateway) to support staff, with no discrimination regarding what kinds of errors should be reported. Upper management was paged automatically when problems were unresolved after a specified amount of time.



- Application code error handling was grossly inadequate for the kind of application. For example, a connection failure might be transient due to the reboot of a server, but the application would give up. Problems in remote connections resulted in orphaned objects and connection pools that ran out of resources instead of closing bad connections and creating new ones in a robust manner.
- There was no scheme for logging entry and exit from critical routines, so when errors occurred, there was insufficient information available to report where the application was in the process.

### **22.3.8 Lack of Monitorability**

There was no way to monitor the progress of the application, except by scrutinizing intermediate output files and checking for errors.

### **22.3.9 Unsupported Concurrency**

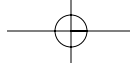
Use of scripting made business logic non-reentrant. Scripts often performed thousands of independent database transactions. Meanwhile, if processing schedules slipped, unintended concurrent access could result from other processes—corrupting data.

### **22.3.10 Overly Complex Administration**

The applications were not designed with operators in mind. Operating them required very complex and interdependent configurations, with no dashboard or tool to provide visibility into the configuration. Given the large number of interoperating applications, errors were inevitable.

### **22.3.11 Lack of Adequate Regression Testing**

There was no automated regression test suite. Testing was performed by running the current day's data and comparing it manually with small sets of calculations done through inde-



pendent means as well as comparing it with the prior day's data to see if it "looked right." This approach was a byproduct of the application domain (financial services and accounting), which has a tradition of validating data instead of processes.

### **22.3.12 No Specification of Failure Handling Requirements**

There were no requirements documented for how the application should behave when something went wrong, so it was no wonder that there were no provisions in the application for failure resiliency. This was largely a result of the fact that the problem response team was not an authoritative player in the requirements definition process.

### **22.3.13 Absence of Failure Testing**

There was no failure test suite. Failures were viewed as "bugs" and troubleshoot, rather than planned for. In a complex environment involving many independent systems and networks, failure resiliency is critical and connection or other component failures cannot be treated as extraordinary by applications.

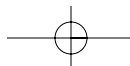
### **22.3.14 Inadequate Design or Design Documentation**

Design documents were few and far between and out of date.

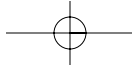
## **22.4 Remediation**

Organizational changes were recommended and implemented, but the focus here is on the technical problems and software development and maintenance processes. The recommendations and ensuing remediation activities included the following:

1. Migrate logic from PERL and BASH scripts to a programming language supporting encapsulation of function.
2. Design an error handling and reporting strategy using the pattern "Using Checked Exceptions for All Lifecycle Errors" (see Section 18.2.1).



3. Design an application context propagation approach and mechanism using the pattern “Application-Based Propagation of Client Context” (see Section 18.2.5).
4. Design an application event assessment and reporting approach and mechanism using the pattern “Adding Manageability Aspects to the Exception Base Class” (see Section 18.2.6).
5. Design an approach for generating messages at critical processing points to enable monitorability. (See the pattern “Heartbeat” in Section 16.4.1, the pattern “Embedded Progress Interpreter” in Section 16.4.2, and the principle “Log Only at Predetermined Layer Interfaces” in Section 18.2.2.)
6. Design an application management and operation console.
7. Establish a regression test suite (see the principle “Build a Regression Test Suite for the Application’s Transactions” in Section 13.12).
8. Enhance the development process to require a design document (see the principle “Ensure That Architectural Concepts and Decisions Are Documented and Disseminated” in Section 19.2), and to require that component specifications must include failure response requirements (see the principle “Ensure That All Implied or Assumed Requirements Are Identified” in Section 19.2).
9. Enhance the configuration management processes and formalize the separation of environments (see the principle “Ensure That There Exists Full Configuration Management of All Artifacts” in Section 19.2).
10. Establish a process, including independent design reviews, that includes all stakeholders (see the principle “Ensure That Design Is Reviewed by Different Areas of Expertise” in Section 19.2), including the operational and response teams.



## Reflection

This case study is an excellent example of a situation in which an application that is developed by a small team can degrade over time. It is also an example of the problems faced by support staff in complex environments in which there are many systems that need to be maintained and often restarted, upgraded, or reconfigured. Basically, applications built for small close-knit environments do not scale to larger and more complex environments. Increasing scale needs to include provisions for enhancing the processes as well as enhancing the applications, their resiliency, and their ability to be monitored.

