

Case Study: Dramatic Software Reliability Enhancement In Only Six Weeks

Abstract

A critical law enforcement infrastructure application had been experiencing failures at an increasing rate. The system was examined by a team of analysts and its flaws categorized. A plan was developed and executed to recover from failures caused by these flaws so that the system could still operate in a pseudo-reliable manner even though the flaws continued to exist. The repairs are not considered to be viable on a permanent basis, but will “buy time” while the system is improved.

Background

The application in question is a critical system used by law enforcement across the US. The system must have high availability, reliability, and security. The system was originally developed as a proof-of-concept, but became a de-facto production system. Over time, the system started to exhibit reliability problems, at a seemingly increasing rate as the code was maintained. Funds were allocated to replace the system with a true production-quality system, but in the meantime the reliability of the current system needed to be shored up.

Assured By Design helped the prime contractor for the system to plan and execute the work described here. The system is implemented as a set of Windows services running on Windows Server 2003 under a Windows cluster. The services access an Oracle database.

The Good, the Bad, and the Ugly

The underlying causes of problems with the current system ranged from methodical problems, design problems, and staff problems. The staff problems consisted primarily of turnover, and these issues will be discussed in the context of methodology.

Methodological Problems

Key-Person Dependencies

The original technical leader had left the company. This person had been a very hands-on manager who focused entirely on technical decision-making and not at all on creating processes and developing new leaders within the team. As a result, she left behind a process vacuum, resulting in near chaos, because no one on her team was prepared to pick

up where she had left off. Shortly after her departure another key person left, leaving only one person who had an in-depth understanding of the application's complex, interwoven internal design.

Reliability Requirements Were Not Addressed by the Application Design

As in many government projects, there was a large budget for hardware, but spending on software quality was chiseled to the bone. As it turned out, the application was the major source of unreliability. For example, the historical ratio of software-caused system failures to hardware failures was fourteen to one.

Design Problems

No Separation Of business Logic From Infrastructure Logic

Business logic in the software was completely intertwined with implementation-specific code dealing with threading and resource management. This made it practically impossible for a newcomer to tease apart which code performed business logic.

To make matters worse, the original developers of the system had created a technically-intricate implementation of unnecessary complexity, based on layers of state machines. Reading the code required practice.

Multithreading

Multithreading was implemented using an undocumented pattern. The Windows multithreading APIs were not well understood before work began, and were not tested to discern actual behavior. As a result, significant problems resulted, with anomalous behavior, threads living on beyond termination and accessing deallocated objects, and no confidence

that there could not be deadlocks or race conditions.

Application Flow

Transaction flow was sparsely documented. The entire logical flow of the programs was controlled by a state transition table stored in the database. Yet, the state model was not documented. Control flow could only be deduced by executing the program and watching it in a debugger. As a result of the poorly-understood application flow, a race condition occurred that caused the application to crash for many hours, at a time when the prime contractor was bidding for followon work.

Unreliable External Resources

The application utilized several external resources. These included (1) Microsoft Exchange, accessed via MAPI, (2) Oracle, and (3) a SAN file system accessed via a utility API layer that provided parsing and formatting.

To address recovery from resource failure, the team had implemented a thread termination strategy, in order to forcefully end stuck resource connections. This was poorly thought out and not prototyped, and resulted in a multitude of problems that caused the current release to slip and eventually be canceled due to technical problems that the team could not fix.

Inadequate Error Handling

Error handling within the application was handled in a largely ad-hoc manner, dealing with those conditions that were observed during testing but no effort to comprehensively identify potential error states before they occurred.

Absence of Testability Features

The application had been developed with no consideration of testability. Thus, all the well-known benefits of continual testing throughout a development cycle were not obtainable.

A lack of encapsulation of functionality (monolithic state-driven functions rather than decomposition into a hierarchy of well-defined functions with their own interfaces) made it difficult to unit test. The application design did not incorporate a transactional layer, and this made it impossible to create a transactional regression test suite. The lack of a "business functionality layer" made it difficult to create automated tests of the application's user-level functionality. As a result, significant testing was saved for the end of each release and was largely manual. In addition, there was no concurrency testing, and limited failure recovery testing.

Lack of Manageability

A significant investment had been made into monitoring the operational environment, using BMC application monitoring software, but no provisions had been made for the monitorability of the application state. In particular, the application did not generate health events that were externally monitorable. Failure events were merely logged.

Ineffective Logging and Monitoring

Production-level logging produced no diagnostic information – even though the application was nowhere near a production-stable state. When logging was increased to a level sufficient to allow diagnosis, the logs saturated – generating files too large to even open for examination.

New Setting

When the project manager left, the new team leader did his best to hold things together. It took a year to put processes in place. In the meantime, the quality of deliverables suffered greatly, putting future work at risk, as described above.

Eventually, the entire focus shifted to reliability, at the expense of all other priorities. This was at the direction of the customer. Despite this, the customer did not realize the full scope of the problems, and perceived that there were merely some "bugs" that had to be hunted down and fixed. As a result, an almost trivial level of funding was provided for the "reliability release".

Remediation Planning

The reliability enhancement effort began with an exhaustive review of the code and design. The sole objective was to uncover sources of reliability and document the as-built design (at a high level) in the process. These activities took several weeks, in a highly rushed mode, and resulted in a fairly good understanding of the conceptual design but limited understanding (except in the minds of the remaining programmer) of the line-by-line implementation. Thus, at this point, others on the team were no closer to being able to modify the existing code, but at least there was a collective understanding of how things fit together and flowed. Some of the artifacts produced in this period included:

- A comprehensive state transition diagram for all application transactions.
- A set of control flow diagrams –

one for each important module (a total of about fifteen).

- A list of coding errors that needed to be fixed. There were uncovered by inspection of the code.
- A document describing the threading design.

After the review period, a full-team all-day brainstorming session was held, facilitated by the new team manager, in which all problems were identified across the project, from every perspective. This occurred on a Friday, and over the weekend several team members independently drafted sets of ideas that they presented on Monday for how to move forward. A second team-wide meeting was held to develop a plan of action. The objectives expressed were to (1) increase reliability, and (2) increase maintainability. This meeting resulted in an outline for a plan of action. The plan outline defined seven teams for performing analysis and each drafting a plan for execution in its respective areas. These teams were: (1) Multithreading, (2) Transaction integrity, (3) Error handling, (4) Logging, (5) Testing, and (6) Monitoring.

Planning Outcome

A decision was made to make reliability the sole focus of the release: there was insufficient time to address maintainability in the same release.

Multithreading

Given other problem areas that had to be addressed, there was insufficient time to verify the nature of the multithreading problems, and so it was decided to abandon thread termination and terminate and restart the entire

process (a Windows service) whenever thread termination was necessary. We would rely on the Windows Server cluster to automatically and promptly restart the service.

Transaction Integrity

It was found that there was widespread lack of isolation and atomicity, due to the simultaneous use of MAPI and Oracle in separate transactions. There was insufficient time in the current release to implement a comprehensive repair to the identified flaws. However, the risks that existed were primarily that individual logical transactions would fail. The risk of this apparently was not severe, because it had not been an identified support issue. If a transaction failure caused an internal error that crashed a thread, we were already implementing a mechanism to recover from that. Therefore, for the current release, we recommended no action.

Error Handling and Logging

A comprehensive review of error reporting levels was performed. The levels were made consistent, and a strategy was devised to ensure that useful information was logged for production-level error reporting, in combination with a new logging strategy (described below). In addition, a rolling log was implemented, that preserved detailed information for a limited time and then purged detail-level information only if a severe error did not happen during that time.

Testing

There was not sufficient time to address the problem of testability.

Monitoring

TheBMC monitoring configuration

was modified so that it would automatically stop a service if the service stopped writing health messages to a log. The Windows cluster would then automatically restart the stopped service. To achieve this, a custom-written BMC plugin was obtained and installed.

The solution strategies are summarized in Table 1.

Followup

In the months following the deployment of the repairs, the application logs showed that the application experienced a thread crash or other internal failure averaging about four times per month. However, in each case, the

application detected the failure and restarted itself, in most cases without losing any data, and in all cases resuming normal operation within minutes. The experience to the users was of a system with greatly improved reliability, and few detectable failures.

Conclusion

It is possible to shore up an application's reliability, even if the application has severe flaws of many kinds. The fixes will not last forever, but this approach can buy time during which the true sources of unreliability can be planned and addressed in the proper manner.

<i>Team Focus</i>	<i>Solution Strategy Chosen</i>
Multithreading	<ul style="list-style-type: none"> When a thread hangs, detect it, and restart the entire process.
Transaction integrity	<ul style="list-style-type: none"> No change.
Error handling	<ul style="list-style-type: none"> Reduce redundancy of error reporting.
Logging	<ul style="list-style-type: none"> Implement a rolling log.
Testing	<ul style="list-style-type: none"> No change.
Monitoring	<ul style="list-style-type: none"> Monitor event log for absence of output, to detect if a service is hung, and if so, restart it.

Table 1: Solution Strategies