Efficient Strategies for Building Trustworthy Software Systems
Manas Kumar Yogi1 I. Ramyambika Sushruta2

1Assistant Professor 2B.Tech Student
1,2Department of Computer Science and Engineering
Pragati Engineering College Surampalem, Andhra Pradesh, India

Abstract— This paper presents the need for efficient strategies for building trustworthy systems. We incorporate several design issues like open and closed principle or substitution principles, reusability. Using this approach the trustworthiness of the system can be increased from 40-45%. If this factor increases then the marketability of the software too increases. We present the significance of various factors which contribute to the construction of such a system.

Key words: Trust, fault tolerance, software rejuvenation, stability, features

I. INTRODUCTION
Software design usually means a description of the implementation of a software system after it is finished. This “bottom-up” approach can result in working systems, but they are difficult to enhance and it is even more difficult to determine how trustworthy they are. By trustworthy we mean that a software product or component is safe, reliable, and secure. Software design is an important first step, not the final step, when creating trustworthy systems. This chapter provides design techniques and constraints on the software implementation that will lead to making a system trustworthy. The goals of the design process are to create a simple and concise solution. Simplicity improves reliability, and conciseness reduces the time and cost of implementation. Software system development is often dominated by schedule and cost. Sometimes performance and functional technical requirements become an issue. Rarely has trustworthiness been considered in any but the most critical systems, but this is changing. Society as a whole is beginning to recognize that not only must software designers consider how the software will perform, but also they must account for the consequences of failures. Trustworthiness encompasses this concern.

II. SIMPLIFY THE DESIGN
Before getting to the design review, a design simplification process eliminates “gold plating” that was identified as one of the top ten risk items for project success. Gold plating is producing software embodying the most complicated interpretation of the requirements, which occurs when designers have limited domain knowledge and do not understand the few places in the design where generalizations are critical, so they generalize everywhere. It is also the consequence of not using prototypes during the requirements phase. For example, in his seven-team experiment, Boehm remarked:

For example, when confronted with a request such as, “Some users would like to enter data by rows as well as columns,” the developers who relied on specifications would tend to say, “Sure, that’s just another sentence in the specification document.” When confronted with this sort of comment in their prototype review, prototypers had a better feel for the programming implications and tended to say, “We’ll put that in if we have time.” The 3:1 range in product sizes is remarkable, considering that each team was developing essentially the same product. The main reason for this effect appeared to be that prototyping fostered a higher threshold for incorporating marginally useful features into a software product. The process of prototyping gave software developers a more realistic feel for the amount of effort required to add features to a project, and the lack of a definitive specification meant that prototypers were less locked into a set of promises to deliver capabilities than were the specifiers. In the somewhat rueful words of one of the specifiers, remarking on his team’s efforts to fulfill the promises in their ambitious specification, “Words are cheap.”

III. SOFTWARE FAULT TOLERANCE
If we cannot avoid a failure, then we must constrain the software design so that the system can recover in an orderly way. Every software process or object class should provide special code that recovers when triggered. A software fault-tolerant library with a watchdog daemon can be built into the system. When the watchdog detects a problem, it launches the recovery code peculiar to the application software. In call processing systems, this usually means dropping the call but not crashing the system. In administrative applications where keeping the database is key, the recovery system may recover a transaction from a backup data file or log the event and rebuild the database from the last checkpoint. Designers are constrained to explicitly define the recovery method for each process and object class using a standard library. Many highly available and reliable applications are deployed on Microsoft’s Windows NT. Transaction processing and process replication technologies make these applications industrial strength and resistant to failures.

Transaction processing is the most widely used technique for fault tolerance among commercial fault-tolerant products. With a transactional processing system, applications usually have a well-defined transaction boundary, such as updating a record or keeping a communication channel operating in the face of bit errors. When a fault occurs, both the client and the server abort the ongoing transaction and roll back to a clean state. This approach was used in the Safeguard ABM system in its “mission mode.” Special code was added to each transaction to rationally respond to failures. For example, on a divide by zero abort, the processor would clear tracking data for a target that might have vaporized during atmospheric re-entry. Process replication allows faster recovery than transactional processing and provides recovery for non-transactional processes. It is ideal for military, avionics, and telecommunication applications that must continually manage or monitor some physical device. A cold replication assumes there is only one active copy of a fault-tolerant...
process. The order of priority is when the active copy fails, recover the failed process locally; if the local recovery fails, migrate the process to another machine. This process can be done with a cold, warm, or hot replication design. In warm replication, one or more backup processes run on a network, and the primary process periodically checkpoints its state to its backup processes. Only the primary process can provide services to client applications; the backup process receives only checkpoint messages from the primary process. If the primary process fails, one backup quickly becomes the primary and resumes services. A hot replication scheme monitors all replicas of a fault-tolerant process. When a failure occurs with one server, the failure is masked and the computation continues if there is one server running. No rollbacks are necessary on either the client or the server. To successfully implement the checkpoint technique, one needs facilities not provided by Windows, these being application monitoring, application failure recovery, application checkpoint/message logging, file replication, Windows events logging/replay, IP packets dispatching, and IP address fail-over. Commercially, Avaya offers a reusable library called Software-implemented Fault Tolerance (SWIFT). It is a set of reusable software modules for building reliable, fault-tolerant Windows NT, LINUX, UNIX, and JAVA applications. These modules can either stand alone or be integrated into existing software products to provide fault tolerance. Therefore, SWIFT is designed especially for object, process, and application replications using cold, warm, and hot replication schemes. SWIFT detects hang failures in addition to crash failures. These modules emerged from fundamental Bell Laboratories research in fault-tolerant software.

A. **SWIFT S Library Modules Include:**

1) Watchd for process failure detection, recovery, replication management, and distributed system services. It detects application process failures and machine crashes. This watchdog daemon process can run on either a single machine or on a network and uses an adaptive diagnosis protocol to detect machine failures so each Watchd pings its neighbour Watchd; if its neighbour fails, Watchd pings its next neighbour; and so on. Once Watchd detects a failure with an application program, it will restart the application program automatically. If the client application fails more than the threshold given to Watchd, Watchd will reboot the system and restart the application. It also is used to restart an application for rejuvenation purposes.

2) Winckp for transparent process checkpointing and mouse/keyboard events logging and replaying.

3) Libft for data checkpointing, communication messages logging, and recovery.

4) REPL for online incremental file replication and disaster recovery.

5) One-IP for IP packets dispatching, fail-over, and rerouting.

**IV. SOFTWARE REJUVENATION**

The third constraint is to limit the state space in the execution domain. Today’s software runs nonperiodically, which allows internal states to develop chaotically without bound. Software rejuvenation is a new concept that seeks to contain the execution domain by making it periodic. An application is gracefully terminated and immediately restarted at a known, clean, internal state. Failure is anticipated and avoided. Nonstationary software processes are transformed into stationary ones. One way to describe this is rather than running a system for 1 year with all of the mysteries that untried time expenses can harbor, run it only 1 day, 364 times. The software states would be reinitialized each day, process by process, while the system continued to operate. Increasing the rejuvenation period reduces the cost of downtime but increases overhead. One system collecting online billing data operated for 2 years with no outages on a rejuvenation interval of 1 week.

A Bell Laboratories experiment showed the benefits of rejuvenation. A 16,000 line C program with notoriously leaky memory failed after 52 iterations. Seven lines of rejuvenation code with the period set at 15 iterations were added, and the program ran flawlessly. Rejuvenation does not remove bugs; it merely avoids them with incredibly good effect.

This phenomenon was first recognized in the 1970s in the software development for the BISCOM store-and-forward message switching system used by five telephone companies to process customer service requests. The problem then was that hash tables were used to index into a file system. The service requests were different sizes, and the service request numbers followed a structured pattern. The original design tried to maintain the file structures for 30 days or more, which led to many clashes and secondary indices. When service requests were fulfilled, they were deleted. The garbage collection software that tried to reclaim file space was complicated. After months of system aborts, angry customers and frustrated software developers, rejuvenation was born. The system was shut down every night for backup, report generation, and other administrative tasks and the file structure was maintained from shutdown to startup. At startup, the file manager would accept all existing files as new input, recompute the hash tables, and restore the messages. Hash table conflicts became rare. Garbage collection problems were insignificant. The system was more reliable. This added 10 minutes of elapsed time to the startup process and was easily accommodated in the administration procedures. The execution life of the file system was 1 day.

The idea of rejuvenation was applied to Lucent’s Billing Data System (BILLDATS) in the 1980s when it was ported to UNIX. There were no reported outages for at least the first 10 years of use at over 50 customer sites. Rejuvenation worked and then was extended to a UNIX library of features. BILLDATS collects billing information from automatic message accounting transmitters situated in or close to switching offices. BILLDATS is the “middleman” in the billing process. The system collects, validates, and adds identification information regarding origination and destination. This information is transmitted directly to the Revenue Accounting Office, which processes the billing information. Some of BILLDATS more interesting features are as follows:

1) Runs under UNIX

2) Rejuvenates daily
3) Can store 12 to 44 million calls
4) Inserts the switch type and ID onto every call record
5) Collects data from up to 600 switches

The FAA’s Voice Switching Communication System (VSCS) was upgraded in 2003 from UNIX to WinNT. Harris Corporation supplied VSCS. As a result, the Microsoft problem of clock expiration after 49.7 days became an FAA problem that led to a massive failure. Harris did not use rejuvenation technology that could have prevented this failure. During VSCS development, the issue of WinNT as an industrial strength, reliable platform raged in the telecommunications software trade press. Even though Microsoft had upgraded WinNT with clustering technology in a two-node failover configuration, industry skeptics argued against the risk of moving away from UNIX.

The Microsoft software contains an internal clock designed to shut the system down after 49.7 days to prevent it from becoming overloaded with data. Shutdown is better than allowing an overloaded system to keep running and potentially give controllers wrong information about flights. This strategy was the right one given the design and uncertainty of the traffic load. When we try to run software beyond its specified domain, we often fail in obscure ways. A better strategy would have been to use rejuvenation technology weekly and roll over to backup hardware to eliminate the risk of this fault becoming a failure. Greg Martin, the chief FAA spokesman in Washington, said the failure was not an indication of the reliability of the radio communications system, which he described as “nearly perfect.” Harris programmers were operating at, or better than, the state-of-the-practice. They were SEI 3, and apparently the application was robust. The problem is that the software industry is not aware of nor using available tools that would prevent many failures. Even worse, the same problems reoccur because we rarely study software failures with the intention of teaching better methods.

V. HIRE GOOD PEOPLE AND KEEP THEM

This constraint might have been the first because it is so important, but any software organization can adopt the first three constraints as they set about improving the quality of their staff. Hiring good people is not easy. Every shop claims to have the “very best” people; obviously, very few actually could.

The high correlation between defects in the software product and staff churn is chilling. Defects are highly correlated with personnel practices. Groups with ten or more tasks and people with three or more independent activities tend to introduce more defects into the final product than those that are focused. Large software changes are more error-prone than small ones, with changes of 100 words of memory or more being considered large. Hatton reports that defects grow exponentially with size, which may have some relationship to the average size of human working memory. The high 0.918 correlation between defects and personnel turnover rates is telling. When Boeing improved their work environment and development process, they saw 83% fewer defects, gained a factor of 2.4 in productivity, improved customer satisfaction, and improved employee moral.

VI. LIMIT THE LANGUAGE FEATURES USED

Most communications software is developed in the C or C++ programming languages. Les Hatton’s book, Safer C, 16 describes the best way to use C and C++ in mission-critical applications. Hatton advocates constraining the use of the language features to achieve reliable software performance and then goes on to specify instruction by instruction how to do it. He says, “The use of C in safety-related or high integrity systems is not recommended without severe and automatically enforceable constraints. However, if these are present using the formidable tool support (including the extensive C library), the best available evidence suggests that it is then possible to write software of at least as high intrinsic quality and consistency as with other commonly used languages.” For example, a detailed analysis of source code from 54 projects showed that once in every 29 lines of code, functions are not declared before they are used.

C is an intermediate language between high level and machine level. There are dangers when the programmer can drop down to the machine architecture, but with reasonable constraints and limitations on the use of register instructions to those very few key cases dictated by the need to achieve performance goals, C can be used to good effect. The alternative of using a mixture of assembly language and high-level language code brings with it the headaches of managing configurations and integrating modules from different code generators. The power of C can be harnessed to assure that source code is well structured. One important constraint is to use C function prototypes or special object classes for interfaces.

VII. LIMIT MODULE SIZE AND INITIALIZE MEMORY

All memory should be explicitly initialized before it is used. Memory leak detection tools should be used to make sure that a software process does not grab all available memory for itself, leaving none for other processes. This creates gridlock as the system hangs in a wait state because it cannot process any new data.

VIII. CHECK THE STABILITY

Software developers know that their systems can exhibit unexpected, strange behavior, including crashes or hangs, when small operational differences are introduced. These are not random events. They may be the result of new data, execution of code in new sequences, or exhaustion of some computer resource such as buffer space, memory, hash function overflow space, or processor time. Fixes and upgrades create their own errors. The fact that the only recourse has been exhaustive retesting limits the growth of software productivity in enhancements to existing systems and modules. Experienced software managers know to ask, “What changed?” when a system that has been performing reliably suddenly and catastrophically fails. Under current methods of software production, systems are conditionally stable only for a particular set of input and a particular configuration.
A. Software System Is Stable If A Bounded Input Creates A Bounded Output. Instabilities Develop In The Following Circumstances:

(1) Computations cannot be completed before new data arrive. (2) Round-off errors build or buffer usage increases to eventually dominate system performance. (3) An algorithm embodied in the software is inherently flawed.

Feedback control theory makes it possible to design adaptive software that meets prespecified performance requirements. Design controllability and observability are possible with feedback control. Controllability is a measure of the ability to use a system’s external inputs to manipulate its internal state. Observability is a measure of how well internal states can be inferred by knowledge of external outputs. Many real-time systems make control decisions. These decisions are usually made by software and based on feedback from the hardware under its control (termed the plant). Such feedback commonly takes the form of an analog sensor that can be read via an A/D converter. A sample from the sensor may represent position, voltage, temperature, or any other appropriate parameter. Each sample provides the software with additional information upon which to base its control decisions.

The ethical software engineer would not implement this specification as it stands. This feedback control technique is being used to analyze these software problems:

1) Web caching QoS
2) Active queue management in networks
3) Processor thermal controls
4) Online data migration in network storage
5) Real-time embedded networking
6) Control middleware
7) Real-time scheduling
8) Target tracking

We must learn to approximate nonlinear systems with linear ones to use this technique widely. Be aware that modeling mapping control objectives for a system to feedback control loops is challenging.

IX. BOUND THE EXECUTION DEMAND

Software needs to perform reasonableness checks on all inputs and outputs. For example, the system should not accept a result of an altitude calculation that claims that an aircraft is at 67ft off the ground when the plane is warming its engines on the ground. The domain experts specify a range of acceptable values for all inputs and outputs, and the software needs to validate that the values fall within their defined ranges. When they do not, then fault-tolerant software is invoked. Input validation on all data coming from user fields on an input screen is now common. A good design approach is to have two stages of abstraction. The first allows the human factors designer to arrange the fields to best fit the task. The second allows domain experts to validate the data.

Then data pass through an interface object or structured language to the processing software. The processing programs receive data that are normalized and validated, which isolates the processing programs from the idiosyncratic behavior of the human operator and of the external environment. It reduces the need to change the application to stay in lock step with screen changes. If the resulting system does not provide the required performance because of this indirection, it is best to buy a faster machine. This is a hardware problem, not a software problem. Should this argument fail with the budget manager, then the designer must invent Fast Path processing to bypass the overhead of indirection. The architecture becomes more tightly coupled and therefore more complex, which causes the software to become harder to fix and extend. Fast Path processing is used for just this purpose, to bypass the overhead of indirection, by IBM in their database access system.

Unfortunately designers are not always as careful as IBM was in bounding and qualifying their interface. Here is a case in which a simple index was not bounded. We have the ethical responsibility to teach the cause of the failure to our colleagues so that designers can avoid it in the future.

X. REDUCE ALGORITHM COMPLEXITY

Often we find ourselves trying to solve a problem that is similar to a problem we’ve already solved. In these cases, often a quick way of solving the new problem is to transform each instance of the new problem into instances of the old problem, solve these using our existing solution, and then use these to obtain our final solution. This is perhaps the most obvious use of reductions. Another, more subtle use is this: suppose we have a problem that we’ve proven is hard to solve, and we have a similar new problem. We might suspect that it, too, is hard to solve. We argue by contradiction: suppose the new problem is easy to solve. Then, if we can show that every instance of the old problem can be solved easily by transforming it into instances of the new problem and solving those, we have a contradiction. This establishes that the new problem is also hard. One way to simplify computations is by making reasoned approximations. Approximate reasoning formalism proposed by fuzzy logic can be used to obtain a logic in which the models are the probability distributions and the theories are the lower envelopes. In such a logic the question of the consistency of the available information is strictly related with the one of the coherence of partial probabilistic assignment.

XI. FACTOR AND REFACTOR

Factoring is the mathematical technique of finding common terms in an equation. Software designers need to look for common requirements, functions, and code throughout software development. “Refactoring” tweaks the factoring concept and applies it to software design. Refactoring defines the software technology aimed at reducing the size of the software by finding and eliminating redundant functions and code and dead-end code. Refactoring is the redesign of software in ways that do not change its functionality. The idea is that the first and second iterations of the software design and implementation stressed understanding the feature, the problem domain, and getting the software to work. Refactoring is left to the third iteration—“make it work better.” Refactoring can be used for small changes to incrementally improve structure. As defined by Martin Fowler:

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is
a series of small behavior preserving transformations. Each transformation (called a “refactoring”) does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it’s less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Refactoring can also be used to make major changes to a module by indicating it needs to be rewritten, but by keeping its functions and its interfaces constant while changing its internal structure. One approach is “refactoring to patterns” that matches refactoring—the process of improving the design of existing code—with patterns, the classic solutions to recurring design problems. Refactoring to patterns suggests that using patterns to improve an existing design is better than using patterns early in a new design. Refactoring was used to great effect in the early 1970s when it was applied to the redesign of a very buggy radar controller. The software was redesigned and reimplemented to make it robust with exactly the same functionality and interfaces. The project was called “Radar Control Maintainability Improvement.” The idea was embraced by the customers because of catastrophic problems with the existing code. By not changing the functions or the interfaces, new features in other modules could be integrated without close release coordination with the new radar controller. The radar controller size was reduced from 5355 LOC to 2681 LOC, and the concept of “design for maintainability” was created. The goal of the designers and implementers was to reduce CPU, RAM, and I/O use while delivering a failure-free module. They accomplished their goal. Allocating as much as 20% of the effort on a new release to improving the maintenance of the system pays large dividends by making the system perform better, avoiding failures induced by undesired interactions between modules, and reducing the time and space constraints on new feature designs. The goal is to reduce the amount of processor time old modules use, the amount of memory they occupy, the amount of I/O they trigger while holding their interfaces fixed. Other modules may be modified or new ones added to provide new features. This strategy naturally leads to reuse within the system. The greatest economic benefit is to reuse software at the application level.

Unfortunately this process is not widely deployed because of the emphasis on new features. A bug-ridden store-and-forward system did use this concept through the 1970s and as a result grew to be extremely reliable. Rather than being tossed out and replaced with the next new thing, it continued to switch messages until 1995, when spare parts could no longer be obtained for the hardware. An unexpected benefit of “design for maintainability” was that new modules and those being upgraded for new features were also more reliable because developers did not face harsh space or time constraints. They could focus on getting their module to work while others concerned themselves with reducing the size of modules that did not contain new features.

XII. CONCLUSION

The dependency on software systems intensifies the consequences of software failures. This paper presents the importance of trustworthy systems in software industry. Several software vendor consortia plan to develop “Trusted Computing” platforms. These initiatives focus primarily on security, but trustworthiness is a much broader concept. The hope for the software industry rests with people who recognize this responsibility and embrace it. The software industry seems to be exempt from liability suits and from the legal need to practice due diligence. The underlying problem with making systems trustworthy is not technical—it is the legal and business structure of the software market. This tacit exemption slows the adoption of trustworthy technology because every business responds primarily to what the customer demands in order to prosper and conversely to what will damage the business if it is not provided. Our paper is a sincere effort to indicate the importance of valuable design considerations while building up a software system keeping in mind the trust factor.

REFERENCES