Design for Safety

Unfortunately, everyone had forgotten why the branch came off the top of the main and nobody realized that this was important.

Trevor Kletz
What Went Wrong?

Before a wise man ventures into a pit, he lowers a ladder — so he can climb out.

Rabbi Samuel Ha−Levi Ben Joseph Ibm Nagrela

Design for Safety

- Software design must enforce safety constraints
- Should be able to trace from requirements to code (vice versa)
- Design should incorporate basic safety design principles

Safe Design Precedence

HAZARD ELIMINATION

- Substitution
- Simplification
- Decoupling
- Elimination of human errors
- Reduction of hazardous materials or conditions

HAZARD REDUCTION

- Design for controllability
- Barriers
  - Lockins, Lockouts, Interlocks
- Failure Minimization
  - Safety Factors and Margins
  - Redundancy

HAZARD CONTROL

- Reducing exposure
- Isolation and containment
- Protection systems and fail−safe design

DAMAGE REDUCTION

↑ Decreasing cost
Increasing effectiveness

Hazard Elimination

- SUBSTITUTION

  - Use safe or safer materials.

  - Simple hardware devices may be safer than using a computer.

  - No technological imperative that says we MUST use computers to control dangerous devices.

  - Introducing new technology introduces unknowns and even unk−unks.

- SIMPLIFICATION

  Criteria for a simple software design:

  1. Testable: Number of states limited
     - determinism vs. nondeterminism
     - single tasking vs. multitasking
     - polling over interrupts
  2. Easily understood and readable
  3. Interactions between components are limited and straightforward.
  4. Code includes only minimum features and capability required by system.
     - Should not contain unnecessary or undocumented features or unused executable code.
  5. Worst case timing is determinable by looking at code.

▫ SIMPLIFICATION (con't)

- Reducing and simplifying interfaces will eliminate errors and make designs more testable.
- Easy to add functions to software, hard to practice restraint.
- Constructing a simple design requires discipline, creativity, restraint, and time.
- Design so that structural decomposition matches functional decomposition.

▫ DECOUPLING

- Tightly coupled system is one that is highly interdependent:
  - Each part linked to many other parts.

    Failure or unplanned behavior in one can rapidly affect status of others.
  - Processes are time−dependent and cannot wait.

    Little slack in system
  - Sequences are invariant.
  - Only one way to reach a goal.
- System accidents caused by unplanned interactions.
- Coupling creates increased number of interfaces and potential interactions.

▫ DECOUPLING (con't)

- Computers tend to increase system coupling unless very careful.
- Applying principles of decoupling to software design:
  - Modularization: How split up is crucial to determining effects.
  - Firewalls
  - Read−only or restricted write memories
  - Eliminate hazardous effects of common hardware failures

▫ ELIMINATION OF HUMAN ERRORS

- Design so few opportunities for errors.
  - Make impossible or possible to detect immediately.
- Lots of ways to increase safety of human−machine interaction.
  - Making status of component clear.
  - Designing software to be error tolerant
  - etc. (will cover separately)
- Programming language design:
  - Not only simple itself (masterable), but should encourage the production of simple and understandable programs.
  - Some language features have been found to be particularly error prone.

▫ REDUCTION OF HAZARDOUS MATERIALS OR CONDITIONS

- Software should contain only code that is absolutely necessary to achieve required functionality.
  - Implications for COTS
  - Extra code may lead to hazards and may make software analysis more difficult.

- Memory not used should be initialized to a pattern that will revert to a safe state.

Turbine−Generator Example

Safety requirements:

1. Must always be able to close steam valves within a few hundred milliseconds.

2. Under no circumstances can steam valves open spuriously, whatever the nature of internal or external fault.

Divided into two parts (decoupled) on separate processors:

1. Non−critical functions: loss cannot endanger turbine nor cause it to shutdown.

   less important governing functions
   supervisory, coordination, and management functions

2. Small number of critical functions.

Turbine−Generator Example (2)

- Uses polling : No interrupts except for fatal store fault (nonmaskable)
  - Timing and sequencing thus defined
  - More rigorous and exhaustive testing possible.
- All messages unidirectional
  - No recovery or contention protocols required
  - Higher level of predictability
- Self−checks of
  - Sensibility of incoming signals
  - Whether processor functioning correctly
- Failure of self−check leads to reversion to safe state through fail−safe hardware.
- State table defines:
  - Scheduling of tasks
  - Self−check criteria appropriate under particular conditions

---

Hazard Reduction

- Passive safeguards:
  - Maintain safety by their presence
  - Fail into safe states
- Active safeguards:
  - Require hazard or condition to be detected and corrected

Tradeoffs:

- Passive rely on physical principles
- Active depend on less reliable detection and recovery mechanisms.

BUT

- Passive tend to be more restrictive in terms of design freedom and not always feasible to implement.

Design for Controllability

Make system easier to control, both for humans and computers.

- Use incremental control:
  - Perform critical steps incrementally rather than in one step.
  - Provide feedback

    To test validity of assumptions and models upon which decisions made

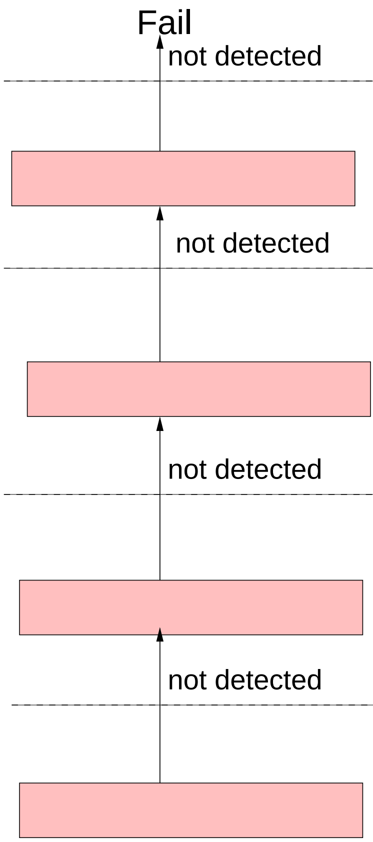
    To allow taking corrective action before significant damage done.
  - Provide various types of fallback or intermediate states
- Lower time pressures
- Provide decision aids
- Use monitoring

Monitoring

Difficult to make monitors independent:

- Checks require access to information being monitored but usually involves possibility of corrupting that information.
- Depends on assumptions about structure of system and about errors that may or may not occur
  - May be incorrect under certain conditions
  - Common incorrect assumptions may be reflected both in design of monitor and devices being monitored.

A Hierarchy of Software Checking



- Observe system externally to provide independent view
- Use additional hardware or completely separate hardware.
- Often observe both controlled system and controller.

- Independent monitoring by process separate from that being checked.
- May check:
  data being passed between modules
  consistency of global data structures
  expected timing of modules or processes

- Can detect coding errors and implementation errors.
- Use assertions: statements (boolean expressions on system state) about expected state of module at different points in execution or about expected value of parameters passed to module.
  e.g. range checks, state checks, reasonableness checks

- Used to detect hardware failures and individual instruction errors.
  e.g., memory protection violation, divide by zero
- Checksums
- Often built into hardware or checks included in operating system.

Software Monitoring (Checking)

- In general, farther down the hierarchy check can be made, the better:
  - Detect the error closer to the time it occurred and before erroneous data used.
  - Easier to isolate and diagnose the problem
  - More likely to be able to fix erroneous state rather than recover to safe state.
- Writing effective self−checks very hard and number usually limited by time and memory.
  - Limit to safety−critical states
  - Use hazard analysis to determine check contents and location
- Added monitoring and checks can cause failures themselves.

Barriers

- LOCKOUTS
  - Make access to dangerous state difficult or impossible.
  - Implications for software:
    - Avoiding EMI
    - Authority limiting
    - Controlling access to and modification of critical variables

      Can adapt some security techniques

- LOCKIN
  - Make it difficult or impossible to leave a safe state.
  - Need to protect software against environmental conditions.

    e.g., operator errors
    data arriving in wrong order or at unexpected speed

    Completeness criteria ensure specified behavior robust against mistaken environmental conditions.

- INTERLOCK
  - Used to enforce a sequence of actions or events.
    1. Event A does not occur inadvertently
    2. Event A does not occur while condition C exists
    3. Event A occurs before event D.
  - Examples:

    Batons
    Critical sections
    Synchronization mechanisms

Remember, the more complex the design, the more likely errors will be introduced by the protection facilities themselves.

Example: Nuclear Detonation

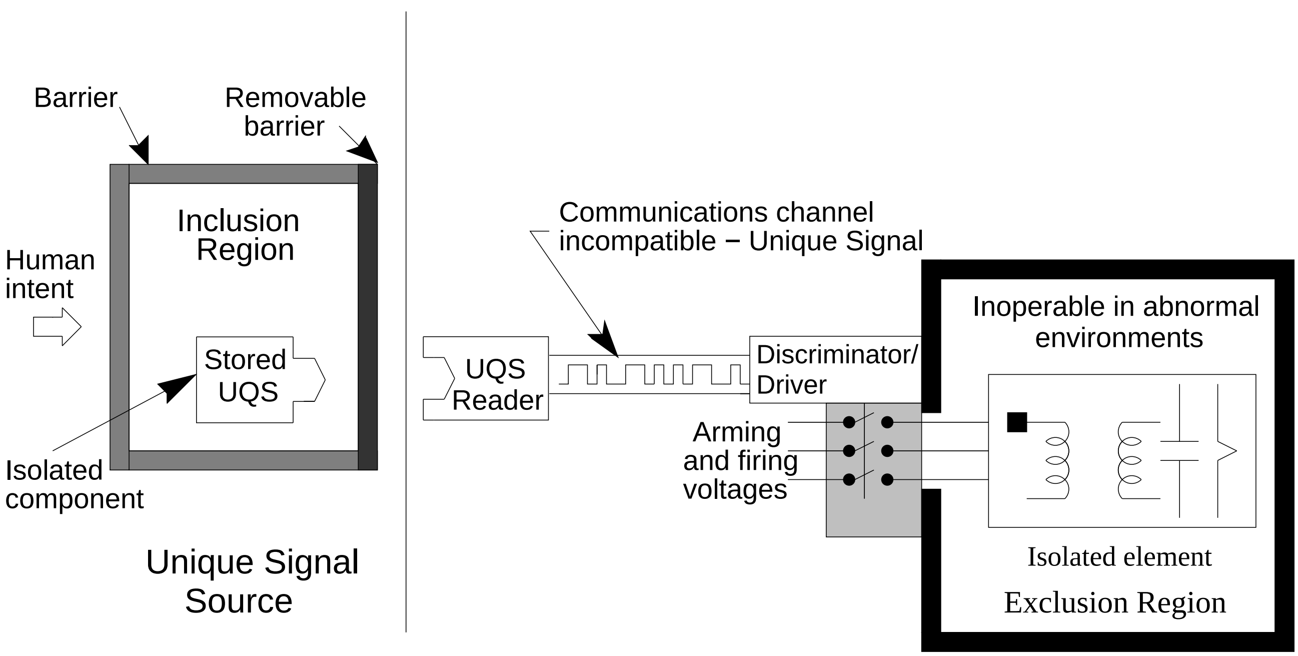
- Safety depends on NOT working
- Three basic techniques (called ''positive measures'')

1. Isolation
   - Separate critical elements (barriers)
2. Inoperability
   - Keep in inoperable state, e.g., remove ignition device or arming pin
3. Incompatibility
   - Detonation requires an unambiguous indication of human intent be communicated to weapon.
   - Protecting entire communication system against all credible abnormal environments (including sabotage) not practical.
   - Instead, use unique signal of sufficient information complexity that unlikely to be generated by an abnormal environment.

Example: Nuclear Detonation (2)

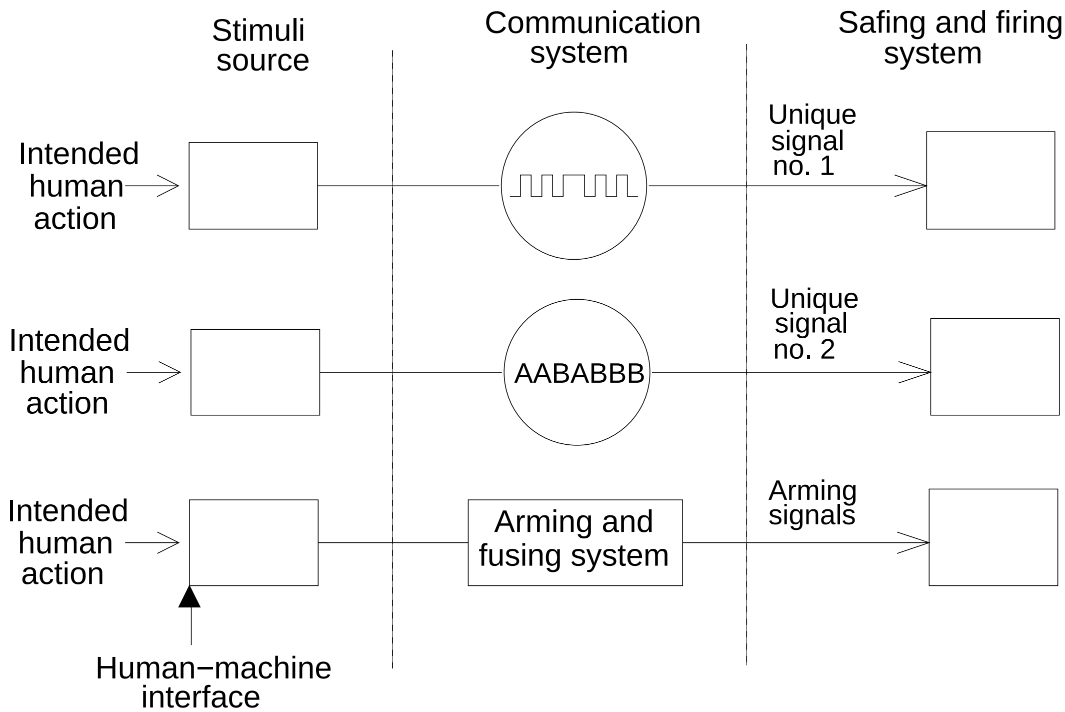
- Unique signal discriminators must:
  1. Accept proper unique signal while rejecting spurious inputs
  2. Have rejection logic that is highly immune to abnormal environments
  3. Provide predictably safe response to abnormal environments
  4. Be analyzable and testable
- Protect unique signal sources by barriers.
- Removable barrier between these sources and communication channels.

Example: Nuclear Detonation (3)



Example: Nuclear Detonation (4)

May require multiple unique signals from different individuals along various communication channels, using different types of signals (energy and information) to ensure proper intent.



Failure Minimization

- SAFETY FACTORS AND SAFETY MARGINS

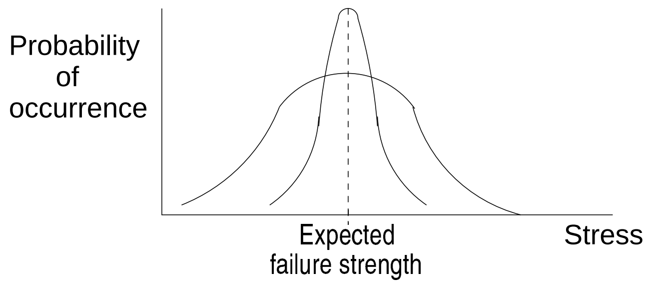
Used to cope with uncertainties in engineering:

- Inaccurate calculations or models
- Limitations in knowledge
- Variation in strength of a specific material due to differences in composition, manufacturing, assembly, handling, environment, or usage.

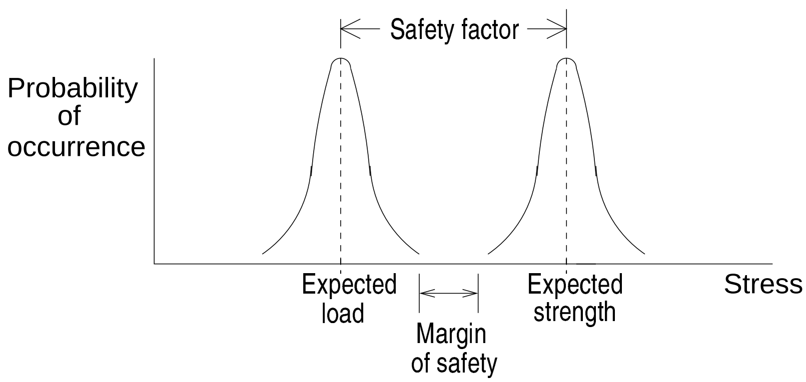
Some ways to minimize problem, but cannot eliminate it.

Appropriate for continuous and non−action systems.

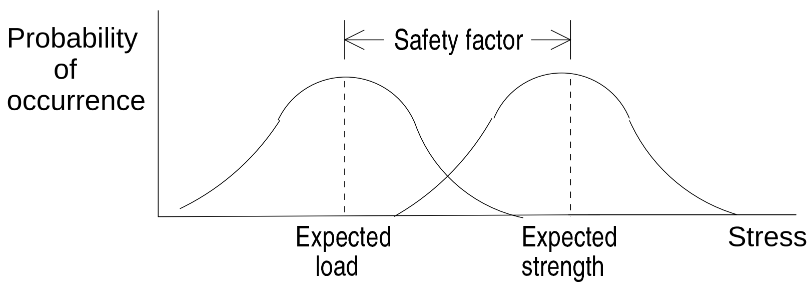
Safety Margins and Safety Factors



(a) Probability density function of failure for two parts with same expected failure strength.



(b) A relatively safe case.



(c) A dangerous overlap but the safety factor is the same as in (b)

▫ REDUNDANCY

Goal is to increase reliability and reduce failures.

- Common−cause and common−mode failures
- May add so much complexity that causes failures.
- More likely to operate spuriously.
- May lead to false confidence (Challenger)

Useful to reduce hardware failures. But what about software?

- Design redundancy vs. design diversity
- Bottom Line: claims that multiple version software will achieve ultra−high reliability levels are not supported by empirical data or theoretical models.

▫ REDUNDANCY (con't.)

- Standby spares vs. concurrent use of multiple devices (with voting)
- Identical designs or intentionally different ones (diversity).
- Diversity must be carefully planned to reduce dependencies.

  Can also introduce dependencies in maintenance, testing, repair

- Redundancy most effective against random failures not design errors.

REDUNDANCY (con't.)

- Software errors are design errors.

  Data redundancy: extra data for detecting errors

  e.g. parity bit and other codes
  checksums
  message sequence numbers
  duplicate pointers and other structural information

  Algorithmic redundancy:

  1. Acceptance tests (hard to write)
  2. Multiple versions with voting on results

Multi (or N) Version Programming

- Assumptions:
  - Probability of correlated failures is very low for independently developed software.
  - Software errors occur at random and are unrelated.
- Even small probabilities of correlated failures cause a substantial reduction in expected reliability gains.
- Conducted a series of experiments with John Knight

  Failure independence in N−version programming

  Embedded assertions vs. N−version programming

  Fault Tolerance vs. Fault Elimination

Failure Independence

- Experimental Design:
  - 27 programs, one requirements specification
  - Graduate students and seniors from two universities
  - Simulation of a production environment: 1,000,000 input cases
  - Individual programs were high quality
- Results:
  - Rejected independence hypothesis: Analysis of reliability gains must include effect of dependent errors.
  - Statistically correlated failures result from:

    Nature of application

    "Hard" cases in input space
  - Programs with correlated failures were structurally and algorithmically very different.

Conclusion: Correlations due to fact that working on same problem, not due to tools used or languages used or even algorithms used.

Consistent Comparison Problem

- Arises from use of finite−precision real numbers (rounding errors)
- Correct versions may arrive a completely different correct outputs and thus be unable to reach a consensus even when none of components "fail.".
- May cause failures that would not have occurred with single versions.
- No general practical solution to the problem .

Self−Checking Software

Experimental Design:

- Launch Interceptor Programs (LIP) from previous study.
- 24 graduate students from UCI and UVA employed to instrument 8 programs (chosen randomly from subset of 27 in which we had found errors).
- Provided with identical training materials.
- Checks written using specifications only at first and then participants were given a program to instrument.
- Allowed to make any number or type of check.
- Students treated this as a competition among themselves.

Fault Tolerance vs. Fault Elimination

Techniques compared:

- Run−time assertions (self−checks)
- Multi−version voting
- Functional testing augmented with structural testing
- Code reading by stepwise abstraction
- Static data−flow analysis

Experimental Design:

- Combat Simulation Problem (from TRW)
- Programmers separate from fault detectors
- Eight version produced with 2 person teams

  Number of modules from 28 to 75

  Executable lines of code from 1200 to 2400
- Attempted to hold resources constant for each technique.

Self−Checking Software (2)

	Already Known Errors				Other Errors Detected			Added Errors
		Detected						
	#	SP	CR	CD	SP	CR	CD	
3a			1					
3b	4							
3c								
6a			2				1	1
6b	3							1
6c								
8a				2				
8b	2							1
8c							1	3
12a		1					1	
12b	2							2
12c			1				1	2
14a								
14b	2							4
14c								
20a								1
20b	2		1		1			2
20c			1					
23a		2						4
23b	2							
23c								
25a			2				1	
25b	3			1				1
25c								
Total	60	3	8	3	1	0	5	22
		Spec	Read	Chks	Spec	Read	Chks	
		KNOWN			NEWLY FOUND			ADDED

Fault Tolerance vs. Fault Elimination (2)

Results:

- Multi−version programming is not a substitute for testing.
  - Did not tolerate most of faults detected by fault−elimination techniques.
  - Unreliable in tolerating the faults it was capable of tolerating.
- Testing failed to detect errors causing coincident failures.
- Cast doubt on effectiveness of voting as a test oracle.
  - Instrumenting the code to examine internal states was much more effective.
- Intersection of sets of faults found by each method was relatively small.

N−Version Programming (Summary)

Doesn't mean shouldn't use, but should have realistic expectations of benefits to be gained and costs involved:

- Costs very high (more than N times)
- In practice, end up with lots of similarity in designs (more than in our experiments)
  - Overspecification
  - Cross Checks

  So safety of system dependent on quality that has been systematically eliminated.

  And no way to tell how different 2 software designs are in their failure behavior.
- Requirements flaws not handled, which is where most safety problems arise anyway.

Recovery

- Backward

  Assume can detect error before does any damage.

  Assume alternative will be more effective.

- Forward

  Robust data structures.

  Dynamically altering flow of control.

  Ignoring single cycle errors.

- But real problem is detecting erroneous states.

Hazard Control

- LIMITING EXPOSURE

  - Start out in safe state and require deliberate change to unsafe state.
  - Set critical flags and conditions as close to code they protect as possible.
  - Critical conditions should not be complementary, e.g., absence of an arm condition should not be used to indicate system is unarmed.

- ISOLATION AND CONTAINMENT

- PROTECTION SYSTEMS AND FAIL−SAFE DESIGN

Protection Systems and Fail−Safe Design

- Depends upon existence of a safe state and availability of adequate warning time.
- May have multiple safe states, depending upon process conditions.
- General rule is hazardous states should be hard to get into and safe states should be easy.
- Panic button
- Watchdog timer: Software it is protecting should not be responsible setting it.
- Sanity checks (I'm alive signals)
- Protection system should provide information about its control actions and status to operators or bystanders.
- The easier and faster is return of system to operational state, the less likely protection system is to be purposely bypassed or turned off.

Damage Reduction

- May need to determine a ''point of no return'' where recovery no longer possible or likely and should just try to minimize damage.

Design Modification and Maintenance

- Need to reanalyze
- Need to record design rationale.