



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Learning from Famous Software Failures

APPLYING COURSE CONCEPTS
TO REAL DISASTERS

Why Study Failures?

George Santayana: "Those who cannot remember the past are condemned to repeat it."

In software engineering:

- Failures expose what happens when good practices are ignored
- Real disasters teach faster than textbook examples
- **Every case maps to concepts from our course modules**

Today: 5 famous failures analyzed through the lens of Chapters 01-04

Learning Objectives

By the end of this lecture, you will:

1. **Identify** how missing requirements caused each failure (Ch04)
2. **Explain** testing gaps that allowed bugs to escape (Ch03)
3. **Recognize** code quality issues that contributed to disasters (Ch02)
4. **Connect** CI/CD and deployment failures to our workflow (Ch01-02)
5. **Apply** lessons learned to your own projects

Our Course Modules as Analysis Framework

Chapter	Topic	Key Concepts
Ch01	Modern Dev Tools	Git, IDE, Python setup, collaboration
Ch02	Code Quality	PEP8, Ruff, Pyright, CI/CD, refactoring
Ch03	Testing	Unit tests, AAA, equivalence classes, boundary analysis, coverage
Ch04	Requirements	Stakeholders, FR/NFR, traceability, Design by Contract

Each failure will be analyzed using these concepts.

The 5 Cases We'll Analyze

Case	Year	Domain	Loss
Therac-25	1985-87	Medical	6 deaths/injuries
Ariane 5 Flight 501	1996	Space	\$370 million
Mars Climate Orbiter	1999	Space	\$327 million
Knight Capital	2012	Finance	\$460 million in 45 min
HealthCare.gov	2013	Government	\$500+ million, 6 users day 1



Case 1: Therac-25 — When Software Replaces Hardware Safety

What Happened?

Therac-25: Radiation therapy machine (1985-1987)

- Delivered **100x intended radiation doses** to patients
- **6 accidents:** 4 deaths, 2 serious injuries
- Root cause: Software-only safety (hardware interlocks removed)

Key insight: The Therac-6 and Therac-20 had similar software bugs, but hardware interlocks prevented accidents. The Therac-25 removed those interlocks.

 Leveson & Turner, IEEE Computer 1993

The Technical Bug: A Race Condition

Operator workflow (fast typist):

1. Select treatment mode: "X" (X-ray)
2. Wait for screen...
3. Change mode to "E" (Electron) within 8 seconds
4. Press "Set" button

Bug: Software updated mode internally,
but hardware turntable hadn't moved yet.
→ High-energy beam fired through wrong collimator

This is a classic race condition - two processes accessing shared state without proper synchronization.

Analysis: Requirements Engineering

Missing Stakeholder Analysis:

- Manufacturer assumed operators would be slow/careful
- No requirement: "System shall prevent beam if turntable not in position"

No Traceability:

- Safety requirements not linked to test cases
- No FR/NFR documentation for race condition handling

Implicit Assumptions (Ch04 "Design by Contract"):

Precondition (undocumented): Mode change takes > 8 seconds
Reality: Fast operators could change in < 8 seconds

Analysis: Testing Fundamentals

No Boundary Value Analysis:

- What happens at the boundary where software/hardware states diverge?
- No tests for "mode change during turntable movement"

Missing Equivalence Classes:

Class	Tested?	Example
Normal operation (slow)	✓	Mode → Wait 10s → Set
Fast mode change	✗	Mode → Change < 8s → Set
Concurrent operations	✗	Multiple state changes

100% code coverage ≠ safe system (Ch03 Coverage lesson)

Analysis: Code Quality

Software Reuse Without Verification:

- Code reused from Therac-6/20 without re-testing assumptions
- Original bugs were **masked** by hardware, not **fixed**

No Static Analysis:

- Race conditions detectable by modern tools
- No equivalent of Ruff/Pyright for concurrent code review

Dead Code Problem:

- Error handling code existed but returned cryptic "MALFUNCTION" numbers
- Operators learned to ignore them (like ignoring linter warnings!)

Discussion: Therac-25

Think about your Road Profile Viewer:

1. What implicit assumptions have you made about user behavior?
2. Do you have tests for "unusual" user interaction patterns?
3. If your `find_intersection()` has a bug, what's the worst outcome?

Therac operators were blamed initially - "user error." Sound familiar?



Case 2: Ariane 5 Flight 501 — The \$370 Million Type Conversion Error

What Happened?

Ariane 5 Flight 501 (June 4, 1996)

- First launch of Europe's new heavy-lift rocket
- **Exploded 37 seconds after liftoff**
- Loss: \$370 million (rocket + 4 satellites)

Root cause: A 64-bit floating point → 16-bit integer conversion overflow

 [ESA Inquiry Board Report](#)

The Technical Bug: Integer Overflow

```
-- Inertial Reference System (Ada code)
BH : Integer_16;  -- Horizontal Bias, 16-bit signed

-- Conversion that caused the crash:
BH := Integer_16(Floating_Point_Value);
-- Ariane 4: Value always < 32,767 ✓
-- Ariane 5: Value reached 32,768+ → OVERFLOW!
```

The overflow triggered an exception → System shutdown → Rocket lost control

Both primary AND backup systems crashed (identical software!)

Analysis: Testing & Boundary Analysis

Classic Boundary Value Failure:

Parameter	Ariane 4 Range	Ariane 5 Range	Tested?
Horizontal velocity	0 - 32,000	0 - 64,000	×
BH (Horizontal Bias)	Fits in 16-bit	Overflows!	×

From our Ch03 lecture:

"Test at boundaries: minimum, maximum, just inside, just outside"

The 16-bit boundary (32,767) was never tested with Ariane 5's trajectory!

Analysis: Requirements Engineering

Software Reuse Without Requirement Review:

- Ariane 4 SRI (Inertial Reference System) reused "as-is"
- **No requirements audit** for new flight profile

Missing NFR (Non-Functional Requirement):

NFR-FLIGHT-001: System shall handle horizontal velocities
up to 70,000 units (Ariane 5 max + margin)

ESA Report: "The requirement that BH should fit in 16 bits was stated in an obscure part of a document. In the code itself, it was nowhere to be found!"

Analysis: Code Quality & Type Safety

This is exactly what Pyright catches!

```
# Python equivalent of the Ariane bug:
def calculate_bias(velocity: float) -> int:
    return int(velocity) # Pyright: OK, but dangerous!

# Better with explicit bounds checking:
def calculate_bias(velocity: float) -> int:
    if velocity > 32767:
        raise ValueError(f"Velocity {velocity} exceeds 16-bit range")
    return int(velocity)
```

Modern type systems can prevent these bugs - Ada had types, but the checks were disabled for "performance"!

Analysis: Dead Code That Killed

The conversion code wasn't even needed after liftoff!

Timeline:

- T-0: Liftoff
- T+37s: BH calculation overflows (code still running)
- T+37s: Exception → Shutdown
- T+40s: This code's purpose ends anyway!

The alignment function served NO PURPOSE after liftoff.
It was kept running for 40 seconds based on Ariane 4 requirements.

Ch02 lesson: Dead code isn't just clutter - it's a liability.

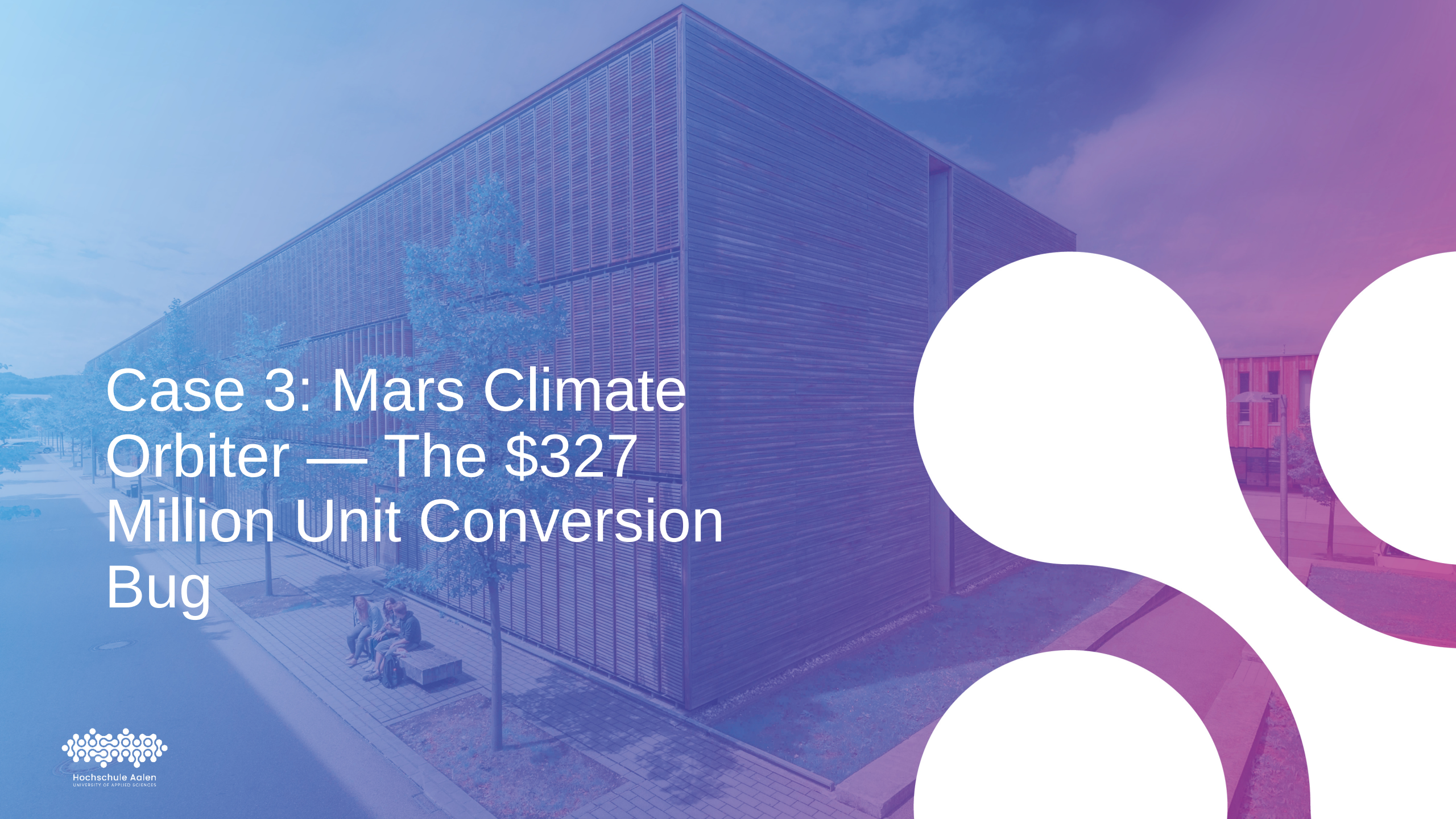


Discussion: Ariane 5

From your Road Profile Viewer experience:

1. Have you reused code from another project without checking assumptions?
2. What's the valid range for `angle_degrees` in `find_intersection()` ?
3. Do you have tests for boundary values like `angle = 360.0` ?

The Ariane team had 10 years of "working" code. It only worked because the environment was different.



Case 3: Mars Climate Orbiter — The \$327 Million Unit Conversion Bug

What Happened?

Mars Climate Orbiter (September 23, 1999)

- NASA spacecraft to study Martian climate
- Lost during Mars orbit insertion
- Passed **57 km above surface** instead of planned 150 km
- Burned up in Martian atmosphere

Root cause: One team used metric (N·s), another used imperial (lbf·s)

 [NASA Mishap Investigation Report](#)

The Technical Bug: Interface Mismatch

Lockheed Martin software (SM_FORCES):

Output: Thruster impulse in pound-force seconds (lbf·s)

NASA navigation software:

Expected: Thruster impulse in Newton seconds (N·s)

Conversion factor: $1 \text{ lbf} \cdot \text{s} = 4.45 \text{ N} \cdot \text{s}$

Result: Navigation thought thrusters were 4.45x weaker than reality

→ Trajectory corrections accumulated wrong → Lost spacecraft

A classic interface contract violation.

Analysis: Requirements & Interfaces

Missing Interface Contract (Design by Contract):

```
# What should have existed:  
class ThrusterData:  
    """  
    Precondition: impulse values in SI units (Newton-seconds)  
    Postcondition: returns trajectory correction in meters  
    """  
    impulse_ns: float # Explicit unit in type/name!
```

NASA Report: "The Software Interface Specification (SIS) stated metric units, but the code produced imperial units."

Ch04 lesson: Requirements must be explicit AND verified!

Analysis: Integration Testing

Unit Tests Passed, Integration Failed:

Test Level	Lockheed	NASA	Combined
Unit tests	✓ Pass	✓ Pass	N/A
Module tests	✓ Pass	✓ Pass	N/A
Integration tests	N/A	N/A	✗ Never done!

From Ch03 Testing Pyramid:

*"Unit tests verify functions work. Integration tests verify they work **together**."*

End-to-end verification was "too late" and "too weak."

Analysis: Code Quality & Types

Modern Python would catch this:

```
# Bad: Implicit units
def calculate_impulse() -> float:
    return 4.5 # lbf·s? N·s? Who knows!

# Good: Explicit units via types
from dataclasses import dataclass

@dataclass
class NewtonSeconds:
    value: float

def calculate_impulse() -> NewtonSeconds:
    return NewtonSeconds(4.5) # Clear and type-safe!
```

Units as types = impossible to mix! (Pyright would catch mismatches)

The Deeper Problem: Process Failures

NASA Investigation Board Chairman:

"The problem was not the error. It was the failure of NASA's systems engineering and checks and balances to detect the error."

Ch02 CI/CD lesson: Quality gates must catch errors automatically!

- No automated unit checks on interface values
- No range validation ("is this impulse value plausible?")
- Manual reviews missed it for 9+ months of flight

Discussion: Mars Climate Orbiter

In your Road Profile Viewer:

1. What are the interfaces between your modules?
2. Do you have explicit contracts for function inputs/outputs?
3. What happens if `find_intersection()` receives coordinates in different units?

Both Lockheed and NASA code worked perfectly - they just didn't work together.



Case 4: Knight Capital — \$460 Million Lost in 45 Minutes

4

What Happened?

Knight Capital (August 1, 2012)

- Deployed new trading software to 8 servers
- **1 server missed the deployment**
- Old "dead" code on that server triggered by new feature flag
- System executed 4 million erroneous orders in 45 minutes
- **Loss: \$460 million** (company nearly bankrupt)



SEC Enforcement Order

The Technical Bug: Deployment + Dead Code

2005: "Power Peg" feature uses flag X in trading router
2007: Power Peg deprecated, but code left in place
2012: New "RLP" feature reuses flag X

Deployment to 8 servers:

- Servers 1-7: New code with RLP ✓
- Server 8: Old code with Power Peg still present x

August 1, 2012:

- Orders with flag X sent to Server 8
- Server 8 runs Power Peg (never reports completion)
- Server 8 buys/sells infinitely → \$460M loss

Analysis: CI/CD & Deployment

No Automated Deployment (Ch02 CI/CD):

```
# What should have existed:
deploy:
  script:
    - for server in $(cat servers.txt); do
      deploy_to_server $server
      verify_deployment $server
    done
    - run_integration_tests
    - if tests_fail; then rollback_all; fi
```

Reality: Manual deployment with no verification checklist

SEC Report: "Knight did not have adequate safeguards to limit risks."

Analysis: Dead Code

Remember from Ch02:

"Dead code costs: Must update for API changes, readers must understand it, may use outdated logic if revived."

Knight Capital's dead code:

- "Power Peg" function unused since 2007
- Left in codebase for 5 years
- Activated accidentally by flag reuse

Our linting rule (Ruff F401/F841): Remove unused code!

Analysis: Testing the Deployment

What tests were missing:

Test Type	Purpose	Knight Had?
Smoke test	Basic functionality after deploy	×
Canary deploy	Test on subset first	×
Config verification	All servers have same code	×
Integration test	New feature with real system	×

SEC finding: "97 automated emails warned of errors before market open. Knight did not act on them."

Analysis: Version Control

Git would have helped:

```
# Proper deployment workflow:
git tag v2.1.0-rpl
git push origin v2.1.0-rpl

# Deploy same tagged version to ALL servers
for server in production-{1..8}; do
    ssh $server "git checkout v2.1.0-rpl && restart_service"
done

# Verify all servers on same commit:
for server in production-{1..8}; do
    ssh $server "git rev-parse HEAD"
done
```

Knight's reality: Manual file copy, no version tracking

Discussion: Knight Capital

For your development workflow:

1. How do you deploy code? Is it automated and repeatable?
2. Do you have dead code in your project? How would you know?
3. What's your rollback plan if a deployment fails?

Knight's technician made one mistake. The system should have caught it.



Case 5: HealthCare.gov — When 6 Users Registered on Launch Day

What Happened?

HealthCare.gov (October 1, 2013)

- Federal health insurance marketplace launch
- **4 million visitors** on day 1
- Only **6 people** successfully registered
- Site crashed within 2 hours of launch
- Cost: \$500+ million, political disaster

 [GAO Report GAO-14-694](#)

The Technical Problems

Launch Day Issues:

1. 250,000 concurrent users (expected: 50,000) → 5x underestimate
2. Database bottlenecks in user registration
3. API timeouts between 60+ connected systems
4. SSL certificate issues
5. Memory leaks causing cascading failures

Root Cause: Big-bang launch without adequate testing

GAO: "CMS launched without verification that systems met performance requirements."

Analysis: Requirements Engineering

Missing NFRs (Non-Functional Requirements):

Requirement	Should Have Been	Actual
Concurrent users	"Support 300,000 users"	Not specified
Response time	"Page load < 3 seconds"	Not specified
Availability	"99.9% uptime"	Not specified

GAO: "Task orders issued when key technical requirements were unknown, including number of states and potential enrollees."

Analysis: Stakeholder Chaos

60 contracts to 33 different vendors:

Problem: No single system integrator!

CMS thought CGI was the lead integrator.

CGI said: "That was not our understanding."

Result: Nobody owned the integration.

Nobody was responsible for end-to-end testing.

Nobody knew if the pieces would fit together.

Ch04 lesson: Clear stakeholder roles and ownership!

Analysis: Testing Failures

The Testing Pyramid Inverted:

What HealthCare.gov had:

```
  /-----\  
  /  Some Unit  \      ← Individual contractor tests  
 /-----\  
 /  ???  Module ???  \    ← Unclear ownership  
/-----\  
/  NO E2E TESTING!!!  \  ← Never done at scale
```

GAO: "Systems were not fully tested prior to launch,
and test documentation was missing key elements."

Analysis: CI/CD Failures

No Continuous Integration Across Teams:

- Each contractor had their own build process
- No shared CI pipeline
- Integration happened "at the end" (big-bang)
- **Key governance review delayed** from March to September
 - Just weeks before October launch!

Ch02 lesson: Integrate early and continuously!

The Recovery: What Worked

After the disaster, they applied our course principles:

1. **Small, incremental releases** (not big-bang)
2. **Continuous monitoring** and alerts
3. **Clear ownership** of each component
4. **Performance testing** before each release
5. **Rapid rollback** capability

By December 2013, the site was working. Same technology, different process.

Discussion: HealthCare.gov

For your team projects:

1. Who is responsible for integration testing?
2. What happens when multiple people work on the same codebase?
3. Have you tested with realistic load/data volumes?

Every contractor's code worked. The integration was the failure.

Synthesis — Mapping Failures to Course Concepts

Failure → Chapter Mapping

Case	Ch01 (Tools)	Ch02 (Quality)	Ch03 (Testing)	Ch04 (Reqs)
Therac-25	-	Dead code	Race condition tests	Safety reqs
Ariane 5	-	Type safety	Boundary testing	Interface reqs
MCO	-	Units in types	Integration tests	Contract spec
Knight	Git/Deploy	Dead code, CI/CD	Smoke tests	-
HCgov	-	CI/CD	E2E testing	NFRs, stakeholders

Every failure maps to concepts we've covered!

The Common Patterns

Pattern 1: Implicit Assumptions

- Therac: "Operators are slow"
- Ariane: "Values fit in 16 bits"
- MCO: "Everyone uses metric"

Pattern 2: Testing Gaps

- Boundary values not tested (Ariane)
- Integration not tested (MCO, HCgov)
- Deployment not tested (Knight)

Pattern 3: Process Failures

- No traceability (all cases)
- Dead code kept (Therac, Knight)

Your Defense: Course Toolkit

Problem	Solution from Course
Implicit assumptions	Design by Contract (Ch04)
Missing boundary tests	Boundary Value Analysis (Ch03)
Type confusion	Pyright + explicit types (Ch02)
Dead code	Ruff linting (Ch02)
Integration failures	CI/CD pipeline (Ch02)
Missing NFRs	Requirements checklist (Ch04)
Unclear ownership	Stakeholder analysis (Ch04)

Final Exercise: Self-Assessment

For your Road Profile Viewer, check:

Item	Yes/No
All function inputs have explicit types?	
Boundary values tested (angle=0, 90, 180, 360)?	
Integration between modules tested?	
Dead/unused code removed?	
CI pipeline runs on every push?	
NFRs documented (performance, accuracy)?	

How many can you check? What's your priority to fix?

Key Takeaways

1. **Failures are rarely "one bug"** - they're systemic process failures
2. **Every course concept prevents real disasters:**
 - Types prevent MCO (Ch02)
 - Boundary tests prevent Ariane (Ch03)
 - CI/CD prevents Knight (Ch02)
 - NFRs prevent HCgov (Ch04)
3. **Tests are executable requirements** - if you can't test it, you don't understand it
4. **Integration is where systems fail** - test modules together, not just alone

References & Further Reading

Primary Sources:

- [Leveson & Turner: Therac-25 Investigation \(IEEE 1993\)](#)
- [ESA: Ariane 501 Inquiry Board Report](#)
- [NASA: Mars Climate Orbiter Mishap Report](#)
- [SEC: Knight Capital Enforcement Order](#)
- [GAO: HealthCare.gov Report \(GAO-14-694\)](#)

Books:

- Nancy Leveson: *Safeware: System Safety and Computers* (1995)
- Robert Glass: *Software Runaways* (1998)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

THE ENGINEERS WHO BUILT
THESE SYSTEMS WERE SMART
AND EXPERIENCED

Your advantage: You know these processes from day one.