Let’s learn from some famous failures of design!
Today’s slides via Jerry Saltzer (MIT)

One of the lead architects behind Multics
Most ideas in modern operating systems were first invented and proven out in Multics (1960’s)

Influential throughout computer systems design

Saltzer gave this as an invited talk at SOSP in 1999
I’ve built today’s slides from those and other sources (including Kaashoek & Morris’s 2009 MIT 6.033)
Why do systems fail?

Complexity has no hard edge

Learning from failures: common problems

Fighting back: avoiding the problems
Too many objectives

Ease of use
Availability
Scalability
Flexibility
Mobility
Security
Networked
Maintainability
Performance
Cost
...

No systematic methods to build systems to meet all of these objectives!
The “tarpit”

Many objectives
+ Few methods
+ High $d(tech)/dt$

= High risk of failure
Complexity: no hard edge

where do you stop?
Today’s goal: learn from failure!

“The concept of failure is central to design process, and it is by thinking in terms of obviating failure that successful designs are achieved...”

[Henry Petroski]

Many books on this topic are available.
“Keep digging” principle

Complex systems fail for complex reasons

- Find the cause
- Find a second cause
- Keep looking
- Find the mind-set

[Petroski]
Classic example: Pharoah Snerfu’s Pyramids

It took them three tries

- Facing fell off
- Internal structural issues
- Hacks to compensate
  (Big wood logs - failed!)
- Third time’s the charm

Try 1: Meidum (52° angle)

Try 2: Dashur/Bent
(52° to 43.5° angle)

Try 3: Red pyramid
(43° angle)
The famous example: Therac-25 (1980’s)

Precision radiation machinery for cancer treatment

Low vs. high-dose modes
- Broad area vs. narrow-area dosing
- Different “shields” placed in front of radiation emitter

Earlier machines had hardware interlocks
- But software is more flexible, more configurable

1985-1987: several patients died of overdoses

Inside the treatment room Cox was hit with a powerful shock. He knew from previous treatments this was not supposed to happen. He tried to get up. Not seeing or hearing him because of the broken communications between the rooms, the technician pushed the “p” key, meaning “proceed.” Cox was hit again. The treatment finally stopped when Cox stumbled to the door of the room and beat it with his fists.
As a result of the Therac-25 accidents, the FDA now requires documentation on software for new medical and other products: a paper trail, in other words, that can be examined by an independent body and retraced for flaws. In January, 1995, the International Electrotechnical Commission will recommend software safety standards for medical equipment, standards developed partly as a result of the Therac-25 accidents. Engineers can find their productivity cut nearly in half by such requirements, and there have been complaints in the high-tech community that software documentation is hampering competitiveness. The University of Washington's Jonathan Jacky still feels it's better than relying on what he calls "the stereotype of the eccentric genius programmer." At least, he told me, "the chances of a hazard getting into the community are a lot less. This run of Therac-25 accidents made it clear how wrong thing could go." At the time of the accidents no educational standard was required of computer-software programmers. "That's still true," says Jacky. "The knowledge of people out there is extremely variable -- some people working on these things are far better than others. That's what documentation on software is supposed to catch."
United Airlines / Univac (late 1960’s)

Goals: Automated reservations, ticketing, flight scheduling, fuel delivery, kitchens, and general administration

Started 1966, target 1968, scrapped 1970

Spent $50M (~$300M in today’s money)

Ambitious over-response to American Airlines’ SABRE, still in use today

The same thing happened with Burroughs & TWA
CONFIRM (late 80’s)

Hilton, Marriott, Budget, American Airlines
Link air, car, and hotel reservations

Started 1988, scrapped 1992, $125M (~$250M in today’s dollars)

Second system
DB integration problems
DB not crash-recoverable
Bad-news diode

[Communications of the ACM 1994]
FAA’s Advanced Automation System (1980’s)

Federal Aviation Administration: wanted to replace a 1972 system
Real-time nation-wide route planning

Started 1982, scrapped 1994 ($6B, roughly 2-3x more in today’s dollars)

**Big ambitions**
Changing ideas about user interfaces
12 years of evolving requirements, technologies
12 years of a culture of not finishing
Big dollars, lots of Congressional “involvement”
London Ambulance Service (1991-2)

Ambulance dispatching (“911” services)
Started 1991, scrapped 1992
20 lives lost in 2 days!

No testing/overlap with the old system
Required big changes in procedure
Users not consulted during design
Unrealistic schedule (5 months)
Inexperienced low-cost bidder

What is clear from the Inquiry Team’s investigations is that neither the Computer Aided Despatch (CAD) system itself, nor its users, were ready for full implementation on 26 October 1992. The CAD software was not complete, not properly tuned, and not fully tested. The resilience of the hardware under a full load had not been tested. The fall back option to the second file server had certainly not been tested. There were outstanding problems with data transmission to and from the mobile data terminals. There was some scepticism over the accuracy record of the Automatic Vehicle Location System (AVLS). Staff, both within Central Ambulance Control (CAC) and ambulance crews, had no confidence in the system and were not all fully trained.

The physical changes to the layout of the control room on 26 October 1992 meant that CAC staff were working in unfamiliar positions, without paper backup, and were less able to work with colleagues with whom they had jointly solved problems before. There had been no attempt to foresee fully the effect of inaccurate or incomplete data available to the system (late status reporting/vehicle locations etc.). These imperfections led to an increase in the number of exception messages that would have to be dealt with and which in turn would lead to more call backs and enquiries. In particular the decision on that day to use only the computer generated resource allocations (which were proven to be less than 100% reliable) was a high risk move.
IBM Workplace OS (1991-6)

One microkernel operating system for all IBM products
PDAs, desktops, servers, supercomputers
“Personalities” for OS/2, AIX, OS/400, Windows
  Binary compatibility / translation
  Tried to hack Windows to use external OS services
Support for x86, PowerPC, ARM

Started in 1991, scrapped in 1996 ($2B, >$3B today)

Factoring out common services is hard

PowerPC chip (new hardware) was late, buggy, slow
New software needed new hardware (for speed)
New hardware needed new software (for compatibility)

Intra-IBM personality conflicts, cooperation problems
[Fleisch HotOS 1997]
IBM Workplace OS (1991-6)

One microkernel operating system for all IBM products
PDAs, desktops, servers, supercomputers
“Personalities” for OS/2, AIX, OS/400, Windows
  Binary compatibility / translation
  Tried to hack Windows to use external OS services
Support for x86, PowerPC, ARM

Started in 1991, scrapped in 1996 ($2B, >$3B today)
Factoring out common services is hard
PowerPC chip (new hardware) was late
New software needed new hardware
New hardware needed new software

Intra-IBM personality conflicts, cooperation problems
[Fleisch HotOS 1997]

Irony: IBM was doing “virtual machines” decades earlier than anybody else, but they didn’t pursue it here, where it might have made sense.
Many, many more

Portland, Oregon, Water Bureau, $30M, 2002
Washington D.C., Payroll system, $34M 2002
Southwick air traffic control system $1.6B 2002
Sobey’s grocery inventory, $50M, 2002
King’s County financial mgmt system, $38M, 2000
Australian submarine control system, $100M, 1999
California lottery system, $52M
Hamburg police computer system, $70M, 1998
Kuala Lumpur total airport management system, $200M, 1998
UK Dept. of Employment tracking, $72M, 1994
Bank of America Masternet accounting system, $83M, 1988
FBI Sentinel case management software, 2006
Recurring themes
Excessive generality / ambition

“Biting off more than they can chew”
Versus starting small and having an incremental plan

American’s SABRE did one thing, reservations, and it worked well
Other systems tried to do everything rather than incremental things

Related: “Second-system effect”
Version 1 was simple
Version 2 was too ambitious
Famous example: the “Osborne effect”
Pre-announced Osborne 2, customers stopped buying the Osborne 1. Company died.
Bad modularity

One developer makes one change, and everything else breaks
You’ve seen this if you take a bad yellow-bulb suggestion from IntelliJ.

Arguably, the single greatest triumph of UNIX (and of Microsoft COM) is encouraging small, simple programs / modules with limited interfaces between them.

If you don’t know your way around the UNIX command-line, you’re missing out.

% find src/edu/rice -name '*.java' -print | grep -v Test | xargs wc -l

70 src/edu/rice/io/Files.java
200 src/edu/rice/json/Builder.java
1052 src/edu/rice/json/Parser.java
61 src/edu/rice/json/Scanner.java
295 src/edu/rice/json/Utils.java
350 src/edu/rice/list/IList.java
81 src/edu/rice/list/IQueue.java

...
Inexperience (or ignoring good advice)

Not everybody recognizes their limitations
Limited skills vs. strong ambitions

Also includes recognizing when your tools are inadequate
Not every task should be solved with the same tools.
Not every design that works one on one machine will scale to a million machines.

There are known knowns; there are things we know that we know.

There are known unknowns; that is to say, there are things that we now know we don’t know.

But there are also unknown unknowns – there are things we do not know we don’t know.

-Donald Rumsfeld
The “bad-news diode”

Making promises you can’t keep
Companies claiming they’ll make deadlines, when they know they won’t
Engineers claiming they’ll make deadlines, when they know they won’t
Bad news doesn’t tend to flow upward in organizations (= embarrassment)

And sometimes management doesn’t want to know!

This is a common theme in dysfunctional organizations.
Brooks’ Law: Adding more engineers to a late software project makes it later.
Fighting Back!
Fighting back: Control novelty

Only one big new idea at a time
Reuse existing components
But it’s hard to say “no”
Second-system effect
Technology is better
Ideas work just fine in isolation
Market pressure

Requires strong, knowledgeable management
Fighting back: Simplify!

You don’t need ten ways to do any given thing

In the small: mixing and matching paradigms can be a mess
There’s a reason we’re forcing you to stick with functional programming in Comp215. (And you’re running into a lot of friction where new Java8 code and older Java code collide.)

In the large: the same problems happen
There are serious benefits to having one database abstraction, one networking abstraction, etc., then add compatible layers only when necessary.

Good aesthetics and abstractions yield more successful systems
Parsimonious (minimal), orthogonal, elegant, readable, ...
Fighting back: Design iteration

Design to iterate quickly, and iterate your design

Get something simple working soon
Find out what the real problems are

Get customer feedback, build it into the dev schedule
Deploy in phases, interact with the customer often

“Every successful complex system is found to have evolved from a successful simple system.” - John Gall
Fighting back: Find bad ideas fast

Question the requirements
Example: F35 required to support three very different configurations (land, navy carrier, VTOL) vs. A10 - a singular plane built around a very big gun

Try ideas out, but don’t hesitate to scrap them
Build this into the engineering timeline!
Fighting back: Find bad ideas fast

Question the requirements
Example: F35 required to support three very different configurations (land, navy, VTOL) vs. A10 - a singular plane built around a very big gun

Try ideas out, but don't hesitate to scrap them
Build this into the engineering timeline!

Test Pilot Admits the F-35 Can’t Dogfight
New stealth fighter is dead meat in an air battle
Find bugs early

Design reviews, coding reviews, regression tests, build tests, performance measurements, alpha/beta releases, etc.

Incentives, not penalties, for reporting errors!
Summary

Follow principles that can help avoid failure
Limit novelty
Simplify
Get something working soon
Iteratively add capability
Incentivize reporting errors
Jettison bad requirements early
Use good tools
...

Strong outside pressures push you to violate these principles
Good management is just as important as good engineering