SOSP lecture notes (Saltzer, December 14, 1999)

SLIDE 1—title & speaker

Intro: My name is Jerry Saltzer

This is the first SOSP I have attended since retiring five years ago, and there are a lot of new faces. I used to be more involved--I was somewhere on program of each of the first ten SOSP's. But recently I've been working more with the digital library folks, who provide a nice application to try systems ideas out on. It is fun to be back.

Most of you have undoubtedly already read Lampson's Hints paper and Brooks's book, know everything there is to say about controlling complexity, and will just be either nodding in agreement or shaking heads in violent disagreement with what I am going to say. I hope, though, that maybe a couple of these ideas will strike a few people here as useful.

SLIDE 2—plan

Coping With Complexity

Everyone knows the standard techniques of coping:

- **Modularity**
- **Abstraction**
- **Hierarchy**
- **Layered design**
- **Watch for incommensurate scaling**

So I won't talk about them very much today.
Instead, I want to talk about why these things don't always work,
and what one can do about it.

- **Sources Of Complexity**
- **Learning From Disaster** and success
- **Fighting Back**
- **Admonition**

Our fundamental problem: we have a mismatch.

SLIDE 3—our system

1. Too many objectives and ideas

2. Too few principles

+ end–to–end arguments
+ economy of mechanism
+ open design

but when you come right down to it,

- we don't really know how to divide systems up
- we use ad hoc solutions (naming, fault-tolerance)
- we have no way to compare different designs (every system is different)
- we have no way to predict how the user will adapt to our design
- we have no way to assure correctness (we usually don't know what it means)

3. On top of that we have a unique Amplifier and Accelerator: High D(technology)/Dt

SLIDE 4—the tar pit

Result: The tar pit is never far below us. (If you haven't run aross this piece of imagery from Fred Brooks, here it is: The Sabre-toothed tiger can, by struggling, get any one paw out, but the others just sink in deeper.)

- No ultimate barrier———it just gets worse

SLIDE 5—no barrier

cause: interactions and conflicts among seemingly unrelated functions

e.g. security (minimize extra copies, use encryption)
fault-tolerance (make extra copies, don't lose keys)
naming (need to trust it for security, need to make it fault-tolerant, too)
performance (don't encrypt, use numbers instead of names)

Complexity is relative to understanding--it may yield to insight & ingenuity, or it may not.

Unfortunately, we can't depend on breakthroughs in understanding, so we must try something else while waiting.

The "something else" is: Learn from Failure

SLIDE6—Failure/pyramids

A long tradition in civil engineering systems: bridges, buildings, etc. When they fail, engineers study them to find out why.

The Pyramids provide a nice example of the length of the tradition:

1. **Dating: Over time, they got larger (psychology at work)**
2. **A plot of height versus time shows one gap with no pyramids**
3. **In the middle of the gap are some large piles of rocks**
4. **There is a sudden change (lowering) of the height/width ratio as we pass the gap**

5 One pyramid, at the end of the gap, changes that ratio in the middle

Learn from failure:

SLIDE7—deeper learning

Shipwrecks: They happen frequently; naval architects study them carefully

Hancock building: simulating 8 wind directions wasn't enough
(in between, the stresses were 100% higher)

Aircraft systems: NTSB tries to get to the bottom of every accident; pilots are encouraged to report problems without penalty.

Method: Study/analyze/go beyond the first explanation, to look for *all* the contributing causes, including the mind set that allowed the failure mechanism to be overlooked

Reason: A complex system usually fails for complex reasons; a working system often works for reasons different from those originally thought.

Mars lander: units mismatch? That happens all the time. The real problem was omitted cross—checks that should have caught the error.

Our problem: *Learning from failure is not done nearly enough in computer systems*

(Next: disaster slides)

Disasters

SLIDES 8-18—disasters

1. NYC 10K lights (incommensurate scaling, too many ideas)

2. California DMV [Wall Street Journal April 27, 1994: The California legislature approves spending $500,000 to investigate what went wrong...San Jose Mercury News April 27, 1994: The system conversion proved more challenging than originally anticipated and was ultimately abandoned by the DMV. super-database, 2nd system effect)

3. UAL/UNIVAC (too many objectives, 2nd system)

4. CONFIRM (dull tools (machine language), 2nd system, extensive use of the bad news diode.

5. ALS (2nd system) [N.B.: examples are in order by cost, but not inflation–adjusted.]

6. SACSS [San Francisco Chronicle, 2 May 1997, p. A22. "Due to significant problems in the SACSS application, it is unclear whether the project will ever fulfill the mandate of the federal government or the child support enforcement needs of California's 58 counties."]

7. Taurus (Ross Anderson, 12 Mar 93: It didn't work and a review showed that there was no reasonable prospect of it working; it seems that it just got too complex to cope with. Roger Needham: such fiascos can be compared to the civil engineering disasters of the nineteenth century such as the collapse of the Tay bridge. Civil engineers eventually got their act together, but

there was a long learning process in which they worked out how to structure their approach to large problems and combine the maths with the project management in a way that worked. President of stock exchange lost his job)

8. IBM Workplace OS for the PowerPC: Kernel plus "personalities" for Taligent (Mac) Pink, OS/2, DOS, AIX, and AS/400 was just too complicated. Ex: Each architecture had its own memory management, some byte-oriented, some page-oriented; common memory manager was too complicated. The processor was buggy. DOS was little-endian, AIX was big-endian; no solution was found. They used C++ and started by defining the classes, but first defs weren't quite right. Then they discovered that changing defs is hard. (It is hard to find the right modularity) 400 staff (Mythical Man Month) See quote on slide. At the end, an entire IBM division was scrapped, along with the Boca Raton site.

9. IRS (replace everything at once—too many objectives)

10. AAS (2nd system, display replacement may be sufficient--but now that is in trouble, too.)

11. London Ambulance (only 20% of calls getting through, many ambulances were sent to the same call. "No-one seemed to know who the Project Manager was, indeed from the interviews, several people each gave a different name for the manager and some people had no name to offer whatsoever.") In 1994 a new group undertook the project using good methods, and in 1996 went on-line successfully.

Standish Group (Cape Cod)

SLIDE19—Standish

1995 study of 8,000 examples only 20% of computer systems projects are "successful". Of the 50% that are "challenged" they are delivered with average 2X budget, 2X completion time, and 2/3 the planned functions(whatever that means.)

Current events

These are all history. Maybe by now people have learned how to do better. So we open the newspaper and what do we find?

Wall Street Journal CCXXXIII, 81 (Eastern Edition), Tuesday, April 27, 1999, page A8. "New Air-Traffic System to be delayed, FAA Says" "An interim version of the system functioned at a substantially slower speed than the current system. December (Boston) -> April (Boston) -> summer (some less busy area)" (This is the Display-Only replacement for the Advanced Automation System scrapped in 1994.)

The Economist 351, 8115 (April 17th 1999) page 62 "The mice bite back". The introduction of a new $125M computer system at the Immigration and Nationality Directorate has led to such delays in processing asylum applications that there are reports of rats gnawing away at vital documents....thousands of lost files...60,000 applications behind..." "...the new air–traffic–control system for London...four years late and 75% over budget...Post Office benefits-card system 2 years behind schedule...modernization of national insurance system...160,000 pensioners underpaid...more than 1M unprocessed jobless benefit claims...17M national insurance records waiting to be processed...these and other recent computer blunders will cost (British) taxpayers more than 1B pounds.

Fall 1999: Hershey missed Halloween, Whirlpool can't deliver dryers.

Review the recurring problems

SLIDE 20—recurring probs

- **Incommensurate scaling (traffic lights, Taurus)**
- **Too many new ideas (second–system effect)**
  (traffic lights, UAL, TWA, ALS, CONFIRM, Workplace OS, AAS)
- **The bad-news diode (Therac, Confirm)**
- **The mythical man-month (Workplace OS)**
- **Modularity is harder than it looks (Workplace OS)**

SLIDE21—not enough

You begin to get the picture that there may be more to building systems than applying Modularity, Abstraction, Hierarchy, and Level definition. Why aren't these techniques enough?

--> They all assume that you understand the problem.

--> It is easy to create abstractions, but much harder to create the RIGHT abstractions.

--> It is easy to stack your abstractions up in layers, but hard to decide that you have the RIGHT layers.

--> Modularity and hierarchy have the same problem: you have to understand what you are doing.

Fighting back: Control the amount of novelty

SLIDE 22—control novelty

Successful complex systems evolve from successful simple systems. They are rarely built from scratch. Airplanes and bridges are almost always small steps from previous airplanes and bridges. But computer systems tend to take big steps. A new system always has many more objectives than previous one.

Why?

- **the first system gave a taste of how good it COULD be**
- **First system was known to be challenging, so it was spartan. Second system gets all the omitted things.**
- **Hardware costs have decreased dramatically (allows appetite to grow)**
- **Each idea looks feasible in isolation**
- **Very hard to judge how complexity grows with function**

Result: Too many new ideas to digest smoothly

SLIDE 23—it just gets worse

Ex: Windows 2000, Word 98

How to control novelty?

SLIDE 24—novelty

Something Simple working soon

Either

- A simple system

- A small change to an existing system(only one new idea)

Why?

- allows design iteration

- checks assumptions

- gets early feedback

- helps calibrate timetable

Example: Hyatt Reservation system

- use large, known–to–work components (Tuxedo, UNIX, Informix)
came in ahead of time, under $15M budget,
with extra features

Another example: PowerPC 604 (6 x 10^6 gates)--small step from previously working PowerPC 601

- lots of repetition: large Cache (identical memory cells)
(6-way)_instruction issue = 6 identical ALU's

Clearing House for Interbank Payments System (CHIPS)

reconciles $1T/day of transactions among 120 large U.S. banks
(Small step from previous system)

Basic conclusion: You won't get it right, so make sure you can change it.

SLIDE 25—feedback

Feedback: Design for iteration, then
Iterate the design

Add one new problem at a time
Find ways to find bad ideas and flaws early [more on this later]
Use iteration-friendly design [more on this later]
Act on the feedback--where are the design assumptions wrong?

Brooks version of this observation:

SLIDE 25—Brooks

Rationalism vs Empiricism…

On the Rationalism side we have Aristotle, Descartes, and Dijkstra

While on the Empiricism side we have Galileo, Hume, and Knuth

This is also the distinction between *proving* programs and *testing* programs

Good system builders are usually empiricists

SLIDE 26—find bad ideas

Any list of system objectives usually contains some bad ideas. Why?

- **Faster/cheaper hardware makes additional function look tempting**
- **Ideas workable in isolation sometimes don't work well in conjunction.**
- **Hard to identify bad ideas without trying them**
- **Imposed objectives from multiple sources**
- **Local optimization by aggressive designers**

Result: must be prepared to scrap out functions; requires strong, knowledgeable management.

SLIDE 27—design loop

The design loop: turn-around time after a flaw is discovered:

Why *months* after production starts?

1. Not easy to detect problems (e.g., 25% performance loss, or 1 error/1000 hrs.)

2. Not easy to discover source

3. Flaw is deeply embedded

4. Original designer is cold on the problem (or has left to go to work for Microsoft.)

SLIDE 28—find flaws early

Invent ways to discover flaws early

- **Something simple working soon (Hyatt Res. System)**
- **alpha test, beta test**
- **simulate, simulate, simulate**
  Boeing 777 simulated everything: the design, the operation, the manufacturing steps
  PowerPC 604
  F16 Aircraft flip-over at equator found by simulation
- **plan, plan, plan (CHIPS)**
  start 1986 5.5 years of planning & developing tests
  install 1992 0.5 years of implementation

- **design reviews, coding reviews, regression tests (every previously found bug), performance meters**
- **Most important: The feedback scheme is part of the system design**
  Ross Anderson, on British ATM laws: The wrong legal system can inhibit feedback and iteration.

Understand the bad-news diode: Good news flows up and down a hierarchy. Bad news does not flow up. Upper management is the last to know.

Use Iteration-Friendly design methods

SLIDE 29—iteration-friendly

Authentication logic
Space shuttle alibis
Fault tolerance tolerable/intolerable classification

General approach

- **document the assumptions as carefully as possible**
- **make sure there are feedback paths**
- **when the feedback arrives (during design or a report of failure): review assumptions for omissions and errors**

Provides a way to systematically converge on a working, successful system

SLIDE 30—concept integrity

Conceptual integrity

Brooks: Separate architecture from implementation

Saltzer: Hire a dictator: One mind controls the design

Superdesigners make good systems:

Reims cathedral (8 generations of implementers!)
Macintosh: Russ Atkinson and Steve Jobs
Visicalc (spread sheet): Bob Frankston
X Windows: Bob Scheifler & Jim Gettys
SunOS: Bill Joy

Good Esthetics --> more successful systems (stolen from Brooks)

Obstacles: Modularity is harder than it looks

SLIDE 31—obstacles

Hard to get the right boundaries between modules

Requires keen judgement to
do it right, Function here? there?
keep cost reasonable, Two modules or three?
not compromise function

Initial boundaries *will* be wrong --> Then what?

Committed components and Committed designers.
Implementers inside system can't see it, but have developed a stake in the wrong design.

Management must be good enough to realize and force change
must be technically competent. C/W designing toasters-- "management must motivate people" "a good manager can manage anything." no technical competence, rapid turnover (2 yrs to next post.)

Development technology must allow change. Class libraries may be problematic; type defs propagate. (Ex. IBM workplace OS)

Management is challenging:

High complexity means...

Must use very good designers, but ...

They are Prima Donnas, so ...

Need even better managers

“Good programmers are creative. Even in situations that do not call for creativity.”

Summary: Fighting Back

SLIDE 32—summary

End with a story of success:

London Ambulance Service was replaced with the Central Ambulance Control System
- **different team**
- **less novelty, lower ambitions**
- **much user input and feedback**
- **gradual deployment with feedback**
- **flexible target dates**

***Success*, Jan. '96**

Admonition

SLIDE 33—admonition

Be very careful. Make sure that one of your system designs doesn't turn up as a disaster example in a future version of this lecture...