What 6.033 teaches us about software engineering

Michael Ernst

May 13, 2002

6.033: coping with complexity

- modularity
- hierarchy
- client-server
- layers
- virtualization
- coordinating sharing
- naming
- authentication, confidentiality
- redundancy
- transactions

6.170: Coping with complexity

- abstraction: information hiding
- modularity: separation of concerns
- specification: expressing (only) non-hidden details
- experience (sort of)

Outline

Why do systems fail?
How to avoid failure
Planning, failure, and iteration
Conclusion

The single most important factor for a successful system

- requirements
- design
- management
- scheduling
- implementation
- documentation: internal and external
- testing: validation and verification
- deployment
- maintenance

Brooks's "tar pit"

- You can pull any one paw out of the tar

Complex failures

People and systems are amazingly resilient

They can tolerate single failures very well

Complex systems usually fail for complex reasons

- each individual reason might be simple

Accidents in North American Mountaineering 1992

American Alpine Club and Alpine Club of Canada

Cornice collapse: unroped, inattention

Fall on waterfall ice: unroped, exceeding abilities, haste

Fall on rock: protection failure, inadequate protection

Stranded: exceeding abilities, bad weather

Fall on rock: inadequate protection, exceeding abilities

Fall on rock: placed no protection (leader), inadequate protection (belayer), exceeding abilities, off route, inexperience

Slip on snow: unroped, inadequate equipment, exceeding abilities, inexperience

Fall on rock: placed no protection, exceeding abilities

Fall on rock: climbing unroped, inadequate equipment, exceeding abilities, inexperience

Outline

Why do systems fail?

How to avoid failure

- Avoid complex failures
- Apply 6.033's principles (appropriately)
- Avoid the seven deadly sins of system building

Planning, failure, and iteration

Conclusion

Avoid complex failures

Simplify

- Remove components that may fail (avoid special-case code)
- Make the system easier to understand
- Abstract

Test

- Discover unknown failures
- Fix them to regain cushion against system failure

Apply the ideas of 6.033 appropriately

Example: End-to-end principle

- subcomponents must work pretty well
- retries must be possible

Apply to Therac-25 radiation therapy machine

- most of the time, it works
- keep sending packets (patients)
- ignore any that get dropped

Avoid the seven deadly sins of a system builder

These are lessons of 6.033, 6.170, and the school of hard knocks.

Understand the goal (lust)

Carefully determine and specify requirements

- definitions may be the most important part

Understand the threat model

Abstract judiciously during this step

Achieve conceptual integrity (gluttony)

KISS: Keep It Simple, Stupid!

Full system may be necessarily complex

- Real-world requirements
- Non-ideal components
- Maybe the complexity is unnecessary after all

Ensure there is a conceptually simple core

Be humble (pride)

Estimate accurately

Brooks: lack of time is the key impediment to successful system construction

Software is harder to estimate

- little repetition of previous systems (this is good!)

Know your limits and those of your technology

- avoid the second system effect

Be disciplined (sloth)

Software is (too) malleable

- it seems temptingly easy to change

Programmers need discipline.

- documentation, testing, etc.

On small projects, anything works!

- discipline still works best

Use good tools as well as good process

Communicate (anger)

Software engineering is about communication: with the machine, users, colleagues, yourself

Why is the "mythical man-month" mythical?

Communication is greatly eased by modularity, abstraction, and specification

Don't over-optimize (greed)

Early optimization has uncertain benefits

Optimization has certain costs:

- increased complexity
- increased likelihood of errors
- loss of conceptual integrity

Humans are the scarce resource: optimize that

Maintain effectively (envy)

"Maintenance" is a misnomer
- fix defects
- adapt to changing environment
- adapt to changing requirements

Documentation matters

Debugging matters

Outline

Why do systems fail?

How to avoid failure

Planning, failure, and iteration

Conclusion

Your first try will be wrong

Requirements are unknowable a priori

You don't understand the details

Neither does the customer

Working through the details helps to clarify (and change) one's understanding

Thus, details will change

Fully detailed upfront design will make mistakes

Throwing one away (or not)

Brooks 1975: "Throw one away"

Brooks 1995:
- incorporate feedback in all parts of the process
- iterate back to the appropriate point
- always have a running system
- use an "agile process"

(See chapter 19. Brooks also changed his mind about information hiding.)

Extreme Programming (XP)

- Customer onsite
- Test first (optimize last)
- Pair programming
- Frequent integration
- Refactoring
- No planning for the future
- Measure progress

Works well on medium-size projects.

Outline

Why do systems fail?

How to avoid failure

Planning, failure, and iteration

Conclusion

Why do you care?

Course VI graduates, 6.033 and 6.170 are the most valuable subjects they took at MIT.

Both are about coping with complexity via

- principles
- abstractions
- techniques
- tools

What to do next

Remember these lessons

Be skeptical (but justify your skepticism)

Build good systems

Have fun!