TOP FIVE CHALLENGES IN SOFTWARE DEVELOPMENT

www.construx.com

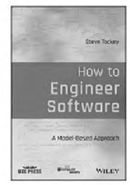
Construx®

COPYRIGHT NOTICE

Construx®

About Steve Tockey

- First code Oct, 1975. First paid code Jun, 1977
- Education
  - Bachelor of Computer Science (L&S), UC Berkeley, 1981
  - Master of Software Engineering, Seattle University, 1993
- Employment
  - HSS ('77), LLNL ('84), Boeing ('87) , Rockwell Collins ('96), Construx ('98)
  - Adjunct professor, Seattle U MSE ('94-'96, '99-'07)
- Publications
  - Over 20 technical papers, articles
  - Return on Software, Addison Wesley, 2005
  - How to Engineer Software, Wiley / IEEE Press, 2019
  - Chapter editor for three KAs in IEEE-CS SWEBOK Guide v4
- Professional volunteer
  - IEEE-CS Certification Committee chair
  - Conference paper referee, e.g., CSEE&T
- Hobbies
  - Travel, foodie, ancient computers (pdp-8, pdp-10, pdp-11, IMSAI 8080)



Outline

- Product success vs. project success
- Typical software project outcomes
- Top five root causes of poor performance
  - Get well plan

Product Success vs. Project Success

- Successful software project ...
  - is on-time
  - is within budget
  - delivers all agreed-on functionality
  - has appropriate quality
- And …
  - team is stronger

Reference: [DeMarco97]

Typical Software Project Outcomes

- 18% of projects fail to deliver any usable software
- Of projects that do deliver, average
  - 42% late
  - 35% over budget
  - 25% under scope
  - Abundance of delivered defects
- 2019 US software budget ~$340 billion
  - ~$61 billion in cancellations
  - ~$72 billion in cost overruns
  - ~$41 billion in scope under-runs
  - Funders expected to pay only ~$166 billion for functionality actually delivered!

Reference: [Standish13]

Top Five Root Causes of Poor Performance

- Data supports
  - (1) Vague, ambiguous, incomplete requirements
  - (2) Inadequate project management
- Professional experience suggests
  - (3) Uncontrolled design, code complexity
  - (4) Over-dependence on testing
  - (5) "Self-documenting code" is myth

(1) Vague, Ambiguous, Incomplete Requirements

- Direct result of using natural language
  - Built in ambiguity
    - Different words often have same meaning
    - Same word often has different meanings
  - Verbose-ness
    - Need too many words to provide sufficient precision

(1) Vague, Ambiguous, Incomplete Requirements

Built In Ambiguity in Natural Languages

- "*Youths steal funds for charity*"
  - (Reporter Dispatch, White Plains, NY, February 17, 1982)
- "*Large church plans collapse*"
  - (Spectator, Hamilton, Ontario, June 8, 1985)
- "*Police discover crack in Australia*"
  - (International Herald Tribune, September 10, 1986)
- "*Sisters reunited after 18 years in checkout line at supermarket*"
  - (Arkansas Democrat, September 29, 1983)
- "*Air Force considers dropping some new weapons*"
  - (New Orleans Times-Picayune, May 22, 1983)

Reference: [Cooper87]

(1) Vague, Ambiguous, Incomplete Requirements

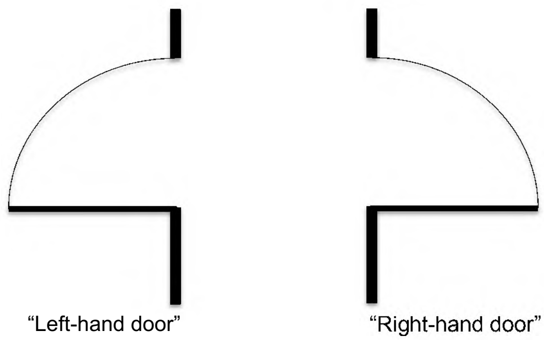
Software Requirements Ambiguity

"The system shall detect a ¼ inch defect in a pipe section"

(1) Vague, Ambiguous, Incomplete Requirements

Verbose-ness in Natural Languages

*"The main floor guest bathroom shall have a door.
That door shall be a right-hand door.
That right-hand door shall be oriented so the
hinges are on the South side of the door frame."*



(1) Vague, Ambiguous, Incomplete Requirements (cont)

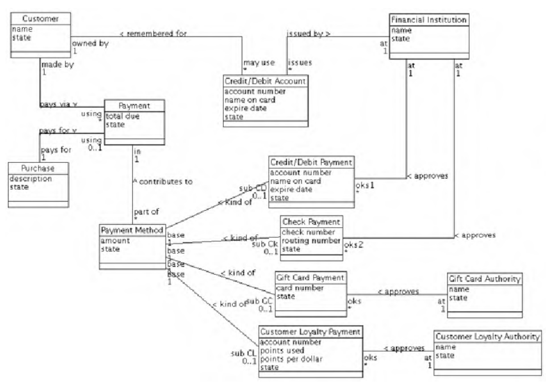
The people who design and build houses gave up trying to describe them in natural language over one hundred years ago. What makes you think you can successfully describe something that's orders of magnitude more complex using natural language?

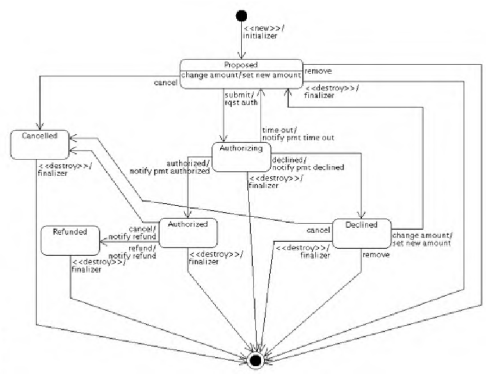
Most requirements aren't changing, they are only being clarified.

(1) Vague, Ambiguous, Incomplete Requirements

Get Well Plan

- Acceptance test-driven development / Behavior-driven development
  - With functional coverage criteria
- (Semi-) formal requirements specification languages





(2) Inadequate Project Management

- Insufficient goal alignment (e.g., constantly changing priorities)
- Inadequate planning (incl. poor work decomposition)
- Overly optimistic estimation
- Failure to acknowledge inherent uncertainty
- Lack of active risk management
- Inadequate tracking (incl. weak supplier / subcontractor oversight)
- Insufficient change control
- Noisy, crowded facilities
- …

(2) Inadequate Project Management

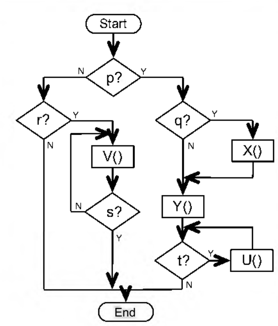
Get Well Plan

- Prioritize based on economic business case
  - Applies to change requests, too
- Explicitly charter projects
  - Authority, agent, completion criteria, resources, constraints, priorities, assumptions
- Actively manage project risks
  - Identify, analyze, prioritize, control
- Use planning template(s), work patterns, checklists, definition of done
- Depend on better estimation practices, particularly collection, use of historical data
  - ~~Expert judgment~~, Analogy, Decomposition, Statistical
- Allow for inherent uncertainty (Cone of Uncertainty)
- Track project status objectively (e.g., peer review, earned value, definition of done, velocity-based sprint planning, burndown)
- Pay attention to Peopleware ([DeMarco99])
- …

(3) Uncontrolled Design and Code Complexity

- Structural (syntactic) complexity
  - Cyclomatic complexity
  - Depth of decision nesting
  - Number of parameters
  - Fan out
  - …
- Semantic complexity
  - Poor abstraction
  - Weak or non-existent encapsulation
  - Low cohesion, high coupling
  - Reactive, not proactive, product family development
  - …

```
if p
  then
    if q
      then X()
    Y()
    while t
      do U()
  else
    if r
      then
        repeat
          V()
        until s
```



See, for example: [Tockey19]

(3) Uncontrolled Design and Code Complexity

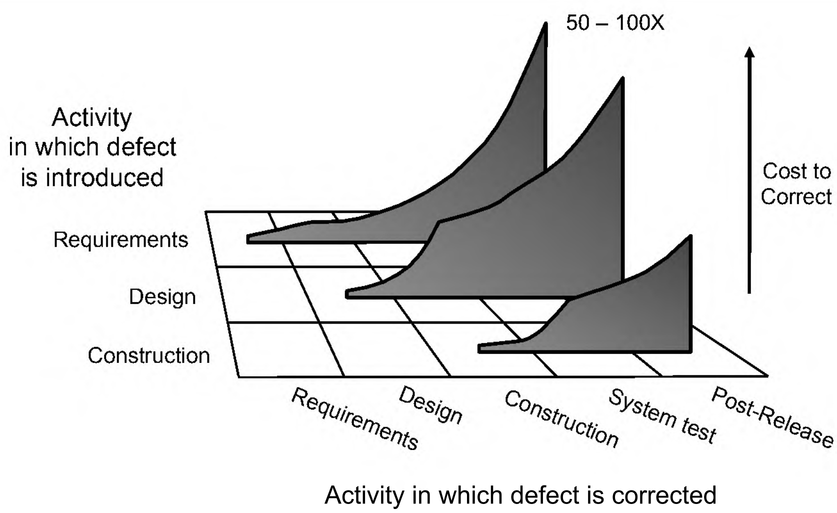
Get Well Plan

- Measure, control structural (syntactic) complexity

	Green	Yellow	Red
Cyclomatic complexity	1 .. 9	10 .. 14	15+
Depth of decision nesting	1 .. 4	5 .. 6	7+
Number of parameters	0 .. 4	5 .. 6	7+
Fan out	0 .. 7	8 .. 10	11+

- Pay attention to fundamental design principles
  - Abstraction, encapsulation (Design by Contract™), high cohesion, loose coupling, proactive product family development

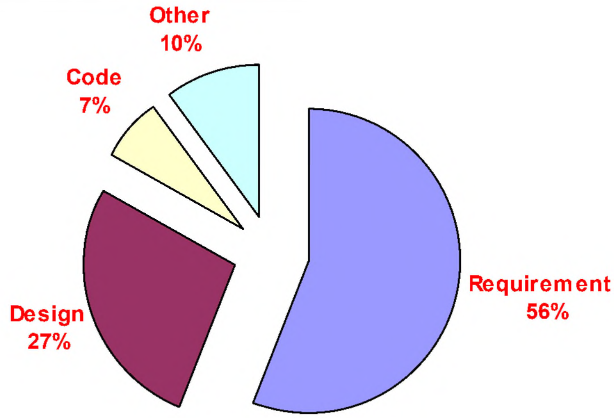
(4) Over-Dependence on Testing



Reference: [McConnell98]

(4) Over-Dependence on Testing

Frequency of Defects



~83% of defects exist before that code is written

Reference: [Mogyorodi03]

(4) Over-Dependence on Testing

Rework Percentage (R%)

$$R\% = \frac{\text{Project effort spent on rework}}{\text{Total effort spent on project}}$$

Size (developers)	Measured R%
350	57%
50	59
125	63
100	65
150	67

"Rework is not only the single largest driver of cost and schedule on a typical software project; it is bigger than all other drivers combined!"

See: Construx "How Healthy is Your Software Process?" white paper

(4) Over-Dependence on Testing

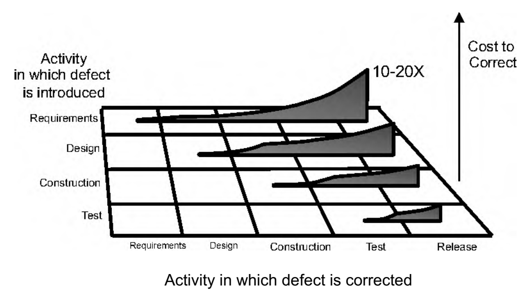
Strategies to Reduce R%

- Find, fix defects earlier
  - Model-based development
  - Acceptance test-driven development
    Behavior-driven development
  - UI Prototyping
  - Collaborative work
  - Peer review
  - Early QA involvement
  - Frequent integration
  - …

(4) Over-Dependence on Testing

Strategies to Reduce R% (cont)

- Reduce defect cost growth rate
  - Model-based development
  - Control design, code complexity
  - …
- Avoid defects
  - Model-based development
  - ATDD / BDD with functional coverage
  - Standards, templates, checklists
  - …



(4) Over-Dependence on Testing

Defects Are Not Only About Product Quality

> *"An engineer wants their system to be fit for purpose and chooses methods, tools and components that are expected to achieve fitness for purpose. It's poor engineering to have a system fail in testing, partly because that puts the budget and schedule at risk but mainly because it reveals that the chosen methods, tools or components have not delivered a system of the required quality, and that raises questions about the quality of the development processes."*
>
> —Martyn Thomas

> *"The real value of tests is not that they detect [defects] in the code, but that they detect inadequacies in the methods, concentration, and skills of those who design and produce the code."*
>
> —C. A. R (Tony) Hoare (paraphrased)

(5) "Self-Documenting Code" is a Myth

- What is this code intended to do?
- Why does this code look the way it does?
  - Has to be vs. happens to be

New development 20%	Maintain existing code 80%

(5) "Self-documenting Code" is a Myth

Get Well Plan

- Focus on making documentation value-added
  - Don't document for development's sake, document for maintenance's sake
    - Document requirements to communicate intent
    - Document design to communicate why, much more than how

> *"Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do"*
> —Donald Knuth

Reference: [Knuth92]

SUMMARY

Construx®

Key Points

- Product success != project success
- Typical software projects perform quite poorly
  - Failed, late, over-budget, under-scope, abundant defects
  - Financial implications staggering
- Top five root causes of poor performance
  - (1) Vague, ambiguous, incomplete requirements
  - (2) Inadequate project management
  - (3) Uncontrolled design, code complexity
  - (4) Over-dependence on testing
  - (5) "Self-documenting code" is myth
- Can address each of top five root causes

References

- [Cooper87] Gloria Cooper, *Red Tape Holds Up New Bridge*, Perigee Books, 1987
- [DeMarco97] Tom DeMarco, *The Deadline*, Dorset House, 1997
- [DeMarco99] Tom DeMarco and Tim Lister, *Peopleware*, 10th Anniversary Edition, Yourdon Press, 1999
- [Knuth92] Donald E. Knuth, *Literate Programming*, Center for the Study of Language and Information, Leyland Stanford Junior University, 1992
- [McConnell98] Steve McConnell, *Software Project Survival Guide*, Microsoft Press, 1998
- [Mogyorodi03] Gary Mogyorodi, "What Is Requirements-Based Testing?", *Crosstalk*, March 2003
- [Standish13] The Standish Group, *CHAOS Manifesto*, The Standish Group, West Yarmouth, MA, 2013
- [Tockey19] Steve Tockey, *How to Engineer Software: A Model-Based Approach*, Wiley – IEEE Press, 2019

Construx

www.construx.com

We believe that each and every software team can be successful.

We believe that developing the professional skills of organizations, teams, and individuals is the best way to make software projects more successful.

For information about our professional development, consulting, and training services, contact stevet@construx.com or hello@construx.com