The Design of Design, Part Two

CSCI 5828: Foundations of Software Engineering
Lecture 27 — 12/02/2014

Goals

- Cover material from two chapters of Parts II and III of Fred Brooks's The Design of Design
  - Collaboration in Design
  - Rationalism versus Empiricism in Design
- The other chapters touch on topics such as
  - Identifying the budgeted resources of a project
  - Why constraints are friends in design
    - Think Ruby on Rails => out of the box provides 80% of what you need for creating database-backed websites
  - The use of exemplars in design

Collaboration in Design

- In Part 2 of the book, Brooks examines issues related to collaboration in the design process
  - He starts by pointing out that truly great designs are often attributed to one person or two people working together
  - He then contrasts this with the way in which collaboration is highlighted and encouraged in society
- On one hand we are told
  - we need to work together to achieve great things
- On the other we have the derogatory phrase
  - “design by committee”
- The danger with collaboration is the potential loss of conceptual integrity
  - The challenge then is how to maintain integrity while doing team design

Why Has Design Shifted From Solo to Teams?

- Brooks examines two forces driving the shift to collaboration in design
  - Technological Sophistication
    - The increasing sophistication of every aspect of engineering is a primary driver
      - Brooks cites examples of applied mathematicians performing computational fluid dynamics on a supercomputer to get the right mix of aqueous and oily components of shampoo correct
      - Cites how it used to be possible to keep track of progress in computer science by monitoring two conferences and two journals
  - Time to Market
    - Teams are needed to get a new design to market; a market leader can often maintain 40% of market share over the long run of a product category; with global communications, ideas spread quickly

Costs of Collaboration (I)

- If you have a task that can be performed without communication
  - then using more workers, gets the task done faster
- The problem with design is that design tasks are rarely “partitionable”
  - they require lots of communication or iteration
- As a result, you cannot necessarily make a design process go faster by simply throwing more people at it => communication costs will soon dominate
- Simply partitioning a design task is itself “a task”
  - Think about dividing a software system into modules
    - It’s tough to get the decomposition correct on the first try
      - There will be gaps and there will be misunderstandings discovered late in the process that will then require rework

Costs of Collaboration (II)

- Learning/Teaching Cost
  - If a one-person design job consists of two parts learning (`l`) and designing (d), the total work when shared across n people is not
    - `work = l + d`
  - but at least
    - `work = nl + d`
- Each person must spend the same amount of time learning about the design and gaining a shared vision before the design work can proceed
  - And anyone teaching people what needs to be learned is no longer working on `d` but on `l`
- It is this simple set-up that led to Brooks's Law: "Adding people to a late software project makes it later"

Costs of Collaboration (III)

- Communication Cost during Design
  - If you have `n` people working on a design, there are `n(n-1)/2` possible communication paths between them
    - For a team of 100 people, there are 4,950 possible communication paths between them
    - If a design task requires communication/coordination (and all design tasks do) then work spent doing communication adds to the work of the overall design task
- Change Control
  - Finally, when a team works on a shared design, there needs to be a way to keep track of all the changes => indeed, given the fact that design is an iterative process requiring lots of changes, change control is paramount!

Reflection

- It should now be clear why there is diversity in software engineering tools
  - Teams make use of tools for communication
    - Trello, Slack, Basecamp, issue trackers, wikis, irc, e-mail, etc.
  - Teams make use of tools for version tracking
    - git and Github, subversion, mercurial, etc.
  - Teams make use of tools for dependency tracking
    - Maven, ruby gems and bundler, npm, Leiningen, etc.
- None of these are compilers, debuggers, editors, etc. which are needed to perform the basic task of programming => most of the tools are for communication and coordination

Additional Points

- Brooks then covers a variety of topics briefly in the remainder of the chapter
  - The need for one or two people to be in charge of a design to ensure conceptual integrity of the overall system and its user interface
  - The benefits of having multiple designers in performing requirements elicitation and exploring design alternatives
    - brainstorming and design competitions
  - The need for design review and the benefits of graphical representations
  - The need for students in computer science to work in contexts that provide them “real-world” design experience before doing work or research in software design or collaborative support for design
- But…

CSCW: Brooks gets it wrong

- Unfortunately, in this chapter, Brooks launches into a wrong-headed attack on the field of Computer Supported Cooperative Work
  - In general, whenever he mentions CSCW in this book, he follows a pattern
    - States a straw-man representation of what CSCW is about
    - Attacks the straw man argument (which doesn't actually represent CSCW)
    - Then states things that the supermajority of CSCW researchers would agree with
- I can only guess that Brooks had a bad encounter or two with a CSCW researcher or CSCW paper and generalized to the entire field
  - It's an unfortunate mistake => CSCW is a vibrant area of research

Rationalism versus Empiricism in Design

- In this chapter, Brooks returns to the two major types of design and how they relate to computer science and software engineering
- Can one design a complex object correctly by thought alone?
  - Rationalism says “yes”; Empiricism says “no”
- Rationalism
  - with the right training, background knowledge, and experience, a designer can achieve “flawless” designs
- Empiricism
  - flawless designs are impossible; our designs will always have problems; we must therefore learn how to determine those flaws by experiment and then iterate on the design to remove them

Software Design

- Is a computer program a mathematical object to be fashioned in abstraction and made correct by proof?
    - Yes
- But, rationalists will assert that programs can be made correctly by properly specifying properties and invariants in advance and then proceeding to refine the program such that those constraints are never violated
    - We talk about this via the phrase “the program worked right the first time!”
- Empiricists will counter that, yes, in principle, a program can be designed this way; the problem is not with the design medium but with the designers
    - we make too many mistakes for this to happen; especially when working on large, complex systems

The Results of Empiricism

- Since empiricists believe that humans will inevitably make mistakes, we have developed a design methodology that includes
  - understanding the problem (requirements)
  - designing before building
  - early prototypes
  - an iterative and incremental approach to implementation
  - testing throughout the process
- This approach works as it is modeled on the approach that enabled big advances in science
  - It’s not to say that this approach is easy, it’s not; lots of hard work involved

The Allure of Rationalism in CS and SE

- The allure of rationalism remains in CS and SE
  - lot of work in *formal methods* that attempt to prove programs correct
    - there are (successful) attempts, for instance, to associate formal proofs with secure operating system kernels
      - this approach however cannot scale to entire systems
    - there is a more empirical counterpart, formal verification, that attempts to show that the program is correct by performing comprehensive testing and modeling
      - In this approach, the challenge is scalability both in modeling large, complex systems as well as their state space; as with most empirical approaches, progress is being made
- Brooks notes that no other design discipline attempts to do this: prove a design correct via rigorous formal methods => empirical approaches dominate

Summary

- In parts II and III of The Design of Design, Brooks looks at various topics related to the process of design
  - We examined two chapters in detail
    - Collaboration in Design
    - Rationalism versus Empiricism in Design
- Scale, Specialization, and Time to Market have shifted design work from a single individual to a team working together
  - This shift leads to problems since design work requires communication and coordination; understanding this reveals insight into the structure and tools of modern software development
- Empirical approaches to software design are critical for making progress; approach based on rationalism still exist but are limited due to problems with scalability

Coming Up Next

- Lecture 28: Chapter 6 in Concurrency Textbook => CSPs and Channels
- Lecture 29: Chapter 8 in Concurrency Textbook => Lambda Architecture
- Lecture 30: TDB