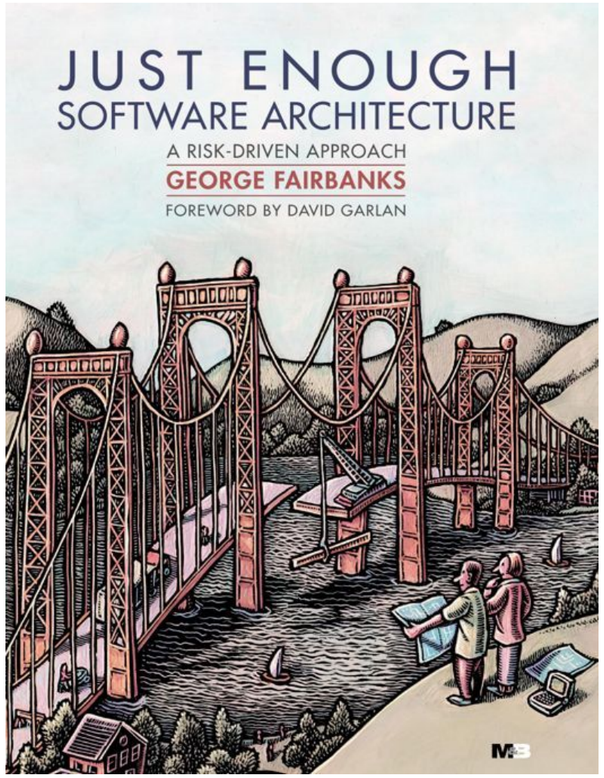
Large Scale Software Engineering

Week 02

Mike Helmick
University of Cincinnati
CS6028
Spring 2014

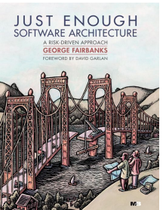
Software Architecture & Software Development Process

- On to Chapter 2 and Chapter 3



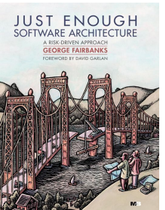
Software Architecture

- Architecture acts as the skeleton of a system
- Architecture influences quality attributes
- Architecture is (mostly) orthogonal to functionality
- Architecture contrains systems



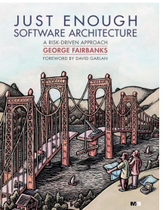
Architecture acts as the skeleton of a system

- All software systems have an architecture
- Usually it is intentional
  - Unintentional architectures tend to be very constraining
- No single right answer



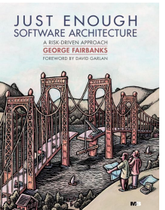
Architecture influences quality attributes

- Externally visible properties
    - security, latency, modifiability, reusability, etc.
- Some architectures might be better or worse
    - Trade-offs: Possibly the most common discussion I've had in software design.
- For example, we can optimize the architecture for reusability and software maintenance, at the expense of runtime performance.



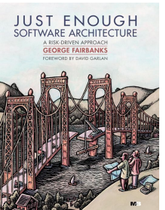
Architecture is (mostly) orthogonal to functionality

- You can take an inappropriate architecture and force it to work in the desired context
  - For example, a peer-to-peer word processor would work, but have weird properties
  - a desktop social network work work, but require each person to visit a single terminal, poor experience
- Architecture constraints are *guide rails* pointing your application in the right direction



Constraints

- Embody judgement
- Promote conceptual integrity
  - “Reduce needless creativity”
- Reduce complexity
- Understand runtime behavior

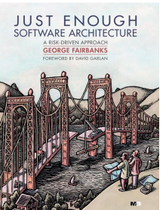


When is architecture important?

- When there is risk involved
- For many simple things, pulling an off the shelf package will suffice. If you aren't
  - … concerned about scalability
  - … concerned about security
  - … concerned about modifiability / maintenance
  - … committed to keeping the first codebase
    - i.e. I can get a demo of something running in ruby on rails in a few hours, but it wouldn't scale to millions of concurrent users.

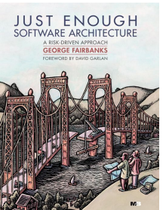
Small solution space

- Architecture becomes important when the problem is extremely constrained
  - Flight control systems
  - Medical device control systems



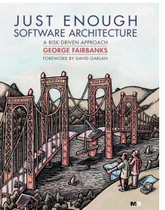
High failure risk

- If a system has sensitive data
- If a system has severe consequences should a failure occur
- Lots of embedded systems fall into this area (as well as the small solution space)



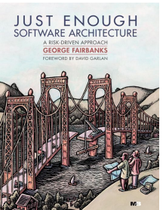
Difficult quality attributes

- Extremely low latency requirements (Google search)
- Extremely high user scalability requirements



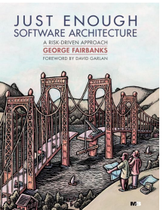
New Domain

- Imagine building your first ever mobile app, when you've built nothing but interactive Web sites in the past
- You can't possibly just do what you've always done in the past



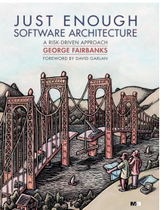
Product lines

- Shared sets of products often have shared architectures



When is architecture important?

- Best answer (from the book)
    - How bad would it be if you got the architecture wrong?



Presumptive Architectures

- An architecture dominant in the domain, a *reference architecture*
- A good match for the common risks in a particular domain
    - 3-tiered systems are often a good match for IT systems, and often used for that reason

How should software architecture be used

- Architecture-indifferent design
- Architecture-focused design
- Architecture hoisting

Architecture-indifferent design

- What it sounds like - developers don't pay attention to the architecture
- “Organic” system
  - This isn’t necessarily a bad thing
- Can be good for short-lived systems, prototypes, limited use systems
  - But can break down if you need to scale

Architecture Focused Design

- Deliberate attention paid to design
    - Suitable and doesn’t impede goals (This is a pretty low bar)
- Make architecture decisions that help you overcome challenges and achieve your goals
- Examples (from the book)
    - Ordering convention for acquiring locks
    - Memory allocation / free standard (when you don’t have garbage collection)

Architecture-Focus Design

- Seeking a global view / global solutions to problems
- Architecture requirements are not always clearly stated
- Level of documentation
  - Depends on complexity / size of project
  - Depends on size of the team
- Independent of your software development process

Architecture Hoisting

- More than just architecture-focused design
- An architecture is selected to guarantee a property
  - By hoisting the solution for some property into the architecture, developers no longer need to work about it

Comparison

- With architecture-indifferent and architecture-focused
    - There's not code that "guarantees" a specific latency
- With architecture-hoisting
    - You can find this code, in the form of load balancing, server management, etc.

Examples of Architecture-Hoisting

- Much of the Java EE stack
    - Application server managers concurrency (hoisting concurrency)
    - Application server scaling (hoisting scaling)
    - EJB hoists concurrency, scalability, and persistence
- Ruby on Rails
    - Hoists persistance
    - Chooses not to hoist concurrency, deferring that to the server chosen (i.e. modrails)

Trade-Offs

- Almost all decisions in software engineering are decisions about trade-offs
- Choosing a garbage collected language
  - Makes development much easier
  - Comes with a runtime performance cost

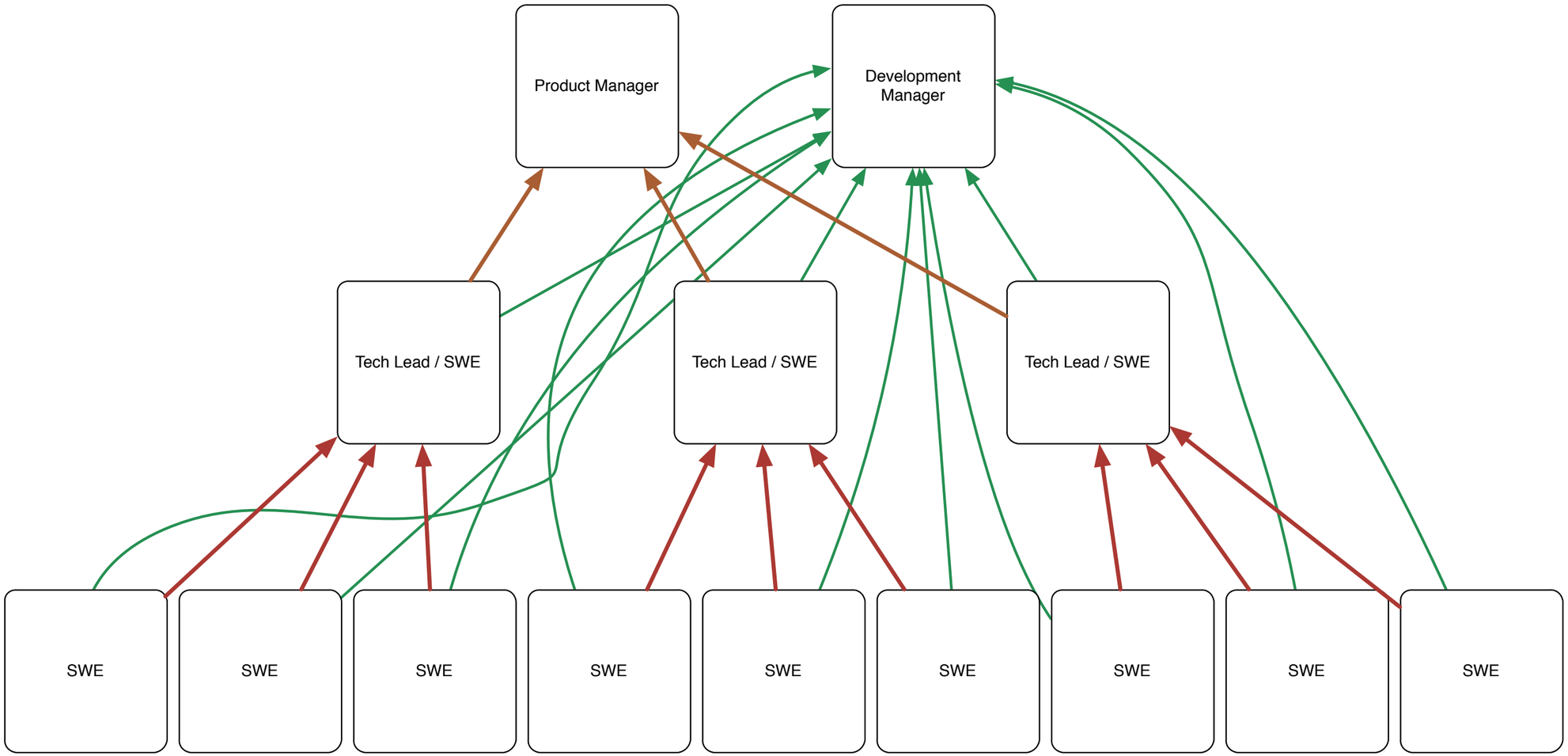
Tyranny vs Liberation

- Architecture-hoisting can be seen as either of these things
- Tyranny
  - Forces developers into a system with constraints, possible additional process (i.e. non-coding tasks)
- Liberation
  - Allows developers to focus on features and their product without worrying about minutia of the architecture

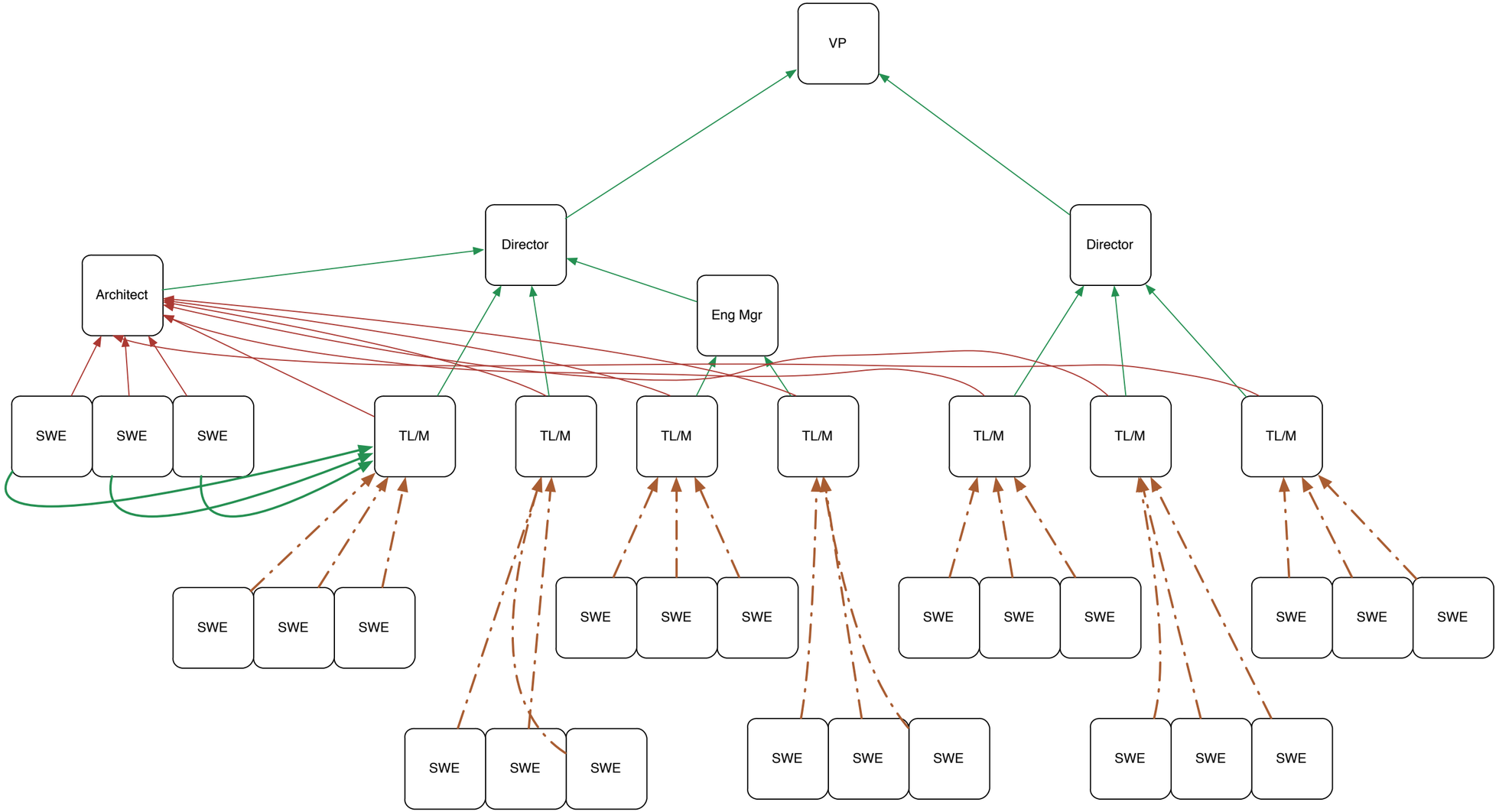
Roles in Large Organizations

- Possible division of labor (book version first)
  - Enterprise architect
    - Responsible for overseeing many applications
    - Sets the tone / common components for many applications, stays out of the details of individual applications
  - Application architect
    - Responsible for a single application, confirming to the enterprise architecture standards
  - Software Engineer
    - Responsible for coding / implementing features of a specific application
    - -or- a general SWE may be working on an architecture product / framework

More Traditional Breakdown



Hybrid Role Breakdown



Pros-Cons

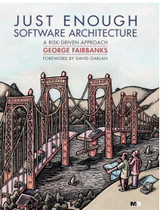
- Enterprise architects can add value in certain organizations
  - Where uniformity is a desirable trait
  - Where software isn't the primary business
- I find that allowing more freedom is a good thing in "tech" companies
  - Larger software development base
  - More tolerance to risk / failure on any individual project

On style

- For large systems:
  - Architecture-focused design is the minimum bar
  - I actually prefer architecture-hoisting
    - If done right, you won't notice it is there
    - Optimize your development staff for productivity

UNIVERSITY OF Cincinnati

Chapter 3 - Risk-Driven Model



Software Engineering

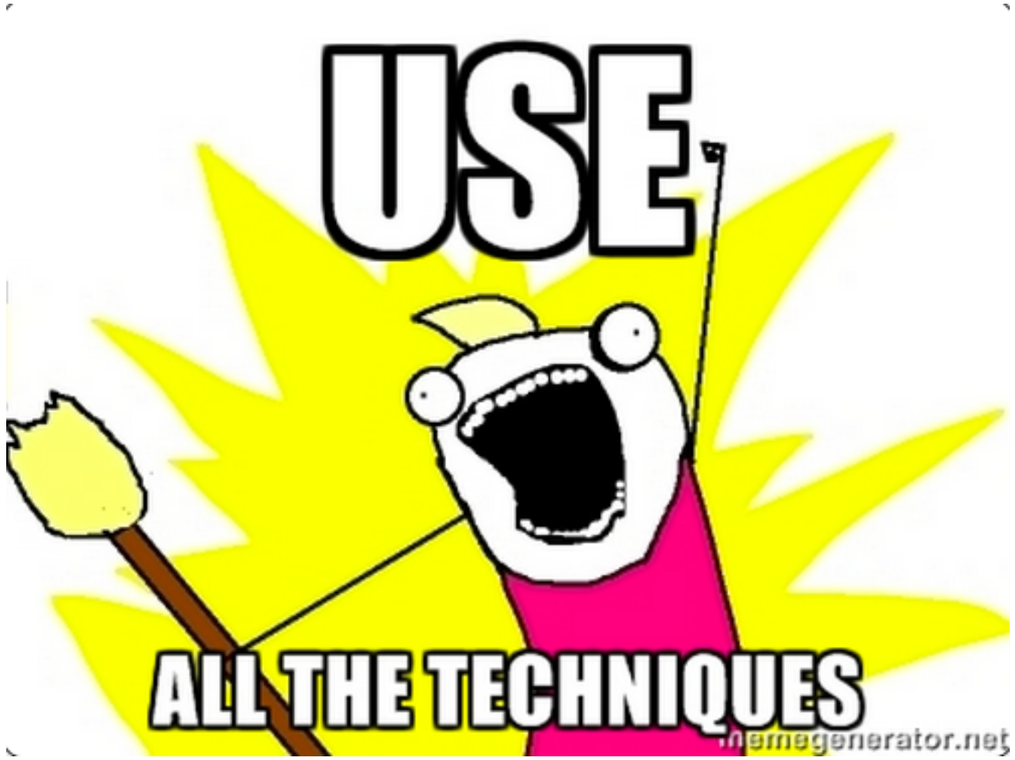
- A large part of software engineering is identifying, and mitigating, risk
  - Identify many possible designs (not all, you can’t identify all possible designs)
  - Select low ricks (not necessarily lowest risk)
  - Move forward
  - repeat

Henry Pertoski, on Engineering as a whole

"The concept of failure is central to the design process, and it is by thinking in terms of obviating failure that successful designs are achieved. … Although often an illicit and tacit part of the methodology of design, failure considerations and proactive failure analysis are essential for achieving success. And it is precisely when such considerations and analyses are incorrect or incomplete that design errors are introduced and actual failures occur."

To Address Risk

- Software developers invented
    - Software design techniques
    - There are now dozens? hundreds? thousands? of software designs, design techniques, and tools to choose from
- So, what should you choose?

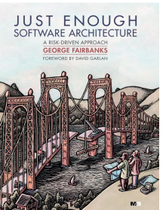


Can’t do that…

- Taking the time to use all of the techniques available would lead to a product that never launches
- So - How much design?

How Much Design?

- **No up-front design**: Just write code. Design happens while coding.
- **Use a yardstick**: x% of the time doing design, x[1]% doing coding, etc..
- **Build a documentation package**: Use a comprehensive set of techniques to produce a complete written design document
- **Ad hoc**: React to project needs, decide on the spot how much design to do. Case by case basis.



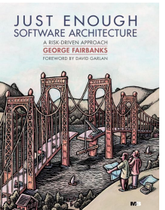
Ad-Hoc

- Author says “perhaps the most common”
    - I agree
- On “Build a documentation package”
    - This kind of described the waterfall method of software development
    - I’ve never seen this be successful for the kind of applications we are talking about

The risk-driven model

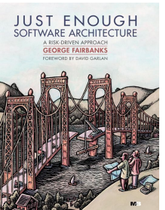
- Guide to applying a *minimal set of architecture techniques* to reduce their *most pressing risks*

    1. Identify and prioritize risks
    2. Select and apply a set of techniques
    3. Evaluate risk reduction



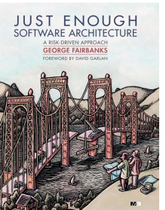
The risk-driven model

- Don’t waste time on low-impact techniques
- Don’t ignore your serious risks



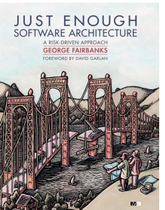
Risk or feature focus

- What you choose to promote has impact
    - -or- What you choose to measure someone's performance on, gets done
- Feature driven vs quality focus
- If you emphasize features, your quality will inevitably start to drop (failure to identify risks)
    - You are racking up technical debt



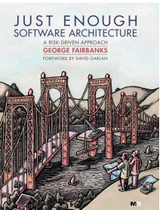
Logical rationale

- -or- logical dissonance
- Not everyone will agree on what the greatest risks are
- This gets solved by design and design review:
  - What is the problem?
  - Why is it a problem?
  - How do you propose to solve it?
  - ***What are the alternatives you considered, and why were they not chosen?***



Are you risk driven now?

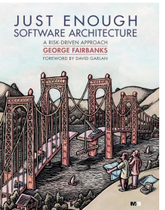
- Techniques should vary
    - If you are always using the same techniques, are you really addressing you project's risks
- Failure to address risk can lead to
    - under design / over design
- Guidelines can go wrong here



Examples of poor guidelines

- The team must always (or never) build full documentation for each system
- The team must always (or never) build a class diagram, a layer diagram, etc..
- The team must spend 10% (or 0%, x%) of the project time on architecture

These kinds of guidelines ignore the task at hand, and potentially ignore the actual risks that the project faces.



Example mismatch

- Standard 3-tier system
    - Front end is open to the internet, handles initial user requests
    - middle tier is behind the firewall

Risks

- The chance of failure times the impact of that failure
- These are mostly ambiguous, especially in software
    - i.e. they are nearly impossible to measure in a general sense
- So, more accurately for our software engineering purposes
    - risk = *perceived* probability of failure x *perceived* impact

Still Imprecise

- You can perceive a risk where it doesn't exist
- You can fail to perceive a risk where it does exist

Describing Risk

- It is best to describe risks in a way that you can latter test to ensure you have mitigated
- Example from the book
  - “During peak loads, customers experience user interface latencies greater than five seconds”

Engineering and Project management risks

- *Engineering Risks*
  - Things related to analysis, design, and implementation of the product
  - **Production deployment, support and maintenance** (big hole, missing from the book)
- *Project Management Risks*
  - Schedules, sequencing, team size

Identifying Risks

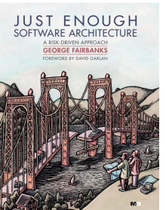
- The ability to identify risk comes with experience
    - Once you observe a failure, it becomes part of your experience, and you will remember it in the future
- You will miss some risks

Prototypical Risks

- Some domains have well known risks
    - They are part of most projects in the domain
- These often arise from “lessons learned” or failure “postmortem” documents / reviews

Prototypical risks

Project Domain	Prototypical Risks
Information Technology	Complex, poorly understood problem Unsure we're solving the real problem May choose the wrong COTS software Domain knowledge scattered across people Modifiability
Systems	Performance, reliability, size, security Concurrency Composition
Web	Security Application scalability Developer productivity / expressability



Prioritizing Risks

- This can be difficult to do
    - Not everyone will agree on what is the most important to mitigate
- Take into account the
    - priority of stakeholders (business people / customers)
    - priority of developers
- It may be the case, that handling something that isn't the biggest project risk first, will make a "more important" risk easier to deal with

Techniques

- Analysis
  - extensive modeling, simulation
  - Designing a new distributed transaction protocol
- Solutions
  - Known, existing, previously working solutions
  - Implement paxos, a well known, working solution

Techniques Mitigate Risks

- Design is a mysterious process
    - The best designers I have seen (and stated in the book) are able to make huge leaps from problem to solution
    - Sometimes these kinds of leaps are bad, they don’t capture the risk, and don’t explain to the rest of the team how we got where we are going
- Capturing design decisions and general advice can work (so says the book)
    - In practice, I’ve never seen this done on a company wide scale
    - I have seen collections of lessons learned.
        - X happened, this is why, this is the list of work to fix it. Please learn from our mistake.

Optimal basket of techniques

- Are there techniques that can work to reduce more than one of your project risks?
    - Standard *optimization problem*
- Just because something worked before, doesn't mean it is appropriate here.
    - Don't just pull a solution out of the basket, always do your analysis

Risk Elimination

- Can’t be done
- The only way to do this is spend forever developing your project, and never launching
  - i.e. there isn’t enough time for this
- You have to balance it with project management risk / requirements

Modeling

- Our textbook describes various modeling techniques
  - UML / ADL
- For your really big, fast moving systems….
  - This typically isn’t done
  - Things are moving to fast to put a lot of formalism around the process

When to stop

- Who much design and architecture should you do?
    - Clearly this could go on forever
    - Time spent doing design, is time spent not building your system

Effort and Risk

- Effort should line up with risk in some way
  - If something isn't a huge deal if it fails
    - no sensitive data lost
    - no functionality severely compromised
    - easy to recover from
  - But - pay attention to your scale and failure rates
    - If something happens 1 in every 1 billion requests (statistically), how long before you get a billion requests, and then how frequently would this occur?
    - This analysis might surprise you, and up the priority on something

Incomplete Architecture Designs

- If you take the risk-driven approach to architecture and design
- You could have an incomplete design
    - Since you only spend upfront time on the areas where you perceive risk
- In general, this is probably OK, of course there are risks here
    1. You could be totally around about an area not being risky
    2. You could find that this missing hole of design causes delay late

Evolutionary Design

- Means that the design grows while the system is implemented
  - Can lead to chaos
  - Can lead to success
    - This is what a lot of agile process systems end up following
    - I have found this to be very successful in practice

Planned Design

- Plans worked out in details first, and then you code
- Sometimes called *Big Design Up Front* (BDUF)
- In practice, I have found this to be unsuccessful
  - Inherent inability to responding to changing requirements
  - Sometimes clarity on requirements is only apparent during development
    - Waiting delays discovering these issues

Minimal planned design

- between evolutionary and planned design
- Used because of a (real) worry that only evolutionary design may leave you with problems that are expensive or impossible to overcome
  - but, planned design can be difficult and also full of errors
- To me, it usually make sense to do some up front architecture and design
  - Enough to ensure that your biggest risks and/or desired quality attributes are addressed
  - If you want your system to scale, you are going to need to do some architecture work up front
    - or else you will be very luck to not have to throw away the first version of your system

Software Development Process

- How the team goes about developing software - too obvious?
    - Having a process helps you set priorities
    - There can be many different processes, some named, some ad-hoc, that can fit with the risk-driven architecture approach

Vocabulary

- The book talks about risk as a shared vocabulary
    - This is important, but not just for risk
- The book talks about managers who aren't technical
    - This happens. This is where having a dedicated technical leader can help.
        - Someone that interfaces with both the engineering team and the business stakeholders
    - Ideally - the direct management will be former/current engineers

Baked-in Risk

- The process you choose can be seen as a risk driven decision
- Guaranteed delivery dates might use a heavy/waterfall approach to ensure that X functionality is delivered on Y date.
- Concern about initial delivery / time to market, leads towards agile processes
- Based on my experience:
  - The successful projects I have been a part of / observed, use some kind of agile / ad-hoc process.
  - Delivery window: If you are able to shorten the time between releases, this is a great, baked-in risk mitigation. It makes the cost of missing a deadline much lower

Waterfall Method

- Long iterations that deliver an entire project, or an entire set of features (can be used ongoing)
- Planned design - complete design.
    - Designed are signed off on, and what is in the design is what gets built
- Move on to construction phase
- Move on to test phase
- Move on to delivery phase
- Not a lot of movement / backtracking

Iterative

- Builds the system in rapid cycles, called iterations
    - An iteration can build new, or can rework
- The risk-driven approach
    - Assess the level of architecture / design that needs to be done for each iteration

Spiral Model

- An iterative model where the riskiest items are addressed first
- With this, there should be less architecture / design in successive iterations

Rational Unified Process (RUP)

- iterative / spiral processes
- Highlight addressing risk early
- Somewhat commercial
    - There are tools that you can buy. Now an IBM product, used to be from Rational

Extreme Programming (XP)

- Suggests avoiding up front design work, evolutionary design

Many Others

- There are many other processes out there
    - I have worked places that have done waterfall
    - I have worked places that have done scrum (an agile process)
    - I have done at least 3 projects that were some kind of “ad-hoc” process

Risk-driven Model and Agile Process

- Risks become part of your planning for each iteration
    - Even with a fully agile process, you need to do some planning
- For starting a new system - I would spend a little more time up-front on overall system architecture
- Risks become part of the backlog (what doesn't fit into the current iteration)
    - You can't address all features in a single iteration, you can't address all risks in a single iteration

Prioritizing Risks and Features

- I suggest gate criteria to get around this
    - These risks must be addressed before launch
    - with the understanding that prioritizing features over those risks delays launch
- This happens even after launch
    - A P0/P1 bug is discovered, not addressing it is the largest risk to your system
        - You probably drop everything and fix it

Architecture Refactoring

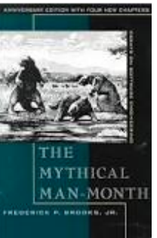
- Eventually, you will realize that the best design wasn't chosen
- It is possible to refactor all or part of your architecture
  - Similar to how you might refactor small bits of code
  - But on a larger scale
- This takes time
  - It may involve a whole system rewrite, or a partial system rewrite (extracting functionality)
  - It may involve a data migration (usually painful)

Alternative to Risk-Driven Model

- **No design**
  - This is hard to achieve - probably impossible to build a truly scalable system
- **Documentation package**
  - Fully document everything you do, build models, etc
- **Yardsticks**
  - X% time on each task
- **Ad hoc**
  - make the decision in the moment
  - In practice, this works quite well.
  - The author calls this an informal risk-driven model

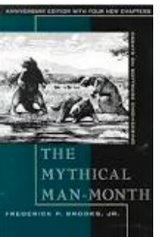
The Mythical-Man Month

- Frederick Books' classic collection of essays on software engineering
  - Originally published in 1975
  - Still relevant today



The Tar Pit

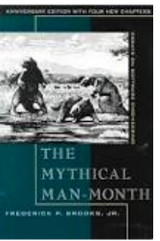
- He describes large systems development as a tar pit
  - “many great and powerful beasts have threshed violently in it”
  - Everyone seems surprised by the stickiness of the problem
    - Hard to discern the nature of it



The Garage Team

- Back then, and still today, you hear about a team of 2 or 3 people in a garage doing something that a large team doesn't seem to be able to do

- Why can't large industrial teams move that fast?



What is being produced

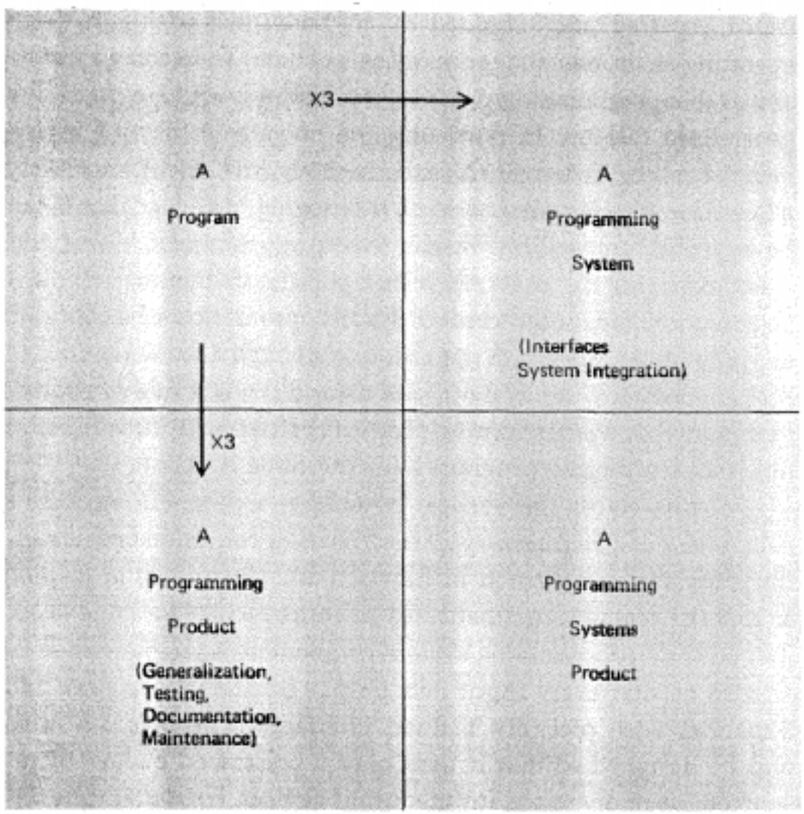
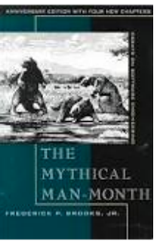


Fig. 1.1 Evolution of the programming systems product



Comparison

- The “Program” is something that works in the small
  - Easily produced by a small team, doesn’t scale, not robust, not resilient to faulty input
- 3x cost to make the product robust
- 3x cost to make the product work in a larger systems ecosystem
- 9x cost the program in the small, to the generalized program in the large

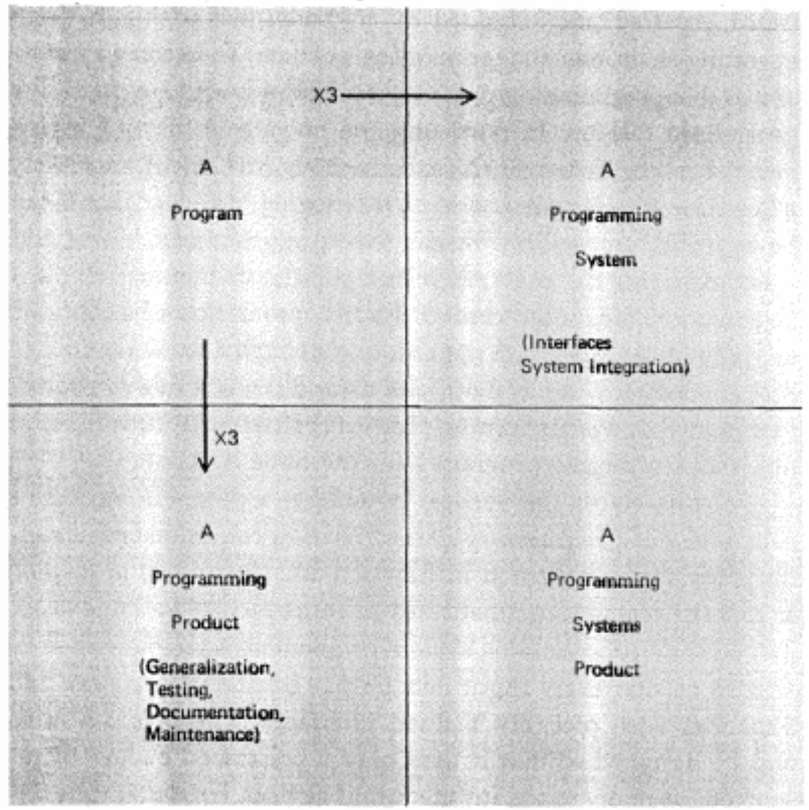
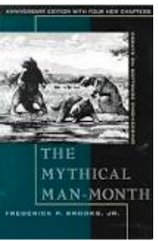


Fig. 1.1 Evolution of the programming systems product



Large Scale Software Engineering

- Requires that you get to the “programming systems product” quadrant
  - Maybe not for launch
  - But if you ever want to
    - disconnect from your pager
    - be able to stop treading water long enough to develop new features…

The “Woes of the Craft”

- You must perform perfectly
  - *If you get some basic distributed systems / redundancy right, you can be less than perfect sometimes*
- Have to work in a team
  - *You must communicate, you must work together with others to build a large system*
- Whatever you are working on is already, or about to be obsolete