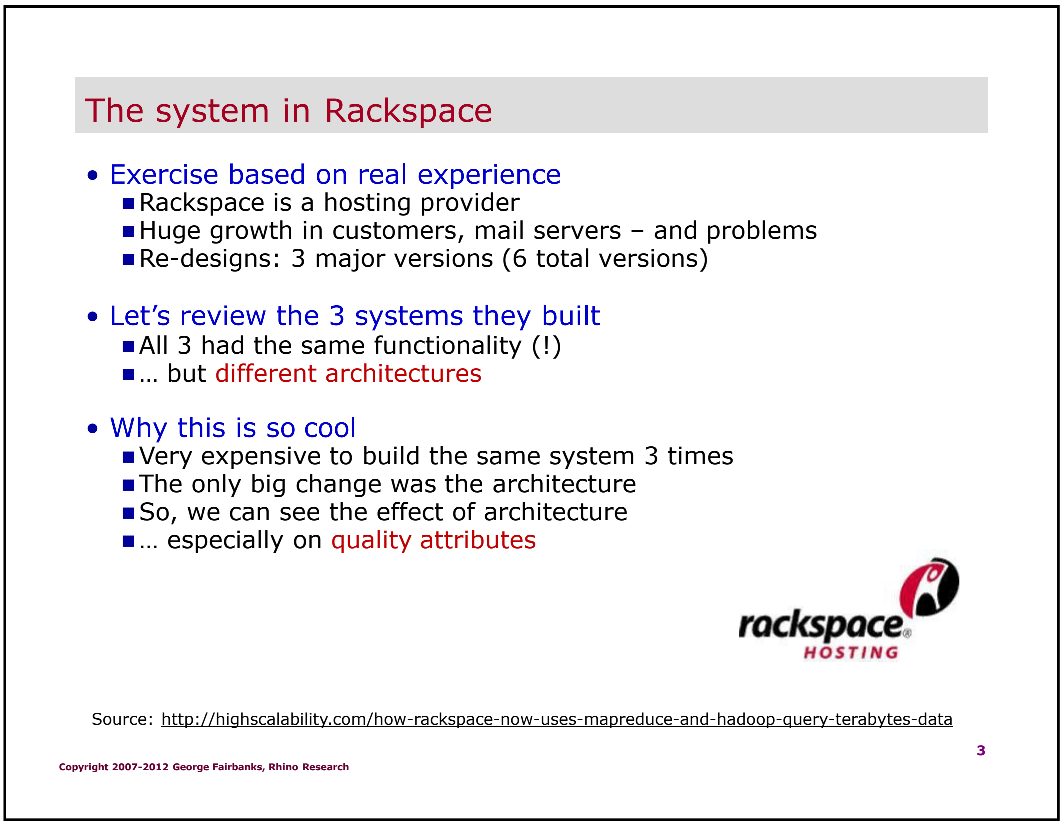
Software Architecture with real world example

Rhino Research

Let’s look into a real system: Rackspace

- Here’s the situation
  - Rackspace is a hosting provider
  - It rents mail servers
  - Customers have problems
  - It uses the mail log files to diagnose their problems
- The big question:
  - How would Rackspace build it?
- How to build it?
  - … different architectures yield different qualities
- Why is this hard?
  - It has hundreds of servers
  - It generates GBs of logs daily
  - Collecting logs takes time
  - Searching logs takes time
- Options
  - Central collection of logs?
  - Distributed searching of logs?
  - Pre-process logs to speed up queries?

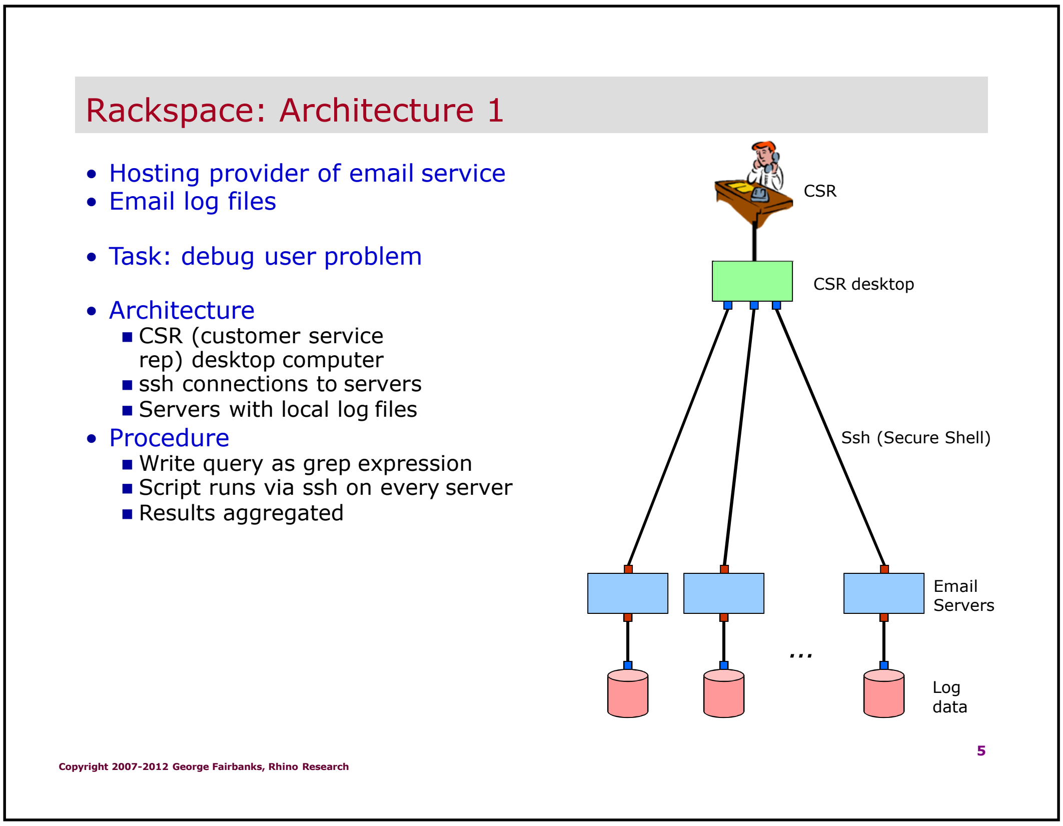


The system in Rackspace

- Exercise based on real experience
  - Rackspace is a hosting provider
  - Huge growth in customers, mail servers – and problems
  - Re-designs: 3 major versions (6 total versions)
- Let’s review the 3 systems they built
  - All 3 had the same functionality (!)
  - … but different architectures
- Why this is so cool
  - Very expensive to build the same system 3 times
  - The only big change was the architecture
  - So, we can see the effect of architecture
  - … especially on quality attributes

Source: http://highscalability.com/how-rackspace-now-uses-mapreduce-and-hadoop-query-terabytes-data

3



Rackspace: Architecture 1

- Hosting provider of email service
- Email log files
- Task: debug user problem
- Architecture
  - CSR (customer service rep) desktop computer
  - ssh connections to servers
  - Servers with local log files
- Procedure
  - Write query as grep expression
  - Script runs via ssh on every server
  - Results aggregated

5

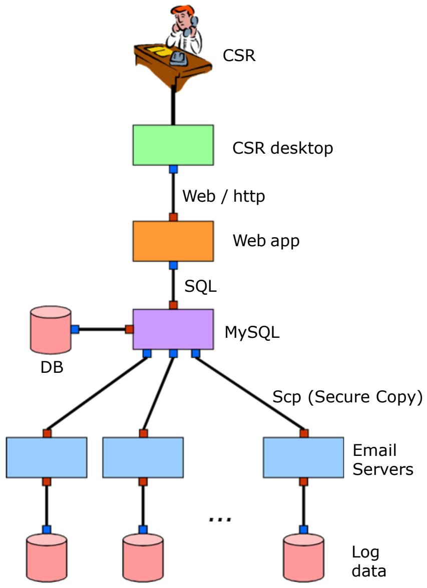
Rackspace: Architecture 2

- Hosting provider of email service
- Email log files

- Task: debug user problem

- Architecture
  - CSR desktop computer
  - Web application
  - MySQL database
  - scp log transfer
  - Servers with local log files

- Procedure
  - Every 10 minutes, send log files to MySQL server; delete original
  - Parse and load logs into MySQL
  - Combine new logs with old
  - Send query to MySQL server; answered from DB data



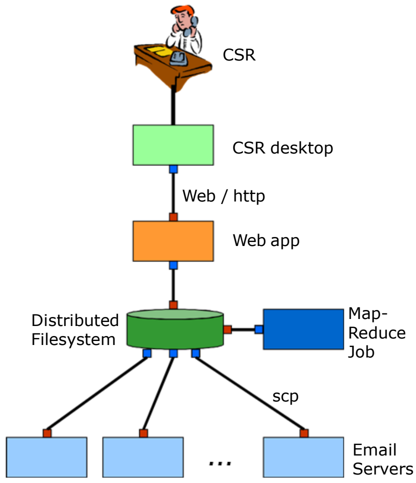
Rackspace: Architecture 3

- Hosting provider of email service
- Email log files

- Task: debug user problem

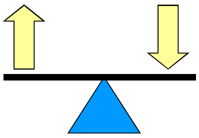
- Architecture
  - CSR desktop computer
  - Web application
  - Distributed filesystem
  - Map-Reduce job cluster
  - Servers with local log files

- Procedure
  - Log data continuously streamed from email servers to distributed filesystem (HDFS - Hadoop Distributed File System)
  - Every 10 minutes, Map-Reduce job runs to process log files, create index
  - Web app queries index



Rackspace: Quality attribute tradeoffs

- Tradeoff: Data freshness
  - V1: Queries run on current data
  - V2: Queries run on 10 minute old data
  - V3: Queries run on 10-20 minute old data
- Tradeoff: Scalability
  - V1: Noticeable email server slowdown (dozens of servers)
  - V2: MySQL speed/stability problems (hundreds of servers)
  - V3: No problems yet
- Tradeoff: Ad hoc query ease
  - V1: Regular expression
  - V2: SQL expression
  - V3: Map-Reduce program

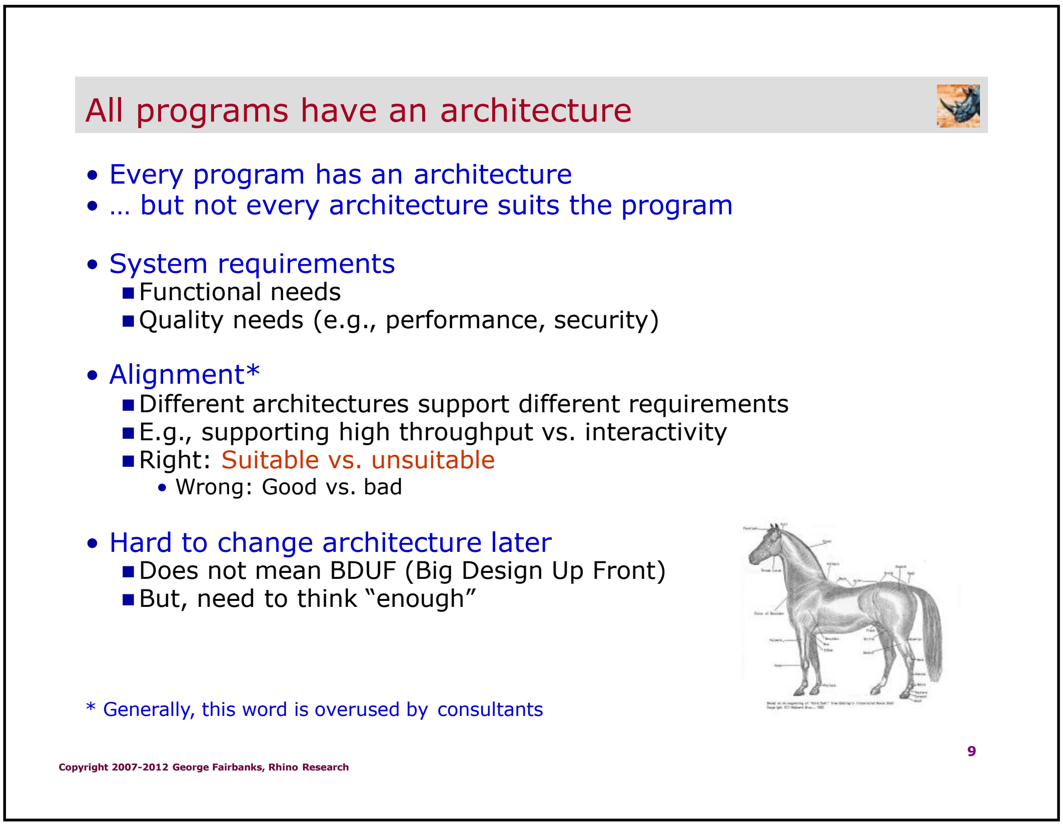


What is software architecture?

The **software architecture** of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. [Documenting Software Architectures (SEI) 2010]

Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. [IEEE 2000]

- In loose language:
  - It's the "big picture" or "macroscopic" organization of the system
- Problem with these definitions
  - Why are some detailed designs architectural, others not?
  - Architecture includes whatever architects say it does



All programs have an architecture

- Every program has an architecture
- … but not every architecture suits the program

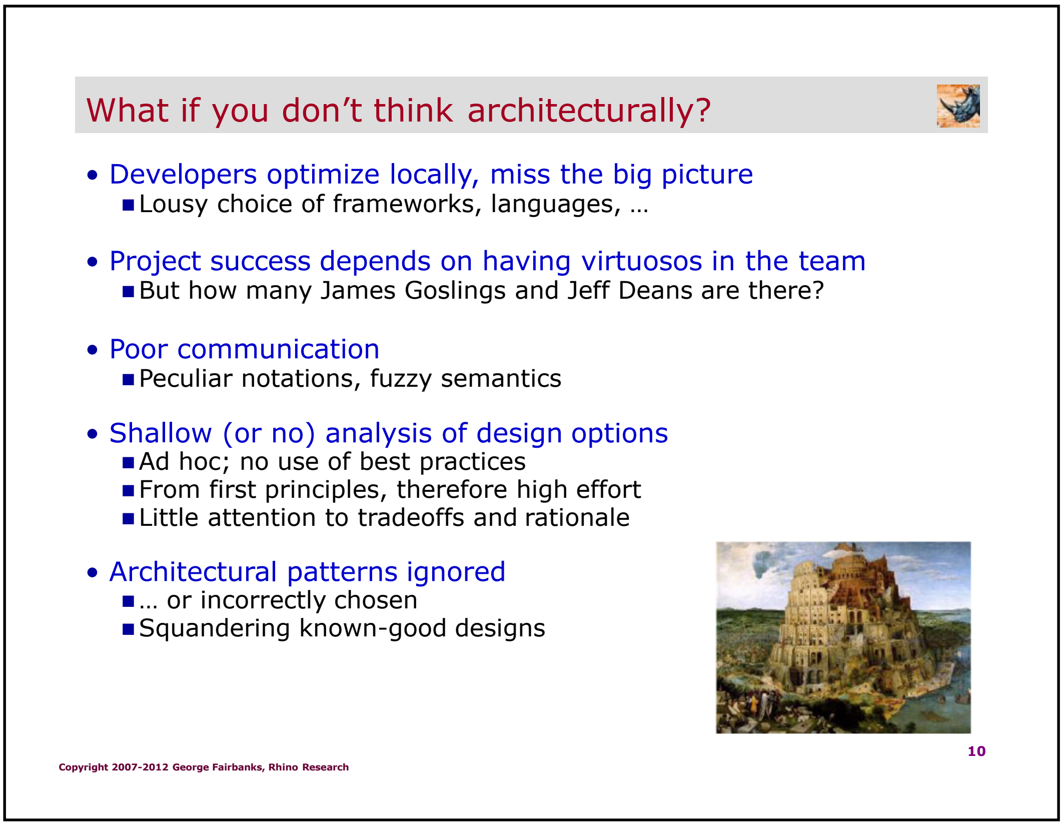
- System requirements
  - Functional needs
  - Quality needs (e.g., performance, security)

- Alignment*
  - Different architectures support different requirements
  - E.g., supporting high throughput vs. interactivity
  - Right: Suitable vs. unsuitable
    - Wrong: Good vs. bad

- Hard to change architecture later
  - Does not mean BDUF (Big Design Up Front)
  - But, need to think "enough"

* Generally, this word is overused by consultants

9



What if you don't think architecturally?

- Developers optimize locally, miss the big picture
  - Lousy choice of frameworks, languages, …

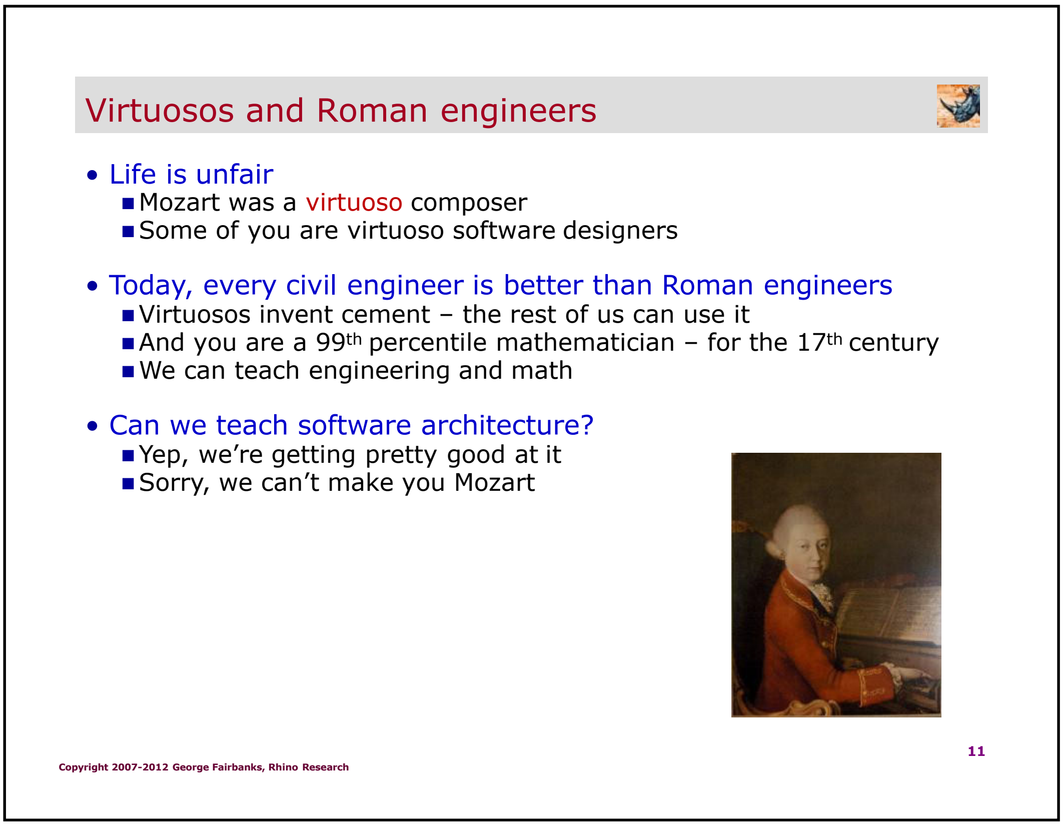
- Project success depends on having virtuosos in the team
  - But how many James Goslings and Jeff Deans are there?

- Poor communication
  - Peculiar notations, fuzzy semantics

- Shallow (or no) analysis of design options
  - Ad hoc; no use of best practices
  - From first principles, therefore high effort
  - Little attention to tradeoffs and rationale

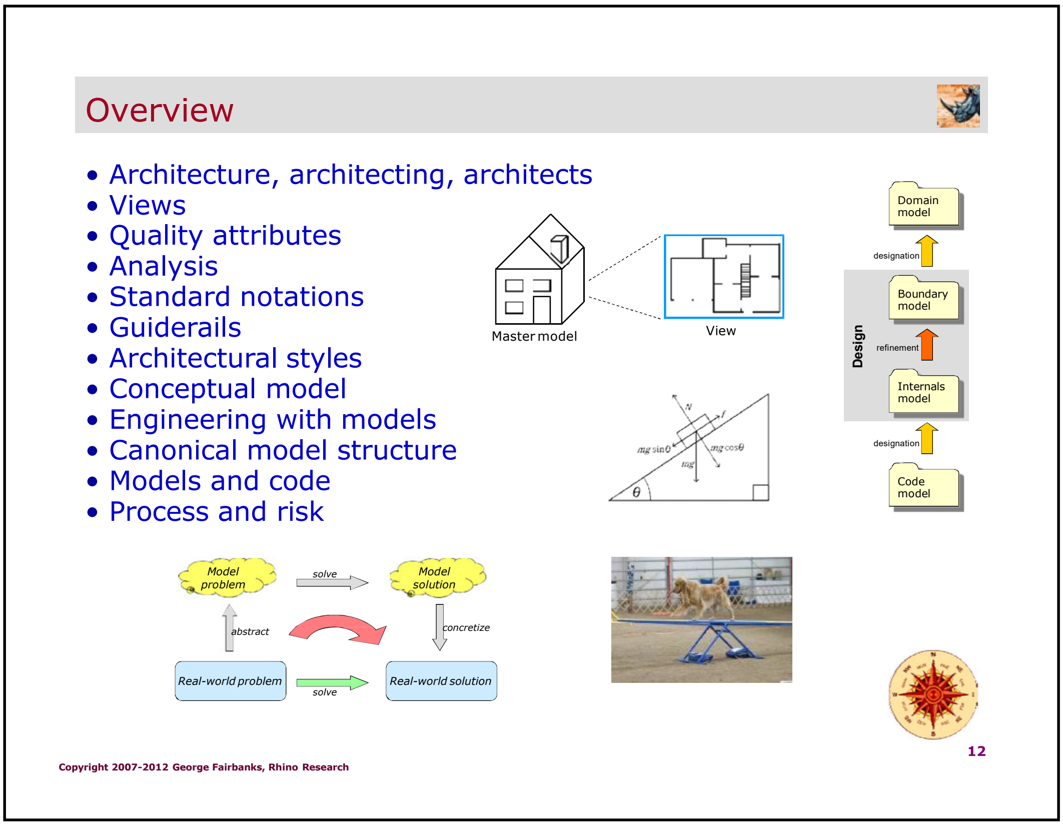
- Architectural patterns ignored
  - … or incorrectly chosen
  - Squandering known-good designs

10



Virtuosos and Roman engineers

- Life is unfair
  - Mozart was a virtuoso composer
  - Some of you are virtuoso software designers
- Today, every civil engineer is better than Roman engineers
  - Virtuosos invent cement – the rest of us can use it
  - And you are a 99th percentile mathematician – for the 17th century
  - We can teach engineering and math
- Can we teach software architecture?
  - Yep, we're getting pretty good at it
  - Sorry, we can't make you Mozart



Overview

- Architecture, architecting, architects
- Views
- Quality attributes
- Analysis
- Standard notations
- Guiderails
- Architectural styles
- Conceptual model
- Engineering with models
- Canonical model structure
- Models and code
- Process and risk

Architecture vs. architecting vs. architect

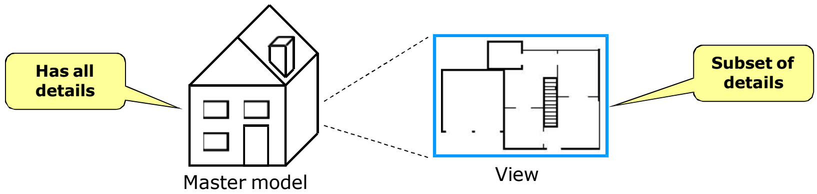
- Must keep these ideas separate:
  - The **job title/role** “architect”
  - The **process** of architecting/designing (also: when)
  - The **engineering artifact** called the architecture
- Course focus: **architecture** (the engineering artifact)
- Every system has an architecture
  - Identify it by looking back (avoids tangling with process & roles)
  - E.g., “Aha, I see it is a 3-tier architecture”
- Help disentangling
  - Car architectures
  - Rackspace architectures

See: Just Enough Software Architecture, Ch 1 Sec 5

Views

Views

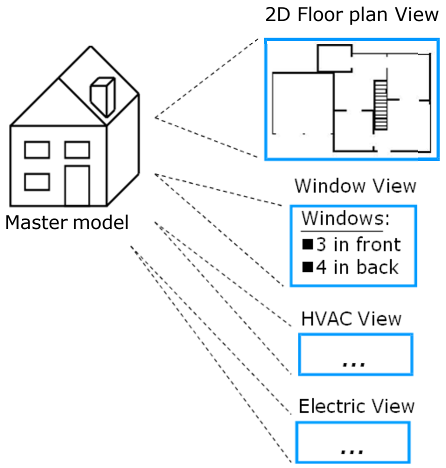
- Definition
  - A **view** is a **projection** of a model showing a subset of its details
  - A view is a relationship between two models
- Views: the modeling workhorse
- Projections from **master model**
  - Master model has all details
    - I.e., THE design
  - Views are projections of the master model
    - Subsets of its information
- Master model may not concretely exist
  - E.g., build top-down 2D view of house, imagine 3D model
  - Imagine 3D house modeling software
    - Can project any cross-section (view)
    - Ignore concrete representation of 3D model (arbitrary choice)



See: Just Enough Software Architecture, Ch13

Multiple views

- Example house views:
  - 2D view of floor plan
  - Electric wiring circuits
  - CAT5 wiring and routing
  - HVAC distribution
  - Plumbing
  - Landscaping
  - Inventory of windows
  - Taxation
  - Zoning



See: Just Enough Software Architecture, Ch13

Architectural viewtypes

- Definition:
  - A **viewtype** is a category of views that are easy to reconcile with each other.
    - E.g., physical, political views of a house
- Reconciling views in a viewtype
  - Easy within viewtype
    - E.g., electrical and floorplan = easy
  - Hard between viewtypes
    - E.g., taxation and roofing = hard
- Standard architectural viewtypes
  - Module viewtype
    - Source code, config files, module dependencies
  - Runtime viewtype (aka component and connector, C&C viewtype)
    - Components, connectors, ports
  - Allocation viewtype
    - Servers, geography

See: Just Enough Software Architecture, Ch13

Quality Attributes

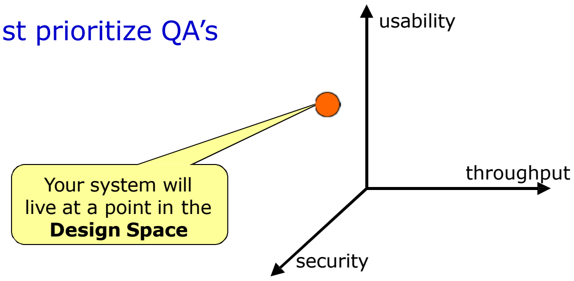
Quality attributes (QA's)

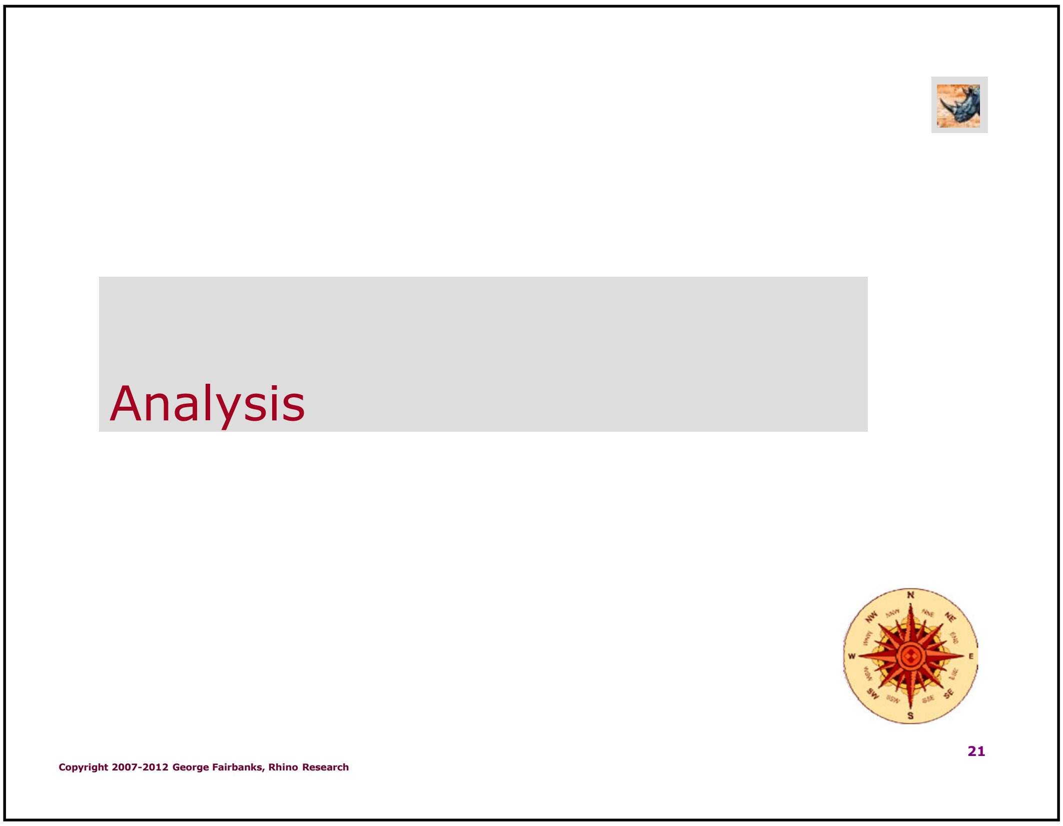
- Definition: A **quality attribute** is a dimension of quality used to evaluate a software system.
  - E.g., performance, scalability, modularity, usability, security
  - A.k.a., non-functional qualities, extra-functional qualities, the "ities"
- Generally, *any* architecture can achieve *any* feature
  - **BUT**: qualities will suffer or be harder to achieve
- Why study QA's?
  - Significant failure risks from QA's
  - Intersection of business & technology
- Software architecture & QA's
  - Architecture decides range of QA possibilities
  - Architectures evaluated w.r.t. QA's

See: Just Enough Software Architecture, Ch 12 Sec 10

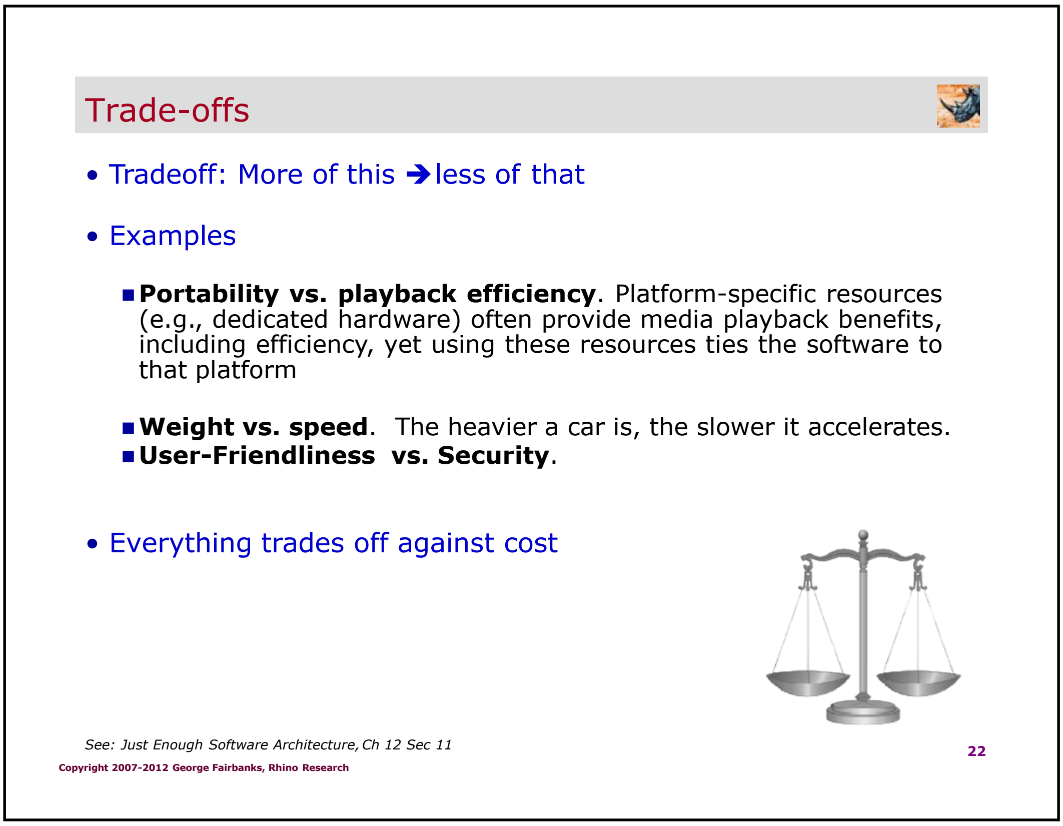
QA's as independent dimensions

- Optimizing one QA
  - Architecture, design changes to maximize a QA
  - Generally, 1 QA is easy
- Optimizing across many QA's
  - Cannot maximize all QA's at once
  - Tradeoffs
    - E.g., most usable UI might be less secure
    - E.g., highest throughput is batch mode UI
  - "**Design space**" – choosing is hard
- Consequence: Must prioritize QA's





Analysis



Trade-offs

- Tradeoff: More of this ➔less of that
- Examples
  - **Portability vs. playback efficiency**. Platform-specific resources (e.g., dedicated hardware) often provide media playback benefits, including efficiency, yet using these resources ties the software to that platform
  - **Weight vs. speed**. The heavier a car is, the slower it accelerates.
  - **User-Friendliness vs. Security**.
- Everything trades off against cost

See: Just Enough Software Architecture, Ch 12 Sec 11

Trade-offs

- **User-Friendliness vs. Security**.
- In real-world scenarios, system designers face the challenge of striking a balance between user-friendliness and security. For example, a more secure system might enforce stronger password policies, limit the duration of persistent logins, and maintain rigorous authentication steps.
- However, these measures could potentially lead to a less user-friendly experience.

- Ultimately, the design must align with the organization's risk tolerance and the nature of the data or transactions involved. Striking the right balance is crucial to provide users with a positive experience while ensuring the security and integrity of the system.

See: Just Enough Software Architecture, Ch 12 Sec 11

Architecture drivers

Architecture drivers

- Template: stimulus and response
  - Stimulus: agent or situation that triggers scenario
  - Response: reaction to stimulus
- Each QA scenario can be graded by:
  - Importance to stakeholder (high, medium, low)
  - Difficulty to implement (high, medium, low)
- Architecture drivers are
  - QA scenarios
  - or functional scenarios (eg use cases)
  - that are rated (H,H)

Examples

- S1 (H,H):
  - When a librarian scans a book copy for checkout, the system updates its records and is ready to scan the next one within 0.25 seconds.
- S2 (M,H):
  - When librarian station cannot contact the main system, librarians can continue to check books in and out.

QA scenarios and drivers from Bass et al., Software Architecture in Practice, 2003
See: Just Enough Software Architecture, Ch 12 Sec 11

Rational architecture decisions

- Design rationales explain **why**
- They should align with your quality attribute priorities

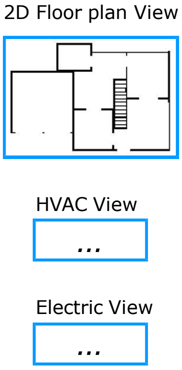
<x> is a priority, so we chose design <y>, accepting downside <z>.

- An example:
  - Since avoiding vendor lock-in is a high priority, we choose to use a standard industry framework with multiple vendor implementations, even though using vendor-specific extensions would give us greater performance.
- But: Good intentions can go awry
  - E.g., performance optimization hindering modifiability

See: Just Enough Software Architecture, Ch 12 Sec 11

Analyzing views

- Views make analysis easier
  - Choice of view essential
- Some views have custom visualizations
  - Usually improve analysis or comprehension
- Which view makes the question easy?
  - What is shortest path?
  - Estimated temperature?
  - Impact of short in bathroom?
  - Good afternoon reading light?
  - Tax burden of new wing?



2D Floor plan View

HVAC View

...

Electric View

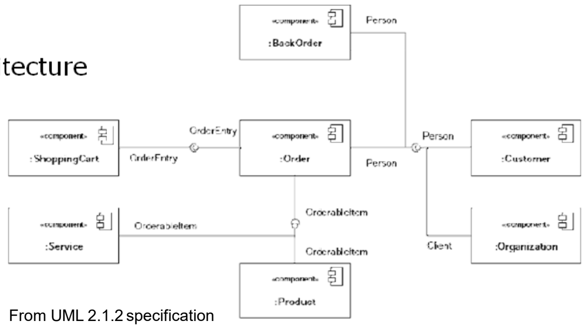
...

See: Just Enough Software Architecture, Ch15

Standard Notations

Standard notation (UML)

- “By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.”
  – Alfred Whitehead, 1911
- Clear, consistent notation
  - Aids communication
  - Aids analysis
- UML
  - Not perfect for architecture
  - One size fits all



From UML 2.1.2 specification

Guiderails

Guiderails (constraints)

- Developers **voluntarily constrain** systems
  - Counter-intuitive
  - Ensures what a system **does not do**
  - I.e., guiderails
- Constraints help ensure outcomes
  - E.g., ensure quality attributes are met
  - **No constraints = no analysis**
- Examples of architectures ➔ QA's
  - Plugins must use cross-platform API to read files ➔ portability
  - EJBeans must not start own threads ➔ manageability
  - EJBeans must not write local files ➔ distribution

See: Just Enough Software Architecture, Ch 16 Sec 4

Architectural Styles

Architectural styles

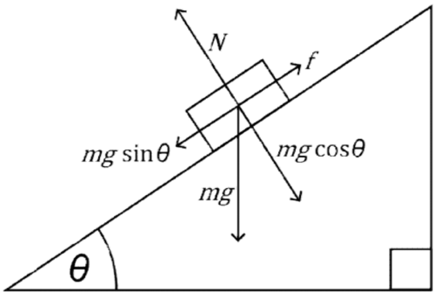
- Examples
  - Big ball of mud
  - Client-server
  - Pipe-and-filter
  - Map-reduce
  - N-tier
  - Layered
  - …
- Each predefines
  - Elements (e.g., pipes, map functions)
  - Constraints, …
- Benefits
  - Known tradeoffs
  - Known suitability
  - Compact terminology for communication

See: Just Enough Software Architecture, Ch14

Conceptual Model

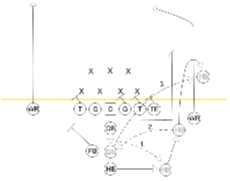
What is a conceptual model?

- What is a conceptual model?
  - A conceptual model is a set of concepts that can be imposed on raw events to provide meaning and structure.
- It organizes chaos
  - Enables intellectual understanding
  - Fits big problems into our finite minds
- Synonyms:
  - Conceptual framework
  - Mental model

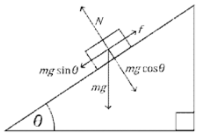


See: Just Enough Software Architecture, Ch7

Examples of conceptual models

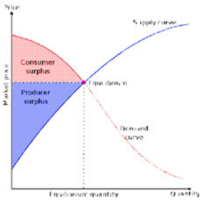


Sports: Plays, strategies, assignments



Physics: Free Bodies

Energy cycle



Econ: Supply & demand

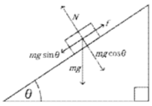
Accounting: Debits & credits

35

Conceptual model of software architecture

- Model relationships
  - Views & viewtypes
  - Designation
  - Refinement
- Canonical model structure
  - Domain model
  - Design model
    - Internals model
    - Boundary model
  - Code model
- Quality attributes
- Design decisions
- Tradeoffs
- Responsibilities
- Constraints (guide rails)

- Viewtypes
  - Module
  - Runtime
  - Allocation
- Module viewtype
  - Modules
  - Dependencies
  - Nesting
- Runtime viewtype
  - Components
  - Connectors
  - Ports
- Allocation viewtype
  - Environmental element
  - Communication channels



See: Just Enough Software Architecture, Ch7

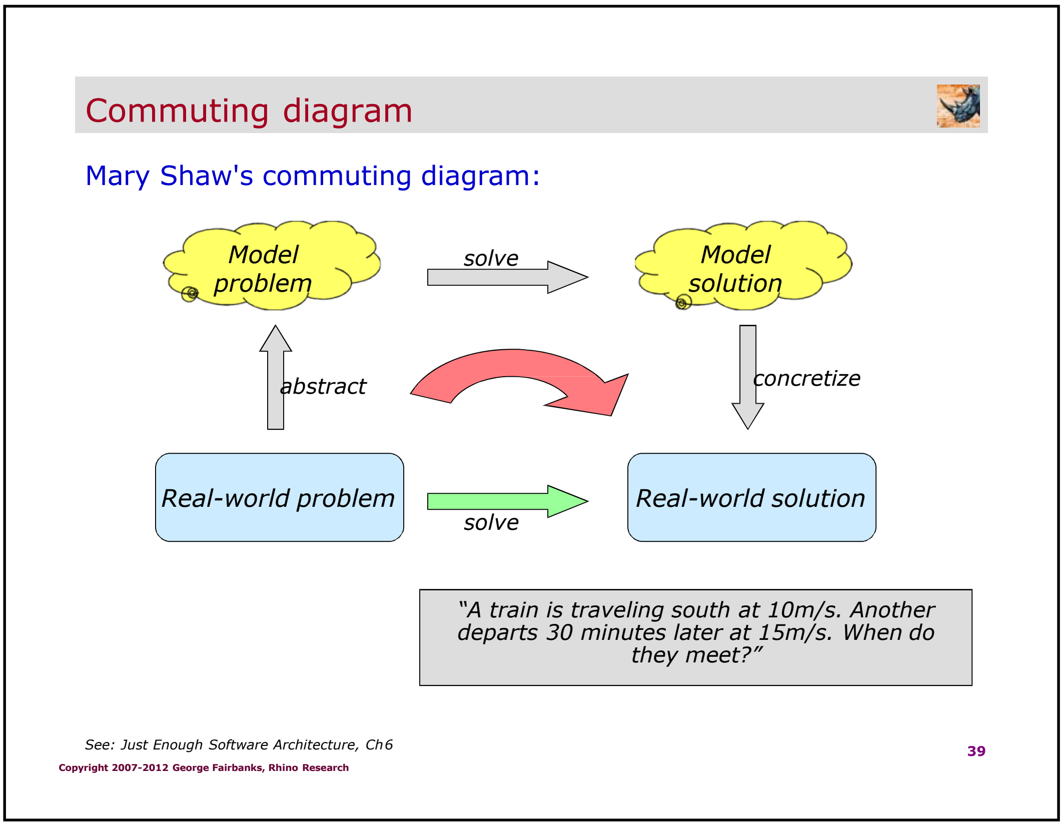
36

Engineering with Models

Why use models?

- We battle complexity and scale with models
  - Models fit in our heads
  - Models help us analyze the problem
- So, what kinds of (meta) models?
  - Enterprise Architecture: many competing (meta) models
  - Application Architecture: general consensus
- Use != Build
  - How much you write down is a choice
  - But you need a (meta) model

See: Just Enough Software Architecture, Ch6



Commuting diagram

Mary Shaw's commuting diagram:

Model problem

solve

Model solution

abstract

concretize

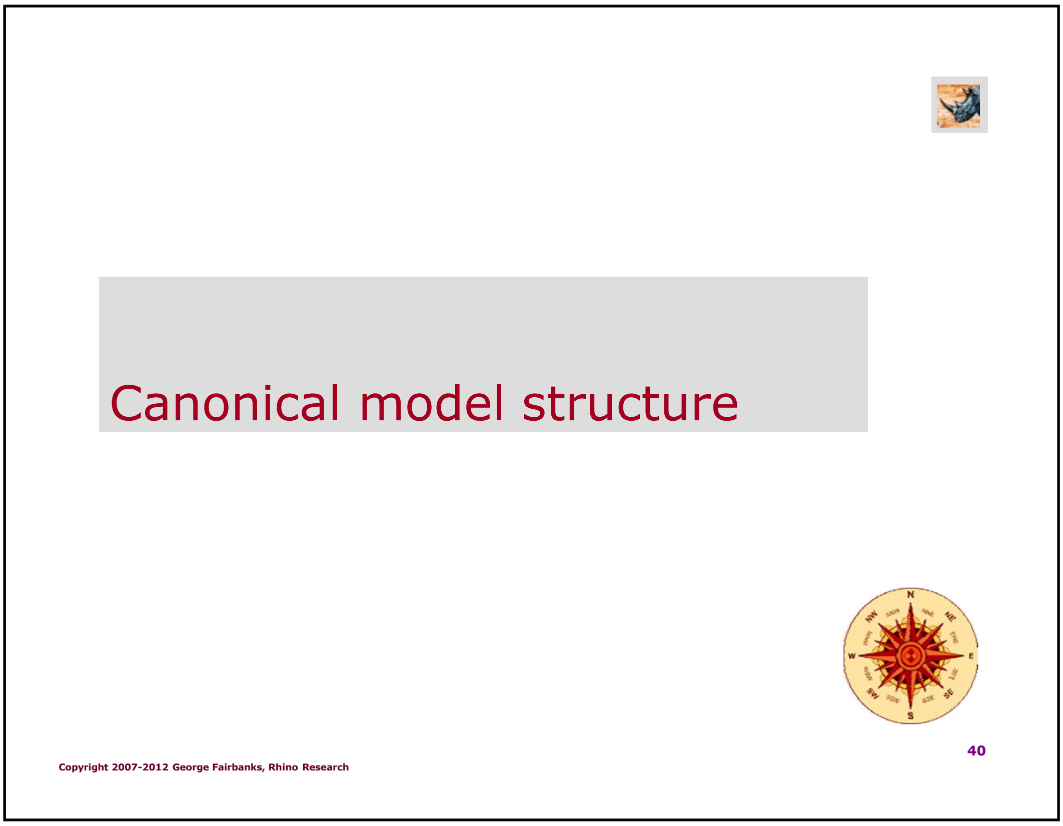
Real-world problem

solve

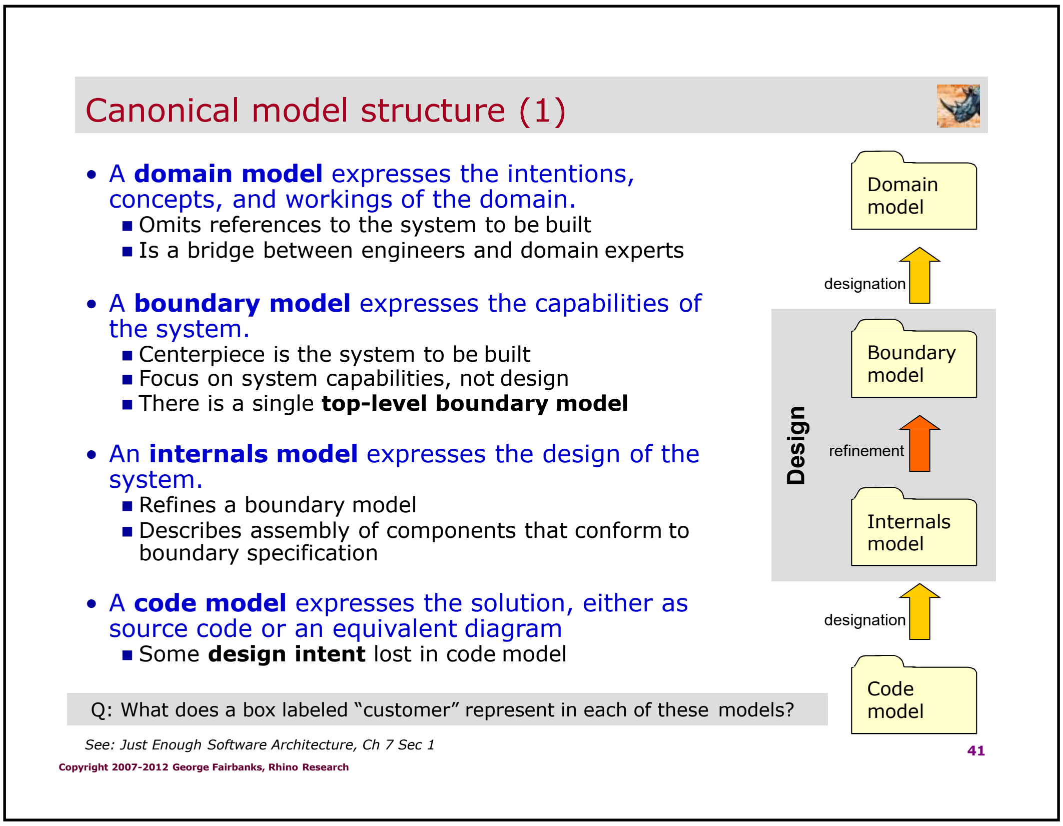
Real-world solution

"A train is traveling south at 10m/s. Another departs 30 minutes later at 15m/s. When do they meet?"

See: Just Enough Software Architecture, Ch6



Canonical model structure

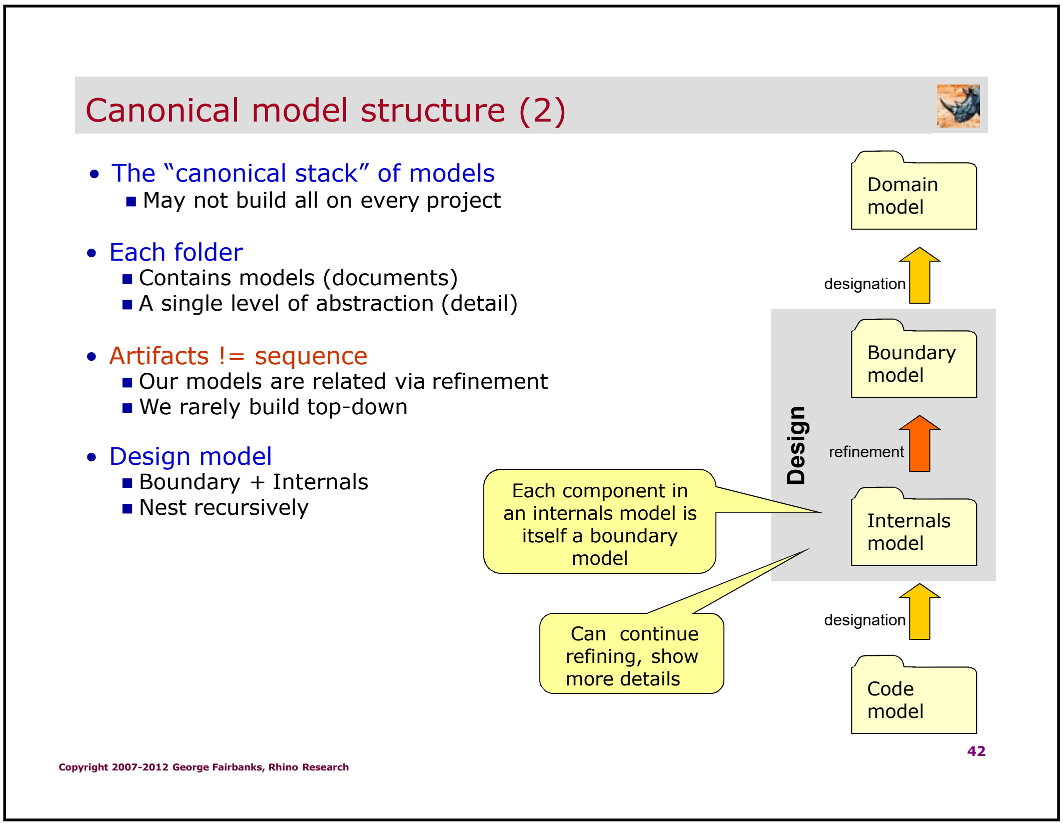


Canonical model structure (1)

- A **domain model** expresses the intentions, concepts, and workings of the domain.
  - Omits references to the system to be built
  - Is a bridge between engineers and domain experts
- A **boundary model** expresses the capabilities of the system.
  - Centerpiece is the system to be built
  - Focus on system capabilities, not design
  - There is a single **top-level boundary model**
- An **internals model** expresses the design of the system.
  - Refines a boundary model
  - Describes assembly of components that conform to boundary specification
- A **code model** expresses the solution, either as source code or an equivalent diagram
  - Some **design intent** lost in code model

Q: What does a box labeled "customer" represent in each of these models?

See: Just Enough Software Architecture, Ch 7 Sec 1



Canonical model structure (2)

- The "canonical stack" of models
  - May not build all on every project
- Each folder
  - Contains models (documents)
  - A single level of abstraction (detail)
- Artifacts != sequence
  - Our models are related via refinement
  - We rarely build top-down
- Design model
  - Boundary + Internals
  - Nest recursively

Domain model
designation
Design
Boundary model
refinement
Internals model
designation
Code model

Each component in an internals model is itself a boundary model

Can continue refining, show more details

Models and Code

Architecture vs code – different things easy to see

- When reading code, want to know:
  - Who talks to who
  - Invariants and constraints
  - Messages sent and received
  - Styles and patterns
  - Performance requirements or guarantees
  - Data structures used for communication
  - Etc.
- Easy to see in architecture model, hard to see in code
- Why?
  - A single object rarely has a big impact on QA's
  - Cannot infer design from code
    - e.g., "never call A from B", "always do X before Y"
- Yet
  - Code-level decisions "bubble up" into QA's
  - Architecture decisions directly influence QA's

See: Just Enough Software Architecture, Ch10

Architecturally evident coding style

- Current practice
  - Provide hints useful to humans
  - Use “totalExpenses” instead of just “t” variable
  - Intention revealing method names
- Idea: Express architectural ideas
  - Provide hints about architecture
  - Do more than is necessary for program to compile
  - Preserve design intent
- Benefits
  - Avoid future code evolution problems
  - Improve developer efficiency
    - Reduce time spent inferring from code
  - Lower documentation burden
  - Improve new developer ramp-up

See: Just Enough Software Architecture, Ch 10 Sec 3

Process & Risk

Engineering failures

> *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 implicit 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.*
> *[Henry Petroski, Design Paradigms, 1994]*

Required

- Considering failures
- Analyzing options
- Designing a solution

You can choose

- When design happens
- Which analyses
- Formality / precision
- Depth

See: Just Enough Software Architecture, Ch3

Insight #1: Decide effort using risks

- At any given moment, you have worries and non-worries
  - Worry: Will the server scale up?
  - Worry: Will bad guys steal customer data?
  - Response time will be easy to achieve
  - We have plenty of RAM
- Cast these worries as engineering risks
  - Focus on highest priority risks
- Good news: **prioritizing risks is easy for developers**
  - They can tell you what they are most worried about
  - I.e., possible failures

See: Just Enough Software Architecture, Ch 3 Sec 4

Insight #2: Techniques mitigate risks

- Many architecture techniques exist
  - Protocol analysis
  - Component and connector modeling
  - Queuing theory
  - Schedulability analysis
  - Threat modeling
  - …
- Techniques are not interchangeable
  - E.g., cannot use threat modeling on latency risks
- So, must **match risks with techniques**
  - I.e., mapping from risks → techniques
  - Inspired by Attribute Driven Design (ADD)

See: Just Enough Software Architecture, Ch 3 Sec 4

Risk-Driven Model

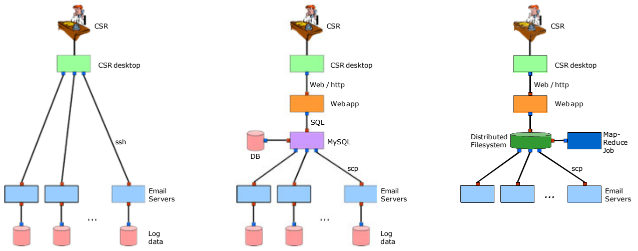
- **The Risk-Driven Model:**
  1. Identify and prioritize risks
  2. Apply relevant architecture activities
  3. Re-evaluate
- **Must balance**
  - Wasting time on low-impact techniques
  - Ignoring project-threatening risks

Ron ArmsCtrong, CC

See: Just Enough Software Architecture, Ch 3 Sec 4

Process, risk, and Rackspace example

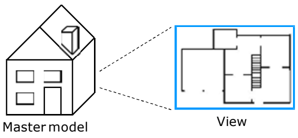
- Agile/lean architecture
  - Agile processes: few design techniques
  - Architecture: many design techniques
  - Use the risk-driven model to combine
- Rackspace: Did they proceed rationally?
  - Should they have done Big Design Up Front (BDUF)?
  - Should they have evolved the architecture?
  - What risks did they face?

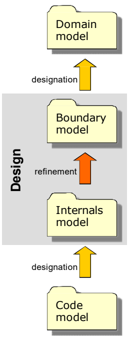


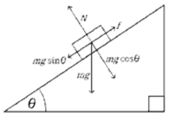
Version 1   Version 2   Version 3

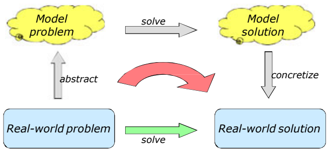
Summary

- Architecture, architecting, architects
- Views
- Quality attributes
- Analysis
- Standard notations
- Guiderails
- Architectural styles
- Conceptual model
- Engineering with models
- Canonical model structure
- Models and code
- Process and risk


Master model
View


Domain model
designation
Boundary model
refinement
Internals model
Design
designation
Code model


N
f
mg sinθ
mg cosθ
mg
θ


Model problem
solve
Model solution
abstract
concretize
Real-world problem
solve
Real-world solution