Architecture Hoisting

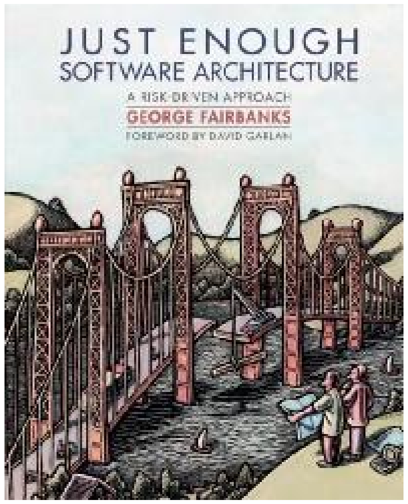
George Fairbanks

14 March 2012

Rhino Research

Software Architecture Consulting and Training

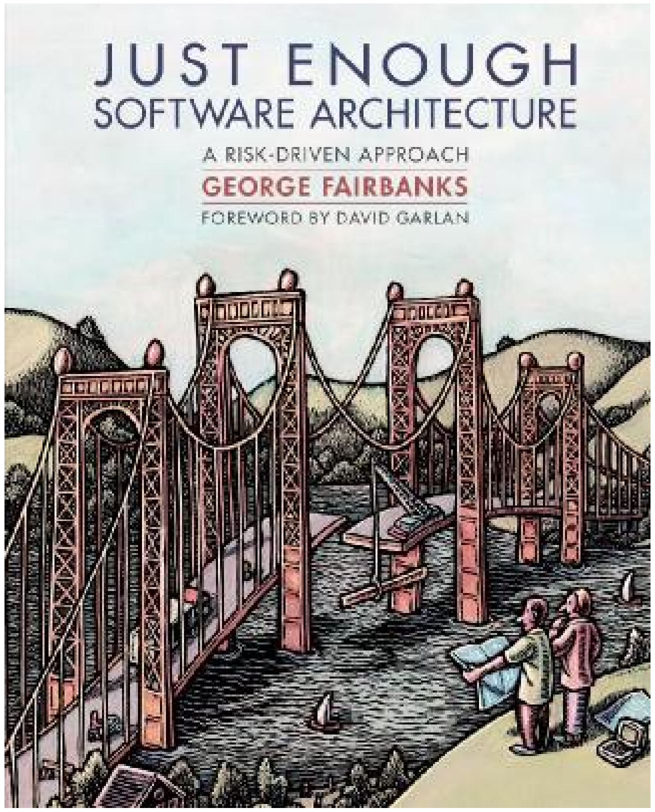
http://RhinoResearch.com



About me (George Fairbanks)

- PhD Software Engineering, Carnegie Mellon University
- Thesis on frameworks and static analysis (Garlan & Scherlis advisors)
- **Program chair: SATURN 2012**; Program committee member: WICSA 2009, ECSA 2010, ICSM 2009; CompArch 2011 local chair
- Architecture and design work at big financial companies, Nortel, Time Warner, others
- Teacher of software architecture, design, OO analysis / design

- Author: ***Just Enough Software Architecture***



Talk preview

- Design is about enabling our reasoning / analysis
- Different kinds of design ideas: extensional and intensional
  - Extension maps straight to code
  - Intensional does not, and so causes all the trouble
  - QAs usually related to intensional design ideas
- In the small:
  - Vigilance can maintain invariants
- In the large:
  - Vigilance can maintain architecture guiderails
  - … but it's hard and error-prone
  - Complexity grows with scale
  - So we may choose architecture hoisting

A story of two unhappy developers

Unhappy developer 1

- Imagine a developer grumbling about using a framework:

  "This application framework stinks! It constrains what I can do, it's bureaucratic, and worse: it forces me to use Java!"

Unhappy developer 2

- Pwn to Own, bugs in Chrome browser allowed remote execution of code, March 2012
- Exploit based on "Use after free" bug
- Hard to avoid such bugs

NASA Mission Data System

- Spaceflight software development is conservative
- It is also hard
- Requires careful analysis of the mission, the software, and risks
  - Systems Engineers build detailed models
  - Old: Software, written in C, to run the mission
  - Old: Impenetrable mapping from SE à C code
    - E.g., variable for spacecraft velocity in local variable

- MDS
  - Clear mapping from SE models to software
  - Declarative rules about topology à control theory analysis
  - "Hoisted" many problems into the architecture
  - Dan Dvorak, Kirk Reinholtz, Nicholas Roquette, Kenny Meyer

- Outcome
  - Funding issues with NASA
  - Still writing mission code in C

Servlets and EJB

- With servers, concurrency is inherent
- Server = shared code + request handlers
- Requirement: safe concurrency

- Servlets: Guideline to write re-entrant handlers
  - Developers must be **vigilant**

- EJB: App server never overlaps use of handlers
  - For all requests, for any two requests x and y, overlapInTime(x, y) à handler(x) != handler(y)
  - Developers mostly oblivious to concurrency

- Tradeoff with performance
  - In many cases, re-entrant servlets will be faster

Approach to architecture; hoisting

- Architecture Hoisting:
  - Giving architecture the specific responsibility to handle a QA requirement or property
  - Or emergent behavior of architecture ensures QA/property
  - Example: EBJ hoists concurrency concern
- Architecture-Focused: Consciously choosing architecture to suit the QA requirements
- Architecture Indifferent: Opposite of architecture-focused
- Note: Runtime presence & hoisting
  - Hoisting usually requires runtime presence of architecture
  - E.g., message bus to guarantee durable message delivery, app server to guarantee request handling properties
- This catches you up with what’s in the book

Let’s understand hoisting better

- Unsatisfied because:
    - Nature poorly understood
    - Definition not clear
    - Applicability: what is hoisting good for?
        - Bad: features; good: QAs, properties
        - … but that’s pretty fuzzy

- This talk will discuss
    - What architecture hoisting is
    - When it is applicable
    - What alternatives you have
    - Background: the conceptual model you need

- To get there, let’s enrich our conceptual model
    - Intensional / extensional; constraints / guiderails

Intensional and extensional design intent

- Intensional: elements are universally quantified
  - For all; e.g., all filters can communicate via pipes

- Extensional: elements are enumerated or named
  - E.g., the system is composed of a client, an order processor, and an order storage components

- Architecture and design:
  - Mixture of intensional and extensional elements
  - (Next slide elaborates)

- Code:
  - Only extensional elements
  - Difficult or impossible to express intensional ideas
    - E.g., invariants, responsibility allocations, protocols
  - Will follow invariants, but cannot express them

Credit: Amnon Eden and Rick Kazman

Architecture vs. code: intensional / extensional

- Code: hard / impossible to show intensional
- Code will show examples of intensional rule

	Architecture model element	Translation into code
Extensional (defined by enumerated instances)	Modules, components, connectors, ports, component assemblies	These correspond neatly to elements in the implementation, though at a zoomed-out higher level of abstraction (e.g., one component corresponds to multiple classes)
Intensional (quantified across all instances)	Styles, invariants, responsibility allocations, design decisions, rationale, protocols, quality attributes and models (e.g., security policies, concurrency models)	Implementation will conform to these, but they are not directly expressed in the code. Architecture model has general rule, code has examples.

Invariants and guiderails are intensional

- Invariants, constraints, and guiderails are intensional

- In the small: invariants
  - E.g., invariant on linked list data structure
  - Static and dynamic invariants
  - Normal tactic: developer vigilance
  - Q: What must you do/ensure when editing the linked list code?

- In the large: constraints / guiderails
  - Normal tactic: developer vigilance (but at much larger scale)
  - Guiderails are self-imposed design constraints

- Consequence?
  - Hard to express them in code
  - Easy to break them during maintenance

Why use invariants / constraints / guiderails?

- Wait a second, why are we imposing constraints anyway?

- Hypothesis: **The goal of design is to enable reasoning**

- Thought experiment:
  - Start with comprehensible / analyzable design
  - Progressively transform it so that it computes same answer but is harder and harder to understand (design obfuscator)
  - What we've removed is the part that helps us reason

- Problem: Emergent properties / QAs / intensional intent hard to design and reason about (complexity)
- Solution: **Use invariants/guiderails to reduce complexity**
  - They improve our reasoning abilities
  - Important in architecture, where scale is bigger

Memorize this list

January 4

February 1

March 7

April 4

May 2

June 6

July 4

August 1

September 5

October 3

November 7

December 5

Memorize this constraint / guiderail

The first Wednesday of each month in 2012

Guiderail implementation options

- Global requirement
  - E.g., Handle concurrency safely (no race conditions)

- Q: How to achieve the global requirement?
- A: Use a guiderail

- Guiderail option 1: Design guidance + developer vigilance
  - "Always write thread-safe handlers"

- Guiderail option 2: Hoist the guiderail
  - Architecture ensures no handler interactions

- Guiderail option 3: Static analysis
  - Assuming it exists
  - Or you can write one (accurately)

Design technique: architecture hoisting

- How do I hoist something into the architecture?

- Architectural hoisting design technique:

  1. Find the "forall" in the global property
  2. Use an architectural mechanism to ensure that all legal architectures embody the property

- This transforms the problem
  - From: programming in the large
  - To: programming in the small

- Coming up: what are the mechanisms?

Image from Erik B., pizdaus.com, CC Attribution license

Recap: where we are

- Design is about enabling our reasoning / analysis

- Different kinds of design ideas: extensional and intensional
  - Extension maps straight to code
  - Intensional does not, and so causes all the trouble
  - QAs usually related to intensional design ideas

- In the small:
  - Vigilance can maintain invariants

- In the large:
  - Vigilance can maintain architecture guiderails
  - … but it's hard and error-prone
  - Complexity grows with scale
  - So we may choose architecture hoisting

Definition: Architecture hoisting

- Architecture hoisting is a design technique where the architecture ensures an intensional design constraint (i.e., a guiderail), thereby achieving a global property.

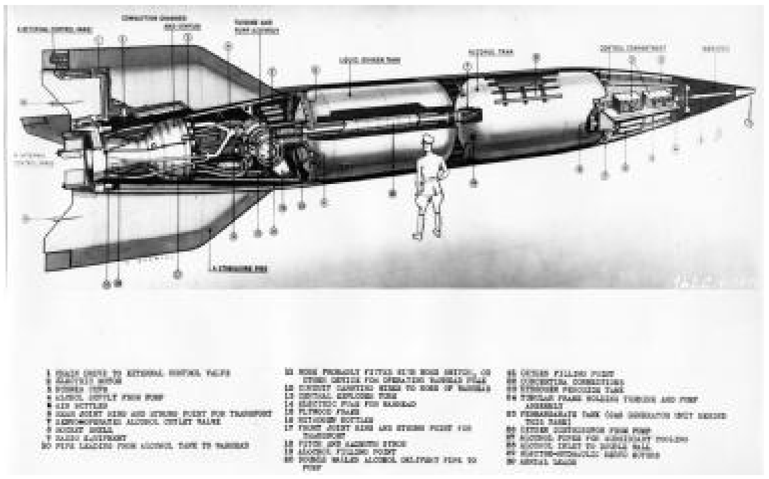
- It is more than:
  - Architecture providing some shared services
  - Architecture providing some structure
- … as these are common without ensuring a design constraint holds

- Hoisting is closely tied to intensional design intent
  - Hoisting is one option to achieve it
  - (the other being developer vigilance)

Image from Erik B., pizdaus.com, CC Attribution license

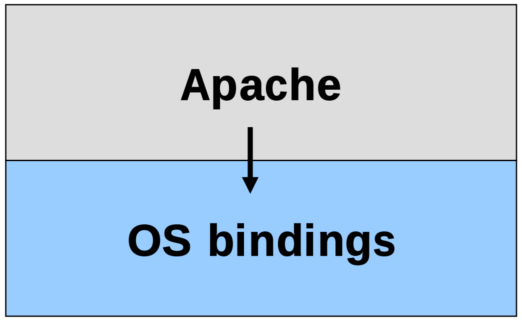
Mechanisms to implement hoisting

- Layer / Virtual Machine
- Language + compiler / interpreter / runtime
    - E.g., type system, garbage collection
- Domain-Specific Language
- Framework
- API
    - E.g, add(key,value)
- Type system
- Library



Example: Apache Portable Runtime

- The Apache web server runs on several platforms
  - Linux, Mac, Windows, ...
- To satisfy that requirement, they built the APR
  - Apache Portable Runtime (APR)
  - Cross-platform library/layer for sockets, files, etc.
- The web server depends on the APR
  - No platform-specific code in the web server
  - Web server calls should go through the APR



Test case: XBMC cross-platform plugins

- XBMC is a cross-platform media player
    - Extension mechanism: plugins
    - Plugins written in Python

- Problem: Plugins cross-platform
- Solution: Guidelines for safe cross-platform Python

- Q1: Is this hoisting?
- Q2: What can it guarantee?

- Perhaps not all hoisting guarantees the property?
    - ... and compare with the APR hoisting?

Degrees of strength

- Guarantee
  - Hard because of general-purpose programming languages
  - Opportunities to break constraints
  - Better test: Will it hold with well-intentioned developers?
  - Example: EJB, other app servers

- Affordance
  - Examples: Provide a platform-independent library (Apache Portable Runtime)

- Guideline / best practice
  - More like non-hoisting. Depends heavily on vigilance

- Discussion
  - Even the “guarantee” strength requires developer vigilance, if only to conform to the framework or guidelines or API.
  - Mechanisms with relatively stronger guarantees: Frameworks with inversion of control; languages; DSL; virtual machine

Design guidance: obstacles

- Do you recognize the requirement?
- Do you know at least one way to achieve it?
- If you provide design guidance, how do you communicate it?
- Once you've provided guidance, how do you ensure it's followed?
- Does vigilance ensure success?
  - Even with good faith by developers, code paths are complex
  - Humans are bad at this kind of reasoning

- Example: Pwn to Own, Chrome browser, March 2012
  - Exploit based on “Use after free” bug
  - I.e., there was a broken intensional design constraint

- **Conclusion: Healthy skepticism of design guidance**

Back to the grumbling framework developer

- I hear that you are frustrated with using the web framework (re your web posting titled “frameworks suck”)
- Our group is using that framework so that we can satisfy a requirement to uphold a global property: all web inputs must be sanitized.
- The framework provides a single, hoisted implementation for request processing and ensures the intensional design constraint (a guiderail) is always maintained.
- The alternative is that all developers must be constantly vigilant in upholding that property and must be able to reason through every possible code path, identifying and eliminating paths where the design constraint could be broken.

à Compare with the grumbling Chrome developer

Sociology

- The developer may be right
    - The framework might stink
    - It might be a bad design
- Example: EJB 1, 2, 3
    - EJB 1, 2, & 3: Concurrency restrictions: win
    - EJB 1 & 2 syntax: fail
    - EJB 1 & 2 persistence: fail
- Hoisting is a design technique, not a silver bullet
    - It is a strait-jacket for developers
    - Communicate your rationale to them

Tradeoffs of using hoisting

- Standardized, hoisted solution:
    - Suitable for some applications
    - ... unsuitable for others
- Example
    - Building garbage collection into language
    - ... harder to build realtime applications
- Disallows local choices
    - Cannot resolve/optimize the tradeoff locally

Ron ArmsCtrong, CC

Conceptual models

- So, hoisting is often a good solution
  - ... but other times not
  - Did we learn anything?

- Yes! We enriched our **conceptual model of software design**
  - This is the essence of engineering knowledge
    - Techniques have pros/cons, applicability
  - How was the fwk developer's conceptual model impoverished?
  - What concepts and connections was it missing?

- Big goal: Next generation of developers is better than the prior

- How to win
  - Improve our conceptual models
  - Making connections
  - Being able to teach them

Reflection

- Many of us have never built something at the fringe of engineering
- Examples
  - A safe web browser
  - Reliable space software

- The best developers using the best techniques: fail

- In this context, consider:
  1. The state of the art of software design techniques
  2. Static analysis and statically typed languages
  3. Architecture hoisting

Summary

- Architecture hoisting is a design technique where the architecture ensures an intensional design constraint (i.e., a guiderail), thereby achieving a global property.
  - Suitable for intensional design intent, not extensional

- Intensional constraints are hard to ensure in code
  - Option 1: developer vigilance
  - Option 2: architecture hoisting

- Mechanisms:
  - Framework, language, DSL, API, library, layer / virtual machine

- Good: Enforce an intensional design constraint
- Mixed: Sociology
- Bad: Tradeoffs because we standardize the solution

Shameless plugging

- E-book
  - One price, 3 formats
  - PDF, ePub, Mobi
  - No DRM
  - RhinoResearch.com $19.50

- Hardback
  - Amazon.com $19.50