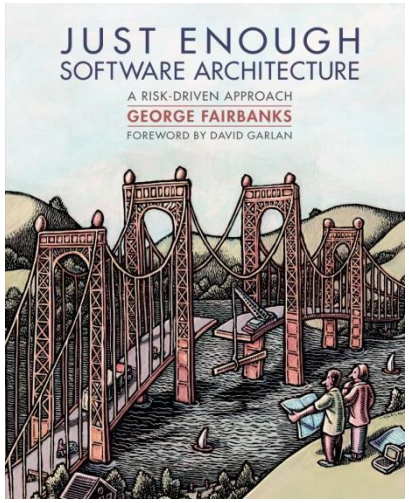
Architectural Hoisting

George Fairbanks

SATURN 2014
6 May 2014



Rhino Research
Software Architecture Consulting and Training
http://RhinoResearch.com

Talk summary

- **Architectural hoisting** is:
  - … a new name for an old technique
  - … the enforcement of a constraint with code mechanisms

- Mechanisms to enforce constraints:
  - Language runtime, DSLs, frameworks, VMs, layers, libraries

- PITS: **vigilance** can maintain invariants
- PITL: complexity grows with scale
  - So we often choose architectural hoisting

- Benefits
  - Reveals similarity between PITS and PITL techniques
  - Highlights difference in enforcement
  - Clarifies role of architecture

Part 1: The Basics

Example 1: Servlets and EJB

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

- **Servlets**: Developers ensure re-entrant handlers
  - Developers must be **vigilant** about concurrency
  - (Servlet container default)

- **EJB**: App server never overlaps use of handlers
  - EJB developers can mostly ignore concurrency

- Tradeoffs
  - In many cases, re-entrant servlets will be faster
  - Developers may make mistakes with Servlets

Define: Vigilance

Think of the guard in a castle.
Bad things happen if he sleeps.

dil·i·gence[1]
/'dilәjәns/

noun

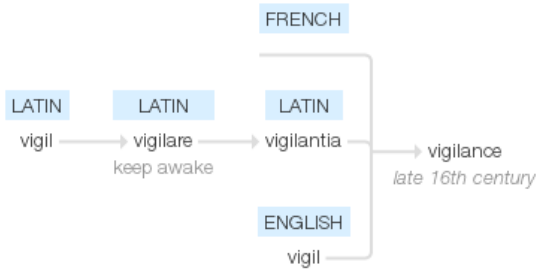
1. careful and persistent work or effort.
   synonyms: conscientiousness, assiduousness, assiduity, hard work, application, concentration, effort, care, industriousness, rigor, meticulousness, thoroughness; More

vig·i·lance
/'vijәlәns/

noun
noun: **vigilance**

1. the action or state of keeping careful watch for possible danger or difficulties.

Origin



late 16th century: from French, or from Latin *vigilantia*, from *vigilare* 'keep awake,' from *vigil* (see vigil).

Translate vigilance to Choose language

Use over time for: vigilance

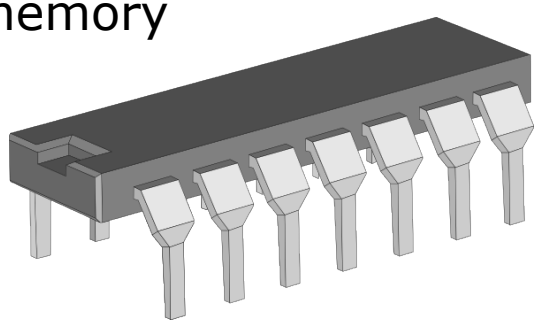
Example 2: Memory (resource) management

- All programs handle memory
- (Similar for all resource management)
  - Fonts, handles, connections, …

- **C / C++**
  - Developers must be **vigilant** to allocate / free memory

- **Java**
  - Runtime garbage collection

- Tradeoffs
  - Garbage collection yields pauses in execution
  - Developers may make mistakes and leak memory



Questions

- What are some examples of **vigilance** in your life?
- What are some examples of **vigilance** in software?
- Can you think of places where vigilance has been, or could be, replaced by automation or some other structure?

??

Two paths to the same goal

Same goals, different paths

- Goal: Safe web concurrency
  - Path 1: Servlets
  - Path 2: EJB

- Goal: Memory management
  - Path 1: Manual
  - Path 2: Garbage collection

Two paths to the same goal

Same goals, different paths

- Goal: Safe web concurrency
  - Path 1: Servlets
  - Path 2: EJB
- Goal: Memory management
  - Path 1: Manual
  - Path 2: Garbage collection

Not obvious, but critical

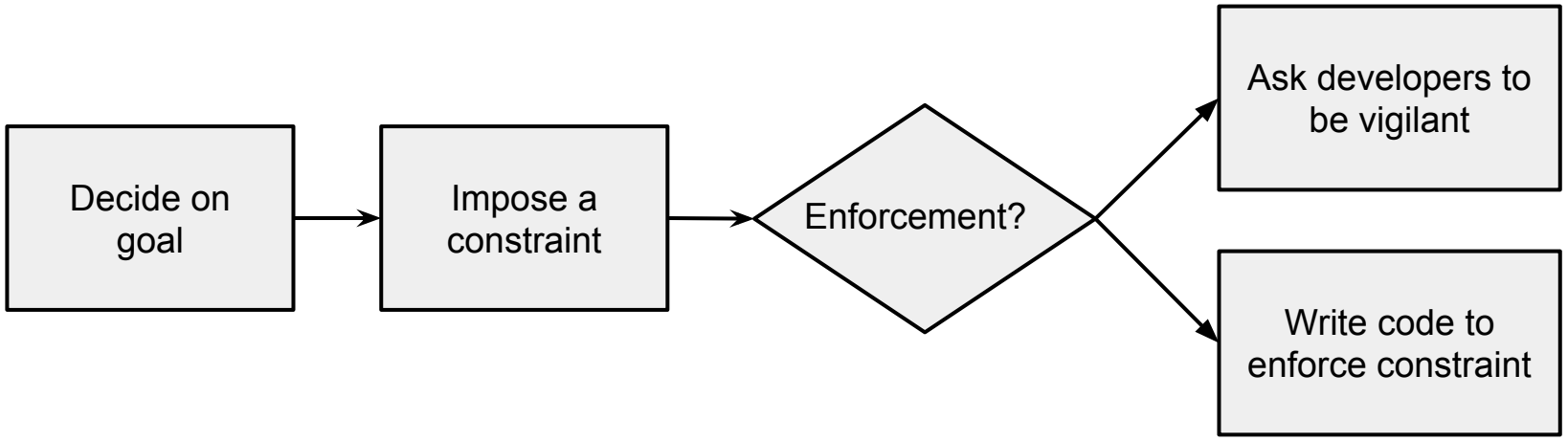
Observations

1. Same goals but different paths
2. There's a **constraint** hiding behind the goal
   a. No two requests that overlap in time share the same handler
   b. Every "alloc" eventually is paired with a "free"
3. Paths differ in enforcement
   a. Vigilance vs code

Summary: Decision process

You want a system that achieves a goal, so:

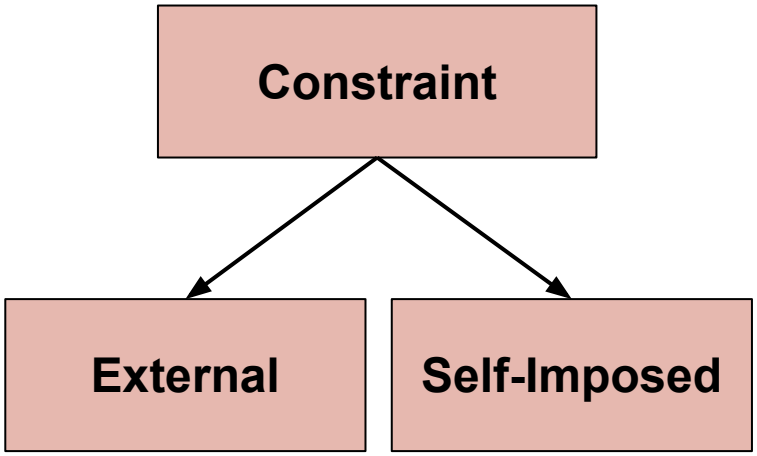
1. You impose a constraint
2. You choose how to enforce
3. **Option 1**: Tell developers about the constraint
   a. E.g., Servlets / manual memory management
4. **Option 2**: Write code to enforce the constraint
   a. E.g., EJB / garbage collection



Constraints

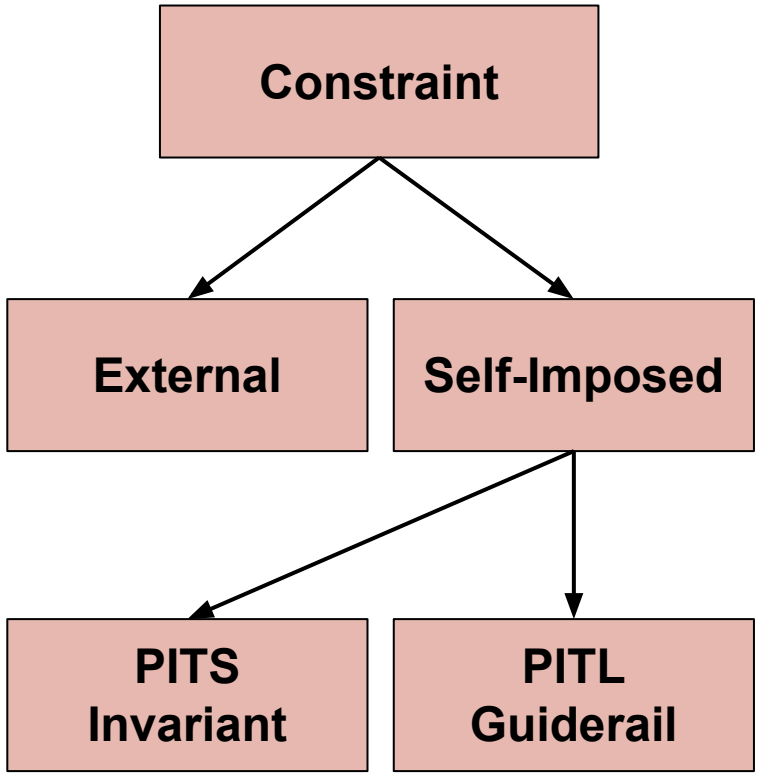
Constraints, invariants, guiderails

- What are the differences between these similar terms?
- **External** = annoying constraint imposed by someone else



Constraints, invariants, guiderails

- What are the differences between these similar terms?
- **External** = annoying constraint imposed by someone else
- A **guiderail** is an architectural constraint that is self-imposed to make the system easier to build or reason about.
- **PITS** / **PITL** = Programming in the Small Programming in the Large
  DeRemer & Kron 1975



Questions

- What are some examples of **externally imposed constraints**?
- What are some PITS **invariants**?
- What are some PITL **guiderails**?

??

Why use constraints / invariants / 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 (a 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 1
February 5
March 5
April 2
May 7
June 4
July 2
August 6
September 3
October 1
November 5
December 3

Chunking and recall

- Story of student recalling long strings of digits
  - Strategy: chunking
    - Famous race times, e.g., "4:47" time
    - Each time is a chunk
  - Long string of digits → shorter string of chunks

- For those that made progress on memorizing the dates, probably recognized
  - … months in sequence
  - … digit was 1-7

- **General lesson**
  - Recognize pattern → **lower cognitive burden**
    - (Patterns and constraints are kissing cousins)
  - Lower cognitive burden → **easier to reason**

Enforcing invariants and guiderails

What does it mean to enforce a constraint (or guiderail)?

- They are not made from steel
- They can be made of spoken words, written words, or code

Guiderail enforcement mechanisms

- 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**
  - E.g., always write thread-safe handlers
  - I'll use "vigilance" as shorthand for "guidance + vigilance"

- **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)

Intensional & extensional

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

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.

Questions

- Are the following **intensional** or **extensional**?

1. All deliveries go to back of building.
2. Sunsets are beautiful.
3. Portland is beautiful.
4. Charles is William's brother.

??

Invariants and guiderails are intensional

- Invariants, constraints, and guiderails are usually **intensional**

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

- **PITL**: guiderails
  - Normal enforcement: developer vigilance (at much larger scale)
  - Alternative: **guiderails**

- Because they are intensional
  - Hard to express them in code
- If enforcement is **vigilance**
  - Easy to break them during maintenance

Architectural hoisting

Architectural hoisting

- Hoisting is not a new design technique
- You have been doing it for years
- You likely did not recognize it
- Now you will see it all the time
  - Fnord reference

NASA JPL Mission Data System (MDS)

- 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: Inscrutable mapping from SE → C code
    - E.g., variable for spacecraft velocity in local variable

- MDS
  - Clear mapping from SE models to software
  - Declarative rules about topology → 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

MDS website

Definition: Architectural hoisting

- **Architectural hoisting** is
  - the enforcement of a guiderail with code mechanisms.
  - a design technique where the architecture ensures an intensional design constraint (i.e., a guiderail), thereby achieving a global property.

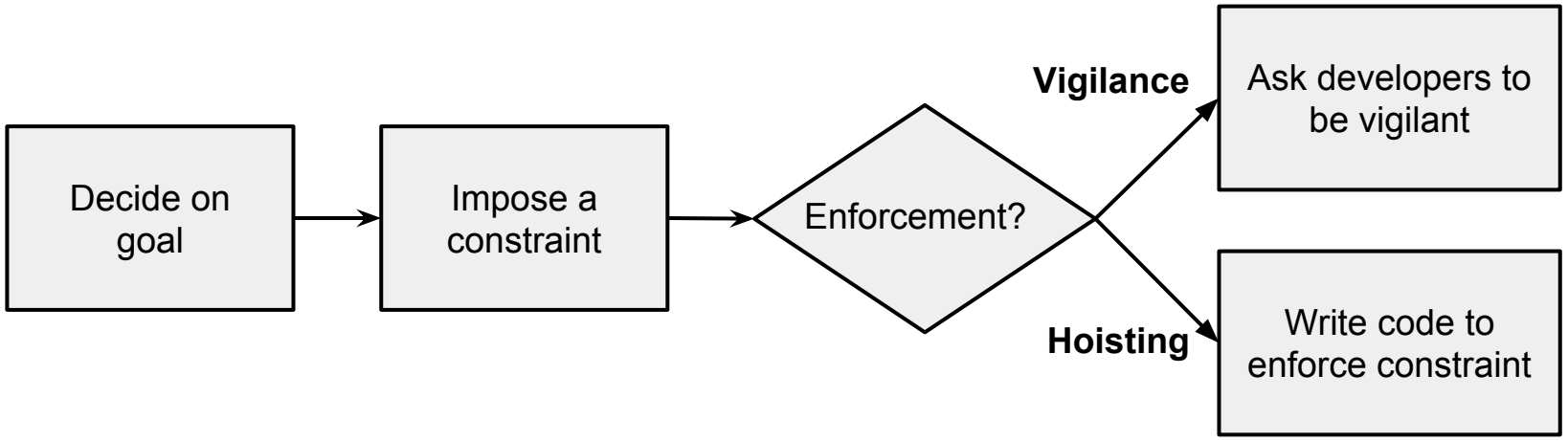
- Hoisting is closely tied to **intensional design intent**
  - Hoisting is one option to achieve it
  - … the other being developer vigilance

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

Updated: Hoisting decision process

You want a system that achieves a goal, so:

1. You impose a constraint (**a guiderail**)
2. You choose how to enforce
3. **Vigilance**: Tell developers about the constraint
   a. E.g., Servlets / manual memory management
4. **Hoisting**: Write code to enforce the constraint
   a. E.g., EJB / garbage collection



Design technique: architectural hoisting

- How do I hoist something into the architecture?

- Architectural hoisting design technique:

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

Enforcing a guiderail: Degrees of strength

Guarantee

- Guarantees hard with general-purpose programming languages
- Examples: Language runtime, DSL.

Buttress

- Provided that developers are non-malicious, will the property hold?
- Examples: Frameworks with inversion of control.

Affordance

- Well-intentioned developers can follow the guideline with ease.
- Examples: Provide a platform-independent library (e.g., APR)

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

Note: Even the "guarantee" strength requires developer vigilance, if only to conform to the framework or guidelines or API.

Hoisting mechanisms

Language runtime

- Eg: array bounds checking, pointers, garbage collection
- Strength: Strong (Guarantee)

DSLs (Domain Specific Languages)

- Eg: Struts configuration files, business rules engine
- Strength: Strong (Guarantee)

Frameworks

- Eg: Eclipse, Ruby on Rails
- Strength: Medium (Buttress)

VMs and architectural layers

- Eg: web browser sandboxing
- Strength: Medium to Strong (Buttress to Guarantee)

Libraries (modules) and runtime services (components)

- Eg: Authentication service, data access object, hashtable API
- Strength: Weak (Affordance)

Hoisting: Is this simply architecture?

- Every system has an architecture, sometimes by **accident**
  - All architectures provide structure
  - Many provide shared services
  - So, is this not hoisting?

- Hoisting is **always intentional**
  - You recognize a goal
  - You impose an intensional constraint
  - You choose an enforcement mechanism

- If you look back, you'll probably find you hoisted
  - You may not have thought so rigorously

=?

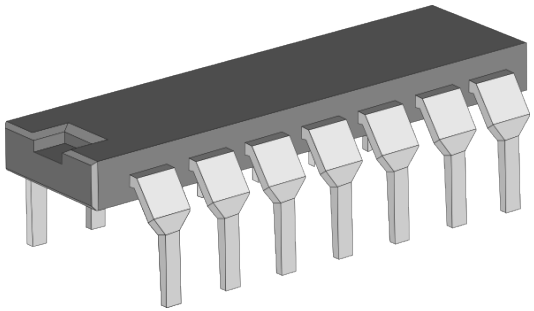
Examples

Example 1 reprise: Safe web concurrency

- Let's look at the Servlets / EJB example again
- What is the **goal**?
  - All requests must be handled in threadsafe manner.
- What is the **guiderail**?
  - For all requests, for any two requests x and y, overlapInTime(x, y) → handler(x) != handler(y)
- What are the **mechanism** options?
  - Servlets: vigilance
  - EJB: application framework
  - Note that both Servlets and EJB are app frameworks, but the Servlets fwk does not hoist concurrency concerns.

Example 2: Reprise memory/resource management

- Let's look at the memory management example again
- What is the **goal**?
  - No memory/resource leaks.
- What is the **guiderail**?
  - Every allocation always eventually has a matching deallocation.
- What are the **mechanism** options?
  - Vigilance: Allocation/deallocation patterns
  - Library/service: resource counting / smart pointers
  - Language runtime: garbage collector



Example 3: Cross-platform plugins

XBMC cross-platform plugins

- Plugins/modules in Python
- Who: General developers
- Hoisting mechanism: Vigilance (via message board; no centralized advice)
- Effectiveness: Poor

Apache web server modules

- Plugins/modules in C
- Who: Sophisticated developers
- Hoisting mechanism: provided cross-platform library (Apache Portable Runtime)
- Effectiveness: Good

XBMC cross-platform plugins

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

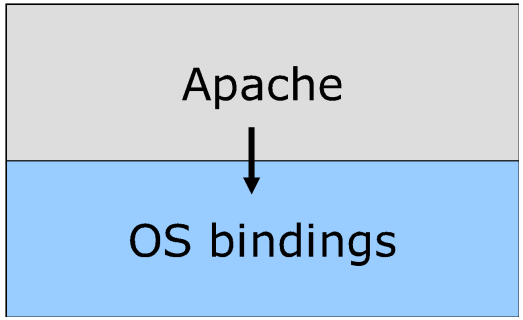
- Problem: Plugins cross-platform
- Solution: Guidelines for safe cross-platform Python

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

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

Apache Modules & 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

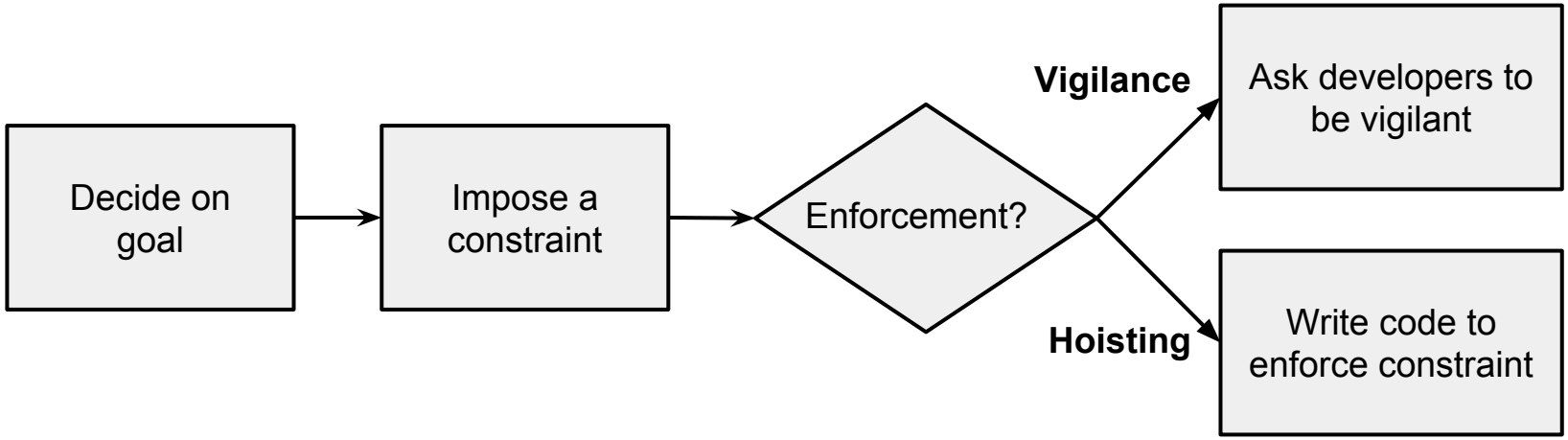


Summary

Recap: where we are

- Programming in the small (PITS):
  - **Vigilance** can maintain invariants

- Programming in the large (PITL):
  - Vigilance can maintain architecture **guiderails**
  - … but it's hard and error-prone
  - Complexity grows with scale
  - So we may choose **architectural hoisting**



Part 1 Exercise

Exercise

For each of these design mechanisms:

- Language runtime
- DSLs
- Frameworks
- VMs and architectural layers
- Libraries (modules) and runtime services (components)

Write down:

1. **Guiderail**: an example of a guiderail it could be used to enforce
2. **Story**: come up with an example of how it can be used to hoist
3. **Enforcement**: how much strength it has in enforcing that example
4. **Alternatives**: Other ways to enforce the guiderail
5. **Tradeoffs**: Evaluate your proposal vs. the alternatives

Use the following pages as a workbook.

Exercise

Mechanism: Language runtime

Guiderail

Story

Enforcement

Alternatives

Tradeoffs

Exercise

Mechanism: Domain Specific Language (DSL)

Guiderail

Story

Enforcement

Alternatives

Tradeoffs

Exercise

Mechanism: Application Framework

Guiderail

Story

Enforcement

Alternatives

Tradeoffs

Exercise

Mechanism: Virtual Machine / Architectural Layer

Guiderail

Story

Enforcement

Alternatives

Tradeoffs

Exercise

Mechanism: Library (module) / Runtime service (component)

Guiderail

Story

Enforcement

Alternatives

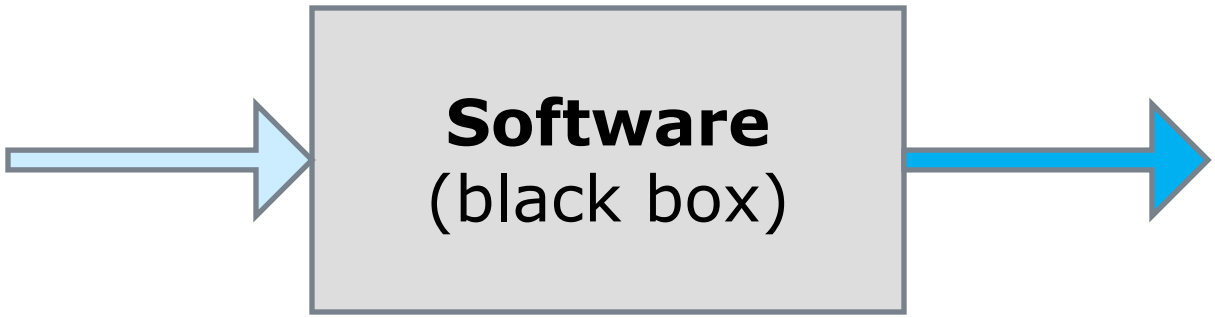
Tradeoffs

Part 2: Advanced

Software design

What is design about? (Blackbox level)

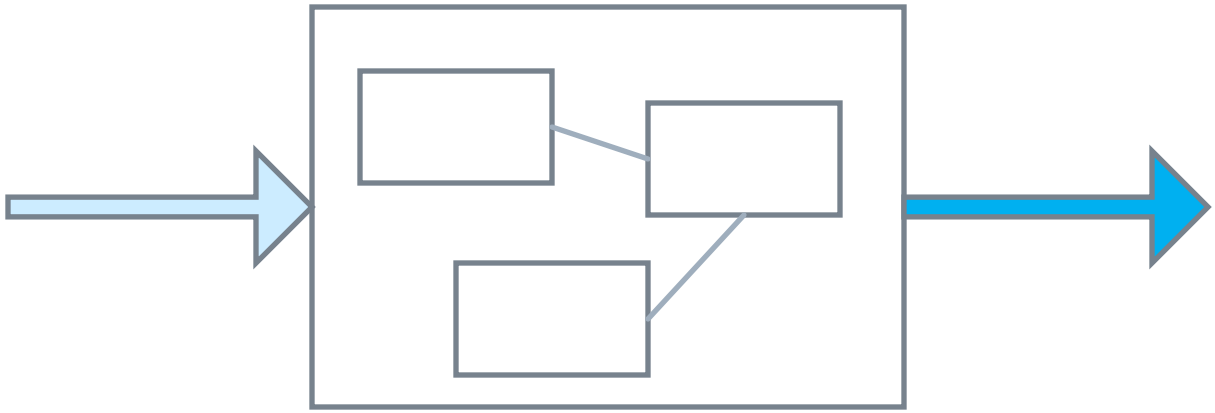
- It must behave as desired
  - Functionality
  - Quality attributes



- But we don't really trust that **black boxes** work
  - Why not?
  - Empirical results vs. general trust

What is design about? (Whitebox level)

- To have confidence it works
  - Must be able to reason about its design
  - No reason = faith



- Now we must evaluate it as a **whitebox**
  - Understand its parts
  - Understand their interactions

Reasoning through a proposed solution

- Hypothesis: **The goal of design is to reason through an abstract solution that is intended to solve the problem**
  - Designers never set out to build a video game and accidentally build a thermostat
  - They must be (overall, informally) reasoning about what the system will do (functionality and qualities)

- Corollary: **Designs that cannot be analyzed are useless**
  - (Except output of machine learning or similar)
  - Also: consider a music visualizer – stumble across a good design

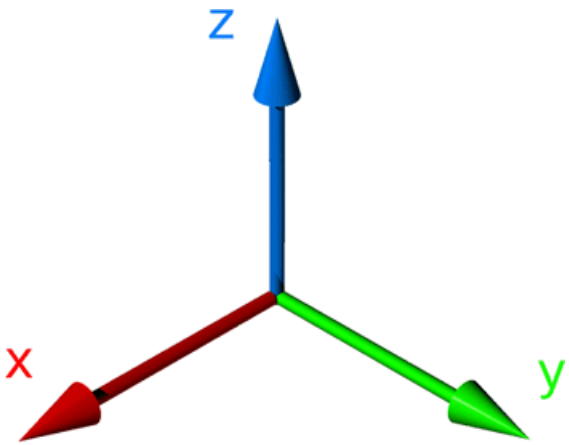
- Corollary: **Given equally suitable designs, designers will/should choose the one that is easier to analyze**
  - I.e., prefer the simpler design

What will software do when it runs?

- It is hard to understand what software will do when it runs
- Theorists will focus on the **halting problem** as a limit
- Developers reason about more mundane situations
  - Do we always close a file that is opened?
  - Is there a handler for every type of request?
  - Can I port the software to the platform of my choice?
  - Do I always sanitize the inputs from the web?
- Both **look at code** and **imagine the runtime**
  - Djikstra's Goto Considered Harmful is this topic

Constraints simplify analysis

- Intensional design constraints simplify analysis
- **PITS**: With invariants, linked lists are easier to reason about
- **PITL**: With guiderails, systems are easier to reason about
  - E.g., in a pipe-and-filter architecture, filters only communicate via pipes and are oblivious to what other filters do
- Constraints limit the size of the **design space**



Simpler machine to analyze

- Design simplicity aids reasoning by providing a **simpler machine**
  - E.g., idempotent operations in distributed systems
- Alternative is an arbitrarily complex data structure or computation
- To achieve **features**
  - Must know what system will **do**
- To achieve **qualities** (reliability, performance, etc.)
  - Must know what system will **NOT do**
- Thought experiment
  - Imagine a "code complicator" that takes your carefully designed source code and contorts it into a mess that still manages to compute the same result
  - Are **constraints** the opposite, a "**de-complicator**"?

Questions

- Which is easier to predict?
  - Where a **building** will go
  - Where a **train** will go
  - Where a **car** will go
  - Where a **person** will go

- Why?

??

Evaluating hoisting

Vigilance: harder than you imagine

Recall, vigilance is shorthand for "**design guidance + vigilance**"

- 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
  - Scherlis: PITS vigilance OK; PITL static analysis better

- Example: Pwn to Own, browser, 2011-2014
  - Exploit based on "Use after free" bug
  - I.e., there was a broken intensional design constraint

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

But, can you convince developers?

Unhappy framework developer

- 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 browser developer

- Pwn to Own, bugs in browsers allowed remote execution of code, 2012-2014
- Exploit based on "Use after free" bug
- Hard to avoid such bugs

Dear 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.

→ Compare with the grumbling browser developer

Sociology

- The grumbling framework 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

Is hoisting a good idea?

Consistency vs. flexibility

- Standardizing means unsuitability for some applications.
- E.g., garbage collectors and realtime applications.
- Local choices disallowed, cannot customize the global approach

Are developers trusted?

- APR vs. XBMC plugins.

Sociology

- Hard to solve problems within the constraints
- Subjective: Cons magnified, Pros minimized by team since the problem is no longer visible.

Expenses

- Investment in shared infrastructure. Time to pay off.
- Homegrown vs. standard implementations.

Summary:

- Benefit of computers enforcing rules vs.
- Overly-rigid (cannot bend rules)

Part 2 Exercise

Exercise

Consider the following mechanisms for hoisting:

- Language runtime
- DSLs
- Frameworks
- VMs and architectural layers
- Libraries (modules) and runtime services (components)

1. Imagine you are a developer.
   a. Discuss your gut feel about being asked to use each of these mechanisms.
   b. Rank them in order of preference.

2. Imagine you are an architect.
   a. How does your preference differ if your chosen guiderail is being enforced?

Exercise workspace

Conclusion

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. Architectural hoisting

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
  - Must be able to see the Fnords

- 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

Summary

- **Architectural hoisting** is:
  - … a new name for an old technique
  - … the enforcement of a constraint with code mechanisms

- Mechanisms to enforce constraints:
  - Language runtime, DSLs, frameworks, VMs, layers, libraries

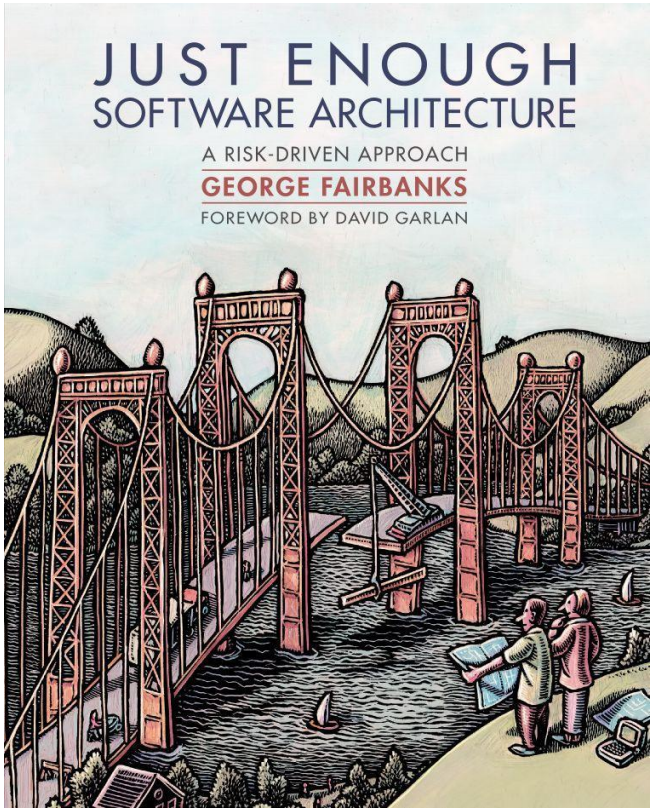
- PITS: **vigilance** can maintain invariants
- PITL: complexity grows with scale
  - So we often choose architectural hoisting

- Benefits
  - Reveals similarity between PITS and PITL techniques
  - Highlights difference in enforcement
  - Clarifies role of architecture

About me (George Fairbanks)

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

E-book on Google play store
Hardback on Amazon, etc.



Backup

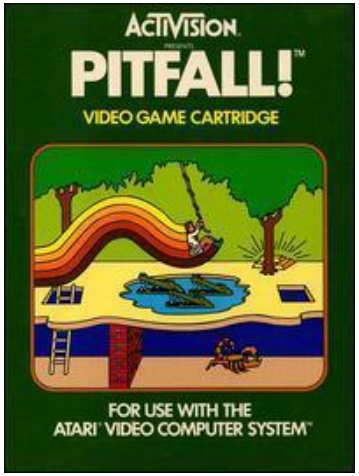
Test: Design pitfalls

- Let's test our hypothesis by considering **design pitfalls**
  - Design pitfalls are ways the design could go wrong

1. Forget to evaluate the design WRT a given quality
   - E.g., deployability, maintainability
2. Not notice a failure path / mode
3. Abstract solution not possible to implement
   - E.g., the APIs don't actually allow it
4. Implementation is more complex than abstract solution, yielding more failure possibilities

- Consistent with the design goal being reasoning

- Contrast these with mechanical **coding bugs**
  - Off-by-one, protocol misuse, …



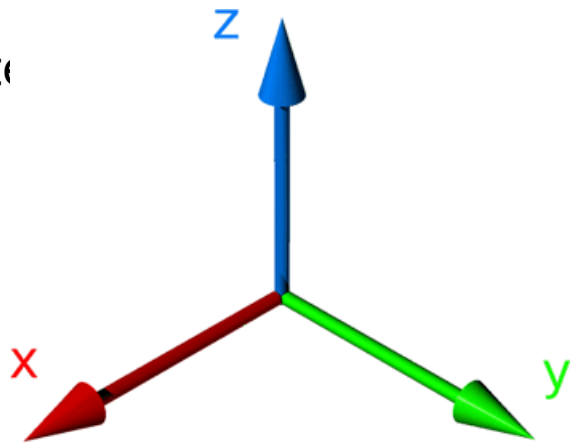
Design space

- We often talk about a **design space**
  - Many dimensions, mostly orthogonal
  - Yes, this is pretty vague

- A system inhabits a point in the design space
  - E.g., it is "an encapsulated, 3-tier, object-oriented system"
  - It follows some design principles and not others
    - E.g., it's OO not functional, 3-tier not peer-to-peer

- Each piece of design guidance limits the design space
  - Hopefully with good effect
  - Meant to guide developers to successful syst

- Choosing / combining guidance
  - Some force choices; some combine
  - A set of design guidance → **style of design**
  - Some styles recognized, such as OO and Unix-style design



Role of failure in engineering **TODO**

Petroski

Djikstra on reasoning **TODO**

Djikstra. It is difficult to reason about a program's behavior, so developers deliberately impose simplicity in order to promote their ability to reason. Djikstra's “Goto Considered Harmful” paper advocated simplifying a program's static structure (eliminating Goto statements, using standardized control flow) in order to better reason about its runtime behavior. Similarly, developers impose invariants on classes and data structures so that they can reason about the code more easily and therefore gain confidence that it works as intended.

Programming in the large

Programming in the small and large **TODO**

Summary of the PITL and PITS

PITL usually called architecture today

Different set of concerns.

What does a programming language do? **TODO**

What does a programming language do … and not do?

Programming languages are good at expressing extensional ideas but poor at intensional ones

Modern programming languages are good at expressing the solutions we have designed but less good at expressing the designs themselves. In particular, we often design systems such that "it always does this" or "it never does that" but our programming languages cannot directly express such design ideas. In a language like C or Java it is not possible to express the invariant on a linked list.

What might an architecture language do?**TODO**

- Configuration of modules (or components) like Spring / Guice?
  - Extensional. But could have Intensional parts.
- Expression of quality attribute constraints?
- Allow dynamic adaptations based on quality attribute constraints?
  - Eg Cloud auto-scaling
- What can we do to approximate it?

Architectural hoisting is PITL programming (without the language).

Constraints in the small: Invariants **TODO**

Constraints are intensional.
Choice is forced between vigilance and hoisting
Use of invariants in the small; vigilance to enforce them
Locality (and small size) make vigilance practical

Example: Linked list

Why not vigilance in the large? **TODO**

Scale factors that work against vigilance

- Our reasoning and memory are imperfect, so it's harder to reason about larger problems. Contrast with static analysis.
- Communicating the constraints to the team. Constraints broken through ignorance.
- Huge quantity of constraints, much churn.
- Retroactive changing of constraints (Arianne 5, NiCd battery "memory")

Constraints enable reasoning **TODO**

The same simplifying idea, constraints, is used in PITS and PITL to enable reasoning

- PITS uses invariants, PITL uses guiderails: both are constraints.
- This is interesting: The consistency implies it is a deep insight about s/w design.
- This is surprising: In most other ways PITS and PITL are dramatically different.
  - PITS: focus on algorithm & data structures.
  - PITL: focus on quality attributes.