

# Introduction to Software Architecture and Documentation

Claire Le Goues

Michael Hilton

# Administrivia

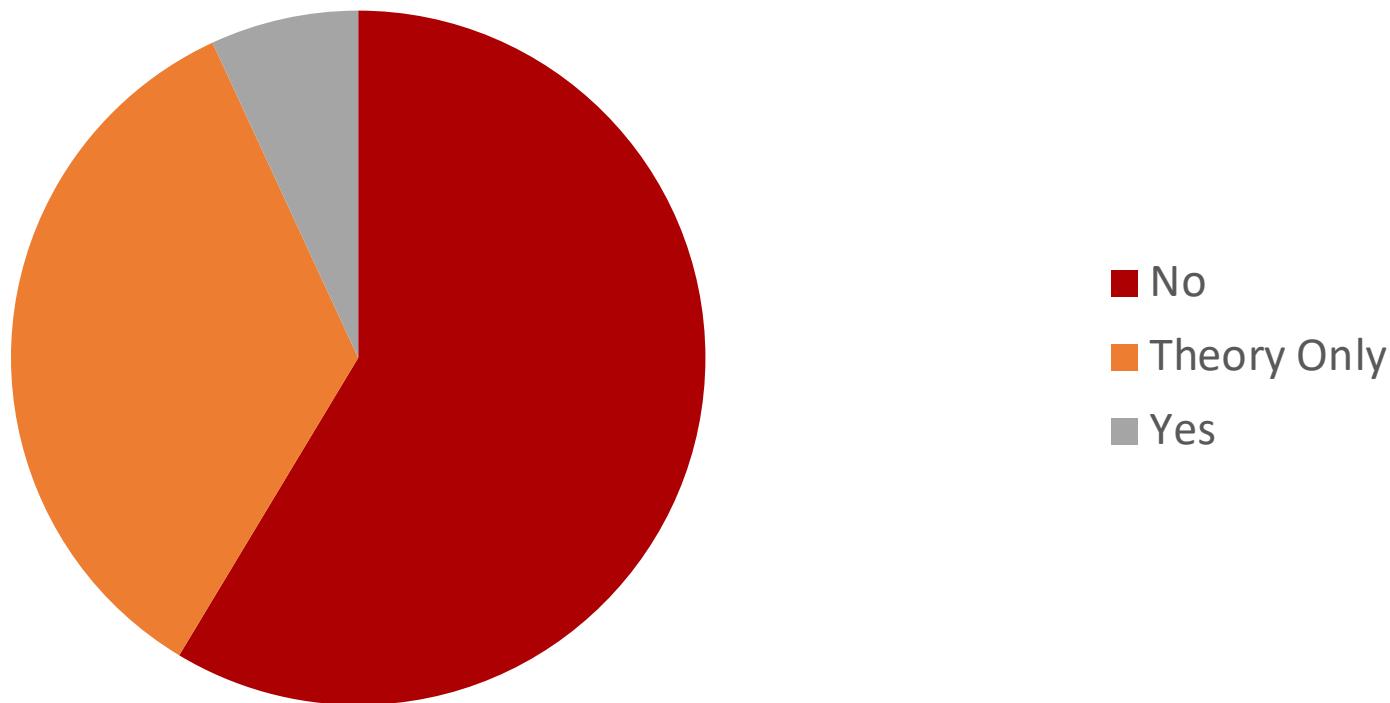
- Recitation instructions posted: do some install ahead of time.
- Schedule your HW3 interviews!

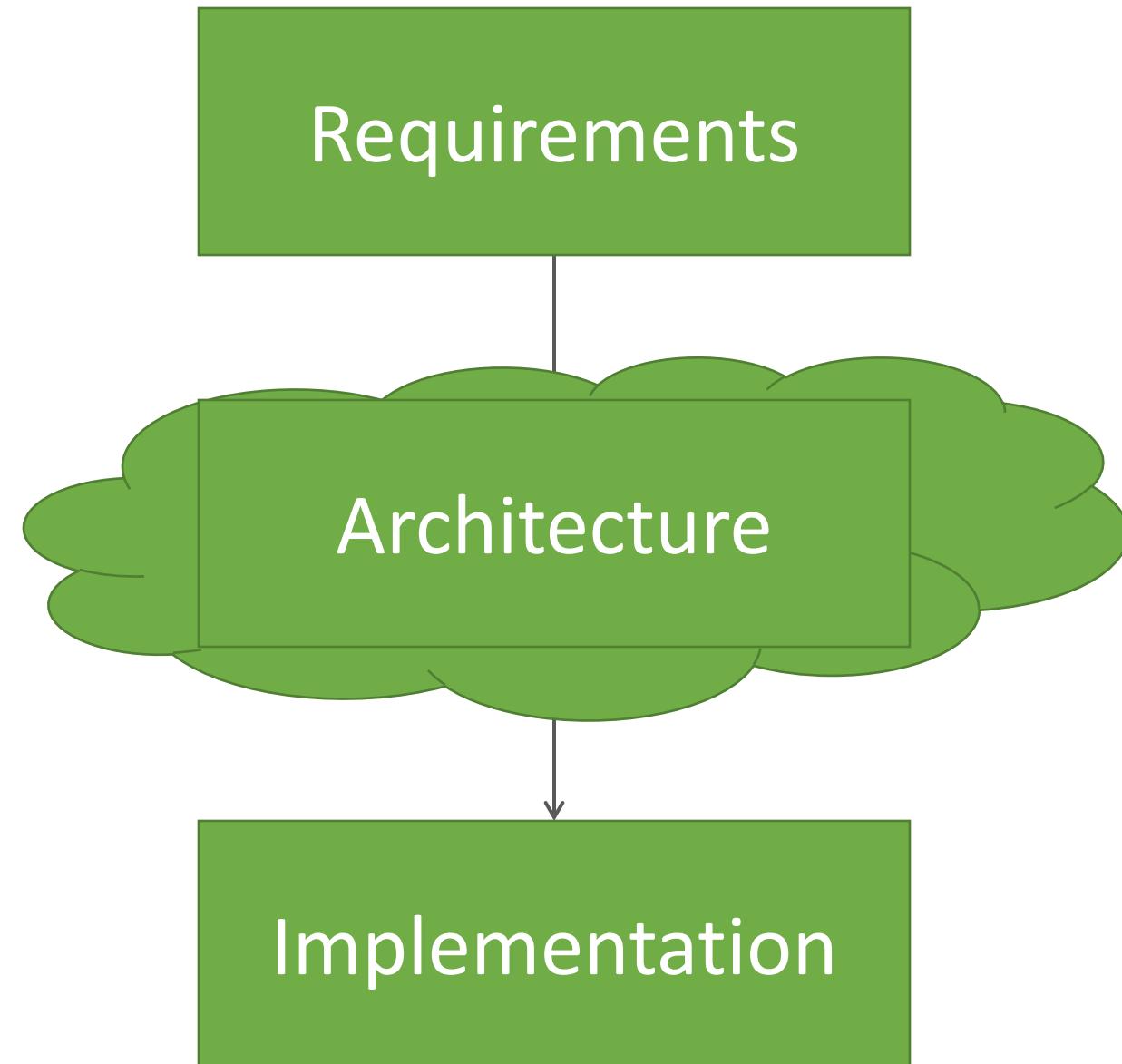
# Learning Goals

- Understand the abstraction level of architectural reasoning
- Approach software architecture with quality attributes in mind
- Distinguish software architecture from (object-oriented) software design
- Use notation and views to describe the architecture suitable to the purpose
- Document architectures clearly, without ambiguity

# About You

I am familiar with how to design distributed,  
high-availability, or high-performance systems





# Quality Requirements, now what?

- "should be highly available"
- "should answer quickly, accuracy is less relevant"
- "needs to be extensible"
- "should efficiently use hardware resources"

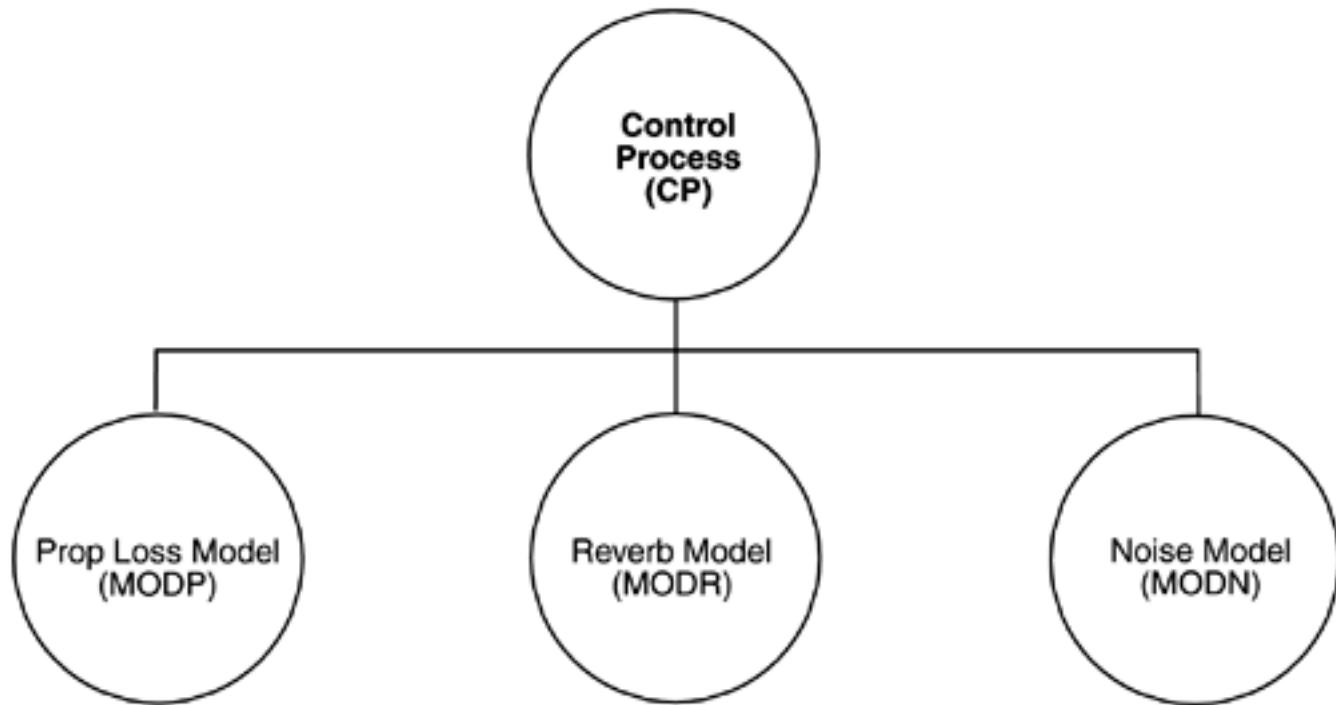
# SOFTWARE ARCHITECTURE

# Software Architecture

*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

[class et al. 2003]

Note: this definition is ambivalent  
to whether the architecture is  
known, or whether it's any good!



# Why Architecture? [BCK03]

- Represents earliest design decisions.
- Aids in **communication** with stakeholders
  - Shows them “how” at a level they can understand, raising questions about whether it meets their needs
- Defines **constraints** on implementation
  - Design decisions form “load-bearing walls” of application
- Dictates **organizational structure**
  - Teams work on different components
- Inhibits or enables **quality attributes**
  - Similar to design patterns
- Supports **predicting** cost, quality, and schedule
  - Typically by predicting information for each component
- Aids in software **evolution**
  - Reason about cost, design, and effect of changes
- Aids in **prototyping**
  - Can implement architectural skeleton early

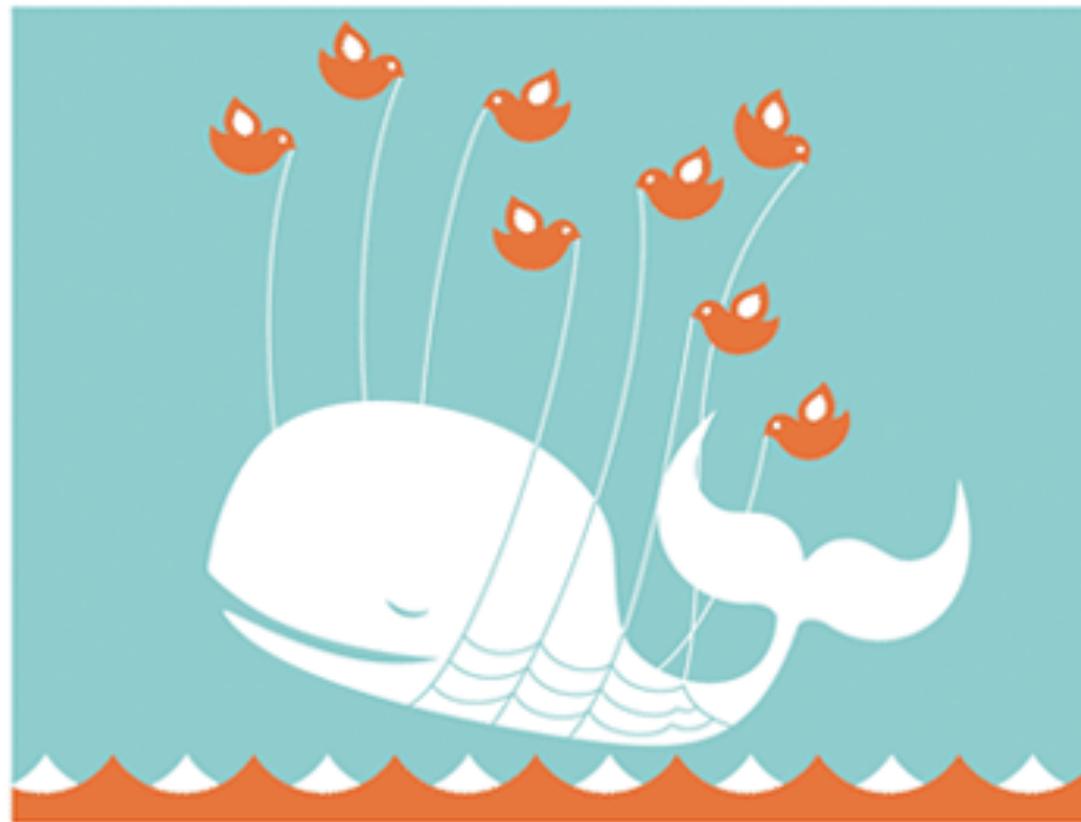
# Beyond functional correctness

- Quality matters, eg.,
  - Performance
  - Availability
  - Modifiability, portability
  - Scalability
  - Security
  - Testability
  - Usability
  - Cost to build, cost to operate

# **CASE STUDY: ARCHITECTURE AND QUALITY AT TWITTER**

## Twitter is over capacity.

Too many tweets! Please wait a moment and try again.

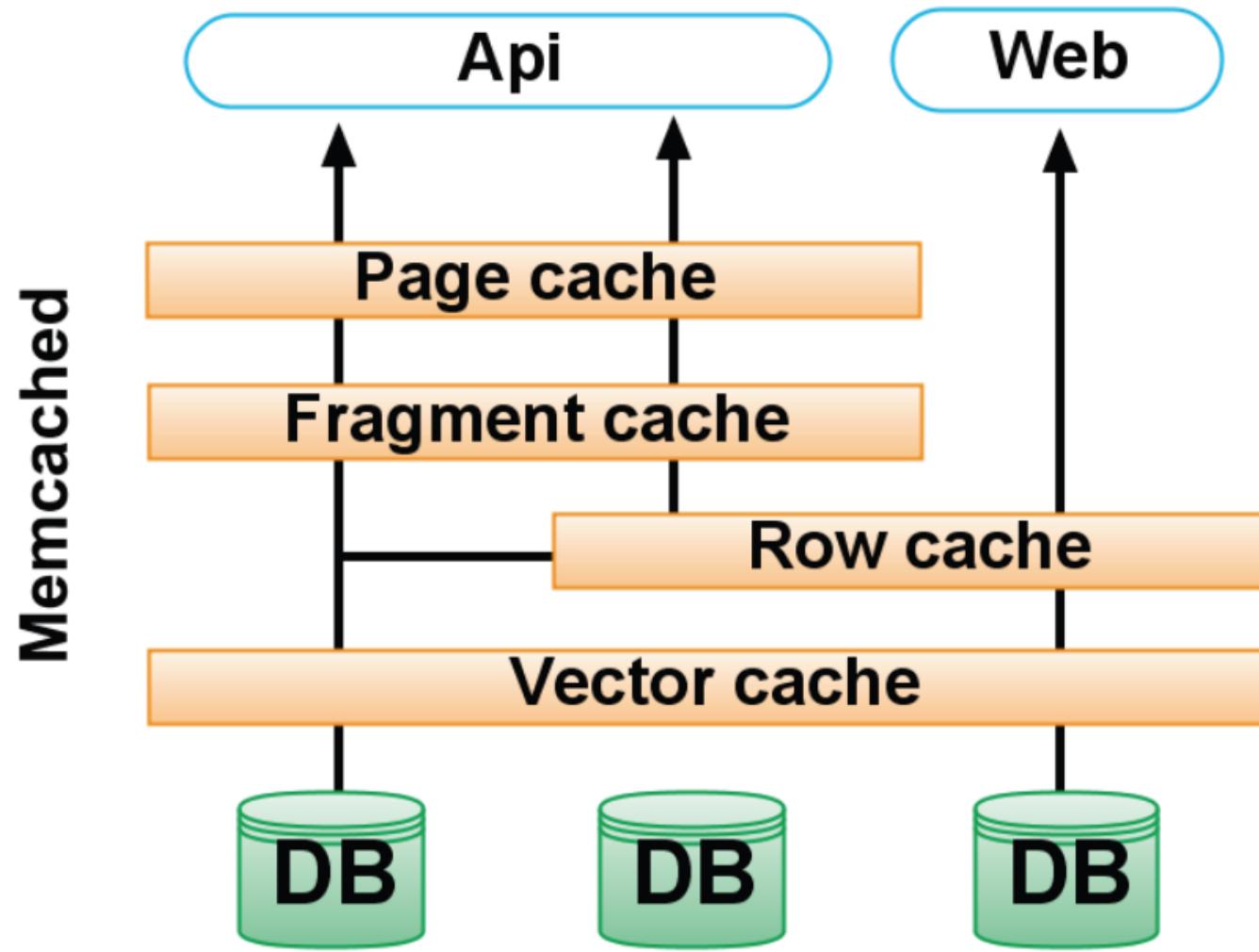




# Inspecting the State of Engineering

- Running one of the world's largest Ruby on Rails installations
  - 200 engineers
- Monolithic: managing raw database, memcache, rendering the site, and presenting the public APIs in one codebase
- Increasingly difficult to understand system; organizationally challenging to manage and parallelize engineering teams
- Reached the limit of throughput on our storage systems (MySQL); read and write hot spots throughout our databases
- Throwing machines at the problem; low throughput per machine (CPU + RAM limit, network not saturated)
- Optimization corner: trading off code readability vs performance

# Caching



# Twitter's Quality Requirements/Redesign goals??

- Improve median latency; lower outliers *performance*
- Reduce number of machines 10x
- Isolate failures *reliability*
- "We wanted cleaner boundaries with "related" logic being in one place"
  - encapsulation and modularity at the systems level (rather than at the class, module, or package level) *maintainability*
- Quicker release of new features
  - "run small and empowered engineering teams that could make local decisions and ship user-facing changes, independent of other teams" *modifiability*

# JVM vs Ruby VM

- Rails servers capable of 200-300 requests / sec / host
- Experience with Scala on the JVM; level of trust
- Rewrite for JVM allowed 10-20k requests / sec / host

# Programming Model

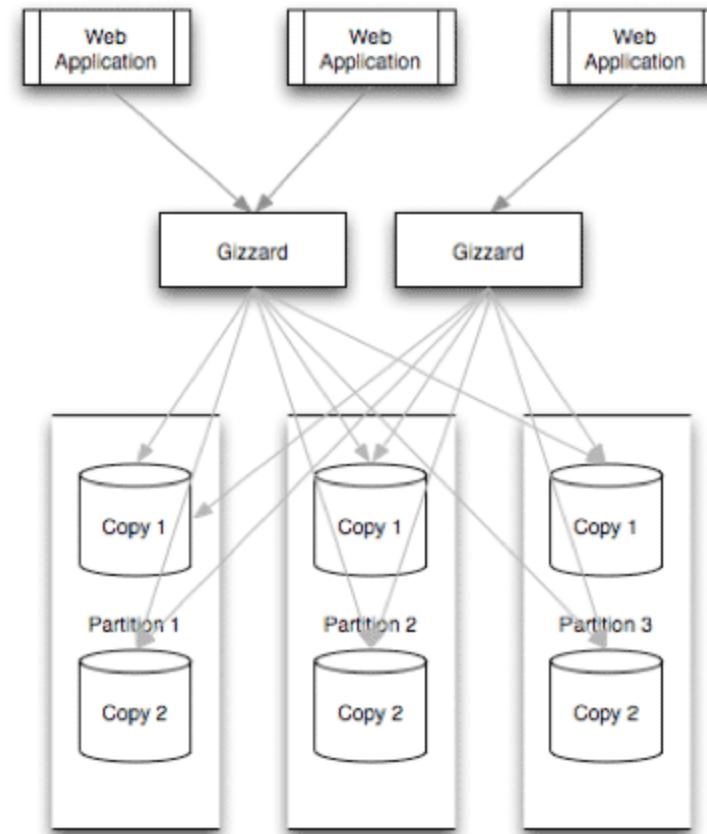
- Ruby model: Concurrency at process level; request queued to be handled by one process
- Twitter response aggregated from several services – additive response times
- *"As we started to decompose the system into services, each team took slightly different approaches. For example, the failure semantics from clients to services didn't interact well: we had no consistent back-pressure mechanism for servers to signal back to clients and we experienced "thundering herds" from clients aggressively retrying latent services."*
- Goal: Single and uniform way of thinking about concurrency
  - Implemented in a library for RPC (Finagle), connection pooling, failover strategies and load balancing

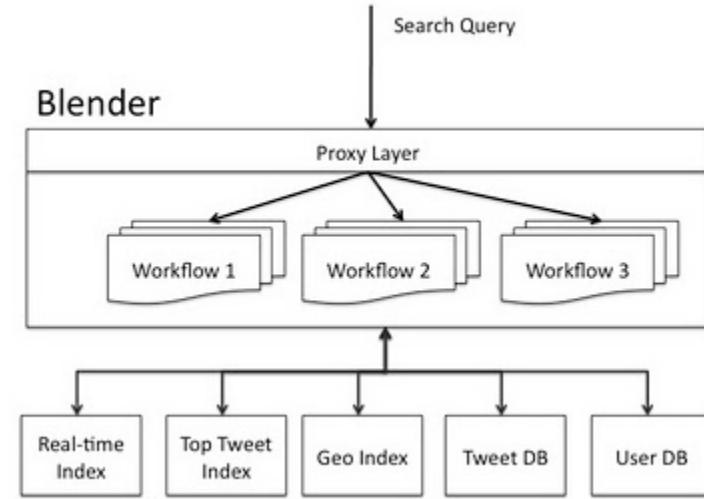
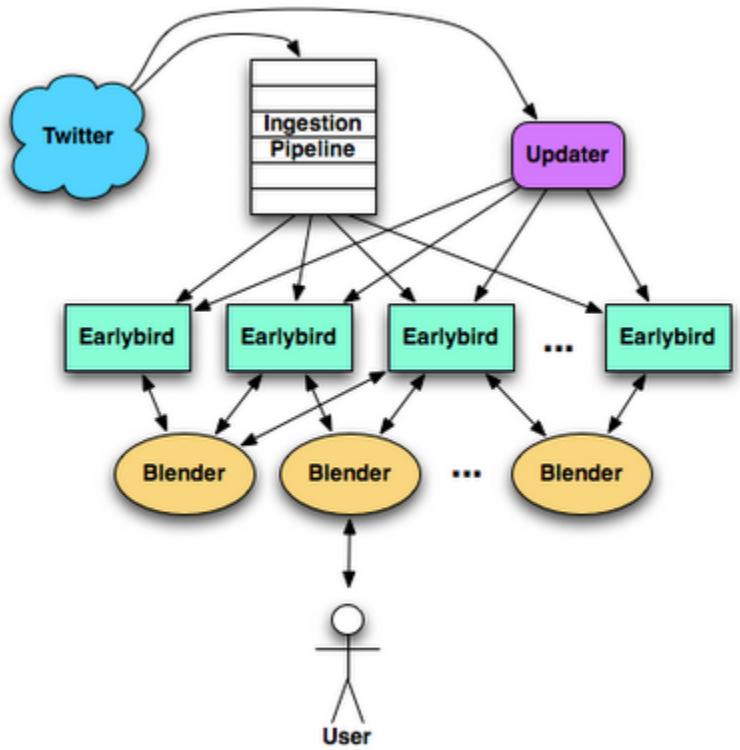
# Independent Systems

- "In our monolithic world, we either needed experts who understood the entire codebase or clear owners at the module or class level. Sadly, the codebase was getting too large to have global experts and, in practice, having clear owners at the module or class level wasn't working. Our codebase was becoming harder to maintain, and teams constantly spent time going on "archeology digs" to understand certain functionality. Or we'd organize "whale hunting expeditions" to try to understand large scale failures that occurred."
- From monolithic system to multiple services
  - Agree on RPC interfaces, develop system internals independently
  - Self-contained teams

# Storage

- Single-master MySQL database bottleneck despite more modular code
- Temporal clustering
  - Short-term solution
  - Skewed load balance
  - One machine + replications every 3 weeks
- Move to distributed database (Glizzard on MySQL) with "roughly sortable" ids
- Stability over features – using older MySQL version

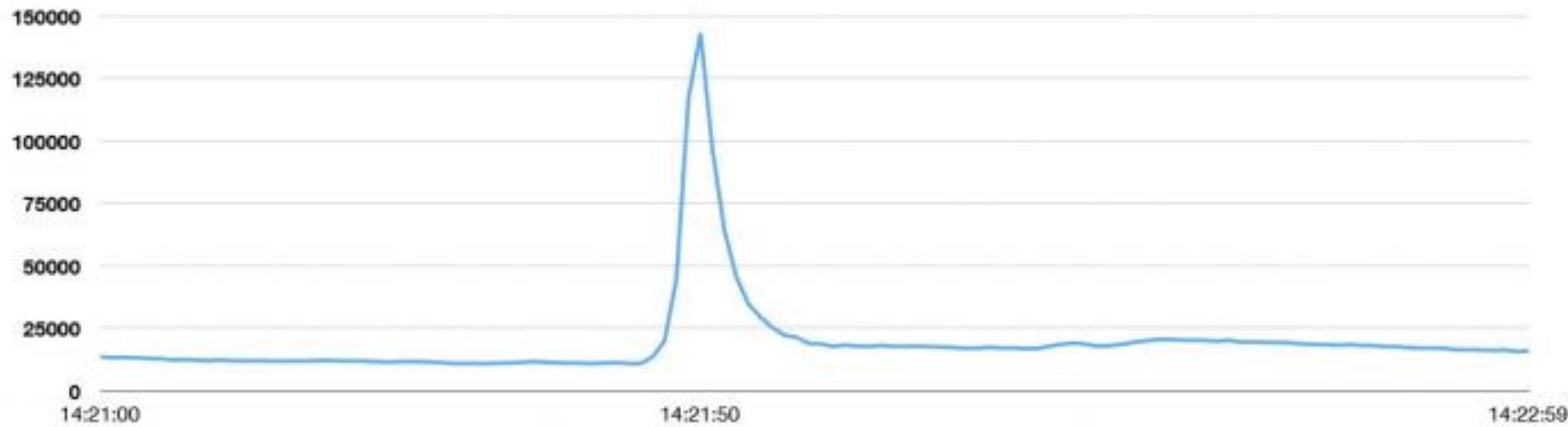




# Data-Driven Decisions

- Many small independent services, number growing
- Own dynamic analysis tool on top of RPC framework
- Framework to configure large numbers of machines
  - Including facility to expose feature to parts of users only





On Saturday, August 3 in Japan, people watched an airing of [Castle in the Sky](#), and at one moment they took to Twitter so much that we hit a one-second peak of 143,199 Tweets per second.

# Key Insights: Twitter Case Study

- Architectural decisions affect entire systems, not only individual modules
- Abstract, different abstractions for different scenarios
- Reason about quality attributes early
- Make architectural decisions explicit
- Question: *Did the original architect make poor decisions?*

# ARCHITECTURE VS OBJECT-LEVEL DESIGN

# Levels of Abstraction

- Requirements
  - high-level “what” needs to be done

→ Architecture (High-level design)

- high-level “how”, mid-level “what”

→ OO-Design (Low-level design, e.g. design patterns)

- mid-level “how”, low-level “what”

→ Code

- low-level “how”

# Design vs. Architecture

## Design Questions

- How do I add a menu item in Eclipse?
- How can I make it easy to add menu items in Eclipse?
- What lock protects this data?
- How does Google rank pages?
- What encoder should I use for secure communication?
- What is the interface between objects?

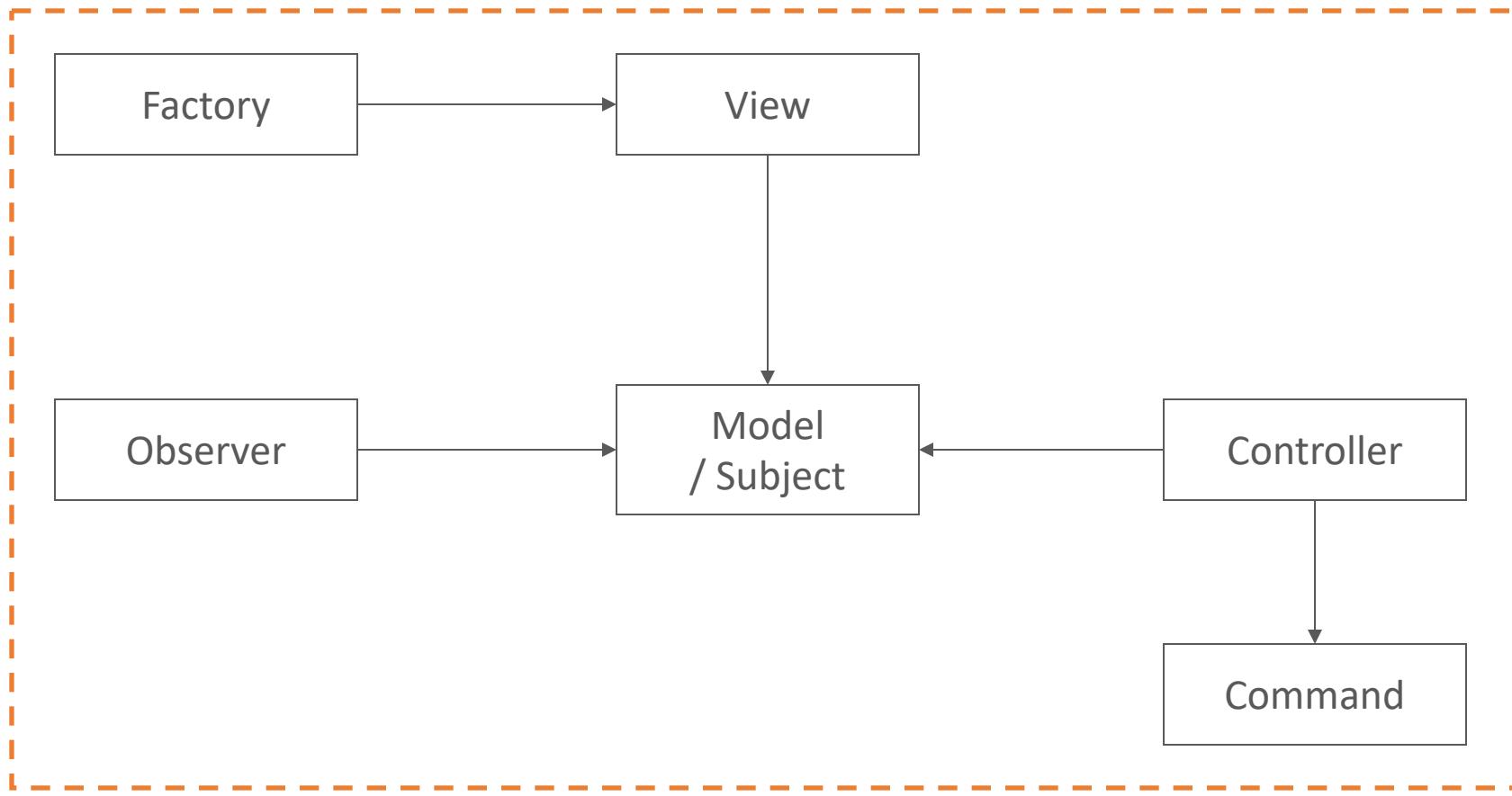
## Architectural Questions

- How do I extend Eclipse with a plugin?
- What threads exist and how do they coordinate?
- How does Google scale to billions of hits per day?
- Where should I put my firewalls?
- What is the interface between subsystems?

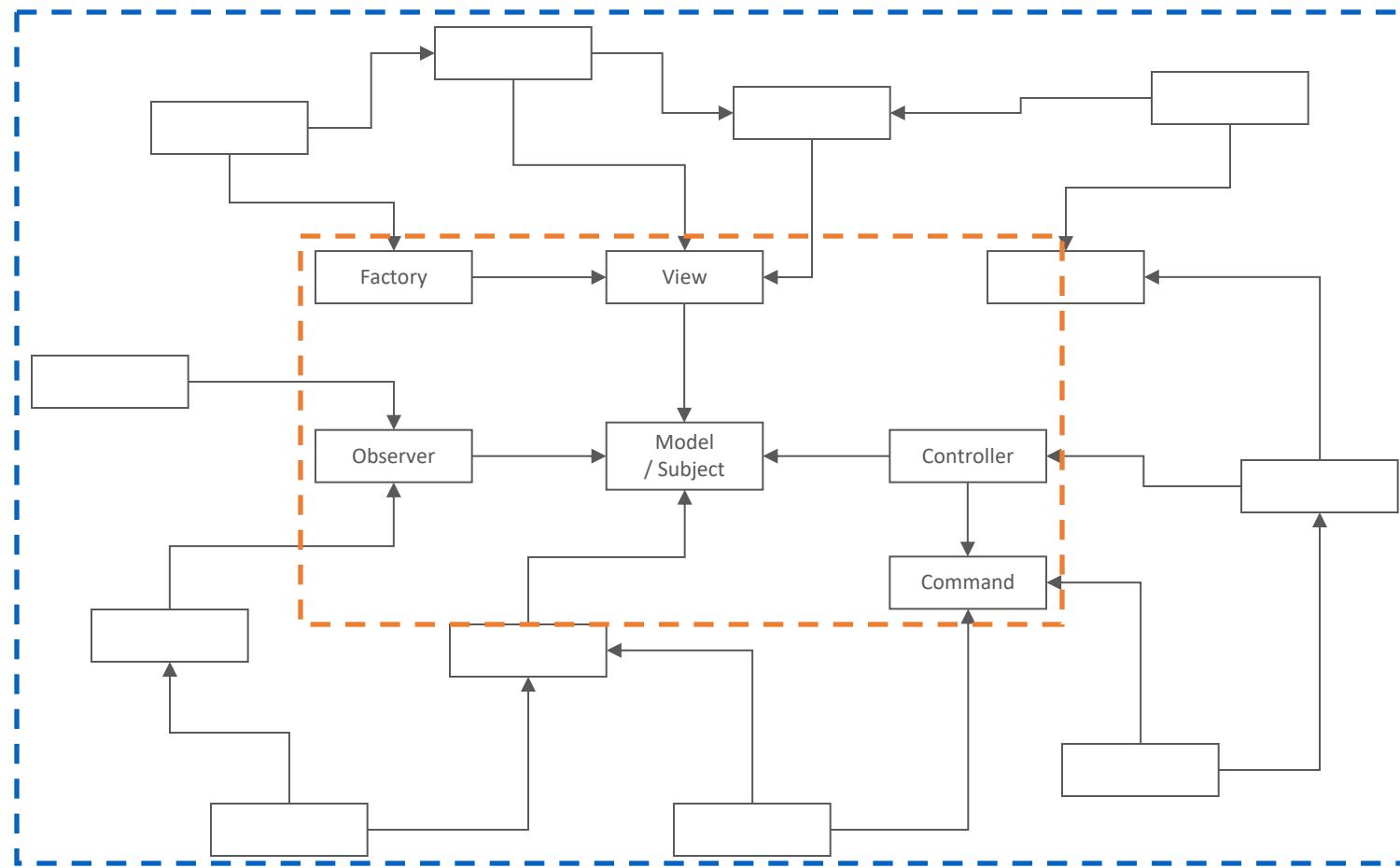
# Objects

Model

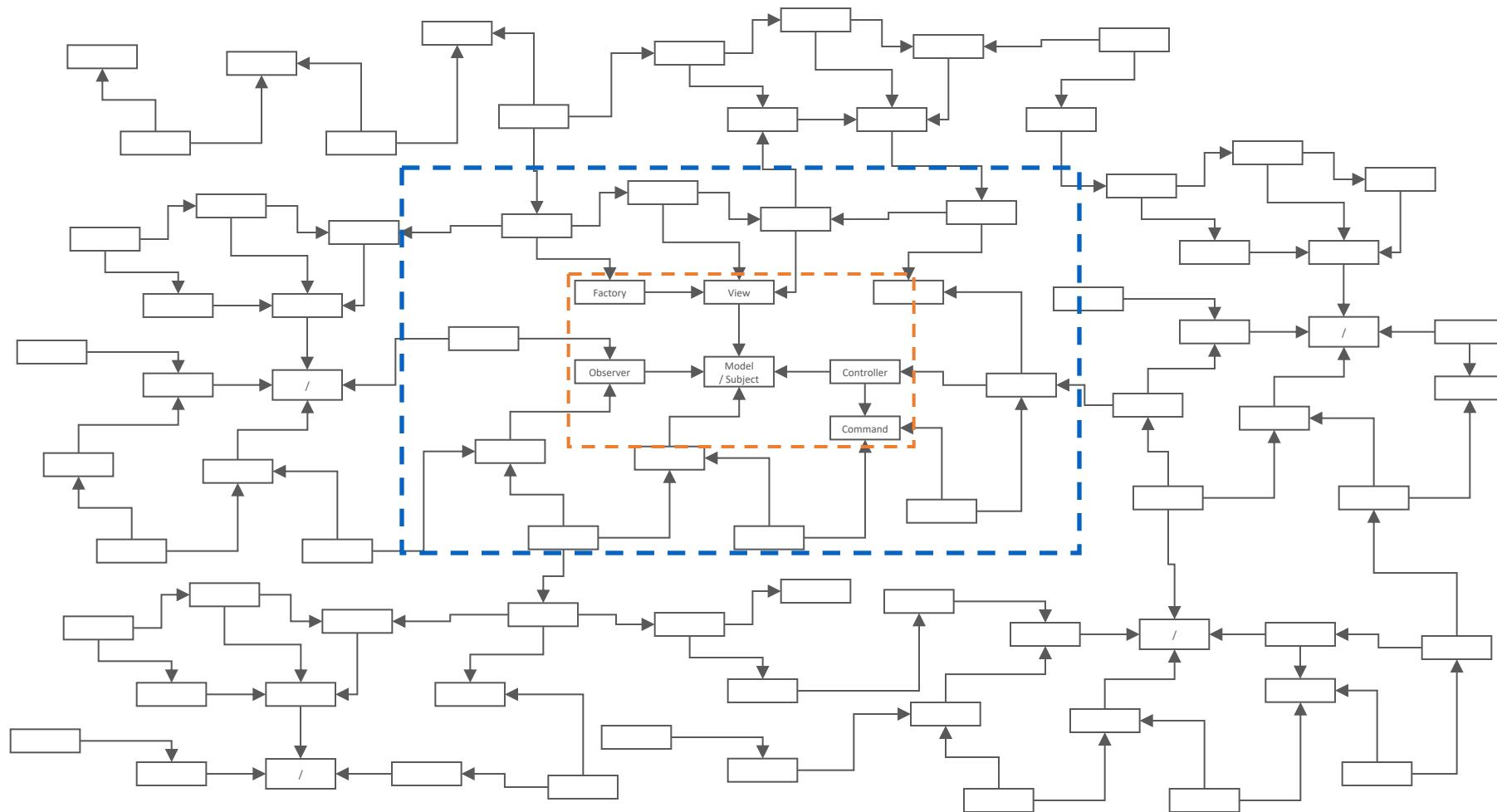
# Design Patterns



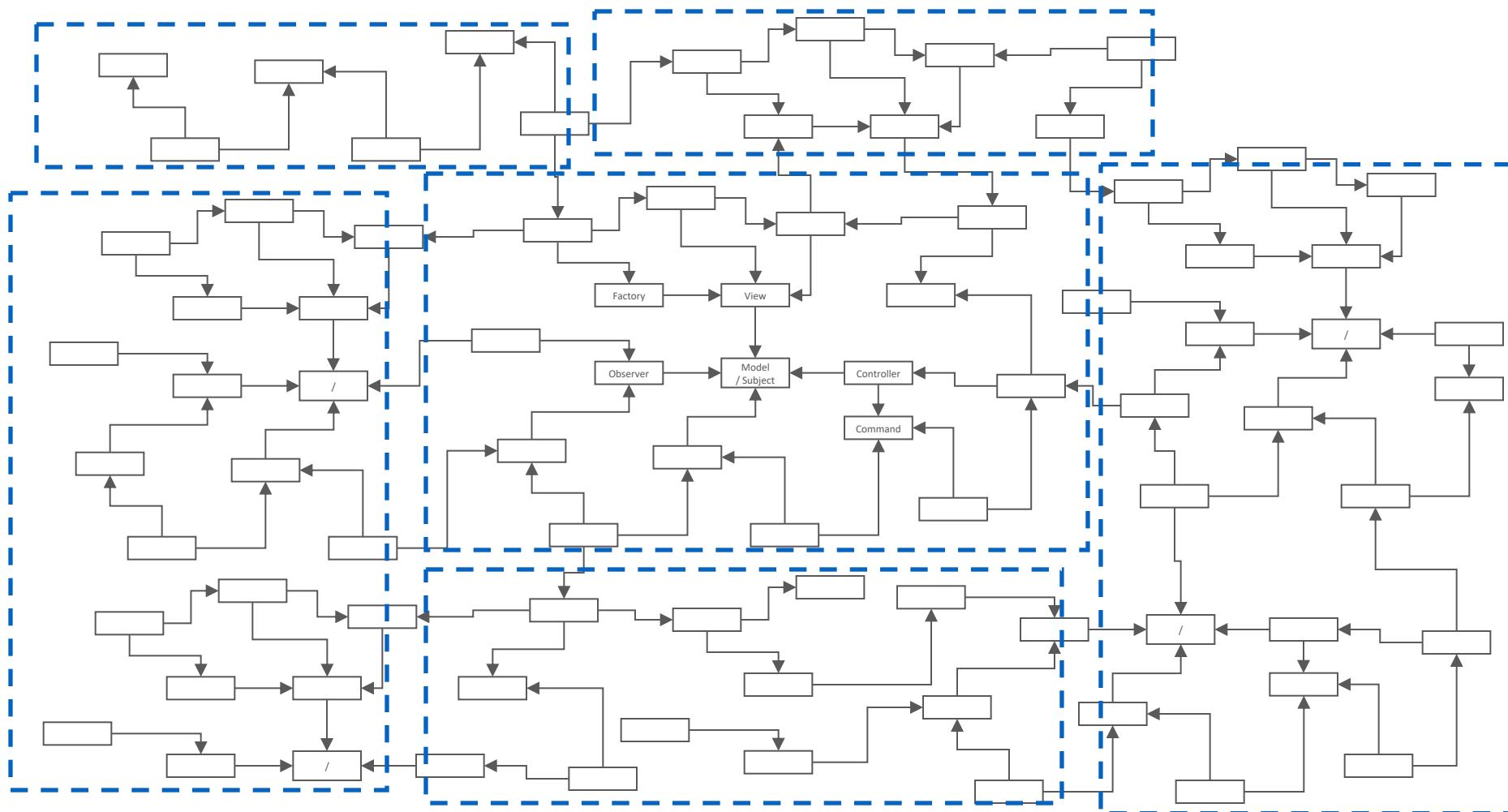
# Design Patterns



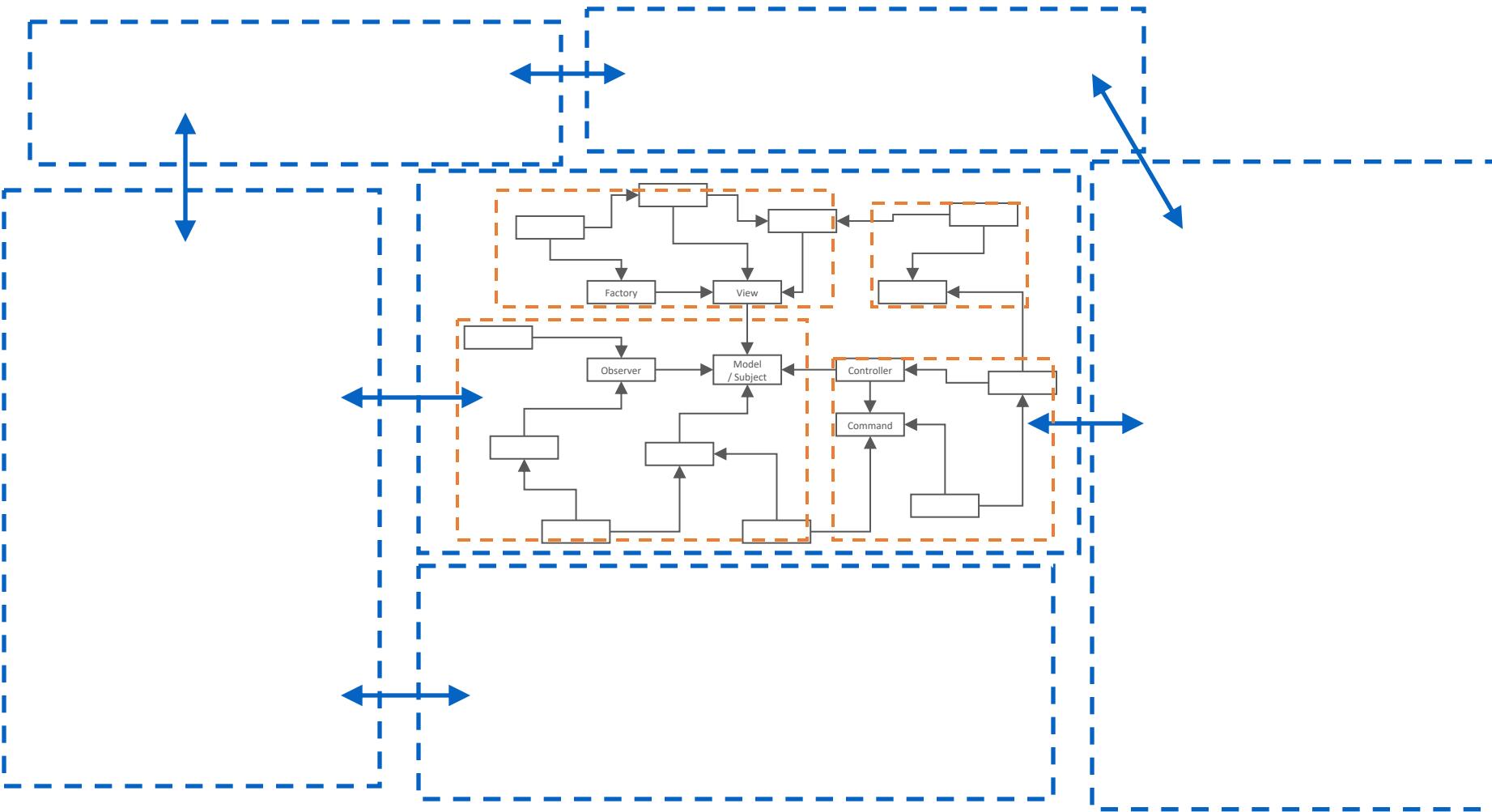
# Design Patterns



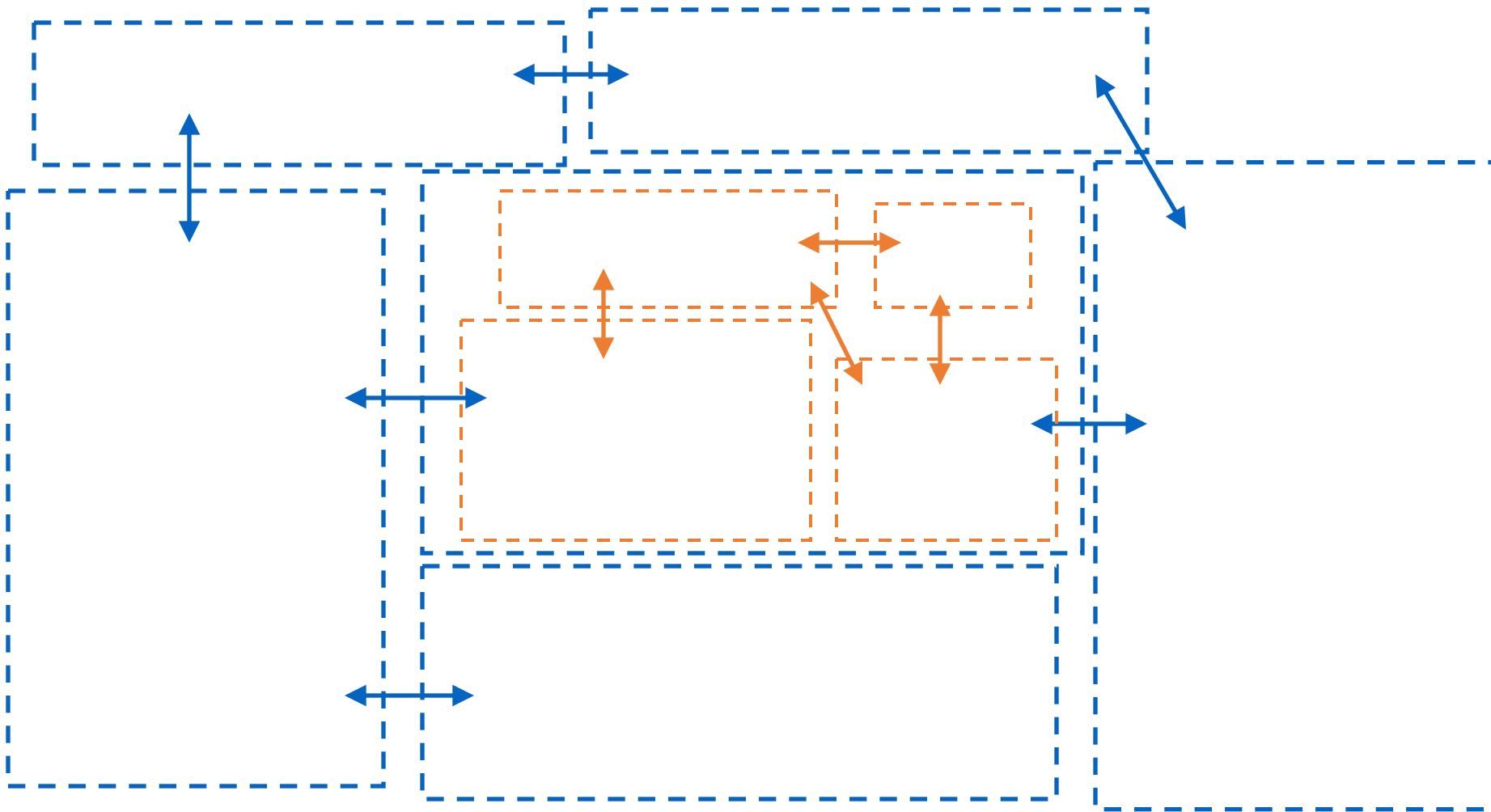
# Architecture



# Architecture



# Architecture



# **ARCHITECTURE DOCUMENTATION & VIEWS**

# Architecture Disentangled

Architecture as  
structures and relations  
(the actual system)

 The picture can't be displayed.

Architecture as  
documentation  
(representations of the system)

 The picture can't be displayed.

 The picture  
can't be  
displayed  
.

Architecture as (design) process  
(activities around the other two)

 The picture can't be displayed.

# Why Document Architecture?

- Blueprint for the system
  - Artifact for early analysis
  - Primary carrier of quality attributes
  - Key to post-deployment maintenance and enhancement
- Documentation speaks for the architect, today and 20 years from today
  - As long as the system is built, maintained, and evolved according to its documented architecture
- Support traceability.











 The picture can't be displayed.

# Common Views in Documenting Software Architecture

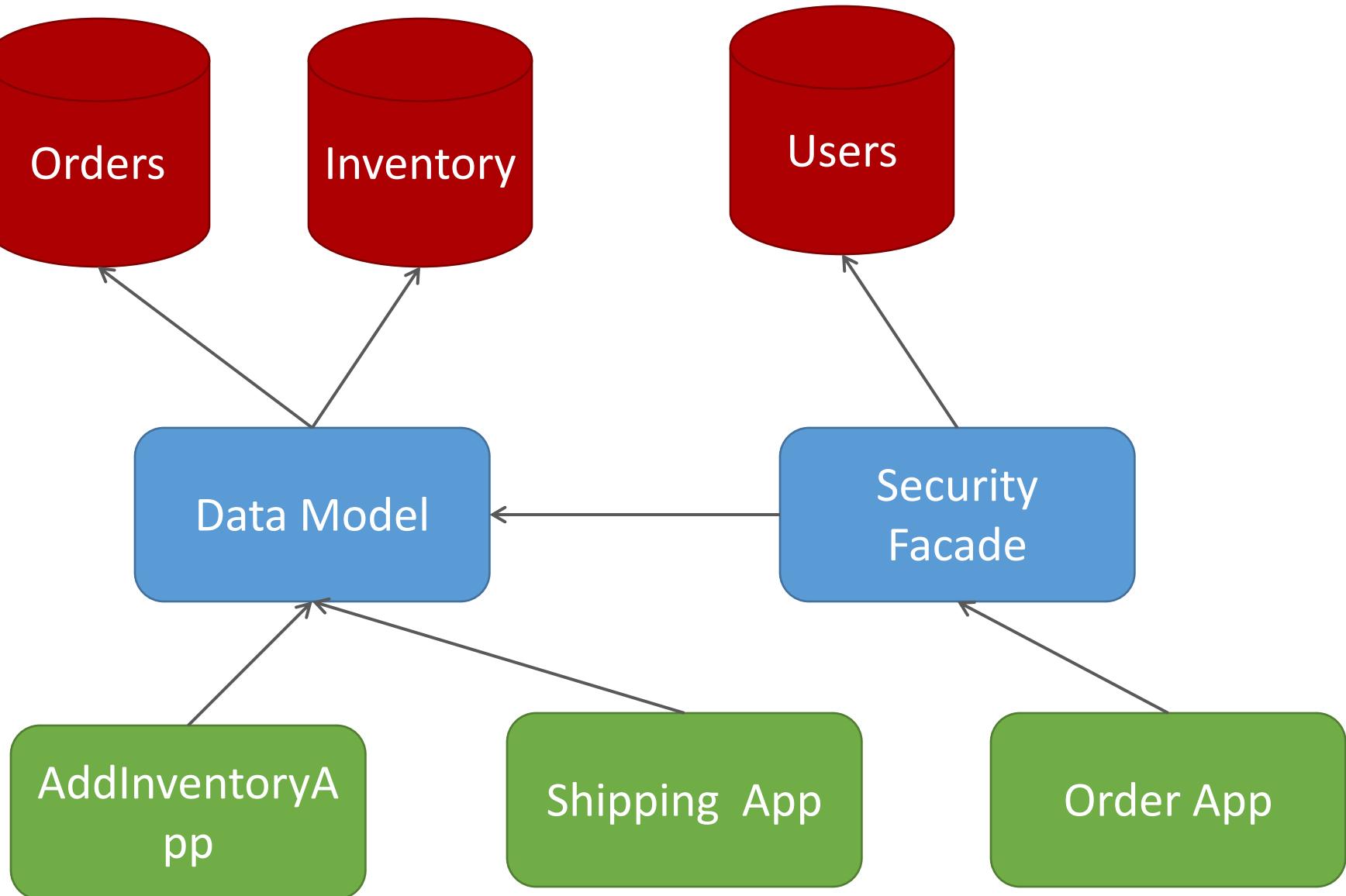
- Static View
  - Modules (subsystems, structures) and their relations (dependencies, ...)
- Dynamic View
  - Components (processes, runnable entities) and connectors (messages, data flow, ...)
- Physical View (Deployment)
  - Hardware structures and their connections

# Views and Purposes

- Every view should align with a purpose
- Different views are suitable for different reasoning aspects (different quality goals), e.g.,
  - Performance
  - Extensibility
  - Security
  - Scalability
  - ...



The picture can't be displayed.





The picture can't be displayed.



The picture can't be displayed.

# What is Wrong Today?

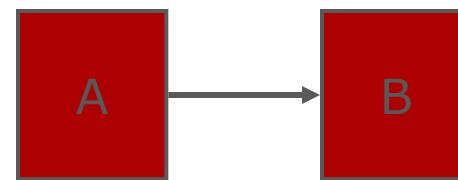
- In practice today's documentation consists of
  - Ambiguous box-and-line diagrams
  - Inconsistent use of notations
  - Confusing combinations of viewtypes
- Many things are left unspecified:
  - What kind of elements?
  - What kind of relations?
  - What do the boxes and arrows mean?
  - What is the significance of the layout?

# Guidelines: Avoiding Ambiguity

- Always include a legend
- Define precisely what the boxes mean
- Define precisely what the lines mean
- Don't mix viewtypes unintentionally
  - Recall: Module (classes), C&C (components)
- Supplement graphics with explanation
  - Very important: rationale (architectural intent)
- Do not try to do too much in one diagram
  - Each view of architecture should fit on a page
  - Use hierarchy

# What could the arrow mean?

- Many possibilities
  - A passes control to B
  - A passes data to B
  - A gets a value from B
  - A streams data to B
  - A sends a message to B
  - A creates B
  - A occurs before B
  - B gets its electricity from A
  - ...



# Recommendations for Recitation and Homework

- Use UML or UML-like notations:
  - Class diagrams for static and physical views
  - Communication diagrams for dynamic view
  - Use correct abstraction level (usually not classes/objects)
- Extend notation as needed
  - Provide a legend explaining the extensions or deviations from standard UML notation

# Further Readings

- Bass, Clements, and Kazman. Software Architecture in Practice. Addison-Wesley, 2003.
- Boehm and Turner. Balancing Agility and Discipline: A Guide for the Perplexed, 2003.
- Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford. Documenting Software Architectures: Views and Beyond, 2010.
- Fairbanks. Just Enough Software Architecture. Marshall & Brainerd, 2010.
- Jansen and Bosch. Software Architecture as a Set of Architectural Design Decisions, WICSA 2005.
- Lattanze. Architecting Software Intensive Systems: a Practitioner's Guide, 2009.
- Sommerville. Software Engineering. Edition 7/8, Chapters 11-13
- Taylor, Medvidovic, and Dashofy. Software Architecture: Foundations, Theory, and Practice. Wiley, 2009.

# Further Readings

- Bass, Clements, and Kazman. Software Architecture in Practice. Addison-Wesley, 2003.
- Boehm and Turner. Balancing Agility and Discipline: A Guide for the Perplexed, 2003.
- Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford. Documenting Software Architectures: Views and Beyond, 2010.
- Fairbanks. Just Enough Software Architecture. Marshall & Brainerd, 2010.
- Jansen and Bosch. Software Architecture as a Set of Architectural Design Decisions, WICSA 2005.
- Lattanze. Architecting Software Intensive Systems: a Practitioner's Guide, 2009.
- Sommerville. Software Engineering. Edition 7/8, Chapters 11-13
- Taylor, Medvidovic, and Dashofy. Software Architecture: Foundations, Theory, and Practice. Wiley, 2009.