

# Architectural documentation, views and tradeoffs, patterns

Claire Le Goues

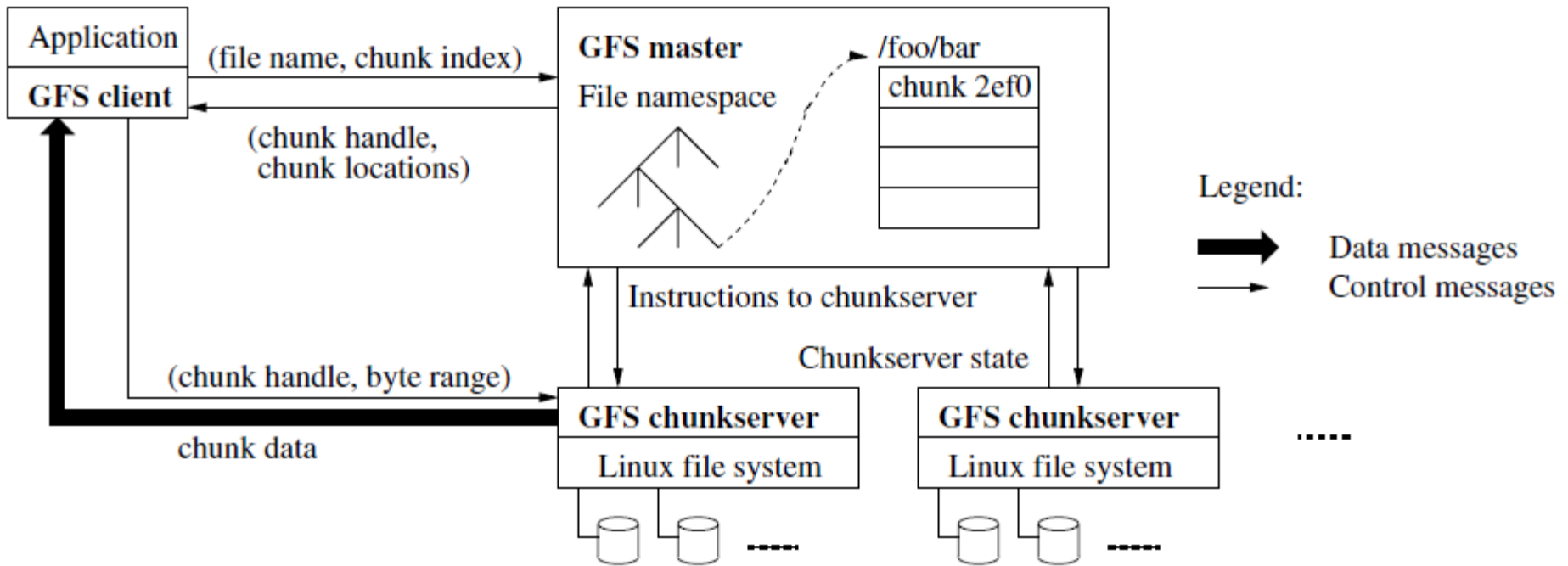
October 8, 2020

# Administrativa

# Learning Goals

- Practice using architecture diagrams to reason about quality attributes.
- Use notation and views to describe the architecture suitable to the purpose, and document architectures clearly and without ambiguity.
- Use diagrams to understand systems and reason about tradeoffs.
- Understand the utility of architectural patterns and tactics, and give a couple of examples.

# CASE STUDY: THE GOOGLE FILE SYSTEM



**Figure 1: GFS Architecture**

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." *ACM SIGOPS operating systems review*. Vol. 37. No. 5. ACM, 2003.

# Assumptions

- The system is built from many inexpensive commodity components that often fail.
- The system stores a modest number of large files.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads.
- The workloads also have many large, sequential writes that append data to files.
- The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file.
- High sustained bandwidth is more important than low latency.

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." *ACM SIGOPS operating systems review*. Vol. 37. No. 5. ACM, 2003.

Qualities:  
Scalability  
Reliability  
Performance  
Cost

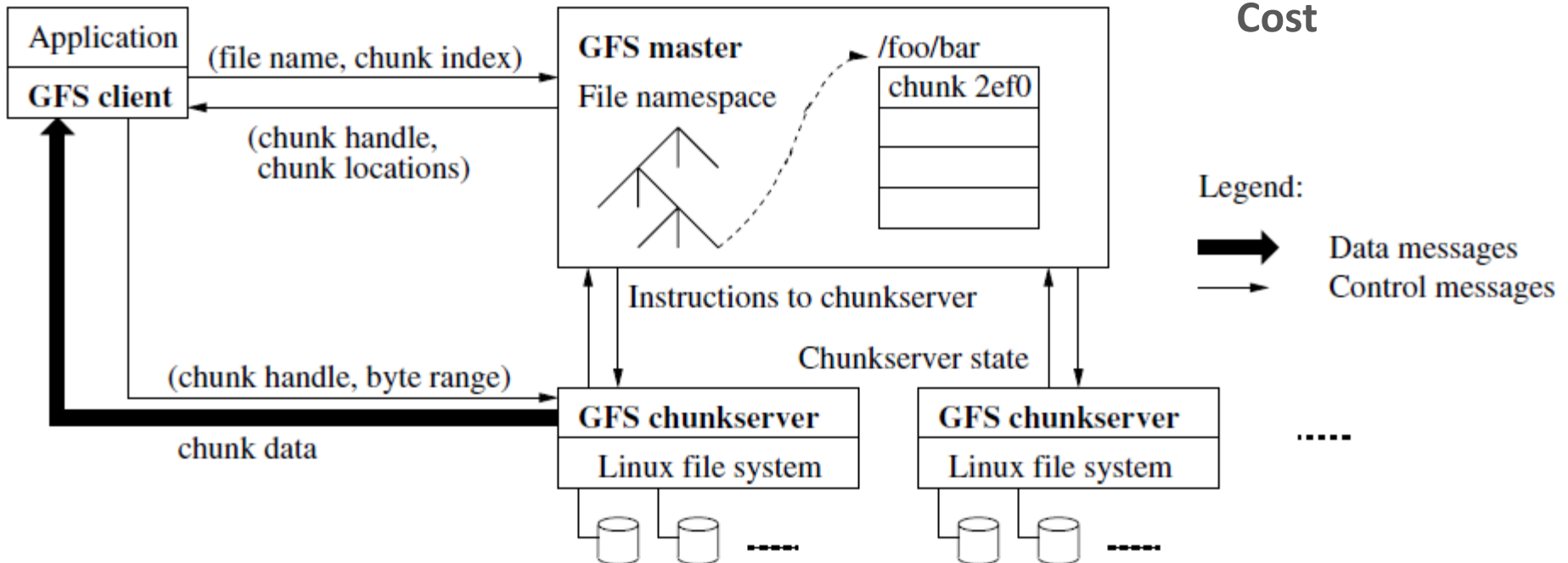


Figure 1: GFS Architecture

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." *ACM SIGOPS operating systems review*. Vol. 37. No. 5. ACM, 2003.

# Questions

1. What are the most important quality attributes in the design?
2. How are those quality attributes realized in the design?



Qualities:  
**Scalability**  
**Reliability**  
**Performance**  
**Cost**

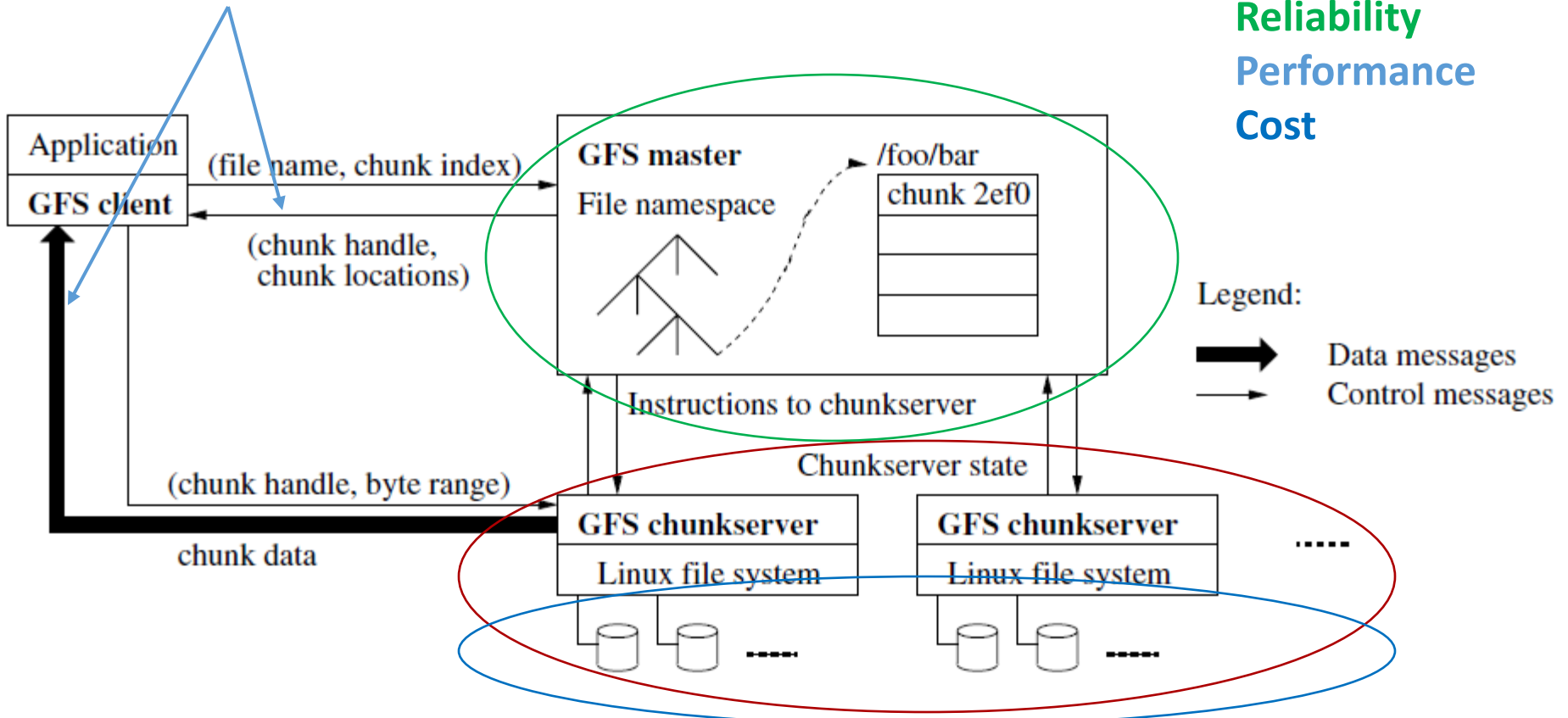


Figure 1: GFS Architecture

# Questions

1. What are the most important quality attributes in the design?
2. How are those quality attributes realized in the design?

# Exercise

For the Google File System, create a physical architecture view that addresses a relevant quality attribute

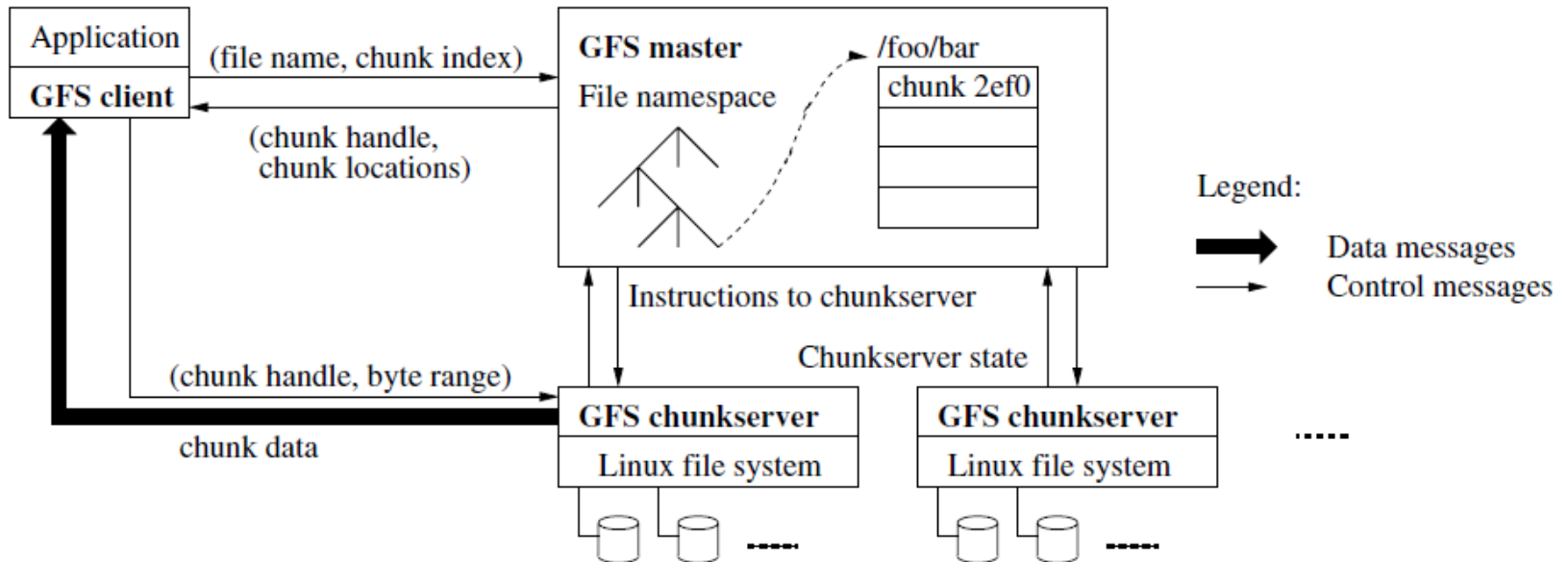
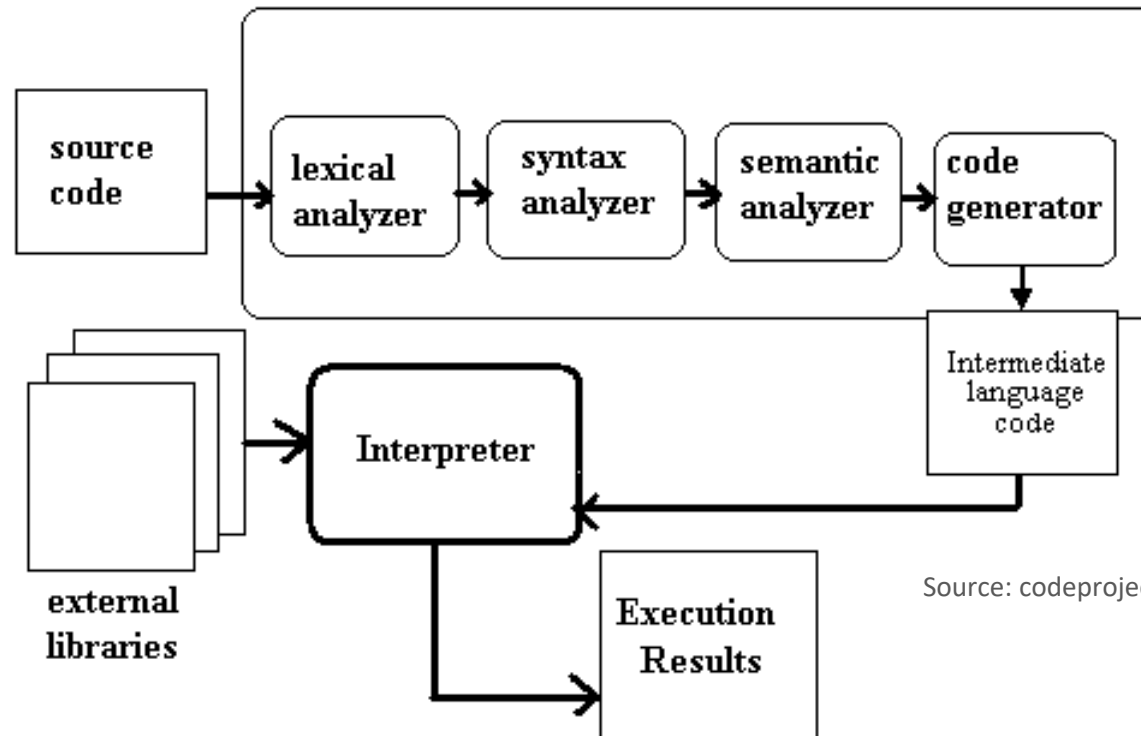


Figure 1: GFS Architecture

# ARCHITECTURE STYLES/PATTERNS

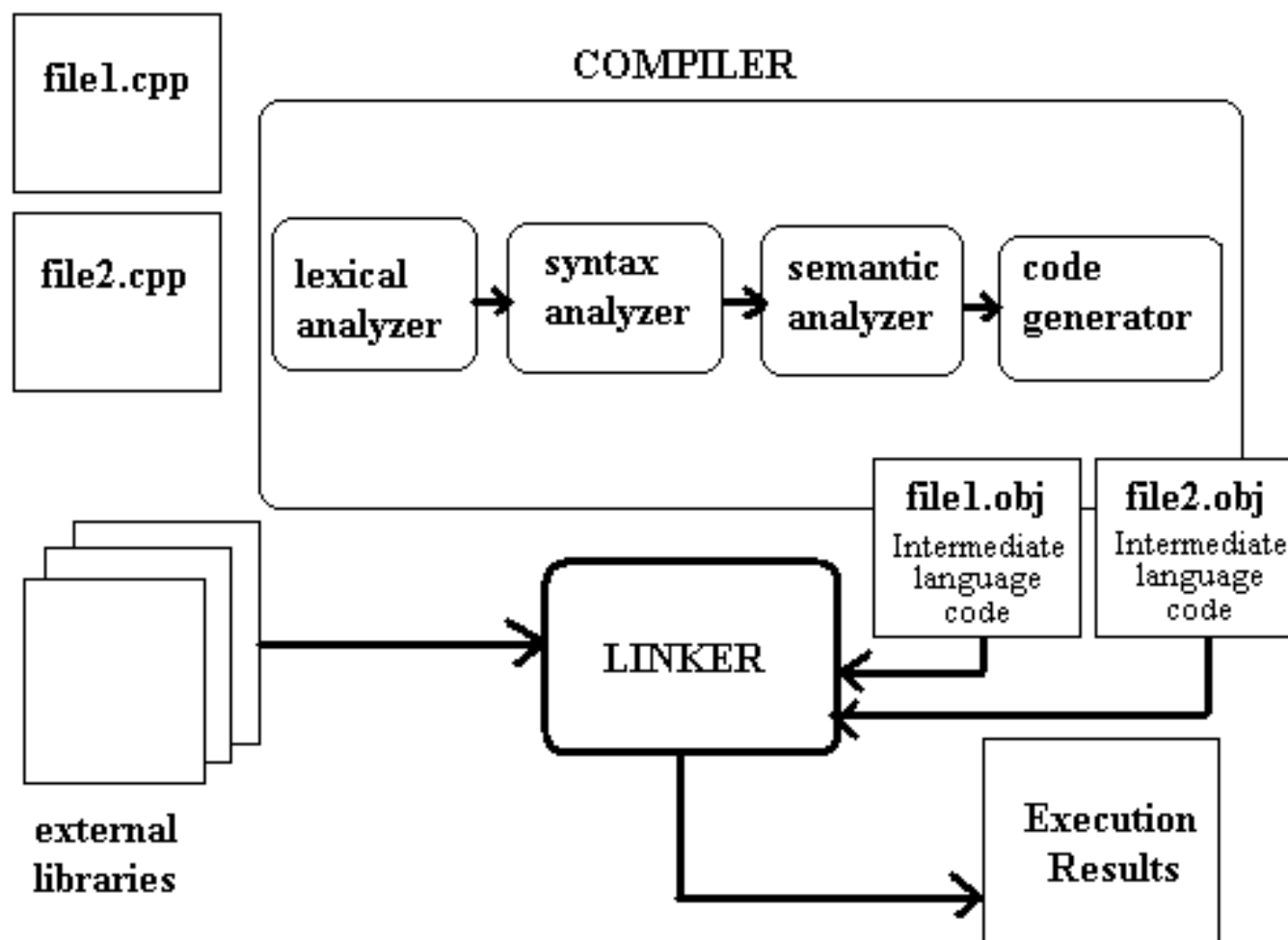
# Architectural style/pattern: broad principle of system organization.

- Describes computational model
  - E.g., pipe and filter, call-return, publish-subscribe, layered, services
- Related to one of common view types
  - Static, dynamic, physical



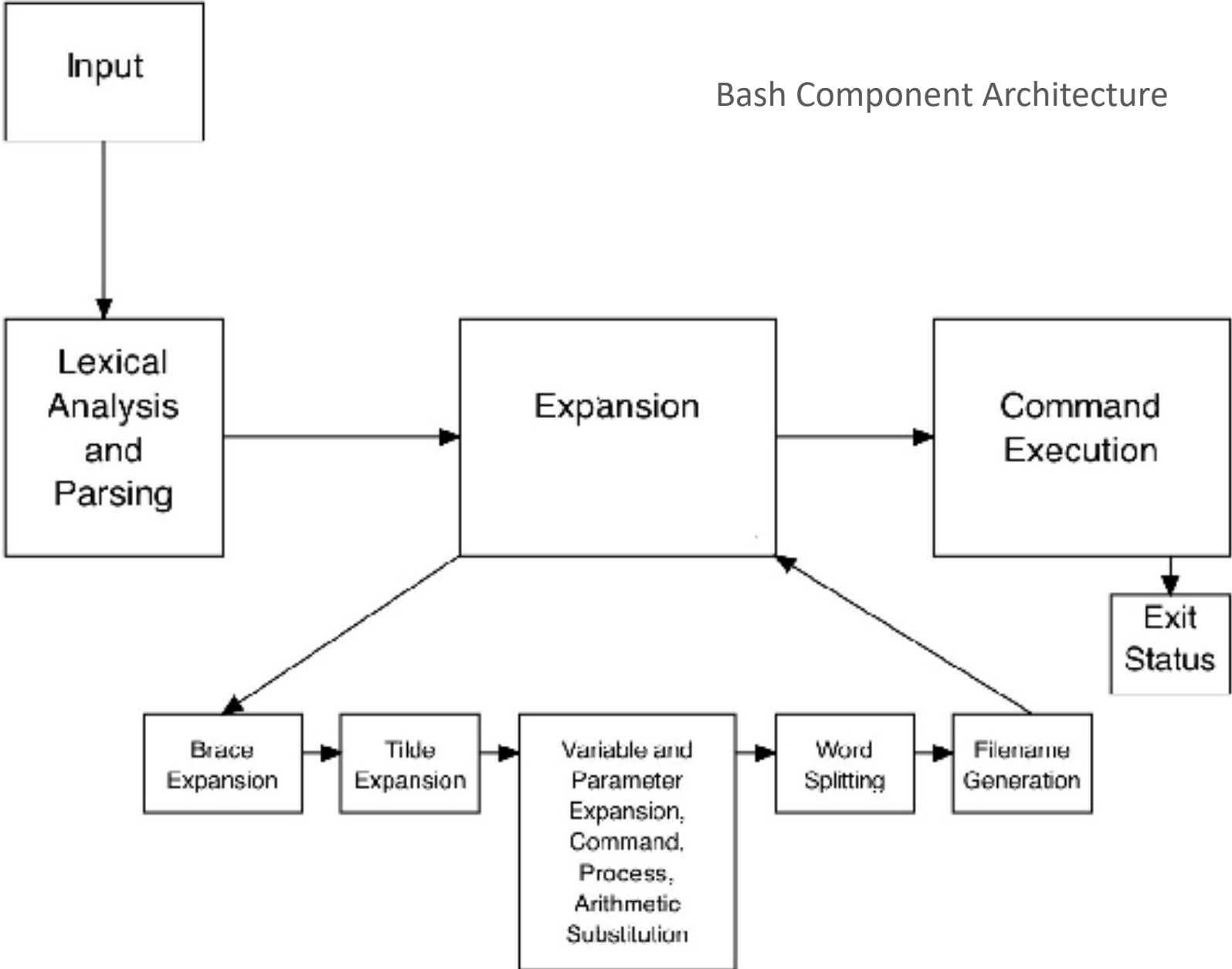
Source: codeproject.org

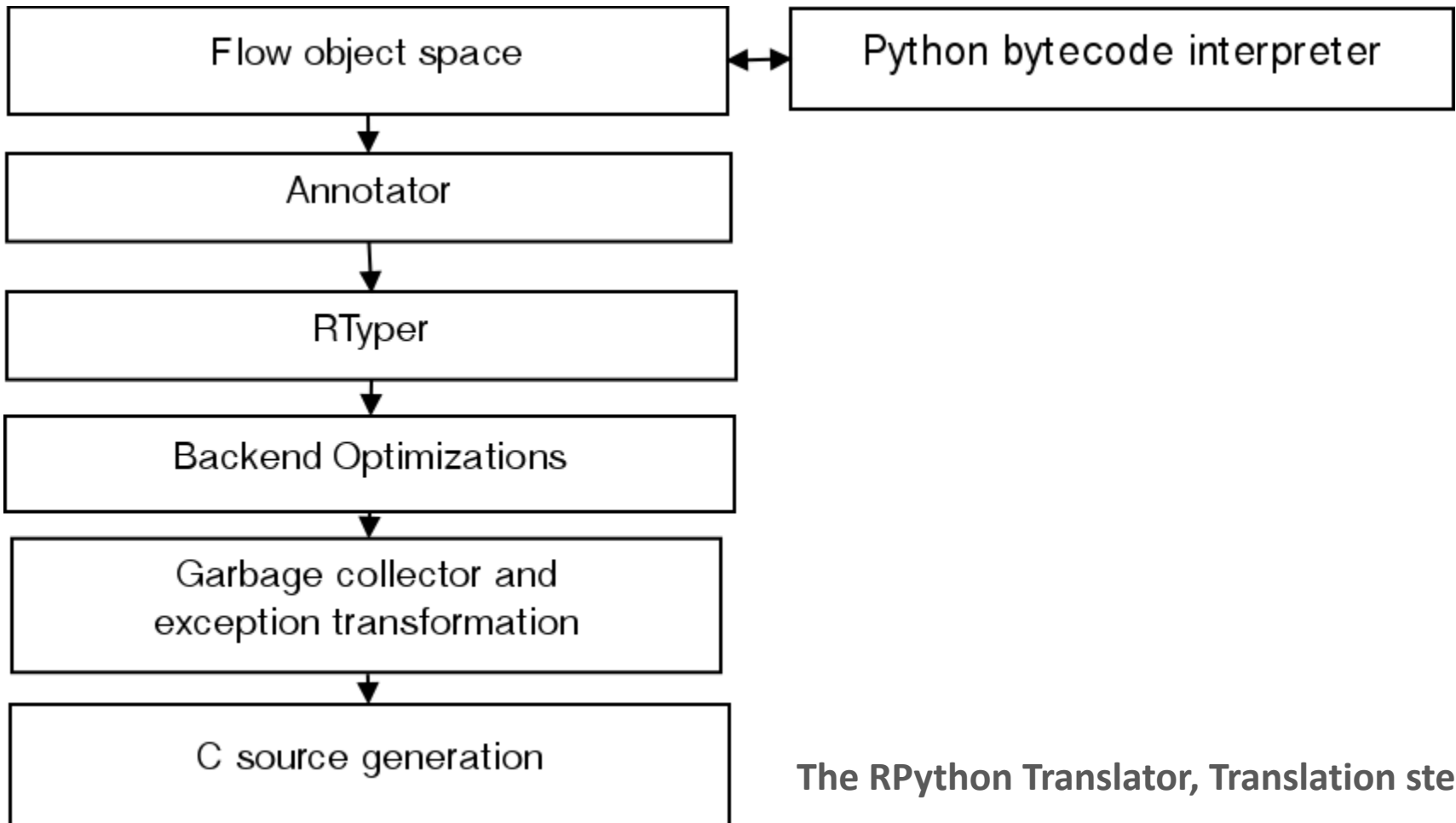
# Architectural style (pattern)



Source: codeproject.org

# Bash Component Architecture





The RPython Translator, Translation steps



# Compiler alternatives (Garlan and Shaw)

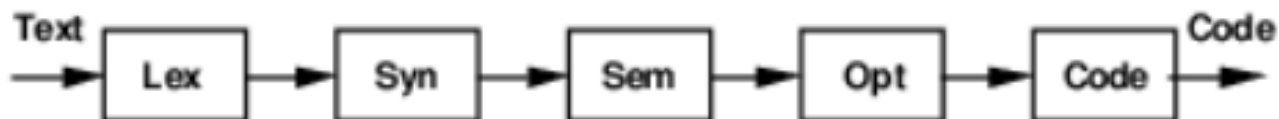


Figure 15: Traditional Compiler Model

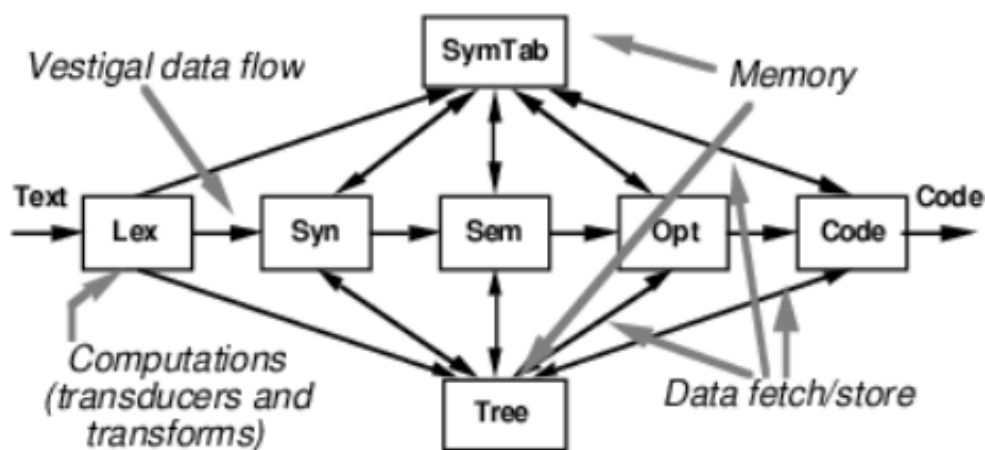


Figure 17: Modern Canonical Compiler

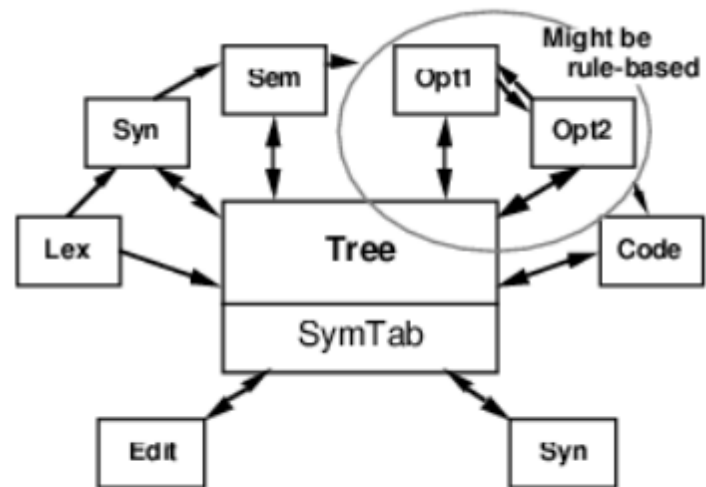


Figure 18: Canonical Compiler, Revisited

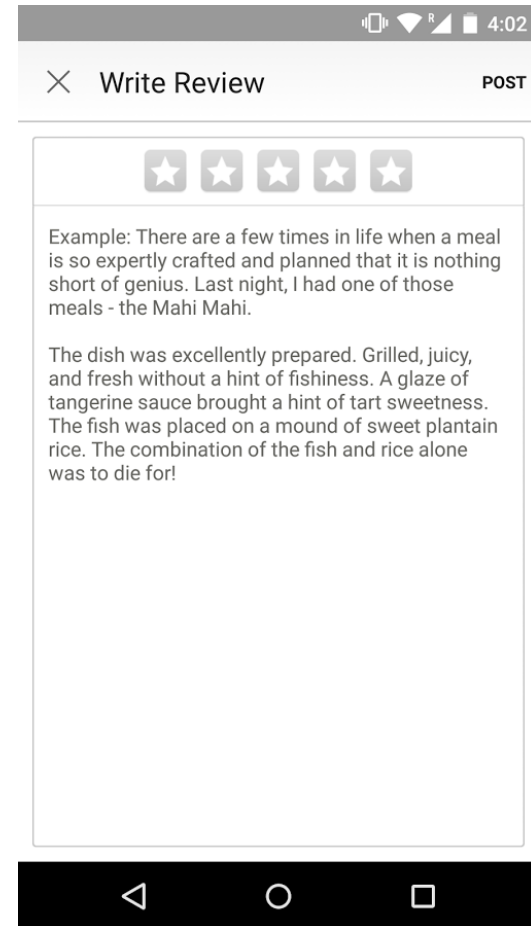
Client

Server

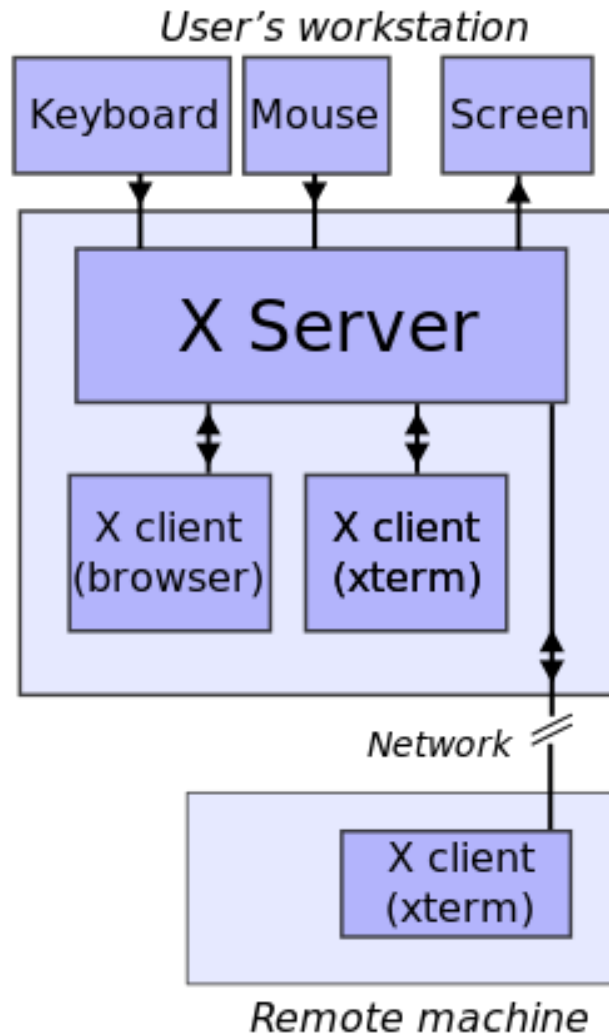
Database

Where to  
validate user  
input?

## Example: Yelp App

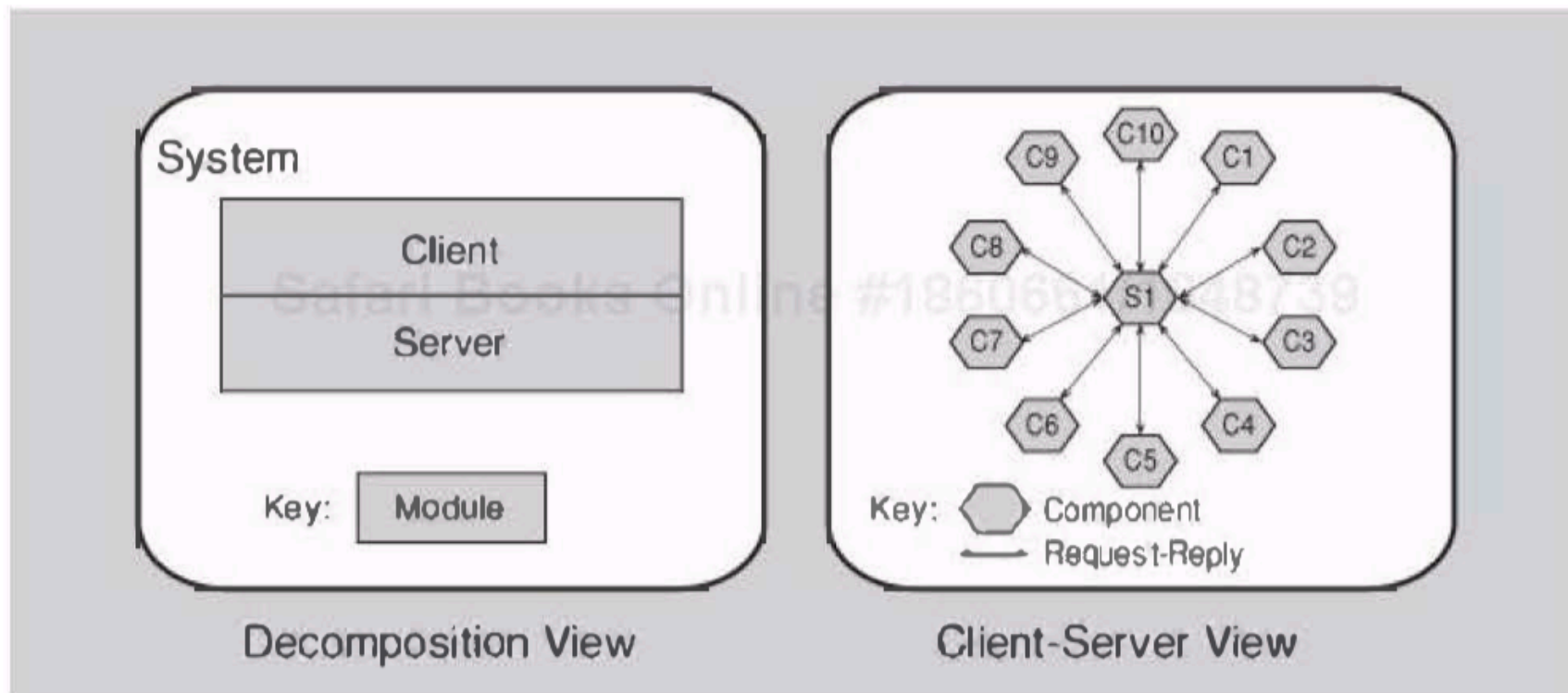


# Client-server style



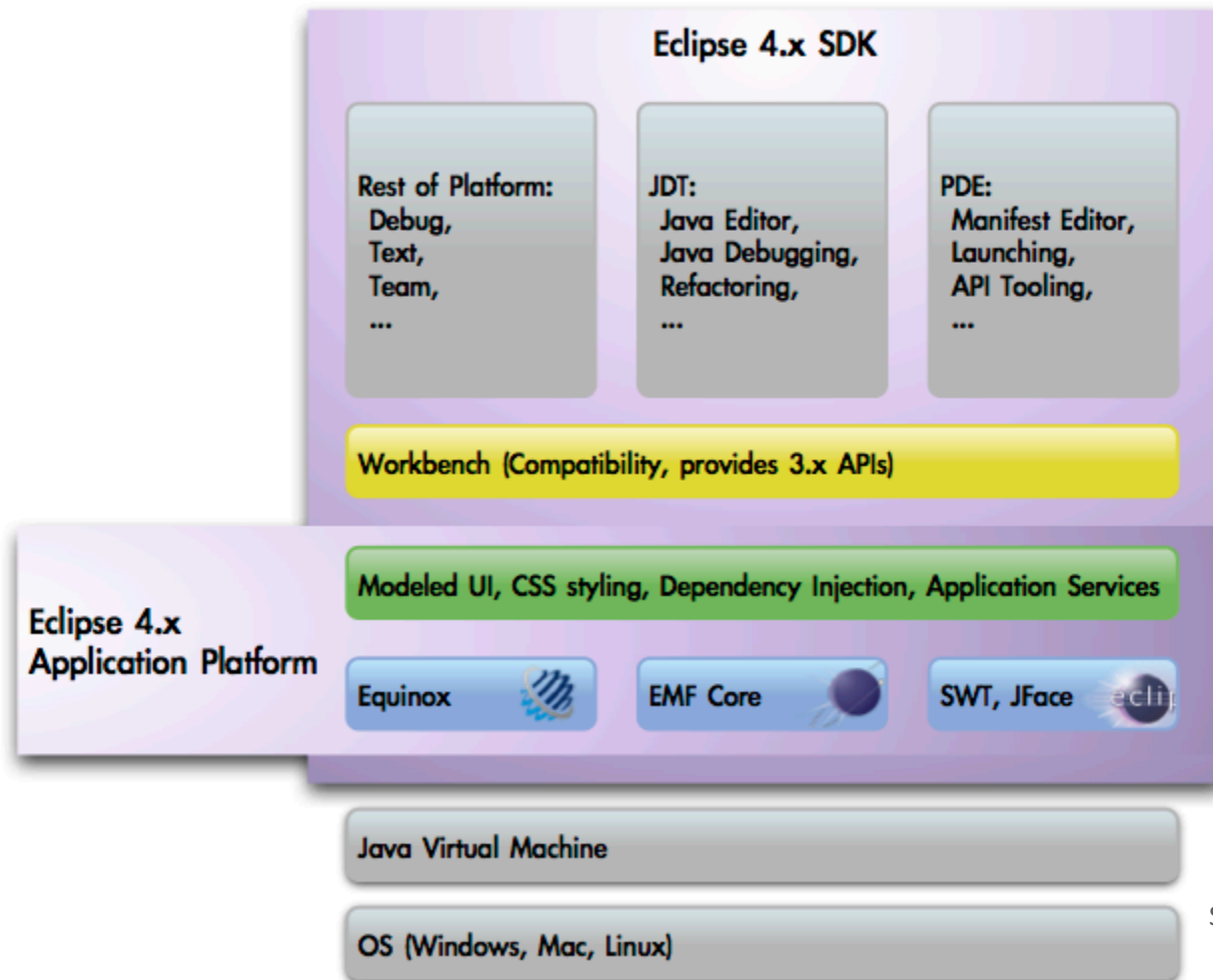
Source: wikimedia commons

# Two views of a client-server system



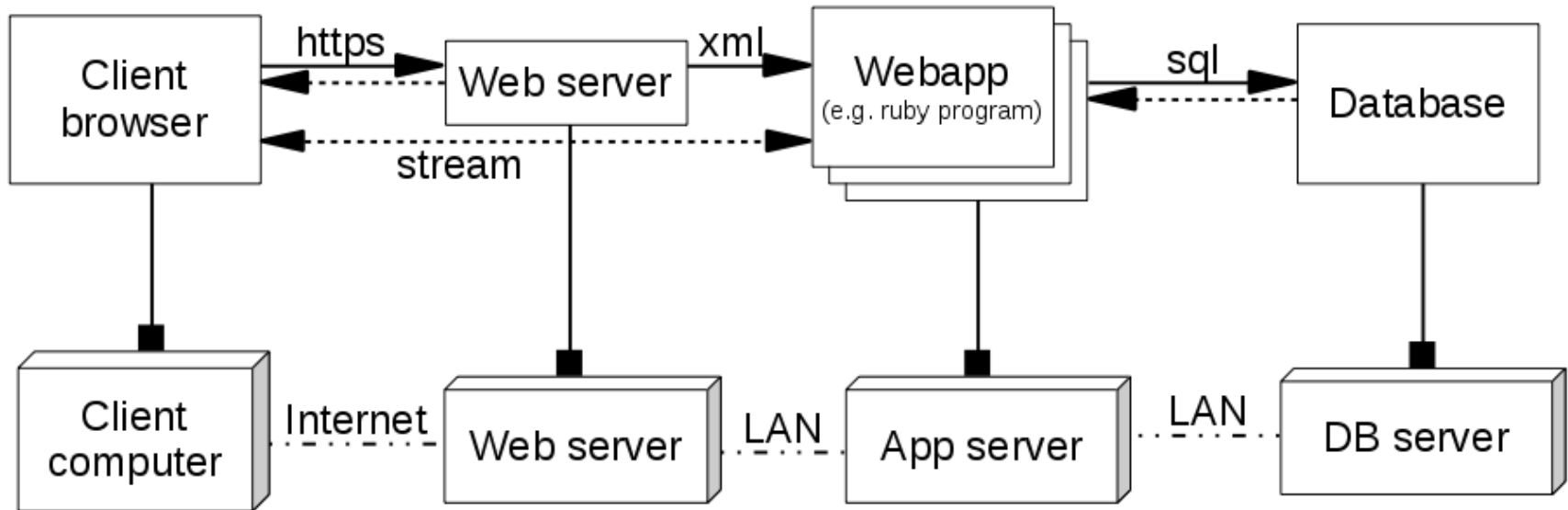
**Figure 1.2. Two views of a client-server system**

# Layered system



Source: eclipse.org

# Tiered architecture



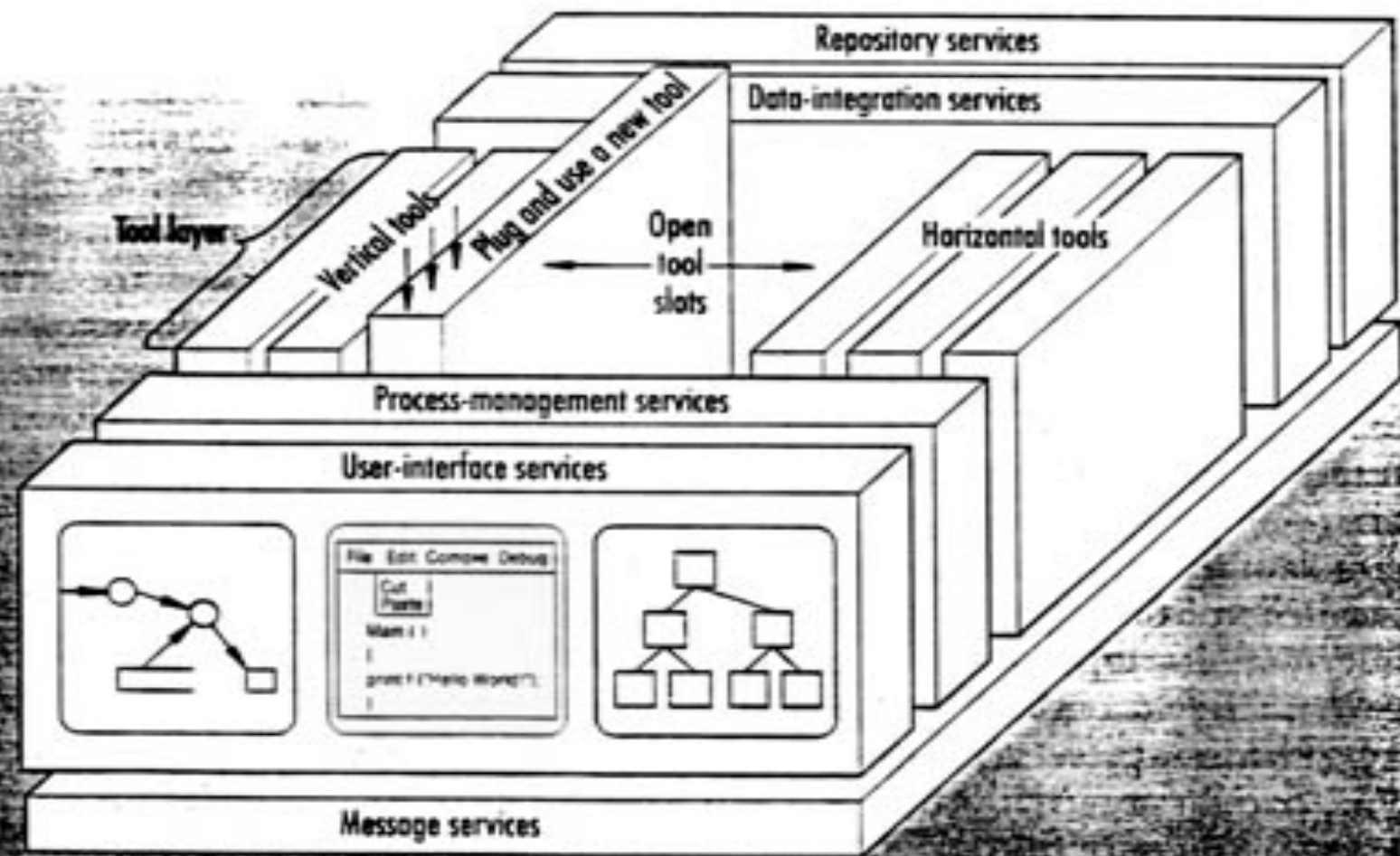
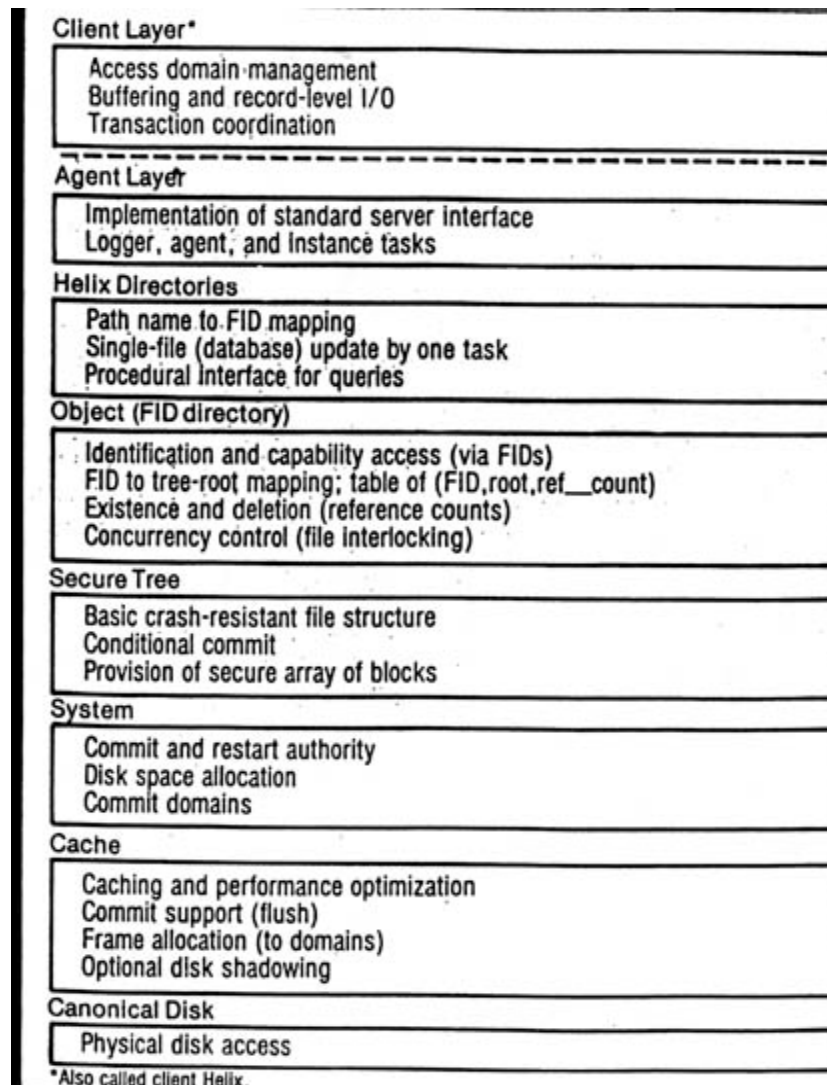


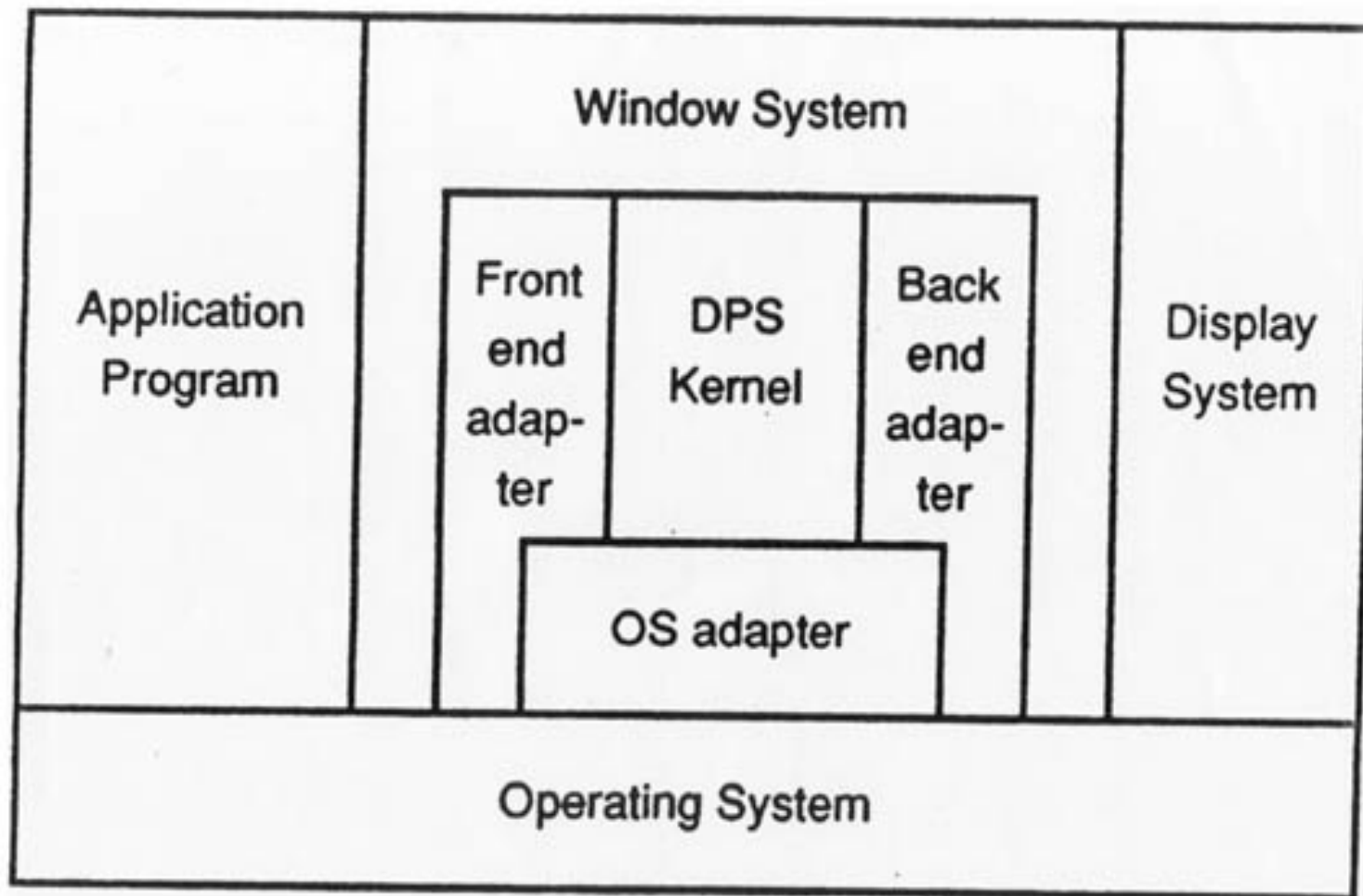
Figure 1. The NIST/ECMA reference model.



**Figure 2. Abstraction layering.**

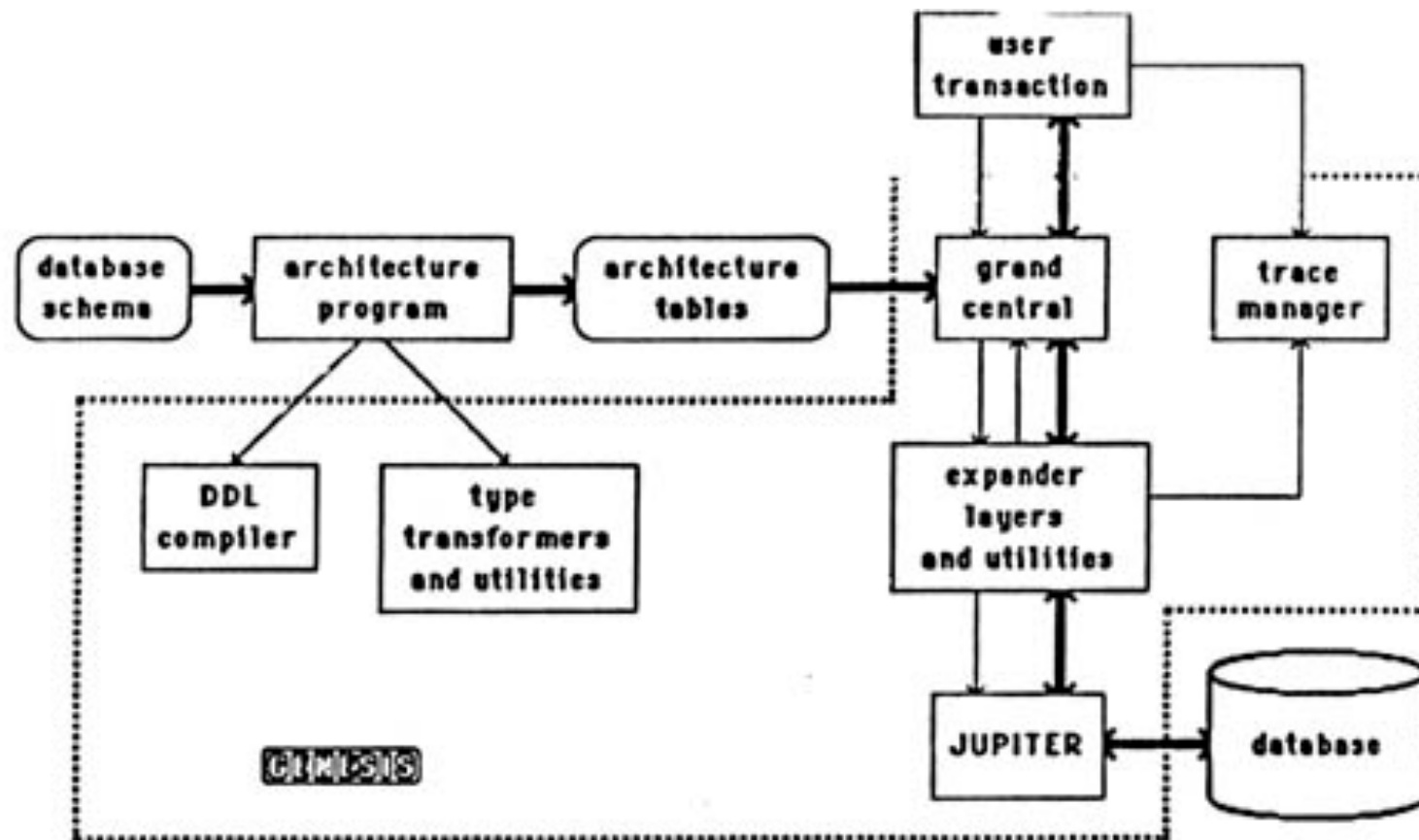
IEEE Software, "Helix: The architecture of the XMS Distributed File System, Marek Fridrich and William Older, May 1985, Vol. 2, No. 3, P. 23







**Figure 2. Display PostScript interpreter components.**

An Overview of the DISPLAY POSTSCRIPT™ System, Adobe Systems Incorporated, March 16, 1988, P. 10

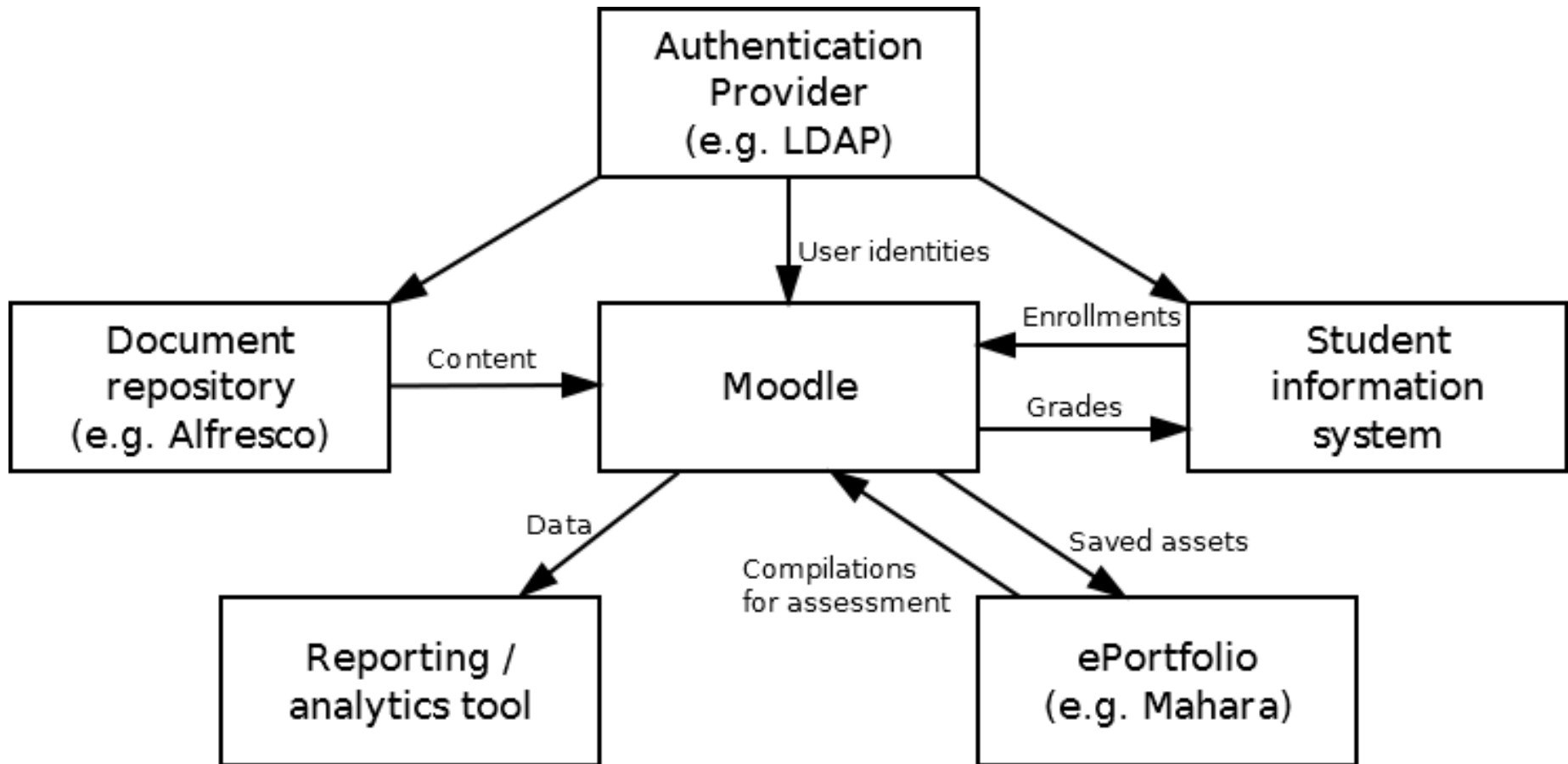


**Legend**

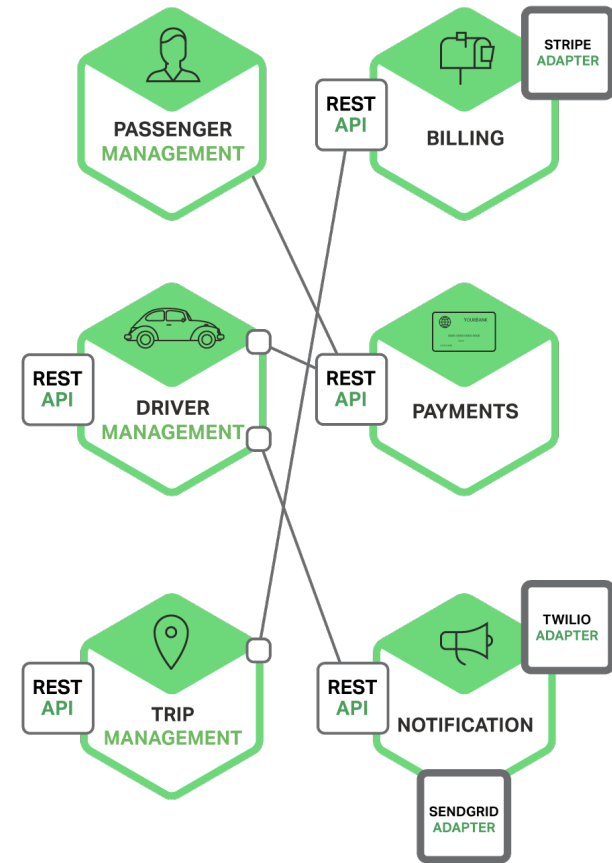
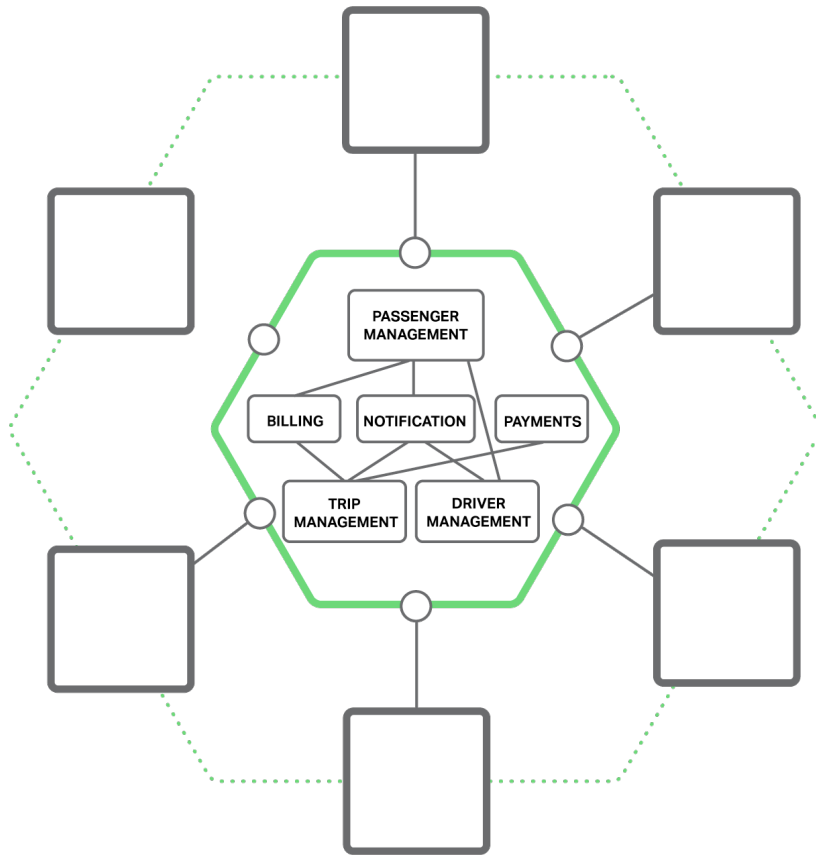
- |   |                   |       |           |
|---|-------------------|-------|-----------|
|  | module or program | A → B | A calls B |
|  | schema or tables  | A → B | data path |

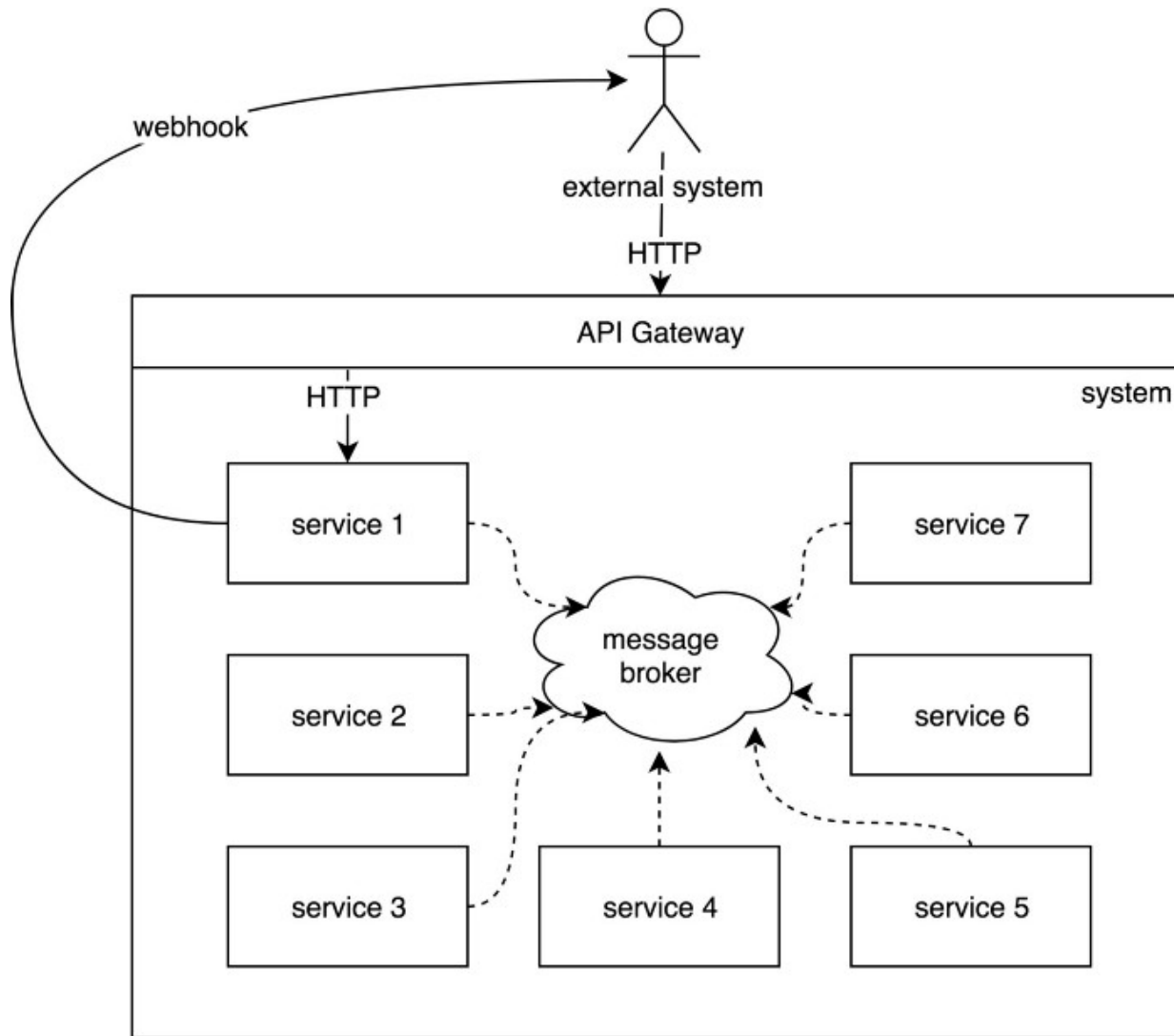
**Figure 3.1 The Configuration of the GENESIS Prototype**

Genesis: A Reconfiguration Database Management System, D. S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise, Department of Computer Sciences, University of Texas at Austin,



Moodle: Typical university systems architecture – Key subsystems





# Architectural Style?

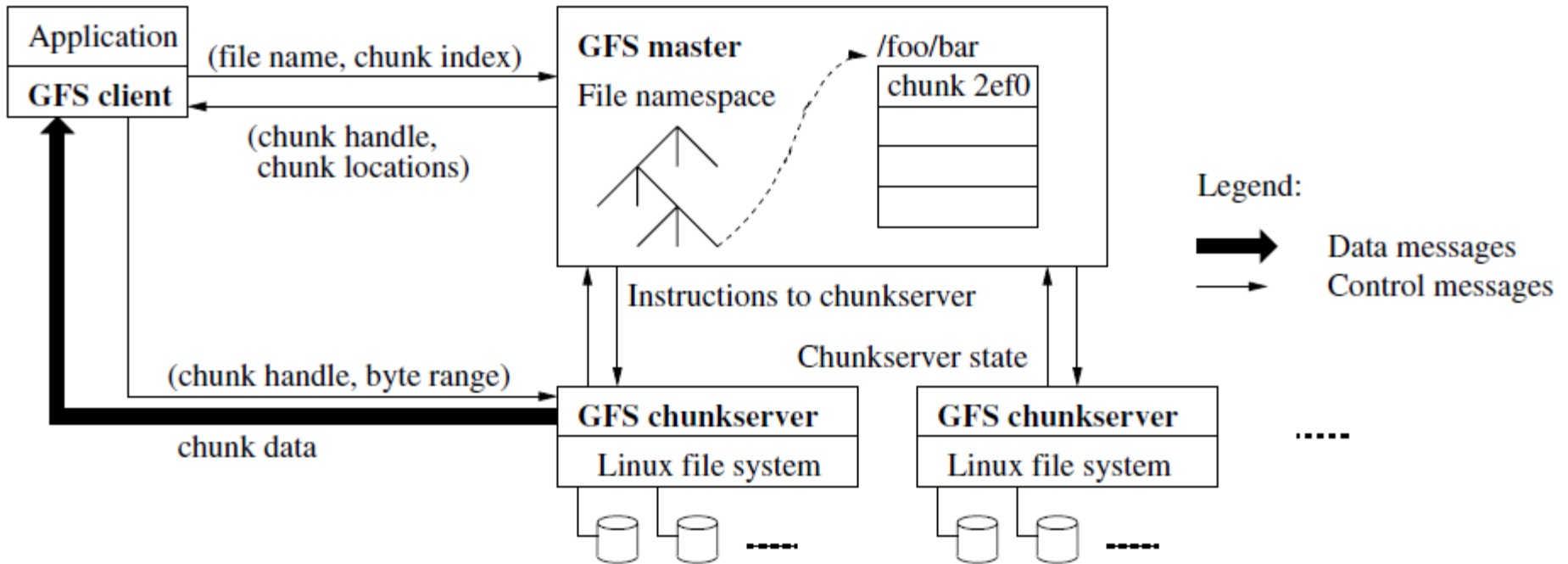


Figure 1: GFS Architecture

# ARCHITECTURAL TACTICS

# Tactics

- Architectural techniques to achieve qualities
  - More tied to specific context and quality
- Smaller scope than architectural patterns
  - Problem solved by patterns: “How do I structure my (sub)system?”
  - Problem solved by tactics: “How do I get better at quality X?”
- Collection of common strategies and known solutions
  - Resemble OO design patterns



**Prioritization**

**Concurrency**

**Nondeterminism**

**Redundancy**

**Authorization**

**Synchronization**

**Voting**

**Cohesion**

**Encryption**

**Separation**

**Authentication**

**Transactions**

**Sandboxing**

**Undo**

**Timestamps**

**Coupling**

**Sanitation**

**Auditing**

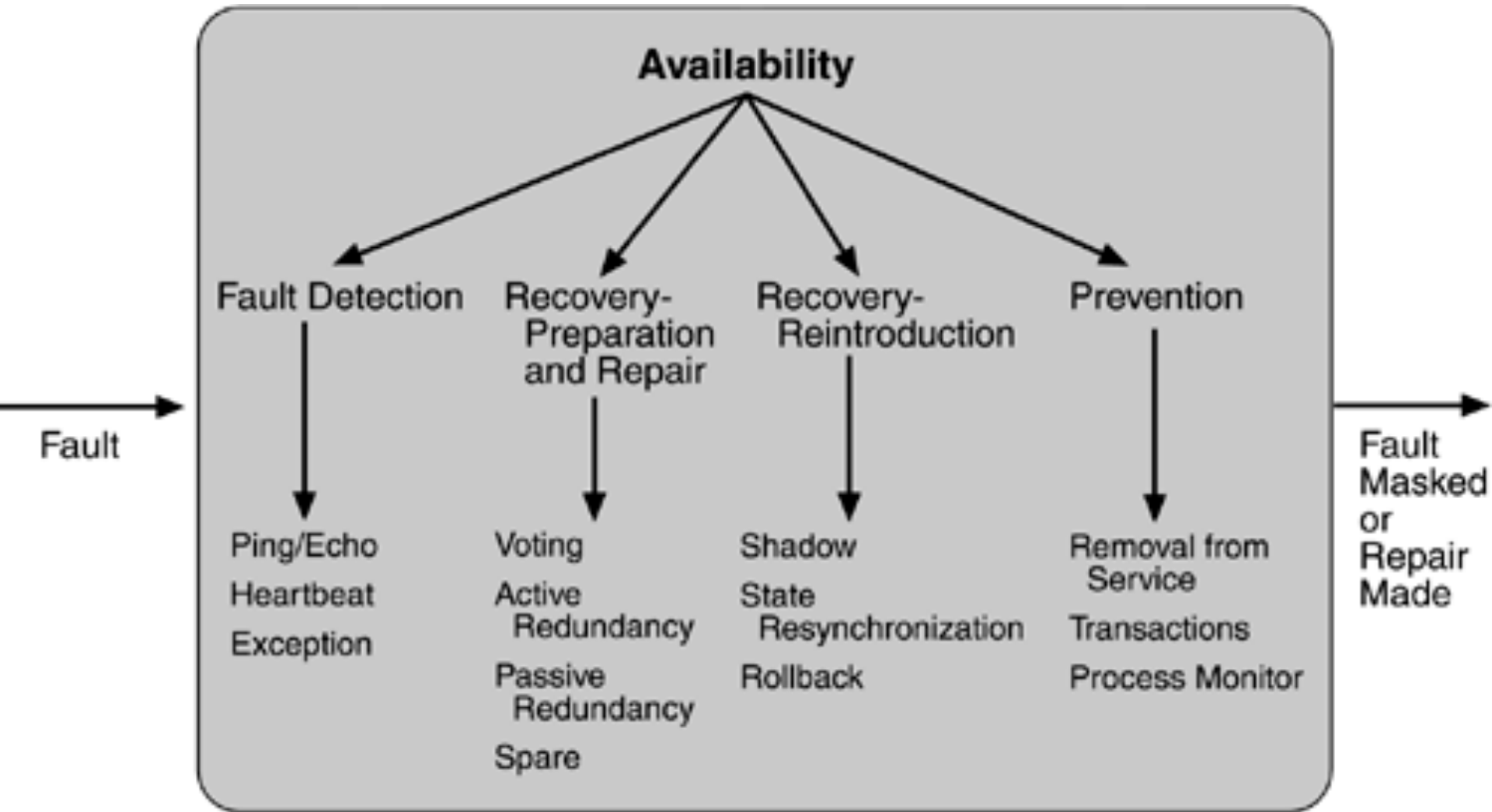
# Example Tactic Description: Record/playback

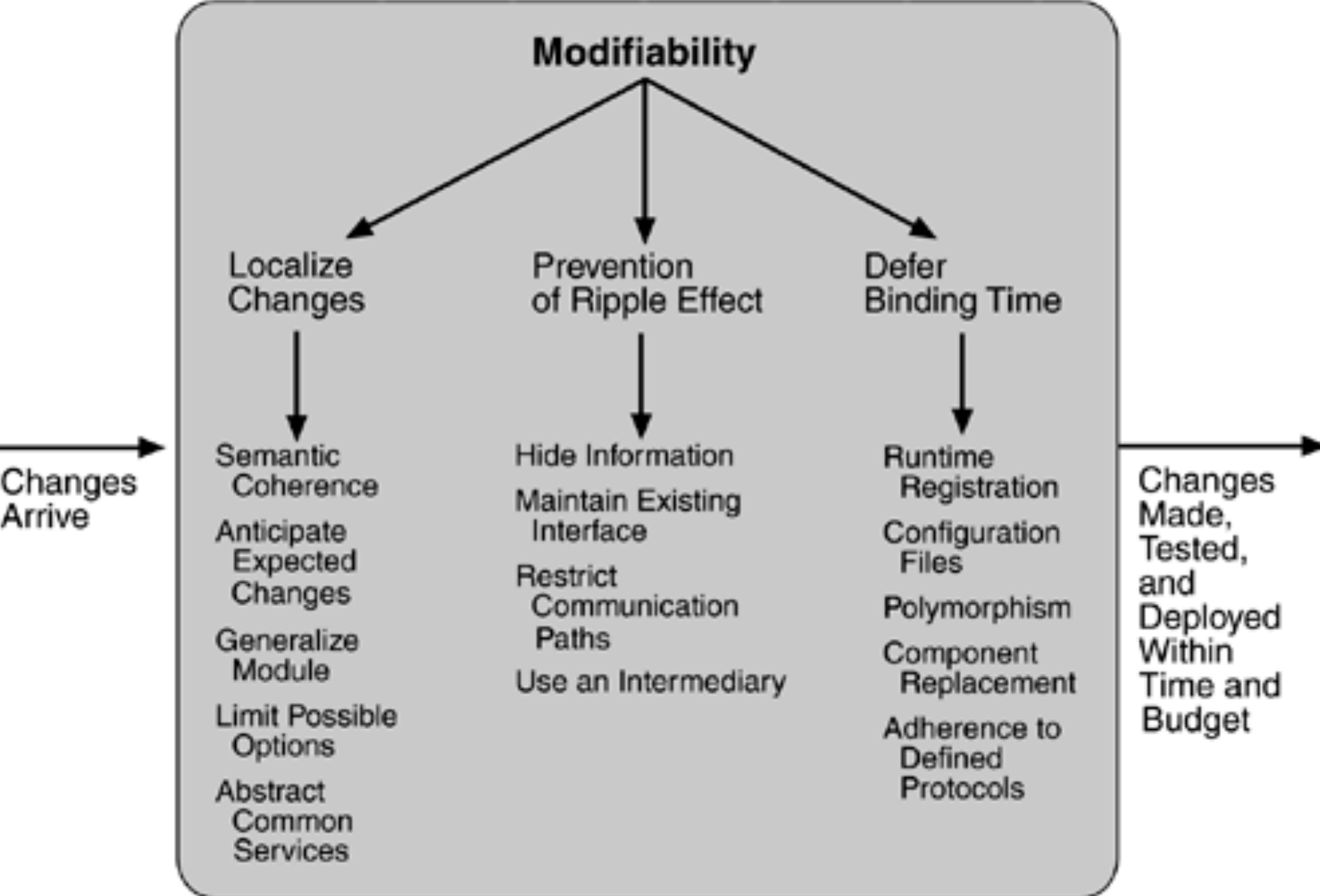
- Record/playback refers to both capturing information crossing an interface and using it as input into the test harness. The information crossing an interface during normal operation is saved in some repository and represents output from one component and input to another. Recording this information allows test input for one of the components to be generated and test output for later comparison to be saved.

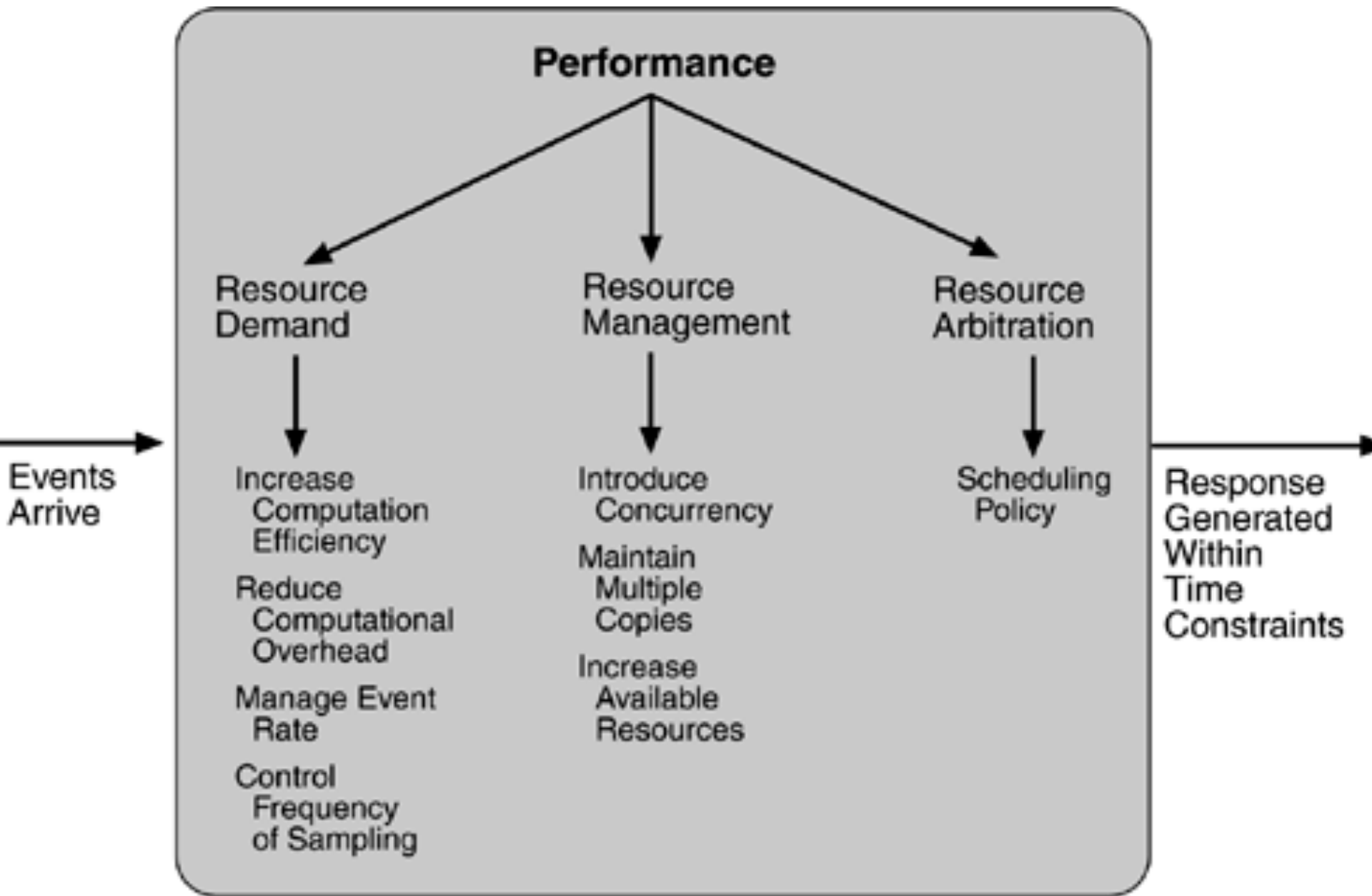
# Example Tactic Description:

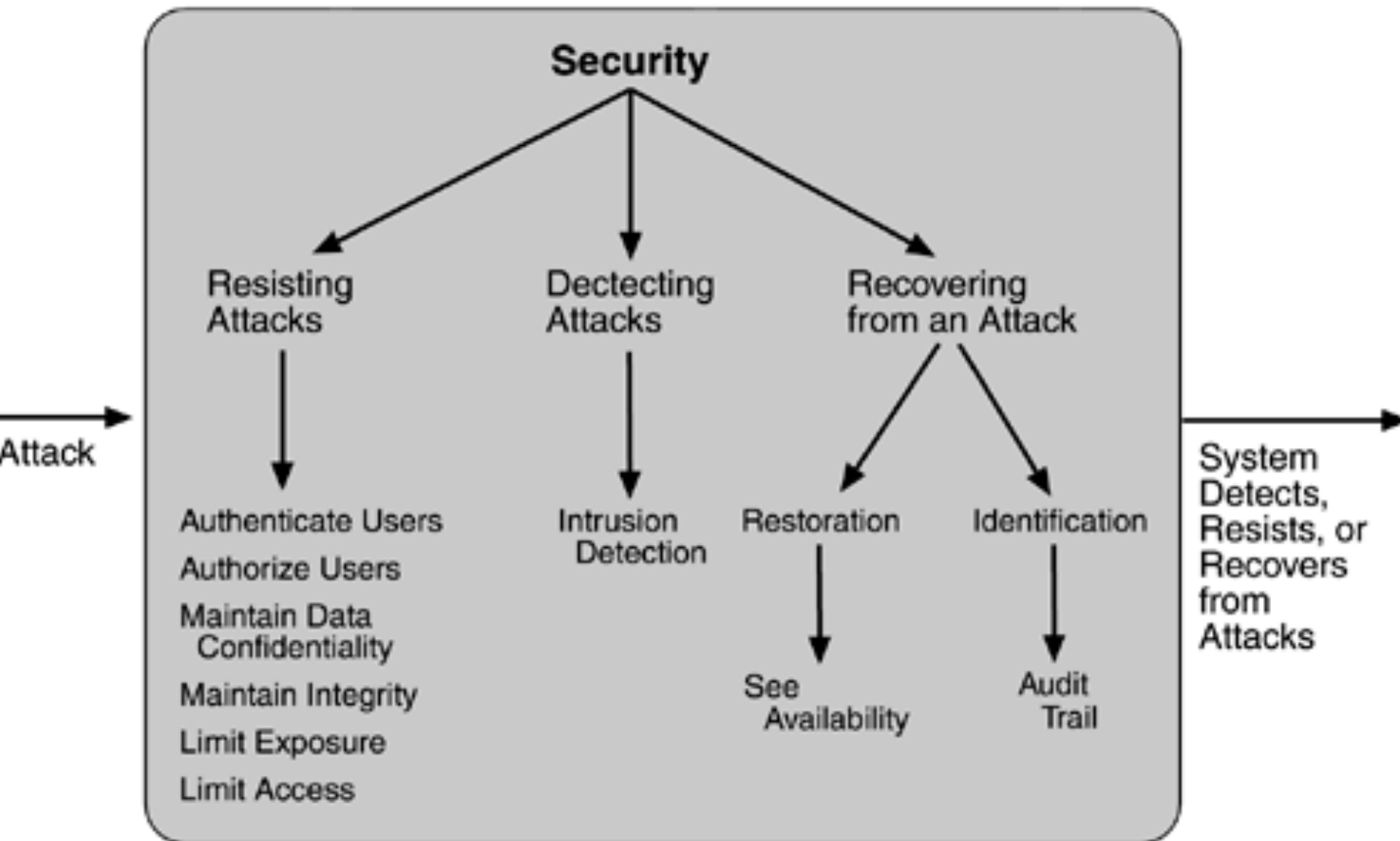
## Built-in monitors

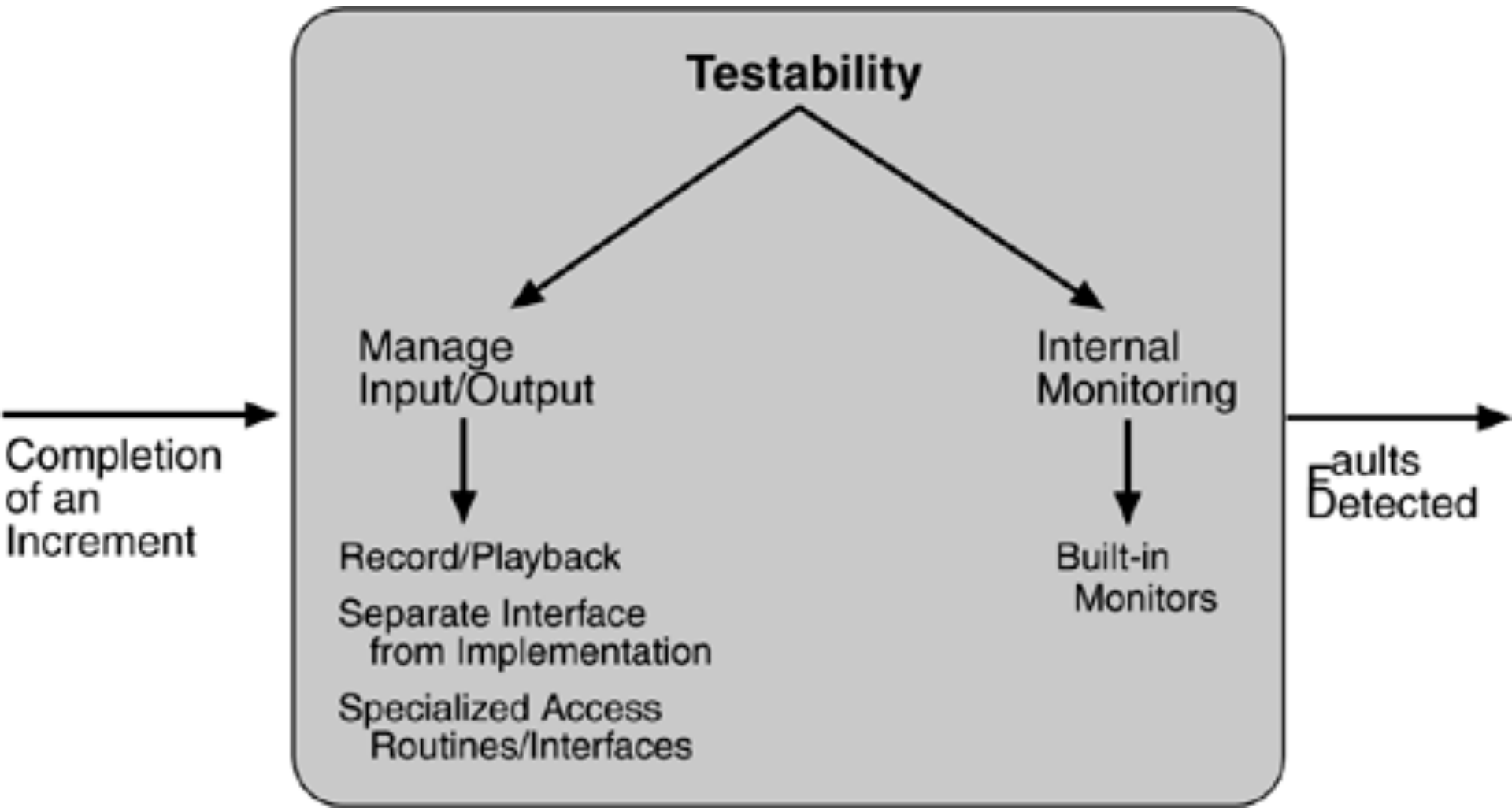
- The component can maintain state, performance load, capacity, security, or other information accessible through an interface. This interface can be a permanent interface of the component or it can be introduced temporarily via an instrumentation technique such as aspect-oriented programming or preprocessor macros. A common technique is to record events when monitoring states have been activated. Monitoring states can actually increase the testing effort since tests may have to be repeated with the monitoring turned off. Increased visibility into the activities of the component usually more than outweigh the cost of the additional testing.



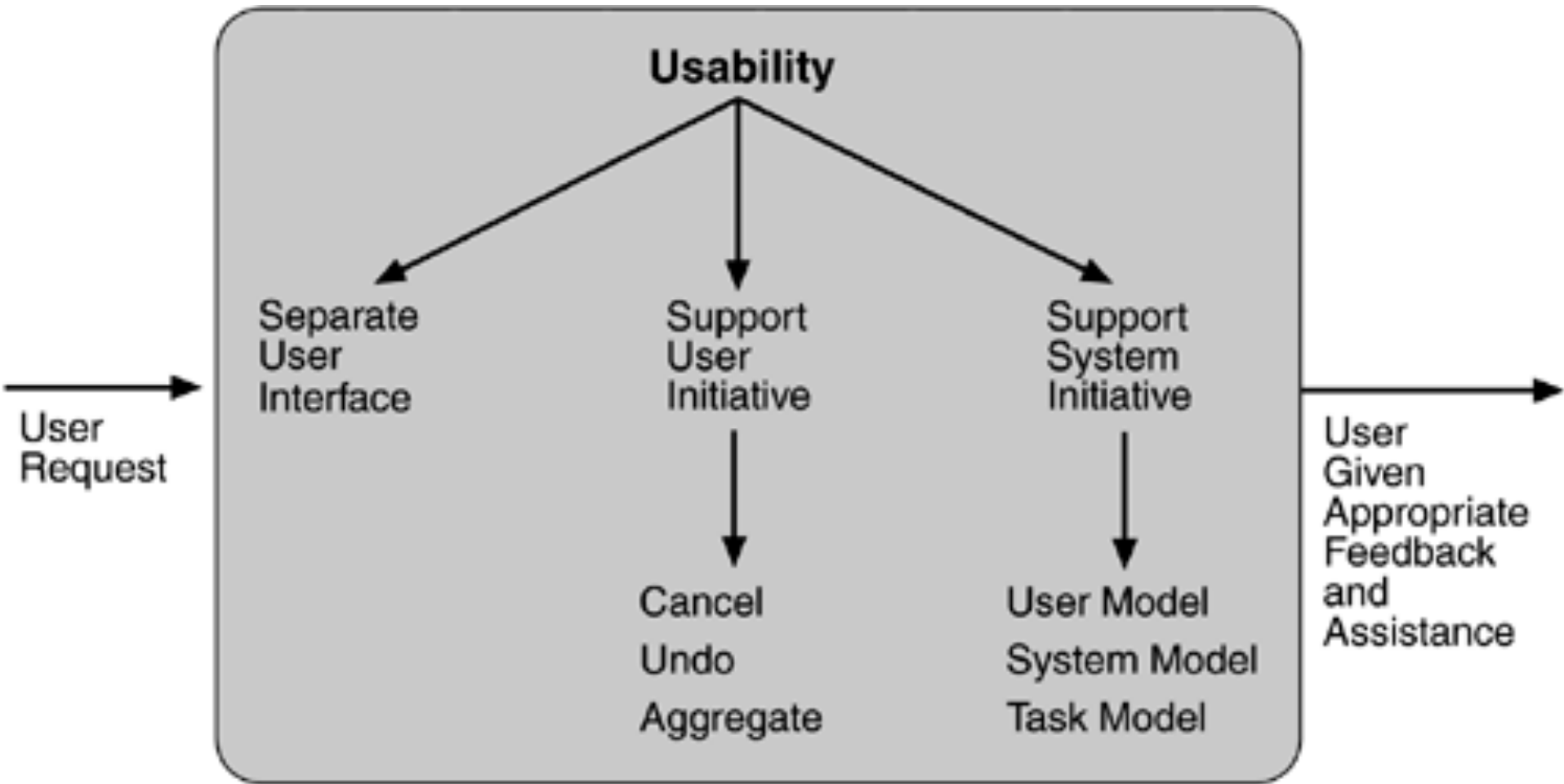


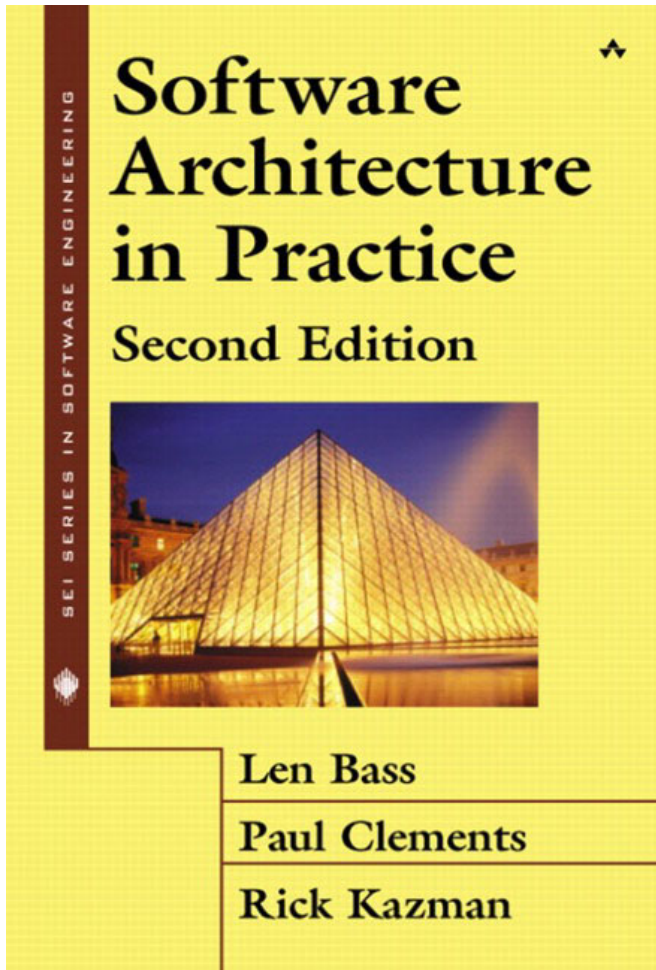












Many tactics described in Chapter 5

Brief high-level descriptions (about 1 paragraph per tactic)

Second and more detailed third edition available as ebook

# Why write a design doc?

The image shows a screenshot of a Twitter thread. The top tweet is from Claire Le Goues (@clegoues) with the text: "Hey there, practicing engineers in my timeline: imagine you were teaching the software engineers of tomorrow". The middle tweet is from Adrienne Porter Felt (@\_apf\_) dated Oct 3, replying to @clegoues. Her text says: "Don't trust anything you read because it's probably outdated ... think of it as a helpful hint and not an authoritative source of Truth". A red rectangular box highlights the text: "When writing documentation, put a date on it". The bottom tweet is from Titus Barik (@barik) dated Oct 3, replying to @clegoues and @adolfont. His text says: "Good design documents are also good design rationales. They tell you why certain choices were made, even if you don't agree with them." The interface includes profile pictures, avatars, and interaction icons like replies, retweets, and likes.

Claire Le Goues @clegoues  
Hey there, practicing engineers in my timeline: imagine you were teaching the software engineers of tomorrow

Adrienne Porter Felt @\_apf\_ · Oct 3  
Replying to @clegoues  
Don't trust anything you read because it's probably outdated ... think of it as a helpful hint and not an authoritative source of Truth  
When writing documentation, put a date on it

Titus Barik @barik · Oct 3  
Replying to @clegoues and @adolfont  
Good design documents are also good design rationales. They tell you why certain choices were made, even if you don't agree with them.

# Why write a design doc?

- Communicate among and with stakeholders in a system.
  - Receive guidance on present decisions
  - Resolve uncertainty
  - Inform future engineers dealing with your system.
- Overall: make sure you are implementing the right thing, and also *not* implementing the *wrong* thing.
- A key element of design documentation is ***feedback***
  - (which is probably why so many orgs use Google Docs...)
- Happy side effect: writing it out can help you clarify your own design thinking, as you specify:
  - What are you going to do?
  - Why are you doing it that way?
  - What assumptions/tradeoffs are you making?

# Who is a design doc *for*?

- Effective communication starts by considering the audience.
- Several possible intended audiences; in this class, we focus on documents intended for generally technical stakeholders:
  - Other engineers on your team
  - Other engineers on other teams
  - Technical/project/product managers
  - Yourself, in the future.
- The design doc should be accessible to an informed and competent engineer in your organization.

# Common pitfalls

- Too little detail, especially on *rationale*.
- Too MUCH detail, especially on the specifics of code.
- Impenetrable diagrams that Future Reader will not be able to discern.
- Documentation that fails to evolve with the real system.

# Common parts/templates

- Overview/feature description: what problem is being solved?
  - High-level requirements, both functional and quality
- Background/key terms
- Goals/non goals
- Design alternatives, tradeoffs, assumptions
- Decision
- Other considerations/elements of design

# Examples: SourceGraph RFCs

<https://about.sourcegraph.com/handbook/communication/rfc>



# 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.