Architectural Styles–Outline


2. Several Example Styles/Tradeoffs

3. A practical Example: ALrm Monitoring software.
Example: A Compiler

Perry & WOlf

Two possible configurations, same components, different connectors.
Architectural Styles

"An Architectural Style . . . defines a *vocabulary* of components and connector types, and a set of *constraints* on how they can be combined . . . there may also exist one or more *semantic models* that specify how to determine a system’s overall properties from the properties of it’s parts”

—Garlan & Shaw

Describing Architectures

1. *vocabulary* types of components & connectors.
2. allowable *structural patterns*
3. underlying *computational model*
4. *invariants* of this style.
5. *common examples* of use.
6. *advantages* and *disadvantages*
**Pipes and Filters Style**

Examples Unix pipes/filters; some batch processing systems; HTTP clients/servers arranged as content filters.

**Vocabulary:** Components → filters and connectors → pipes.

**Computational Model:** Pipes are “data streamers”. Data is processed when available; filters are concurrent processes.

**Invariants:** Filters mutually unaware. Data types are uniform across the pipe line. Filters terminate when EOF is reached or when connection is closed. No state shared across filters.

**Design Trade-offs:** Must consider:

+ve Simple interface and composition; filters reusable, and separately evolvable; can be analyzed for thoughtput, resource consumption etc; naturally concurrent execution.
-ve Each filter is monolithic; not compatible with interactive computing; because of data uniformity, may force “lowest common denominator” data format. Communication patterns are restricted.
Layered Style

Examples: TCP/IP Protocol Stack, Java Virtual Machine

Vocabulary: component → layers connectors → calls/requests

Computational Model: Layers implement services for “enclosing” layers as procedure calls (or Synchronous RPC) using services provided by “enclosed” layers as clients.

Invariants: Layer service requests go only “inside” and only to the immediately enclosed layer.

Design Trade-offs: Must Consider:

+ve Increasing, well-defined layers of abstraction; separation of concerns; greater evolvability.

-ve Levels of abstractions may not always be clear; optimization difficulties (e.g., partial evaluation).
**Bulletin-Board Style**

Examples: Compilers; rule based systems; file systems; chat groups.

Vocabulary: components → knowledge sources  connectors → blackboard datastructure(s) (BB)

Computational Model: Knowledge sources “post” updates to BBDS; other knowledge sources can then observe this and respond with their updates

Invariants: Knowledge sources find out updates in order that they happen; all knowledge sources are “aware” of all data.

Design Trade-offs: Must Consider:

+ve Suitable for evolving applications; new types of data can easily be added. Promises “modular” addition of new knowledge sources.
-ve Effects of an update can be difficult to trace through the system.
Event-Based Style

Examples: Office Automation; monitoring/alarm systems; distributed software configuration; GUls; trading systems.

Vocabulary: components → sources, sinks connector → event channels, multicast groups etc.

Computational Model: Sources generate events; sinks that register events are notified of events.

Invariants: Sources don’t know all the “subscribers”

   event generation and notification are asynchronous.

Design Trade-offs: Must Consider:

+ve Late binding! Components, new events can be introduces at any time.

-ve Event sources have no control over the computational ramifications of
event generation; no centralized control; deadlocks, message “saturation” etc are possible
Client-Server Systems

Examples: Web, POP, SMTP, etc.

Vocabulary: components $\rightarrow$ client, server; connector can vary.

Computational Model: Server is persistent,

Clients initiate, use synch. calls.

Invariants: server location “known” to clients; many clients share a server; servers highly reliable, fast, etc; clients are not.

Design Tradeoffs Consider:

$+$ Scaleability; simple concurrency control; widely used

$-$ “Fat”, hard to maintain/evolve clients; administrative headaches; single point of failure; can be difficult to maintain/evolve.

Three-Tier Systems

Examples: Auction/E-commerce Servers.

Vocabulary: components $\rightarrow$ client, middle/business logic, DB/legacy; connector can vary.

Computational Model: middle/DB layers are persistent and shared. Clients initiate.

Invariants: “Thin” clients. Middle-tier has concurrency control etc. Legacy systems “wrapped” for inter-operability.

Design Tradeoffs Consider:

$+$ “Thin” Clients easy to care for. Separation of Concerns. Inter-operability, reuse/leverage.
-ve Points of Failure; middle layer complexity (e.g., SAP, SSA); wrappers hard to build/care for.
Case Study- Event Monitoring System

What’s a good high-level architecture?
Just one Monitoring Element

Monitoring System

Rule 1
Rule 2
Rule 3
Rule n

Model of monitored object

Events

A Hybrid architecture?