Software Architecture

Bertrand Meyer, Michela Pedroni

ETH Zurich, February-May 2010

Lecture 15: Architectural styles

(after material prepared by P. Müller)
Software architecture styles

Work by Mary Shaw and David Garlan at Carnegie-Mellon University, mid-90s

Aim similar to Design Patterns work: classify styles of software architecture
Characterizations are more abstract; no attempt to represent them directly as code
Software Architecture styles

An architectural style is defined by

- Type of basic architectural components (e.g. classes, filters, databases, layers)

- Type of connectors (e.g. calls, pipes, inheritance, event broadcast)
Architectural styles: examples

Concurrent processing
Dataflow: batch, pipe-and-filter
Object-oriented
Call-and-return: functional, object-oriented
Independent components: interacting processes, event-based
Data-centered (repositories): database, blackboard
Hierarchical
Interpreters, rule-based
Client-server
Peer-to-peer
Concurrent processing

Take advantage of virtual or physical parallelism to split computation into several parts

Variations:
- Processes
- Threads
Concurrent processing: discussion

**Strengths:**
- Separation of concerns
- Increased performance
- Provide users with ability to perform several tasks in parallel (example: browser tabs)

**Weaknesses:**
- Difficulty of synchronization: data races, deadlocks
- Must find out what is parallelizable
- Limits to performance improvement: Amdahl’s law
Amdahl’s Law

\[ \text{speedup} = \frac{\text{old execution time}}{\text{new execution time}} \]

...of computation given \( n \) CPUs instead of 1
Amdahl’s law

What is the performance gain in going from 1 to \( n \) processors?

Assume that \( p \) (with \( 0 \leq p \leq 1 \)) is the portion of the program code that can be parallelized:

\[
\frac{1}{(1-p) + \frac{p}{n}}
\]
Amdahl’s law in practice

Source: Wikimedia commons
Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

\[
\text{speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}
\]

*This and next 4 slides from M. Herlihy, Brown Univ.
Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

\[
speedup = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}
\]
Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

\[
speedup = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}
\]
Example

- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?

\[
speedup = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}
\]
The moral

- Making good use of our multiple processors (cores) means finding ways to effectively parallelize our code
  - Minimize sequential parts
  - Reduce idle time in which threads wait without doing something useful.
Dataflow systems

Availability of data controls the computation
The structure is determined by the orderly motion of data from component to component

Variations:
- Control: push versus pull
- Degree of concurrency
- Topology
Dataflow: Batch-Sequential

Frequent architecture in scientific computing and business data processing

Components are independent programs
Connectors are media, typically files
Each step runs to completion before next step begins
Batch-Sequential

History: mainframes and magnetic tape

Business data processing

- Discrete transactions of predetermined type and occurring at periodic intervals
- Creation of periodic reports based on periodic data updates

Examples

- Payroll computations
- Tax reports
Dataflow: Pipe-and-Filter

Components (Filters)
- Read input stream (or streams)
- Locally transform data
- Produce output stream (or streams)

Connectors (Pipes)
- Streams, e.g., FIFO buffer
Pipe-and-Filter

Data processed *incrementally* as it arrives
Output can begin before input fully consumed

Filters must be *independent*:
- Filters do not share state
- Filters do not know upstream or downstream filters

Examples
- lex/yacc-based compiler (*scan, parse, generate...*)
- Unix pipes
- Image / signal processing
Push pipeline with active source

Source of each pipe pushes data downstream
Example with Unix pipes:

```
grep p1 * | grep p2 | wc | tee my_file
```
Pull pipeline with active sink

- Sink of each pipe pulls data from upstream
- Example: Compiler: \( t := \text{lexer.next_token} \)
Combining push and pull

If more than one filter is pushing / pulling, synchronization is needed
Pipe-and-Filter: discussion

**Strengths:**
- **Reuse:** any two filters can be connected if they agree on data format
- **Ease of maintenance:** filters can be added or replaced
- **Potential for parallelism:** filters implemented as separate tasks, consuming and producing data incrementally

**Weaknesses:**
- Sharing global data expensive or limiting
- Scheme is highly dependent on order of filters
- Can be difficult to design incremental filters
- Not appropriate for interactive applications
- Error handling difficult: what if an intermediate filter crashes?
- Data type must be greatest common denominator, e.g. ASCII
Call and return: functional

Components: Routines
Connectors: Routine calls

Key aspects
- Routines correspond to units of the task to be performed
- Combined through control structures
- Routines known through interfaces (argument list)

Variations
- Objects as concurrent tasks
Functional call-and-return

Strengths:
- Architecture based on well-identified parts of the task
- Change implementation of routine without affecting clients
- Reuse of individual operations

Weaknesses:
- Must know which exact routine to change
- Hides role of data structure
- Does not take into account commonalities between variants
- Bad support for extendibility
Call and return: object-oriented

Components: Classes
Connectors: Routine calls
Key aspects
- A class describes a type of resource and all accesses to it (encapsulation)
- Representation hidden from client classes
Variations
- Objects as concurrent tasks
O-O call-and-return

Strengths:
- Change implementation without affecting clients
- Can break problems into interacting agents
- Can distribute across multiple machines or networks

Weaknesses:
- Objects must know their interaction partners; when partner changes, clients must change
- Side effects: if $A$ uses $B$ and $C$ uses $B$, then $C$'s effects on $B$ can be unexpected to $A$
Event-Based (Publish-Subscribe)

A component may:

- Announce events
- Register a callback for events of other components

Connectors are the bindings between event announcements and routine calls (callbacks)
Event-Based style: Properties

Publishers of events do not know which components (subscribers) will be affected by those events.

Components cannot make assumptions about ordering of processing, or what processing will occur as a result of their events.

Examples

- Programming environment tool integration
- User interfaces (Model-View-Controller)
- Syntax-directed editors to support incremental semantic checking
Event-Based Style: example

Integrating tools in a shared environment

Editor announces it has finished editing a module

- Compiler registers for such announcements and automatically re-compiles module
- Editor shows syntax errors reported by compiler

Debugger announces it has reached a breakpoint

- Editor registers for such announcements and automatically scrolls to relevant source line
Event-Based: discussion

**Strengths:**
- Strong support for reuse: plug in new component by registering it for events
- Maintenance: add and replace components with minimum effect on other components in the system

**Weaknesses:**
- Loss of control:
  - What components will respond to an event?
  - In which order will components be invoked?
  - Are invoked components finished?
- Correctness hard to ensure: depends on context and order of invocation
Data-Centered (Repository)

Components
- Central data store component represents state
- Independent components operate on data store

![Diagram of Data-Centered (Repository) components]

Knowledge Source

Repository

Direct access

Knowledge Source

Computation
Special Case: Blackboard Architectures

Interactions among knowledge sources solely through repository

Knowledge sources make changes to the shared data that lead incrementally to solution
Control is driven entirely by the state of the blackboard

Example

- Repository: modern compilers act on shared data: symbol table, abstract syntax tree
- Blackboard: signal and speech processing
Data-Centered: discussion

Strengths:
- Efficient way to share large amounts of data
- Data integrity localized to repository module

Weaknesses:
- Subsystems must agree (i.e., compromise) on a repository data model
- Schema evolution is difficult and expensive
- Distribution can be a problem
Hierarchical (Layered)

Components

- Group of subtasks which implement an abstraction at some layer in the hierarchy

Connectors

- Protocols that define how the layers interact
Hierarchical

Each layer provides services to the layer above it and acts as a client of the layer below. Each layer collects services at a particular level of abstraction.

A layer depends only on lower layers:

- Has no knowledge of higher layers

Example:

- Communication protocols
- Operating systems
Hierarchical: examples

THE operating system (Dijkstra)
The OSI Networking Model

- Each level supports communication at a level of abstraction
- Protocol specifies behavior at each level of abstraction
- Each layer deals with specific level of communication and uses services of the next lower level

Layers can be exchanged

- Example: Token Ring for Ethernet on Data Link Layer
The system you are designing

Performs data transformation services, such as byte swapping and encryption

Initializes a connection, including authentication

Reliably transmits messages

Transmits and routes data within the network

Sends and receives frames without error

Sends and receives bits over a channel
Hierarchical Style: Example (cont’d)

Use service of lower layer

Virtual connection
Hierarchical: discussion

**Strengths:**
- Increasing levels of abstraction as we move up through layers: partitions complex problems
- Maintenance: in theory, a layer only interacts with layer below (low coupling)
- Reuse: different implementations of the same level can be interchanged

**Weaknesses:**
- Performance: communicating down through layers and back up, hence by-passing may occur for efficiency reasons
Interpreters

Architecture is based on a virtual machine produced in software
Special kind of a layered architecture where a layer is implemented as a true language interpreter

Components

- “Program” being executed and its data
- Interpretation engine and its state

Example: Java Virtual Machine

- Java code translated to platform independent bytecode
- JVM is platform specific and interprets the bytecode
Client-Server

Components

- Subsystems are independent processes
- Servers provide specific services such as printing, etc.
- Clients use these services

Connectors

- Data streams, typically over a communication network
Client-Server example: databases

Front-end: User application (client)
- Customized user interface
- Front-end processing of data
- Initiation of server remote procedure calls
- Access to database server across the network

Back-end: Database access and manipulation (server)
- Centralized data management
- Data integrity and database consistency
- Database security
- Concurrent operations (multiple user access)
- Centralized processing (for example archiving)
Client-Server variants

Thick (or “fat”) client
- Does as much processing as possible
- Passes only data required for communications and archival storage to the server
- Advantages: less network bandwidth, fewer server requirements

Thin client
- Has little or no application logic
- Depends primarily on the server for processing activities
- Advantages: lower IT admin costs, easier to secure, lower hardware costs.
Client-Server: discussion

Strengths:
- Makes effective use of networked systems
- May allow for cheaper hardware
- Easy to add new servers or upgrade existing servers
- Availability (redundancy) may be straightforward

Weaknesses:
- Data interchange can be hampered by different data layouts
- Communication may be expensive
- Data integrity functionality must be implemented for each server
- Single point of failure
Peer-to-Peer

Similar to client-server style, but each component is both client and server

Pure peer-to-peer style
- No central server, no central router

Hybrid peer-to-peer style
- Central server keeps information on peers and responds to requests for that information

Examples
- File sharing applications, e.g., Napster, Gnutella, Kazaa
- Communication and collaboration, e.g., Skype
Peer-to-Peer: discussion

Strengths:
- Efficiency: all clients provide resources
- Scalability: system capacity grows with number of clients
- Robustness
  - Data is replicated over peers
  - No single point of failure (in pure peer-to-peer style)

Weaknesses:
- Architectural complexity
- Resources are distributed and not always available
- More demanding of peers (compared to client-server)
- New technology not fully understood
Conclusion: assessing architectures

*General style can be discussed ahead of time*

Know pros and cons

*Architectural styles ➔ Patterns ➔ Components*