Patterns of Software Architecture

Software Engineering
Alessio Gambi - Saarland University

Based on the work of Cesare Pautasso, Christoph Dorn, and their students
A software system’s architecture is the set of principal design decisions made about the system.

N. Taylor et al.

Abstraction
Communication
Visualization and Representation
Quality Attributes
Every system has a software architecture

What designers want

Prescriptive Architecture

Descriptive Architecture

System Artifacts

Realization

Recovery

Intent
Modeling

• Problem
  Domain model

• Environment
  System Context
  Stakeholders

• System-to-be
  Boundary/Internal Model
  Quality attributes
  Development

• Components
  Computation
  State

• Connectors
  Interaction
  C. C. C. F.

• Views & Viewpoints
  Kruchten 4+1
Design

- Architectural Styles
- Architectural Patterns
- Building Blocks
  - Software Connectors
Architectural Styles

Named collections of architectural decisions that are applicable in a development context. They constrain architectural design decisions, are specific to the system within that context, and elicit beneficial qualities in each resulting system.
Monolithic

- Lack of structure
- No Constraints
- Poor Maintainability
- Possibly Good Performance
Layered

- Communications 1 layer up/down
- Information hiding, no circular deps
- Possibly bad performance
- Good evolvability

Network protocol stacks · Web applications · Virtual Machines
Component Based

• Encapsulation
• Information hiding
• Components compatibility problem
• Good reuse, independent development

CORBA · Enterprise JavaBean · OSGi
Service Oriented

- Components might be outside control
- Standard connectors, precise interfaces
- Interface compatibility problem
- Loose coupling, reuse

Web Services (WS-*) · Cloud Computing
Plugin

- Explicit extension points
- Static/Dynamic composition
- Low security (3rd party code)
- Extensibility and customizability

Eclipse · Photoshop · Browsers’ extensions
Pipe & Filter

• Clean separation: filter process, pipe transport
• Heterogeneity and distribution
• Only batch processing, serializable data
• Composability, Reuse

UNIX shell · Compiler · Graphics Rendering
Black Board

- Collective problem solving via shared data
- Asynchronous components interactions
- Requires common data format
- Loose coupling, implicit data flow

*Database · Tuple space · Expert systems (AI)*
Event Driven

• Produce/React to events
• Asynchronous signals/messages
• Difficult guarantee performance
• Loose coupling, scalable

Sensor Monitoring · Complex Event Processing
Publish/Subscribe

- Event driven + opposite roles
- Subscription to queues or topics
- Limited scalability
- Loose coupling

Twitter · RSS Feeds · Email
Client/Server

- Many clients, active, close to users
- One server, passive, close to data
- Single point of failure, scalability
- Security, scalability

Web Browser/server · Databases · File Servers · Git/SVN
Peer to Peer

- Both server and client at the same time
- Dynamic join/leave
- Difficult administration, data recovery
- Scalability, dependability/robustness

File Sharing · Skype (mixed style) · Distributed Hash Tables
Data Centric

- Persistence layer
- Black board like
- Single point of failure
- (Eventual) Consistency (BASE/ACID)

Relational DB · Key-Value Stores
Rule Based

- Rules dynamically triggered
- Layered
- Possibly hard to understand and maintain
- Evolvability

Business Rule Engines · Expert Systems · Prolog
Mobile Code

- Code migrates (weak)
- Code+execution state migrate (strong)
- Security
- Fault tolerance, performance

JavaScript · Flash · Java Applets · Mobile Agents · Viruses
REST

- Hybrid style
- Stateless interactions/Stateful resources
- Loose coupling, scalability, interoperability

World Wide Web • RESTFul Web APIs
An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.

Layered - Component - Events - Composition
Layered Patterns

• State-Logic-Display
  *separate elements with different rate of change*

• Model-View-Controller
  *support many interaction and display modes for the same content*

• Presenter-View
  *keep a consistent look and feel across a complex UI*
cluster elements that change at the same rate
Model-View-Controller

separate content (model) from presentation (output) and interaction (input)
Presenter-View

extract the content from the model to be presented from the rendering into screens/web pages
Component Patterns

- Interoperability
  
  *enable communication between different platforms*

- Directory
  
  *facilitate location transparency (direct control)*

- Dependency Injection
  
  *facilitate location transparency (inversion of control)*
Interoperability

map to a standardized intermediate representation and communication style
Directory

use a directory service to find service endpoints based on abstract descriptions
Dependency Injection

use a container which updates components with bindings to their dependencies
Notification Patterns

• Event Monitor
  inform clients about events happening at the service

• Observer
  promptly inform clients about state changes of a service

• Publish/Subscribe
  decouple clients from services generating events

• Messaging Bridge
  connect multiple messaging systems

• Half Synch/Half Async
  interconnect synchronous and asynchronous components
Event Monitor

_poll and compare state snapshots_
Observer

detect changes and generate events at the service
Publish/Subscribe

factor out event propagation and subscription management into a separate service
Messaging Bridge

link multiple messaging systems to make messages exchanged on one also available on the others
Half-Sync/Half-Async

Add a layer hiding asynchronous interactions behind a synchronous interface
Composition Patterns

• Scatter/Gather
  send the same message to multiple recipients which will/may reply

• Canary Call
  avoid crashing all recipients of a poisoned request

• Master/Slave
  speed up the execution of long running computations

• Load Balancing
  speed up and scale up the execution of requests of many clients

• Orchestration
  improve the reuse of existing applications
Scatter/Gather

combine the notification of the request with aggregation of replies
Canary Call

use an heuristic to evaluate the request
Master/Slave

split a large job into smaller independent partitions which can be processed in parallel
Load Balancing

deploy many replicated instances of the server on multiple machines
Composition/Orchestration

build systems out of the composition of existing ones
Software Connectors

generic building blocks

Software connectors are first-class entities, have identity, and describe all system interactions. 
Software connectors are application independent and orthogonal to software components.
Remote Procedure Call

- Call

Often used within the client/server architectural style and event-oriented systems as call-backs
Stream

- Send
- Receive

*Fits the pipe & filter architectural style*
Shared Database

- Create
- Read
- Update
- Delete
Disruptor

- Next
- Publish
- WaitFor
- Get
Message Bus

- Publish
- Subscribe
- Notify

Fits the Service Oriented style
File Transfer

- Write
- Copy
- Watch
- Read
Linkage

- Load
- Unload
- Call
- Read/Write
Tuple Space

- In
- Out
- Rd

Fits the Blackboard style and the Master/Worker pattern
Web

- Get
- Put
- Post
- Delete

*Fits the REST architectural style*
Case Study
MediaWiki

• General purpose PHP-based system for Wikis

• The core of WikiMedia project (Wikipedia)

• Long-living project (~14 years)

• In September 2014 all Wikimedia projects served ~23.2 billions of pages
Main Scenarios

• A user requests an article during normal operation and gets the rendered article HTML page.

• An editor saves an edited article during normal operation and the article is saved.
Performance Tactics

- Control Resource Demand
  - Increase the resource efficiency (caching)
  - Prioritize events (deferred article updates)
  - Reduce overhead (precompile PHP and HTML)

- Manage Resources
  - Introduce concurrency (Distributed database)
  - Schedule resources (Load balancer)
  - Multiple copies of data and computations
Caching + Load Balancing
Caching + Load Balancing

Diagram:

- Squid (Caching) connected to Apache.
Caching + Load Balancing

LoadBalancer (Squid) — Squid — LoadBalancer — Apache
Caching + Load Balancing

LoadBalancer (Squid) → Squid → LoadBalancer → Apache

LoadBalancer → Squid → LoadBalancer → Apache
Distribution + Replication
Distribution + Replication
Distribution + Replication

Load Balancer (Master) — Partition Logic (Sharding, Replication)

DB Master (Shard) — DB Slave

Writes — Reads
Security/Availability Tactics

• Prevent Attacks
  • Challenge Tokens (CSRF)
  • Validation (User) and Sanitization (SQL Injection, XSS)

• Resist Attacks
  • Maintain multiple copies of computations.
  • Maintain multiple copies of data

• Recover from Attacks
  • DB Versioning (Recovery from data loss)
Backend

Parser

ArticleEdit

Reads

Writes

ArticleView

Submit Logic

Sanitizer Pipeline

Static Resources

UI Page

Loader

Cache

Skinning

Localization

Cache

Cache

Cache

Cache

Sanitizer Pipeline

Pipeline

Parser

Cache

Reads

User Access

Backend
Extensibility

UI Page

Submit Logic

ArticleEdit

Sanitizer

Static Resources

Loader

Cache

ArticleView

Cache

Parser

Cache

Skinning

Localization

Cache

Backend

Reads

Writes
Extensibility

UI Page

Submit Logic

Loader

Static Resources

Sanitizer

ArticleEdit

Notify

Notify

Parser

Notify

Notify

Reads

Hook Engine

Register Callback

Cache

ArticleView

Skinning

Localization

Cache

Cache

Cache

External Module

Backend

Writes
Configurability/Customizability

- UI Page
  - Submit Logic
  - Static Resources
  - Sanitizer
  - ArticleEdit
    - Writes
  - Parser
    - Reads
  - ArticleView
  - Skinning
  - Localization
  - Loader
  - Cache
  - Cache
  - Cache

- Reads
- Writes
Configurability/Customizability

- Global Variables and Configurations
- Submit Logic
- Static Resources
- ArticleEdit: Writes
- Sanitizer
- ArticleView
- Skinning
- Localization
- Parser: Reads
- Cache
- UI Page
Architectural Styles

“Looking at the source code, it becomes evident that MediaWiki’s development process did not focus on strictly following textbook architectural styles and patterns.”

• Layered
  - FrontEnd/Network, Application, Backend/Database
  - Multi-level caching

• Blackboard
  - Global variables
Architectural Patterns

- Presenter view (skinning)
- Publish/Subscribe (hooks)
- Master/Slave (database)
- Load balancer (network, database)
- Event monitor (cache invalidation)
Cheat Sheets
Architectural Styles

Layered
Client/Server
Data-Centric
Virtual Machine
Rule Based
Plugin
Peer to Peer
REST
Rails
Pipe and Filter
Event-Driven
Publish/Subscribe
Service Oriented
Component Based
Mobile Code
Blackboard
Layered Patterns

- State-Logic-Display
  separate elements with different rate of change
- Model-View-Controller
  support many interaction and display modes for the same content
- Presenter-View
  keep a consistent look and feel across a complex UI

Notification Patterns

- Event Monitor
  inform clients about events happening at the service
- Observer
  promptly inform clients about state changes of a service
- Publish/Subscribe
  decouple clients from services generating events
- Messaging Bridge
  connect multiple messaging systems
- Half Synch/Half Async
  interconnect synchronous and asynchronous components

Component Patterns

- Interoperability
  enable communication between different platforms
- Directory
  facilitate location transparency
- Dependency Injection
  facilitate location transparency

Composition Patterns

- Scatter/Gather
  send the same message to multiple recipients which will/may reply
- Canary Call
  avoid crashing all recipients of a poisoned request
- Master/Slave
  speed up the execution of long running computations
- Load Balancing
  speed up and scale up the execution of requests of many clients
- Orchestration
  improve the reuse of existing applications
Performance Tactics

Control Resource Demand
- Manage sampling rate
- Limit event response
- Prioritize events
- Reduce overhead
- Bound execution times
- Increase resource efficiency

Manage Resources
- Increase resources
- Introduce concurrency
- Maintain multiple copies of computations
- Maintain multiple copies of data
- Bound queue sizes
- Schedule resources

Events arrive --> Response generated within time constraints
Modifiability Tactics

- Reduce Size of a Module
  - Split Module
- Increase Cohesion
  - Increase Semantic Coherence
- Reduce Coupling
  - Encapsulate
    - Use an Intermediary
    - Restrict Dependencies
  - Refactor
    - Abstract Common Services
- Defer Binding

Change Requests → Changes Made and Deployed
Testability Tactics

- Control and Observe System State
  - Specialized Interfaces
  - Record/Playback
  - Localize State Storage
  - Abstract Data Sources
  - Sandbox
  - Executable Assertions

- Limit Complexity
  - Limit Structural Complexity
  - Limit Non-determinism

Tests Executed

Faults Detected