Software Architecture

Software Engineering - 2017
Alessio Gambi - Saarland University

These slides are based the slides from Cesare Pautasso and Christoph Dorn, and updated from various sources.
Architecture

What

How

Architecture
Design in the Large

- Objects and methods
- Modules and components
- Large and complex systems
- Systems of systems
Design in the Large

- Objects and methods
- Modules and components
- Large and complex systems
- Systems of systems
- Size of the team
- Lifetime of the project
- Cost of development
Building software as we build buildings?

- Software is complex, so are buildings (blueprint)
- Architecture implies a systematic process for design and implementation
- Architects put together pieces and materials, they usually do not invent new materials
It’s just an analogy!

- We know a lot about buildings (2000+ years), much less about software
- Software systems do not obey to physical laws
- Software deployment has no counterpart in building architecture
Software Architecture

A software system’s architecture is the set of principal design decisions made about the system.
Software Architecture

A software system’s architecture is the set of principal design decisions made about the system.

Where do the pillars go? \textit{VS} Where do the chairs go?

N. Taylor et al.
Abstraction

Manage complexity in the design
Communication

Document, remember and share design decisions among the team
Visualization

Load Balancer (assigns a web server)

Web Server (PHP assembles data)

Memcache (fast, simple)

Database (slow, persistent)

Depict and highlight important aspects
Representation

Characterize components and behaviors
Quality Analysis

Understand, predict, and control
When SW Architecture Start?

Since the beginning of design!
When SW Architecture Stop?
When SW Architecture Stop?

Never!
Architecture is NOT a development phase
Architecture is NOT only “high-level” design
Every System an Architecture has
Every System an Architecture has
Every System an Architecture has
Architectural Evolution

Decisions are added and changed by multiple actors… sometimes without knowing it
Architectural Degradation
Architectural Degradation

Ideal $P=D$
Architectural Degradation

**Ideal** $P = D$

**Drift** $P \neq D$ and $D$ does not violate $P$
Architectural Degradation

**Ideal** $P = D$

**Drift** $P \neq D$ and $D$ does not violate $P$

**Erosion** $P \neq D$ and $D$ violates $P$
Summary

• Blueprint for construction and evolution
  abstraction • principal design decisions

• Not only about design
  communicate • visualize • represent • assess

• Every system has (an evolving) one
  descriptive • prescriptive • drift • erosion

• Not a phase, not only “high-level” design
Design
How to Design

Even the best architects copy solutions that have proven themselves in practice, adapt them to the current context, improve upon their weaknesses, and then assemble them in novel ways.
How to Design

Even the best architects copy solutions that have proven themselves in practice, adapt them to the current context, improve upon their weaknesses, and then assemble them in novel ways.
How to Design

Even the best architects copy solutions that have proven themselves in practice, adapt them to the current context, improve upon their weaknesses, and then assemble them in novel ways.
Architectural Hoisting

Design the architecture with the intent to guarantee a certain quality of the system.
Architectural Hoisting

Design the architecture with the intent to guarantee a certain quality of the system.

<table>
<thead>
<tr>
<th>Security</th>
<th>place sensitive data behind the firewall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>make critical components stateless</td>
</tr>
<tr>
<td>Persistence</td>
<td>use a database</td>
</tr>
<tr>
<td>Extensibility</td>
<td>use a plug-in framework</td>
</tr>
</tbody>
</table>

George Fairbanks - http://georgefairbanks.com/blog/architectural-hoisting-original/
What makes a “good” SW Architecture?

- No such things like perfect design and inherently good/bad architecture
- Fit to some purpose, and context-dependent
What makes a “good” SW Architecture?

• No such things like perfect design and inherently good/bad architecture

• Fit to some purpose, and context-dependent

• Principles, guidelines and the use of collective experience (method)
Design Principles

Architectural Patterns

Architectural Styles
Design Principles

• Abstraction
• Encapsulation - Separation of Concerns
• Modularization
• KISS (Keep it simple, stupid)
• DRY (Don’t repeat yourself)
Architectural Patterns

Set of architectural design decisions that are applicable to a recurring design problem, and are parameterized to account for the development contexts in which that problem appears.
Architectural Patterns

Set of architectural design decisions that are applicable to a recurring design problem, and are parameterized to account for the development contexts in which that problem appears.

Layered
Notification

Component
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*
State-Logic-Display

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
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*

1. poll(frequency)
2. compare(A,B)
3. notify
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*
Patterns, Patterns, Patterns

Architectural Patterns *not* Design Patterns

Architectural  Design
Patterns, Patterns, Patterns

*Architectural Patterns* not *Design Patterns*

Express fundamental structural organizations

Specify relationships among (sub-)systems

Capture roles in solutions that occur repeatedly

Define the relationships among roles

*Architectural Design*
Architectural Styles

Named collections of architectural decisions and constrains for a specific development context that elicit beneficial qualities in each resulting system
Why Styles?

A common vocabulary for the design elements improve communication by shared understanding

A predefined configuration and composition rules known benefits and limitations ensure quality attributes if constraints are followed

Style-specific analyses and visualizations
Styles and Patterns

General constraints
- Architecture with superior properties
- Styles must be refined and adapted
- One style is dominant

Fine-grained constraints
- Specific to recurrent problems
- The same pattern can be used many times
- Many patterns are combined
Many (and Many More)

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
Monolithic

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

Mainframe COBOL programs · powerpoint · many games
Layered

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

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

Sensor Monitoring · Complex Event Processing
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
Publish/Subscribe

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

Twitter · RSS Feeds · Email
Publish/Subscribe

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

• A great architecture likely combines aspects of several other architectures

• Do no limit to just one pattern, but avoid the use of unnecessary patterns

• Different styles lead to architectures with different qualities, and so might do the same style

• Never stop at the choice of patterns and styles: further refinement is always needed
Modeling
Why modeling?

• Record decisions
• Communicate decisions
• Evaluate decisions
• Generate artifacts
WHAT DO WE MODEL?!
The problem (Domain model)
WHAT DO WE MODEL?!
The system-to-be (Design Model)

*Static and dynamic architecture*

The problem (Domain model)

The environment

*System context*

*Stakeholders*
The system-to-be (Design Model)

Static and dynamic architecture

The problem (Domain model)

The environment
System context
Stakeholders

Quality attributes and non-functional properties
The system-to-be (Design Model)

*Static and dynamic architecture*

The design process

The problem (Domain model)

Quality attributes and non-functional properties

The environment

*System context*

*Stakeholders*
Design Model

Boundary Model

System Context
Interfaces/API
Quality Attributes

Externally visible behavior
Design Model

Boundary Model
- System Context
- Interfaces/API
- Quality Attributes

Internal Model
- Software Components
- Software Connectors
- Component assembly

Externally visible behavior

Internal behavior
Software Components

Reusable unit of composition
Can be composed into larger systems

Locus of computation

Processing
State

State in a system
Software Components

Reusable unit of composition
Can be composed into larger systems

Locus of computation
Processing
State
State in a system

Application-specific
Media Player
Math Library

Infrastructure
Web Server
Database
Composition vs Distribution
Component Roles

- Real-time
- User Interface
- Active
- Passive
- Stateful
- Stateless
<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
</table>
| Components    | Encapsulate state and functionality  
|               | Coarse-grained  
|               | Black box architecture elements  
|               | Structure of architecture                                                   |
| Objects       | Encapsulate state and functionality  
|               | Fine-grained  
|               | Can “move” across components  
|               | Identifiable unit of instantiation                                           |
| Modules       | Rarely exist at run time  
|               | May require other modules to compile  
|               | Package the code                                                             |
Component Interfaces
Provided Interfaces

• Specify and document the externally visible features (or public API) offered by the component

  - *Data types and model*
  - *Operations*
  - *Properties*
  - *Events and call-backs*
Required Interface

- Specify the conditions upon which a component can be (re)used
  - *The platform is compatible*
  - *The environment is setup correctly*
Compatible Interfaces

Adapter

Wrapper
Software Connectors

Model static and dynamic aspects of the interaction between component interfaces
Connector Roles

• Communication
  deliver data and transfer of control

• Coordination
  separate control from computation

• Conversion
  enable interaction of mismatched components

• Facilitation/Mediation
  govern access to shared information
not always directly visible in the code
mostly application-independent
Connectors are abstractions

When to hide components inside a connector?
Remote Procedure Call

- Call
Remote Procedure Call

- Call
Message Bus

- Publish
- Subscribe
- Notify
Web

- Get
- Put
- Post
- Delete
Views and Viewpoints
A subset of related architectural design decisions

The common concerns shared by a view

A subset of related architectural design decisions

Viewpoint

View
Consistency

Views are not always orthogonal and might become inconsistent if design decisions are not compatible (erosion).
4+1

Logical View

Development View

Use Case Scenarios

Process View

Physical View

Philippe Kruchten
Use Case Scenarios

• Unify and link the elements of the other 4 views

• Help to ensure the architectural model addresses all the requirements

• Each scenario can be illustrated using the other 4 views
MusicApp Example

Use Case Scenarios

* Browse for new songs
* Buy song
* Download the purchased song on the phone
* Play the song
Logical View

• Decompose the system structure into software components and connectors

• Map functionalities (use cases) onto the components
Process View

- Model the dynamic aspects of the architecture and the behavior of its parts
- Describe how components/processes communicate
Use Cases: Browse, Pay and Play For Songs
Development View

- Static organization of the software code artifacts
- Map elements in the logical view and the code artifacts
User Interface

Language: Java ME
Repository: SVN

Buy a licence
Music Player

Customer Database
MySQL

Get an SLA with a provider
Payment Service

Songs Repository
MySQL+FileSystem
Physical View

- Hardware environment where the software will be deployed
- Map logical and physical entities
Architecture

What

How

Architecture
References and Readings

• Textbooks

• References
  • Mary Shaw and David Garlan, Software Architecture: Pespectives on an Emerging Discipline, Prentice-Hall, 1996
  • Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal Pattern Oriented Software Architecture: A System of Patterns, Wiley, 1996
  • Martin Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2002
  • Luke Hohmann, Beyond Software Architecture: Creating and Sustaining Winning Solutions, Addison-Wesley, 2003
  • Ian Gorton, Essential Software Architecture, Springer 2006