Software Architecture


Dr. Sebastian Proksch

University of Zurich, Switzerland
“Architecting, the planning and building of structures, is as old as human societies – and as modern as the exploration of the solar system.”

—Rechtin, 1991
Motivation

If the size and complexity of a software system increase, the global structure of the system becomes more important than the selection of specific algorithms and data structures.
“the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution”

—IEEE Standard 1471-2011, Definition of “Software Architecture”
So what does an Software Architect actually do?

Source: http://dilbert.com
• Hierarchically somewhere between Dev and Team Lead

• Often writes some code, but usually not his core job

• Needs to focus on the big picture while understanding detailed technical issues

• Responsible for strategic decisions, e.g., which technology or architectural style to use

• Responsible for decomposing a system into manageable parts

• Responsible for integration of said parts after implementation

• Responsible for interface definitions
About Software Architects

See also:


(mandatory reading for this class)
Software Architecture in the Waterfall Model

- Requirements
- Design
- Implementation
- Verification
- Maintenance

Product requirements document
Software architecture
Software

Source: https://en.wikipedia.org/wiki/Big_Design_Up_Front

“BDUF”
Software Architecture in an Agile Team

Architecture develops with the system

Source: http://scaledagileframework.com/the-principles-of-agile-architecture/
Example Architecture

Source: http://scaledagileframework.com/the-principles-of-agile-architecture/
Basic Elements of a Software Architecture

Components
Connectors
Constraints
Rationale

© Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy
What is a Component?

Unit of *decomposition* of a system

Can be a software package, Web service or module that encapsulates a set of related functions or data.
What is a Connector?

Architectural element that models interactions among components and the rules that govern those interactions.

A connector provides interaction duct(s) and transfer of control/data.
Components vs. Connectors

Components provide application-specific functionality

Connectors provide application-independent interaction mechanisms

© Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy
Connectors:
Interaction Types

Simple interactions
- Procedure calls
- Shared variable access

Complex interactions
- Client-server protocols
- Database access protocols
- Asynchronous event multi-cast

© Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy
Roles of Connectors

Protocol specification

- Types of interfaces
- Assurances about interaction properties
- Rules about interaction ordering
- Interaction commitment (e.g., performance)

Nature of Roles

- Communication
- Coordination
- Conversion
- Facilitation
Connectors as Converters

Connectors enable the interaction of independently developed components that are mismatched in:

- Type
- Number
- Frequency
- Order
Facilitators enable interaction of components by mediation or streamlining. They can:

- govern access to shared information
- ensure proper performance profiles (e.g., load balancing)
- provide synchronization mechanisms (e.g., monitors, guards for critical sections)
Components must be constrained to provide that:

- the requirements are met
- the required functionality is achieved
- no functionality is duplicated
- the required performance is achieved
- modularity is realized
The rationale is the “why?”

For multi-version software, its design rationale must be documented:

- Decomposition into components
- Connections between components
- Constraints upon components and connections
The 4+1 View Model of Software Architecture
The 4+1 View Model of Software Architecture

Source: http://www.wikipedia.org
Logical (Conceptual) View

- Functional requirements
- Orientation on problem domain
- Communication with experts
- Independent of implementation decisions
Logical View: Example

Air traffic management system
Development (Module) View

Organization of modules

- Subsystems
- Coherent parts in the development
- Allocation of effort (development, maintenance)

Organization in hierarchical layers

- OSI communication protocols
Development View: Example

Air traffic management system

- Human computer interface
  - External systems
- ATC functional areas: flight management, sector management, etc.
- Aeronautical classes
  - ATC classes
- Support mechanisms:
  - communication, time, storage, resource management, etc.
- Common utilities

Domains:
- Domain-specific
- Domain-independent
- Customer-specific
Physical View

Mapping of software onto existing/available hardware

Non-functional requirements

• Performance
• System availability
• Fault-tolerance
• Scalability
Note that the term *Physical View* comes from a time before virtualization.

In the cloud, your *Physical View* will map your components to virtual resources.
Process View

Dynamic aspects of the run-time processes

- Process creation
- Synchronization
- Concurrency

“What happens at system runtime”
Process View: Example

Copyright 2005 Scott W. Ambler
Instances of use cases show that the elements of the four views work together seamlessly

Example:

User stories that run through all 4 views

User stories that show the importance and rationale of all elements of all 4 views
An **Architectural Style** is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.

An **Architectural Pattern** is a named collection of architectural design decisions that are applicable to a recurring design problem parameterized to account for different software development contexts in which that problem appears.
Both provide a common vocabulary to talk about different aspects of software architecture.

A style is a common architectural design.

A pattern is a way of solving a common architectural problem.
An architectural style defines a **family** of software systems and their structural organization.

It defines **components**, **connectors**, and **constraints** for their usage in concrete software systems.

© Perry and Wolf, 1992
(Some) Common Architectural Styles

(Many) books are written about architectural styles and patterns.

Some of the most well-known ones:

Buschmann et al.:
Pattern-Oriented Software Architecture, Vol. 1, 2, 4

Fowler:
Patterns of Enterprise Application Architecture
(Some) Common Architectural Styles

Program structure
- Layers
- Client / Server
- Peer-to-Peer
- MVC

Data Flow
- Pipes and Filters
- Event-Driven Architecture
- Blackboard (Factory / Worker)
Architecture Styles

Layers
Layered Architecture

Hierarchically organized system

Layers are components

Layer interfaces and protocols are connectors
<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
</table>
| 7     | Application Layer  
✓ Message format, Human-Machine Interfaces |
| 6     | Presentation Layer  
✓ Coding into 1s and 0s; encryption, compression |
| 5     | Session Layer  
✓ Authentication, permissions, session restoration |
| 4     | Transport Layer  
✓ End-to-end error control |
| 3     | Network Layer  
✓ Network addressing; routing or switching |
| 2     | Data Link Layer  
✓ Error detection, flow control on physical link |
| 1     | Physical Layer  
✓ Bit stream: physical medium, method of representing bits |
Layers: Advantages

- Divide and conquer
- Locality of changes
- Reusability
Layers: Disadvantages

Layers don’t encapsulate everything well
Layers *may* harm performance
Additional layers may increase development effort
Architecture Styles

Client/Server
Servers are actively all the time and passively wait for requests.

Clients are active sporadically and trigger servers.

Servers are (usually) few and fat, while clients are numerous and thin.
Client / Server Example

Client / Server: Advantages

Simple, *time-tested* style

Particularly suited for distributed applications

Updates to the server are easy to roll out (allows e.g., for *partial deployment*)

Clients can often be thin and run on cheap hardware
Client / Server: Disadvantages

- Servers overload easily
- Servers are a single point of failure
Architecture Styles
MVC
Model-View-Controller (MVC)

Models hold data

Views present data

Controllers implement business logics
MVC - Example

Ruby on Rails (Ruby-based Web framework)

Source: https://leanpub.com/rails3dot0-astudentmanual/read
MVC: Advantages

- Separation of concerns
- Supports TDD (Test-Driven Development) well
- Maps well to REST (Representational State Transfer) principles
MVC: Disadvantages

Usually, every change (e.g., adding new feature) touches models, views, and controllers.

It is not usually possible to assign related model, view and controller to different teams.
Architecture Styles

Peer-to-Peer
Peer-to-Peer Components are both server and client at the same time

Everybody is (in principle) communicating with everybody

Two styles:

“pure” P2P

“hybrid” P2P with central servers or super-peers
Peer-to-Peer Example

Figure 1: Skype Network

Source: http://kb.fortinet.com/kb/viewContent.do?externalId=FD30776
Peer-to-Peer
Advantages

Each new participant also adds new resources —> no natural bounds to system scale (scalability!)

Usually few or no single point of failure

Usually no single point of control
Peer-to-Peer: Disadvantages

In pure P2P, **message flooding** is a problem

“**Soft**” state between peers

Protocols tend to get complicated

Software updates often not easy
Architecture Styles
Pipes and Filters
Filters are the components that read an input data stream and transform it into an output data stream.

Pipes are the connectors that provide the output of a filter as input to another filter.
Pipes and Filters: Example

```
find . -type f -print | sort | uniq
```
Pipes and Filters: Advantages

Simple; no complex component interactions

Filters as black-box; substitutable

High maintainability and reusability of individual components
Pipes and Filters: Disadvantages

Whether separation into processing steps is feasible strongly depends on application domain and problem.

Filters require a common data format.

Redundancy in parsing/unparsing.

Process overhead.
Architecture Styles

Event-Driven Architecture
Event-driven Architecture (EDA)

Components are event emitters (publishers, agents) or consumers (subscribers, sinks)

An event dispatcher distributes events to subscribers

--> fundamentally asynchronous
Java Swing
(or more generally - most GUI frameworks)
EDA: Advantages

Components are loosely coupled
High (runtime) extensibility and reusability; components can be easily exchanged
Especially apt to unpredictable and asynchronous environments
Often “feels” fast and responsive
EDA: Disadvantages

No guarantees regarding execution or order of event processing

Data exchange other than with events problematic

Component behavior is tightly coupled with execution environment

Difficult to “grok”
Architecture Styles
Blackboard
Components:

- Central data management
- Independent components for computation (knowledge sources)

Activation of computation:

- Database trigger
- Actual state
Both store data and represent the central point for communication and synchronization of workers.

*Database* can only store data, the workers are “independent”.

*Blackboard* can serve as a factory for tasks and activate workers.
Blackboard: Advantages

High changeability and maintainability

Reusable knowledge sources

**Factory/Worker:** support for fault-tolerance, robustness and redundancy because of loose coupling of workers
Blackboard:
Disadvantages

Hard to test
Difficult to establish good control strategy
Low efficiency
Absence of Architecture
Failure to Maintain Architectural Discipline?

“Big Ball of Mud”

https://de.wikipedia.org/wiki/Heiliger_Pillendreher
Common reasons for Technical Debt:

- “pragmatism” (we used whatever made sense at the time)
- time pressure
- “agile” / “lean” (minimal product)
- architecture follows organizational structure
Conclusions

For non-trivial software systems, a proper architecture needs to be designed (either upfront or in parallel to the code)

Architectures consist of components, connectors and constraints (+ rationale)

The 4+1 Model provides means to derive and document such an architecture

Architectural styles provide software engineers a common language for describing and reasoning about software architecture