Software Architecture
Styles and Paradigms

Frank Buschmann, Kevlin Henney
Software Architecture Styles and Paradigms

Agenda

- Motivation
- Problem Frames
- Domain Properties
- Systems Integration
- Conclusion
Debates on software architecture (1)

Time and again we have debates on what is the best
- Software architecture style and paradigm
- Programming platform
- Technology set
- Programming language
- Development approach

in software engineering.

All of us are right, and all of us a wrong. It simply depends … but on what???

Debates on software architecture (2)

All of us are right, and all of us a wrong. It depends … but on what?
- Obviously not on specific requirements
- Or on personal flavors

Yet we cannot choose arbitrarily!

So …
- What are the factors that impact the choice of architecture?
- Is there any guidance that help us choosing the right architecture in the presence of these factors?
  - To avoid guessing or religious debates
Factors that influence software architectures

Software architectures are influenced by many factors:

- **Domain Model**
  - Entities
  - Workflows
  - Variations

- **Requirements**
  - Functions / features
  - Non-functional qualities

- **Business**
  - Business drivers: innovation, cost
  - Solution / product / product family
  - Sales channels
  - Sales margins

- **Domain Properties**
  - Problem frames
  - Logical / physical processes
  - Deterministic / non-deterministic behavior
  - Distributed / collocated entities
  - Degree of dependability
  - Parallelism
  - Rate of change

- **Execution environment**
  - Single computer / network / cluster / cloud
  - Embedded / PC / Mainframe
  - Single / Multi-core

- **Organization**
  - Size
  - Development locations
  - Maturity level

**Focus of this talk**

Key focus of this talk are architecture styles and paradigms driven by fundamental of classes of application domain and their properties:

- **Domain Model**
  - Entities
  - Workflows
  - Variations

- **Requirements**
  - Functions / features
  - Non-functional qualities

- **Business**
  - Business drivers: innovation, cost
  - Solution / product / product family
  - Sales channels
  - Sales margins

- **Domain Properties**
  - Problem frames
  - Logical / physical processes
  - Deterministic / non-deterministic behavior
  - Distributed / collocated entities
  - Degree of dependability
  - Parallelism
  - Rate of change

- **Execution environment**
  - Single computer / network / cluster / cloud
  - Embedded / PC / Mainframe
  - Single / Multi-core

- **Organization**
  - Size
  - Development locations
  - Maturity level
Imagine you are a software architect in an industry environment and responsible for software that supports the entire value chain.

- Digital Product Planning
- Digital Factory Planning
- Equipment Configuration
- Manufacturing Execution Systems (MES) for production planning and control
- Distributed Control Systems (DCS) and Supervisory Control and Data Acquisition (SCADA) to control the physical production processes
- Programmable Logic Controllers (PLC) for controlling field devices in a plant

What architecture do you choose for each system to ensure it fulfills its task well?
How do you integrate all systems end-to-end, without degrading their individual QoS?
Regardless of their concrete app domain, software can be categorized according to **types of domains** that share fundamental properties.

**Simple Workpiece Problem Frame:**
A tool is needed to allow a user to create and edit a certain class of computer-processible text, graphical object, or similar structures, so that they subsequently can be copied, printed, analyzed, or used in other ways.

- Digital Product Planning
- Digital Factory Planning
- Equipment Configuration
- ...
- UML Modeling
- Structured Drawing

**An observation (2)**

The problem description leads naturally to an **architecture style** that
- Separate the representation of the objects from the tools that operate on them
- Provide plug-in mechanisms for new tools that can operate on the materials, or new materials the tools can operate on

**Presentation**
- Framework for Common GUI Services and Controls
  - Editor 1 ...
  - Editor N

**Business Logic**
- Framework for Business Logic Plug-Ins
  - Business Logic 1 ...
  - Business Logic X

**Data Logic**
- Framework for Object Management and Persistence
  - Object Model Description
    - (XML, EDD)
    - Generated Object Model
    - Database Abstraction Layer
    - RDBMS
    - File Storage

Modeling tools and platforms, e.g., for
- Industrial Engineering
- Eclipse
Generally follow the
- Simple Workpiece Problem Frame
- Tools and Materials Architecture Metaphor
An observation (3)

It is essential to follow the chosen architecture style from base-line architecture all the way down to implementation

- What looks obvious from a birds-eye perspective is often neglected at detailed design and code level
- Breaking the chosen style degrades architecture quality and stability
- Fixing violations of a broken architecture style is expensive!

**Example: broken tools and materials metaphor:**
data objects in the data layer invoke services in the business layer, which manipulate other objects in the data layer.

- **Broken layering:** data objects are dependent on functional interfaces
- **Instability:** uncertainty about termination of call chains and reaching a stable data state
- **High refactoring costs** were necessary to fix the design

---

Problem Frames

Problem Frames are a systematic approach to choose the “right” architecture style for a software system, based on its “class of problem”

- Original work by Michael Jackson
- Five fundamental problem frames
  - Required behavior
  - Commanded behavior
  - Information display
  - Simple workpieces
  - Transformation
- Real-world systems often comprise multiple problems
  - Composite frames – combinations of several problem frames describe a specific problem
  - Multiple frames – different problem areas follow different problem frames
Required Behavior

Required Behavior Problem Frame
There is some part of the physical world whose behavior is to be controlled so that it satisfies certain conditions.

Examples: industrial control systems – SCADA and DCS

Shared repository architecture
- Architecture maintains an image of the physical process under control, represented by its domain objects and their state
- State in the image is modified by value changes from field devices, which triggers functionality; functionality operates on the image, which issues value changes to field devices
- Often high quality of service, real-time behavior

Commanded Behavior

Commanded Behavior Problem Frame
There is some part of the physical world whose behavior is to be controlled in accordance with commands issued by an operator.

Examples: industrial control systems – batch or recipe execution

Command Processor design
- Operator creates requests for specific functionalities or application behavior
- A command processor receives requests and executes them on the application
- The application executes the requested functionality
Information Display

Information Display Problem Frame
There is some part of the physical world about whose states and behavior certain information is continually needed.

Observer design
- Application maintains system state
- Views subscribe to the application to be notified about state changes
- On state changes in the application, the views are notified to update their own state
- Different flavors: push or pull

Examples: HMI functionality in industrial control systems

Simple Workpiece

Simple Workpiece Problem Frame
A tool is needed to allow a user to create and edit a certain class of computer-processible text, graphical object, or similar structures, so that they subsequently can be copied, printed, analyzed, or used in other ways.

Tools and materials architecture
- Separate the representation of the objects from the tools that operate on them
- Provide plug-in mechanisms for new tools that can operate on the materials, or new materials the tools can operate on

Examples: industrial engineering tools – factory and product planning, equipment configuration
Transformation

Transformation Problem Frame
There are some given computer-readable input files whose data must be transformed to give certain required output files. The output data must be in a particular format, and it must be derived from the input data according to certain rules.

Pipes and Filters architecture
- Filters process data and transform it from an input format into an output format
- Pipes transmit data between filters

Examples: medical imaging

Composite and Multi-Frames

Problems solved by real-world systems are likely combinations of multiple problem frames: their architectures thus follow combinations of different architecture styles

Industrial Control Systems
- Functionality follows the required and controlled behavior problem frames
- The need for visualizing the state of the physical process under control follows the information display problem frame

Baseline Architecture
- Model-View-Controller (Observer + Command Processor) for information display and commanded behavior
- Shared Repository for required behavior
Problem Frames in retrospect

Problem Frames are an important vehicle to get first hands on a software architecture. They

- Guide the selection of core architecture styles and patterns for a system’s base-line architecture
- Answer the “why-question” for core design decisions that relate to the nature of application domains
- Provide a framework for subsequent, more specific design decisions

Software Architecture Styles and Paradigms

Agenda

- Motivation
- Problem Frames
- Domain Properties
- Systems Integration
- Conclusion
Another observation (part 1)

Problem frames are an important guidance, but not sufficient for defining a sustainable software architecture. There are other factors beyond specific requirements that shape a system’s base-line architecture.

Required Behavior Problem Frame
There is some part of the physical world whose behavior is to be controlled so that it satisfies certain conditions.

Shared repository architecture
- Architecture maintains an image of the physical process under control, represented by its domain objects and their state.
- State in the image is modified by value changes from field devices, which triggers functionality; functionality operates on the image, which issues value changes to field devices.
- Often high quality of service, real-time behavior

Business Logic Framework
- Managed Object Framework (Data access and manipulation)
- Process Image
- Field Device and Protocol Gateway

Is throughput and latency the key performance driver for our functionality or is it executed in fixed, cyclic time intervals?

Domain objects and their realization paradigms

The fundamental properties of objects in a domain have a strong impact on fundamental architecture styles, decisions, and technology choices

<table>
<thead>
<tr>
<th>Physical objects (e.g., motors, machines, vessels, conveyor belts)</th>
<th>Logical objects (e.g., orders, accounts, invoices, spreadsheets, work plans)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irreversible behavior, objects must obey to physical laws</td>
<td>Irreversible but cancellable behavior</td>
</tr>
<tr>
<td>Deterministic, closed loop control, hard real-time behavior</td>
<td>Reversible behavior</td>
</tr>
<tr>
<td>Cyclic scan paradigm</td>
<td>Soft real-time behavior</td>
</tr>
<tr>
<td>Mirror paradigm</td>
<td>Transaction paradigm</td>
</tr>
<tr>
<td>Load-limiting paradigm</td>
<td>Load-limiting paradigm</td>
</tr>
<tr>
<td>Max. throughput, min. latency, near real-time behavior</td>
<td>Soft real-time behavior</td>
</tr>
</tbody>
</table>

Material adapted from Bernhard Gaissmaier and George Lo, Siemens AG
**Time-Triggered Architecture**

Objects with hard real-time, deterministic and cyclic behavior are best realized using a **Time-Triggered Architecture**

- Components get defined **fixed and cyclic** time budgets for service execution and communication
- Within the time for service execution, components must reach stable intermediate states and be able to exchange data and requests with other components
- Within the time for communication, components must finish their exchange of data and requests with other components
- Reliability is essential to ensure availability

**Examples**

- Industry Automation: Programmable Logic Controllers (PLC)
- Automotive: FlexRay Communication Protocol

**Stateful Components**

Objects mirroring entities and (near) real-time processes of the physical world are best realized using **Stateful Components**

- Process image is kept in-memory inside stateful components to ensure fastest turnaround cycles from field sensors to functionality and UI, and back again to field actors
- Functionality is designed as stateless components
- (R)DBMS used only for configuration and archiving purposes
- High availability and data integrity is achieved using active hot stand-by redundancy of service and UI components

**Examples**

- SCADA and DCS Systems (e.g., tunnel automation)
Stateless Components and SOA

Objects representing entities and (soft real-time) workflows of the logical world are best realized using **Stateless Components**

- All data is kept in a (R)DBMS: accounts, plans, work plans, orders, recipes …
- Functionality to manipulate data is designed as stateless, transaction-oriented components
- A SOA-ESB connects the services
- Availability is achieved using standard technology:
  - Application Server cluster for parallel redundancy of service and UI components
  - Database cluster for high data availability
  - Graceful degradation of functionality in case of system overload

**Examples**
- Industry: MES and ERP systems
- General: Business information systems

Another observation (part 2)

Martin Fowler’s first law of Distributed Object Design: ‘Don't distribute your objects.’ But the reality is that you often can’t avoid it (completely)

- Sometimes the partitioning of the domain is inherently distributed, e.g., the production process in a factory
- Sometimes a concrete application has distribution constraints, e.g., a SCADA or DCS system that spreads across multiple sites or plant areas
- Sometimes, operational qualities suggest the utilization of a network: e.g., performance, scalability, and availability

- But even in centralized deployments, distribution enters through the backdoor: multi-core hardware supports distribution (and parallelism!!) on a single chip
Distribution and concurrency (1)

The distribution and parallelism that is inherently in the domain should be mirrored in the software architecture:

- **Data and functional locality.** Data and functionality is maintained locally to the physical place where it is needed, e.g., for independent areas in a production plant.
  - Optimal utilization of computational power in a network
  - High performance due to locality and reduced use of network communication
  - Better availability through distributed responsibilities – partial failures not result in total failure

- **Independent tasks.** Tasks in the real-world domain that naturally execute in parallel or interleaved should also run in parallel or interleaved in the software, e.g., production of goods in parallel production lines.
  - Task-/user-oriented parallelism and concurrency
  - Support for multi-core hardware, clusters, and clouds
  - In many cases, minimal and largely domain-driven need for synchronization

- **Work split.** Data whose manipulation can be partitioned into several chunks or steps should be handled accordingly in the software, e.g., for image processing.
  - Supports parallelism according to the divide and conquer, map–reduce, task tree, or processing pipeline principles

Distribution and concurrency (2)

Well-known architecture and design patterns support the key principles for “natural” distribution and parallelism. For example:

- **Broker** architectures support data and functional locality.
- **Leader Followers, Half-Sync/Half-Async, Active Object, and Proactor** support the execution of parallel tasks.
- **Master-Slave** supports the divide and conquer principle.
- **Parallel Pipes and Filters** supports the processing pipeline principle.
- **Parallel Layers** supports the task tree principle.

160,000,000 transactions per day in medical financials.

OLTP System
Distribution and concurrency (3)

Programming platforms with “built-in” parallelism suggest using concurrency as an explicit modeling paradigm. In Erlang, for example:

- **Processes** are cheap and plentiful
  - When you need a process – create one!
  - Do not ration processes – create as many as you need!
  - Do not pool processes – reuse is a real pain!

- **Message passing** is cheap
  - Use processes to separate concerns
  - Middle-man processes for data transformation

- **Data copying** is cheap
  - Functional programming to avoid side effects and massive parallelism

Material adapted from Francesco Cesarini, Erlang Training and Consulting Ltd.

Another observation (part 3)

There is nothing permanent except change [Heraclitus, 535–475 BC]

- Business cases
- Requirements
- Design and implementation
- Programming platforms
- Execution environments
- Design technologies

Software architectures must deal with changes and manage them in an explicit and controlled fashion … … otherwise they are lost in architecture drift and design erosion

New!

A system for engineering automation equipment must be open for integrating new tools and new automation equipment
Layering for change and protection

Introduce loosely coupled layers to separate design centers that can **change at a different rate**, in particular:

- User interface and application functionality
- Application functionality and database
- Application and connectors to external systems
- Application and operating system

Layering according to rate of change protects a software architecture from rippling effects

Layering is also an important architecture measure for business protection. For example, open source software can be strictly separated from business-critical components

Stable design centers

Manage change and variation explicitly by creating stable design centers for each component of a software architecture:

- Freeze the component's invariant core logic
- Open the design center via extension hooks to address mandatory variation
- Objectify all aspects that are subject to variation to make them pluggable
- Shield clients from the internals of a design center via explicit, extensible interfaces

Stable design centers realize the "open / close" principle: they are open to extension, replacement, and reduction, but closed to uncontrolled modifications and changes
Domain properties in retrospect

Inherent properties of an application domain influence software architectures significantly and beyond their detailed requirements:

- Physical and logical properties of domain objects
- Distribution and parallelism in the real-world domain
- Change and evolution – rate of innovation

Sustainable software architectures portray these properties explicitly, to ensure the virtual world can "mimic" the real world.

Software Architecture Styles and Paradigms

Agenda

- Motivation
- Problem Frames
- Domain Properties
- Systems Integration
- Conclusion
End-to-end control flow across systems

Integrating systems following different paradigms must not impact correct and reliable behavior of each individual system!

A rule of thumb

- The more autonomous the systems work, and the more diverse their architecture styles are, the looser their coupling should or can be

- The more a stringent end-to-end control of activities and data exchanged across systems is required, the tighter the coupling must be

Separation of tasks is a good thing, on the other hand we have to tie the loose ends together again!

[Edsger Dijkstra]

Integration styles

Architectural styles for systems integration

- Enterprise Application Integration
- Service Oriented Architecture
- Service Component Architecture
- Event Driven Architecture
Enterprise Application Integration

Enterprise Application Integration (EAI) helps diverse applications in an enterprise, including partner systems, communicate with each other to achieve a business objective, irrespective of the platform and geographical location of the applications.

Two architecture patterns support Enterprise Application Integration:

- Hub and Spoke Architecture
- Bus Architecture

Service Oriented Architecture

Service Oriented Architecture (SoA) helps in leveraging existing applications and infrastructures to solve business problems without having to rebuild everything from scratch.

Service Oriented Architecture enables flexibility and helps to manage complexity:
- Well-defined service interfaces
- Standardized communication protocols
- Flexible recombination of services
- Platform independence of services
- Simple extensibility
Service Component Architecture

Service Component Architecture (SCA) is a simplified SoA programming model for:
- **Building** service components
- **Assembling** (existing) components into applications
- **Defining quality of service** aspects such as transactions, security, and reliable asynchronous invocation
- **Deploying** to (distributed) runtime environments

### SCA architecture: Standard definition

![SCA Diagram](image)

Event Driven Architecture

Event-Driven Architecture (EDA) is a software architecture pattern promoting the production, detection, consumption of, and reaction to events.

An event can be defined as "a significant change in state". EDA complements SoA, because services can be started by triggers such as events.
**Enterprise Service Bus (ESB)** is an infrastructure to facilitate SoA – a messaging backbone that does protocol conversion, message format transformation, and routing, accepting and delivering messages from various services and applications.

- It provides APIs for developing services and allowing services to interact with one another reliably.
- Message Oriented Middleware (MOM) is a key part of an ESB architecture.
- A service container is conceptually part of an ESB.

---

**Patterns for systems integration**

Thoughtful decisions on the service / component level help to reinforce and strengthen the properties of the baseline architecture chosen for a specific systems integration project.

Design patterns and practices for systems integration:

- Explicit, role-based interfaces
- Business Delegates
- Dependency Injection
- The service ABC
- Design by contract
- Messaging
- Semantic integration
- Structural integrity

A service (endpoint) consist of three parts – the service ABC:

- **Address** defines where the service is located.
- **Binding** describes how to reach the service in terms of protocols.
- **Contract** defines what the service provides in terms of operations.
Quality of service

A service interface defines the interaction between a service and a requester, but the interaction can have many additional characteristics:

- Reliability and performance guarantees
- Availability
- Security mechanisms
- Service coordination
- State and correlation

Quality of service is supported by:

- The choice of concrete integration technologies and middleware that support the required QoS concerns
- Design patterns for the respective QoS topics

Software Architecture Styles and Paradigms

Agenda

- Motivation
- Problem Frames
- Domain Properties
- Systems Integration

- Conclusion
The “job” of software architecture (2)

Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change [Grady Booch]

Architecture styles and paradigms
- Allow to get hands on an architecture and define its fundamental shape – regardless of, but in support for their concrete requirements
- Govern all subsequent design activities
- Help guiding concrete architecture and technology decisions – or avoiding religious technology debates

Therefore: choose the architecture styles and paradigms for your systems carefully and thoughtfully …
… because they are costly to change if inappropriate