Architectural Styles

Ana Moreira
Fernando Brito e Abreu

Banking software architecture
WebLogic Network Gatekeeper's software architecture

 GNOME Desktop Accessibility architecture
Open Healthcare Framework (OHF) architecture – client side

Open Healthcare Framework (OHF) architecture – server side
Architecture of J2SE 5.0 monitoring and management support

Eclipse plug-in architecture
Architectural Styles: Definition

- Is a set of rules and constraints that prescribe
  - **vocabulary**: which types of components, interfaces & connectors may be used in a system. Possibly introducing domain-specific types
  - **structure**: how components and connectors may be combined
  - **constraints**: on how they can be combined
  - **guidelines**: to support the application of the style (how to achieve certain system properties)
Architectural Styles (2)

- An architecture style defines a family of systems in terms of a pattern of structural organization.

- An architectural style defines
  - a vocabulary of components and connector types
  - a set of constraints on how they can be combined
  - one or more semantic models that specify how a system’s overall properties can be determined from the properties of its parts

Architectural Styles (3)

- Architectural styles are design paradigms for a set of design dimensions
  - Some architectural styles emphasize different aspects such as: subdivision of functionality, topology or interaction style

- Styles are open-ended; new styles will emerge

- An architecture can use several architectural styles

- Architectural styles are not disjoint
Styles vs Patterns

- Style is the classification of a system’s architecture according to those with similar patterns.
- A pattern is a common solution to a common problem; patterns may be classified as idioms, mechanisms, or frameworks.

<table>
<thead>
<tr>
<th>Architectural Styles/Patterns</th>
<th>Design Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emphasis</td>
<td>Reuse conceptual integrity</td>
</tr>
<tr>
<td>Abstraction Level</td>
<td>Architecture</td>
</tr>
<tr>
<td>Use</td>
<td>Concept that needs to be adapted</td>
</tr>
<tr>
<td></td>
<td>Directly applicable</td>
</tr>
</tbody>
</table>

Architectural Styles (or Patterns)

- Object-oriented design patterns express collaboration of classes and objects.
- Architectural patterns express fundamental structural organisation schemas for software systems:
  - set of predefined subsystems
  - specify responsibilities
  - rules and guidelines for organising subsystem relationships
- Architectural patterns exist for:
  - bringing structure into a system
  - organising distributed, interactive, adaptable systems
We can do anything…

Which guidelines and lessons learned to follow in order to get a “good architecture”?

...Architectural Styles Help Structuring

- Architectural styles define high-level design rules
  - Assign responsibilities to components (client, server…)
  - Prescribe patterns of interaction: restrict the way in which components can be connected
  - Promote fundamental principles
    - separation of concerns and anticipation of change
    - low coupling
    - high cohesion

- Architectural styles are based on success stories
  - Almost all compilers are build as “pipe-and-filter”
  - Almost all network protocols are built as “layers”
Architectural Styles and Qualities

An awareness of these styles can simplify the problem of defining system architectures

◆ some styles are likely to promote certain non-functional requirements

◆ e.g. layered architectures:
  ◆ favour reusability and maintainability
  ◆ hinder performance

However, most large systems are heterogeneous and do not follow a single architectural style

Catalogue of Architectural Styles

Styles are…
◆ … pieces of architecture knowledge
◆ … documented in the form of:
  ◆ context-problem/solution/consequences/…

Catalogues are available in books on software engineering and software architecture
◆ e.g. Sommerville’s book contains the discussion of several architectural styles…
Software Architecture Catalogues

- A System of Patterns: Pattern-Oriented Software Architecture, F. Buschman et al.
- Object-Oriented Modeling and Design, J. Rumbaugh
- Design and use of software architectures, J. Bosch.
- Software Architecture in Practice, P. Clements, L. Bass & R. Kazman
- Applied Software Architecture, C. Hofmeister et al.
- Software Product Lines, P. Clements & L. Northrop
- Software Architectures: Perspectives on an Emerging Discipline, M. Shaw.
- ...
Pipe & Filter style

Introduction
Pipe and Filter

- **Concept:** series of filters / transformations where each component is consumer and producer

- **Components:** filters (or pumps or sinks)
- **Connectors:** pipes
- **Interaction style:** data streaming
Topology examples
Pipe and Filter

- **Feedback-loops**: useful when a system does continuous controlling of a physical system

- **Splitting pipes**: this will allow the parallelising the processing

- **Producer (Pump)**: Produces data and puts it to an output port that is connected to the input end of a pipe.

- **Consumer (Sink)**: Gets data from the input port that is connected to the output end of a pipe and consumes the data.

Constraints
Pipe and Filter

Semantic Constraints

- **Filters are independent entities**
  - No sharing of state among filters
  - Identity of their predecessor / successor is unknown
Constraints
Pipe and Filter

- **Constraints** about the way filters and pipes can be joined:
  - Unidirectional flow
  - Control flow derived from data flow

- **Behaviour Types**:
  a. **Batch sequential**
     - Run to completion per transformation
  b. **Continuous**
     - Incremental transformation
     - Variants: push, pull, asynchronous

Typical uses
Pipe and Filter

- **Unix shell programming**
  - Unix commands alone are powerful, but when we combine them together, we can accomplish complex tasks with ease.
  - The way we combine Unix commands is through using pipes ("|") and filters (e.g. `cat`, `awk`, `sort`)

```bash
$ cat basket.txt
Layer1 = cloth
Layer2 = strawberries
Layer3 = fish
Layer4 = chocolate
Layer5 = punch cards
$ cat basket.txt | awk -F= '{print "HAS: " $2}' | sort
HAS: chocolate
HAS: cloth
HAS: fish
HAS: punch cards
HAS: strawberries
$ 
```
Typical uses
Pipe and Filter

Compiler

Source
Lexical Analysis → Syntactic Analysis → Semantic Analysis → Optimize → Code Gen → Code

Typical uses
Pipe and Filter

Digital camera / Signal processing

Active push
Propagated pull in passive chain

Object

Active push + pull

Camera control

user interface

selected options

command execution

RAM

Kodak camera chip (continuously capturing)
### Advantages

**Pipe and Filter**

- Very adequate for batch applications
- Simplicity:
  - no complex component interactions
  - easy to analyze (deadlock, throughput, …)
  - easy to compose filters
  - easy to debug / maintain
  - easy to reuse

### Disadvantages

**Pipe and Filter**

- Interactive applications are difficult to create
- Performance:
  - Enforcement of lowest common data representation, may lead to overall overhead
  - If output can only be produced after all input is received, an infinite input buffer is required (e.g. sort filter)
- If bounded buffers are used, deadlocks may occur
Quality factors

Pipe and Filter

- **Extendibility**: extends easily with new filters
- **Reuse**: +
- **Flexibility**:
  - functionality of filters can be easily redefined,
  - control can be re-routed
- **Security**: –
- **Performance**: allows straightforward parallelisation

---

Heuristics for choice

Pipe and Filter

- If a system can be described by a **regular interaction pattern** of a collection of processing units at the same level of abstraction;
  - e.g. a series of incremental stages (horizontal composition of functionality);
- If the computation involves the **transformation of streams** of data (processes with limited fan-in/fan-out)
Call & Return styles

- Functional decomposition
  - Typical style in the classical (procedural) programming paradigm

- Abstract Data Types / Object-Oriented decomposition
  - Information (representation, access method) hiding
  - Typical style in the Object-oriented paradigm

- Layered decomposition
  - Each level only communicates with its immediate neighbours
  - Client-server

- Services decomposition
  - Typical style in web services / SOAs
Functional Decomposition style

Introduction

Functional decomposition

- **Concept**: a system is a set of hierarchically organized functions, starting on a main program that calls a set of more abstract functions, each corresponding to a functional part of the system.

- Functionality is decomposed subsequently on a delegation-like structure, until basic operations are performed by lower level functions.

- **Components**: functions / procedures (void functions)

- **Connectors**: message calls

- **Interaction style**: functions call other functions
Introduction

Functional decomposition

- Hierarchical decomposition
  - definition/use relationship

- Single thread of control
  - supported directly

- Subsystem structure implicit
  - subroutines aggregated into modules

- Hierarchical reasoning
  - correctness of any part depends on the correctness of its subordinate parts

Topology examples

Functional decomposition

![Diagram of hierarchical decomposition](image)

Main

Sub1

Sub2

Sub3
## Advantages
### Functional decomposition
- Matches well the GUI interaction paradigm (e.g. menus, options)
- Appropriate for solving scientific/mathematical problems
- Many libraries of fine-grained functions exist (e.g. in Fortran or C)
- Easy to follow execution traces (therefore simplifying debugging activities)

## Disadvantages
### Functional decomposition
- Problem decomposition
  - No natural correspondence with real-world entities
- Maintenance and reuse
  - High coupling (side effects are spread easily)
  - Reusability only achievable at lowest decomposition level
- Internal representations is often exposed
  - Difficult to enforce data integrity
Object-oriented style

Introduction

Object-oriented

- Encapsulation
  - restricted access to data/representations

- Inheritance
  - shared definitions of functionality

- Polymorphism
  - actual operation to call determined at runtime

- Reuse/Maintenance
  - encapsulation/abstraction promotes separation of concerns
**Architectural Styles**

### Topology examples
**Object-oriented**

![Topology diagram]

### Advantages
**Object-oriented**

- **Problem decomposition**
  - natural correspondence with real-world entities
  - inheritance allows shared definitions

- **Maintenance and reuse**
  - decreased coupling (change propagation)
  - increased reusability (especially frameworks)

- **Protection of internal representations**
  - encapsulation allows data/state integrity to be preserved
Disadvantages

**Object-oriented**

- Design is harder
  - forces more up-front brain-work
- Side effects
  - encapsulation requires discipline in design
  - many objects can access a single resource
- Identity
  - need to know (import) an object/method’s name
- Inheritance & Polymorphism
  - often non-intuitive

**Layered / Component based style**
Introduction
Layered

- Used to model the **interfacing of sub-systems**.

- Organises the **system into a set of layers** (or abstract machines, or components) each of which provide a **set of services**.

- Supports the **incremental development** of sub-systems in different layers.

- When a layer interface changes, **only the adjacent** layer is affected.

Introduction
Layered

- Each layer provides some facilities
  - hides lower layer
  - provides service to higher layer

- Each layer serves distinct functions
  - upper layers are “virtual machines”
  - support for portability
  - bottom layer is hardware/kernel

- Various scoping regimes
  - opaque versus transparent layers
Topology examples
Layered

- Vertical layered (tiered-based, pure layered)

- Horizontal layered (component based)
## Advantages

### Layered

- Support design based upon increasing levels of abstraction
- Aids in portability
  - each layer is an abstract machine
- Aids in modifiability
  - In pure layered style each layer interacts with at most 2 others
- Aids in reuse
  - standard interfaces must be created

## Disadvantages

### Layered

- Performance penalty
  - e.g. semantic feedback in user interfaces
- May be the wrong model
  - e.g. we may need to control deep functionality
- Finding the right abstractions is difficult
  - if the abstractions are wrong, layers need to be bridged
- Layer bridging
  - often ruins the model
Introduction

Client-server

- **Components:**
  - **Client** - a program that makes requests (to the servers) and handles input/output with the system environment
  - **Server** - a program that services requests from clients

- **Connectors:**
  - Connection channels (e.g. RPCs, http, ODBC)

- **Topology:**
  - Clients are connected to servers
  - Clients are not connected to other clients
  - Binding: from compile to invocation time
Introduction

Client-server

- **Client/Server system**: an application that is built from clients and servers
  - Clients know of (depend on) servers
  - But servers need not know of clients.

- Pitfall: keep an eye on (hidden) dependencies
  - Servers may need to know clients (e.g. for licensing purposes)

Typical application area:
- Distributed multi-user (business) information systems
- Multiple users want to share and exchange data
3-tier reference model

Client-server

- **Presentation logic** (aka GUI): anything that involves system/user interaction; e.g. dialogs (management), forms, reports

- **Application logic** (aka business tier): where the functionality of the application resides; where the actual computation of the system takes place

- **Data management** (aka database tier): storing, retrieving and updating data

---

3-tier reference model - example

Client-server

**Presentation tier**
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

**Logic tier**
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

**Data tier**
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
Historical evolution (1970’s)

Client-server

- Monolithic / centralized servers
- Dumb terminals (Matrix-like 😊)
- Personal computers were invented then

![Diagram of 1-tier architecture with layers: presentation logic, application logic, data management]

Historical evolution (1980’s)

Client-server

- Processing was done on clients (Personal Computers)
- Scalability was limited to ±10 due to contention for files and volume of data-transfer

![Diagram of client-server architecture with shared data layers: presentation logic, application logic, data management]
Historical evolution (1990’s)  
**Client-server**

- Rather than having the client do the processing …
  - Move processing power to server such that the server sends a (condensed) response to request rather than a whole file

**Topology examples:** **Thick client**

**Client-server**

- Significant processing at the client-side
- Application component at the client side usually uses a scripting language (e.g. JavaScript)
- Application component at the server side can also use scripting languages (e.g. PHP)
Topology examples: **thin client**

Client-server

- Largest part of processing at the server-side

![Diagram of client-server architecture with roles: presentation logic, application logic, data management, functional processing, database storage and access, and WWW Browser.]

**Advantages**

Client-server

- **Thick Client**
  - Processing effort is distributed
  - Server’s CPU is spared

- **Thin Client**
  - Network load: low
  - Configuration management: simple (only server)
  - Security: concentrated at server
  - Robustness: stateless clients => easy fault recovery
Disadvantages
Client-server

- Thick Client
  - Network load: high
  - Configuration management: complex (both client & server)
  - Security: complex (both client & server)
  - Robustness: clients have state => complex fault recovery

- Thin Client
  - Processing power is a burden on server(s)
  - Clients’ CPU is wasted

3-tier reference model
Client-server
Multitier example

Client-server

- The number of tiers is increasing:
  - Client tier
  - Security tier
  - Application server
  - Integration tier
  - Data storage tier

Advantages (wrap-up)

Client-server

- Good support for client mobility
- Distribution of data is not difficult
- Makes effective use of networked systems
- Scalability based on cheap hardware (clients)
- Easy to add new servers or upgrade existing servers
Disadvantages (wrap-up)  
Client-server

- Data interchange may be inefficient
- Redundant management in each server
- No central register of names and services
  - It may be hard to find out what servers and services are available
  - Binding between tiers is “hardwired” in the two upper tiers
    - Client code has application server address
    - Application code has database server address

Blackboard style
Introduction

Blackboard

**Concept:** Concurrent transformations on shared data (multiple clients sharing single blackboard)

**Components:** processing units (typically knowledge source)

**Connectors:** blackboard interaction style: asynchronous

**Topology:** one or more transformation-components may be connected to a data-space; there are typically no connections between processing units (bus-topology)

---

Introduction

Blackboard

**Behaviour Types:**

a. **Passive repository**
   - Accessed by a set of components; e.g. database or server

b. **Active repository**
   - Sends notification to components when data of interest changes; e.g. blackboard or active database

**Constraints:**

Consistency of repository: various types of (transaction) consistency
**Introduction**

**Blackboard**

- Data may be structured (DB) or unstructured
- Data may be selected based on content
- Applications may insert/retrieve different data-type per access.

**Topology examples:**

**Blackboard**

[Diagram showing chat clients and a chat transcript]

*Online Chat Room*
Topology examples: Blackboard

Hearsay was developed in the 1970’s by Raj Reddy et al. at Carnegie Mellon University.


Advantages Blackboard

- Reusable & heterogeneous knowledge sources
- Support for fault tolerance and robustness by adding redundant components
- General style suitable for mobile applications with wireless connectivity problems (e.g. RSS feeds)
Disadvantages
Blackboard

- Distributed implementation is complex
  ◆ distribution and consistency issues
- Blackboard may be a bottleneck with too many clients
  ◆ Not that much in dematerialized blackboards (e.g. chatrooms)

Quality Factors
Blackboard

- **Extensibility**: components can be easily added
- **Flexibility**: functionality of components can be easily changed
- **Robustness**:  
  + components can be replicated,  
  - blackboard is single point of failure
- **Security**:  
  - all process share the same data  
  + security measures can be centralized around blackboard
- **Performance**: easy to execute in parallel fashion  
  consistency may incur synchronization-penalty
Implicit Invocation style
(Event-Based Systems)

Introduction
Implicit Invocation Style

Components:
- Programs or program entities that announce and/or register interest in events
  - Events represent happenstances inside an entity that may (or may not) be of interest to other entities

Connectors:
- Direct registration with announcing entities
- Or, explicit event broadcast and registration infrastructure

WEBSTER:
hap·pen·stance
Function: noun
Etymology: happen + circumstance
Semantics: a circumstance especially that is due to chance
**Introduction**

**Implicit Invocation Style**

- **Configurations**
  - Implicit dependencies arising from event announcements and registrations

- **Underlying computational model**
  1. Event announcement is broadcast
  2. Procedures associated with registrations (if any) are invoked

---

**Implicit vs. Explicit Invocation**

**Explicit**

```
op1
  op2
    op3
```

**Implicit**

```
ev1
  op2
    op3
```

*Typical implementation by the Observer design pattern*
Advantages
Implicit Invocation Style

- Allows for decoupling and autonomy of components
- Enhances reuse and evolution
- Easy to introduce new components without affecting existing ones

Disadvantages
Implicit Invocation Style

- Error handling – who should handle exceptions?
- Components announcing events have no guarantee of getting a response
- Event abstraction does not cleanly lend itself to data exchange
Service-Oriented style
(presented in a separate collection of slides)

Wrap-up
Architectural Styles

Benefits of Using Styles

- **Design reuse**
  - Well-understood solutions applied to new problems

- **Code reuse**
  - Shared implementations of invariant aspects of a styles

- **Understandability of system organization**
  - A phrase such as “client-server” conveys a lot of information

- **Interoperability**
  - Supported by style standardization
  - e.g., CORBA, JavaBeans

- **Style-specific analyses**
  - Enabled by the constrained design space

- **Visualizations**
  - Style-specific depictions matching engineers’ mental models
Choosing the Right Style

- Ask questions on whether a certain style makes sense
  - The Internet as a blackboard
  - Does that scale?
- Draw a picture of the major entities
- Distribution
  - e.g. client-server, service-oriented architectures
  - e.g. context-aware mobile application: implicit invocation
- Look for the natural paradigm
  - Almost all compilers are built as “pipe-and-filter”
  - Almost all network protocols are built as “layers”
- Look for what “feels right”

The KWIC case study
KWIC (Key Word in Context) case study [Garlan, Shaw, 1994]

- Problem description [Parnas]:
- The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters.
- Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line.
- The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

KWIC: Example

Input: Sequence of lines
- An Introduction to Software Architecture
- Key Word in Context

Output: Circularly shifted, alphabetized lines
- An Introduction to Software Architecture
- Architecture An Introduction to Software
- Context Key Word in
- in Context Key Word
- Introduction to Software Architecture An
- Key Word in Context
- Software Architecture An Introduction to
- to Software Architecture An Introduction
- Word in Context Key
**KWIC: Shared memory solution**

- **Master Control**
  - Input
  - Circ Shift
  - Alphabetize
  - Output
- **Characters**
- **Index**
- **Alphabetized Index**

- **Advantages:** good performance
- **Disadvantages:** poor modifiability

---

**KWIC: ADT solution**

- **Master Control**
  - Input
  - Setup
  - Search
  - Characters
  - Circular Shift
  - Word
  - Output
- **Alphabetic Shift**

- **Advantages:** good support for reuse
- **Disadvantages:** poor extendability
KWIC: Implicit Invocation Solution

- Advantages: good support for extendability
- Disadvantages: poor control, space utilization

KWIC: Pipes and Filters Solution

- Advantages: good support for reuse; intuitive
- Disadvantages: poor space utilization; non-interactive
## KWIC: Comparisons

<table>
<thead>
<tr>
<th></th>
<th>Shared Data</th>
<th>Abstract Data Type</th>
<th>Implicit Invocation</th>
<th>Pipe and Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change in Algorithm</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Change in Data Rep.</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Change in Function</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Performance</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Reuse</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>