Contents

Key Principals of All Styles

- Seperation of concerns: Functionality should not be spread amongst components unnecessarily
- Single responsibility: Components should do only one thing, and do it well
- Principle of least knowledge: Talk to the interface

Style

Category	Architectural Styles
Communication	Service Orientated, Message Bus
Deployment	Client/Server, N-Tier, 3-Tier
Domain	Domain driven design
Structure	Component based, Object orientated, Layered

Client/Server

- Describes a distributed system that involves a client, and a server
- Simplest form has a server application accessed directly by multiple clients
- e.g. Email clients, git clients, database query tools

Variation

- Client Queue Client: Server acts as a passive queue, which clients access
- Peer to Peer: Developed from CQC, P2P allows clients and servers to swap roles to distribute and synchronise files
  - Messaging systems
- Application Servers: Server hosts and executes application, which clients access them through a UI
  - Manage systems for large applications

Pros/Cons

Pros	
Security	Since server has all the data, easier to secure

Pros	
Centralised Data Store	Access and updates to data are propagated to all clients
Maintenance	Client and server can generally be maintained independently

Cons	
Extensibility	Dependence on a single, centralised server
Reliability	Central server introduces single-point-of-failure

Layered

- Group related functionality into distinct layers, and stack layers on top of each other
- Components in one layer can only interact with other components in the same layer, or in the layer below it

Properties

- Abstraction: Higher layers are not aware of the complexity below
- Encapsulation: Internal details are not revealed and therefore interface is respected
- Reusable: High cohesion and loose coupling make reuse easy
- Typical applications are
  - Accounting systems
  - Web based applications
  - Operating systems

Pros/Cons

Pros	
Abstraction	Complexity is easier to handle when broken down into simpler chunks
Maintainability	Allow separation of concerns
Testability	

Cons	
Performance	Overhead of calls percolating through the layers
Unnecessary complexity	Ease of understanding leads to frequency of mis-use

> There is no problem in Computer Science that cannot be solve with another layer

Message Bus

- A massage based approach to creating an application using a channel to centralise communication
- Allows a system to be built of interacting components that know nothing about each other, except for the interface of the bus

Properties

- Decoupling of component functionarlity
- Modifiability of components, independent of each other
- Scalability of the system can be increased by simply increasing the capacity of the bus
- Typical applications
  - Financial/Trading applications
  - Event handling in operating systems

Pros/Cons

Pros	
Simplicity	Connecting diverse components is exceedingly simple
Performance	System can process messages as fast as the bus can deliver them

Cons	
Security	A message passed on the bus is visibile to everyone
Single point of failure	If the bus fails, everything fails

Pipe and Filter

- A data flow based approach the decomposes an application into discrete tasks, each of which is performed by a single filter
- Data is pushed from one filter to the next using a common pipeline

Pros/Cons

Pros	
Shared-Nothing	Each filter is independent of the other
Highly Concurrent	Can seperate data flow into as many streams as needed
Flexible Topology	Topology is defined outside of the pipe and filter

Cons	
Efficiency Loss	Cost of data transfer can be high
Error Handling	Difficult to handle errors in a coordinated manner

Questions

- What's the difference between
  - A Pipe-and-Filter and a Message-Bus styled architecture
    * Bus vs Filter is responsibile for passing on the data
    * Your capacity in a Message-Bus is limited by the Bus
    * In a Pipe-And-Filter, its limited by the filter
    * Data moves in one sequence for Pipe-and-Filter
    * Many points can access data at the same time in a Message-Bus
  - A Layered Architecture and a Pipe-and-Filter architecture
    * In a layer, there is dependence between the layers
    * Layer two layers above has an abstraction
    * No abstraction in Pipe-And-Filter
    * A layer has no idea what's above or below

Why Styles?

- Common Language

  - Lots of steps between when software is first articulated and deployed
  - Easier to say it functions in a certain style
- Technology agnostic
- Inclusive of patterns and principles

More styles

Model-View-Controller

- Divides an interactive application into three parts:
  - Model - Responsible for data and data management
  - View - Display information to the user
  - Controller - Handle user-input, validate, etc.

Typical Properties of Domain

- Same information needs to be presented in multiple ways
- Model
  - Encapsulates application state
  - Responds to state queries
  - Exposes application functionality
  - Notifies views of chances
- View
  - Renders the models
  - Requests updates from models
  - Sends user gestures to controller
  - Allows controller to select view
- Controller
  - Defines application behaviour
  - Maps user actions to model updates
  - Selects view for response
  - One for each functionality

Pros/Cons

Pros	
Multiple Views	From the same model, different views can be instantiated dynamically

Pros	
Synchronised Views	Change to data is immediately reflected to all viewers
Pluggabilty	Views and controller can be changed without affecting the model

Cons	
Increased Complexity	Simple menu items become excessively complex
Excessive Updates	All changes to model may not need to be propagated
Inefficiency of Data Access	Levels of indirection, in the name of de-coupling

Blackboard

- Useful for problems where no deterministic strategies are known
- Several specialised subsystems pool knowledge to build an approximate/possible answer

Typical Propeties of Domain

- Complete search of solution space if not feasible
- Domain is immature, so no known algorithms to solve problem
- Different algorithms solve different partial problems
- Each person is a knowledge source
  - Comes up with hypothesis
  - Until hypothesis is rejected, it stays on the board

Solution Style

- *Opportunistic* problem-solving using independent experts using a common data-structure

Implementation

- Blackboard
- Knowledge-sources
- Control
- Different Experts

  - Put in their hypothesis
  - "I think this is what's happening and this should happen next"

Pros/Cons

Pros	
Changeability	Supports changing of knowledge sources easily
Experimentation	Strict separation of components allows easy experimentation
Fault Tolerance	All results are hypothesis, so noise in data is okay
Potential Parallelism	Disjoint algorithms can work in parallel on solution space

Cons	
Testability	Results are not reproducible
Low Efficiency	Computationally costly to reject wrong hypothesis
High Development Effort	Since domain is ill-specified, takes years to build

Service-Orientated

- Enable application functionality to be providd and consumed as sets of services, published at a granulatiry relevant to the service consumer
- Services can be invoked, published and discovered, and are abstracted away from implementation

Typical Properties of Domain

- Atomated discovery and usage are essential
- Platform indepedence of service endpoint
- Formal contract places obligations on both consumers are providers

Implementation

- Service Orientated Architecture
  - Application Frontend
  - Service
    * Contract

    * Implementation
        · Business Logic
        · Data Store
    * Interface
- Service Repository
- Service Bus

Pros/Cons

Pros	
Reusability	Small, self-contained, loosely coupled functionality
Maintainability	Can change between versions, as long as contract is not violated
Scalability	Multiple instances can run on the same server

Cons	
Service Management	Orchestration of Choreography is complex
Overhead	Computationally constly to constantly validate parameters and use HTTP