AARHUS UNIVERSITET

# Software Architecture in Practice

## Architectural Patterns / Styles

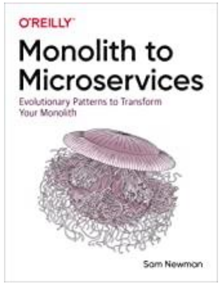
# Motivation

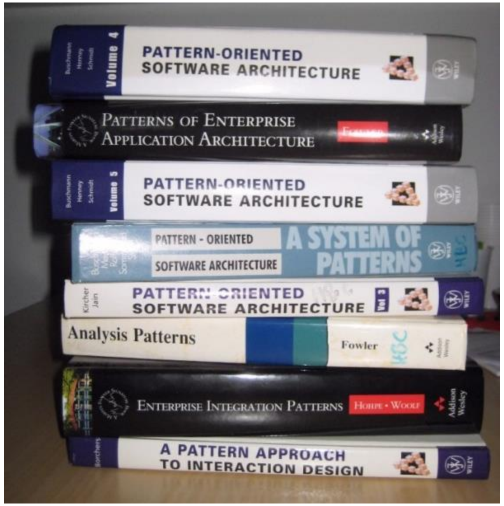
- **Pattern:** A well-proven solution to recurring problem
  - *Rule of three:* Used in three disjoint contexts;
  - Patterns are *discovered* not invented…
- In my mind *architectural patterns form the central toolbox of a software architect*
  - Given these architectural requirements, should I…
    - Build a client-server architecture? SOA? Microservice?
    - Should communication be Pub-Sub? RPC? Message Queue?
    - Should the UI be modelled over MVC or PAC?
- Otherwise we tend to ‘do the same thing as last time’
  - A three-tier client-server thingy ☺

# Bass et al.

- *An architectural pattern describes a **particular recurring design problem** that arises in **specific design contexts** and presents a **well-proven architectural solution** for the problem. The solution is specified by describing the **roles of its constituent elements**, their responsibilities and **relationships**, and the ways in which they **collaborate.*** [§3.4]
- Are more 'coarse-grained' than tactics
  - Often define an overall structure of (part of) the system
  - Often combining several patterns in any given system
    - *Client-server + Broker + Layers + Dep Injection/Test doubles + Load Balancer + Passive Redundancy + …*

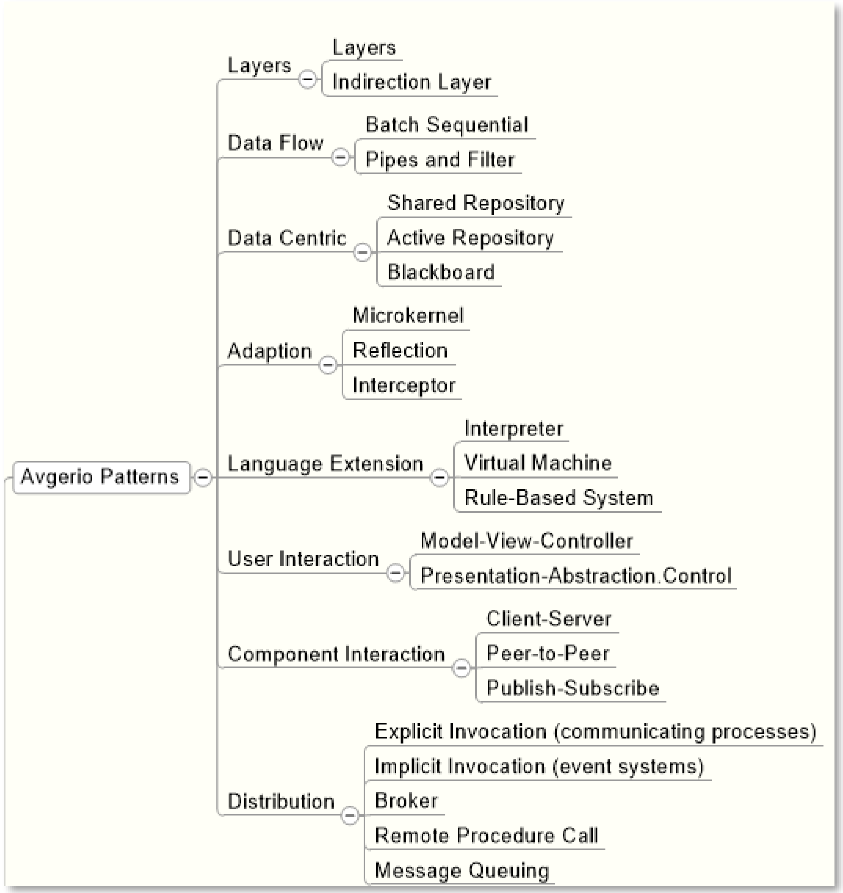
# Sources


O'REILLY
Monolith to Microservices
Evolutionary Patterns to Transform Your Monolith
Sam Newman

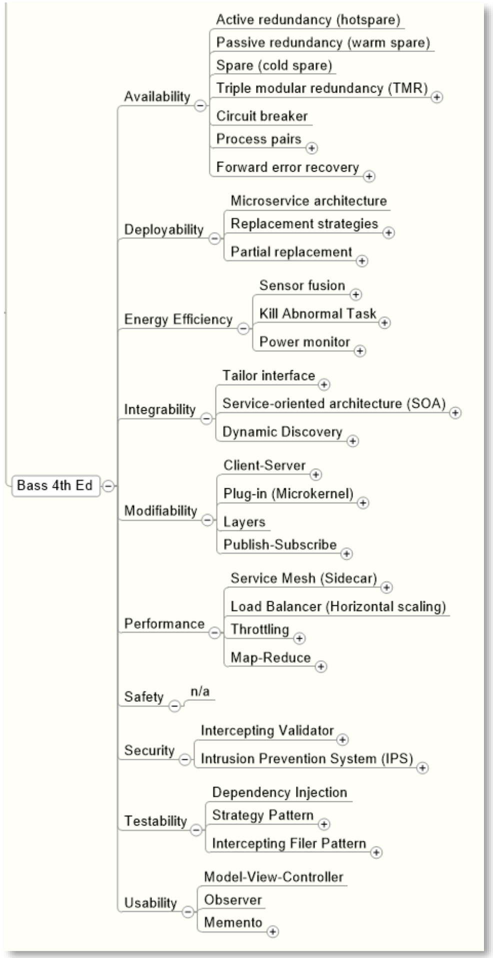

PATTERN-ORIENTED SOFTWARE ARCHITECTURE Volume 4
PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE
PATTERN-ORIENTED SOFTWARE ARCHITECTURE Volume 5
PATTERN-ORIENTED SOFTWARE ARCHITECTURE A SYSTEM OF PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE Vol 3
Analysis Patterns Fowler
ENTERPRISE INTEGRATION PATTERNS Hohpe · Woolf
A PATTERN APPROACH TO INTERACTION DESIGN

- Many sources…
- Our take
  - The minimal vocabulary of common and essential patterns
- Sources
  - Avgeriou et al.
    - An old (but excellent) article, but so are the patterns ☺
  - Bass et al.
    - 4th edition revises the set completely wrt. previous editions…
      - But it has some merit, trying to classify wrt. to QA
      - **But I seriously miss some diagrams…**

# The Overview



- Avgerio Patterns
  - Layers
    - Layers
    - Indirection Layer
  - Data Flow
    - Batch Sequential
    - Pipes and Filter
  - Data Centric
    - Shared Repository
    - Active Repository
    - Blackboard
  - Adaption
    - Microkernel
    - Reflection
    - Interceptor
  - Language Extension
    - Interpreter
    - Virtual Machine
    - Rule-Based System
  - User Interaction
    - Model-View-Controller
    - Presentation-Abstraction.Control
  - Component Interaction
    - Client-Server
    - Peer-to-Peer
    - Publish-Subscribe
  - Distribution
    - Explicit Invocation (communicating processes)
    - Implicit Invocation (event systems)
    - Broker
    - Remote Procedure Call
    - Message Queuing



- Bass 4th Ed
  - Availability
    - Active redundancy (hotspare)
    - Passive redundancy (warm spare)
    - Spare (cold spare)
    - Triple modular redundancy (TMR)
    - Circuit breaker
    - Process pairs
    - Forward error recovery
  - Deployability
    - Microservice architecture
    - Replacement strategies
    - Partial replacement
  - Energy Efficiency
    - Sensor fusion
    - Kill Abnormal Task
    - Power monitor
  - Integrability
    - Tailor interface
    - Service-oriented architecture (SOA)
    - Dynamic Discovery
  - Modifiability
    - Client-Server
    - Plug-in (Microkernel)
    - Layers
    - Publish-Subscribe
  - Performance
    - Service Mesh (Sidecar)
    - Load Balancer (Horizontal scaling)
    - Throttling
    - Map-Reduce
  - Safety
    - n/a
  - Security
    - Intercepting Validator
    - Intrusion Prevention System (IPS)
  - Testability
    - Dependency Injection
    - Strategy Pattern
    - Intercepting Filer Pattern
  - Usability
    - Model-View-Controller
    - Observer
    - Memento

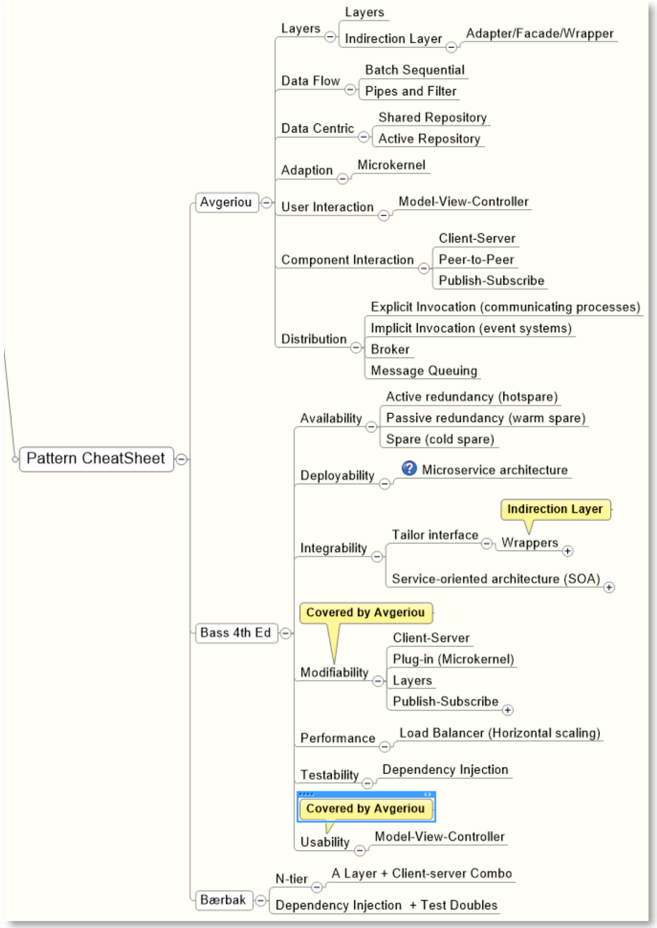
# Gosh!!!

- “53 patterns??? Am I to know all the details of each and everyone at the exam?”
  - No…
    - Overlap between authors (same thing, different name)
    - Some are variants of same *theme*
      - Implicit Invocation ~ Message Queueing ~ Publish-Subscribe
    - Some are “new name/version of a design pattern”
      - Layers = Façade
      - Indirection layer ~ Intercepting Filter/Validator ~ Tailor interface = Decorator/Proxy
    - Some are ‘for other times’
      - Microservice fagpakke is about deployability
    - Some are simply not curriculum…

# So – The CheatSheet for exam

- Down to 22…
  - Or less, considering the overlap…

- Many should be known already to you ☺



- Pattern CheatSheet
  - Avgeriou
    - Layers
      - Layers
      - Indirection Layer
        - Adapter/Facade/Wrapper
    - Data Flow
      - Batch Sequential
      - Pipes and Filter
    - Data Centric
      - Shared Repository
      - Active Repository
    - Adaption
      - Microkernel
    - User Interaction
      - Model-View-Controller
    - Component Interaction
      - Client-Server
      - Peer-to-Peer
      - Publish-Subscribe
    - Distribution
      - Explicit Invocation (communicating processes)
      - Implicit Invocation (event systems)
      - Broker
      - Message Queuing
  - Bass 4th Ed
    - Availability
      - Active redundancy (hotspare)
      - Passive redundancy (warm spare)
      - Spare (cold spare)
    - Deployability
      - ? Microservice architecture
    - Integrability
      - Tailor interface
        - Wrappers (Indirection Layer)
      - Service-oriented architecture (SOA)
    - Modifiability (Covered by Avgeriou)
      - Client-Server
      - Plug-in (Microkernel)
      - Layers
      - Publish-Subscribe
    - Performance
      - Load Balancer (Horizontal scaling)
    - Testability
      - Dependency Injection
    - Usability (Covered by Avgeriou)
      - Model-View-Controller
  - Bærbak
    - N-tier
      - A Layer + Client-server Combo
    - Dependency Injection + Test Doubles

# Pattern versus Style

- In the old days, we talked about **architectural style:**
  - Set of **element types** **Components**
  - Set of **interaction** mechanisms **Connectors**
  - **Topology** of components
  - Semantic **constraints**

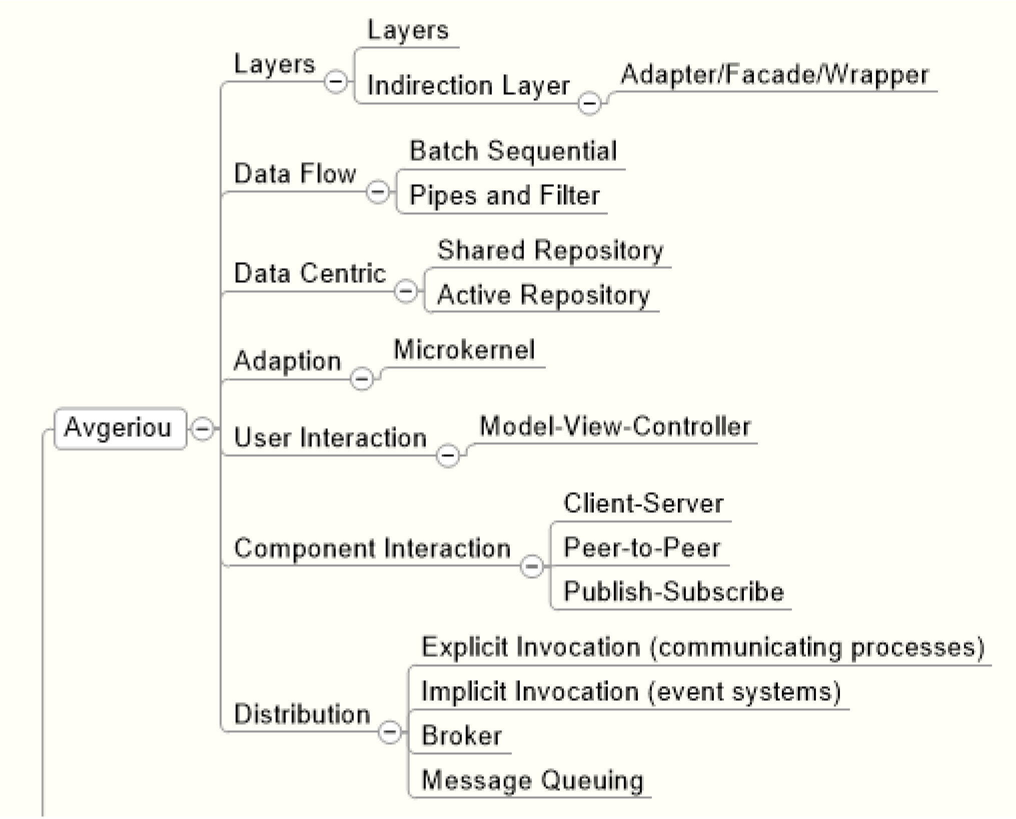
- Exercise:
  - Describe each for ‘client-server’ architectural pattern
- Conclusion:
  - Basically same same…

# Avgeriou et Zdun

Our main source as I like diagrams ☺

# CheatSheet

- A lot of classic patterns…



AARHUS UNIVERSITET

# Layered

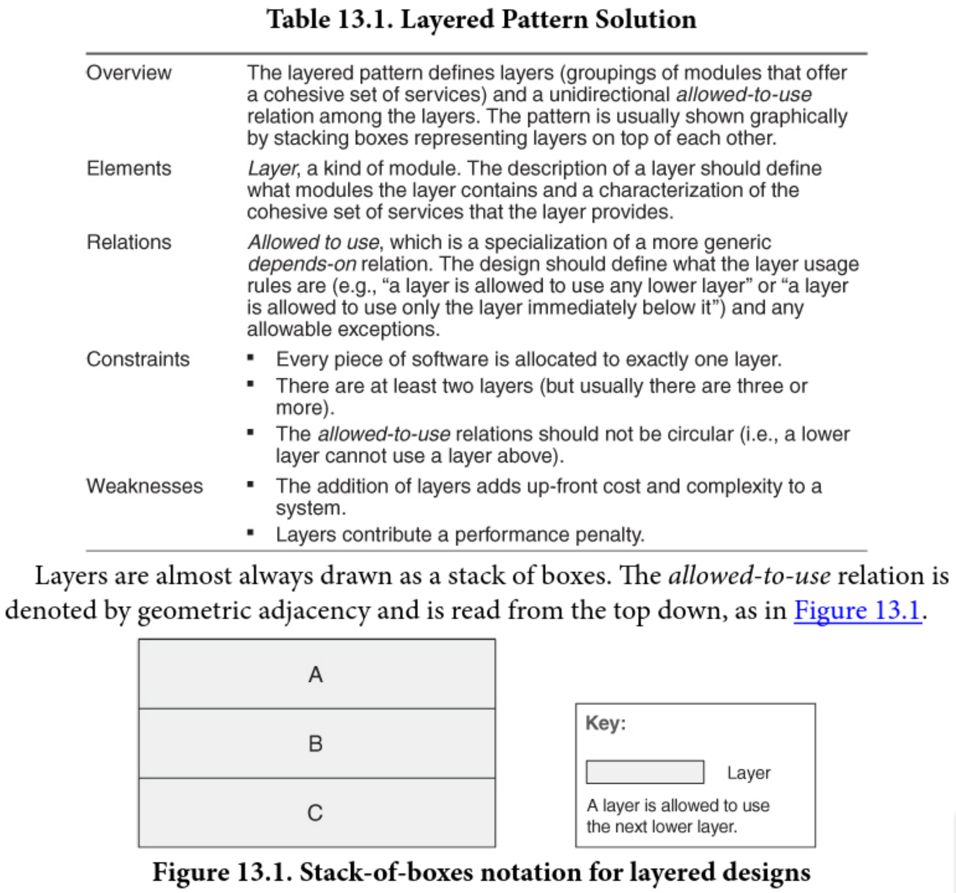


Table 13.1. Layered Pattern Solution

| | |
|---|---|
| Overview | The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other. |
| Elements | *Layer*, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides. |
| Relations | *Allowed to use*, which is a specialization of a more generic *depends-on* relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions. |
| Constraints | ▪ Every piece of software is allocated to exactly one layer.<br>▪ There are at least two layers (but usually there are three or more).<br>▪ The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above). |
| Weaknesses | ▪ The addition of layers adds up-front cost and complexity to a system.<br>▪ Layers contribute a performance penalty. |

Layers are almost always drawn as a stack of boxes. The *allowed-to-use* relation is denoted by geometric adjacency and is read from the top down, as in Figure 13.1.

A

B

C

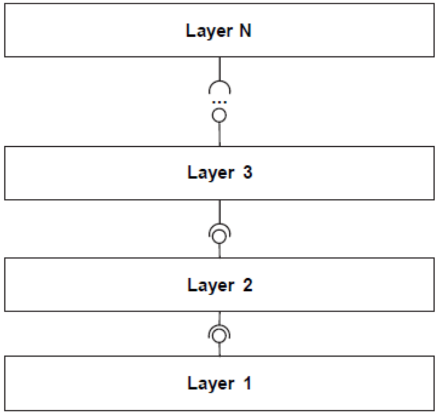
Key:

▭ Layer

A layer is allowed to use the next lower layer.

Figure 13.1. Stack-of-boxes notation for layered designs

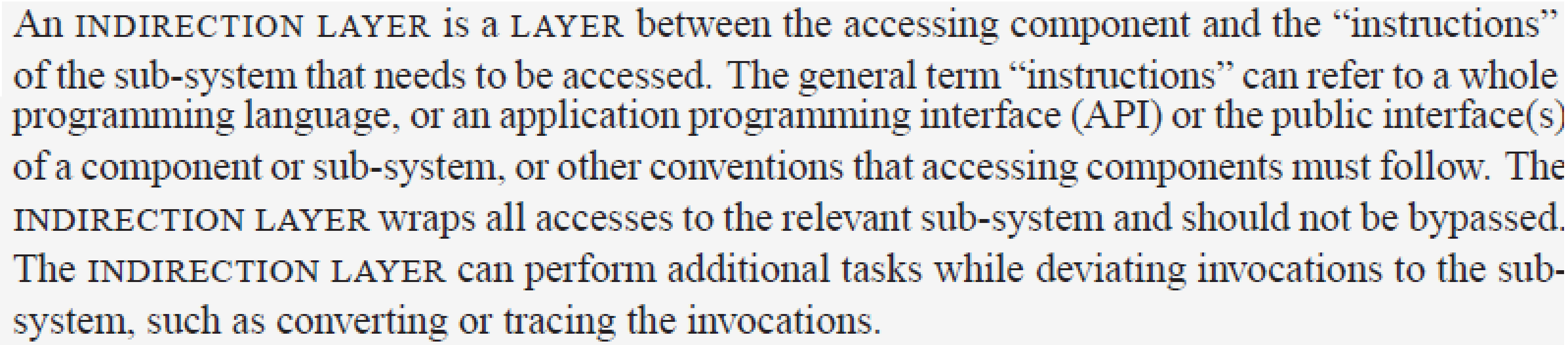
Note: Not a 'tier'
Layer = module
Tier = process



Main QA: Modifiability

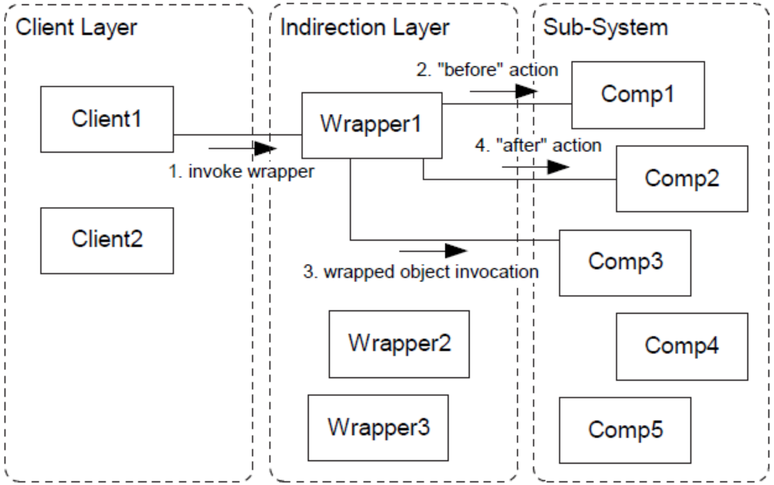
# Indirection Layer

- Adaption one API to another…



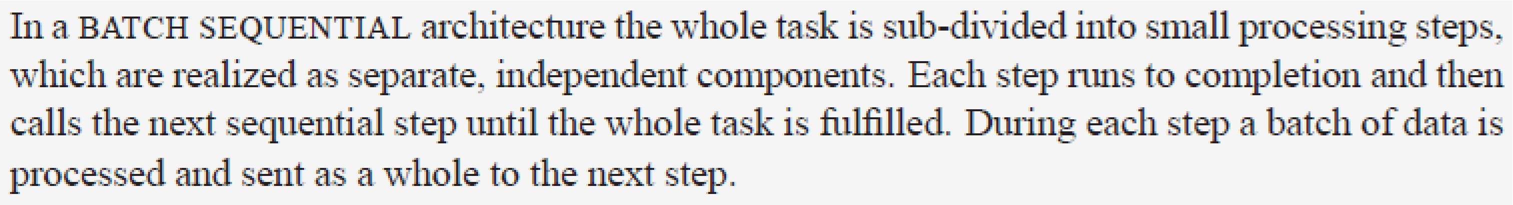
An INDIRECTION LAYER is a LAYER between the accessing component and the "instructions" of the sub-system that needs to be accessed. The general term "instructions" can refer to a whole programming language, or an application programming interface (API) or the public interface(s) of a component or sub-system, or other conventions that accessing components must follow. The INDIRECTION LAYER wraps all accesses to the relevant sub-system and should not be bypassed. The INDIRECTION LAYER can perform additional tasks while deviating invocations to the sub-system, such as converting or tracing the invocations.

- Bass et al.
  - *Tailor interface – Wrapper*
  - The **Adapter** or **Façade** or **Proxy** or **Decorator** pattern…



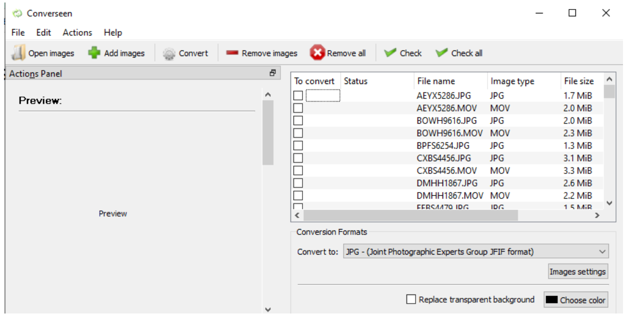
# Batch Sequential

- Simple data processing pattern – data flow through steps



In a BATCH SEQUENTIAL architecture the whole task is sub-divided into small processing steps, which are realized as separate, independent components. Each step runs to completion and then calls the next sequential step until the whole task is fulfilled. During each step a batch of data is processed and sent as a whole to the next step.

- Example:
  - Batch conversion of my iPhone's HEIC pictures into JPG
  - Loop
    - Read image, convert, write



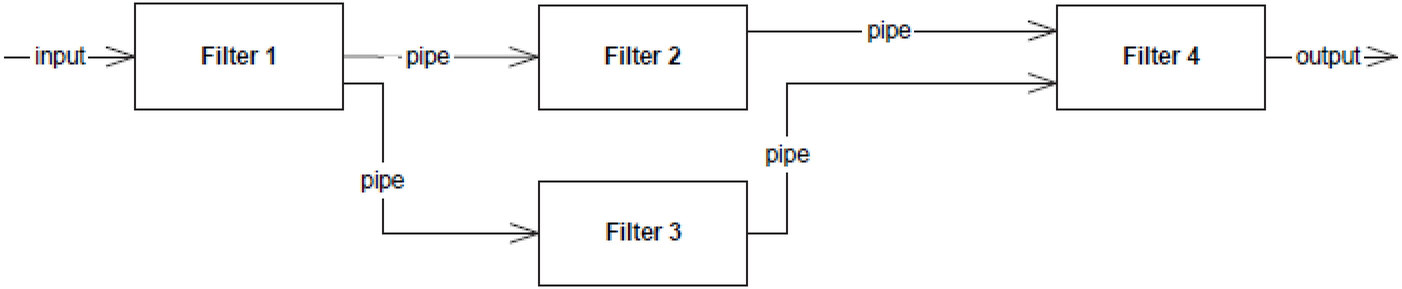
| To convert | Status | File name | Image type | File size |
|---|---|---|---|---|
| ☐ | | AEYX5286.JPG | JPG | 1.7 MiB |
| ☐ | | AEYX5286.MOV | MOV | 2.0 MiB |
| ☐ | | BOWH9616.JPG | JPG | 2.0 MiB |
| ☐ | | BOWH9616.MOV | MOV | 2.3 MiB |
| ☐ | | BPFS6254.JPG | JPG | 1.3 MiB |
| ☐ | | CXBS4456.JPG | JPG | 3.1 MiB |
| ☐ | | CXBS4456.MOV | MOV | 3.3 MiB |
| ☐ | | DMHH1867.JPG | JPG | 2.6 MiB |
| ☐ | | DMHH1867.MOV | MOV | 2.2 MiB |
| ☐ | | EEBS4479.JPG | JPG | 1.5 MiB |

Convert to: JPG - (Joint Photographic Experts Group JFIF format)

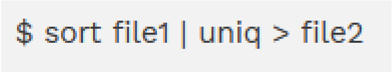
☐ Replace transparent background

# Pipes and Filter

- Computation as data flowing through *pipes* connecting *filters* that each modify/add/remove data



- Note: Each filter is a *concurrent process*
- Example: Unix shell



```
$ sort file1 | uniq > file2
```

  - Sort file, filter unique words and output. Pipe to wc to count words

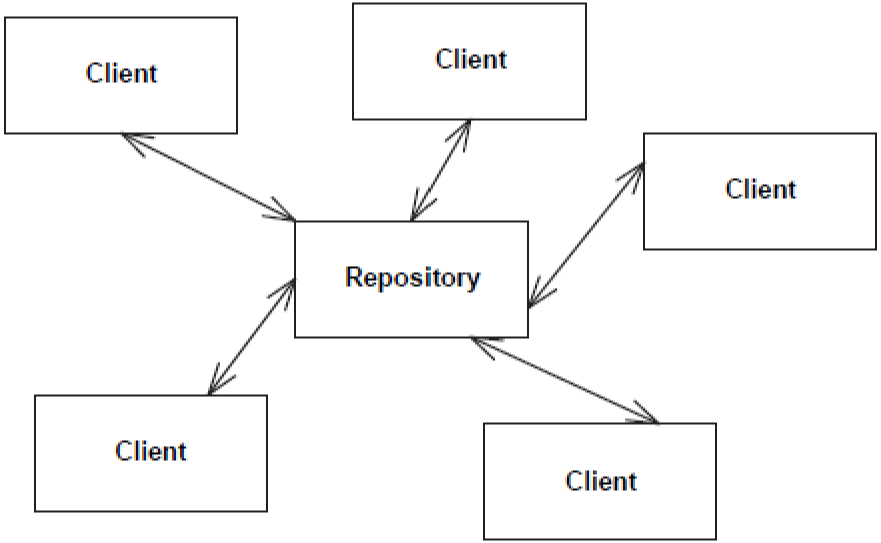
Main QA: Modifiability, Interoperability, Reusability

# Shared Repository

- Repository is *passive*

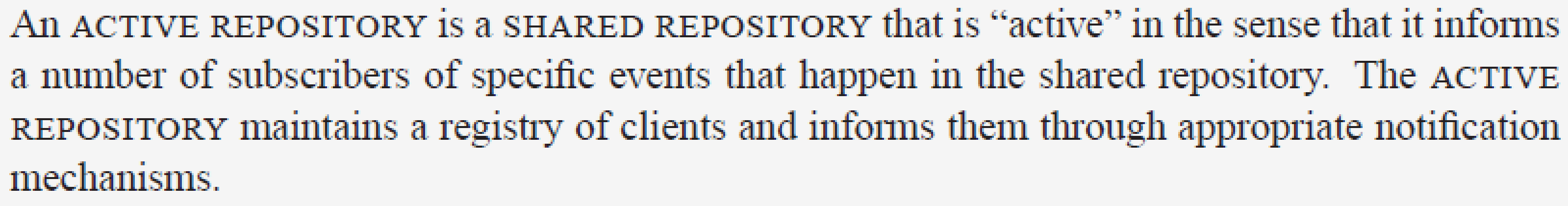
- Examples
  - Version control (git/svn)
  - The database !

- Most 3-tier Information Systems have this (sub)pattern



Main QA: Scalability*, Interoperability, (Consistency)

# Active Repository

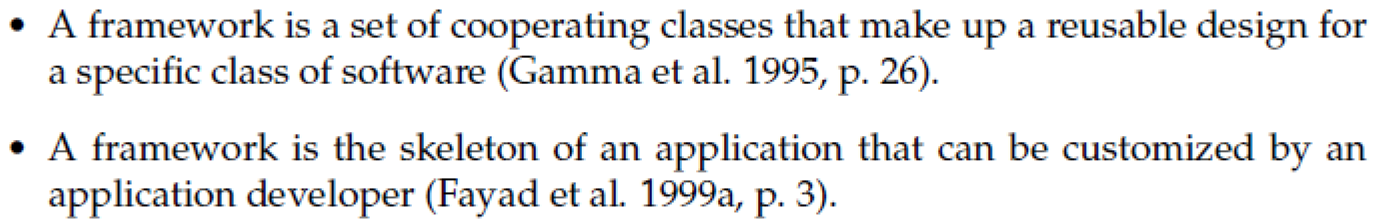


An ACTIVE REPOSITORY is a SHARED REPOSITORY that is "active" in the sense that it informs a number of subscribers of specific events that happen in the shared repository. The ACTIVE REPOSITORY maintains a registry of clients and informs them through appropriate notification mechanisms.

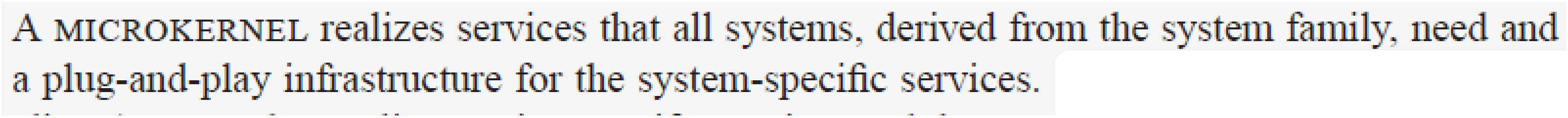
- Shared repository – now with events ☺
  - Notifies subscribers when things change in the repo
  - Architectural equivalent of the **Observer** pattern
    - Subject/Observable and Observer roles are processes

# Microkernel

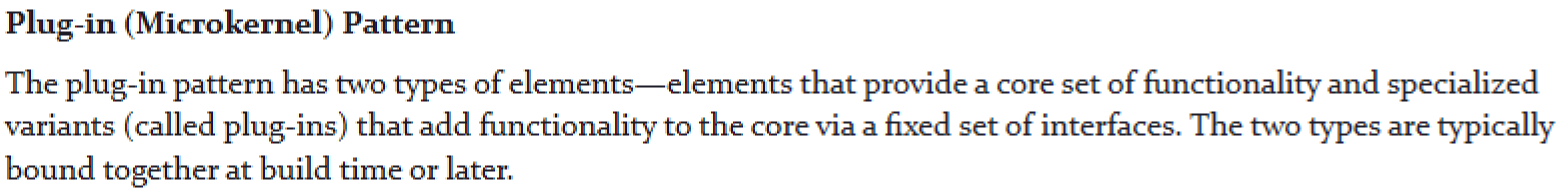
- Also known as/akin to **Framework**
  -



- A framework is a set of cooperating classes that make up a reusable design for a specific class of software (Gamma et al. 1995, p. 26).
- A framework is the skeleton of an application that can be customized by an application developer (Fayad et al. 1999a, p. 3).



A MICROKERNEL realizes services that all systems, derived from the system family, need and a plug-and-play infrastructure for the system-specific services.



**Plug-in (Microkernel) Pattern**

The plug-in pattern has two types of elements—elements that provide a core set of functionality and specialized variants (called plug-ins) that add functionality to the core via a fixed set of interfaces. The two types are typically bound together at build time or later.

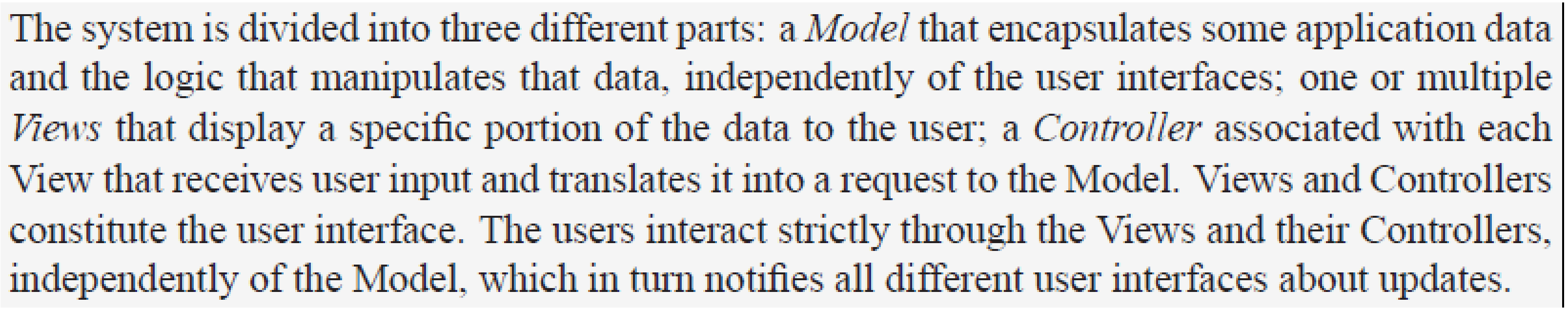
Main QA: Modifiability

Examples: IntelliJ + Eclipse; Gradle

# Microkernel

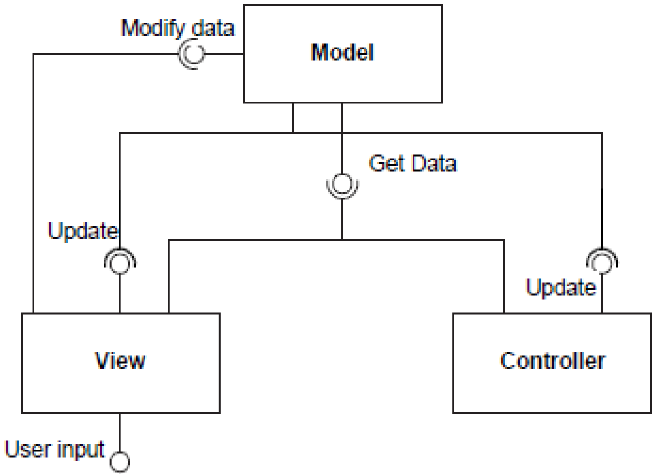
- Classically Frameworks are designed using
  - Program to an Interface + Favor object composition
    - Framework invokes methods defined in interfaces
      - Hiding the actual variant's implementation details
    - Strategy pattern = Specific algorithms for given variant
      - Not SELECT * FROM InventoryTable WHERE name='Bjarne'
      - But 'dbStrategy.fetchWithName("Bjarne")
        - » dbStrategy can be a in-memory hashMap, MongoDB, Redis, MariaDB, …
  - Dependency injection
    - To provide the framework with the concrete implementation
      - new PizzaOrdering( new MariaDBStrategy("ildpizza.inventory.dk", 3306) );

# Model-View-Controller



The system is divided into three different parts: a *Model* that encapsulates some application data and the logic that manipulates that data, independently of the user interfaces; one or multiple *Views* that display a specific portion of the data to the user; a *Controller* associated with each View that receives user input and translates it into a request to the Model. Views and Controllers constitute the user interface. The users interact strictly through the Views and their Controllers, independently of the Model, which in turn notifies all different user interfaces about updates.

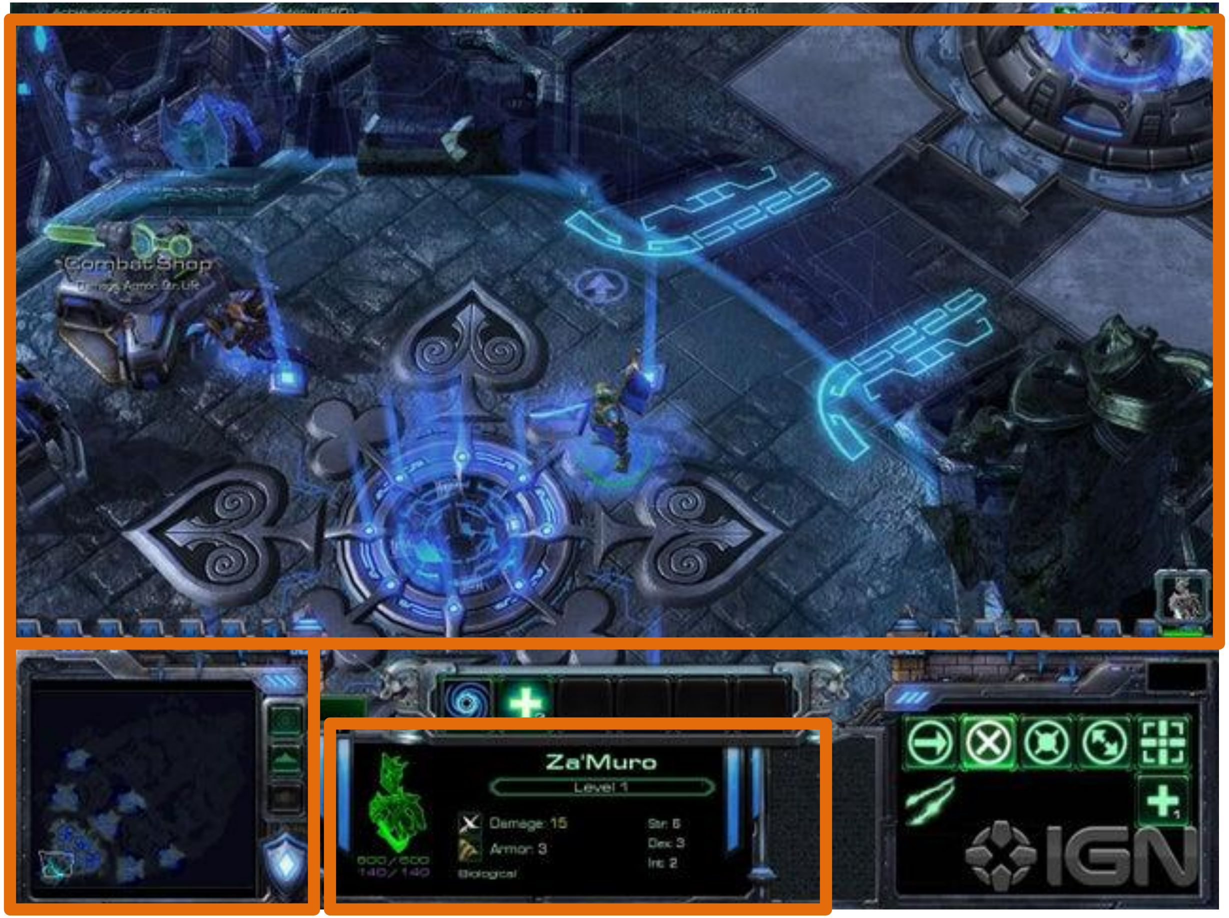
- Organizes graphical user interfaces and user interaction…
  - Basically a combo of two design patterns
    - **Observer**: notify views on state changes
    - **State**: set of controllers interpret UI events into proper model state changes



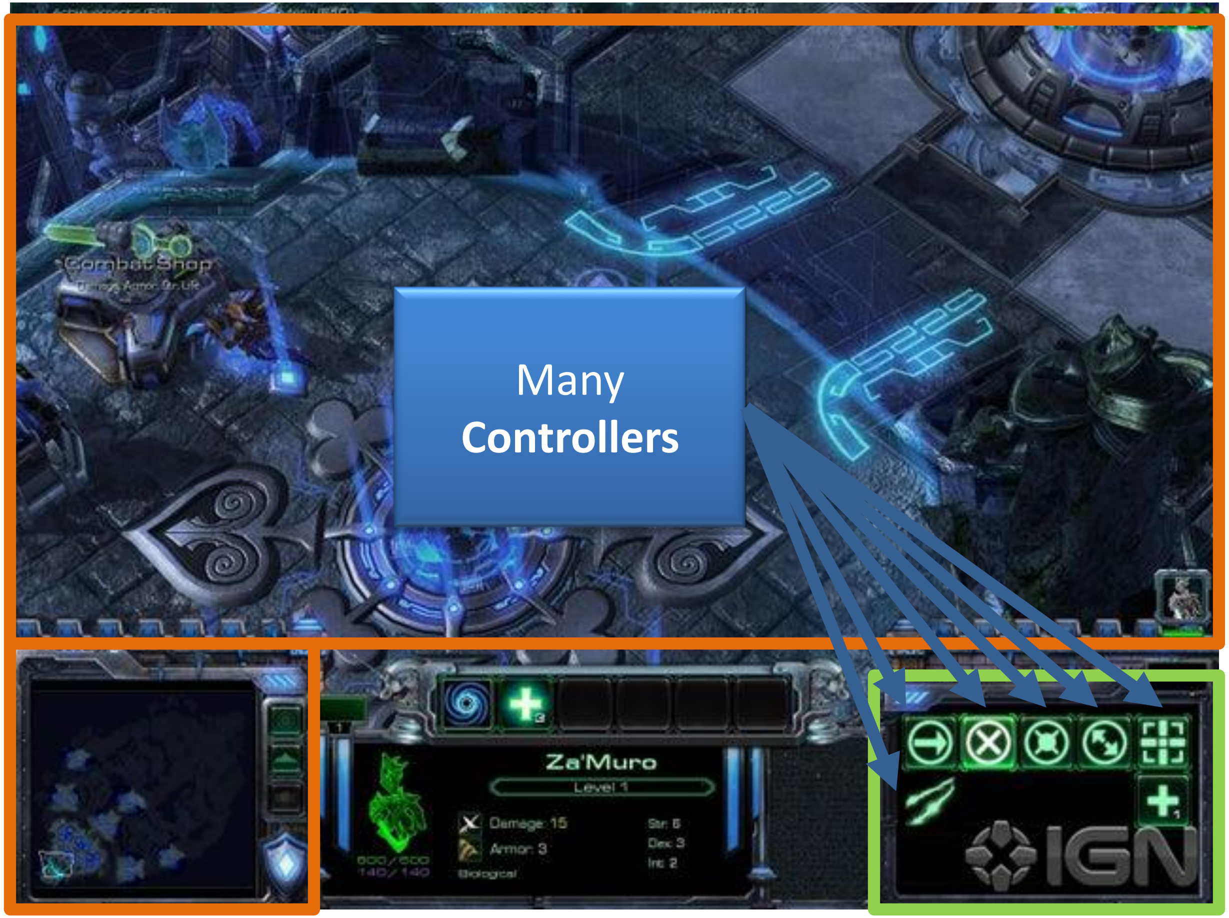
Main QA: Modifiability

# Visualizing MVC

Three **Views**



# Visualizing MVC



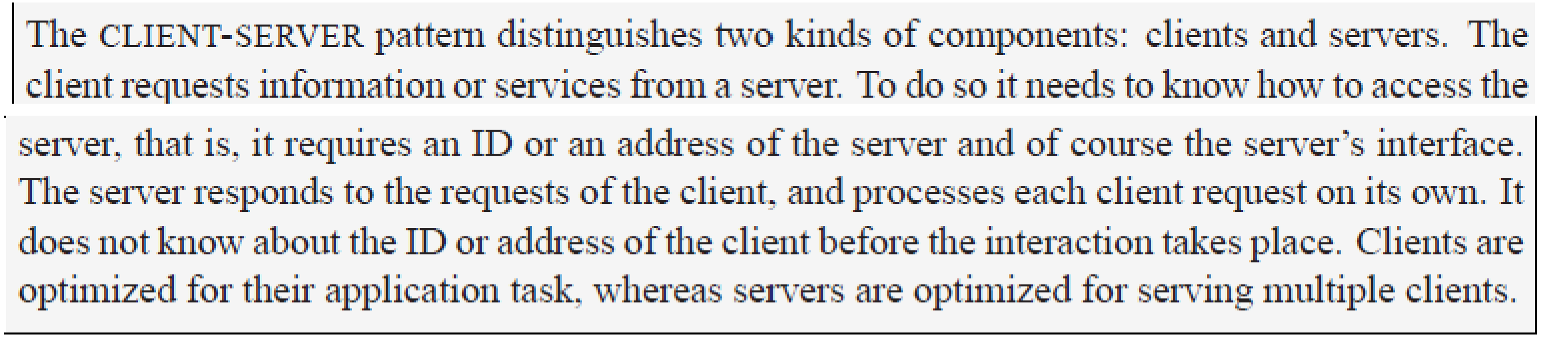
Many **Controllers**

# Variants

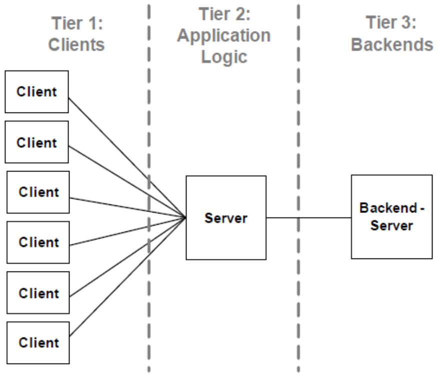
- Model-View-Presenter is a variant
  - But I have still not found a really good reference which tells me exactly what the difference MVP versus MVC is
    - ‘Presenter’ as an intermediary between Model and View (?)
  - Perhaps because I always code MVC rather like MVP
  - Anyway…

# Client-Server

- Should be well known ☺
  - WWW is basically a client-server system



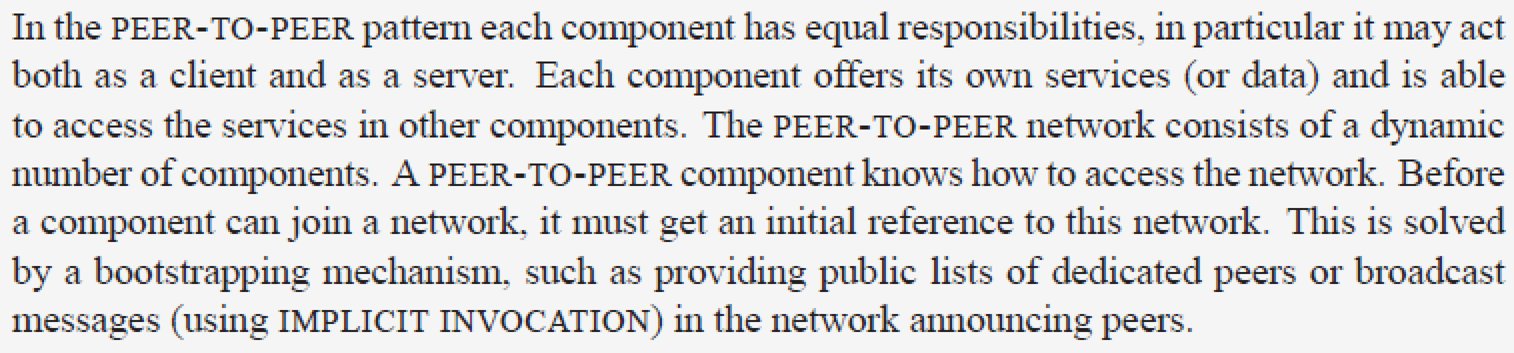
The CLIENT-SERVER pattern distinguishes two kinds of components: clients and servers. The client requests information or services from a server. To do so it needs to know how to access the server, that is, it requires an ID or an address of the server and of course the server's interface. The server responds to the requests of the client, and processes each client request on its own. It does not know about the ID or address of the client before the interaction takes place. Clients are optimized for their application task, whereas servers are optimized for serving multiple clients.



Main QA: Modifiability, Integrability, (Consistency)

# P2P

- Make all clients into servers, and servers into clients ☺



In the PEER-TO-PEER pattern each component has equal responsibilities, in particular it may act both as a client and as a server. Each component offers its own services (or data) and is able to access the services in other components. The PEER-TO-PEER network consists of a dynamic number of components. A PEER-TO-PEER component knows how to access the network. Before a component can join a network, it must get an initial reference to this network. This is solved by a bootstrapping mechanism, such as providing public lists of dedicated peers or broadcast messages (using IMPLICIT INVOCATION) in the network announcing peers.

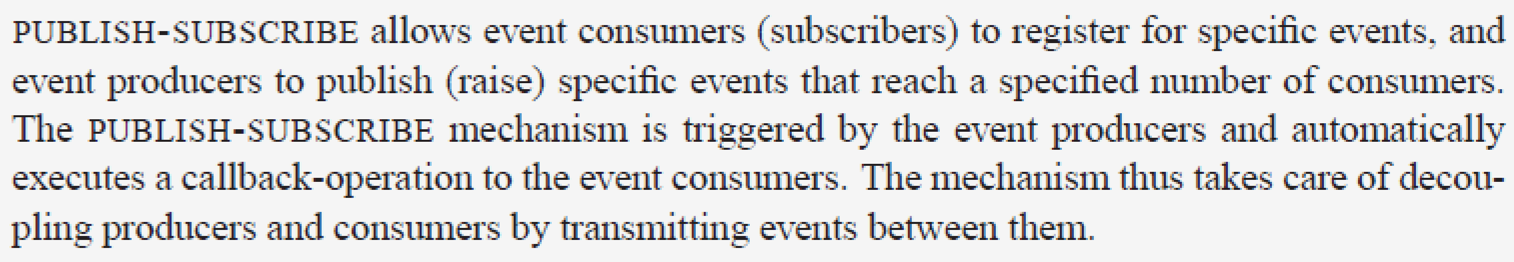
Main QA: Performance, Security (Privacy), Availability

Examples: BitTorrent

# Discussion

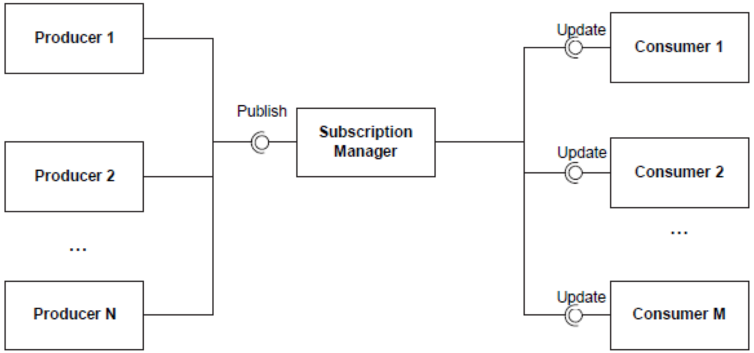
- Many client-server systems on WWW are actually not really client-server but a hybrid with P2P flavors
- Client-Server is strictly
  - Client is active Initiate *all* requests
  - Server is reactive *Only* respond to request
- That is: **Servers never call clients directly!!!**
- Has a profound performance/scalability quality
  - Servers can be stateless
    - Will not story any client information, will not call back
      - Callback to 100.000 clients is *slow!*

# Publish-Subscribe



PUBLISH-SUBSCRIBE allows event consumers (subscribers) to register for specific events, and event producers to publish (raise) specific events that reach a specified number of consumers. The PUBLISH-SUBSCRIBE mechanism is triggered by the event producers and automatically executes a callback-operation to the event consumers. The mechanism thus takes care of decoupling producers and consumers by transmitting events between them.

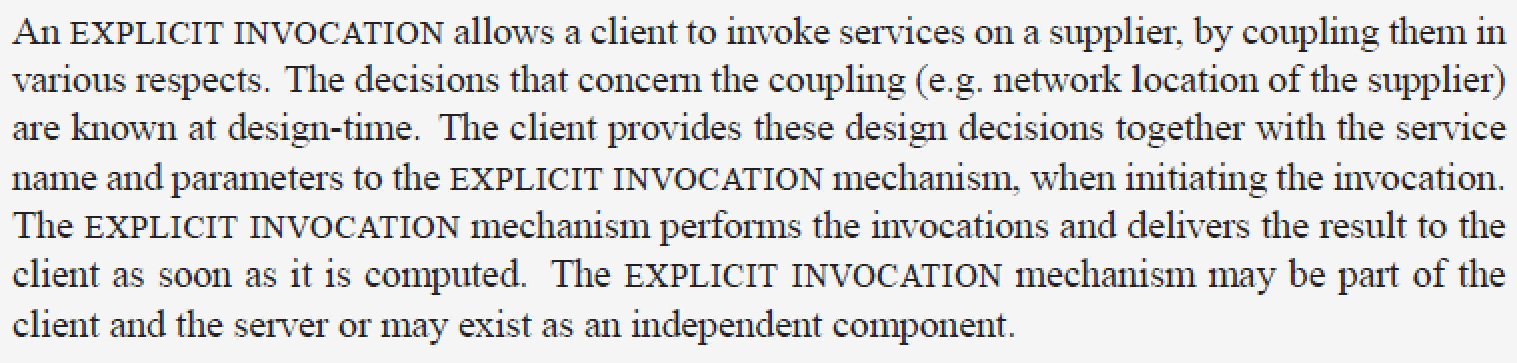
- Aka as 'Event Systems'
  - A Desktop UI is one such architecture
    - I register my app to listen to mouse events in the Windows Message Queue
  - MessageQueue system is the out-of-process variant
    - Example: RabbitMQ



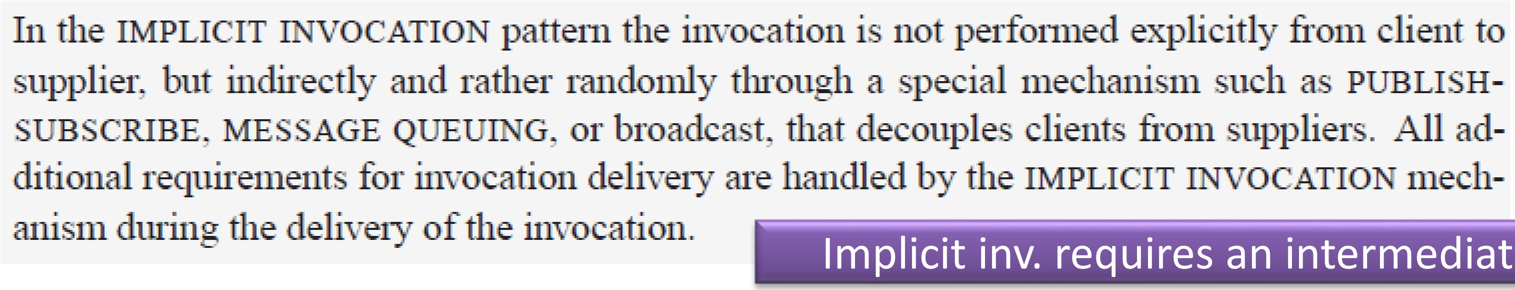
Main QA: Performance, Availability, Interoperability

# Invocation Patterns

- Explicit = I call you (= I know you)
- Implicit = You react on event (= I triggered event)



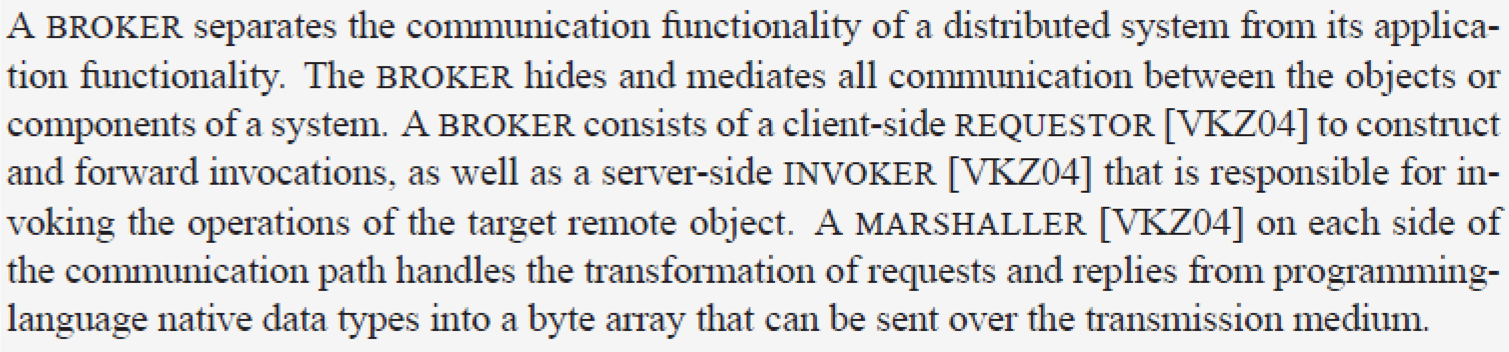
An EXPLICIT INVOCATION allows a client to invoke services on a supplier, by coupling them in various respects. The decisions that concern the coupling (e.g. network location of the supplier) are known at design-time. The client provides these design decisions together with the service name and parameters to the EXPLICIT INVOCATION mechanism, when initiating the invocation. The EXPLICIT INVOCATION mechanism performs the invocations and delivers the result to the client as soon as it is computed. The EXPLICIT INVOCATION mechanism may be part of the client and the server or may exist as an independent component.



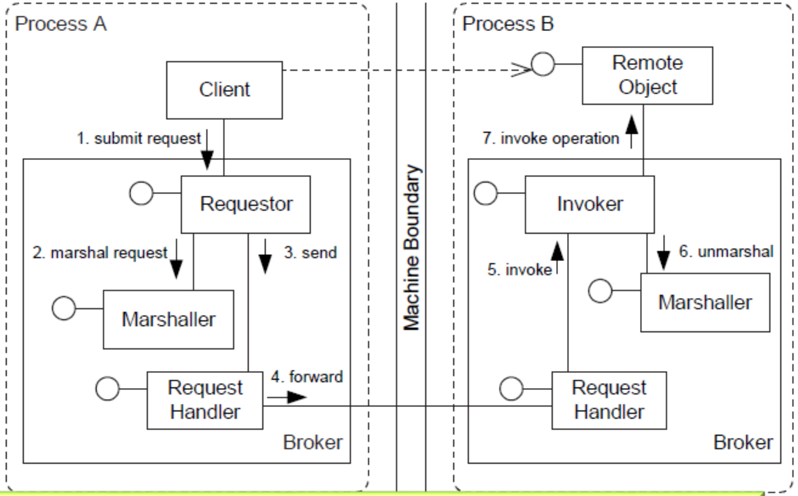
In the IMPLICIT INVOCATION pattern the invocation is not performed explicitly from client to supplier, but indirectly and rather randomly through a special mechanism such as PUBLISH-SUBSCRIBE, MESSAGE QUEUING, or broadcast, that decouples clients from suppliers. All additional requirements for invocation delivery are handled by the IMPLICIT INVOCATION mechanism during the delivery of the invocation.

Implicit inv. requires an intermediate

# Broker



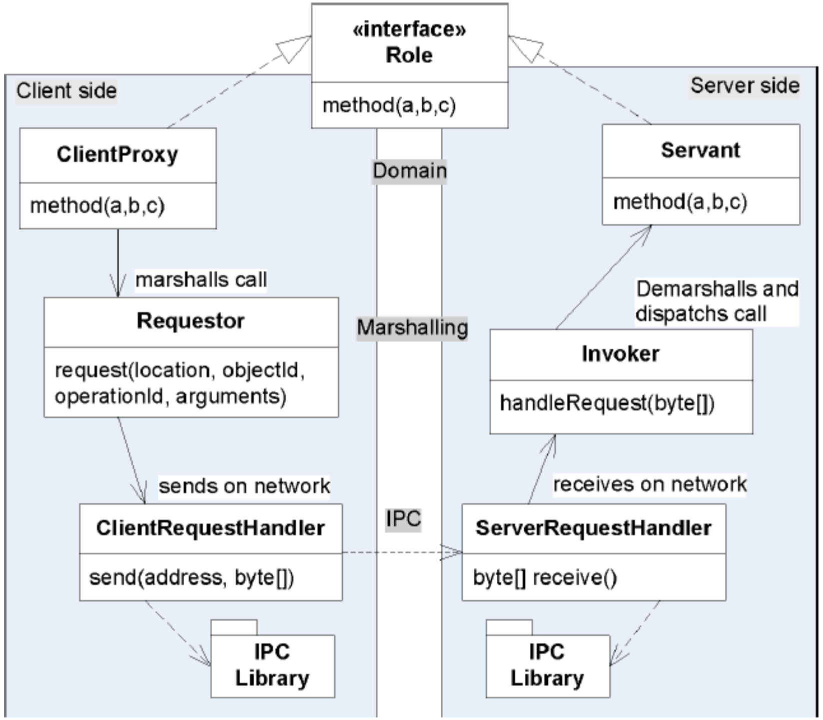
A BROKER separates the communication functionality of a distributed system from its application functionality. The BROKER hides and mediates all communication between the objects or components of a system. A BROKER consists of a client-side REQUESTOR [VKZ04] to construct and forward invocations, as well as a server-side INVOKER [VKZ04] that is responsible for invoking the operations of the target remote object. A MARSHALLER [VKZ04] on each side of the communication path handles the transformation of requests and replies from programming-language native data types into a byte array that can be sent over the transmission medium.



Main QA: Interoperability, Modifiability, Testability

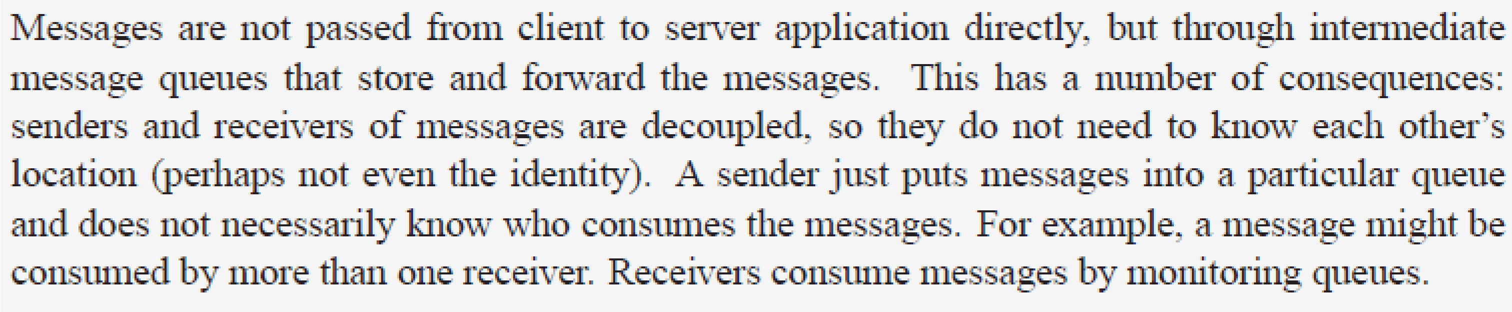
# Variants

- Often, patterns exists in *variants*
- Ex
  - Java RMI
    - Allows *servers to call client methods*
      - *Not a client-server*
  - FRDS Broker
    - Purely client-server
      - No Callback!



# Message Queues

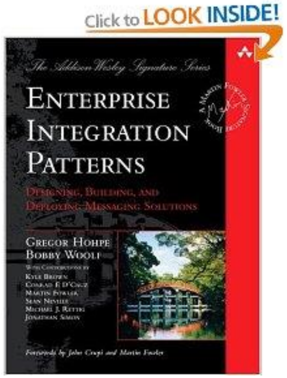
- Also known as Messaging



Messages are not passed from client to server application directly, but through intermediate message queues that store and forward the messages. This has a number of consequences: senders and receivers of messages are decoupled, so they do not need to know each other's location (perhaps not even the identity). A sender just puts messages into a particular queue and does not necessarily know who consumes the messages. For example, a message might be consumed by more than one receiver. Receivers consume messages by monitoring queues.

- “The ‘out-of-process’ Pub-Sub pattern”

- There is a full book on all its subpatterns!

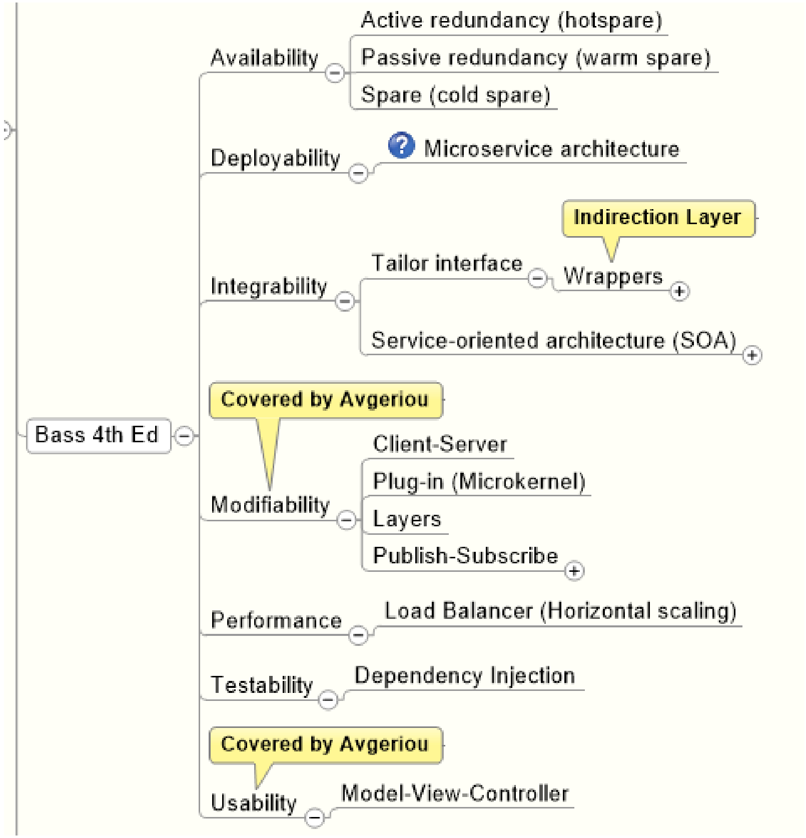

Click to LOOK INSIDE!
The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf

Main QA: Interoperability, Modifiability, Availability, Performance

# Bass et al.

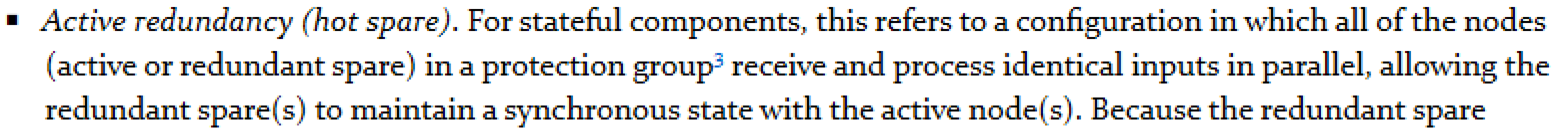
Some Additional Patterns

# CheatSheet


Bass 4th Ed
Availability
Active redundancy (hotspare)
Passive redundancy (warm spare)
Spare (cold spare)
Deployability
Microservice architecture
Integrability
Indirection Layer
Tailor interface
Wrappers
Service-oriented architecture (SOA)
Covered by Avgeriou
Modifiability
Client-Server
Plug-in (Microkernel)
Layers
Publish-Subscribe
Performance
Load Balancer (Horizontal scaling)
Testability
Dependency Injection
Covered by Avgeriou
Usability
Model-View-Controller

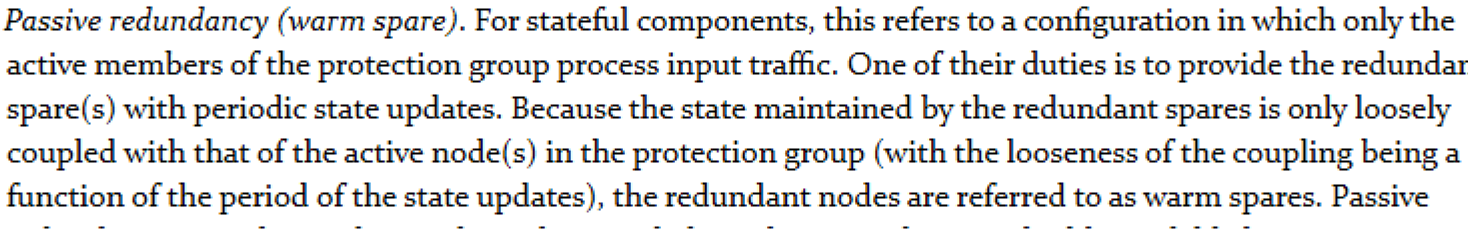
# Active/Passive/Spare Redundancy

- Redundancy = Maintain multiple copies



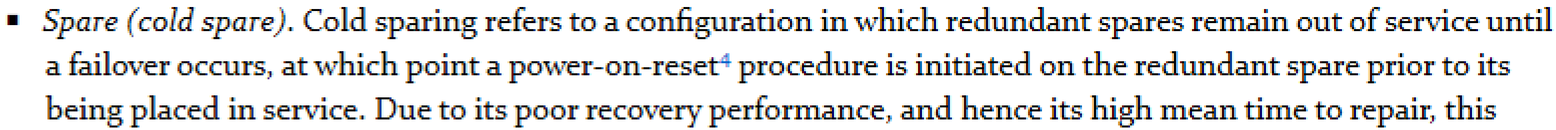
- *Active redundancy (hot spare)*. For stateful components, this refers to a configuration in which all of the nodes (active or redundant spare) in a protection group[3] receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain a synchronous state with the active node(s). Because the redundant spare

  - Millisecond scale



*Passive redundancy (warm spare)*. For stateful components, this refers to a configuration in which only the active members of the protection group process input traffic. One of their duties is to provide the redundant spare(s) with periodic state updates. Because the state maintained by the redundant spares is only loosely coupled with that of the active node(s) in the protection group (with the looseness of the coupling being a function of the period of the state updates), the redundant nodes are referred to as warm spares. Passive

  - Second-minute scale



- *Spare (cold spare)*. Cold sparing refers to a configuration in which redundant spares remain out of service until a failover occurs, at which point a power-on-reset[4] procedure is initiated on the redundant spare prior to its being placed in service. Due to its poor recovery performance, and hence its high mean time to repair, this

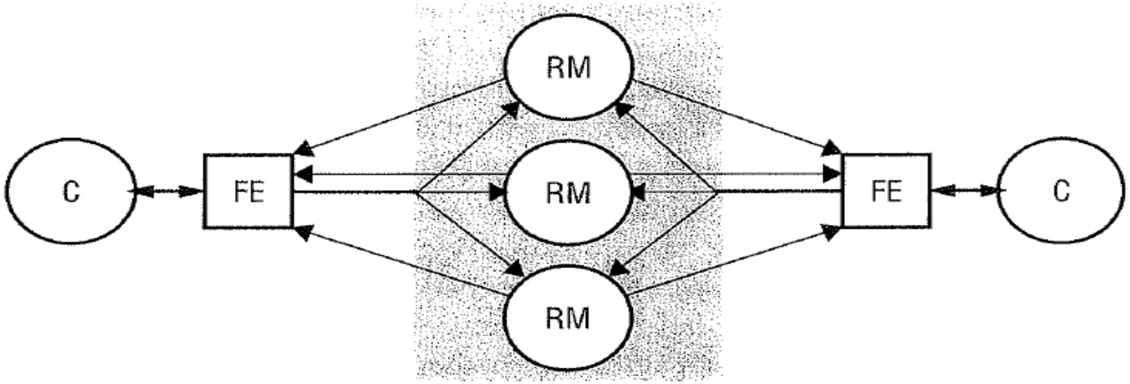
  - Hour scale

Main QA: Availability

# Active

- Active Redundancy
  - Clients *multicast* requests to all nodes
  - All nodes process identically but independently and reply
  - Front-End receives answers
    - May do one of several things: Use first one, compare, vote…

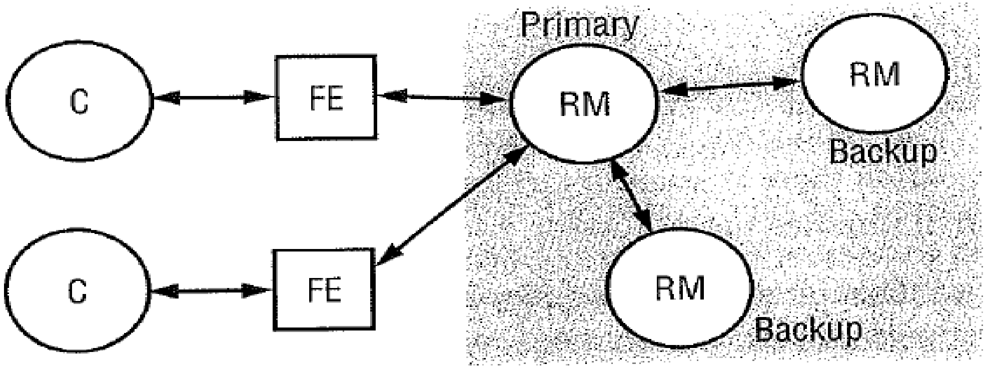
Active replication



# Passive

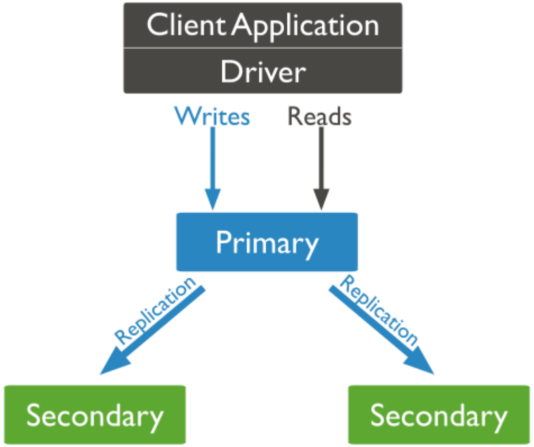
- Passive Redundancy
  - One node is *primary*
    - executes operations (notably writes/updates)
    - sends copies to slaves (in case of write/update)
  - Primary failure
    - One slave is promoted to become primary

The passive (primary-backup) model for fault tolerance



# Example

- MongoDB: **Replica Sets**
  - Built to run on commodity hardware
    - If master/primary fails, will promote slave to master
      - Typical timescale ~30 sec. *Write requests will throw exceptions meanwhile!*

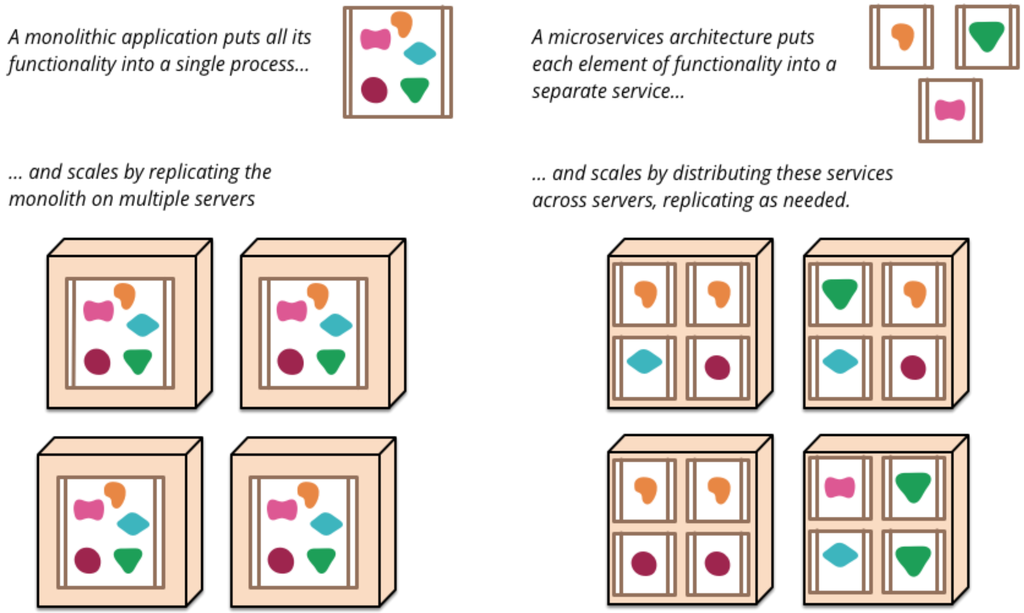


# Spare

- Rather low level ☺

# Microservice Architecture

- Not curriculum as I have a fagpakke on that…


A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

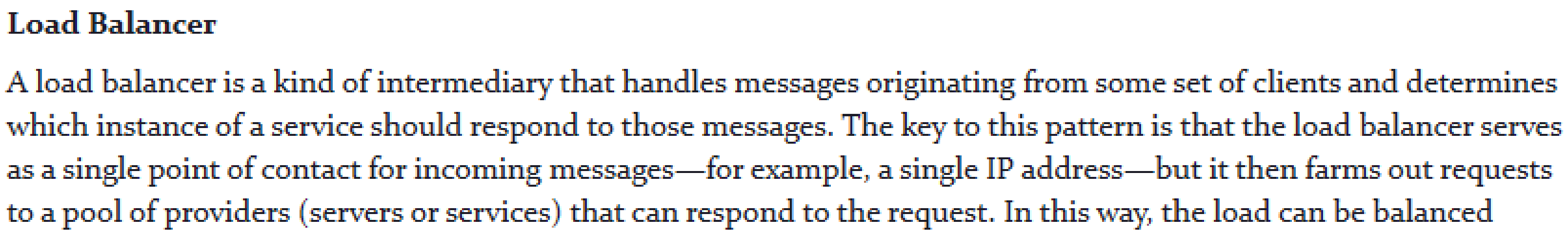
Main QA: Deployability, Availability

# Service Oriented Architecture

- Wikipedia
  - A *service-oriented architecture (SOA)* is an architectural pattern in computer software design in which application components provide *services* to other components via a *communications protocol, typically over a network*. The principles of service-orientation are independent of any vendor, product or technology.
- MacKenzie et al., 2006
  - is a paradigm for organizing and utilizing *distributed capabilities* that may be under the *control of different ownership domains*
  - provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations

Main QA: Integrability

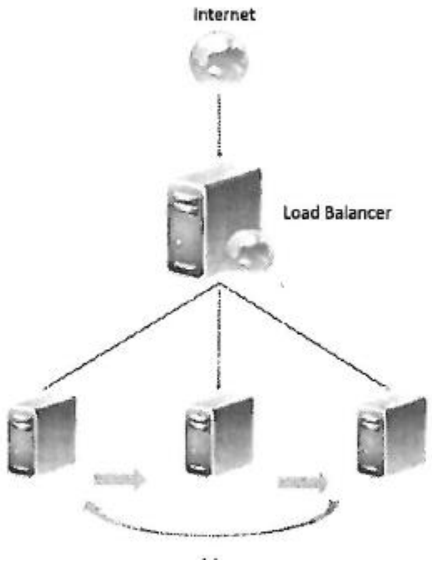
# Load Balancer



### Load Balancer

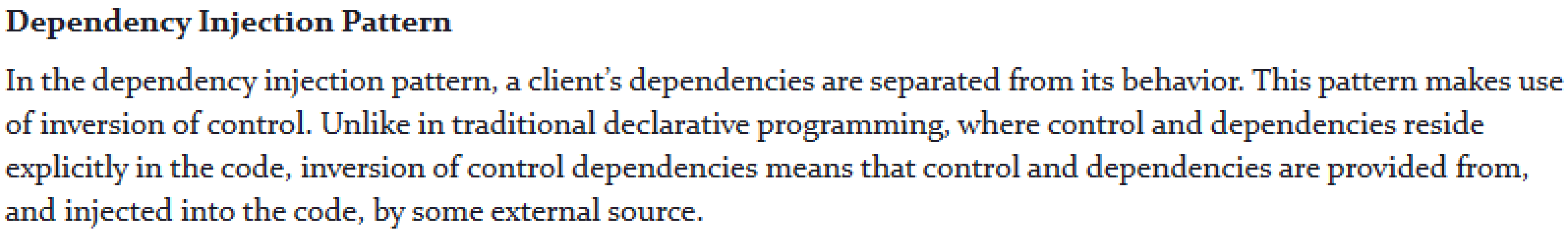
A load balancer is a kind of intermediary that handles messages originating from some set of clients and determines which instance of a service should respond to those messages. The key to this pattern is that the load balancer serves as a single point of contact for incoming messages—for example, a single IP address—but it then farms out requests to a pool of providers (servers or services) that can respond to the request. In this way, the load can be balanced

- A set of machines ‘becomes one’
  - Only viable if the server is designed to be stateless!
    - All state must be held in a storage tier…



Main QA: Performance, Availability

# Dependency Injection



**Dependency Injection Pattern**

In the dependency injection pattern, a client's dependencies are separated from its behavior. This pattern makes use of inversion of control. Unlike in traditional declarative programming, where control and dependencies reside explicitly in the code, inversion of control dependencies means that control and dependencies are provided from, and injected into the code, by some external source.

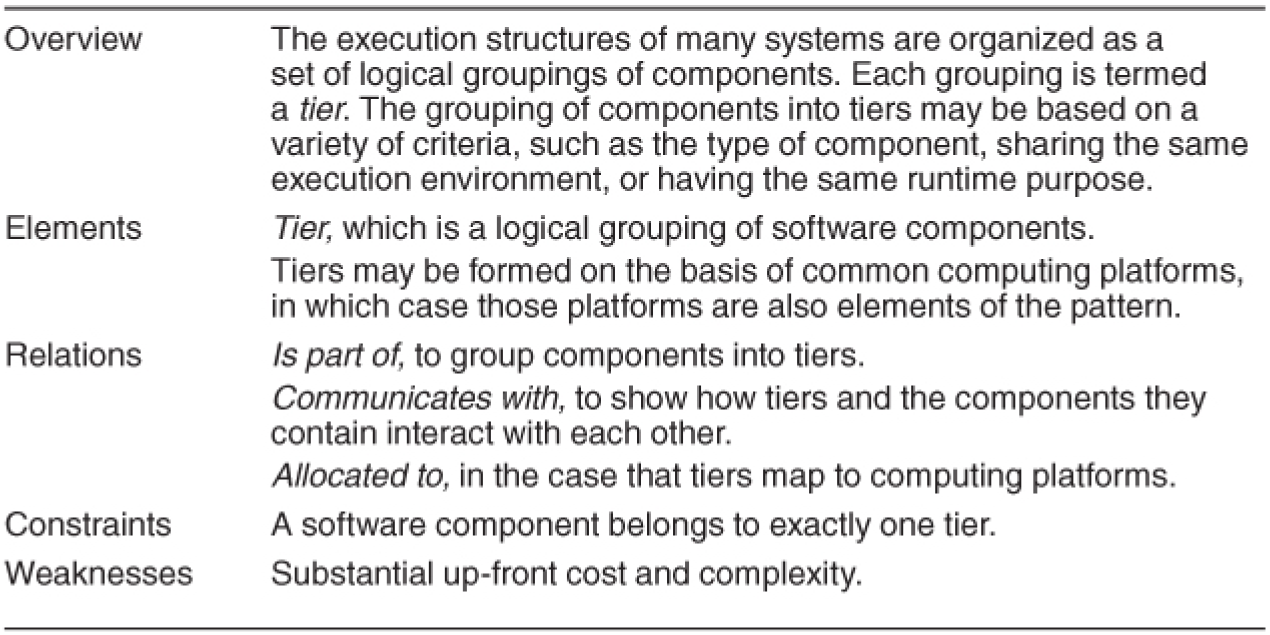
- Adhere to the SOLID principles
  - Or Bærbak's ③-①-② principles
    - Encapsulate what varies (the problematic aspect wrt testing)
    - Program to an interface
    - Favor object composition
  - And then use **test doubles** to replace the real depended on unit

Main QA: Testability

# Multi-Tier

- Basically – Layer pattern in the form of *deployment units*

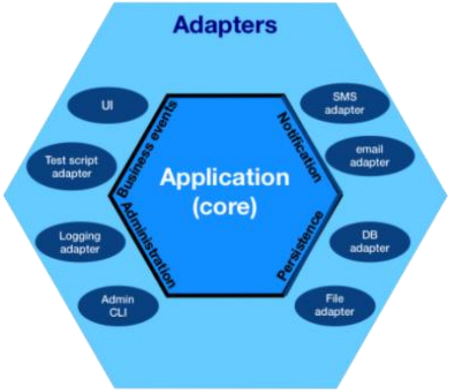
**Table 13.11. Multi-tier Pattern Solution**



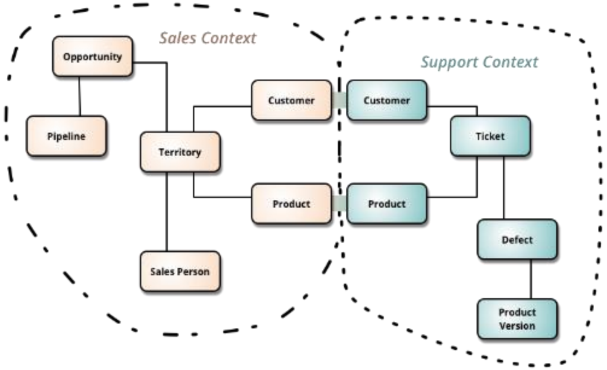
| | |
|---|---|
| Overview | The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a *tier*. The grouping of components into tiers may be based on a variety of criteria, such as the type of component, sharing the same execution environment, or having the same runtime purpose. |
| Elements | *Tier*, which is a logical grouping of software components.<br>Tiers may be formed on the basis of common computing platforms, in which case those platforms are also elements of the pattern. |
| Relations | *Is part of*, to group components into tiers.<br>*Communicates with*, to show how tiers and the components they contain interact with each other.<br>*Allocated to*, in the case that tiers map to computing platforms. |
| Constraints | A software component belongs to exactly one tier. |
| Weaknesses | Substantial up-front cost and complexity. |

# But there are Others…

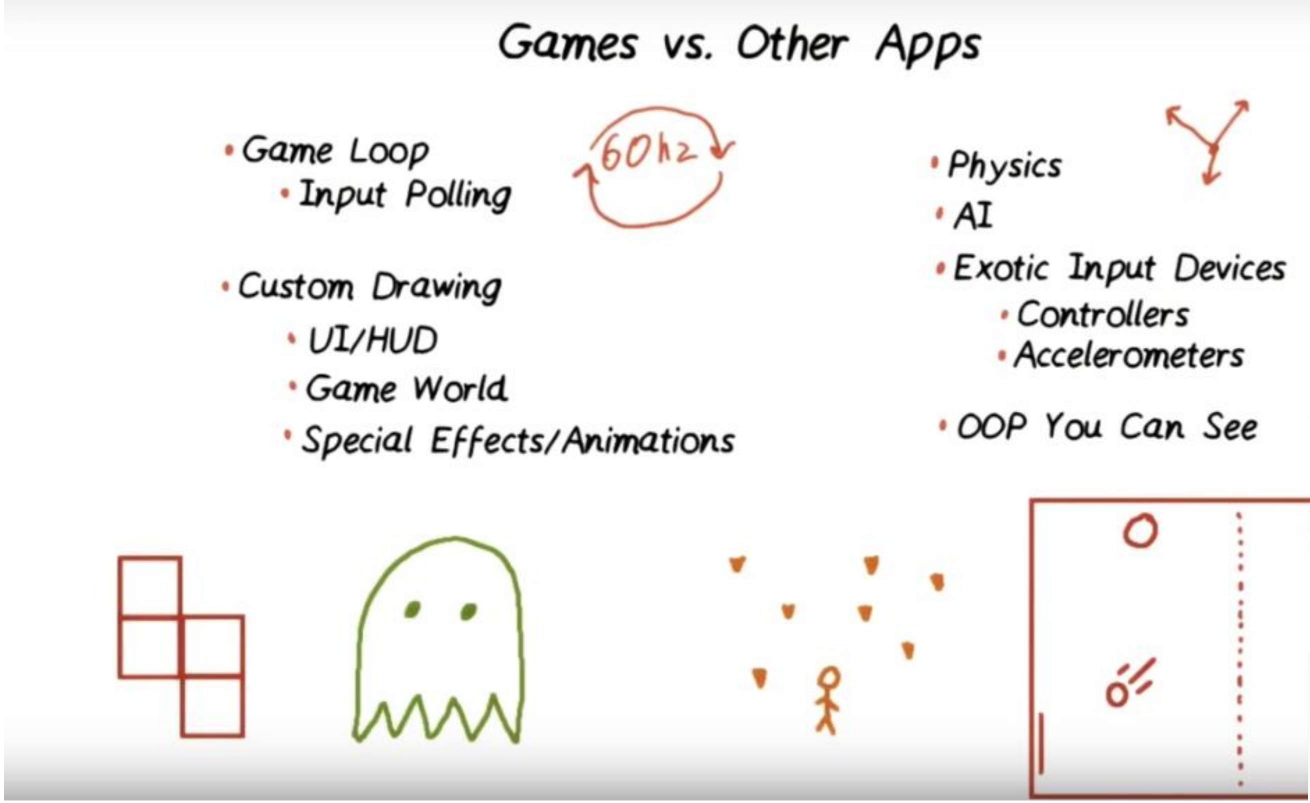
- Hexagonal Architecture
  - Hm hm, Is it not just *program to an interface?*



- Bounded Contexts

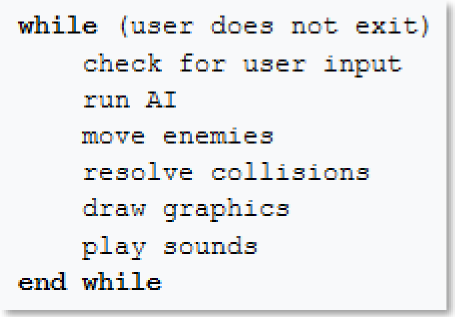


# Game Engine



## Games vs. Other Apps

- Game Loop
    - Input Polling
- Custom Drawing
    - UI/HUD
    - Game World
    - Special Effects/Animations
- Physics
- AI
- Exotic Input Devices
    - Controllers
    - Accelerometers
- OOP You Can See



```
while (user does not exit)
     check for user input
     run AI
     move enemies
     resolve collisions
     draw graphics
     play sounds
end while
```

# Discussion

- Usually you mix and match a large set of patterns for any given system
  - TeleMed
    - Client-server, Shared Repository, (multi-tier), Broker, (Layered), …
  - WoW
    - Peer-to-peer, Client-server, Shared Repository,

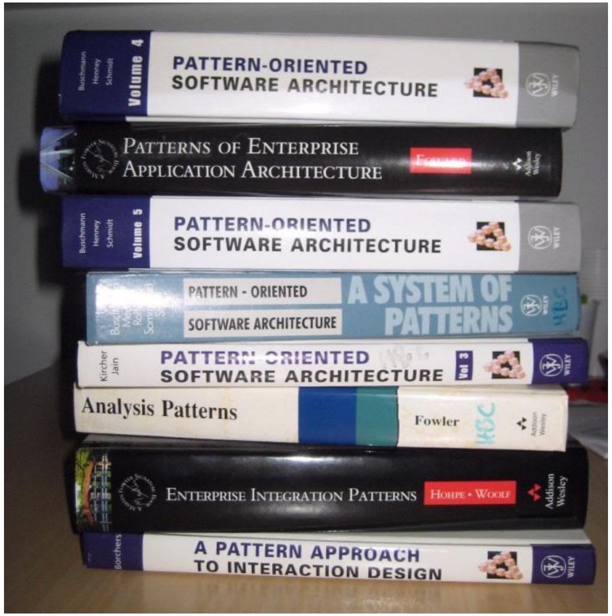
# Phew…

AARHUS UNIVERSITET

# Outlook

# Sources!

- *Lots* of books on architectural patterns
  - ☺ Much to choose from...
  - ☹ Can’t see the wood for all the trees...

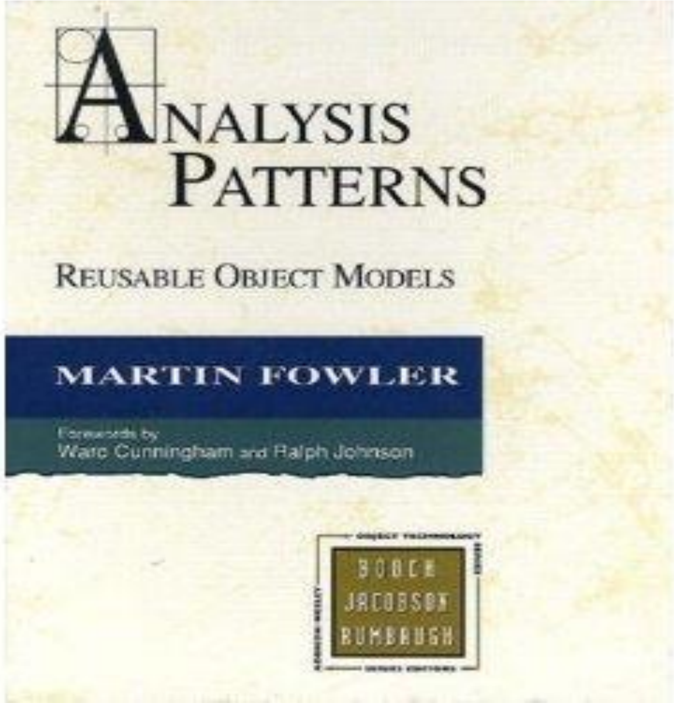

PATTERN-ORIENTED SOFTWARE ARCHITECTURE Volume 4
PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE
PATTERN-ORIENTED SOFTWARE ARCHITECTURE Volume 5
PATTERN - ORIENTED SOFTWARE ARCHITECTURE A SYSTEM OF PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE Vol 3
Analysis Patterns Fowler
ENTERPRISE INTEGRATION PATTERNS HOHPE • WOOLF
A PATTERN APPROACH TO INTERACTION DESIGN

# Breath and Detail

Fast forward in a few central books

# Fowler: Analysis patterns

- A pretty old book (1996), but has some merits...



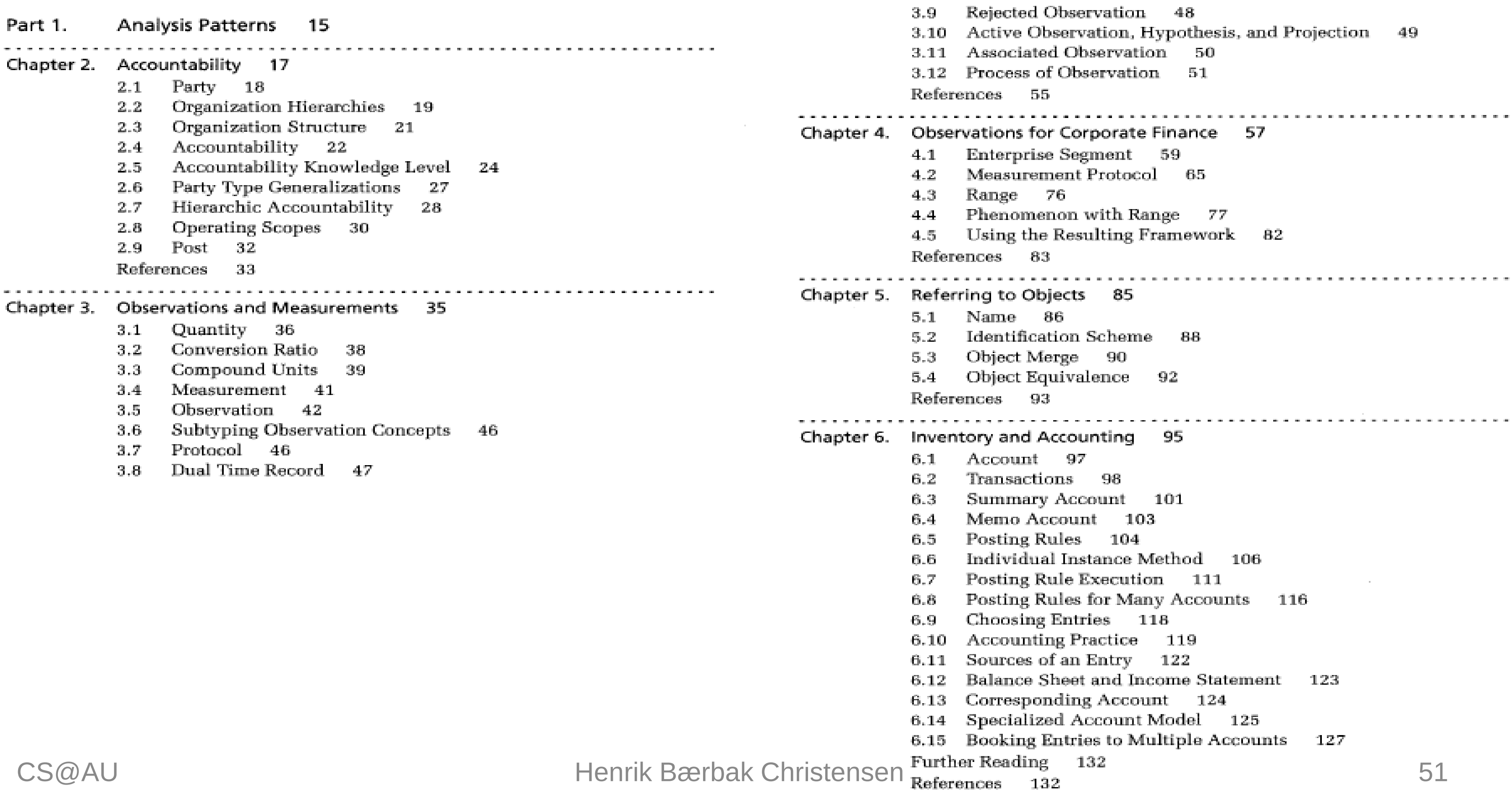
ANALYSIS PATTERNS

REUSABLE OBJECT MODELS

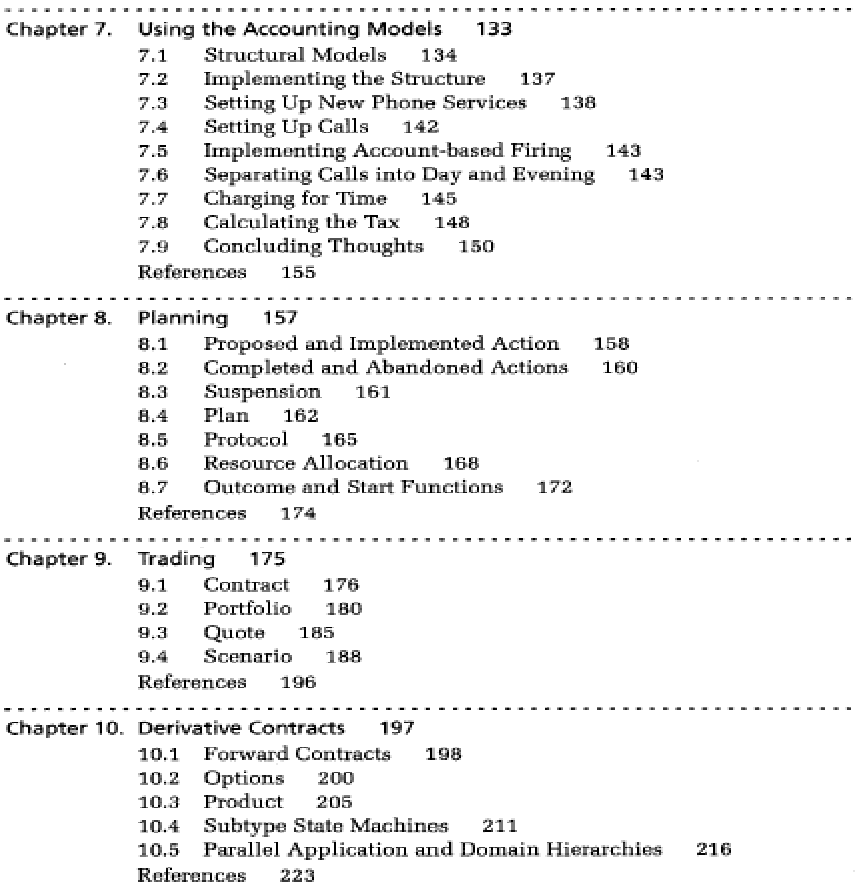
MARTIN FOWLER

Forewords by
Ward Cunningham and Ralph Johnson

# Contents

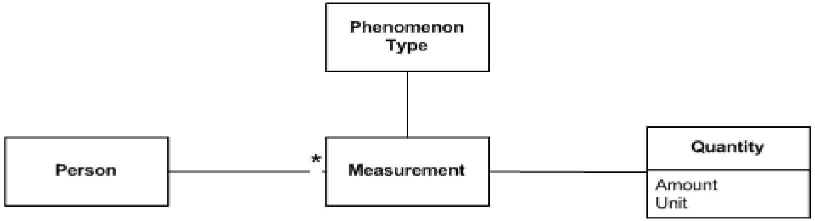


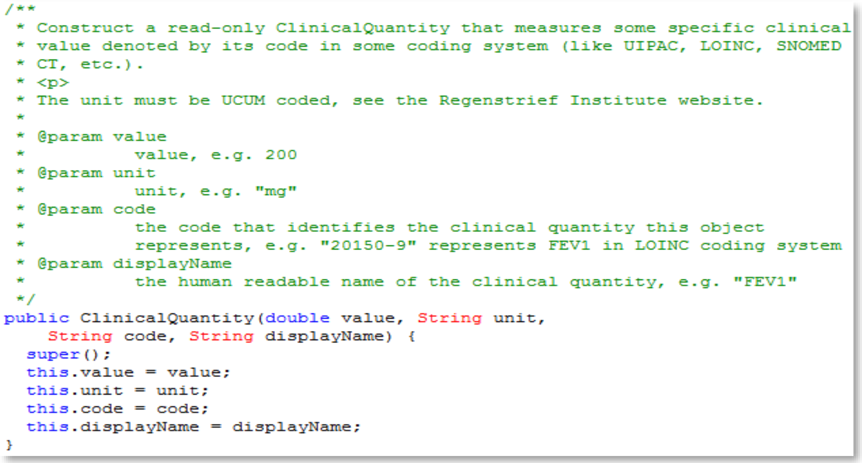
# Contents



# Example

- ”Measurement”
  - In Net4Care/TeleMed we ’reused’ measurements from HL7 which is identical to Fowler’s Measurement pattern

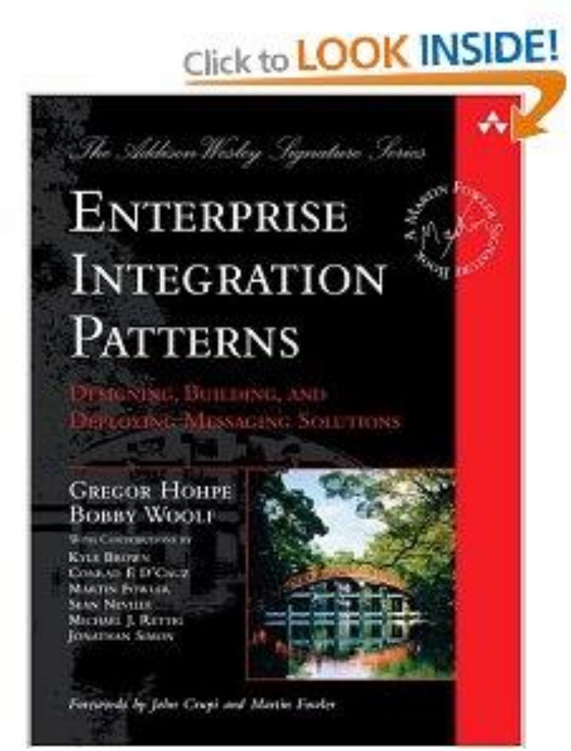




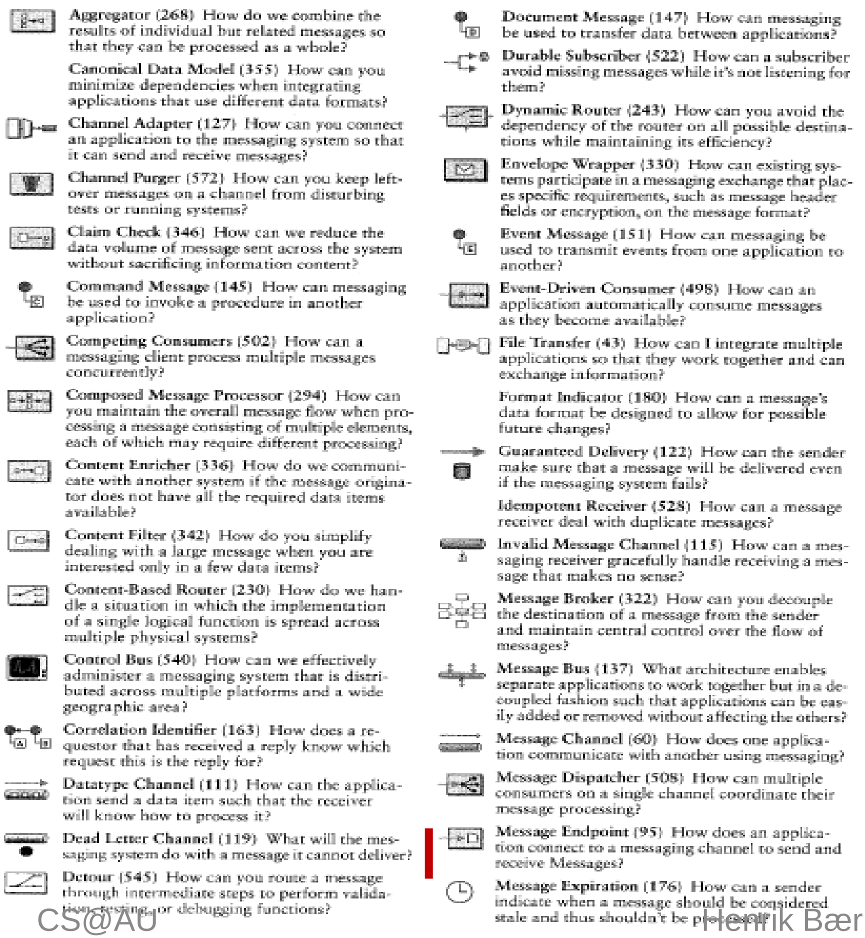
```
/**
 * Construct a read-only ClinicalQuantity that measures some specific clinical
 * value denoted by its code in some coding system (like UIPAC, LOINC, SNOMED
 * CT, etc.).
 * <p>
 * The unit must be UCUM coded, see the Regenstrief Institute website.
 *
 * @param value
 *          value, e.g. 200
 * @param unit
 *          unit, e.g. "mg"
 * @param code
 *          the code that identifies the clinical quantity this object
 *          represents, e.g. "20150-9" represents FEV1 in LOINC coding system
 * @param displayName
 *          the human readable name of the clinical quantity, e.g. "FEV1"
 */
public ClinicalQuantity(double value, String unit,
    String code, String displayName) {
  super();
  this.value = value;
  this.unit = unit;
  this.code = code;
  this.displayName = displayName;
}
```

# Hohpe & Woolf: Enterprise Int. P.

- From 2004
- **Messaging** (MQ)
- Enterprise Service Bus


Click to LOOK INSIDE!
The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler
A Martin Fowler Signature Book

# Contents



**Aggregator** (268) How do we combine the results of individual but related messages so that they can be processed as a whole?

**Canonical Data Model** (355) How can you minimize dependencies when integrating applications that use different data formats?

**Channel Adapter** (127) How can you connect an application to the messaging system so that it can send and receive messages?

**Channel Purger** (572) How can you keep left-over messages on a channel from disturbing tests or running systems?

**Claim Check** (346) How can we reduce the data volume of message sent across the system without sacrificing information content?

**Command Message** (145) How can messaging be used to invoke a procedure in another application?

**Competing Consumers** (502) How can a messaging client process multiple messages concurrently?

**Composed Message Processor** (294) How can you maintain the overall message flow when processing a message consisting of multiple elements, each of which may require different processing?

**Content Enricher** (336) How do we communicate with another system if the message originator does not have all the required data items available?

**Content Filter** (342) How do you simplify dealing with a large message when you are interested only in a few data items?

**Content-Based Router** (230) How do we handle a situation in which the implementation of a single logical function is spread across multiple physical systems?

**Control Bus** (540) How can we effectively administer a messaging system that is distributed across multiple platforms and a wide geographic area?

**Correlation Identifier** (163) How does a requestor that has received a reply know which request this is the reply for?

**Datatype Channel** (111) How can the application send a data item such that the receiver will know how to process it?

**Dead Letter Channel** (119) What will the messaging system do with a message it cannot deliver?

**Detour** (545) How can you route a message through intermediate steps to perform validation, testing, or debugging functions?

**Document Message** (147) How can messaging be used to transfer data between applications?

**Durable Subscriber** (522) How can a subscriber avoid missing messages while it's not listening for them?

**Dynamic Router** (243) How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency?

**Envelope Wrapper** (330) How can existing systems participate in a messaging exchange that places specific requirements, such as message header fields or encryption, on the message format?

**Event Message** (151) How can messaging be used to transmit events from one application to another?

**Event-Driven Consumer** (498) How can an application automatically consume messages as they become available?

**File Transfer** (43) How can I integrate multiple applications so that they work together and can exchange information?

**Format Indicator** (180) How can a message's data format be designed to allow for possible future changes?

**Guaranteed Delivery** (122) How can the sender make sure that a message will be delivered even if the messaging system fails?

**Idempotent Receiver** (528) How can a message receiver deal with duplicate messages?

**Invalid Message Channel** (115) How can a messaging receiver gracefully handle receiving a message that makes no sense?

**Message Broker** (322) How can you decouple the destination of a message from the sender and maintain central control over the flow of messages?

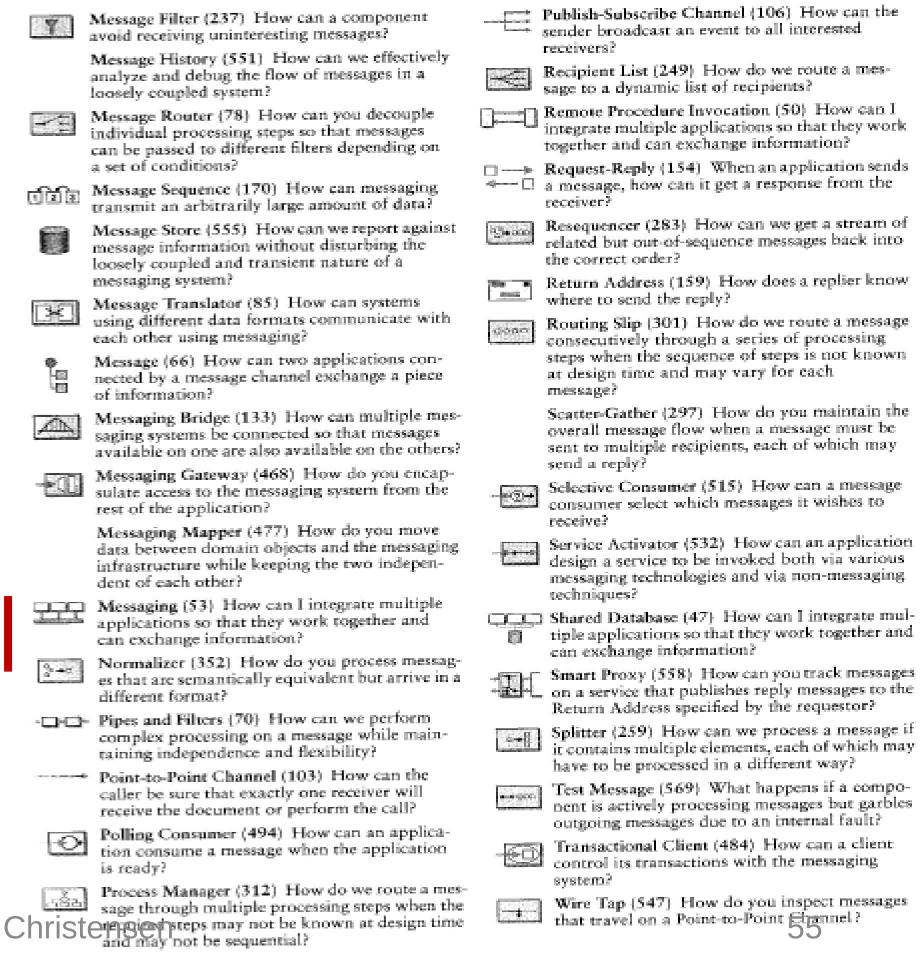
**Message Bus** (137) What architecture enables separate applications to work together but in a decoupled fashion such that applications can be easily added or removed without affecting the others?

**Message Channel** (60) How does one application communicate with another using messaging?

**Message Dispatcher** (508) How can multiple consumers on a single channel coordinate their message processing?

**Message Endpoint** (95) How does an application connect to a messaging channel to send and receive Messages?

**Message Expiration** (176) How can a sender indicate when a message should be considered stale and thus shouldn't be processed?



**Message Filter** (237) How can a component avoid receiving uninteresting messages?

**Message History** (551) How can we effectively analyze and debug the flow of messages in a loosely coupled system?

**Message Router** (78) How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?

**Message Sequence** (170) How can messaging transmit an arbitrarily large amount of data?

**Message Store** (555) How can we report against message information without disturbing the loosely coupled and transient nature of a messaging system?

**Message Translator** (85) How can systems using different data formats communicate with each other using messaging?

**Message** (66) How can two applications connected by a message channel exchange a piece of information?

**Messaging Bridge** (133) How can multiple messaging systems be connected so that messages available on one are also available on the others?

**Messaging Gateway** (468) How do you encapsulate access to the messaging system from the rest of the application?

**Messaging Mapper** (477) How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?

**Messaging** (53) How can I integrate multiple applications so that they work together and can exchange information?

**Normalizer** (352) How do you process messages that are semantically equivalent but arrive in a different format?

**Pipes and Filters** (70) How can we perform complex processing on a message while maintaining independence and flexibility?

**Point-to-Point Channel** (103) How can the caller be sure that exactly one receiver will receive the document or perform the call?

**Polling Consumer** (494) How can an application consume a message when the application is ready?

**Process Manager** (312) How do we route a message through multiple processing steps when the required steps may not be known at design time and may not be sequential?

**Publish-Subscribe Channel** (106) How can the sender broadcast an event to all interested receivers?

**Recipient List** (249) How do we route a message to a dynamic list of recipients?

**Remote Procedure Invocation** (50) How can I integrate multiple applications so that they work together and can exchange information?

**Request-Reply** (154) When an application sends a message, how can it get a response from the receiver?

**Resequencer** (283) How can we get a stream of related but out-of-sequence messages back into the correct order?

**Return Address** (159) How does a replier know where to send the reply?

**Routing Slip** (301) How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design time and may vary for each message?

**Scatter-Gather** (297) How do you maintain the overall message flow when a message must be sent to multiple recipients, each of which may send a reply?

**Selective Consumer** (515) How can a message consumer select which messages it wishes to receive?

**Service Activator** (532) How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?

**Shared Database** (47) How can I integrate multiple applications so that they work together and can exchange information?

**Smart Proxy** (558) How can you track messages on a service that publishes reply messages to the Return Address specified by the requestor?

**Splitter** (259) How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?

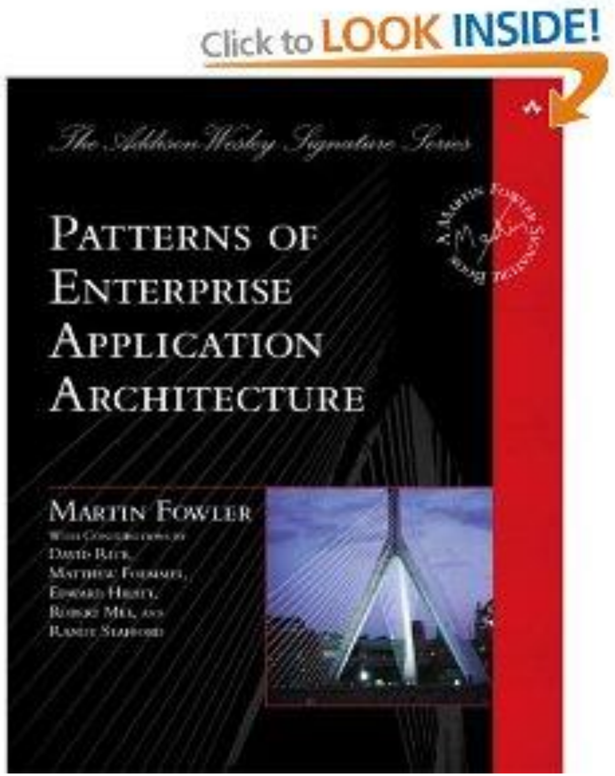
**Test Message** (569) What happens if a component is actively processing messages but garbles outgoing messages due to an internal fault?

**Transactional Client** (484) How can a client control its transactions with the messaging system?

**Wire Tap** (547) How do you inspect messages that travel on a Point-to-Point Channel?

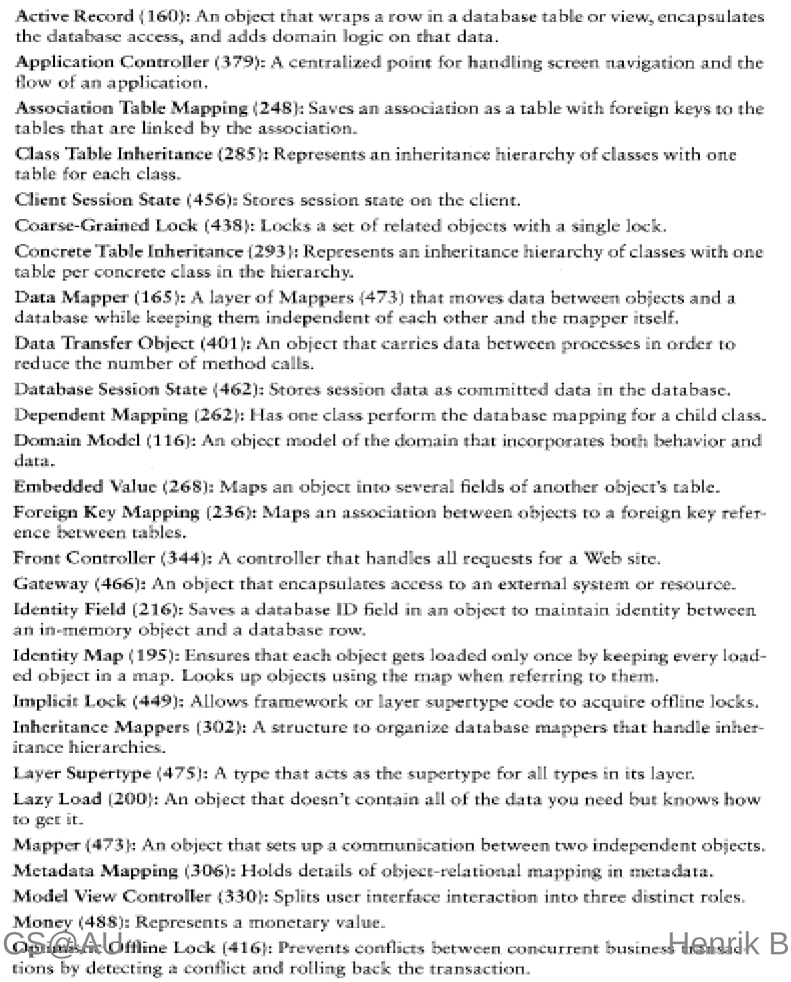
# Fowler: Enterprise Patterns

- From 2003
- Treats Info Systems
  - Emphasis on enterprise domain
  - Less on patterns for quality attributes



AARHUS UNIVERSITET

# Contents



**Active Record (160):** An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

**Application Controller (379):** A centralized point for handling screen navigation and the flow of an application.

**Association Table Mapping (248):** Saves an association as a table with foreign keys to the tables that are linked by the association.

**Class Table Inheritance (285):** Represents an inheritance hierarchy of classes with one table for each class.

**Client Session State (456):** Stores session state on the client.

**Coarse-Grained Lock (438):** Locks a set of related objects with a single lock.

**Concrete Table Inheritance (293):** Represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy.

**Data Mapper (165):** A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself.

**Data Transfer Object (401):** An object that carries data between processes in order to reduce the number of method calls.

**Database Session State (462):** Stores session data as committed data in the database.

**Dependent Mapping (262):** Has one class perform the database mapping for a child class.

**Domain Model (116):** An object model of the domain that incorporates both behavior and data.

**Embedded Value (268):** Maps an object into several fields of another object's table.

**Foreign Key Mapping (236):** Maps an association between objects to a foreign key reference between tables.

**Front Controller (344):** A controller that handles all requests for a Web site.

**Gateway (466):** An object that encapsulates access to an external system or resource.

**Identity Field (216):** Saves a database ID field in an object to maintain identity between an in-memory object and a database row.

**Identity Map (195):** Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them.

**Implicit Lock (449):** Allows framework or layer supertype code to acquire offline locks.

**Inheritance Mappers (302):** A structure to organize database mappers that handle inheritance hierarchies.

**Layer Supertype (475):** A type that acts as the supertype for all types in its layer.

**Lazy Load (200):** An object that doesn't contain all of the data you need but knows how to get it.

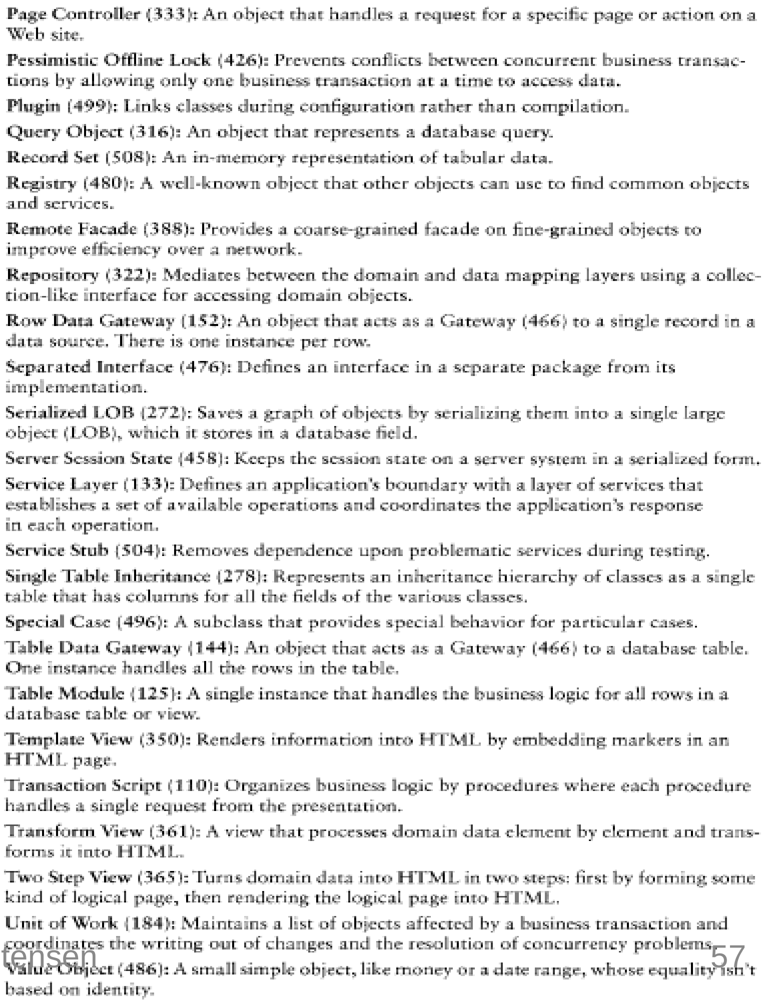
**Mapper (473):** An object that sets up a communication between two independent objects.

**Metadata Mapping (306):** Holds details of object-relational mapping in metadata.

**Model View Controller (330):** Splits user interface interaction into three distinct roles.

**Money (488):** Represents a monetary value.

**Optimistic Offline Lock (416):** Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction.



**Page Controller (333):** An object that handles a request for a specific page or action on a Web site.

**Pessimistic Offline Lock (426):** Prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data.

**Plugin (499):** Links classes during configuration rather than compilation.

**Query Object (316):** An object that represents a database query.

**Record Set (508):** An in-memory representation of tabular data.

**Registry (480):** A well-known object that other objects can use to find common objects and services.

**Remote Facade (388):** Provides a coarse-grained facade on fine-grained objects to improve efficiency over a network.

**Repository (322):** Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

**Row Data Gateway (152):** An object that acts as a Gateway (466) to a single record in a data source. There is one instance per row.

**Separated Interface (476):** Defines an interface in a separate package from its implementation.

**Serialized LOB (272):** Saves a graph of objects by serializing them into a single large object (LOB), which it stores in a database field.

**Server Session State (458):** Keeps the session state on a server system in a serialized form.

**Service Layer (133):** Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

**Service Stub (504):** Removes dependence upon problematic services during testing.

**Single Table Inheritance (278):** Represents an inheritance hierarchy of classes as a single table that has columns for all the fields of the various classes.

**Special Case (496):** A subclass that provides special behavior for particular cases.

**Table Data Gateway (144):** An object that acts as a Gateway (466) to a database table. One instance handles all the rows in the table.

**Table Module (125):** A single instance that handles the business logic for all rows in a database table or view.

**Template View (350):** Renders information into HTML by embedding markers in an HTML page.

**Transaction Script (110):** Organizes business logic by procedures where each procedure handles a single request from the presentation.

**Transform View (361):** A view that processes domain data element by element and transforms it into HTML.

**Two Step View (365):** Turns domain data into HTML in two steps: first by forming some kind of logical page, then rendering the logical page into HTML.

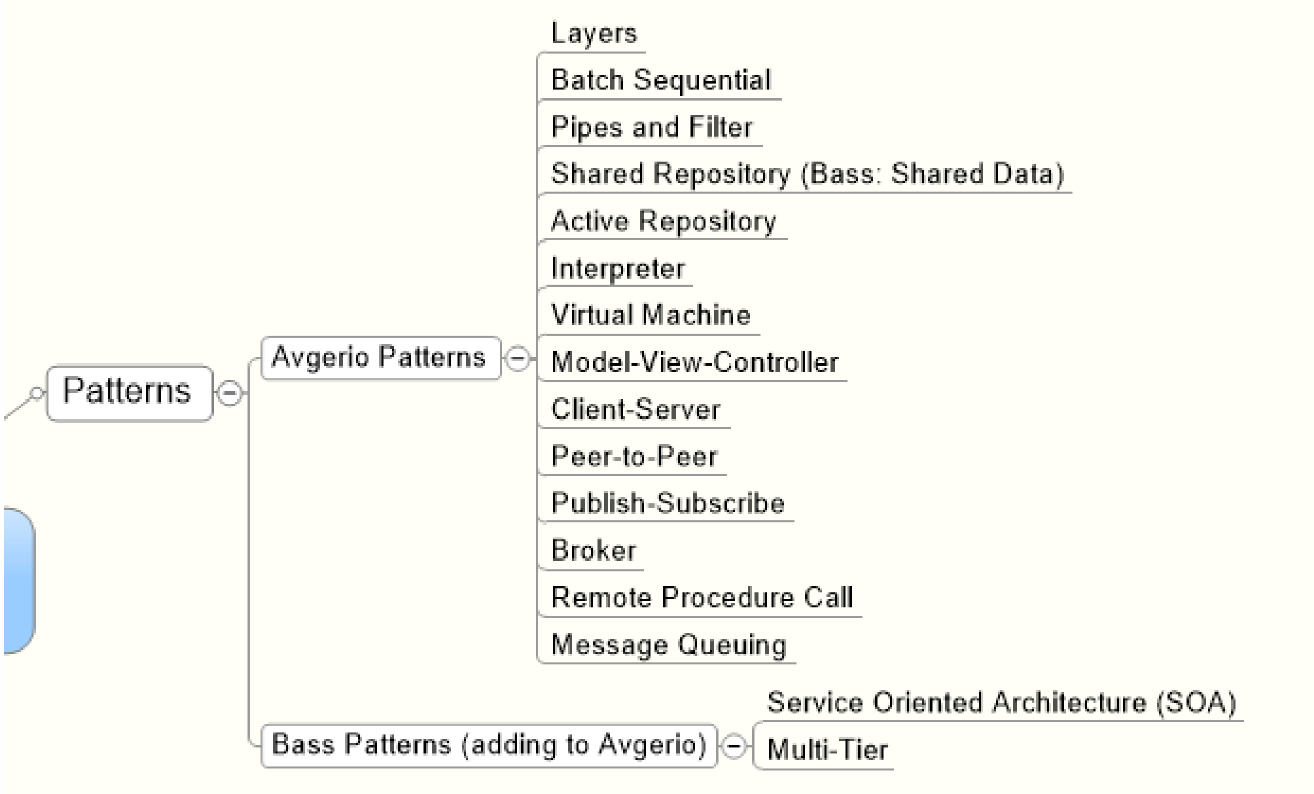
**Unit of Work (184):** Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

**Value Object (486):** A small simple object, like money or a date range, whose equality isn't based on identity.

# Summary

# Cheat Sheet

- Cheat Sheet III


Patterns
Avgerio Patterns
Layers
Batch Sequential
Pipes and Filter
Shared Repository (Bass: Shared Data)
Active Repository
Interpreter
Virtual Machine
Model-View-Controller
Client-Server
Peer-to-Peer
Publish-Subscribe
Broker
Remote Procedure Call
Message Queuing
Bass Patterns (adding to Avgerio)
Service Oriented Architecture (SOA)
Multi-Tier