

CS5351 Software Engineering 2024/25 Semester A

Design Principles and Framework

Dr W.K. Chan

Department of Computer Science

Email: `wkchan@cityu.edu.hk`

Website: `http://www.cs.cityu.edu.hk/~wkchan`

Program Structure

Suppose *J* is an implementation **code** of all features of a project.

- ◆ Where will you place *J* in a program file based on your programming practice?
 - [What if] If *J* will *never* be revised or read again, is it a matter of considering where it is located?
- ◆ *Scenario A*: Put *J* in the body of the main() function.
- ◆ *Scenario B*: If putting the code into different functions is the key, why not put just one statement of *J* into each function?
- ◆ *Scenario C*: Divide *J* by feature, place the “sub-*J*s” in individual functions, and then glue them together.
- ◆ Why do we sometimes “feel” that a feature is easier to codify in one program than another?

=>We care about the program structure

David Parnas [10]

current: x is a list
changed: x is a graph

- ◆ Different ways of decomposing the same program into subroutines produce very different consequences in terms of software maintenance.
- ◆ E.g., X and Y are two programming modules
 - Functional decomposition:
 - $P() = \{ x = X(w); y = Y(x); \dots W(x, y) \dots ; \}$ affect all modules
 - $Y(x) = \{ \text{do whatever we want on the data structure } x \}$
 - Object-oriented decomposition: affect class Z
 - z is an object of class Z
 - $X(z) = \{ \text{do whatever we want on } z, \text{ and } z \text{ maintains its integrity} \}$
 - $Y(z) = \{ \text{do whatever we want on } z, \text{ and } z \text{ maintains its integrity} \}$
 - $P = \{ \dots; X(z); \dots; Y(z); \dots W(z); \}$

the Styles of Program Structure

- ◆ At a higher level, we refer to the software architecture (or system architecture), enterprise patterns, architectural patterns
 - The *big picture* of how a software application is organized
- ◆ At a lower level, we refer to coding idiom, coding principle, design patterns

Where did Architecture begin?



great wall stonehenge pyramid

documentation

- ◆ 1992: Perry & Wolf
- ◆ 1987: J.A. Zachman; 1989: M. Shaw
 - https://en.wikipedia.org/wiki/Zachman_Framework
- ◆ 1978/79: David Parnas, program families
 - “On the criteria to be used in decomposing systems into modules”
<http://dx.doi.org/10.1145/361598.361623>
- ◆ 1972 (1969): Edsger Dijkstra, program families
- ◆ 1969: I.P. Sharp @ NATO Software Engineering conference:
“I think we have something in addition to software engineering”.
- ◆ 1996: A seminal work, “Software Architecture Perspective on an Emerging Discipline,” by David Garlan and Mary Shaw



Popularly used in IS

Software Architecture

- ◆ The basis is *Modularity*
 - i.e., the style to decompose a module into a set of connected sub-modules
 - e.g., The layering style in the OSI model in Computer Networks
- ◆ Our purpose is to use styles to address non-functional needs
- ◆ E.g., How to deal with “Fast start-up time” [non-function req.] when initializing software [functional req.]?
 - Separate the UI code from the program state initialization code
 - Initialize the part of the program state that affects the UI first
 - Initialize a lower-quality, faster-initialization core and transit to a higher-quality, full core.

Software Architecture

- ◆ E.g., How to deal with “Fast start-up time” [non-function req.] when initializing software [functional req.]?
 - Separate the UI code from the program state initialization code
 - => Use the Model-View-Controller pattern to decompose a given module into three lower-level modules (Model, View, Controller)
 - Initialize the part of the program state that affects the UI first
 - => Divide the Model module into a UI-related sub-module, a UI-unrelated sub-module, and a submodule to maintain the model integrity
 - Initialize a lower-quality, faster-initialization core and transit to a higher-quality, full core.
 - => The UI-related sub-module (in Model) should support extensibility and compatibility between submodels. Controller should interoperate₇ between models without affecting the View module

The Strategy We Observe

In architecting, we do:

1. Recursively decompose a module into a set of *interacting* sub-modules. Assign **functional responsibility** to sub-modules to meet the “picked” functional requirement.
 - E.g., submodules for different models and model integrity
2. Assign **non-functional quality attributes** to sub-models
 - E.g., usability by View, extensibility by sub-models, interoperability by Controller
3. Assign **non-functional constraints** on sub-modules
 - E.g., Interoperating different models by the Controller should not affect the View module

We also Observe

- ◆ In architecting, we **neither** work out **nor** decide the following:
 - ◆ How the functional requirement will be implemented
 - e.g., don't bother about the algorithms for software initialization or maintaining the model integrity
 - ◆ How the non-functional requirement will be implemented
 - We only assign the quality attributes and non-functional constraints with explanation/documentation to explain why they are necessary for the implementation to fulfill the non-functional requirement.

Software Architecture, definition

Architecture is the **fundamental concepts or properties** of a system in its environment embodied in its **elements**, **relationships**, and in the **principles of its design and evolution**.

(ISO/IEC/IEEE 42010 Systems and software engineering
— Architecture description. (2011))

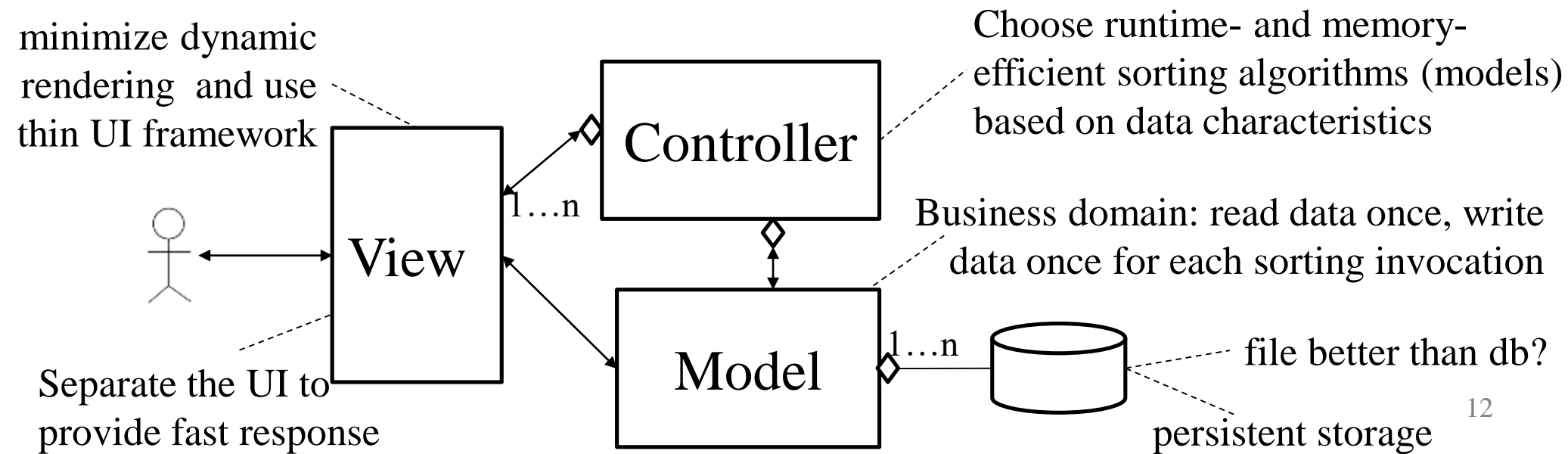
Evolution = maintenance

Software Architecture

- ◆ Architecture is *conceptual*.
 - e.g., We assign meanings to different modules so that their interactions tell the story of how the functional requirements and their quality-related concerns will be addressed.
- ◆ Architecture is about *fundamental* (important) things.
- ◆ Architecture exists in some *context*.
 - E.g., Our architecture for “software initialization” may not make sense if it is a server-based batch-model application.
 - Any non-trivial software system has an architecture. The context dictates whether the architecture is what the system needs.

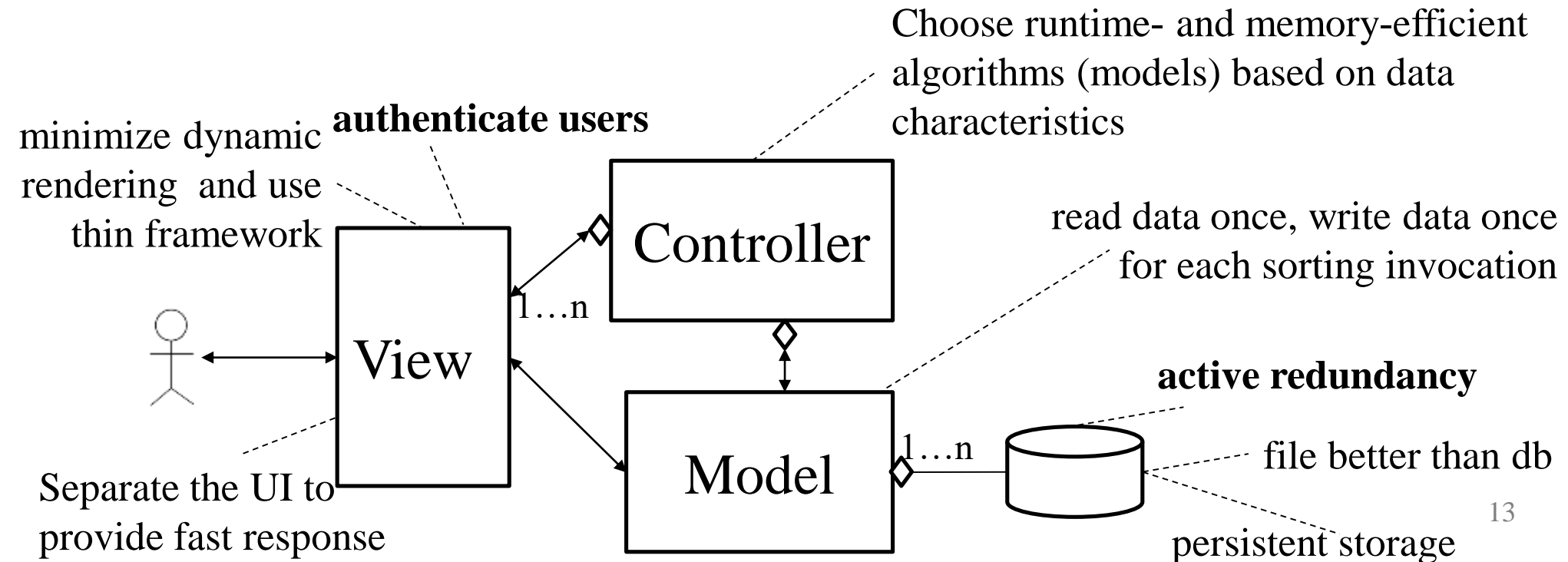
A System for Sorting Data (with “vague” design requirements)

- ◆ The system should
 - respond to users’ click and press quickly
 - run efficiently with a small local memory footprint
 - keep sorting results persistently and efficiently



A System for Sorting Data (with “vague” design requirements)

- ◆ More design requirements are identified during the design
 - Lower-level, within user interface: high security \Rightarrow authenticate users
 - Lower-level, within the data layer: availability \Rightarrow active redundancy



A System for Sorting Data (with “vague” design requirements)

- ◆ We may gradually explore more and more “design requirements” and refine our system to meet each design requirement
- ◆ E.g., the sorting result should be correct
 - Strategy A: Test the system with many test cases
 - => The project itself should satisfy some quality attributes.
Or the code should be testable and conduct sufficient testing in the development process => Testability
 - Strategy B: “Controller” with N-version programming after deployment
 - “N” different sorting algorithms are implemented, and Controller picks the most consistent results as the output.

How Do Software Architects View Their Jobs?

◆ Examples:

- <https://medium.com/@nvashanin/the-path-to-becoming-a-software-architect-de53f1cb310a>
- <https://www.jobhero.com/job-description/examples/computer-software/software-architect>
- <https://www.redhat.com/architect/what-is-software-architect> (see next slide)

Skillset of Software Architects



Software Architects: hard and soft skills needed to become a leader

Software Architects are part engineers, part business leaders. How much of each do you need to land the job? Here's what we found.

Posted: February 25, 2021 | [Matt Shealy](#)

Software Architect: hard skills needed

There's no one-size-fits-all definition because different projects may require different technical knowledge, but there are some skills that all projects will require.

- Unified Modeling Language (UML) is often listed as an essential requirement. What is certainly essential is familiarity with [diagramming complex architectures](#).
- Searching for candidates with a deep knowledge of one or more programming languages essential to the business to support their long-term goals strategically. This could be anything: Java, JavaScript, Python, Ruby, Rust, Go, C, or even COBOL.
- Manage software development practices in a collaborative and agile fashion. That often means a deep familiarity with DevOps practices. This takes hard skills, such as creating an efficient DevOps environment, and soft skills, to keep development and operations teams aligned.

Software Architect: soft skills needed

However, just as important as the technology-related skills are the commonly called "[soft](#)" skills that can drive performance and bring team members together for successful projects.

- **Leadership** - Overseeing the development of a project and coordinating teams of developers to meet design standards requires significant leadership. Software Architects must be able to juggle the needs and demands of projects and teams.
- **Problem-Solving & Conflict Resolution** - Managing and coordinating all of the elements that go into a successful application project requires strong problem-solving skills - both technical and human.
- **Communication** - Communication is a key ingredient in any leadership position. To get the best of teams, Software Architects must clearly explain the mission, deadlines, and expectations.
- **Coaching & Inspiration** - If expectations aren't being met, leaders have to coach and inspire team members to achieve.
- **Organization** - Since Software Architects set the roadmap for development, being organized is key. Often large-scale and intricate UML diagrams are necessary, which requires a systematic and organized way of thinking.
- **Prioritizing** - Software Architects need to quickly prioritize tasks and juggle team members' assignments throughout a product's development.
- **Detailed Thinking** - In any development project, there are a significant number of details that must be managed correctly. This requires extreme attention to detail to make sure the project code meets objectives.
- **Creative Thinking** - The Software Architect has to move teams forward to accomplish a build regardless of the obstacles. This takes the ability to think creatively to find alternate solutions or creative ways to solve problems.

Jobs advertised in Hong Kong

Lead Software Architect

Work with
customers
and team
members

Technical lead

Architect

Both
frontend
and
backend
developers

🕒 5 日前 🏢 全職

Your Part.

- Live your passion for software development in exciting customer projects and be inspired by your colleagues' enthusiasm.
- As a Lead Software Architect, you will be responsible for developing IT architectures of innovative software solutions and providing strategic technology consulting for our customers.
- You will be exposed to and collaborate with customers from a wide range of industries where you will identify the best solutions for these complex organisations.
- You will also be seen as a coach and mentor to your team members.

Your Talent.

- After completing your studies in computer science or equivalent related field (bachelor's/master's), you've continued to develop and you can correctly evaluate new technologies and trends.
- Over several years in your profession, you have gained a broad knowledge of different technologies and methods.
- Your architectural decisions are characterised by non-functional requirements such as availability, flexibility, stability, ease of maintenance, and security while considering economic/business factors.
- On one hand, you are familiar with Enterprise Java technologies, web-based, and desktop clients, application servers, and relational and NoSQL databases. On the other hand, you know how systems outside the Java area are integrated and are familiar with technologies like REST, web services, and messaging.
- Expertise in front-end web development in JavaScript and experience with frameworks like ReactJS and Angular are a bonus!
- You have worked with Docker in both development and production environments.
- You have actively contributed to the career development of your colleagues.
- You are an experienced leader and are undoubtedly a huge reason for the success of adoption of agile in project teams. You have strong experience working closely with project management teams.
- You are aware of the concrete effects of your architectural decisions at the code level and you discuss them openly with developers.
- You have strong communication skills and you are sensitive to different requirements.

Agile process

Jobs advertised in Hong Kong

We are seeking a highly motivated and experienced AI Platform Architect to lead the design, development, and implementation of our cutting-edge AI platform. You will play a pivotal role in shaping the platform's architecture, ensuring its scalability, performance, and security. This is a unique opportunity to work on challenging technical problems and contribute to the advancement of our AI capabilities.

Responsibilities:

- Lead the overall architecture design of the AI platform, including requirement analysis, system design, technology selection, and technical detail management, ensuring a reasonable, efficient, and scalable architecture.
- Lead the AI platform development team, guide the design and development of core business modules, and address platform requirements and algorithm engineering challenges.
- Implement high-performance algorithm training and deployment inference acceleration through distributed computing, GPU acceleration, and other optimization techniques.
- Ensure the platform's scalability, high availability, and security, guaranteeing stable and efficient system operation.
- Stay abreast of the latest AI technologies and tools, continuously optimizing and refactoring the platform architecture.
- Undertake other tasks assigned by supervisors.

Qualifications:

- Bachelor's degree or above in Computer Science, Software Engineering, or a related field. 5+ years of experience in platform development.
- Proven experience in designing and developing AI-related platform architectures.
- Strong understanding and hands-on experience with containerization technologies like Kubernetes and Docker.
- Familiarity with big data processing technologies such as Hadoop and Spark.
- Expertise in microservice architecture, distributed system design, and database technologies like MySQL and Redis.
- Proficient in Java and Python development with a solid foundation in Java technologies, including multithreading, NIO, collections, and memory tuning, demonstrated through practical project experience.
- Deep understanding and knowledge of the artificial intelligence field with a broad knowledge base. Experience in designing and implementing AI projects is a plus.
- Familiarity with MLOps frameworks and tools such as Git and Jenkins.
- Experience with DevOps/Agile development processes and project management. Experience with high-concurrency workloads is preferred.
- Proactive, highly responsible, quick learner, and adaptable to new working environments and challenges.

Architecture

Non-functional
Architecture

Agile process

Architect

- ◆ Responsible for all technical high-level design decisions for a system
 - High-level = the architectural level
 - All decisions made now or deferred to a later time
 - Lead the system to evolve in the right direction
- ◆ Know the business impact of the technical decision
 - E.g., Make Decision 1 for user story 2 may harden the possible solution for user story 3, good? Not good?
- ◆ Embrace, anticipate, and manage changes (and their risks)
- ◆ As the promotor and communicator of project vision among stakeholders (including developers)

Attention to your Course Project

- ◆ Document your architectural decisions in the final report
 - Start planning your system decomposition in Sprint 1
 - The system decomposition you made; the tradeoff of the current system decomposition versus some other system decompositions you have considered
 - The architectural style and patterns adopted in your system decomposition
 - E.g., How does your system decomposition support high “scalability”? Any specific architectural style or tradeoff points you made?

Architecture Exercise

Design A Two-Bed Room Apartment.

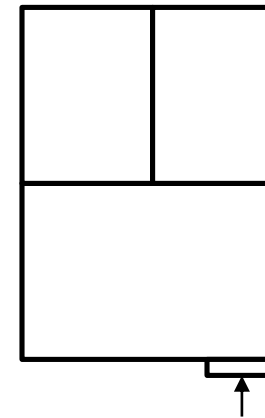
What Will You Do?

Write down the quality attributes you are concerned with in designing the apartment.

- ◆ You have given the following floor plan of an apartment
- ◆ Require to keep all existing walls (you can add doors)
- ◆ Need 1 x Dining/living room (D), 2 x Sleeping Rooms (R), 1 x Kitchen (K), and 1 x Bathroom/toilet (T)



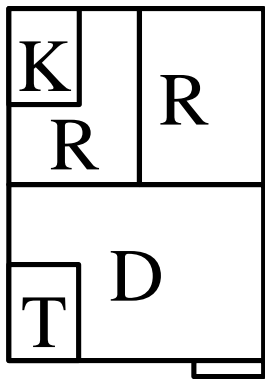
Window side



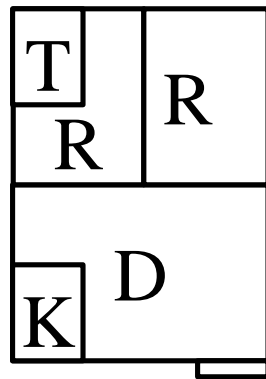
Main entrance

Two-Bedroom Apartment

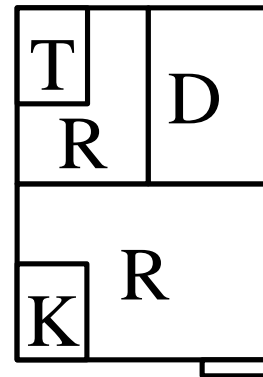
- ◆ Dining/living room (D), Sleeping Room (R), Kitchen (K), and Bathroom/toilet (T)



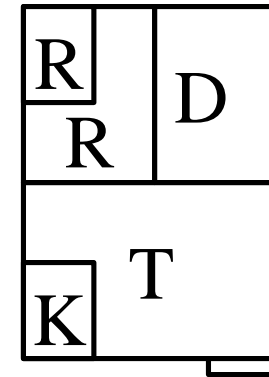
Main entrance



Main entrance



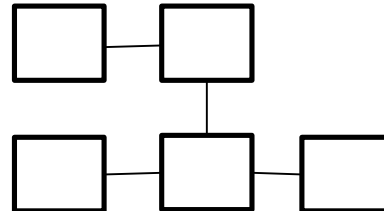
Main entrance



Main entrance

Same component-and-connector organization.

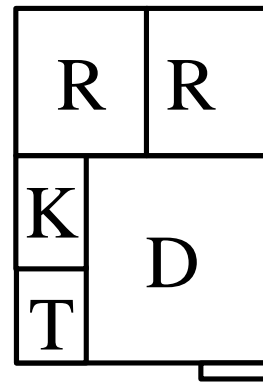
- All functional requirements met!
- Each component is implemented without error!



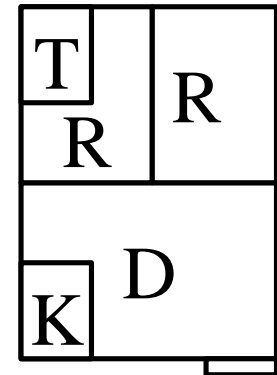
But why do we prefer one to the others?
Something wrong?

{R, R, D, T, K} assigns to the boxes above

Two-Bedroom Apartment



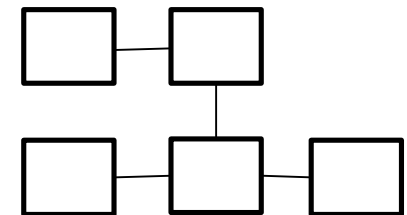
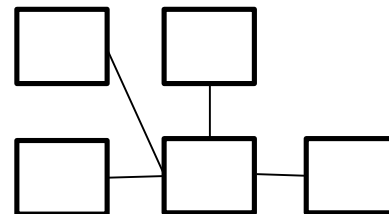
Main entrance



Main entrance

Different component-and-connector organizations. (**different architecture!**)

- All functional requirements met!
- Each component is implemented without error!



Have You Done the Following?

- ◆ Read the floor plan and observe its environmental setting (e.g., the location of windows)?
- ◆ Apply your “common sense” in judging good or bad room organizations?
- ◆ Sketch the Final Floor Plan with room assignment?
- ◆ Sketch some other Floor Plans as alternatives?
- ◆ Compare and Contrast Floor Plans with rationales.
- ◆ Which words (nouns) do you use when you refer to a part of your final floor plan?
- ◆ What are the concerns (quality attributes) you care about in coming up with your final design? (Cleanness? convenience?)

Elements in our Exercise (1/2)

◆ A Domain Model

- A floor plan for an apartment to reflect the reality
- Environmental constraints
 - What you cannot control (outside your control)
 - ◆ One-side window. Main entrance location.
Functionality (1D, 2R, 1K, and 1T by room type)
- Architecturally significant requirement
 - Don't pass through any R when going into D or T.

Elements in our Exercise (2/2)

- ◆ Design
 - The elements you can control
 - E.g., The location of the bathroom and its connected components
 - Evaluation against possible changes in requirements
- ◆ Vocabulary
 - We use the terminologies of our users for effective communications with stakeholders in the domain model
- ◆ View (= capture the design decisions and evaluation)
 - The floor plan is a view of the apartment in the form of a component-and-connector diagram
 - The rationales of some other undesirable views
- ◆ Viewpoint
 - = a set of related concerns on how its views address the concerns in the software architecture

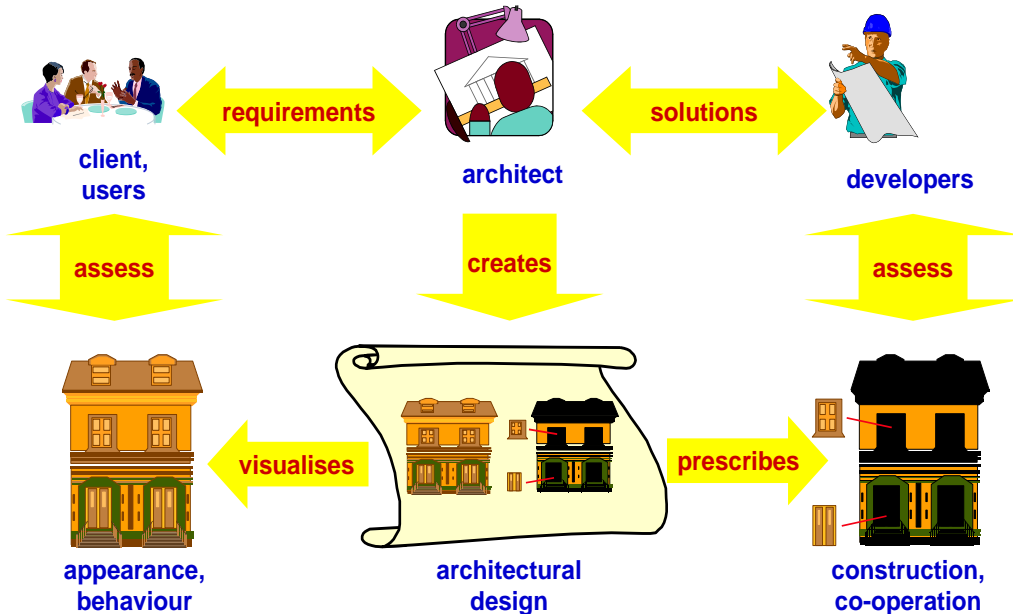
Main Content in Details

Outline

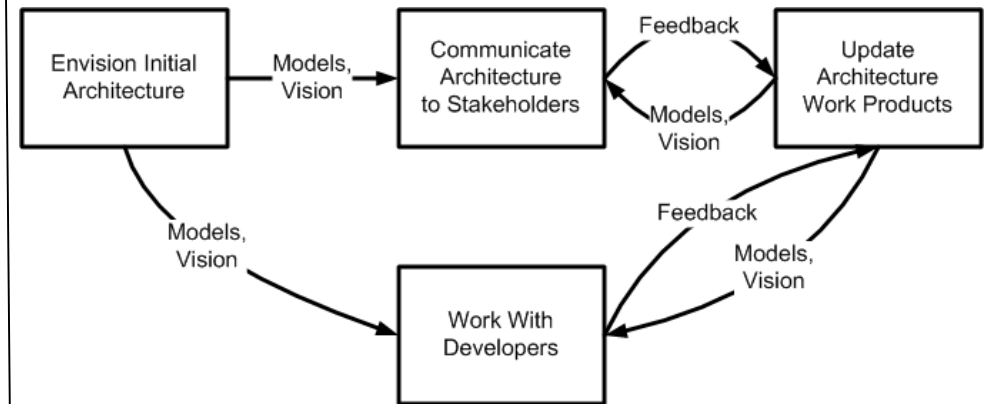
- ◆ **Role of Software Architecture**
- ◆ Driver of Software Architecture
- ◆ Quality Attributes and Patterns/Tactics
- ◆ Attribute-Driven Design
- ◆ Taking Decisions
- ◆ Architectural Styles

Role of Software Architect

In Traditional SW Process



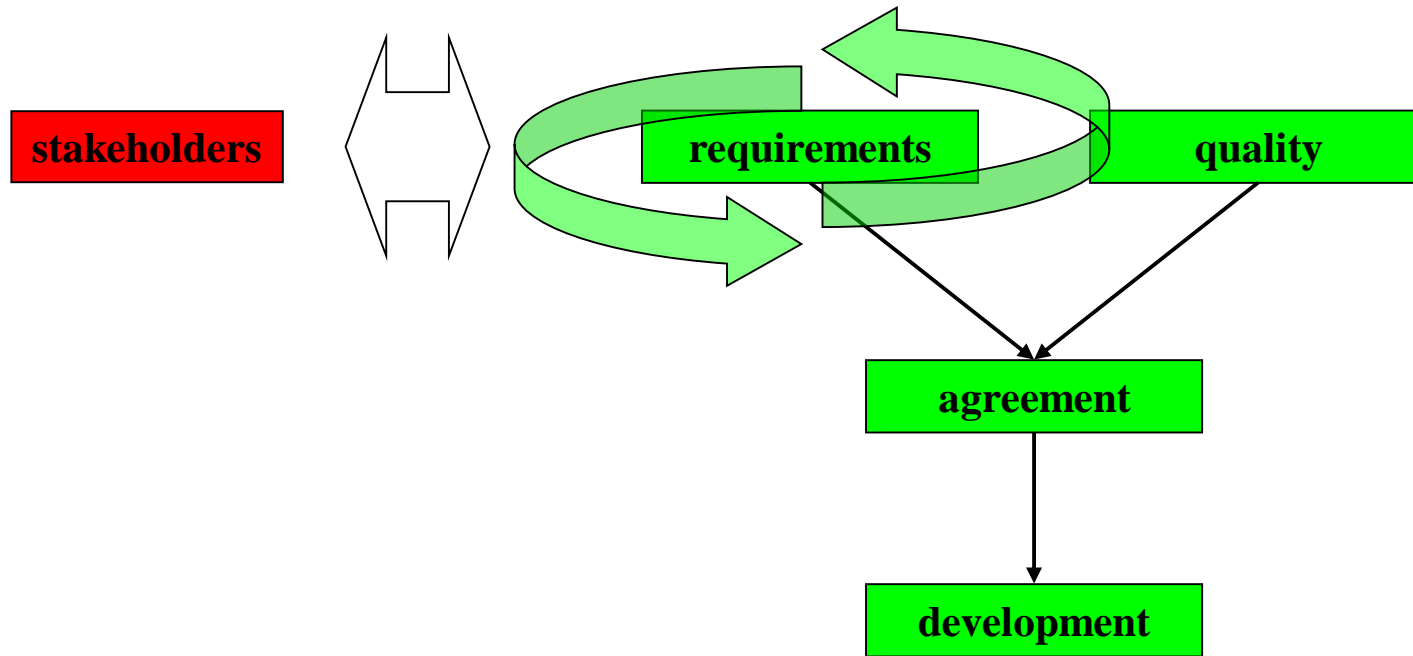
In Agile SW Process



Architecture work products evolve and are fleshed out over time

Copyright 2002-2008 Scott W. Ambler

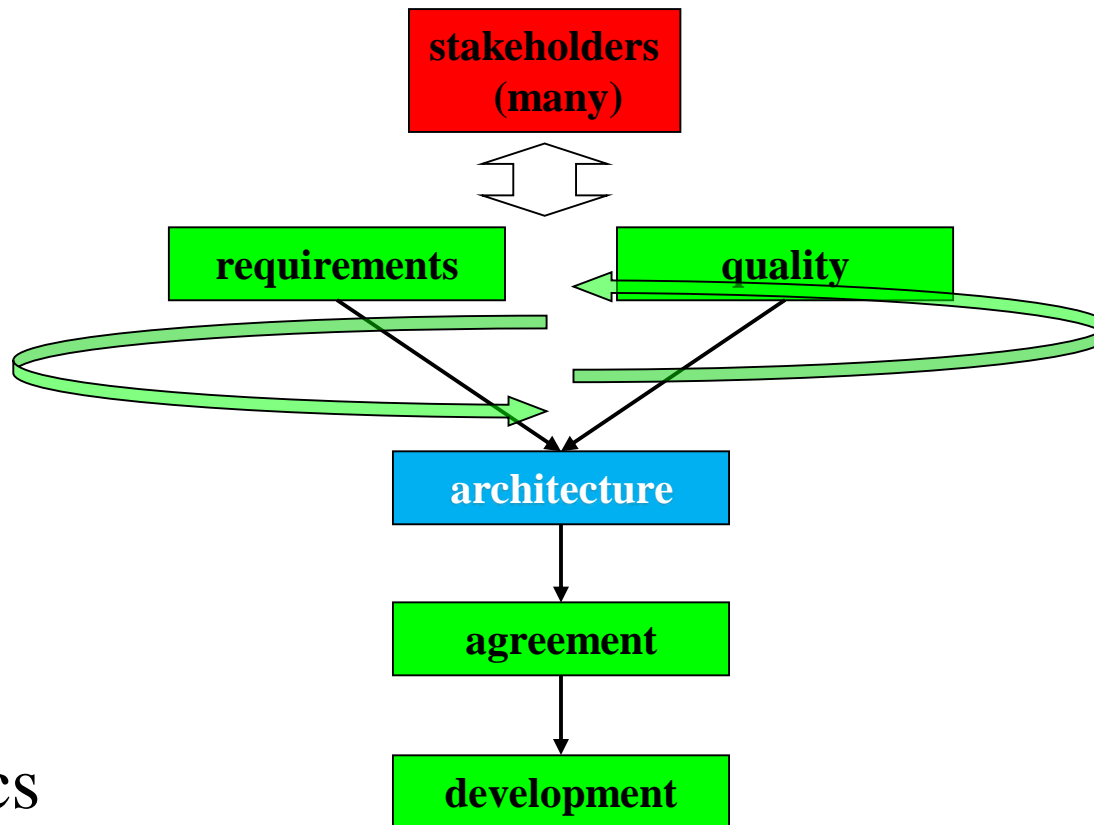
Pre-architecture life cycle



Characteristics

- Iteration mainly on **functional requirements**
- No balancing between functional and quality requirements

Architecture in the life cycle



Characteristics

- Iteration on both **functional and quality requirements**
- Balancing of functional and quality requirements

Why Is Architecture Important?

- ◆ Architecture is the vehicle for stakeholder communication
- ◆ Architecture manifests the **earliest set of design decisions**
 - Constraints on implementation
 - Dictates organizational structure
 - Inhibits or enable quality attributes

Outline

- ◆ Role of Software Architecture
- ◆ **Driver of Software Architecture**

Architectural Driver

- ◆ **Architectural driver** is a **design requirement** that will influence the software architects' **early design decisions**.
- ◆ Note:
 - Functional features (aka functionality) are software requirements, not sufficient to serve as the architectural drivers.
 - Architecture deals with non-functional requirements (e.g., solutions with different quality to achieve the same functionality)

Exemplified Drivers of New Infrastructure

<https://www.hkma.gov.hk/eng/key-functions/international-financial-centre/infrastructure/faster-payment-system.shtml>

Traditional Payment System

◆ Feature

- Fund transfer between accounts
- Settlement

◆ Good qualities

- T+2 settlement
- Single platform, reliable
- Easy to use by merchants & customers

Faster Payment System (FPS)

◆ Feature

- Fund transfer between accounts
- Settlement
- QR code

◆ Good qualities

- **Real-time** and **always-on** fund transfer and settlement
- **Cross-platform**, reliable
- Easy to use by merchants & customers
- **Easy to add new features**
- **Single** common QR code **standard**

Observations on Quality Differences

- ◆ **Keeping:** reliable, easy to use by merchants & customers
- ◆ **Improvement:** T+2 => real-time and always-on
- ◆ **Generalization:** single platform => cross-platform
- ◆ **Extension:** easy to add new features, single standard for QR code

- ◆ So, even if QR code is banned as a feature (and thus, both traditional payment system and FPS has no difference in functional requirement), FPS still needs to address the other quality attributes

Outline

- ◆ Role of Software Architecture
- ◆ Driver of Software Architecture
- ◆ **Quality Attributes and Patterns/Tactics**

Functional Requirements

- ◆ **Functional requirements** specify what the software system needs to do to satisfy the fundamental reasons for the system's existence.

Software Architecture & Quality

- ◆ The notion of **quality** is central in software architecting.
- ◆ Some qualities are observable via execution: performance, security, availability, functionality, usability
- ◆ Some others are not observable via execution: modifiability, portability, reusability, integrability, testability
- ◆ Quality also affects different aspects of software projects

Quality Attributes

- ◆ A quality attribute is a **measurable or testable** property of a system that is used to indicate how well the system satisfies the needs of its stakeholders
- ◆ E.g., Quality Attributes in ISO/IEC 25010:2011 Quality Model

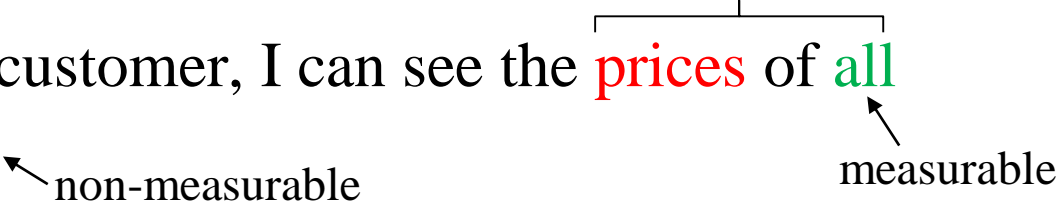


Optional Reading: briefly read the elaboration of each quality attribute to get some ideas on what they are: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>

Measurable / Testable

- ◆ Without knowing how far the current system is from a goal, designing a system to meet this goal is impossible.

Testable if the context of “price” is understood

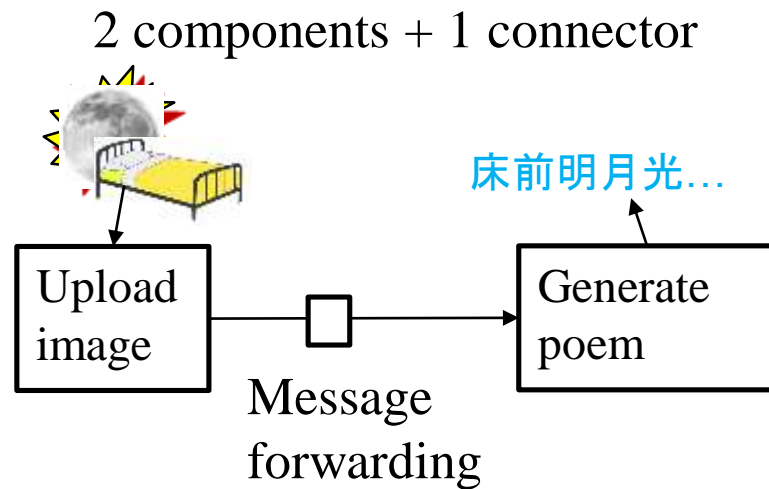
- [**non-measurable**] As a customer, I can see the **prices** of **all** cryptocurrencies **quickly**

 - We can see that if “in a second” is changed to “in 0.02 seconds”, the system requires more designs to meet this goal.
- [**measurable**] As a customer, I can see the **latest and 4 historic** prices of **20** cryptocurrencies I selected **in a second**.

- ◆ As an analyst, we should aim to get the measurable one.

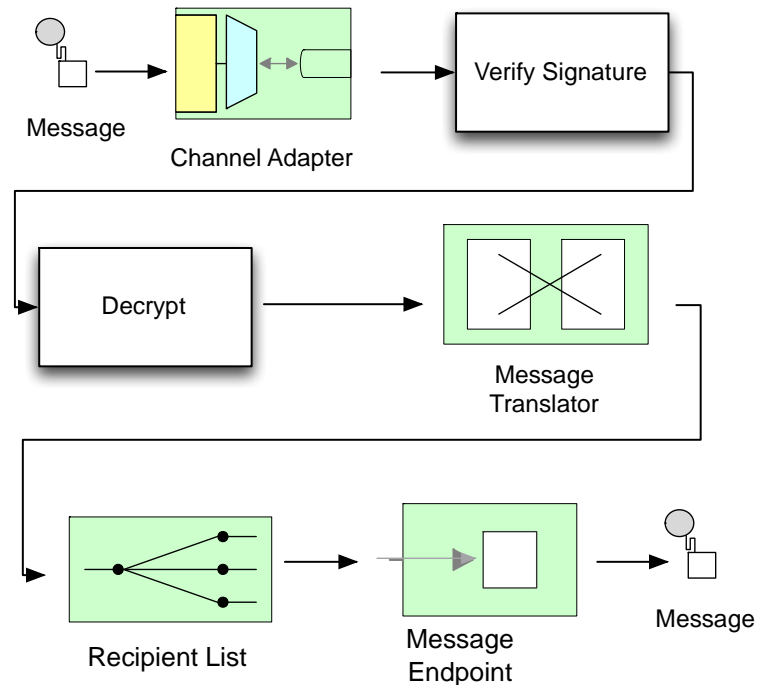
Quality Attributes Tradeoff

- ◆ Many pairs of “-litiity” form tradeoffs.
 - Improving the privacy => lowering usability
 - Higher performance => lower interoperability
 - Higher modularity => longer time-to-market
 - Higher reliability => lower performance
 - Quicker time-to-market and lower cost => lower stability
 - ...
- ◆ We should be aware that addressing one quality attribute by a *system decomposition* should evaluate its impact on some other quality attributes (within the domain model and among the quality attributes of the requirements)

What is a System Decomposition?



Message forwarding
(hide the complexity)



Quality examples:
fast upload; scalable to many users

Exemplified Design Qualities

- ◆ Feasibility
 - ◆ Reusability
 - ◆ Consistency
 - ◆ Simplicity
 - ◆ Stability
 - ◆ Composability
 - ◆ Deployability
-
- ◆ Let's quickly look at some popular design qualities

Feasibility

- ◆ What is the chance of project success?
- ◆ Time to market (feasible for D-day)?
 - Slower: build from scratch; all features; top-down decision making
 - Faster: reuse whenever possible; minimum viable product; developers make decisions
- ◆ Affordability (enough resources to complete the project?)
 - by money?
 - by resources?
 - people (quality and quantity), time, environment (including hardware),...
 - How elastic are the available resources to cater for unanticipated events on top of a “reasonable schedule”?

Reusability

- ◆ Reuse strategy
 - Duplication (multiple copies) versus Reference (shared copy)
 - Baseline is internal to the project or external to it?
 - General-purpose or domain-specific or feature-specific?
 - More generic => fewer constraints on other quality attributes
- ◆ Pre-requisites of reuses available to the project product
 - Open-source? Industry standard? Well-documented interface? Accessibility to the component? Quality?

Consistency

- ◆ Is the design conceptually clean and coherent?
 - understanding a part and the general principle of a component can help generalize the understanding to the whole component
 - Reduce surprise.
 - E.g., Even if there is an exception, it is an exception across as many components as possible so that it forms a disciplined approach to exception handling.
- ◆ Follow the architectural style constraint and document the major architectural decisions
 - *Counterexample*: If modularity is a constraint, don't use “go to” to jump between components without going through their interfaces
- ◆ Follow the design principles and break it with a consistent reason

Simplicity

- ◆ A complex problem does not necessarily need a complex solution.
 - Simple is beautiful
 - One general solution is better than many customized solutions
- ◆ Strategy
 - Minimize variability of components
 - Minimize direct connections between components
 - Make each design decision concise and clean
 - Refactoring the design if needed
 - Identify a fundamental set of primitives and create solutions based on these primitives
 - Change that requires compromising the simplicity? => think twice

Stability

- ◆ Is the architectural design stable?
- ◆ If a design decision depends on N modules, changing one module may lead to changing the other $N-1$ modules.
 - Not a wise design decision.
 - E.g., Use 100 open-source components in the current project. Incompatibility and different paces of development/bug fixing of these components easily make the project's product unstable.
- ◆ A design decision can be fragile in some specific operating environments.
 - Better encapsulate the environmental issue into the underlying platform or framework
- ◆ Using experimental code or features without sufficient evaluation in the project context is risky

Composability

- ◆ Is it easy to compose a higher-level component from supportive components? To what extent?
 - Plug-and-play components?
 - Everything automated?
 - By modifying the mapping script or modifying the code?
- ◆ Are all components ready, or do we need to develop some “glue” scripts? Hard or easy?

Deployability

- ◆ Is it easy to deploy the system into its operating environment?
- ◆ Manual vs. Automated
- ◆ Scheduled update vs. continuous delivery
- ◆ Onsite, cloud-based, or service composition?
- ◆ Version downgrade: not possible vs. possible (easy/difficult)
- ◆ Coordinate with external dependencies needed?
- ◆ Can a subset in an upgrade patch be deployable in a standalone manner?

Apart from Design Quality

- ◆ Quality also exists in other dimensions:
 - Operation
 - Failure
 - Security
 - Change in requirements or features
 - ...

Operation Quality

- ◆ *Performance*: Latency (delay, first response, completion time) and Throughput (output per unit time, bandwidth)
- ◆ *Scalability*: Growth with throughput (# of requests), users (concurrency), data volume (I/O and processing), network scale (# of nodes), and system size (# of components): dynamic scaling possible?
- ◆ *Capacity*: Limit, overload, and utilization rate
- ◆ *Usability*: Easy to learn and recall efficiently? Follow the norm? user satisfaction? Easy to access?
- ◆ *Serviceability*: operation stop vs. continuous operation? Reboot or no reboot? Manual error report vs. automated?
- ◆ *Visibility*: system behavior auditable and logged?

Failure-Related Quality

◆ Reliability

- Mean time between failures (MTBF)
- Mean time to failure (MTTF)

◆ Recoverability

- MTTR: Mean time to recovery, mean time to repair, or mean time to respond

◆ (Functional) Availability

- Can be calculated based on reliability and recoverability
 - E.g., $\text{Availability} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$

Security-related Quality

- ◆ *Authentication*: to confirm user identity
- ◆ *Authorization*: to restrict access
- ◆ *Confidentiality*: to avoid unauthorized data access
- ◆ *Integrity*: to protect the system/data from tampering
- ◆ *Defensibility*: to protect against attacks
- ◆ *Privacy*: to secure confidential data

Change-related Quality

- ◆ *Configurability* (deferring the architectural design is better):
 - How to initialize, (re-)activate, place and bid components?
 - How to add or remove system resources (e.g., nodes or duplicated copies of the same components)
 - How can a feature be toggled on or off?
 - E.g., live parameters > parameters at starting > hardcoded
- ◆ *Customizability*
 - One-size-fits-all, product line, UI skin, JSON for data to decouple from hardcoding the data
- ◆ *Duration*
 - Temporary and then revoked possible? Permanent evolution possible?
- ◆ *Feature Evolution*
 - extensibility for new feature for,; modifiability for existing feature

Chande-related Quality

- ◆ *Compatibility*
 - Standardized interface, format, platform. Version backward/forward compatibility
- ◆ *Portability*
 - Platform independent versus native code? Write once, run anywhere? VM-based?
- ◆ *Interoperability* (with other systems)
 - Standard, presence of mediator, content negotiation
- ◆ *Integration*
 - Standard interface and data format, continuous, real-time, # of integrated components at the same time

Project Health Quality

- ◆ Data persistence, checkpoint, and recovery
- ◆ Backup and disaster recovery plan
- ◆ Maintainability

e.g., Design of Application Programming Interfaces

- ◆ In modern applications, developers often use APIs over the web or from libraries. They sometimes expose application functionalities as APIs for other clients to use.
 - E.g., backend developers wrap their functionality as APIs for frontend developers to build their frontend functionality
- ◆ We want our API design to be stable, open, and supportive to its client applications
- ◆ How [from the design viewpoint]?

e.g., API Design Principles

- ◆ Easy to understand: Usability, Simplicity, Small, Well-documented, Meaningful error messages
- ◆ High quality of services: Scalability, Reliable, Available, Security, Compatibility, Standardized I/O message type,
- ◆ Naming Consistency (from endpoints, functions, I/O, Ordering of elements, similarity with other APIs)
- ◆ Design from the client application's perspective (usability)
- ◆ Principles
 - Explicit interfaces principle
 - Principle of least surprise
 - Small interfaces principle
 - Uniform access principle
 - Few interfaces principle
 - Clear interfaces principle

Home Reading: Search on the web to get an idea about these six principles.

Outline

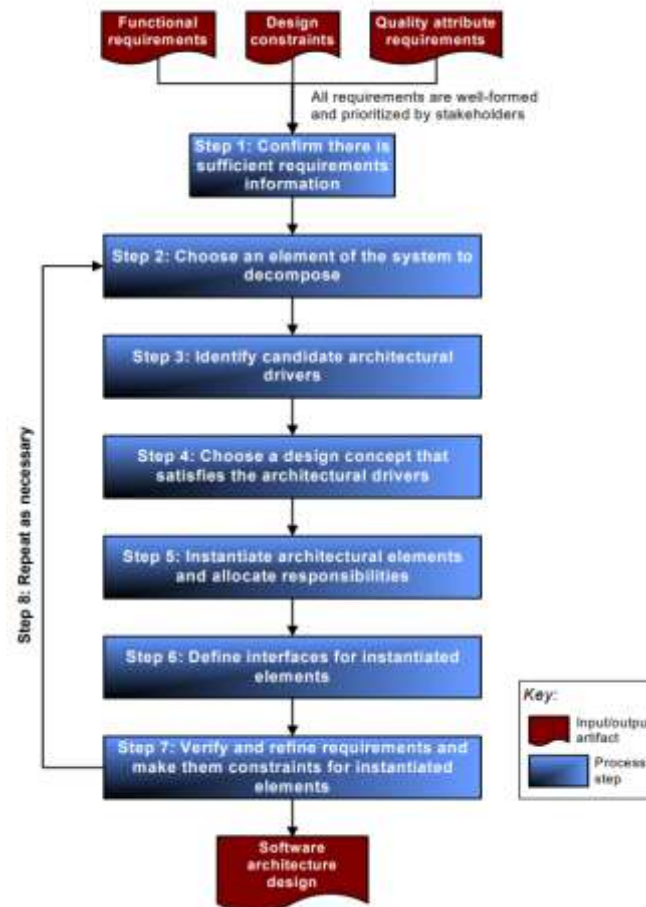
- ◆ Role of Software Architecture
- ◆ Driver of Software Architecture
- ◆ Quality Attributes and Patterns/Tactics
- ◆ **Attribute-Driven Design**
 - We will use it in the tutorial exercise

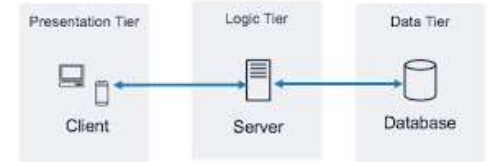
Attribute-Driven Design (ADD)

- ◆ ADD is an architectural design methodology to tackle quality attributes incrementally
- 1. Choose a system module/connector to decompose
- 2. Refine this module:
 - choose architectural drivers (quality is the driving force)
 - choose patterns/tactics/styles that satisfy the drivers
 - apply patterns/tactics/styles and assign responsibilities
- 3. Repeat steps 1-2

Sidebar: The Original ADD

- ◆ The following reference details each step in the original ADD. https://resources.sei.cmu.edu/asset_files/TechnicalReport/2006_005_001_14795.pdf





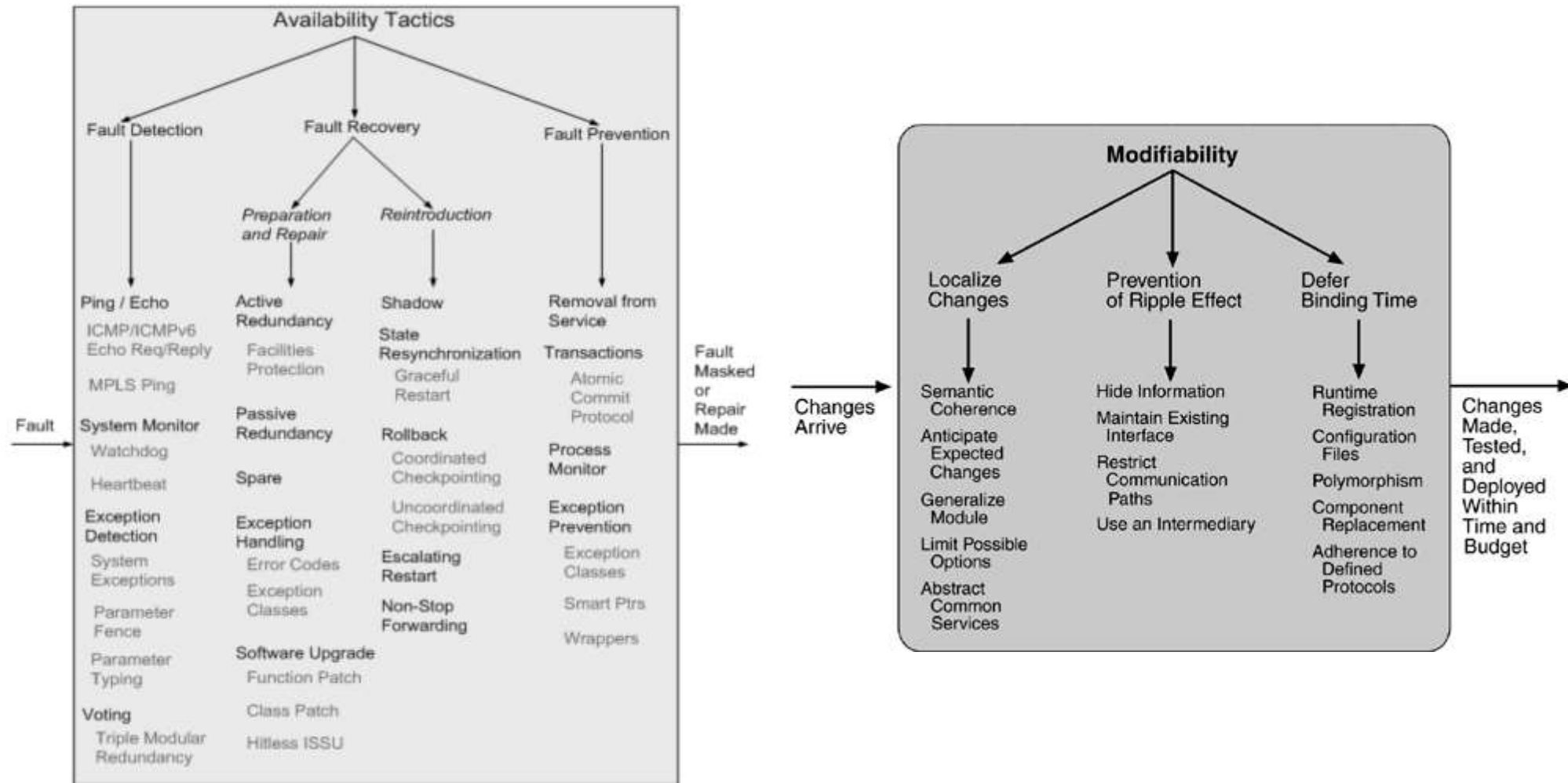
Example ADD iterations

- ◆ Top-level (whole system):
 - address usability
 - \Rightarrow *tactic*: separate user interface \Rightarrow *pattern*: three tier architecture (or MVC in our sorting system example on slide 12).
- ◆ Lower-level, within the presentation tier module:
 - address security
 - \Rightarrow *tactic*: authenticate users
- ◆ Lower-level, within the data tier module:
 - address availability
 - \Rightarrow *tactic*: active redundancy

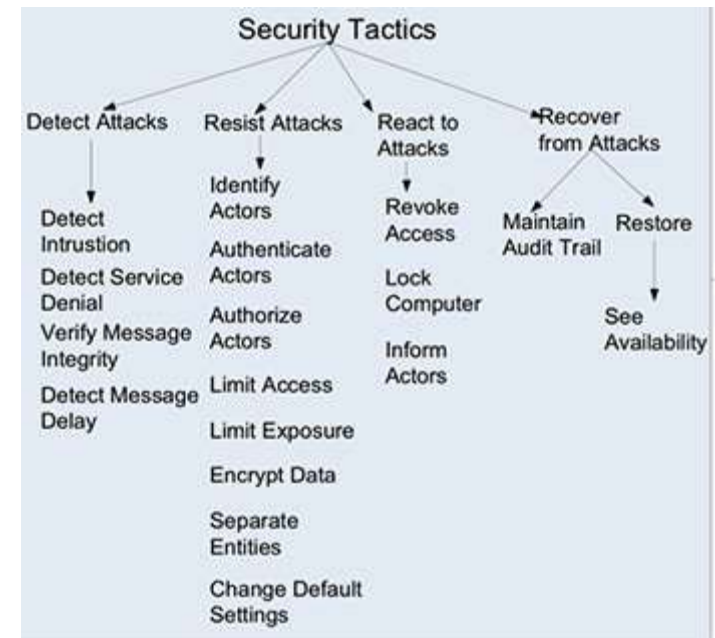
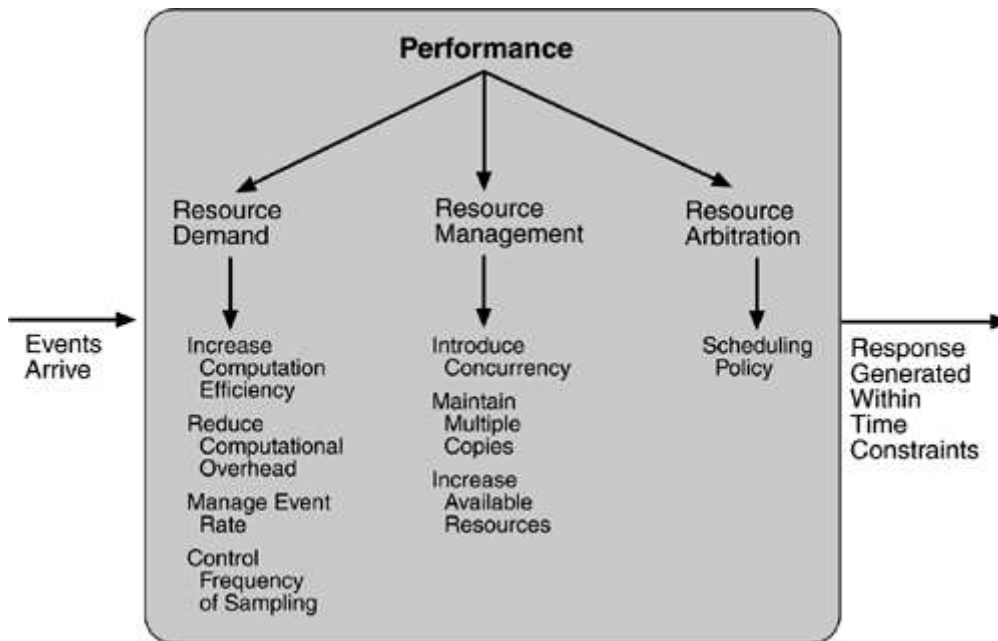
Architectural Tactics

- ◆ ADD iteratively applies one or more architectural tactics to address a quality attribute issue.
- ◆ Architectural tactics are concepts. They are usually used implicitly in a solution.
- ◆ The most often mentioned tactics nowadays are Security tactics (defense approaches).

Availability and Modifiability

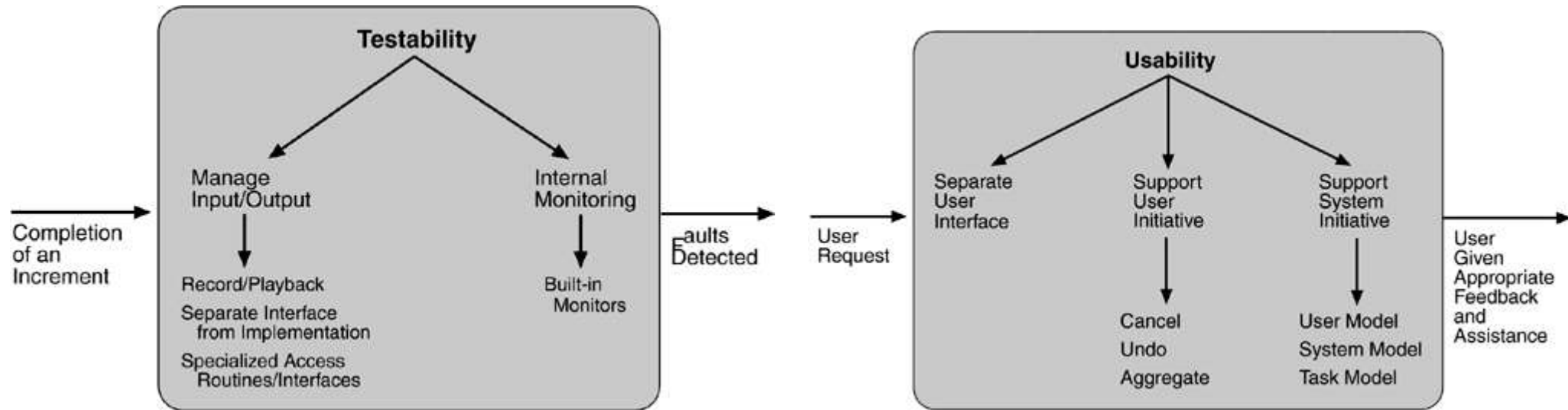


Performance and Security



<https://insights.sei.cmu.edu/blog/security-pattern-assurance-through-round-trip-engineering/>

Testability and Usability



Enterprise Integration Patterns [12]

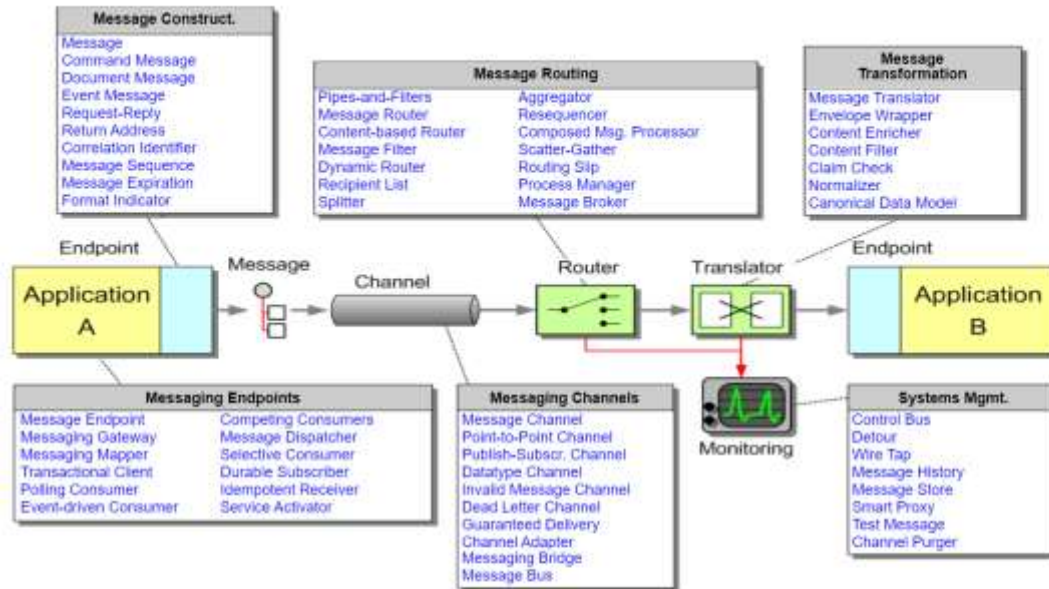
- ◆ Apart from addressing quality attributes by components, we can also address the quality attributes by how we connect these components
- ◆ We use an order processing system to illustrate the idea with a few *Enterprise Integration Patterns [12]* in the next few slides.

Enterprise Integration Patterns [12]

65 patterns

Integration Pattern Language

<https://www.enterpriseintegrationpatterns.com/patterns/messaging/>

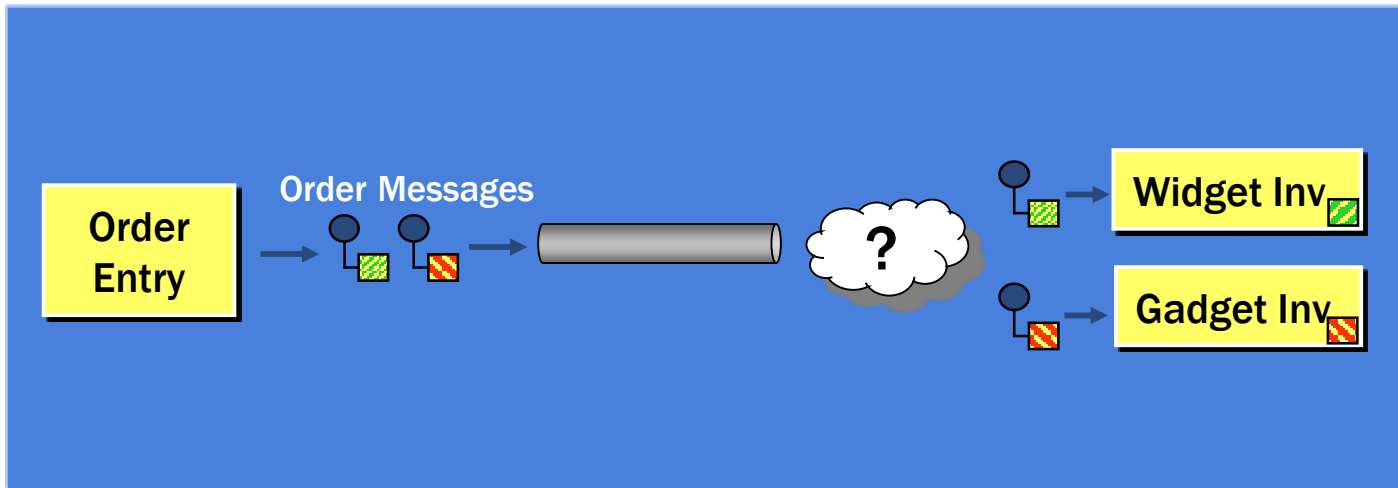


- **Integration Styles** document different ways applications can be integrated, providing a historical account of integration technologies. All subsequent patterns follow the *Messaging* style.
- **Channel Patterns** describe how messages are transported across a *Message Channel*. These patterns are implemented by most commercial and open source messaging systems.
- **Message Construction Patterns** describe the intent, form and content of the messages that travel across the messaging system. The base pattern for this section is the *Message* pattern.
- **Routing Patterns** discuss how messages are routed from a sender to the correct receiver. Message routing patterns consume a message from one channel and republish it message, usually without modification, to another channel based on a set of conditions. The patterns presented in this section are specializations of the *Message Router* pattern.
- **Transformation Patterns** change the content of a message, for example to accommodate different data formats used by the sending and the receiving system. Data may have to be added, taken away or existing data may have to be rearranged. The base pattern for this section is the *Message Translator*.
- **Endpoint Patterns** describe how messaging system clients produce or consume messages.
- **System Management Patterns** describe the tools to keep a complex message-based system running, including dealing with error conditions, performance bottlenecks and changes in the participating systems.

Order Processing System

Question 1?

- ◆ Each provider can only handle a specific type of message
- ◆ Route the request to the "appropriate" provider. But how?
 - Do not want to burden the sender with a decision
 - Letting providers "pick " messages out requires coordination

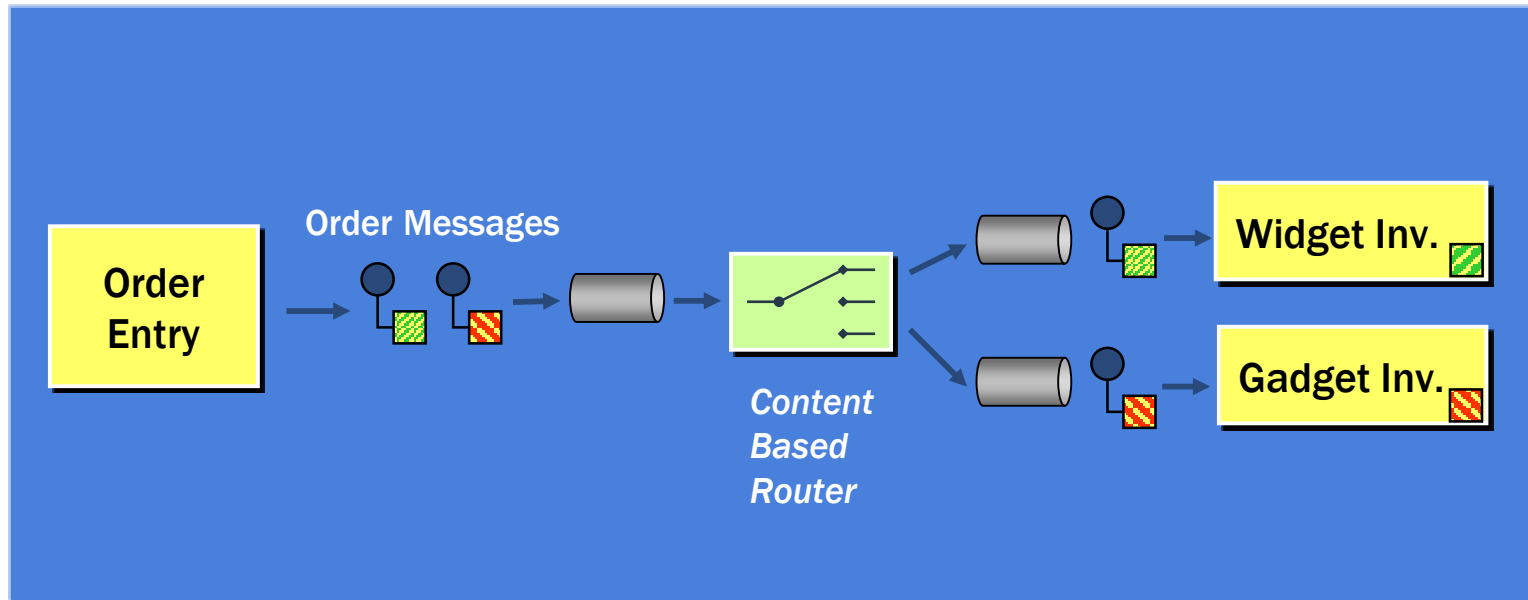


Solution: Content-Based Router

Apache Camel

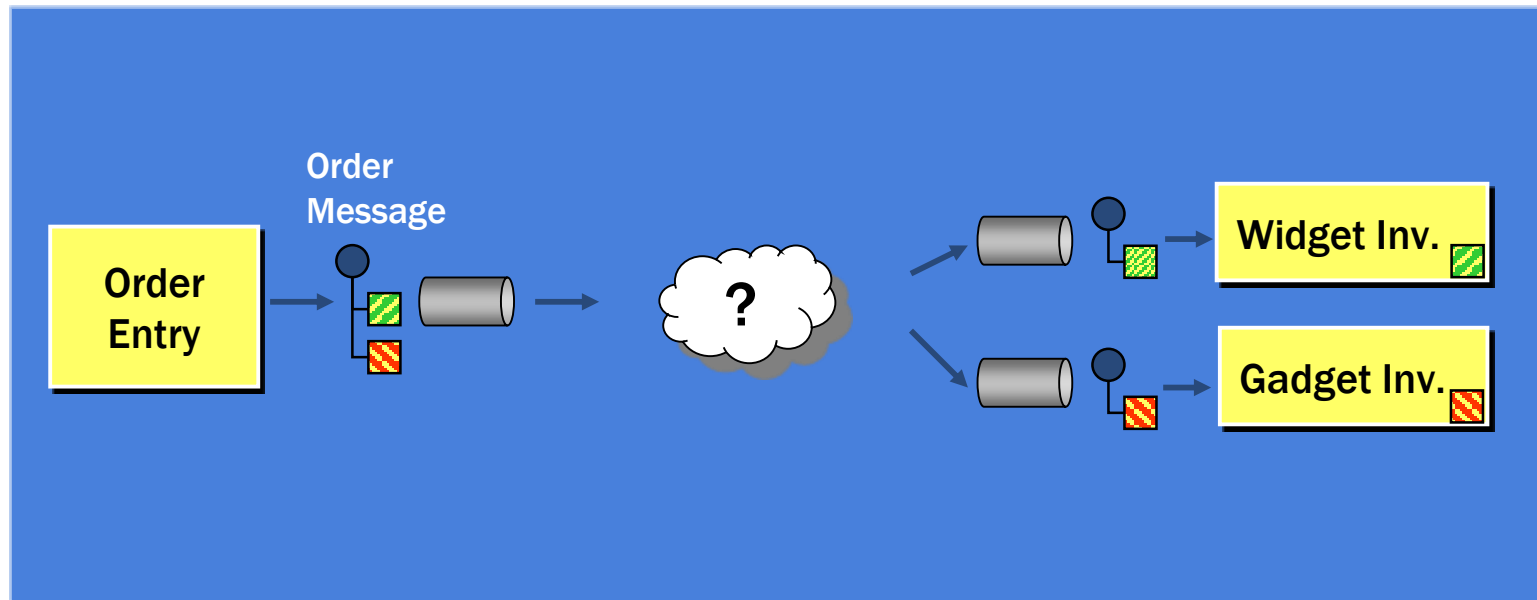
```
RouteBuilder builder = new RouteBuilder() {  
    public void configure() {  
        errorHandler(deadLetterChannel("mock:error"));  
  
        from("direct:in")  
            .choice()  
            .when(header("type").isEqualTo("widget"))  
                .to("direct:widget")  
            .when(header("type").isEqualTo("gadget"))  
                .to("direct:gadget")  
            .otherwise()  
                .to("direct:other");  
    }  
};
```

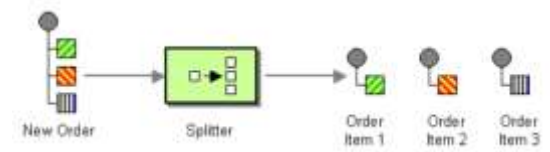
- ◆ Insert a router that selects channels based on the message meta-content
- ◆ Routers forward incoming messages to different channels
- ◆ Message content not changed
- ◆ But if a few messages form a unit, we need to design a bit more (see next slide)



Question 2?

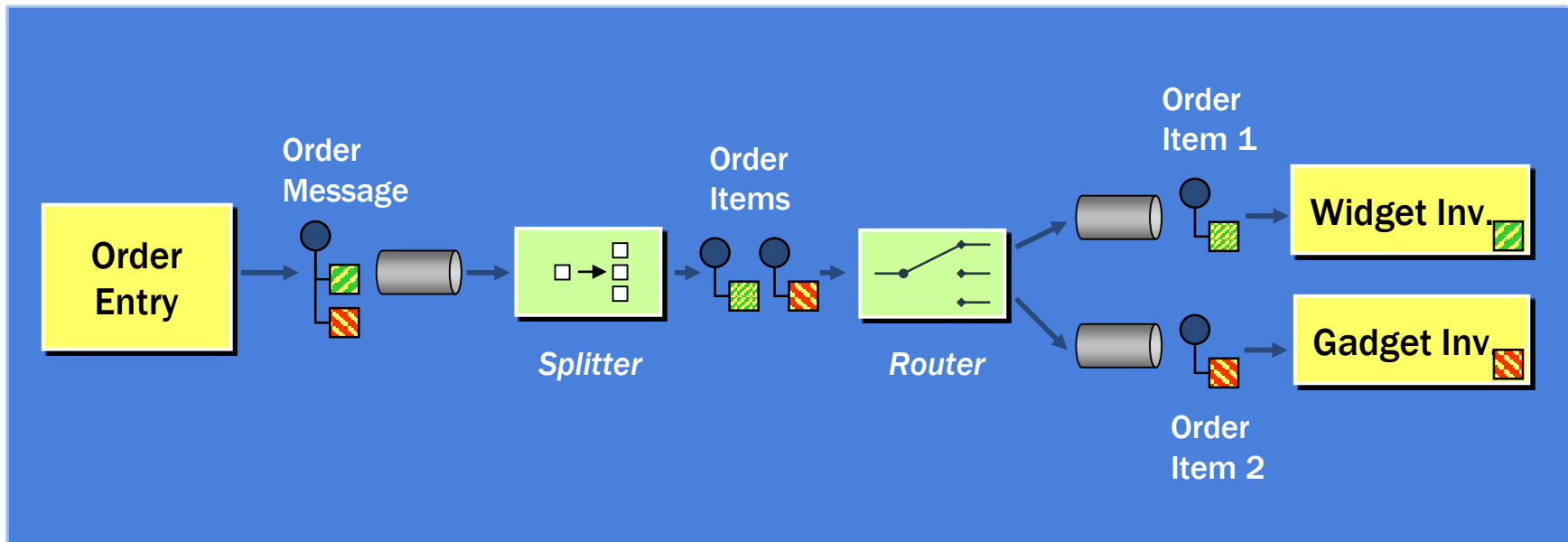
- ◆ How can we process a message that contains multiple elements?
 - need to avoid missing or duplicate elements
 - make efficient use of network resources





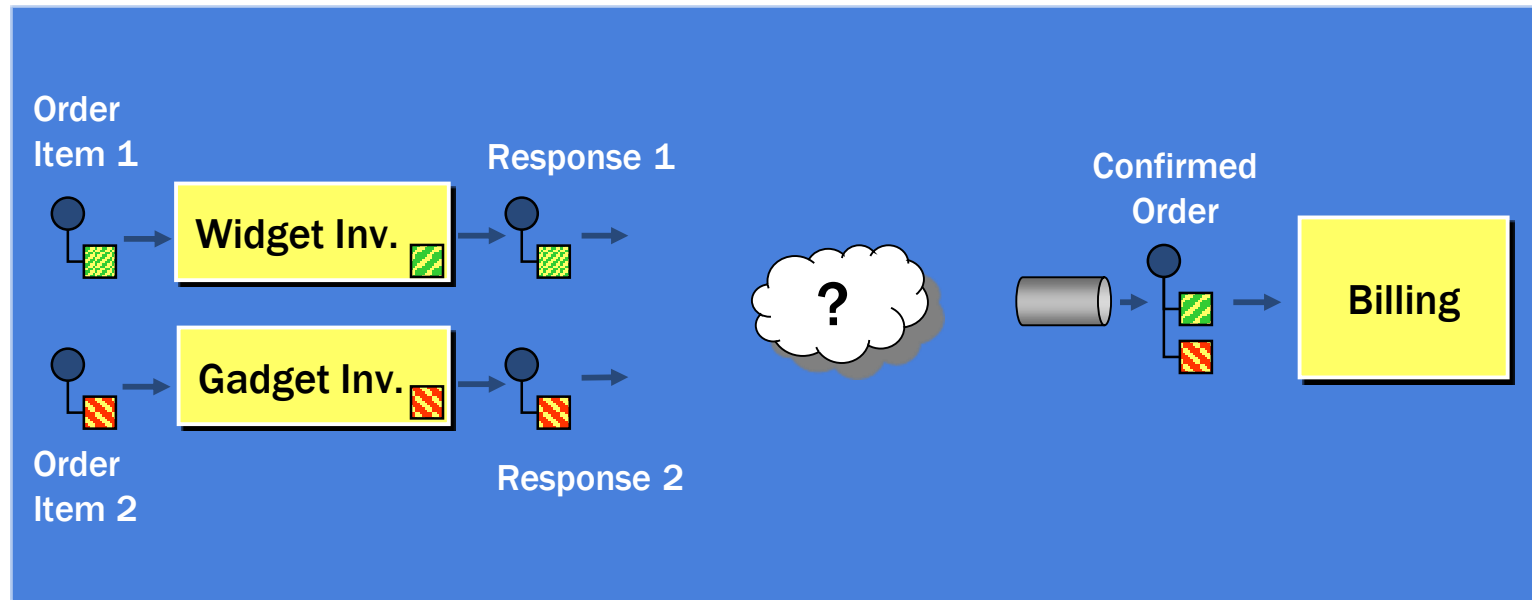
Solution: Splitter & Router

- ◆ Use a *splitter* to break out a composite message into a series of individual messages
- ◆ Then use a router to route the individual messages as before
- ◆ Note that two patterns are composed in the solution
- ◆ we can now split messages, but what about the reverse...?

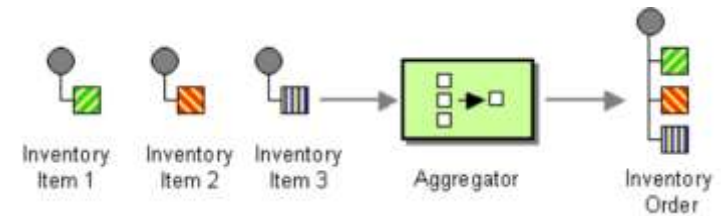


Question 3a? [Single partner]

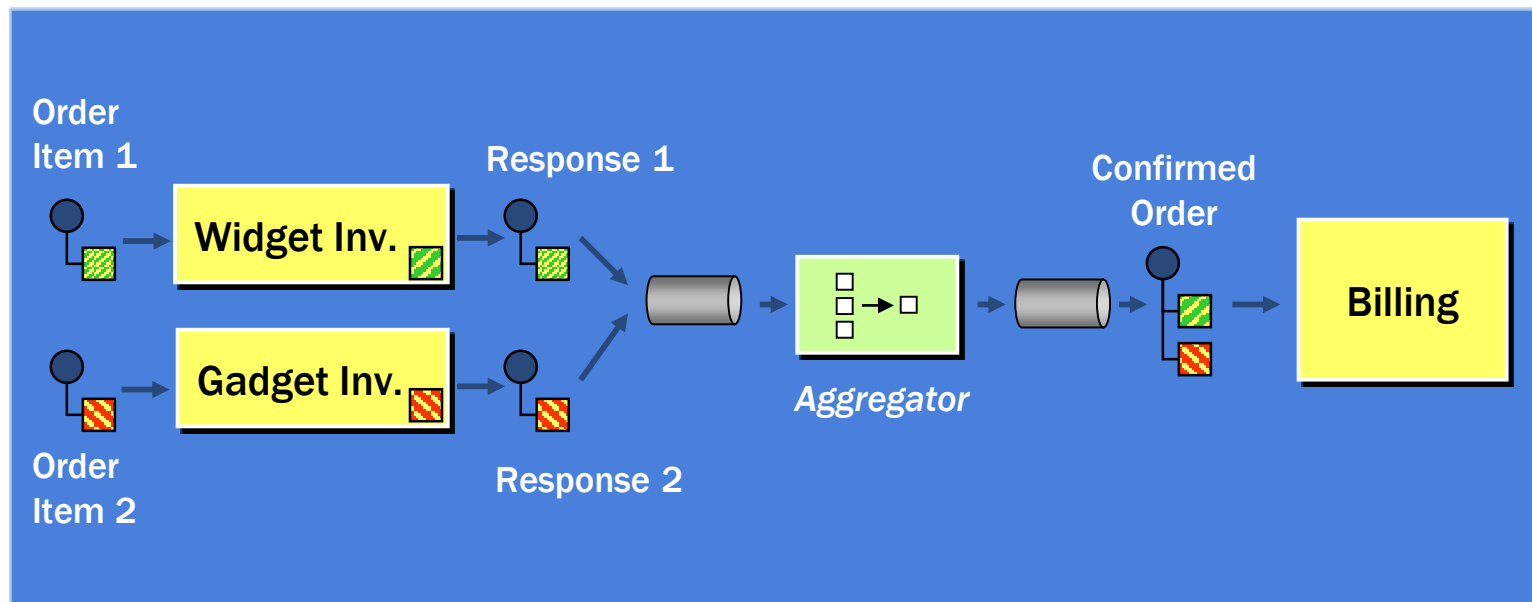
- ◆ How to combine the results of individual but related messages?
 - Messages can be out-of-order, delayed
 - Multiple conversations can be intermixed



Solution: Aggregator

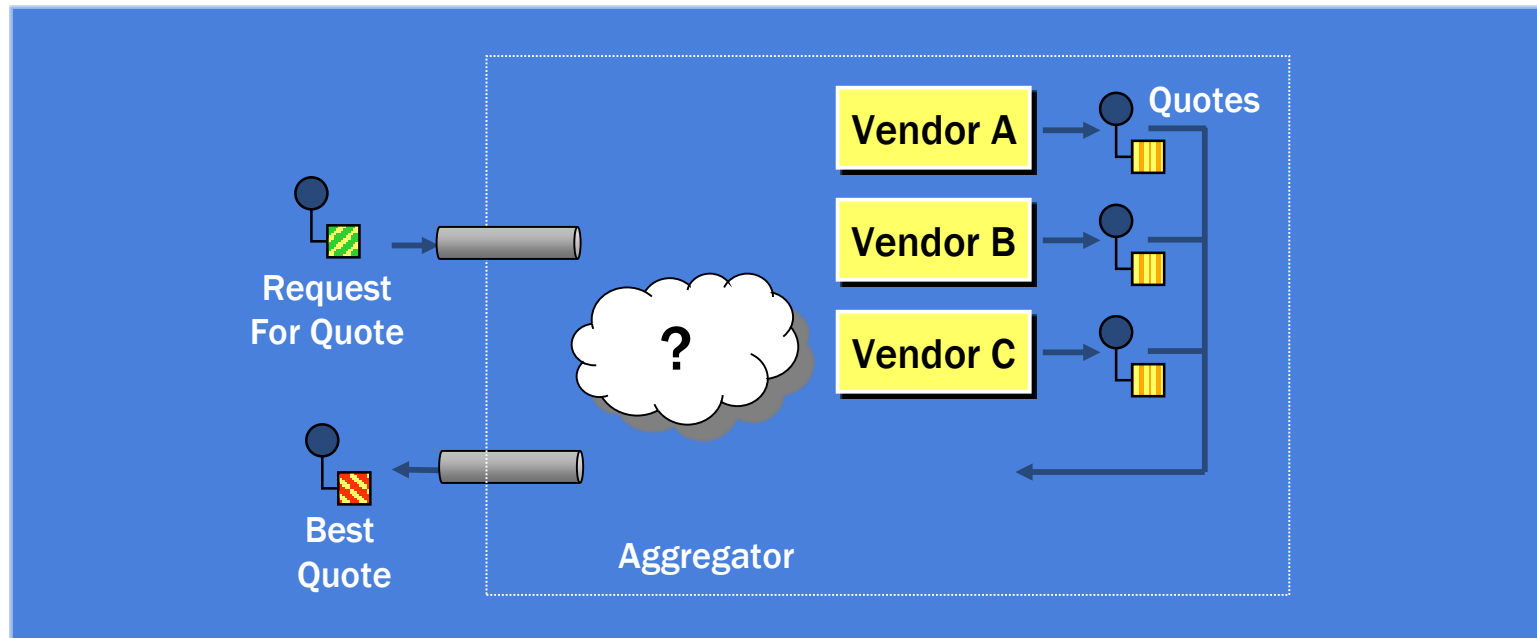


- ◆ Use a stateful filter (an Aggregator)
- ◆ Collects and stores messages until a complete set has been received (completeness condition)
- ◆ Publishes a single message created from the individual messages (aggregation algorithm)



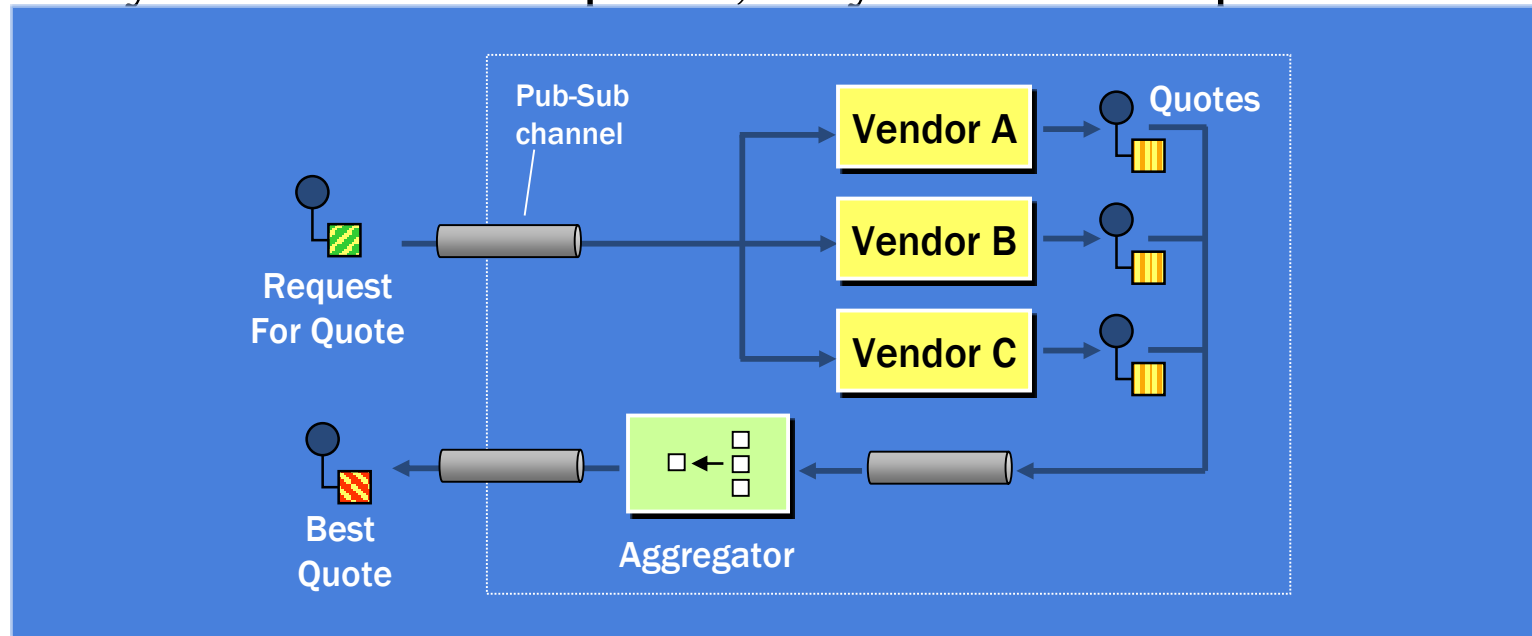
Question 3b: [Multiple partners]

- ◆ How to send a message to a dynamic set of recipients?
- ◆ And return a single response message?



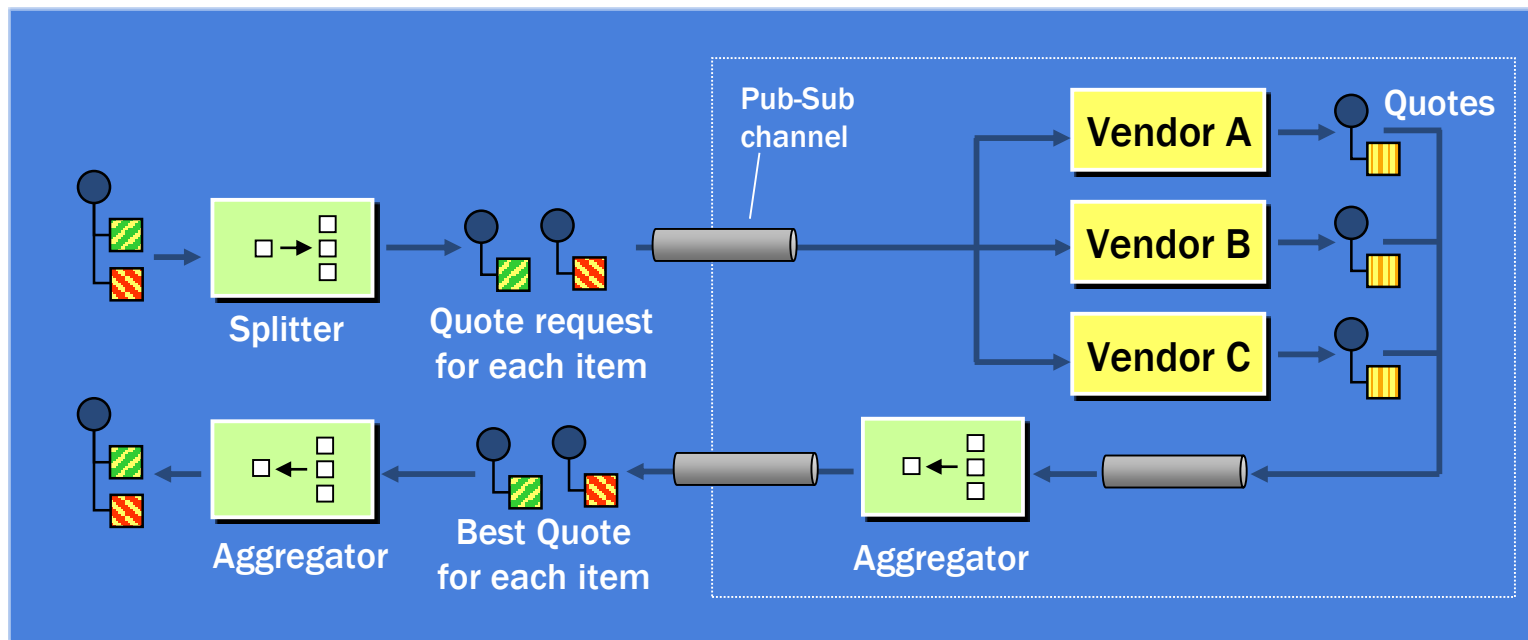
Solution: *Scatter-Aggregator Pair*

- ◆ Send a message to a publish-subscribe channel
- ◆ Interested recipients subscribe to a "topic"
- ◆ Aggregator collects individual response messages
 - may not wait for all quotes, only returns one quote



Complex composition

- ◆ Receive an order message
- ◆ Use splitter to create one message per item
- ◆ Send to scatter/gather, which returns the “best quote” message
- ◆ Aggregate to create a quoted order message



Outline

- ◆ Role of Software Architecture
- ◆ Driver of Software Architecture
- ◆ Quality Attribute and Patterns/Tactics
- ◆ Attribute-Driven Design
- ◆ **Taking Decisions**

Uses of design decisions

- ◆ Identify key design decisions for a stakeholder
 - Make the key decisions quickly available. E.g., introducing new people and making them up-to-date.
 - ..., Get a rationale, evaluate/validate decisions against requirements
- ◆ Evaluate impacts
 - If we want to change an element, what elements are impacted (decisions, design, issues)?
 - Clean up the architecture, identify important architectural drivers
- ◆ Make the design decision explicit
 - E.g., architectural description with rationale
 - E.g., Use the 4+1 architectural view to represent the architecture

Representation

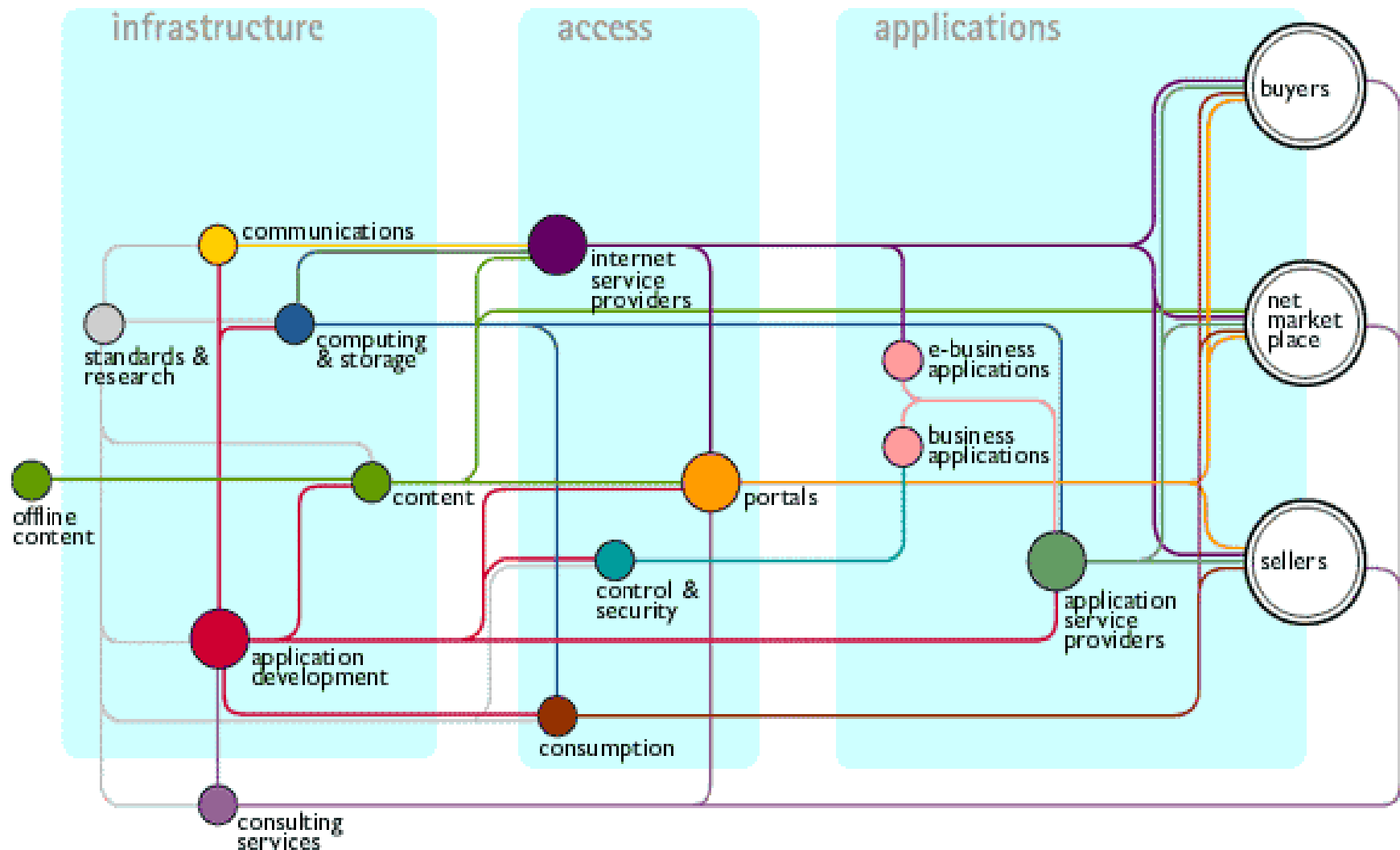
- ◆ While we make our design decision, we need to manifest the design concept as **explicit artifacts (i.e., representation)** to communicate with stakeholders
 - e.g., your team members and your customers or users to address their concerns on the non-functional requirements

Architecture presentations in practice

- ◆ By and large two flavors:
 - Powerpoint slides – for managers, users, consultants, etc
 - UML diagrams, for technicians
- ◆ See an example ... see the next slide

Example

The diagram is an administrator's view.
An accountant's view will be very different as the accountant is concerned with how the transaction money flows in the system and matches up.

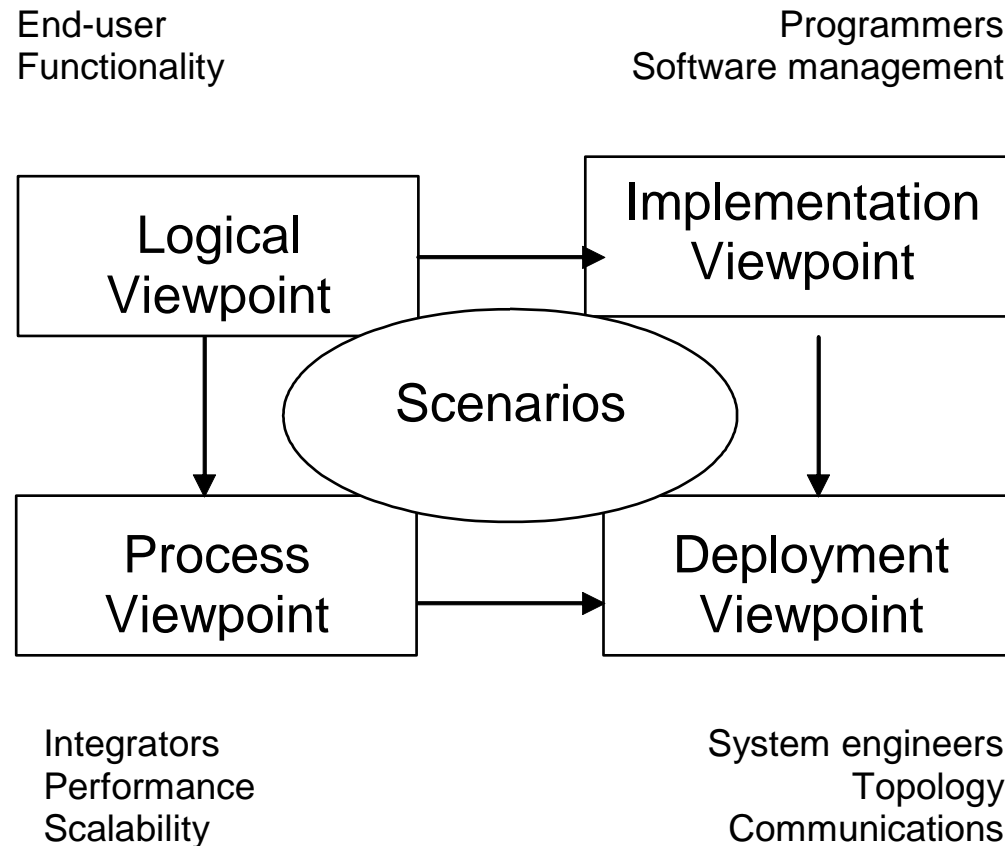


As a result...

- ◆ Different representations
 - For different people
 - For different purposes
- ◆ These representations are both descriptive and prescriptive

e.g., The 4+1 view model

[https://en.wikipedia.org/wiki/4%2B1_architectural_view_model]



The 4+1 view model in UML

- ◆ Logical view
 - Class diagram, sequence diagram, state diagrams
- ◆ Implementation/Development view
 - Component diagram, package diagram
- ◆ Process view
 - Activity diagram
- ◆ Deployment/Physical view
 - Deployment diagram
- ◆ Scenario
 - Use case diagram

Outline

- ◆ Role of Software Architecture
- ◆ Driver of Software Architecture
- ◆ Quality Attributes and Patterns/Tactics
- ◆ Attribute-Driven Design
- ◆ Taking Decisions
- ◆ **Architectural Styles**

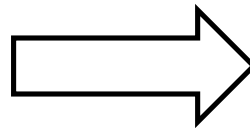
Architectural styles

- ◆ An architectural style is a description of **component and connector types** and a pattern of their runtime control and/or data transfer.

- No formal concept for what is a component and what is a connector

- Learn by examples

- ◆ Examples from Wikipedia



- Blackboard
- Client-server (2-tier, 3-tier, n-tier, cloud computing exhibit this style)
- Component-based
- Data-centric
- Event-driven (or implicit invocation)
- Layered (or multilayered architecture)
- Microservices architecture
- Monolithic application
- Peer-to-peer (P2P)
- Pipes and filters
- Plug-ins
- Representational state transfer (REST)
- Rule-based
- Service-oriented
- Shared nothing architecture
- Space-based architecture

https://en.wikipedia.org/wiki/Software_architecture#Architectural_styles_and_patterns

Basic Kinds of Architecture Style [11]

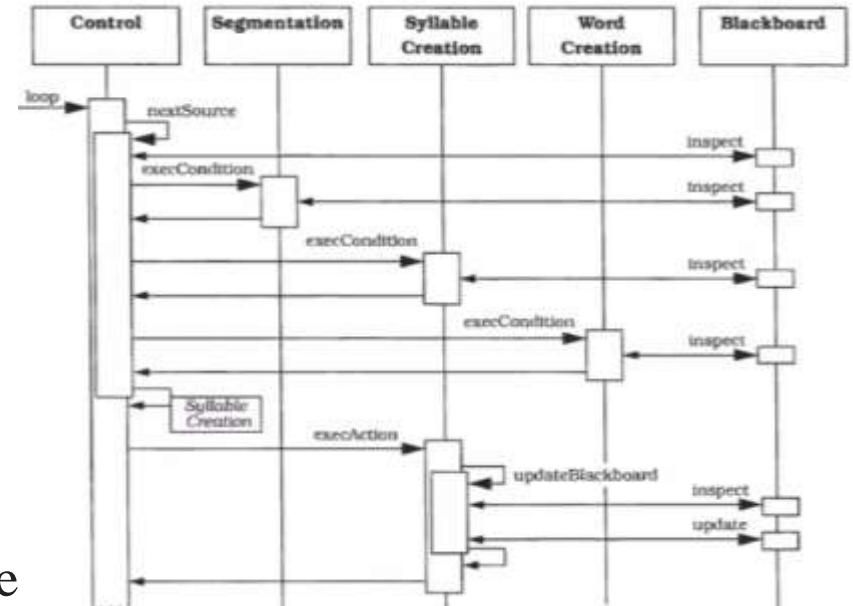
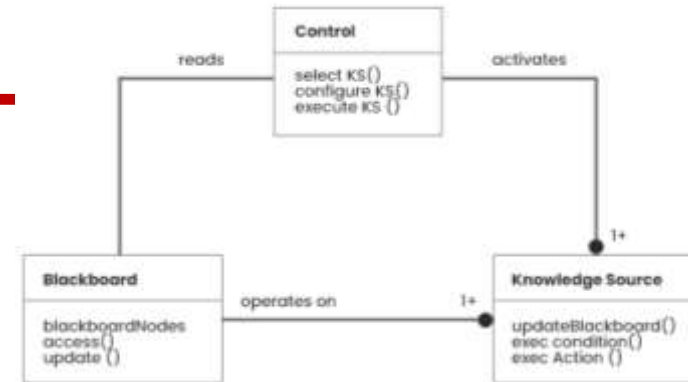
Table 1: List of architectural style

Application Type	Architectural Style
Shared Memory	1. Blackboard 2. Data-centric 3. Rule-based
Distributed System	1. Client-server 2. Space based architecture 3. Peer-to-peer 4. Shared nothing architecture 5. Broker 6. Representational state transfer 7. Service-oriented
Messaging	1. Event-driven 2. Asynchronous messaging 3. Publish-subscribe
Structure	1. Component-based 2. Pipes and filters 3. Monolithic application based 4. Layered
Adaptable System	1. Plug-ins 2. Reflection 3. Microkernel

And many more: e.g., model-view-controller

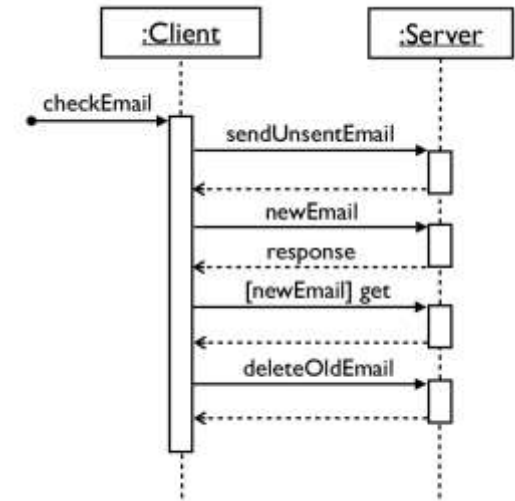
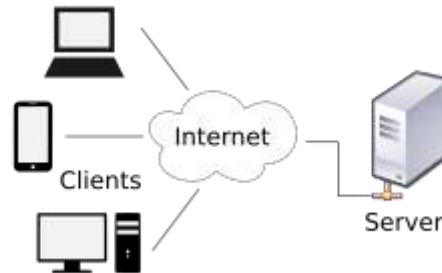
Blackboard

- ◆ **Blackboard (BB)**
 - Data: problem and partial solution.
- ◆ **Knowledge source (KS)**
 - Specialized modules, each providing a solution to some part of the entire problem.
 - Condition() checks whether the module can contribute to the BB
 - Acton() updates BB.
- ◆ **Control**
 - Periodically monitor changes on BB
 - On identifying changes, decide KS to be selected and executed.

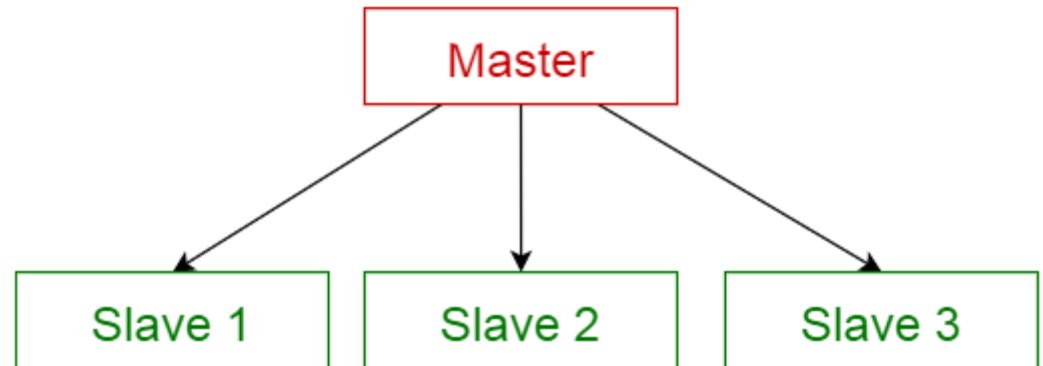
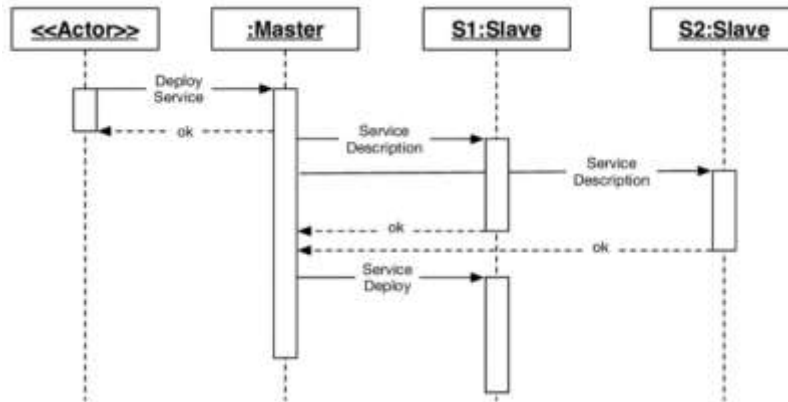


Client-Server and variant

◆ Client-server

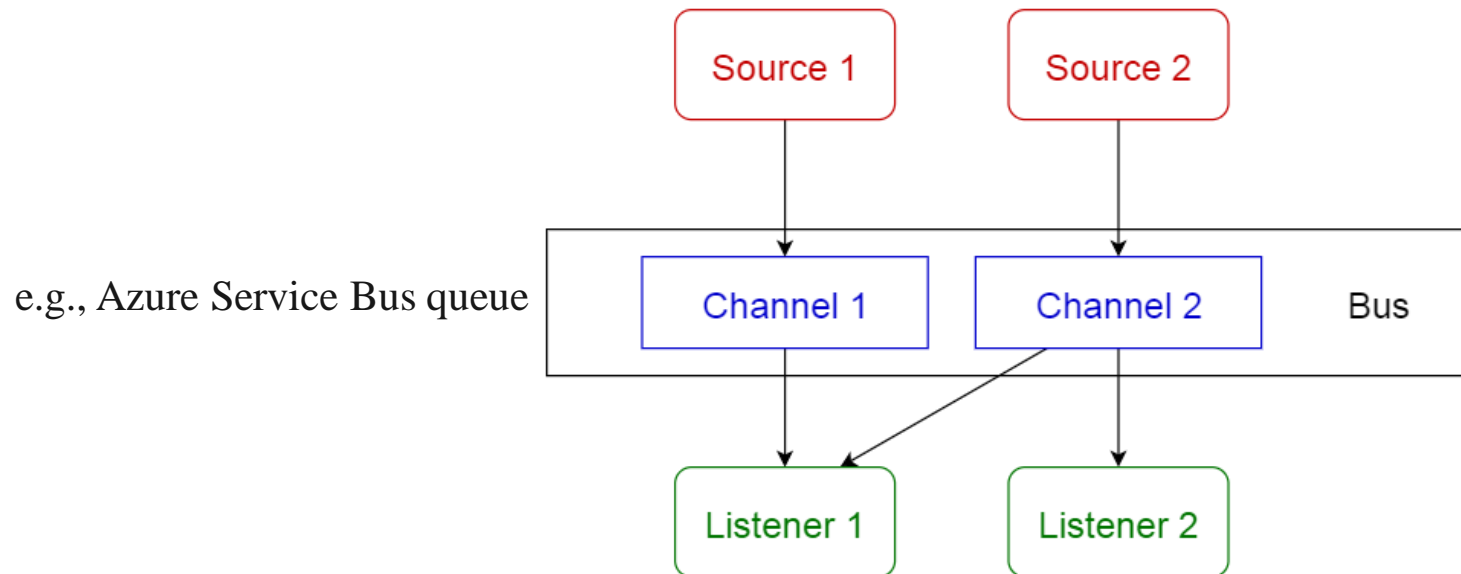


◆ Master-slave

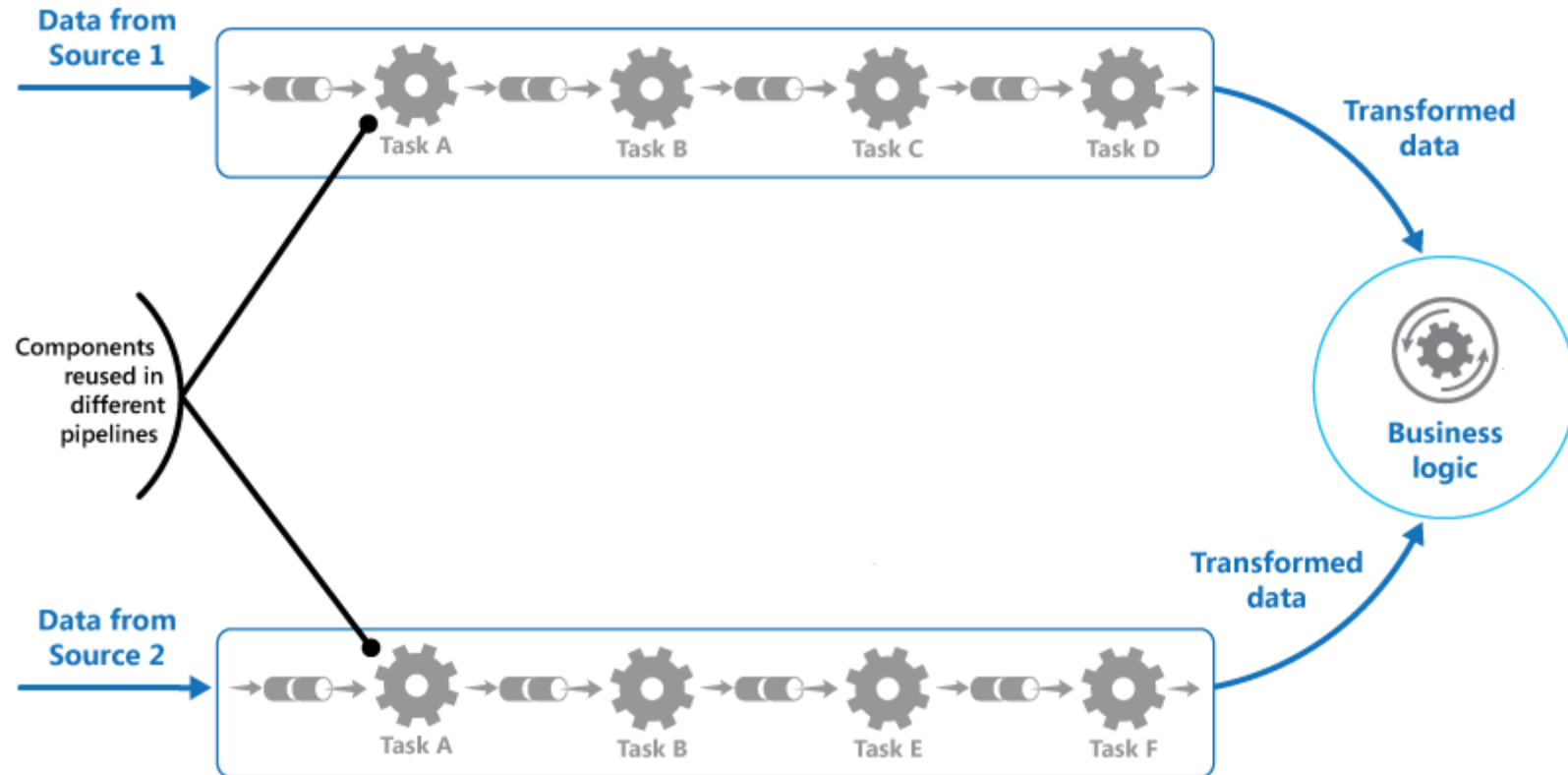


Event-bus/Publish–subscribe pattern

- ◆ Source publishes an event to a channel
- ◆ Channels are grouped into bus
- ◆ Subscriber subscribes to a channel



Pipe-and-Filter



<https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters>

From source 1 to the transformed data, the data goes through a series of transformations (tasks).

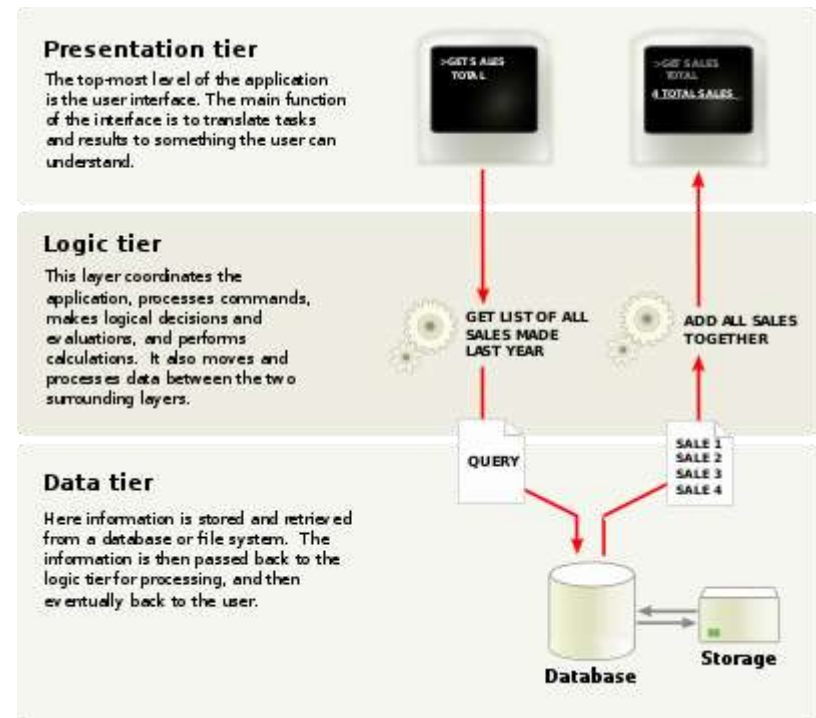


Layered pattern

OSI 7-layer model

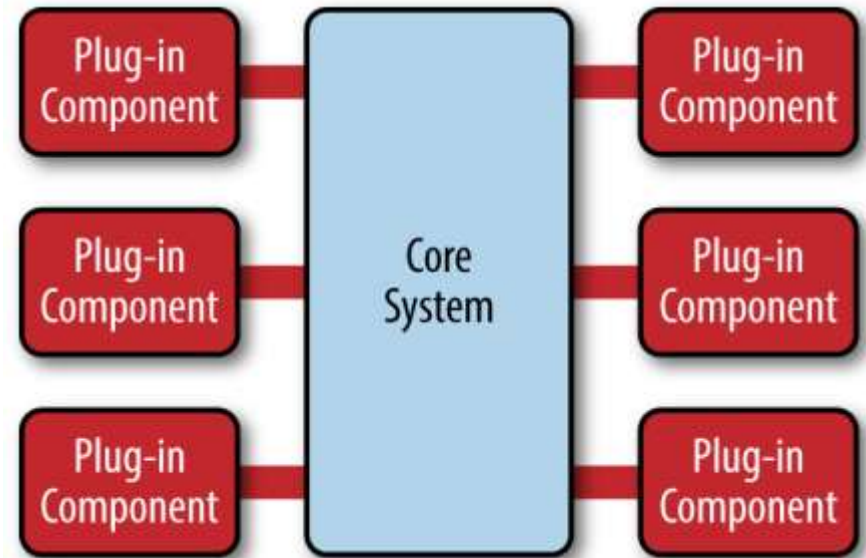
Layer		
Host layers	7	Application
	6	Presentation
	5	Session
	4	Transport
Media layers	3	Network
	2	Data link
	1	Physical

Three-tier architecture

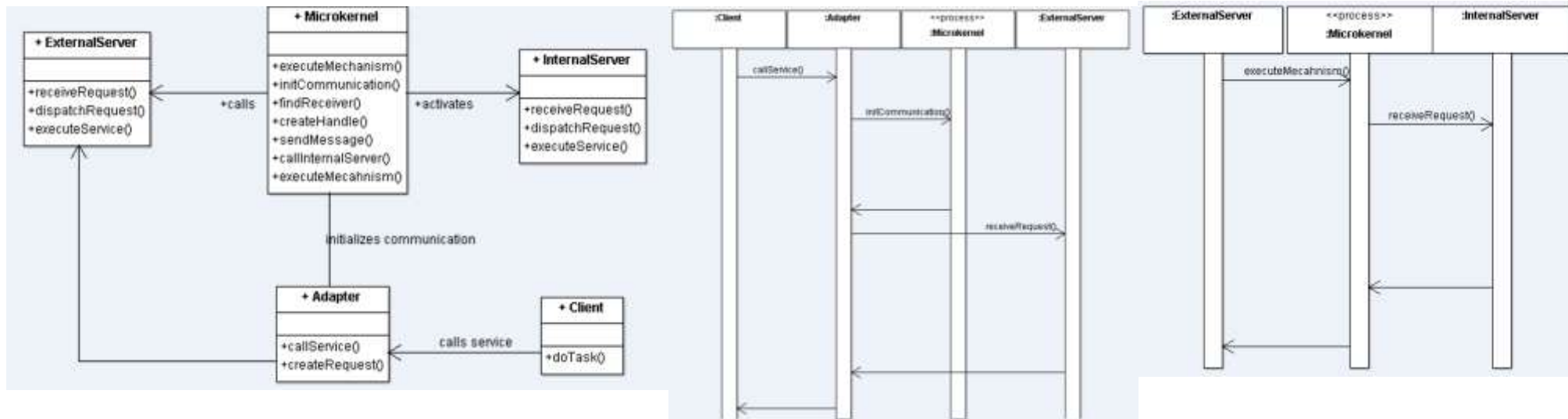


Microkernel

- ◆ A core system is a barebone system
- ◆ All other features are provided as plug-in



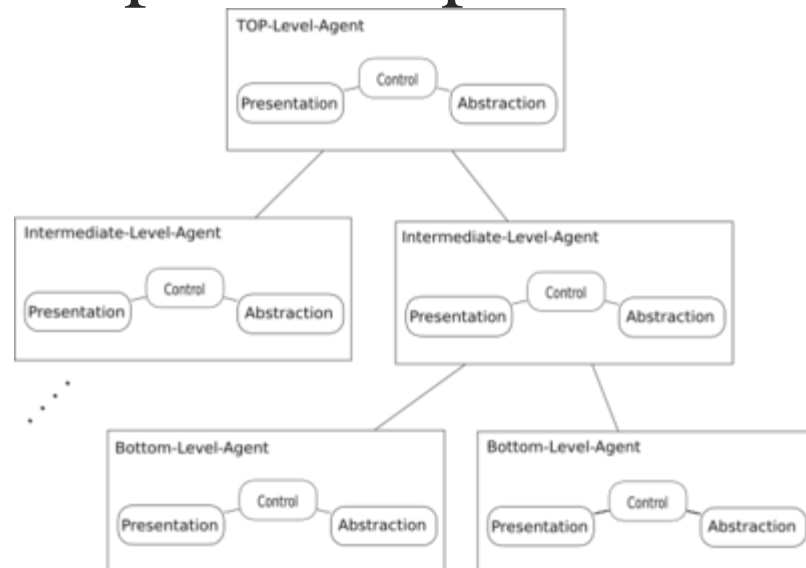
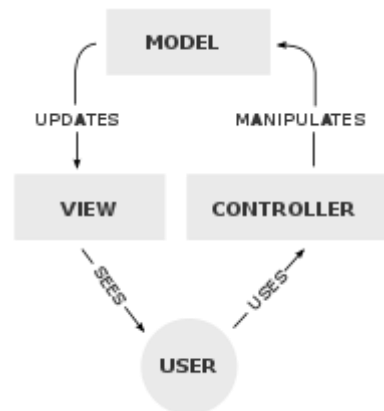
Microkernel



- *Microkernel* provides the core set of functionalities.
- *InternalServer* adds additional important functionalities to the microkernel.
- *ExternalServer* implements higher level functionality derived from the functionality offered by the microkernel.
- *Adapter* bridges between the user (client) and the system (microkernels and external servers) to handle heterogeneous system components.
- **Home reading:** Read the following blog about the advantages and disadvantages of microkernels: <https://www.guru99.com/microkernel-in-operating-systems.html>

Model-View-Controller

- ◆ Popularity used to improve user experience
- ◆ **model** : represent the core functionality and data
- ◆ **view**: displays a part of the model to the user
- ◆ **controller** : accept user input and update models



Summary

- ◆ Software architecture exists within a context (the quality attributes when delivering the functional requirements in the deployment time)
- ◆ We divide the system into modules and their connections by styles to address quality attributes (e.g., through ADD)
- ◆ There are many architectural styles
 - The successful ones are summarized as patterns
- ◆ An architectural description includes the stakeholders' concerns and the rationales of the architecture chosen. They are presented in the form of views and viewpoints.

References and Resources

1. Rick Kazman, Paul Clements, Len Bass. Software Architecture in Practice, Third Edition.
■ 2nd Edition is here: <http://www.ece.ubc.ca/~matei/EECE417/BASS/>
2. 4+1 view model https://en.wikipedia.org/wiki/4%2B1_architectural_view_model
3. Microservice <http://microservices.io>
4. Micro frontend <https://micro-frontends.org>
5. Serviceless architecture <https://martinfowler.com/articles/serverless.html>
6. ISO/IEC/IEEE 42010 Systems and software engineering — Architecture description
<https://ieeexplore.ieee.org/document/6129467>
7. Hans van Vliet https://en.wikipedia.org/wiki/Hans_van_Vliet
8. The summary, pros and cons of emerging patterns
<https://www.slideshare.net/VincentComposieux/architecture-81270165>
9. Software Architecture Design Game: <http://smartdecisionsgame.com/>
10. E.W. Dijkstra, *Communications of the ACM*. **11** (3): 147–148, 1968. doi:[10.1145/362929.362947](https://doi.org/10.1145/362929.362947)
11. A Complete Survey on Software Architectural Styles and Patterns,
<https://doi.org/10.1016/j.procs.2015.10.019>
12. Enterprise Integration Patterns, <https://www.enterpriseintegrationpatterns.com/patterns/messaging/>
13. Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media.
2020. ISBN 978-1492043454. <https://www.oreilly.com/library/view/fundamentals-of-software/9781663728357/>