Architectural Design 개요

목차

Diversity of Systems

Differences

- Scale
- Materials and technologies
- Cost

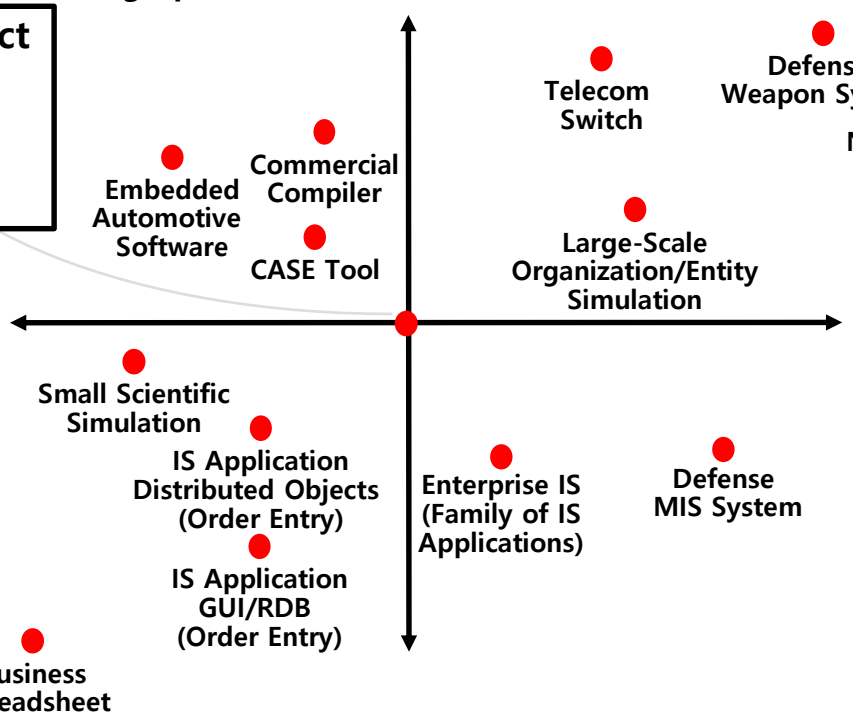
- Risks
- Stakeholders

- Process
- Budge & Schedule
- Skills and development teams

Dimensions of Software Complexity

Higher technical complexity
- Embedded, real-time, distributed, fault-tolerant
- Custom, unprecedented, architecture reengineering
- High performance

An average software project
- 5-10 people
- 3-9 month duration
- 3-5 external interfaces
- Some unknowns & risks

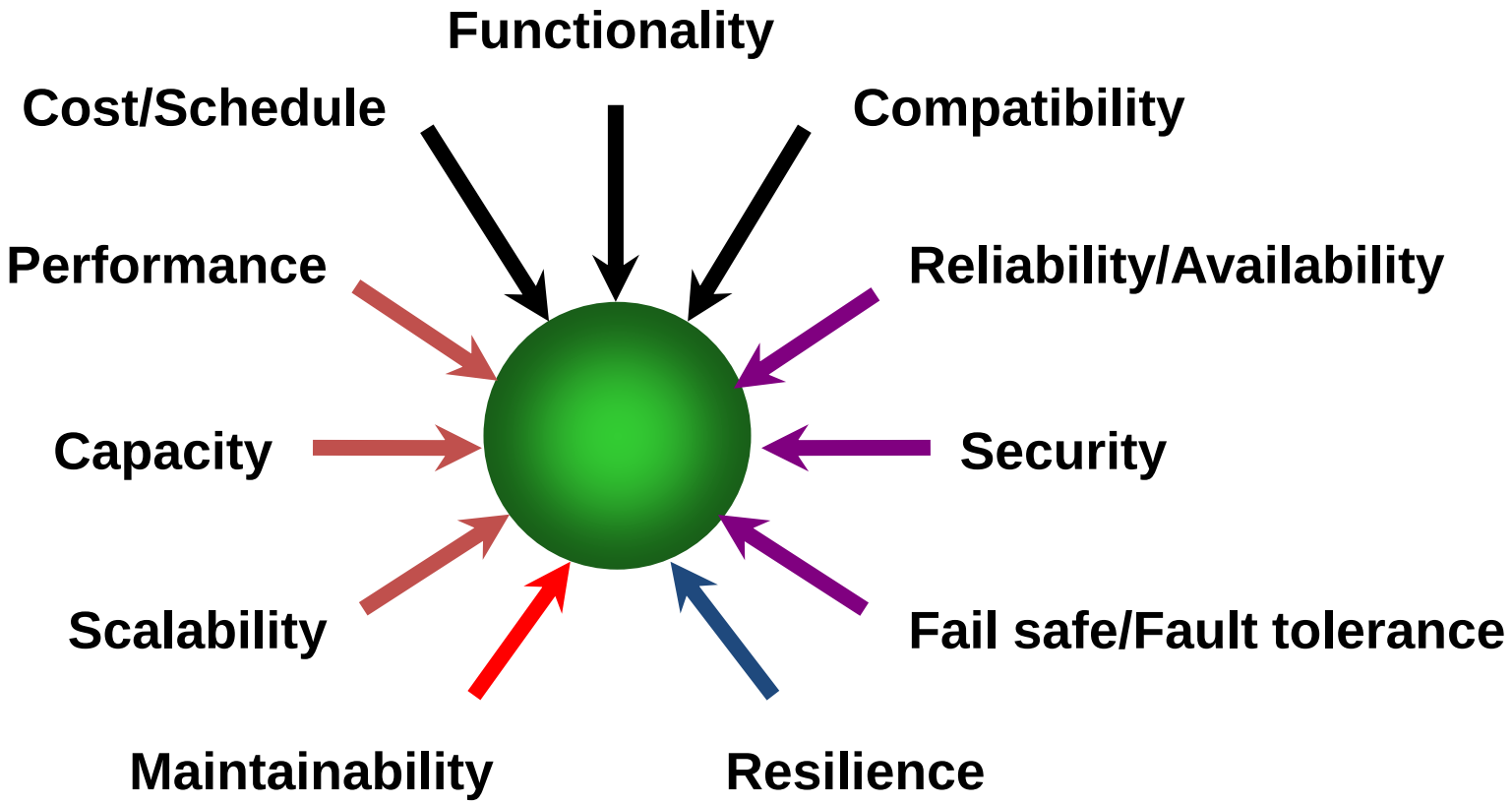


Lower management complexity
- Small scale
- Informal
- Single stakeholder
- "Products"

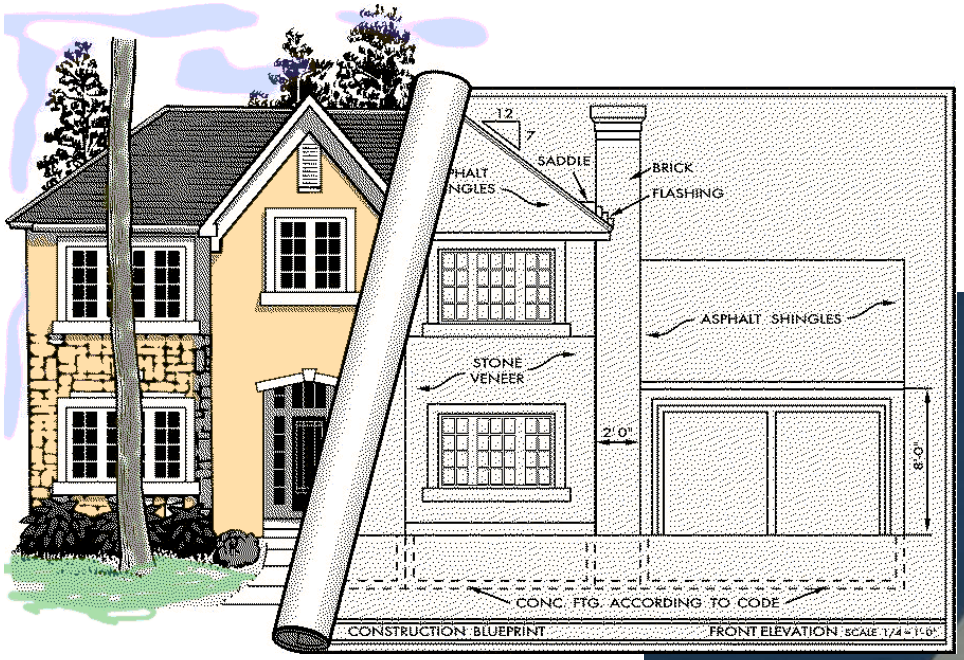
Higher management complexity
- Large scale
- Contractual
- Many stakeholders
- "Projects"

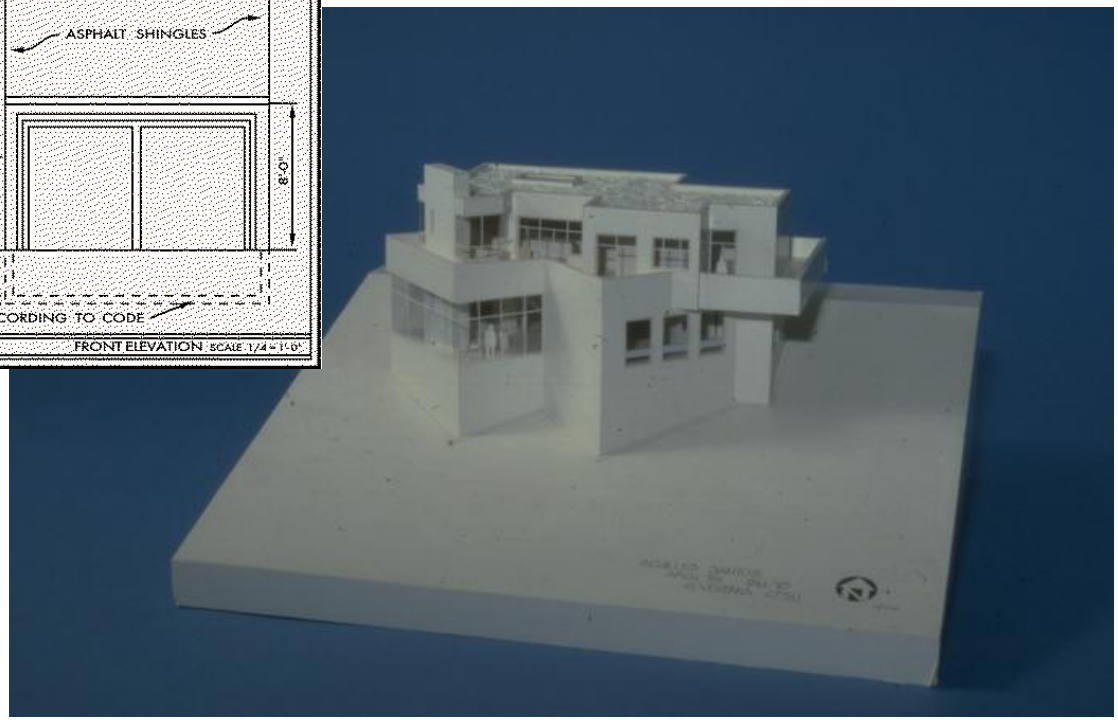
Lower technical complexity
- Mostly 4GL, or component-based
- Application reengineering
- Interactive performance

Forces in Software



Design


12
7
SADDLE
BRICK
FLASHING
ASPHALT SHINGLES
STONE VENEER
2'-0"
8'-0"
CONC. FTG. ACCORDING TO CODE
CONSTRUCTION BLUEPRINT
FRONT ELEVATION SCALE 1/4"=1'-0"



Software Design

❖ The process of defining the software elements(**components)** and **interfaces** based on the requirements

Requirement Analysis

Requirements Specification

Design

Requirements (Needs/ Expectations)

이해관계자 (고객, 사용자, …)

Design Model

Construction / Implementation

Product or System

❖ Kinds of software elements

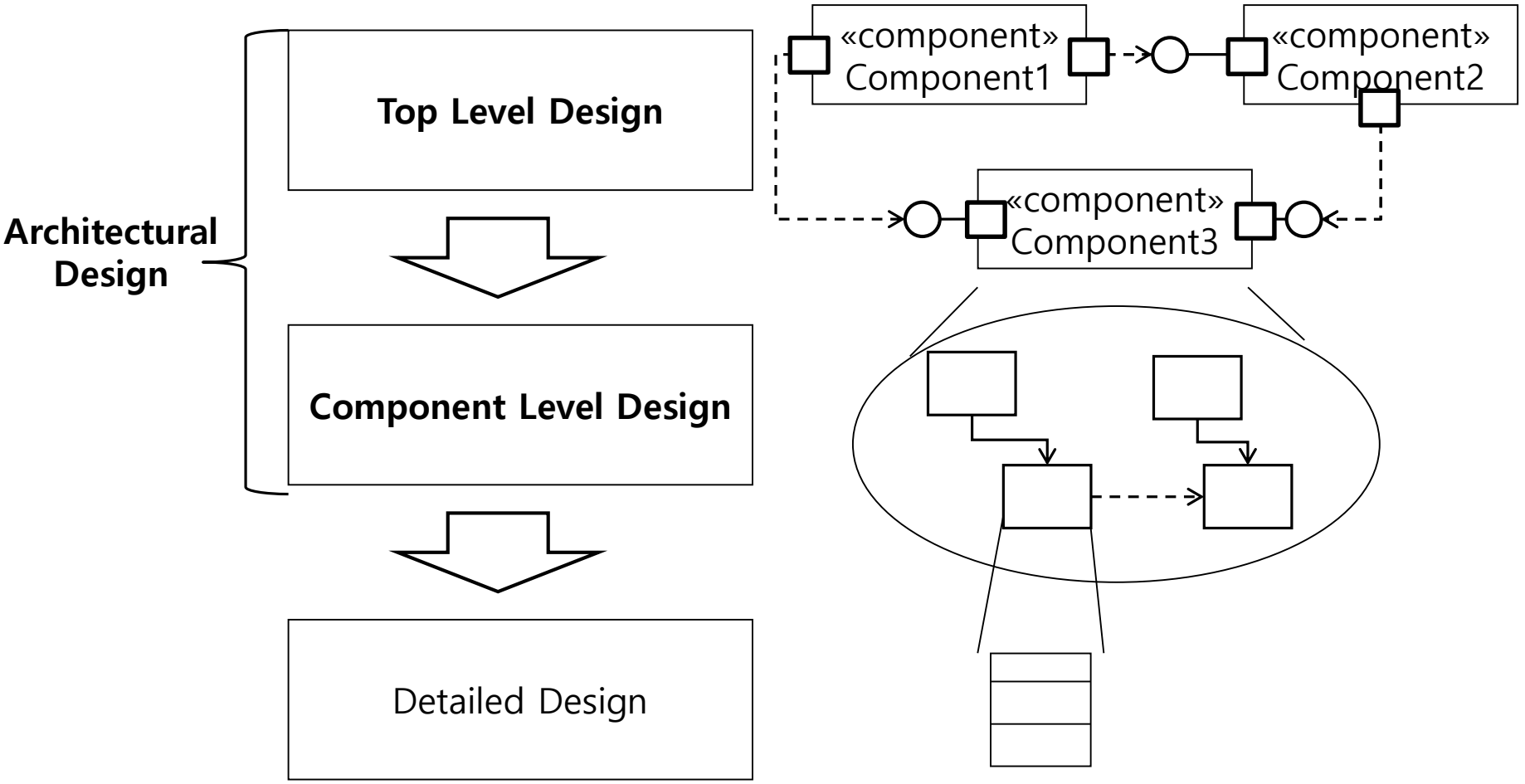
Software Design

❖ The process of

- creating a **specification of software elements**,
  - ✓ Node(Infrastructure view), Component(Structure view), Component instance(Behavior view), Artifact(Deployment view)
- intended to accomplish **goals**,
  - ✓ Functionalities, quality attributes(performance, availability, …)
- subject to **constraints**
  - ✓ **Technical constraints, business constraints**

Typical Design Process

❖ In general, design encompasses three sub-processes.



Focuses of Sub-process

Design Phase	Scope	Design Decisions (What is Specified)
Top Level Design	**System-wide**	• Types and number of nodes • Components(artifacts) running on each node • Communication strategies • Components layering and vertical slices • Global error-handling policies
Component Level Design	**Inter-class**	• Multiple collaborating objects • Design-level classes and objects • Medium-level error-handling policies
Detailed Design	Intra-class	• Details of data members(types, ranges) • Details of member functions (arguments, internal structure, algorithm)

ARCHITECTURAL DESIGN

What is Architecture

The software architecture of a system is the set of structures

Views: Structure, Behavior, …

- needed to reason about the system,

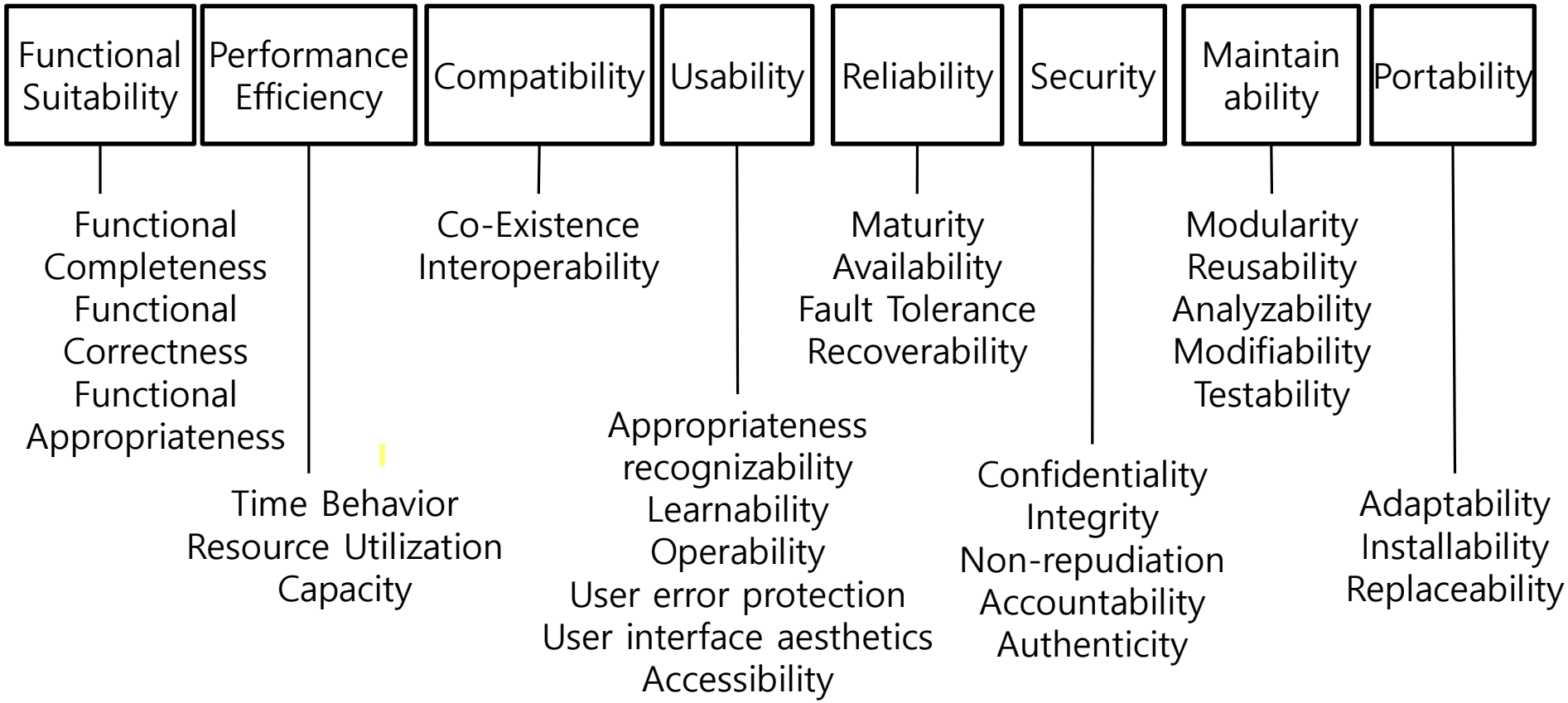
Functionalities, Quality, Managerial Aspects

- which comprise software elements, relations among them, and properties of both.

Depends on View

- Software Architecture in Practice, 3rd edition

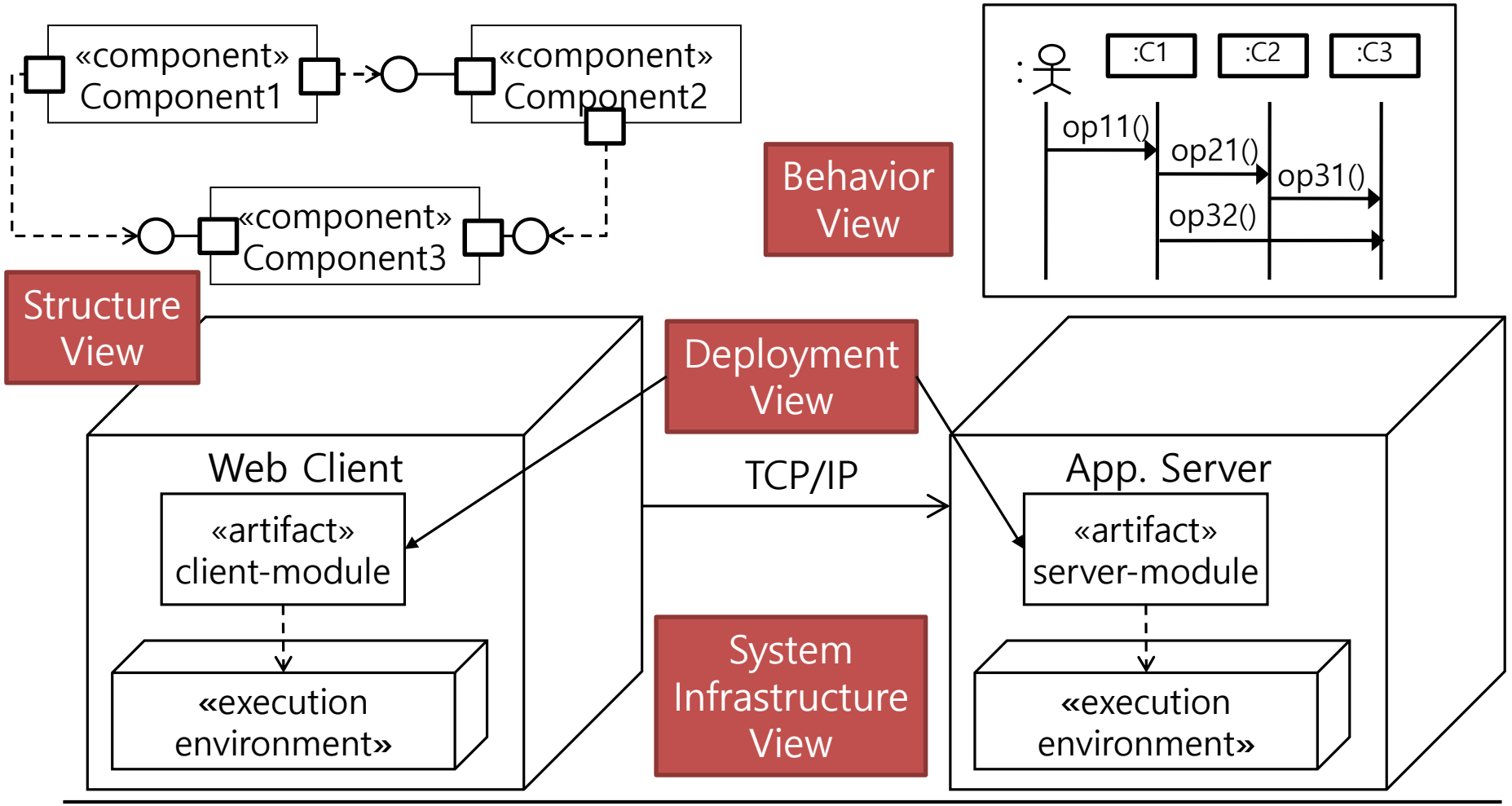
ISO/IEC 25010:2011 Quality Model



ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models

Views of Architectural Design

❖ Multiple views are necessary to represent the whole architecture



Importance of Architectural design

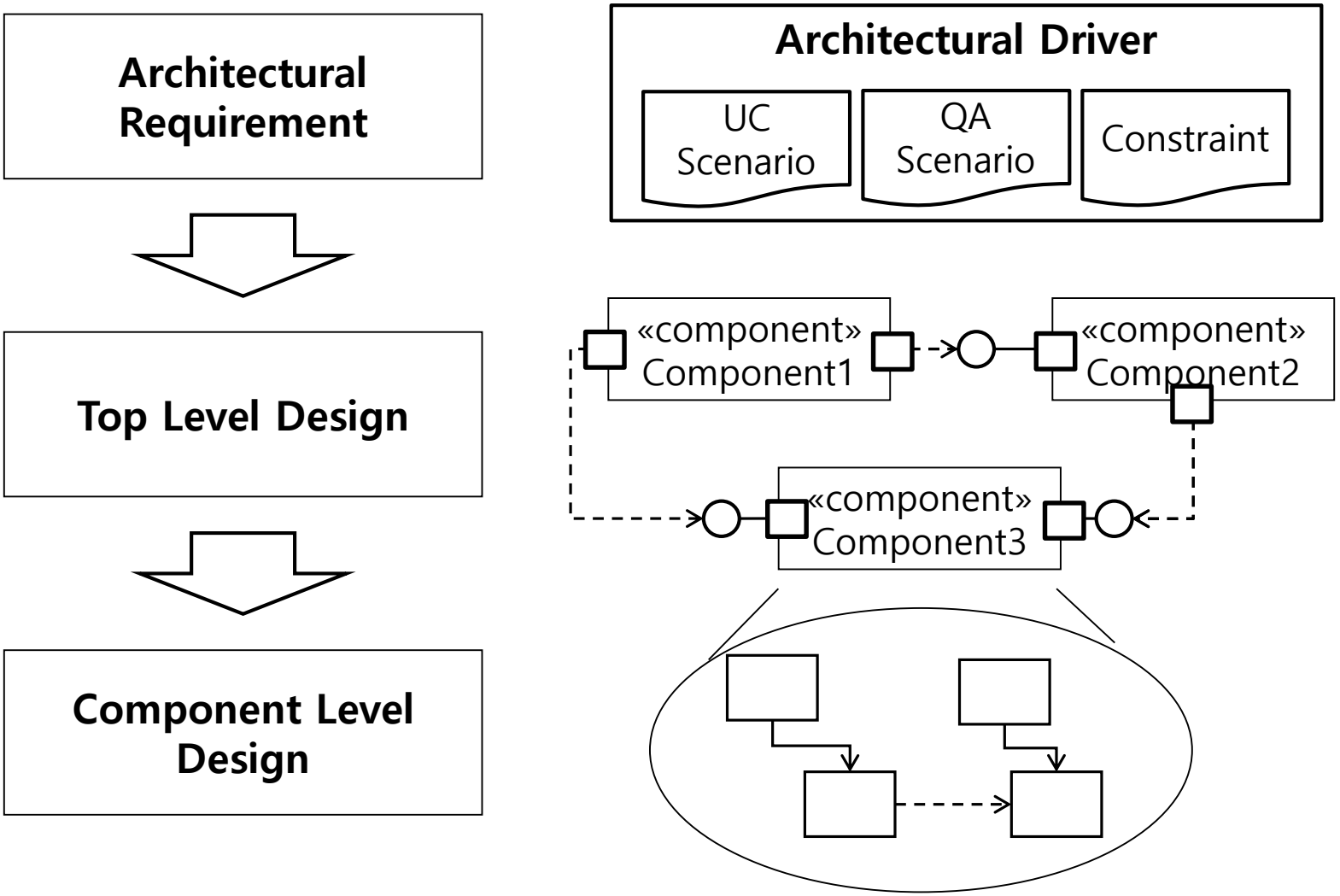
- Without doing some architectural thinking and some early design work, you cannot confidently predict project cost, schedule, and quality.

- A well-designed, properly communicated architecture is key to achieving *agreements* that will guide the team. The most important kinds to make are agreements on interfaces and on shared resources.

- Architecture is a key enabler of agility, if you do not make some key architectural decisions early and if you allow your architecture to degrade, you will be unable to maintain sprint velocity because you cannot easily response to change requests

- The architecture will influence design decisions (e.g. Selection of tools, structuring of development environment) and vice versa

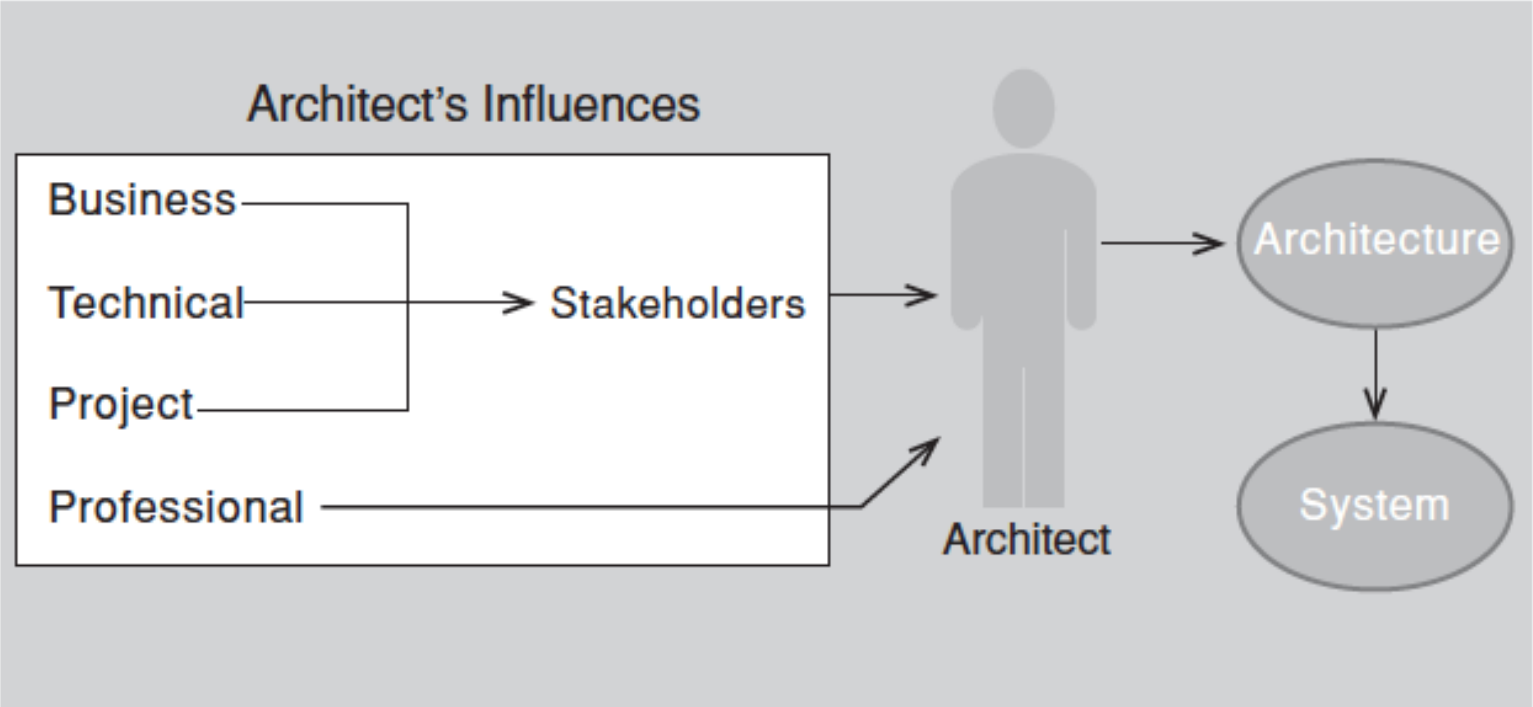
Architectural Design Process



ARCHITECTURAL DRIVERS

How is Architecture Influenced?

- A software architecture is a result of business(business goal) and social influences(business constraint), as well as technical ones(functional requirements, QAs)



Software architecture in practice, 3rd edition(2012)

What is Architectural Driver?

- To begin designing the architecture of a software-intensive system, architects need the key requirements that are most likely to affect the fundamental structure of the system.

- These key requirements will determine the structure of the system; they are the architectural drivers.

- Architectural drivers are not all of the requirements for a system, but they are an early attempt to identify and capture those requirements that are most influential to the architect making early design decisions.

- Uncovering the architectural drivers as early as possible is critical because these early architectural decisions are binding for the lifetime of a system

Architecting software intensive systems-A practitioner's guide(2008)

Architectural Drivers

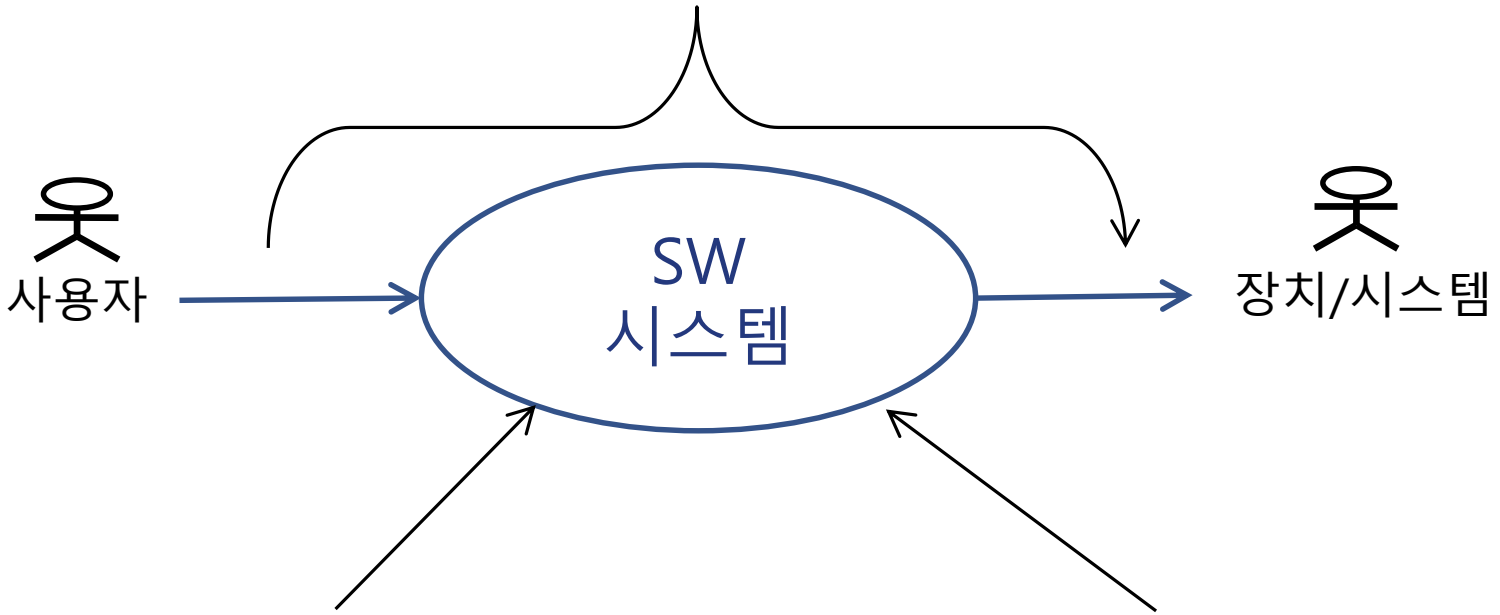
❖ Architectural drivers consist of

- coarse-grained or **high-level** functional requirements,
- quality attribute requirements,
- and technical constraints, business constraints

Driver	Description
Functional requirements	Those general requirements for what the system must do
Quality attributes	Properties that the system must possess, such as availability, security, high performance, and so forth
Constraints	Fixed premade decisions that are in place before design begins

ADs are subset of Requirements

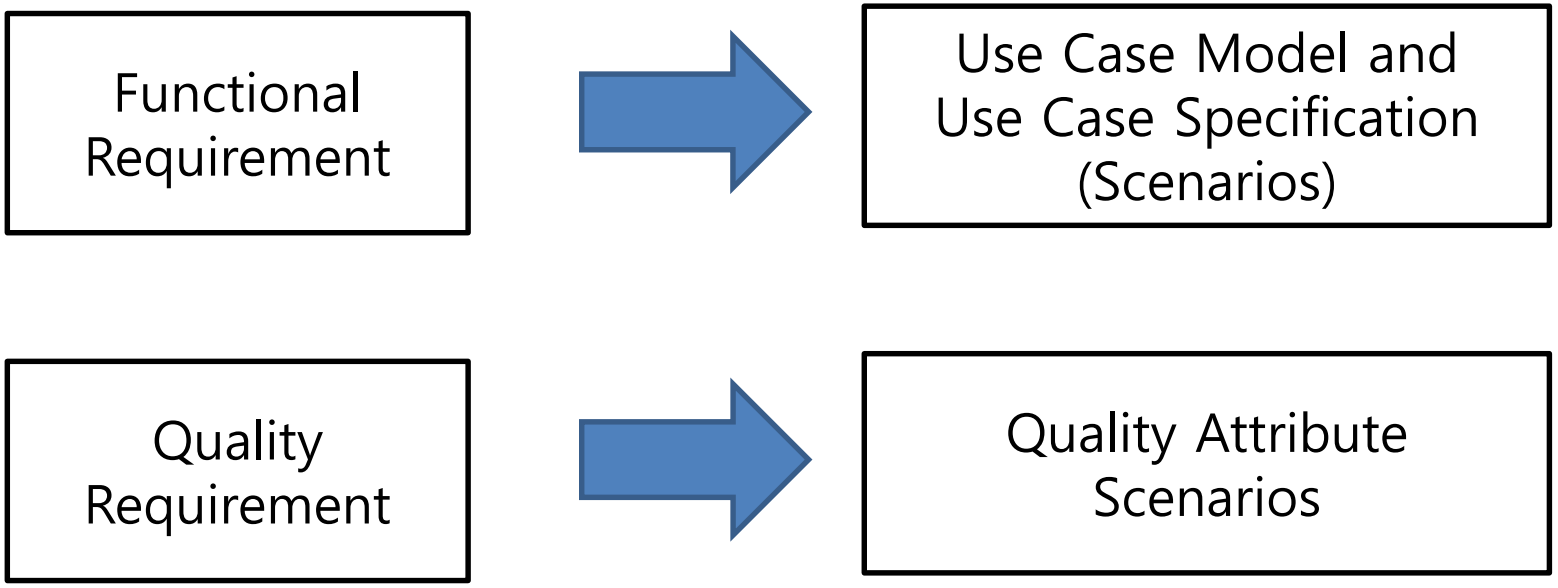
기능적 요구사항
소프트웨어에 주어진 입력에 따른 동작 및 출력에 대한 요구사항



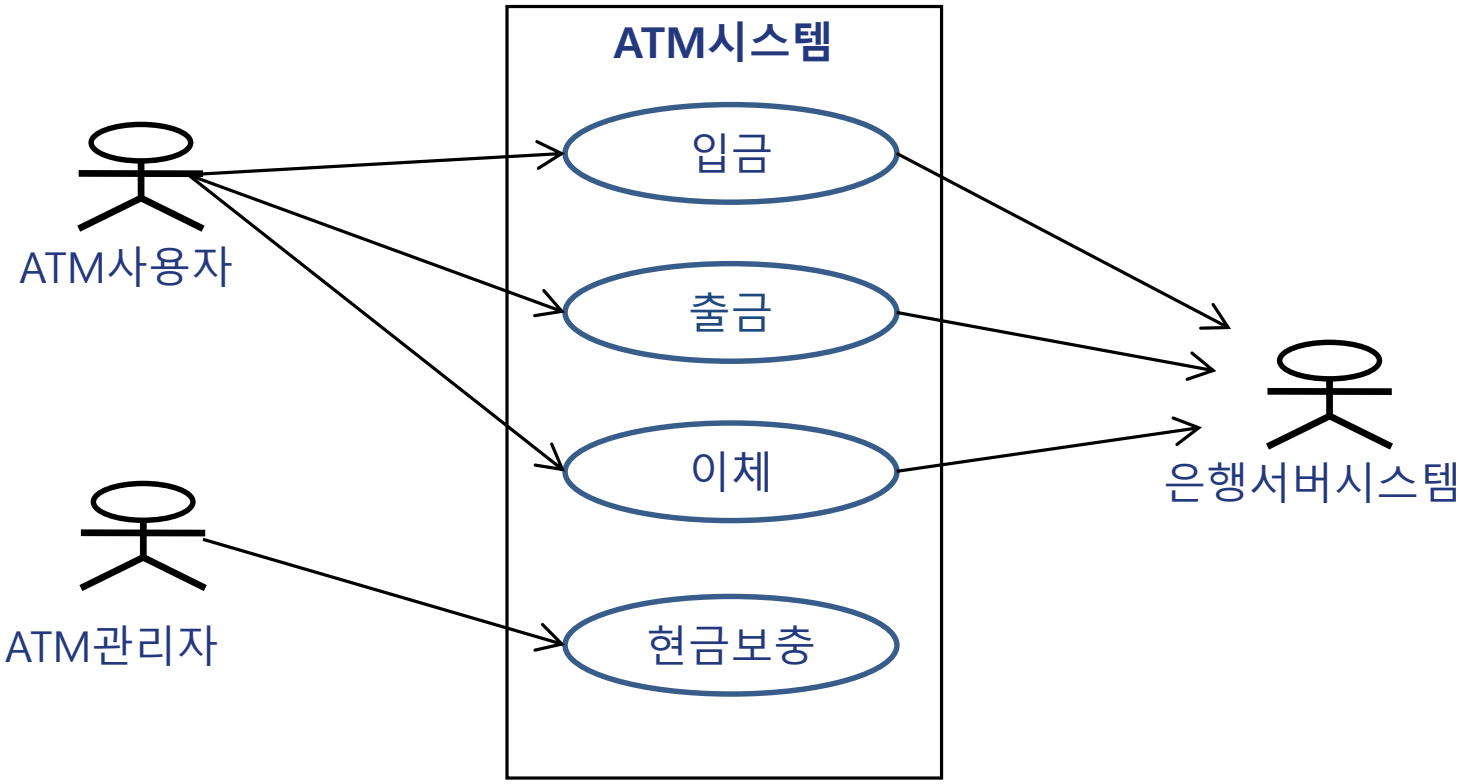
품질 요구사항
성능, 보안, 안전성, 가용성 등과 같은
기능 요구사항에 대한 제약

제약사항
외부 인터페이스에 대한 제약
법규, 표준 등에 대한 준수
설계, 구현, 설치, 운영 등에 대한 제약

Capturing and Describing ADs



Use Case Model: Example



Use Case Scenario

❖ 출금 유스케이스의 시나리오 명세 예

1. **ATM사용자**는 카드입력 장치에 카드를 삽입한다.
2. **시스템**은 삽입된 카드를 판독한다.
3. 시스템은 메뉴 화면을 출력한다.
4. ATM사용자는 "출금"을 선택한다.
5. 시스템은 암호 입력 화면을 출력한다.
6. ATM사용자는 암호를 입력한다.
7. 시스템은 입력된 암호의 정확성을 점검한다.
8. 시스템은 출금 금액 입력 화면을 출력한다.
9. ATM사용자는 인출금액을 입력한다.
10. 시스템은 **은행서버시스템**에게 출금요청을 한다.
11. 은행서버시스템은 요청된 출금에 대한 처리 결과를 시스템에게 통보한다.
12. 시스템은 카드와 지폐를 배출하고, 영수증은 인쇄한다.
13. ATM사용자는 카드, 지폐, 영수증을 수령한다.
14. 시스템은 지폐 배출 문을 닫는다.

Functional Requirement vs Quality Requirement

- In practice, quality attribute requirements and functionality are usually intimately intertwined.
  - It is impossible and meaningless to say a system "shall have high performance."
  - Without associating the performance to some specific behavior in the system, architects cannot hope to design a system to satisfy this need.

Architecting software intensive systems-A practitioner's guide(2008)

- Suppose a functional requirement: "The game shall change view modes when the user presses the <C> button"
  - Performance: How fast should the function be?
  - Security: How secure should the function be?
  - Modifiability: How modifiable should the function be?

Designing software architecture-A practical approach(2016)

Quality Requirement Description

❖ A system will be "modifiable" ➔ too ambiguous

- because every system is modifiable with respect to some changes and not modifiable with respect to others.

❖ Quality attributes

- Measurable or testable properties of a system
- that are used to indicate how well the system satisfies the needs of its stakeholders.

❖ How to express the qualities unambiguously?

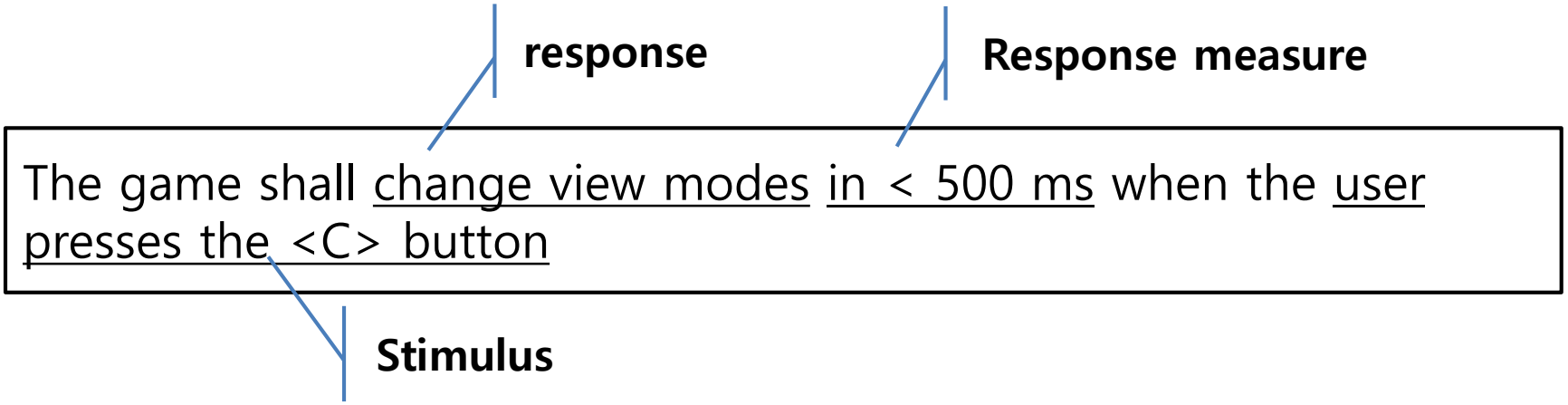
❖ Solution: quality attribute scenarios

Software architecture in practice, 3rd edition(2012)

Brief Quality Attribute Scenario

- A short description of how(**measure**) a system is required to respond(**response**) to some event(**stimulus**).

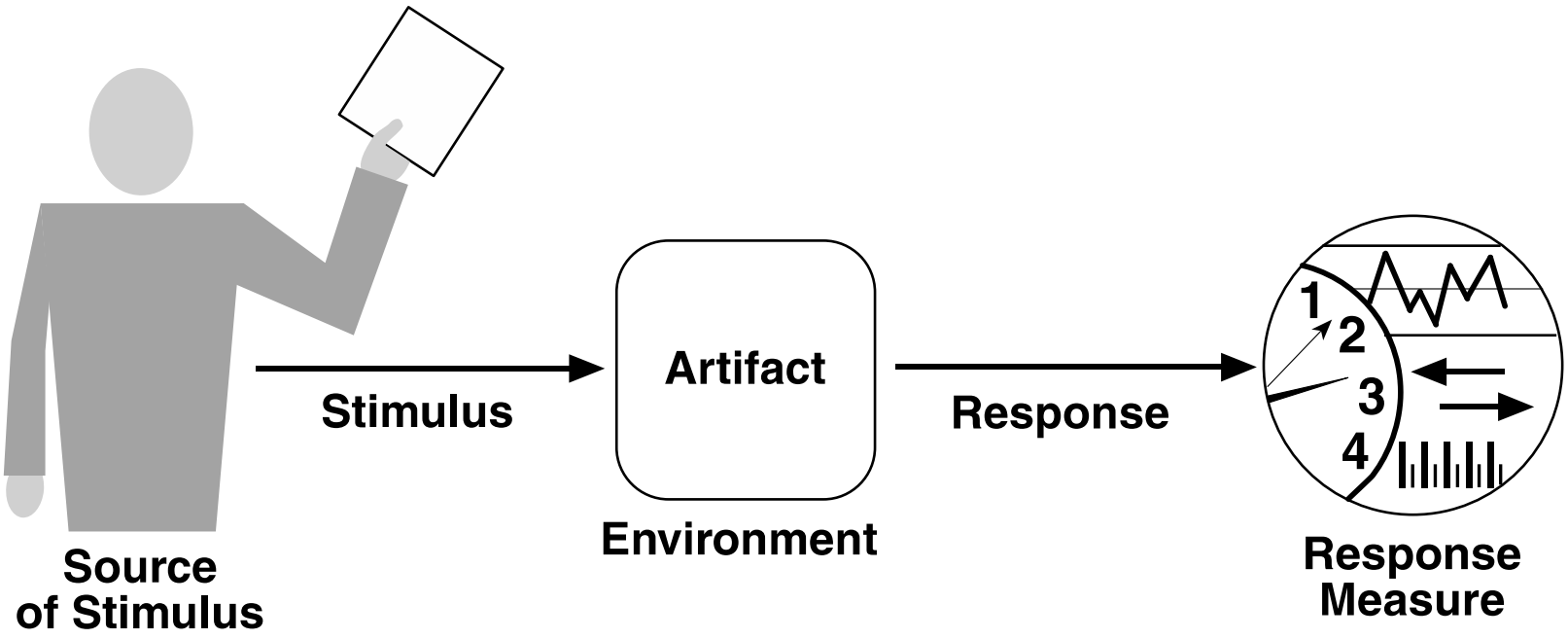
- e.g.) performance scenario



Designing software architecture-A practical approach(2016)

Complete Quality Attribute Scenarios

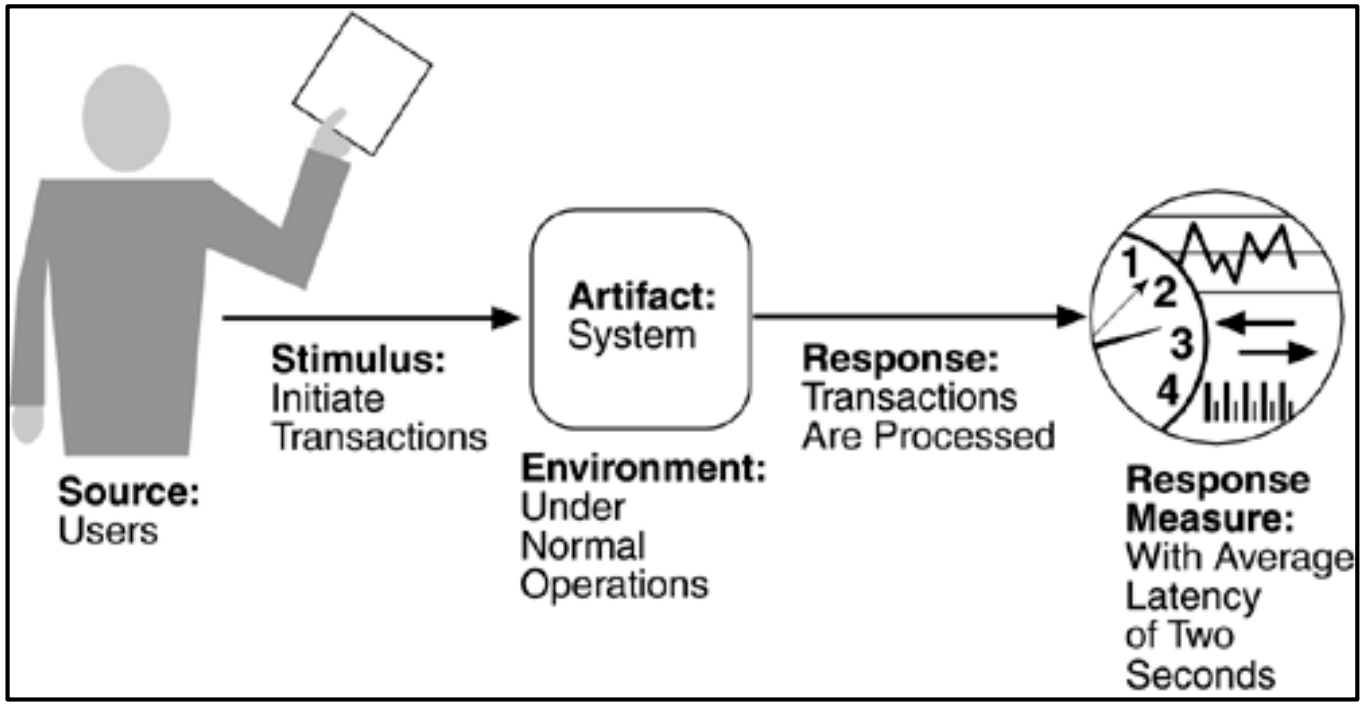
❖ There are six parts of a complete QA scenario



Sample Performance Scenario

- Users<**Source**> initiate transactions<**Stimulus**> under normal operations<**Environment**>.

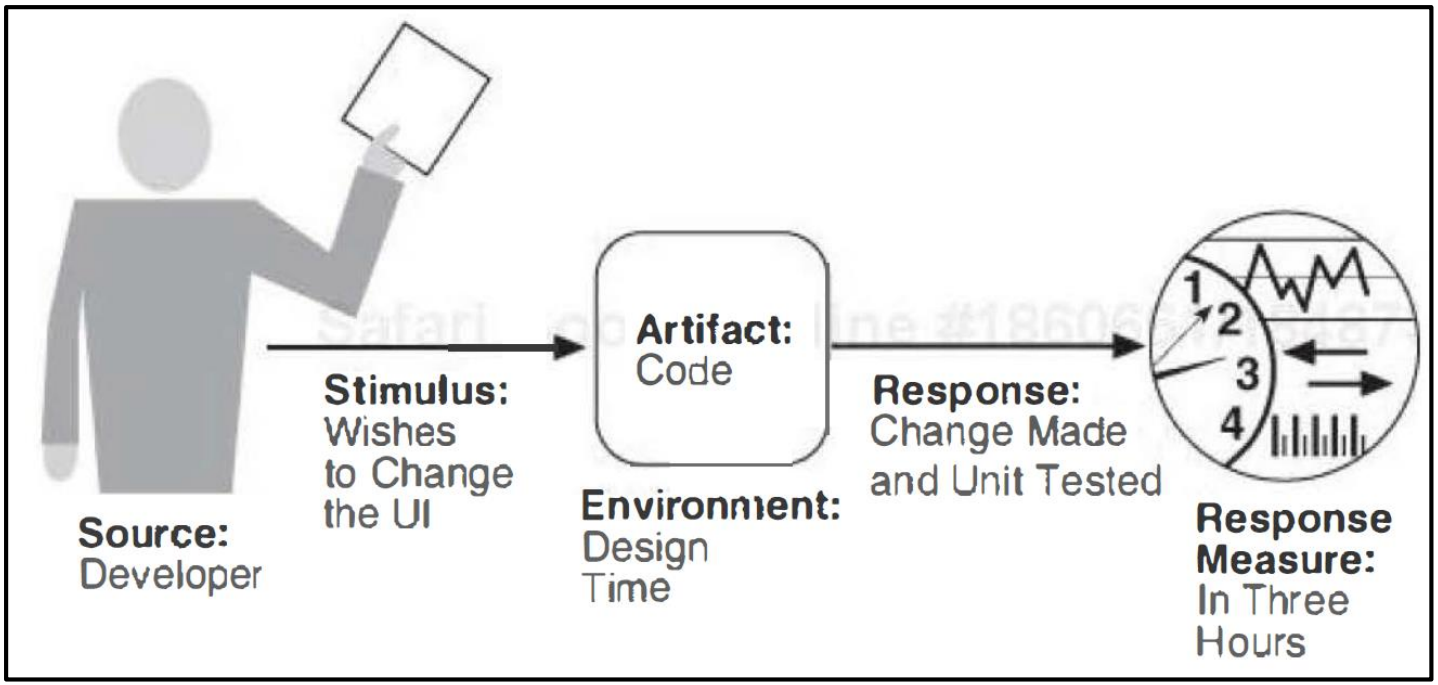
- The system processes the transactions<**Response**> with an average latency of two seconds<**Response Measure**>.



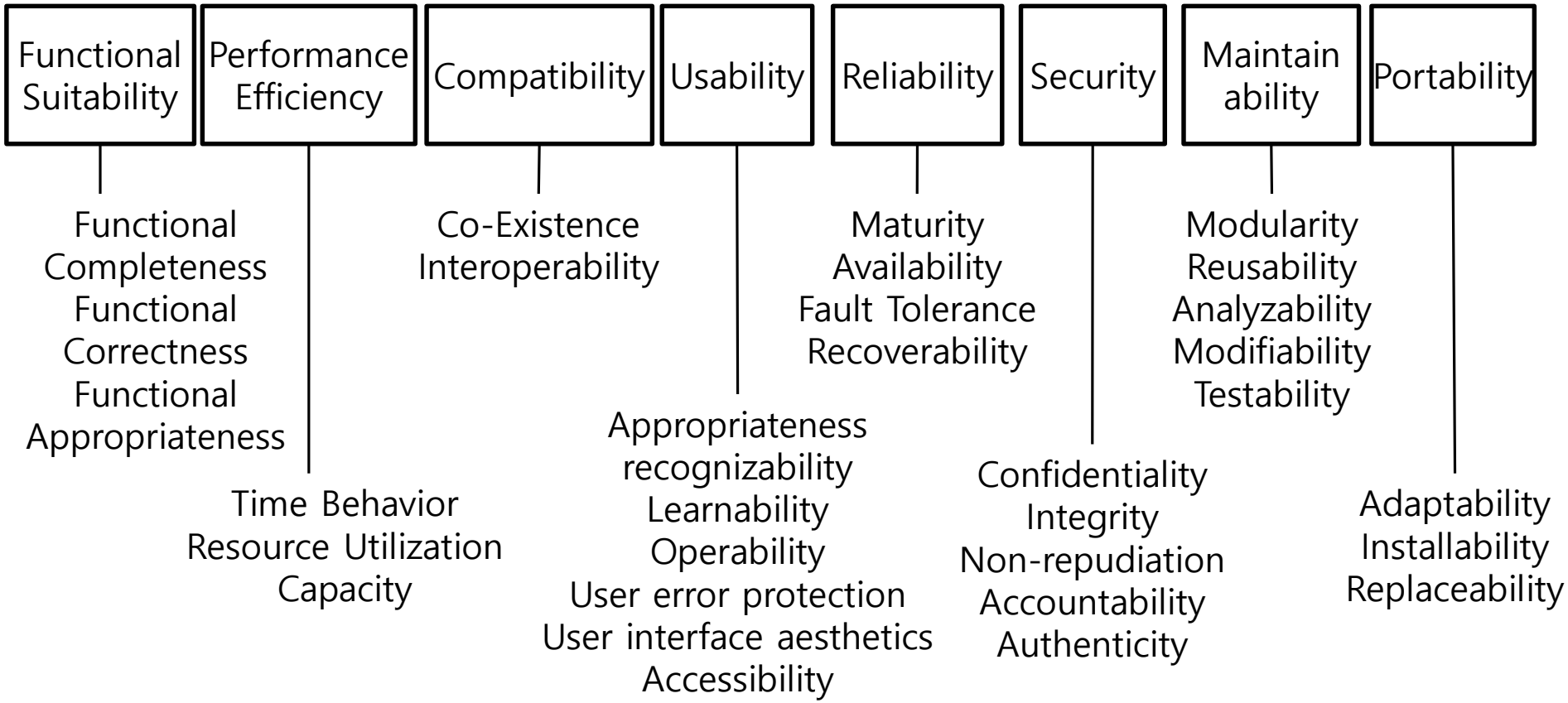
Sample Modifiability Scenario

- The developer<**Source**> wishes to change the user interface<**Stimulus**> by modifying the code at design time<**Environment**>.

- The modifications are made with no side effects <**Response**> within three hours<**Response Measure**>.



ISO/IEC 25010:2011 Quality Model



ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models

Constraint

- A constraint is fixed premade decisions that are in place before design begins
  - Business constraints limit decisions about people, process, costs, and schedule.
  - Technical constraints limit decisions about the technology we may use in the software system.

- Each of these exerts forces on the architect and influences the design decisions that the architect makes

- Constraints limit choice, but well-chosen constraints simplify the problem and can make it easier to design a satisficing architecture

Design It! – From programmer to software architect(2017)

Constraint: Examples

Technical Constraints	Business Constraints
Programming Language Choice Anything that runs on the JVM.	**Team Composition and Makeup** Team X will build the XYZ component
Operating System or Platform It must run on Windows, Linux, and BeOS.	**Schedule or Budget** It must be ready in time for the Big Trade Show and cost less than $80,000.
Use of Components or Technology We own DB2 so that's your database.	**Legal Restrictions** There is a 5GB daily limit in our license

Design It! – From programmer to software architect(2017)

DESIGN VIEWS

Software Architecture

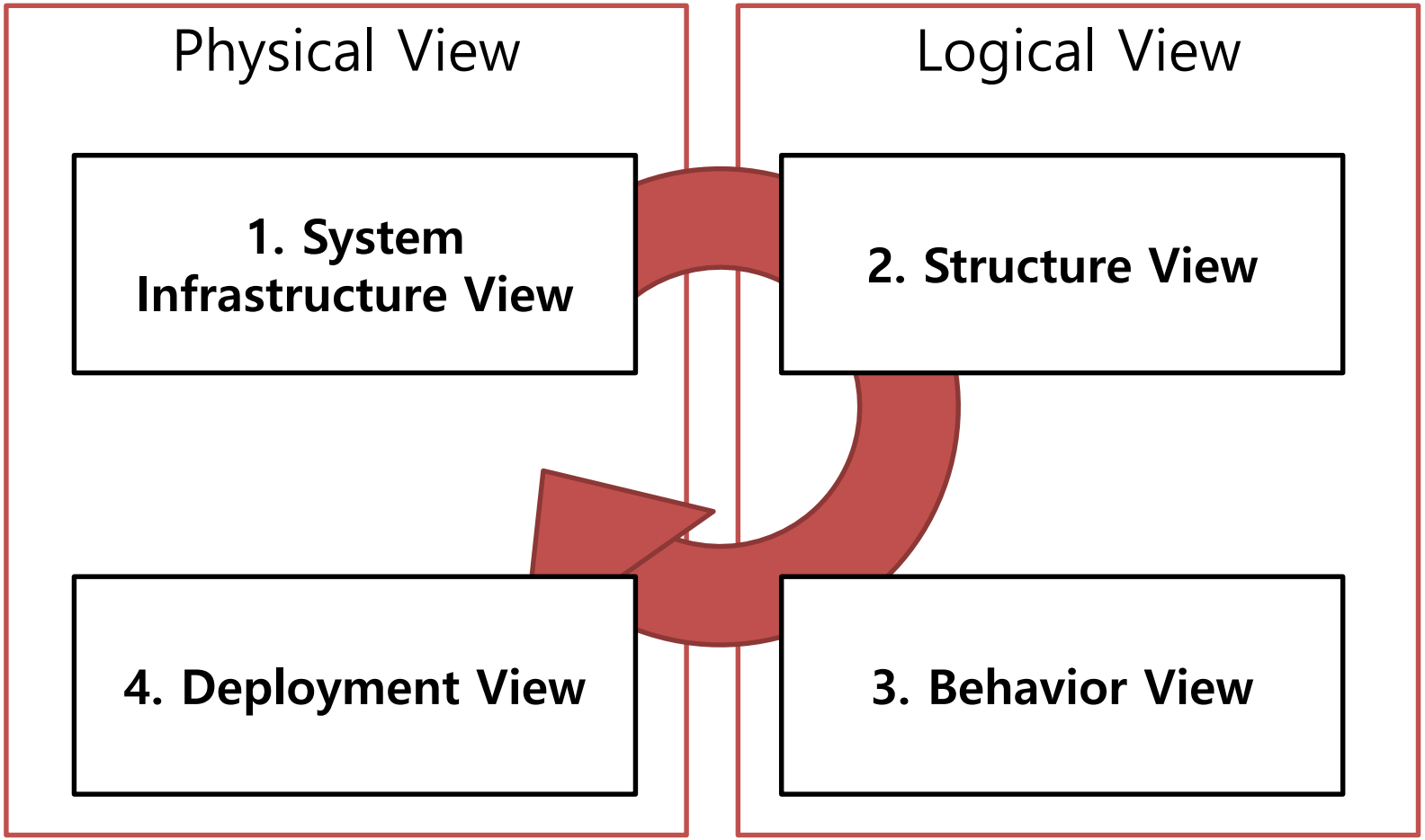
The software architecture of a system is the **set of structures**

- needed to reason about the system,
- which comprise software elements, relations among them, and properties of both.

- Software Architecture in Practice, 3rd edition

Architectural Design Description with Views

❖ Architectural design is described through multiple views.



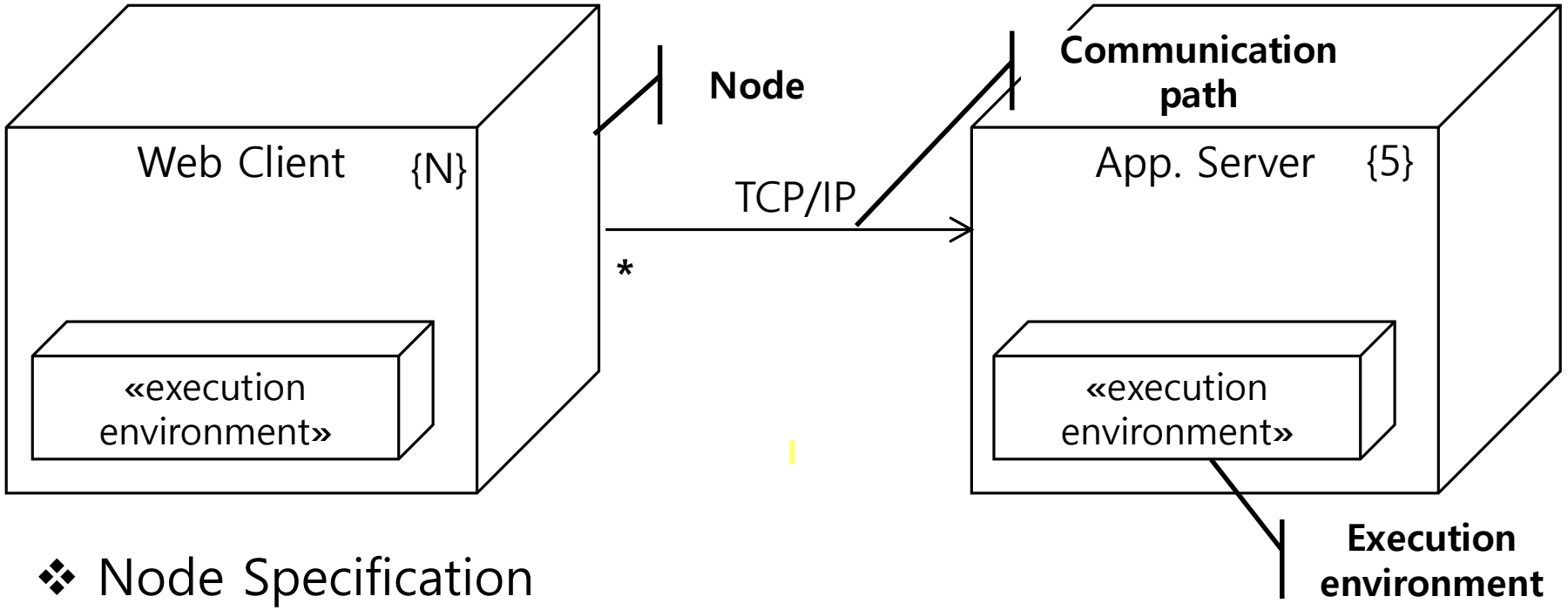
Views

❖ Design Views and Elements

View	Design Elements	UML Diagram
System Infrastructure View	Node Execution Environment Communication Path	Deployment Diagram
Structure View	Component with Port / Class Interface	Component Diagram Class Diagram
Behavior View	Lifeline(Component Instance) and Message	Sequence Diagram
Deployment View	Artifact Deployment	Deployment Diagram

System Infrastructure View

- System Infrastructure Diagram(i.e., deployment diagram)

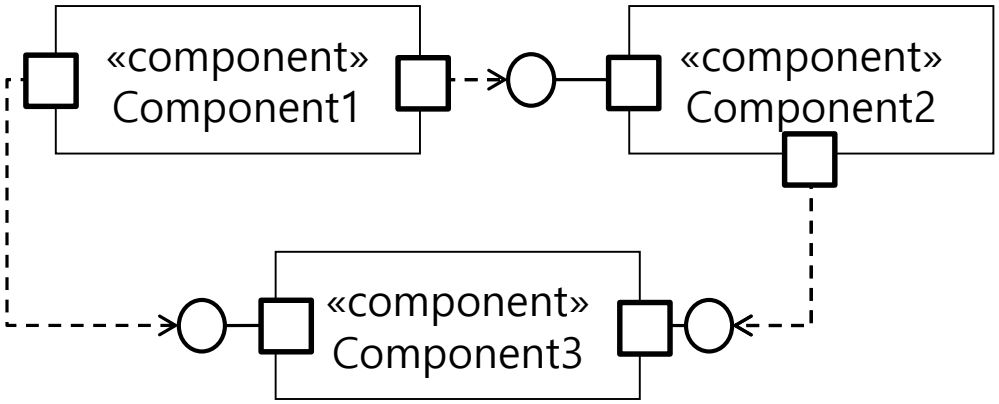


- Node Specification
- Execution Environment Specification
- Communication Path Specification

Structure View

❖ Static Structure Model

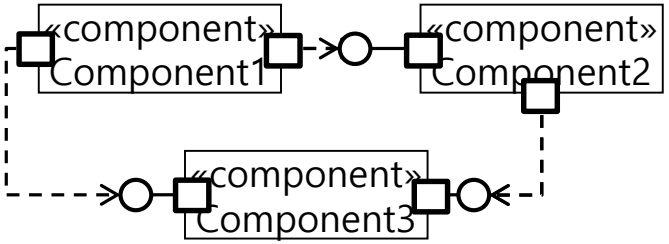
- Static Structure Diagram(i.e., component diagram)



❖ Component Specification

- Interface List
- Interface Specification

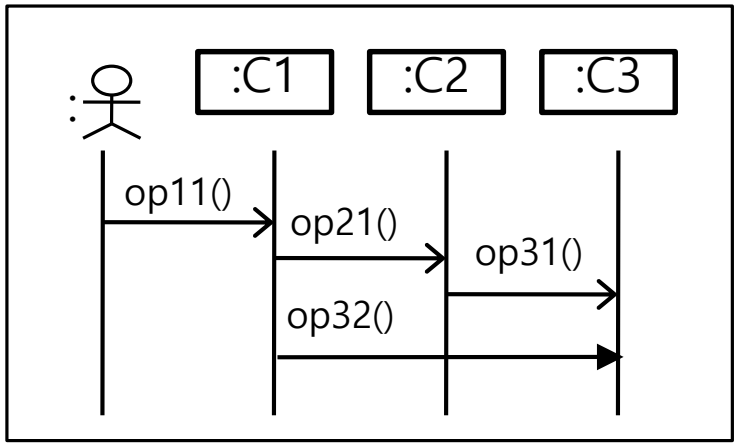
Behavior View



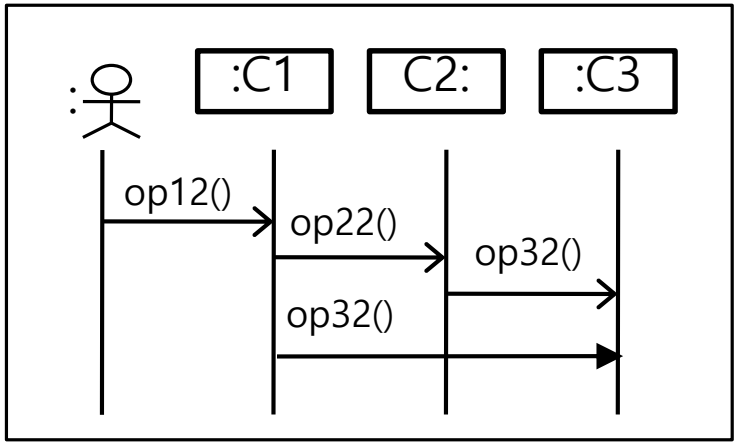
UC-01

UC-nn

❖ Use Case Behavior Model

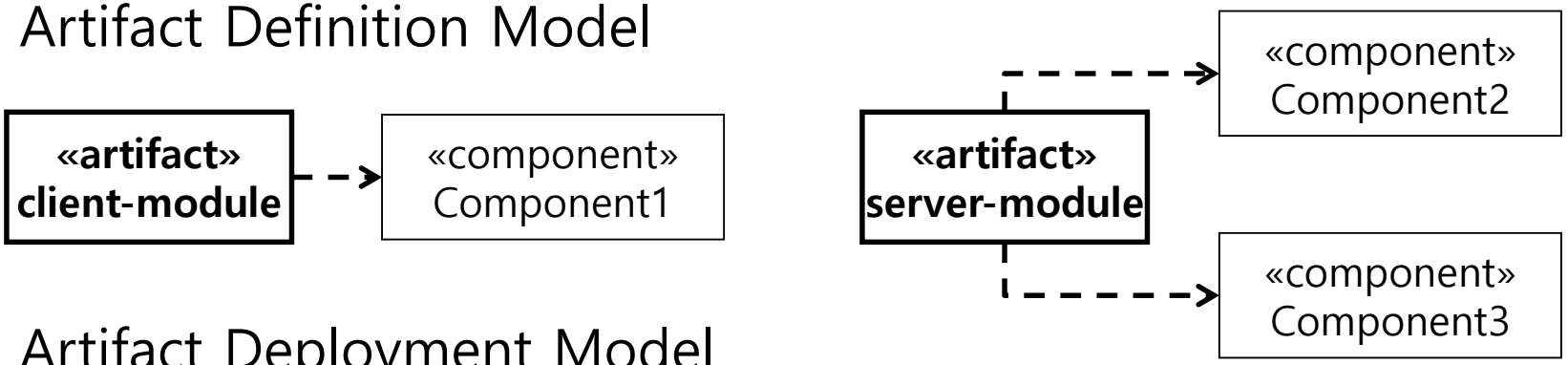


OOO

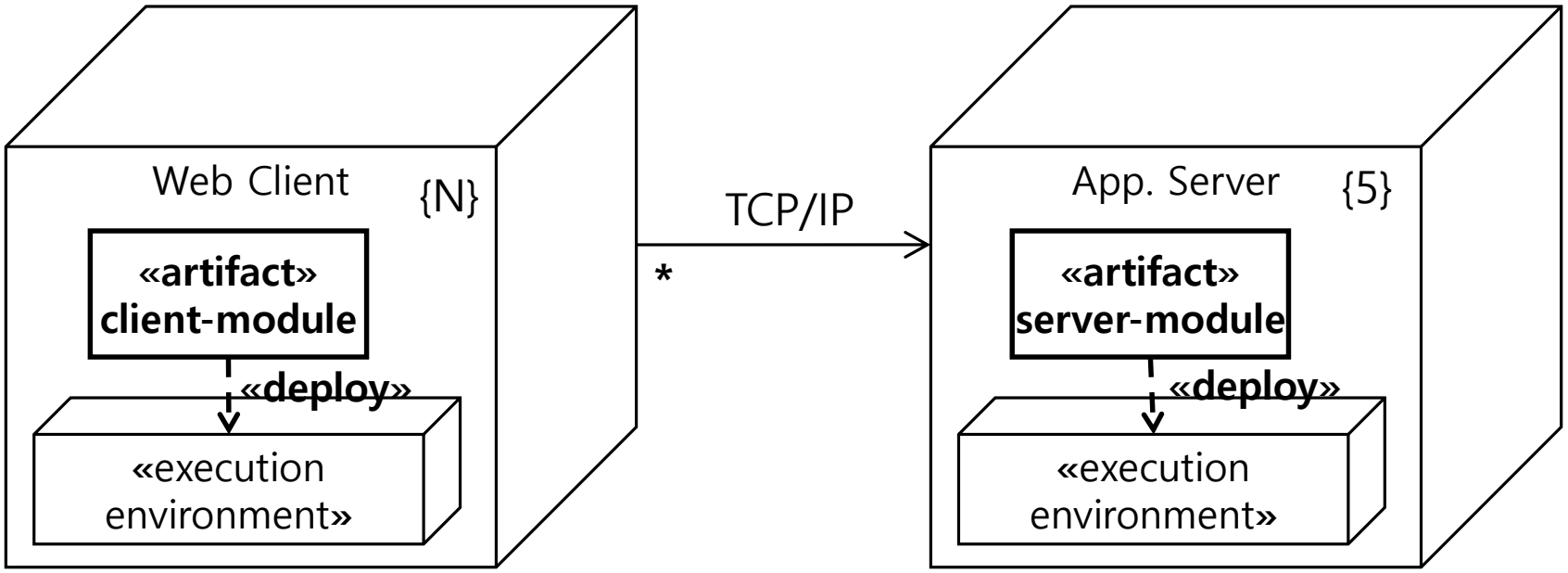


Deployment View

❖ Artifact Definition Model

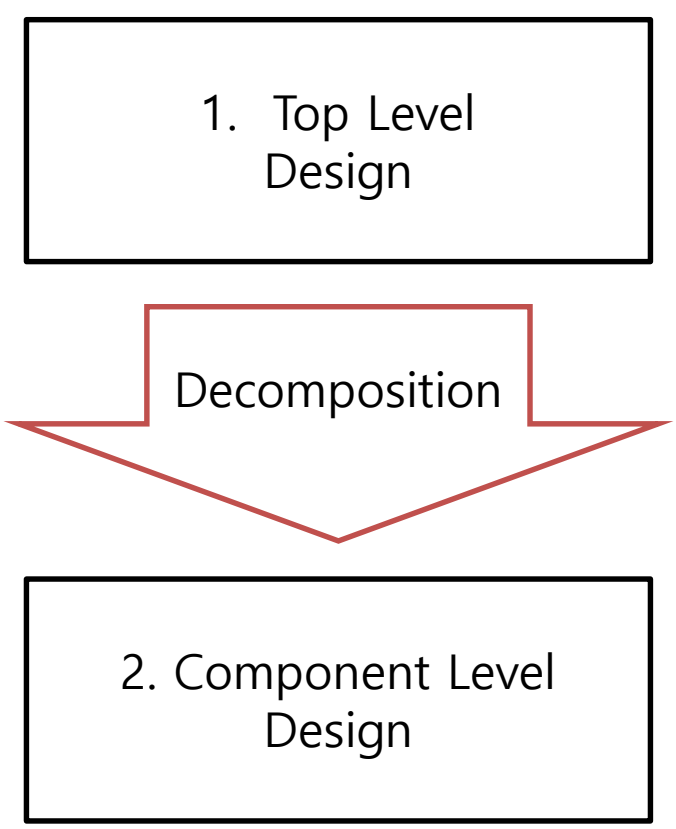


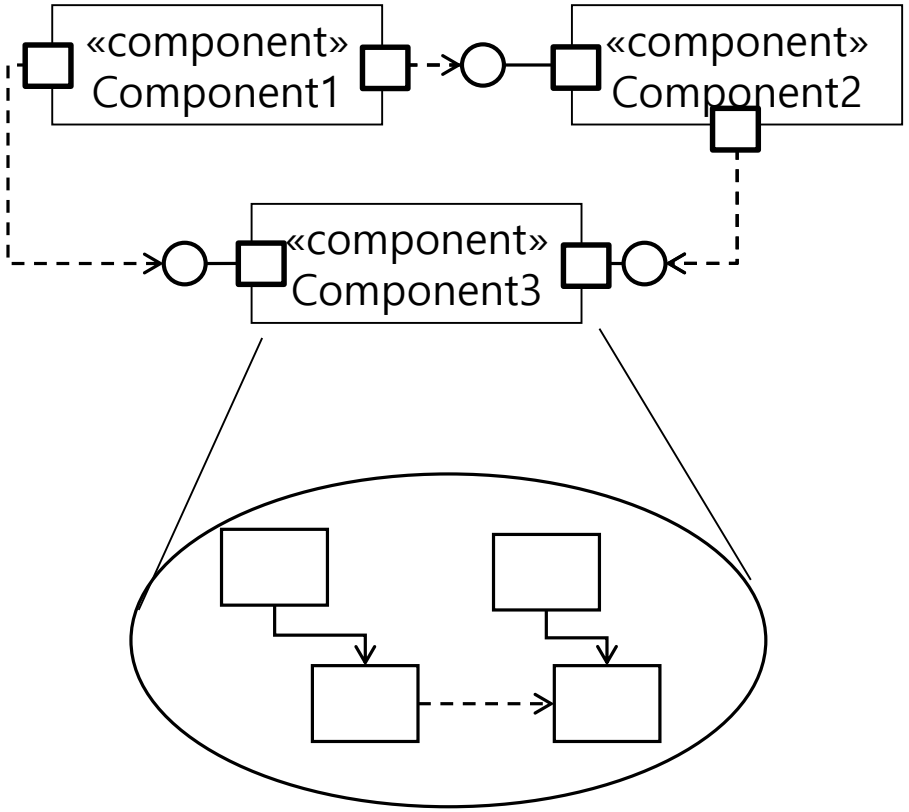
❖ Artifact Deployment Model



Architectural Design Phase

❖ A system' structures are iteratively decomposed

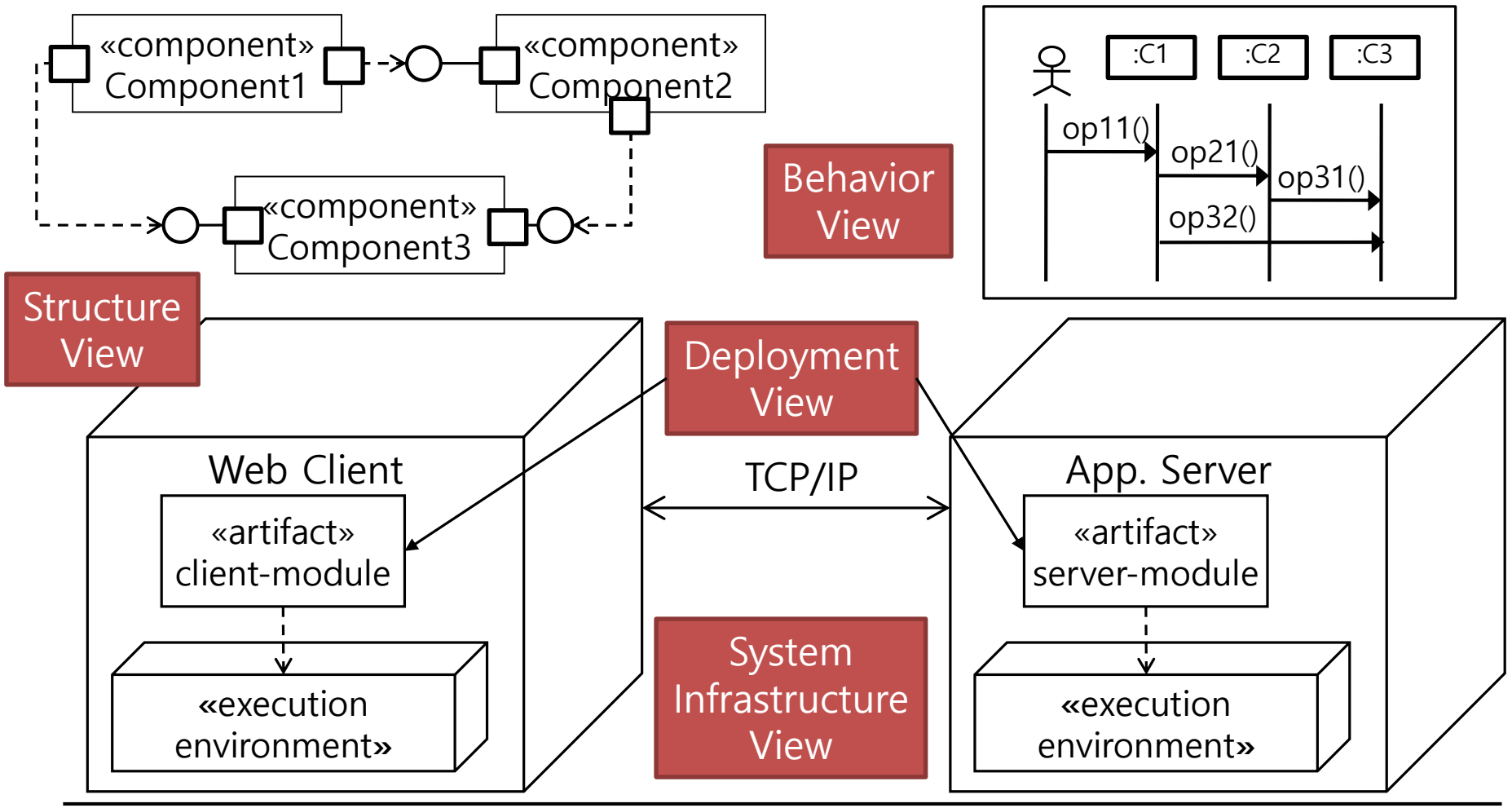




Component Level Design for Component3

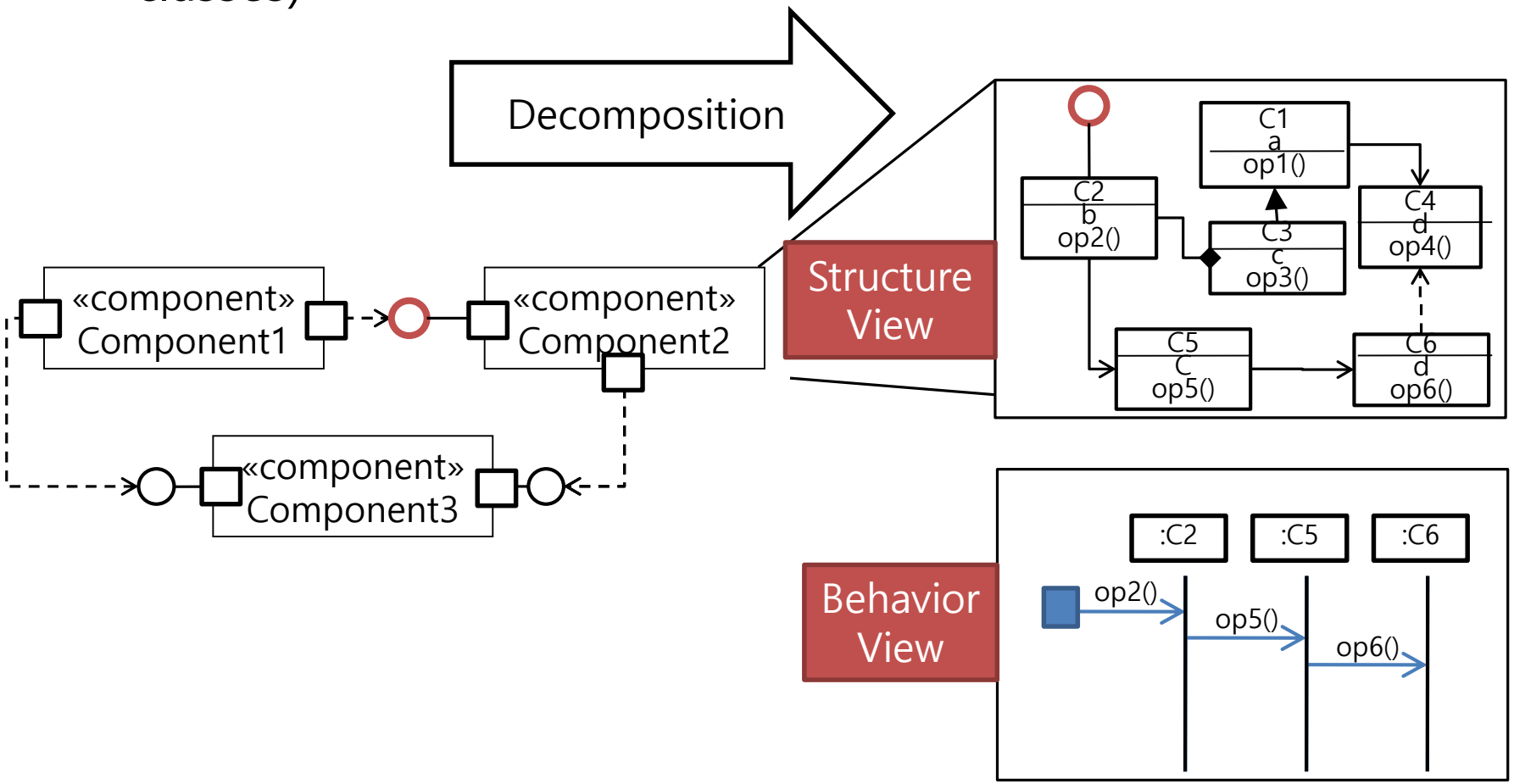
Top Level Design

❖ The first level of decomposition based on architectural drivers



Component Level Design

- Decompose each component into fine-grained elements(i.e., classes)



Views for Describing Architecture

	Top Level Design	Component Level Design
System Infrastructure View	√	
Structure View	√	√
Behavior View	√	√
Deployment View	√	

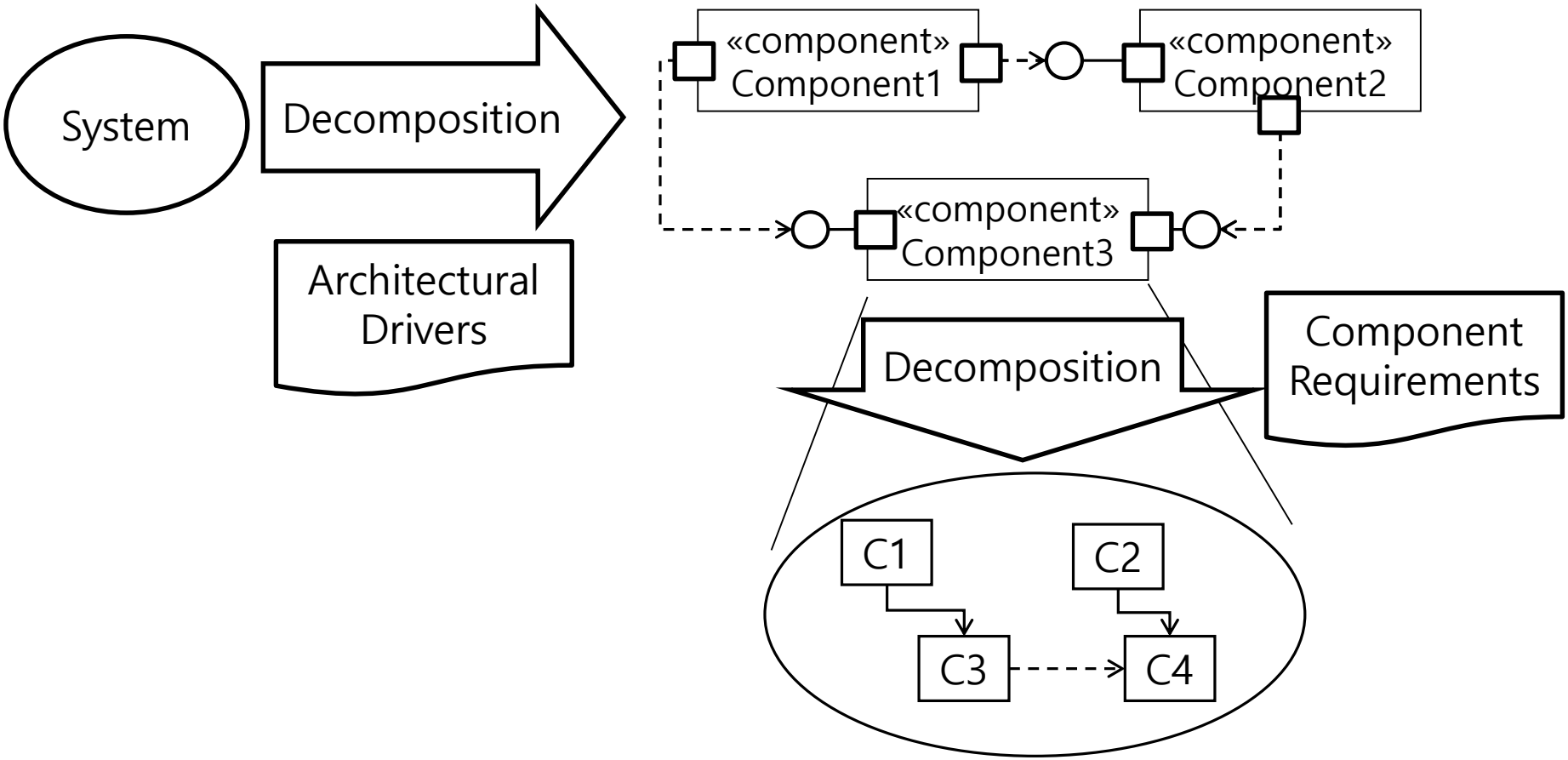
UML Diagrams for Architectural Design

View	Top Level Design	Component Level Design
System Infrastructure View	Deployment Diagram	N/A
Structure View	Component Diagram	Class Diagram
Behavior View	Sequence Diagram	Sequence Diagram
Deployment View	Deployment Diagram	N/A

DESIGN APPROACH

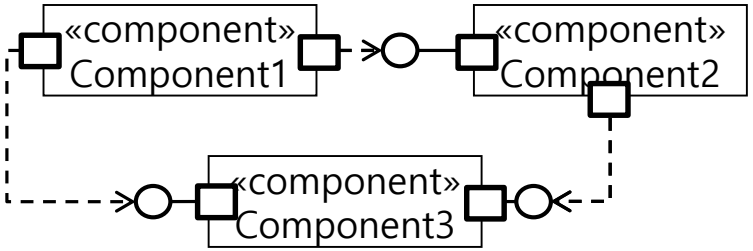
Design is Decomposition

❖ Basically, design is a series of decomposition



Decomposition Strategies

- Functionality: functional decomposition
- QA: achievement of QA; apply tactics for performance, availability, …



«component»
Component123

- Archetypes
- Reuse: Reference architecture, Patterns

- Product line implementation: common vs variable
- Build-versus-buy

- Team allocation

Design Techniques

- Reference architectures
- Patterns
- Tactics
- Externally developed components

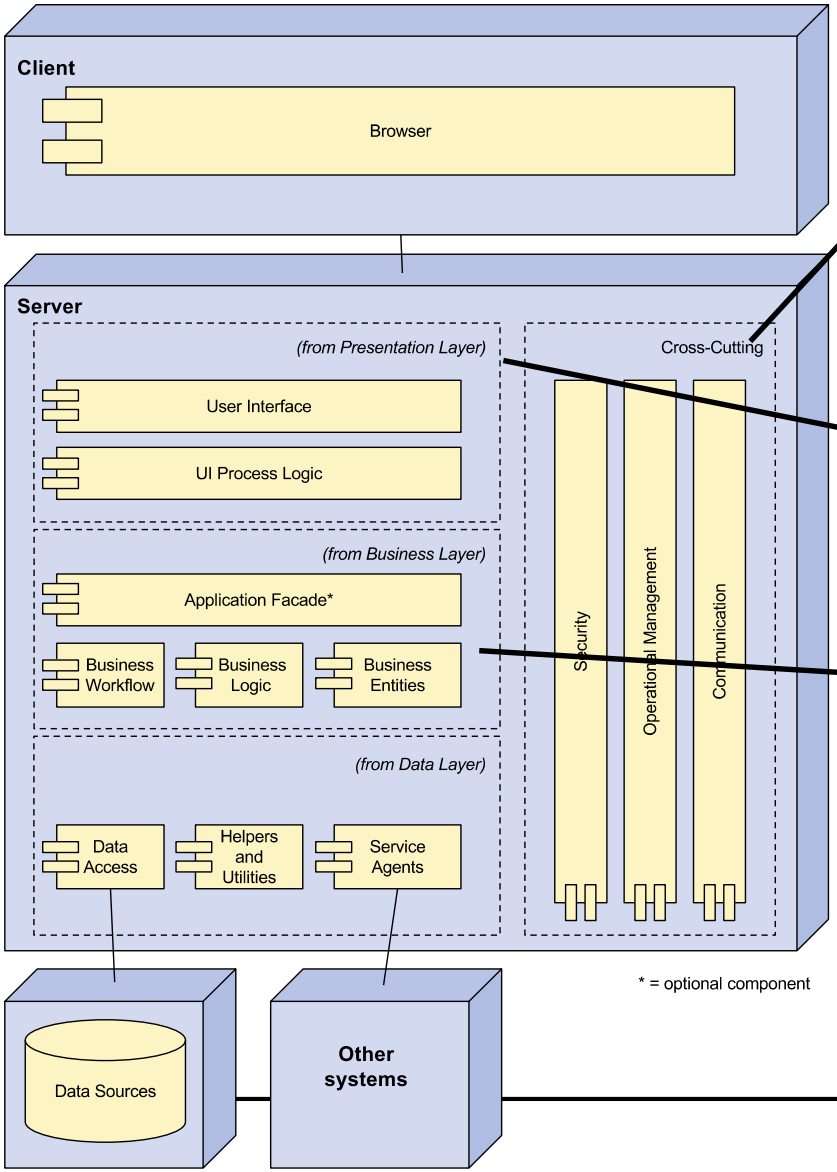
Reference Architectures

- Blueprints that provide an overall logical structure for particular types of applications.
- It has been proven in business and technical contexts and typically comes with a set of supporting artifacts that eases its use

- Typical reference architectures include
  - Web application
  - Mobile application
  - Lambda architecture

- Architectural patterns vs Reference architecture
  - Architectural patterns(such as "Pipe and Filter" and "Client Server") define types of components and connectors for structuring an application either logically or physically
  - Reference architectures provide a structure for applications in specific domain, and they may embody different styles.

Reference Architecture: Web Application



This reference architecture introduces cross-cutting concerns, such as security and communication

This reference architecture incorporates patterns and often constrains these patterns. For example, three layers are enforced.

This reference architecture incorporates other patterns such as an Application Façade and Data Access Components

Microsoft Application Architecture Guide, 2nd edition(2009)

Architectural Design Patterns

- Patterns are conceptual solutions to recurring design problems that exist in a defined context.

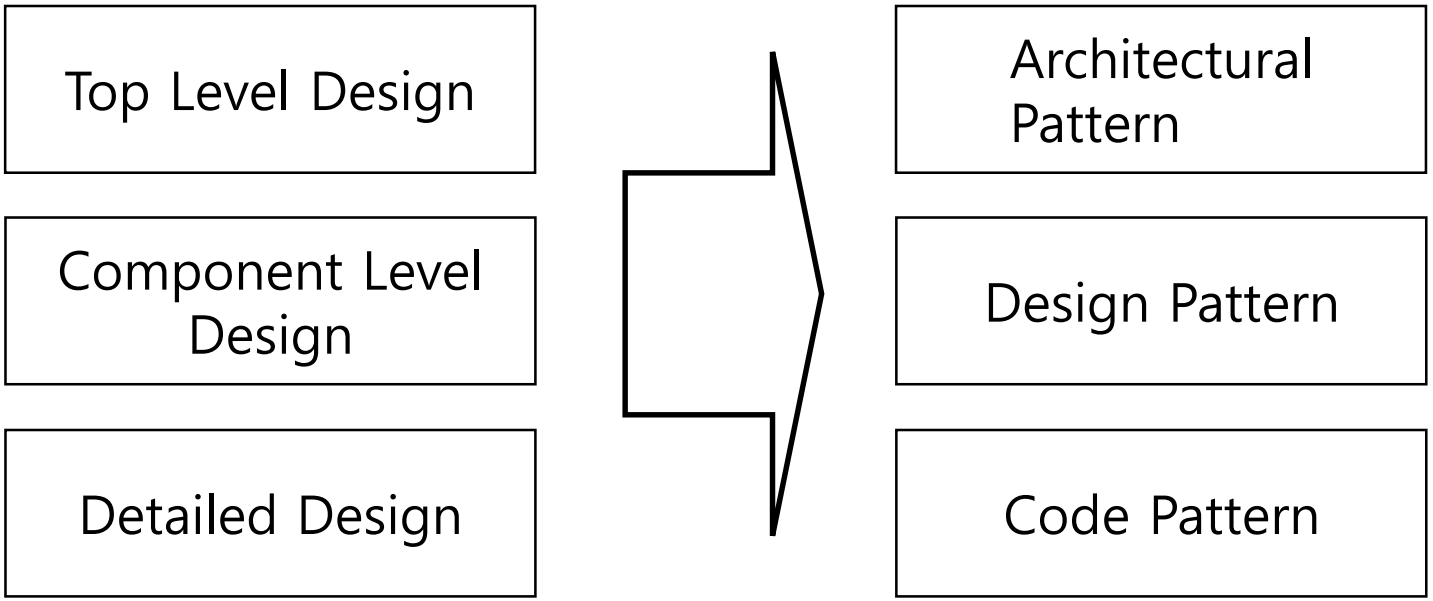
- There are catalogs with patterns that address
  - Decisions at varying levels of granularity.
  - Quality attributes such as security or integration.

- A pattern is architectural when its use directly and substantially influences the satisfaction of some of the architectural drivers

Patterns

- Patterns are an well established solution to a recurring problem.

> Patterns help you learn from other's successes,
> instead of your own failures
> **Mark Johnson** (cited by B. Eckel)



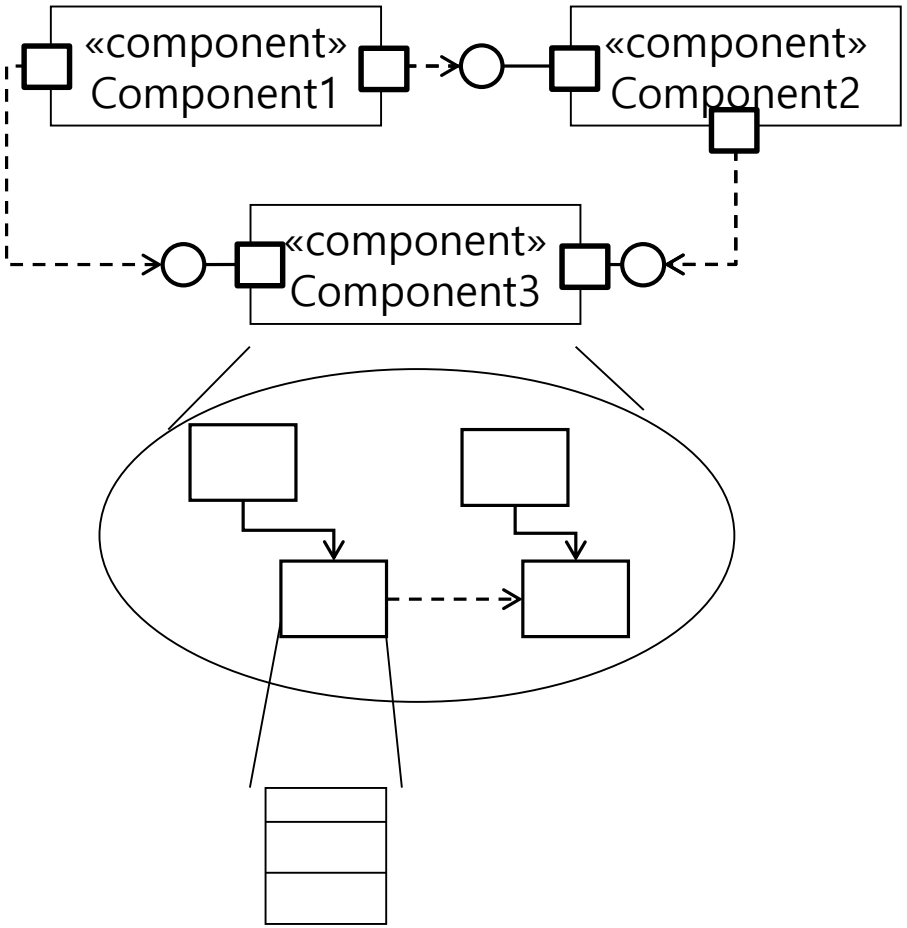
Patterns

❖ Patterns can be classified according to the level of design

Architectural Design Patterns

Design Patterns

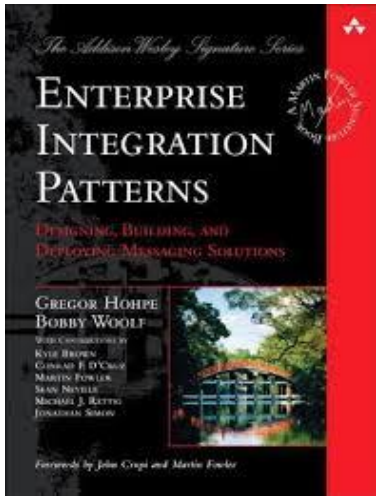
Code Patterns(Code Idioms)



Architectural Patterns

Pattern-oriented software architecture Vol. 1(1996)–5(2007)

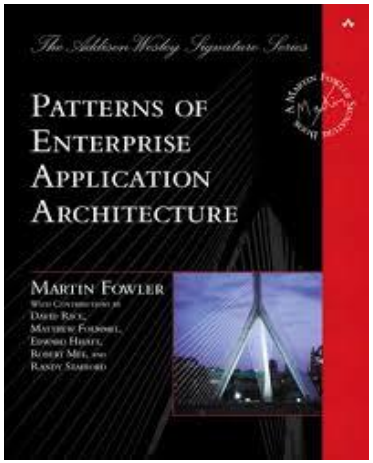
- Client Server, Broker, Client-Dispatcher-Server
- MVC
- Pipe and Filter
- Peer-to-Peer, Publish-Subscribe, SOA



Enterprise integration patterns, 2003

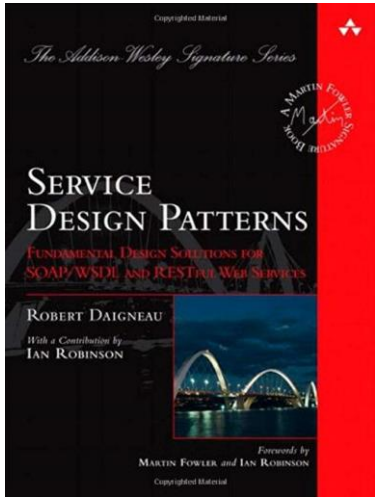
- Messaging channels
- Message construction
- Message routing
- Message transformation
- Message endpoint

Architectural Patterns



Patterns of enterprise application architecture, 2002

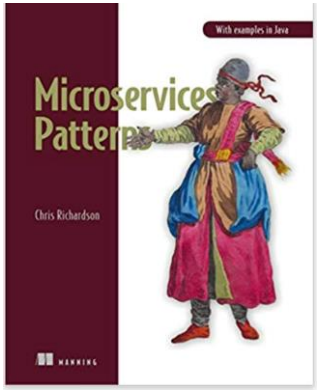
- Web server patterns
- OR Mapping patterns
- Concurrency patterns
- Distribution patterns



Service design patterns, 2011

- Web service API: RPC API, Message API, Resource API
- Client/Server Interactions: Request/Response, Request/Ack, …
- Request and response management: Service controller, Data transfer object, Request mapper, Response mapper
- Distribution patterns

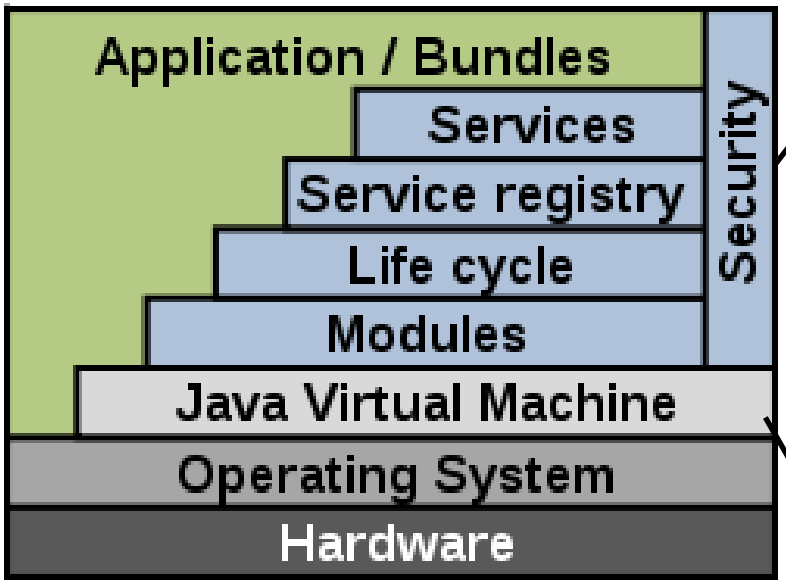
Architectural Patterns



Microservices Patterns: With examples in Java, 2018

- Decomposition patterns
- Discovery patterns: Client-side discovery, Server-side discovery
- Communication style patterns: Remote procedure invocation, Messaging
- Deployment patterns: Multiple services per host, Service-per-VM, Service-per-container, serverless deployment

Pattern Example: Layers



Sidecar

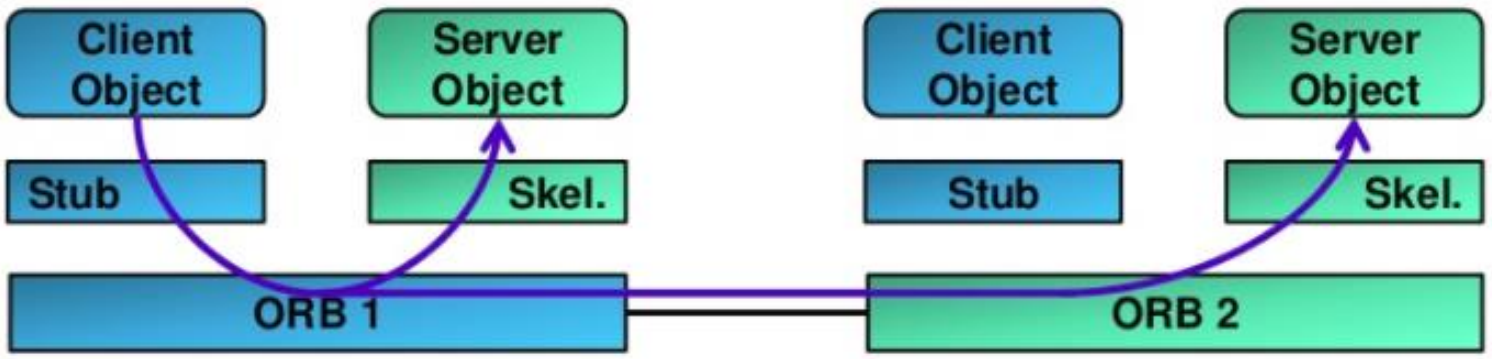
"Sidecars" like this often contain common utilities, such as error handlers, communication protocols, or database access mechanisms.

Layer bridging

Legend should specify whether layer bridging is allowed or not. That is, can a layer use any lower layer, or just the next lower one?

Pattern Example: Broker Pattern - CORBA

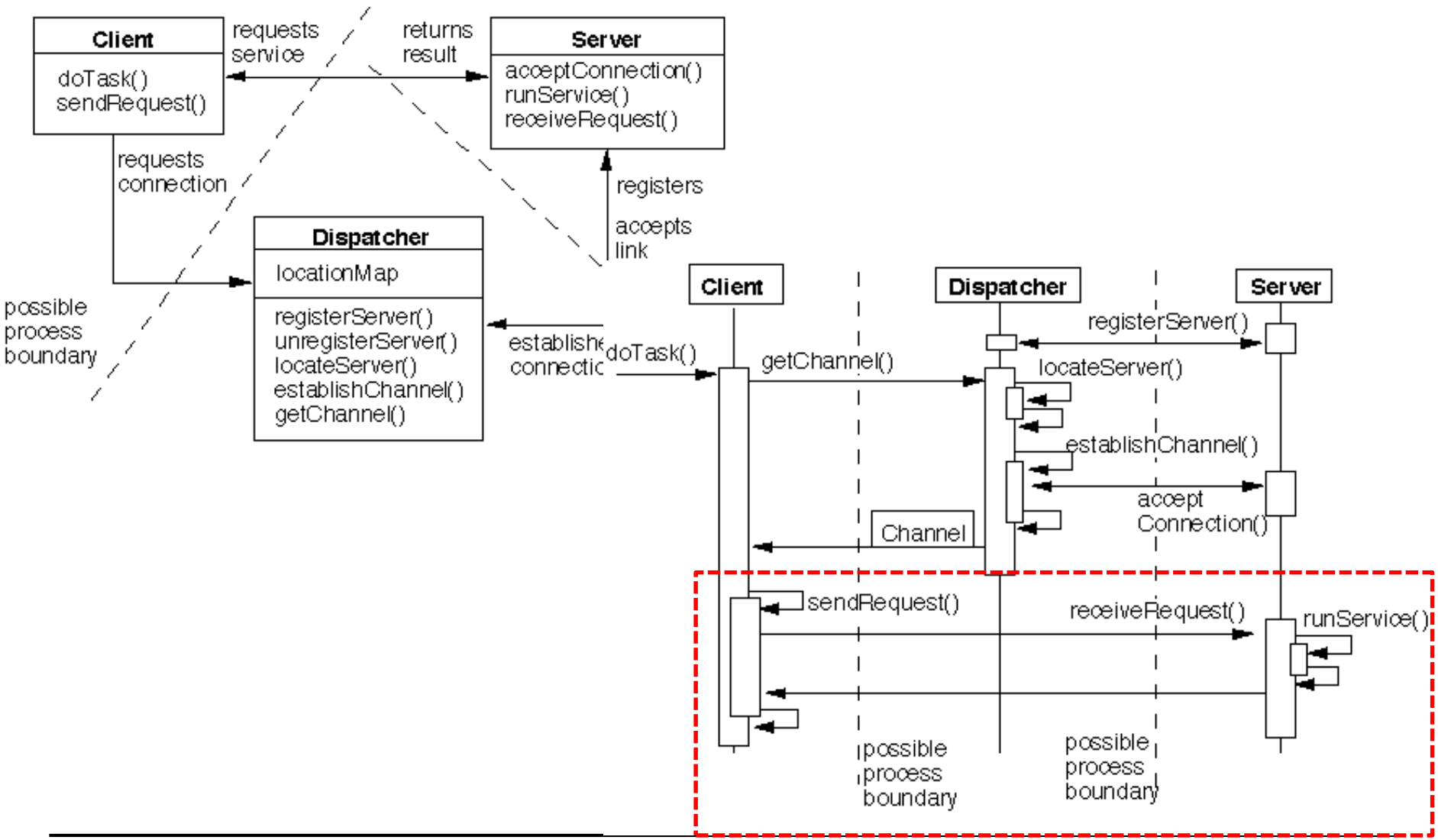
❖ ORB(Object Request Broker)



Local broker
When a request arrives for a server that is maintained by the local broker, the broker passes the request directly to the server

Remote broker
If the specified server is hosted by another broker, the local broker finds a route to the remote broker and forwards the request using this route.

Pattern Example: Client-Dispatcher-Server Pattern

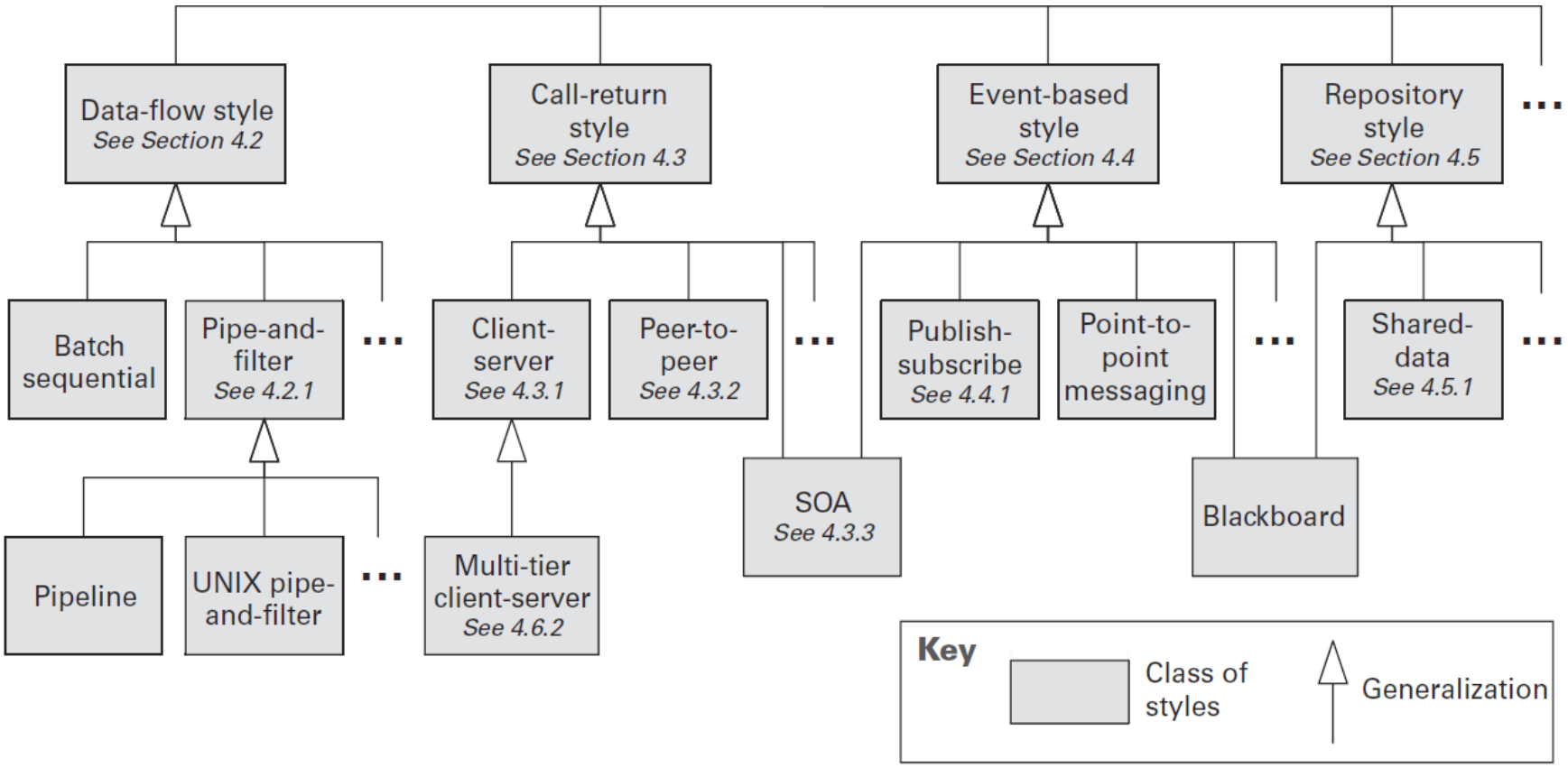


Architectural Pattern Catalogue

Category	Patterns
General	Layer Pattern, Hexagon Pattern, Pipe-and-Filter Pattern, Model-View-Controller Pattern
Distributed Computing	Client-Server Pattern, Shared Data Pattern, Multi-Tier Pattern, Service-oriented Architecture Pattern, Peer-to-Peer Pattern
Communication	Messaging Pattern, Broker Pattern, Publisher-Subscriber Pattern
Event Handling	Reactor, Proactor, Connector-Acceptor, Asynchronous Completion Token

Pattern-oriented software architecture Vol. 4(2007)

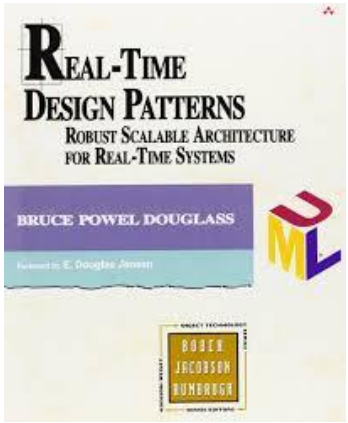
Architectural Pattern Classification



Documenting software architecture: Views and beyond, 2nd edition(2010)

Design Patterns

- Creational patterns
- Structural patterns
- Behavioral patterns

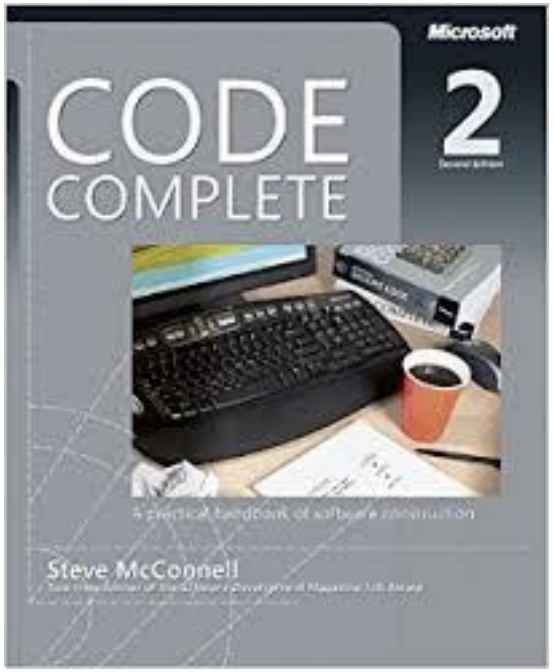


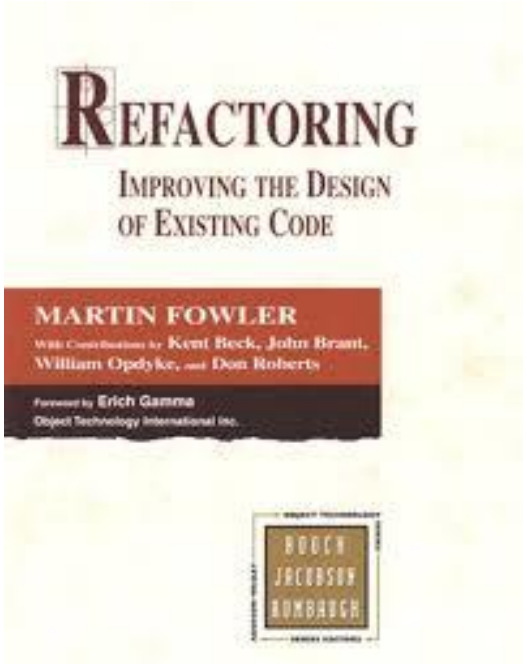
- Subsystem and component architecture patterns
- Concurrency patterns
- Memory patterns
- Resource patterns
- Distribution patterns
- Safety and Reliability patterns

Design Pattern Catalog - GoF

	Creational	Structural	Behavioral
Class-level	Factory Method	Adapter (class)	Interpreter Template Method
Object-level	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Code Patterns





Tactics

- They are techniques that architects have been using for years to achieve a particular quality attributes.
- For example, if you want to design a system to have low latency or high throughput,
  - You can mediate the arrival of events or manage resources resulting in responses that are produced within some time constraints.

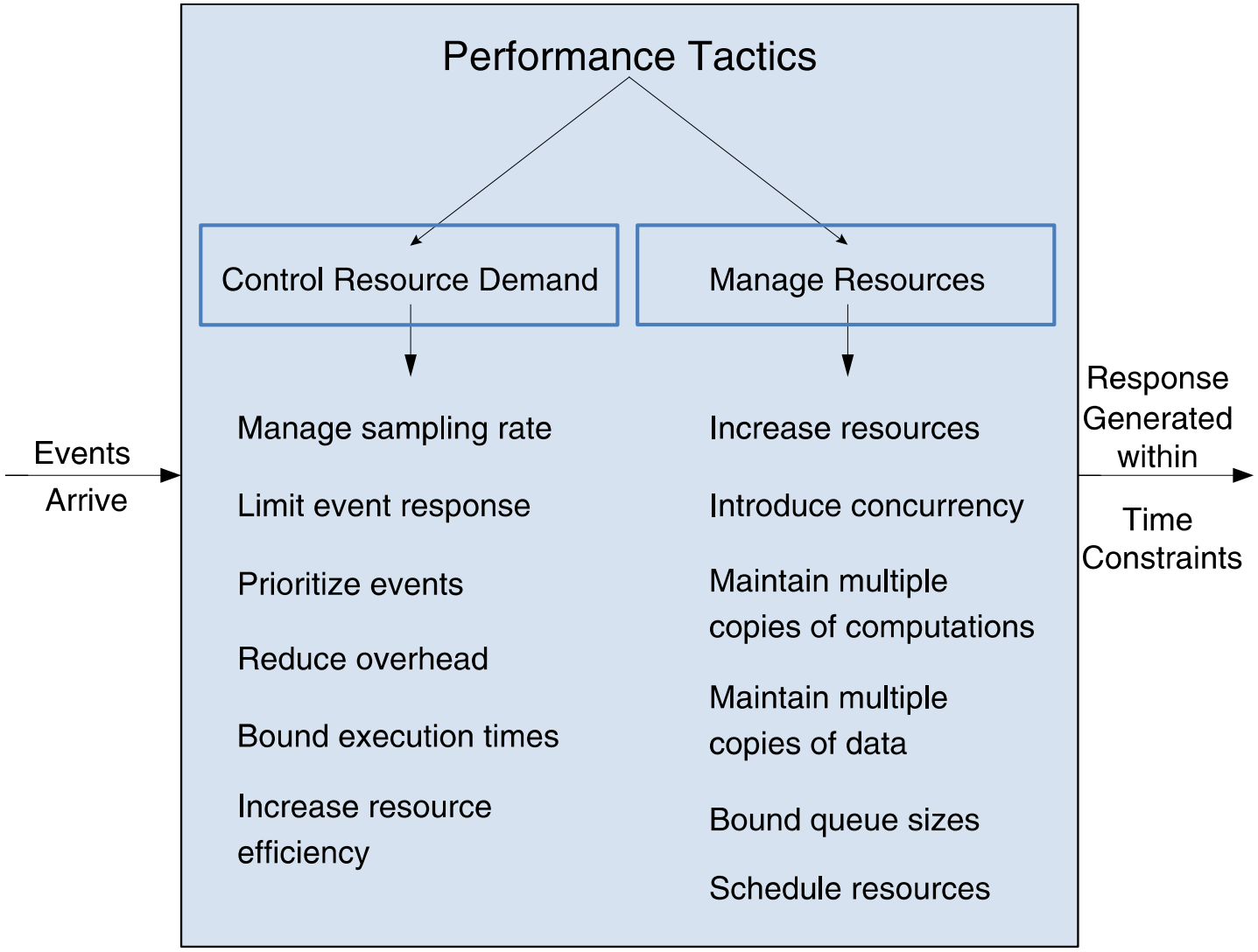
- Tactics focus on single quality attribute such as
  - Performance
  - Availability
  - Modifiability
  - Interoperability
  - Usability
  - Testability

Software architecture in practice, 3rd edition, 2012

Tactics

Quality Attribute	Tactics
Performance	Control resource demand, Manage resources
Availability	Defect faults, Recover from faults, Prevent faults
Interoperability	Locate, Manage interfaces
Usability	Support user initiative, Support system initiative
Modifiability	Reduce size of a module, Increase cohesion, Reduce coupling, Defer binding
Testability	Control and observe system state, Limit complexity

Performance Tactics



Externally Developed Components

- Patterns and tactics are abstract in nature, they need to be implemented.

- There are two ways to achieve this
  - (build) code the elements obtained from tactics and patterns
  - (buy) associate technologies with one or more of these elements in the architecture.

- This "buy versus build" choice is one of the most important decisions you will make as an architect

- We consider technologies to be externally developed components, because they are not created as part of the development project.

Externally Developed Components

- Technology families
  - A technology family represents a group of specific technologies with common functional purposes.
  - It can serve as a placeholder until a specific product or framework is selected.
  - Example: ORM
- Products
  - A self-contained functional piece of software that can be integrated into the system that is being designed and that requires only minor configuration or coding
  - Example: MySQL
- Application frameworks
  - Reusable software element, constructed out of patterns and tactics, that provides generic functionality addressing recurring domain and quality attribute concerns across a broad range of applications
  - Example: Hibernate, Spring, JSF
- Platforms
  - A platform provides a complete infrastructure upon which to build and execute applications.
  - Example: Java EE, .NET, Google Cloud

DESIGN PRINCIPLES

Design Principles

❖ A set of design principles that are expected to satisfy

- **Cohesion/Coupling**
- **Complexity**

- **SRP**
- **OCP**
- **LSP**
- **ISP**
- **DIP**

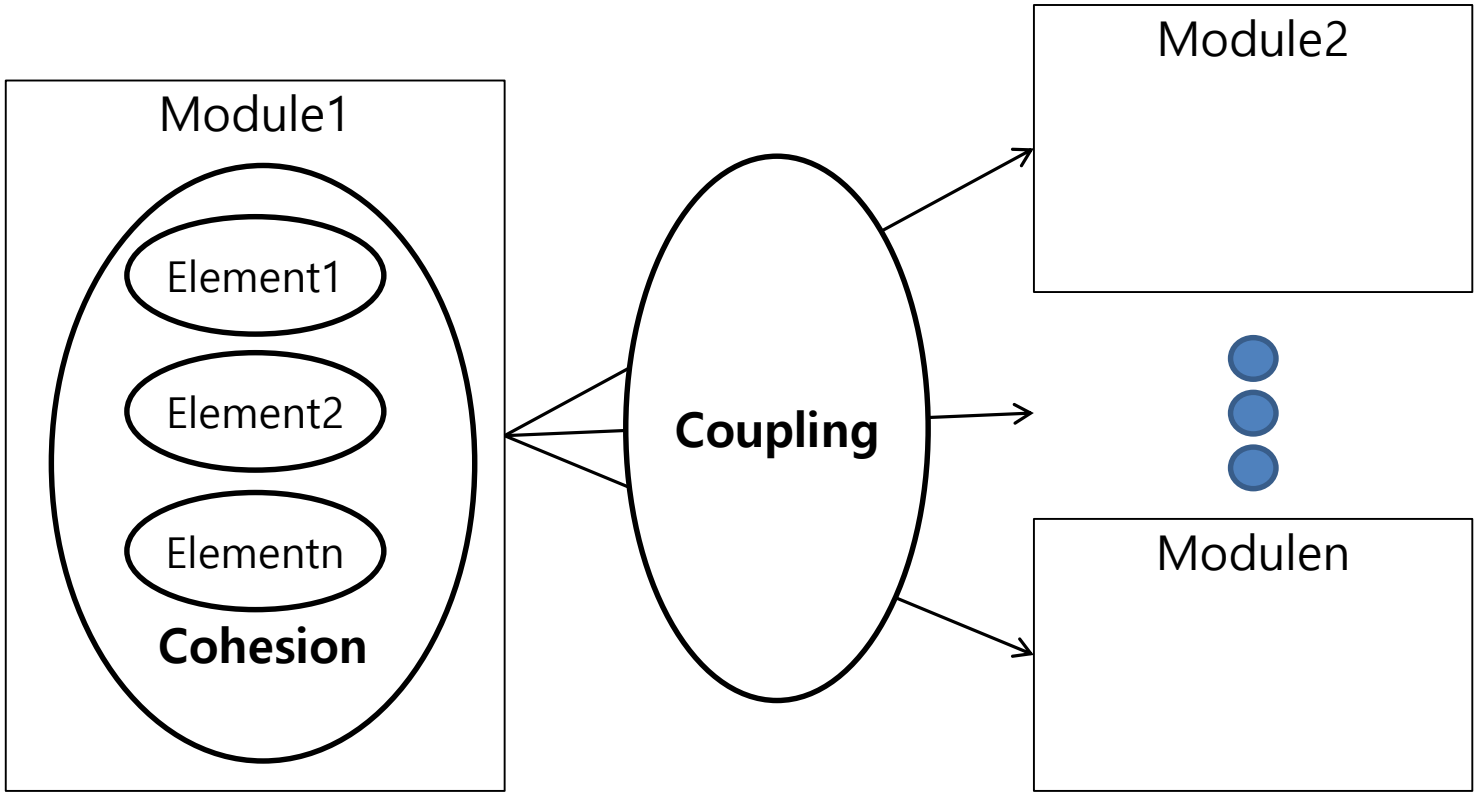
Architectural Drivers

- Reference architecture
- Architectural/design patterns
- Tactics
- Externally developed components

Architectural Design

Cohesion vs Coupling

- Coupling: Degree of **interdependence** between two modules.
- Cohesion: Strength of **functional relatedness** of elements within a module

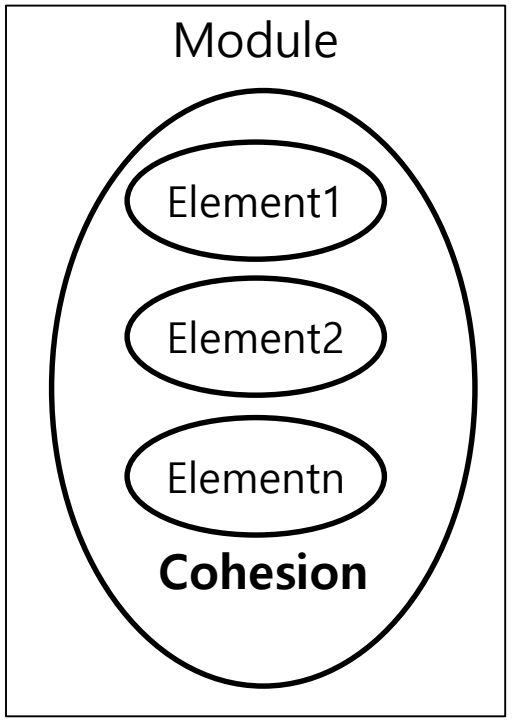


Cohesion

- Strength of **functional relatedness** of elements within a module

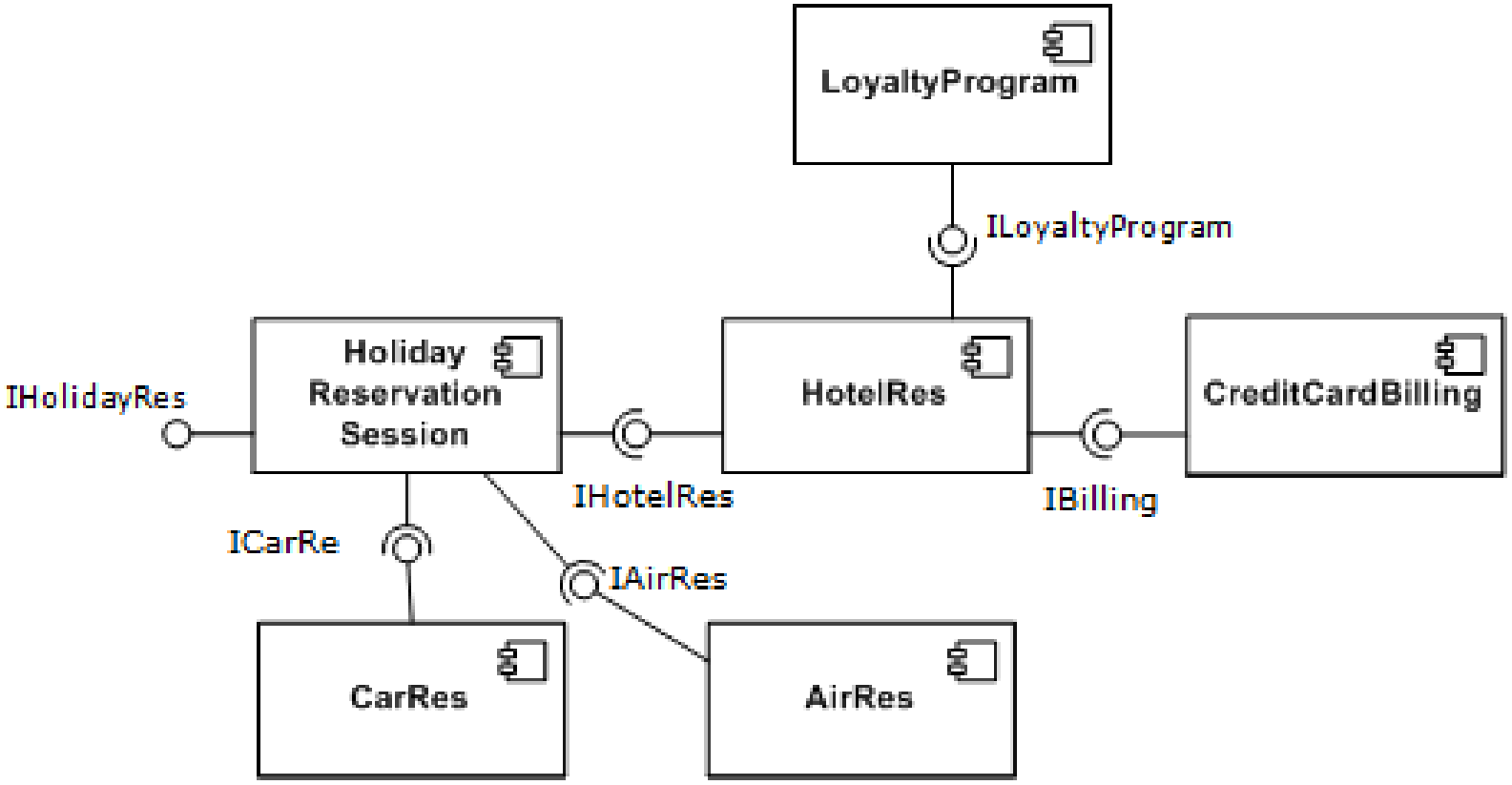
- Cohesion is a universal concept
  - Function cohesion
  - Class cohesion
  - Package cohesion
  - Component cohesion

- Cohesion metrics for Classes
  - LCOM
  - LCC/TCC



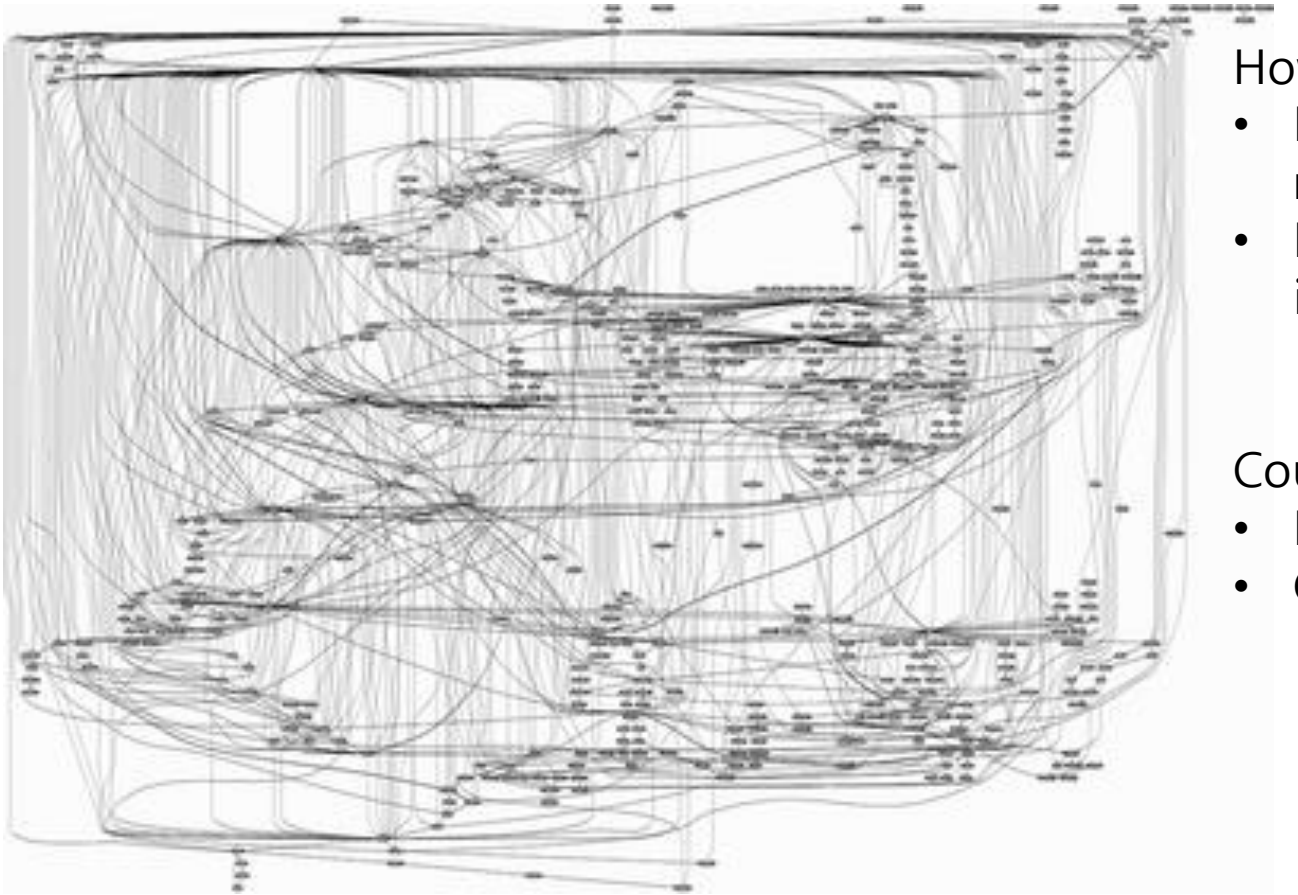
Component Cohesion

❖ UML Component Diagram



Coupling

- Highly coupled systems are harder to understand and maintain.



How to achieve low coupling

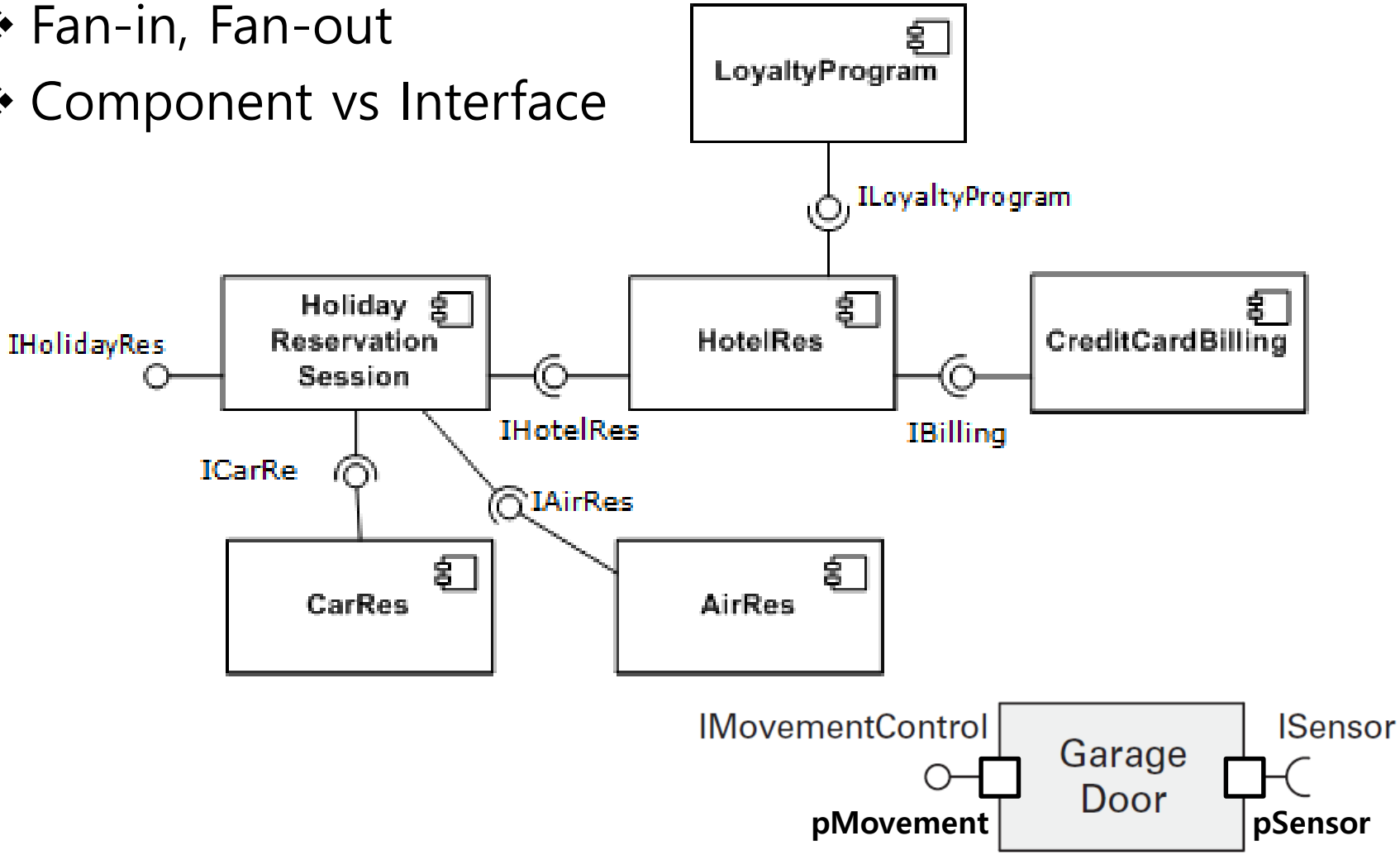
- Eliminate unnecessary relationships
- Minimize dependency on implementations(DIP)

Coupling metrics

- Fan-out, Fan-in
- CBO, RFC

Component Coupling

- Fan-in, Fan-out
- Component vs Interface



SOLID Principles

Five Principles of Object-Oriented Design for
Maintainable and Extensible System

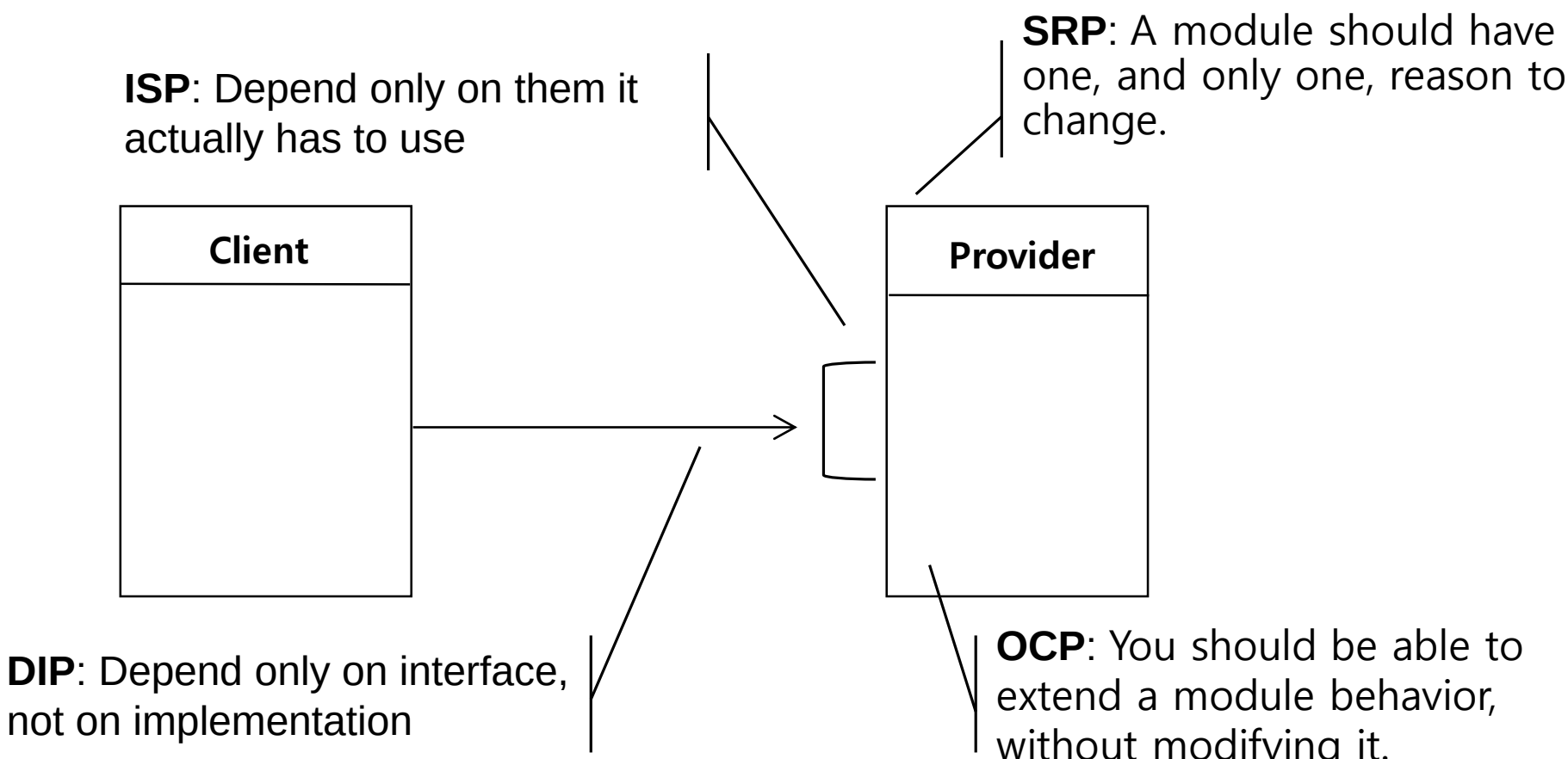
- **S**ingle **R**esponsibility **P**rinciple
- **O**pen **C**losed **P**rinciple
- **L**iskov **S**ubstitution **P**rinciple
- **I**nterface **S**egregation **P**rinciple
- **D**ependency **I**nversion **P**rinciple

By Robert Martin

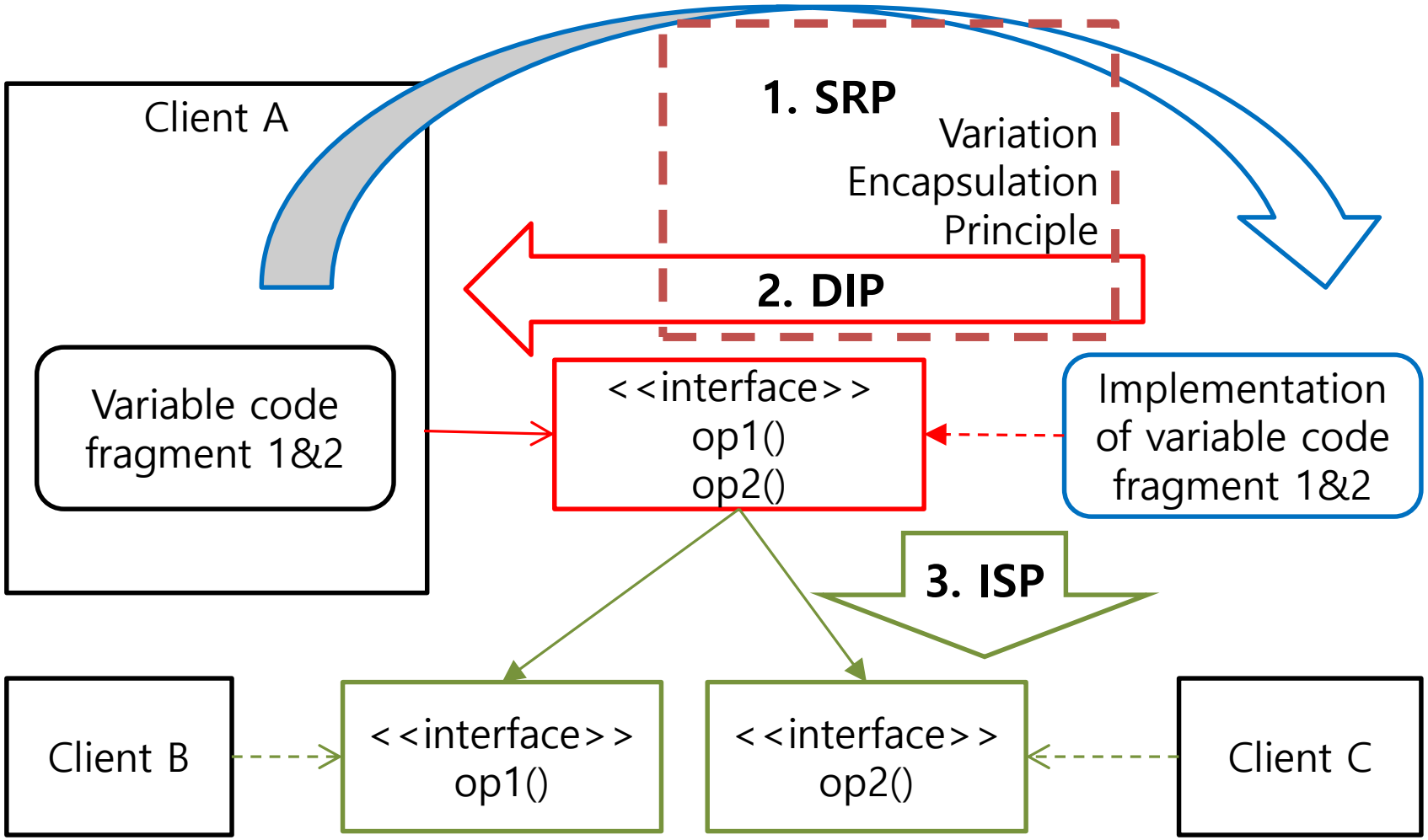
SOLID - Summary

SRP	Single Responsibility Principle	A module should have one, and only one, reason to change.	Separate the module into multiple ones for each reason.
ISP	Interface Segregation Principle	Client should not be affected by the interface it does not use.	Make fine grained interfaces that are client specific.
OCP	Open Closed Principle	You should be able to extend a module behavior, without modifying it.	Provide extension points for any possible change.
LSP	Liskov Substitution Principle	Derived classes must be substitutable for their base classes.	Subclasses should conform to pre/post condition of its superclass
DIP	Dependency Inversion Principle	Do not depend on what are prone to change	Depend on interface, not on implementation.

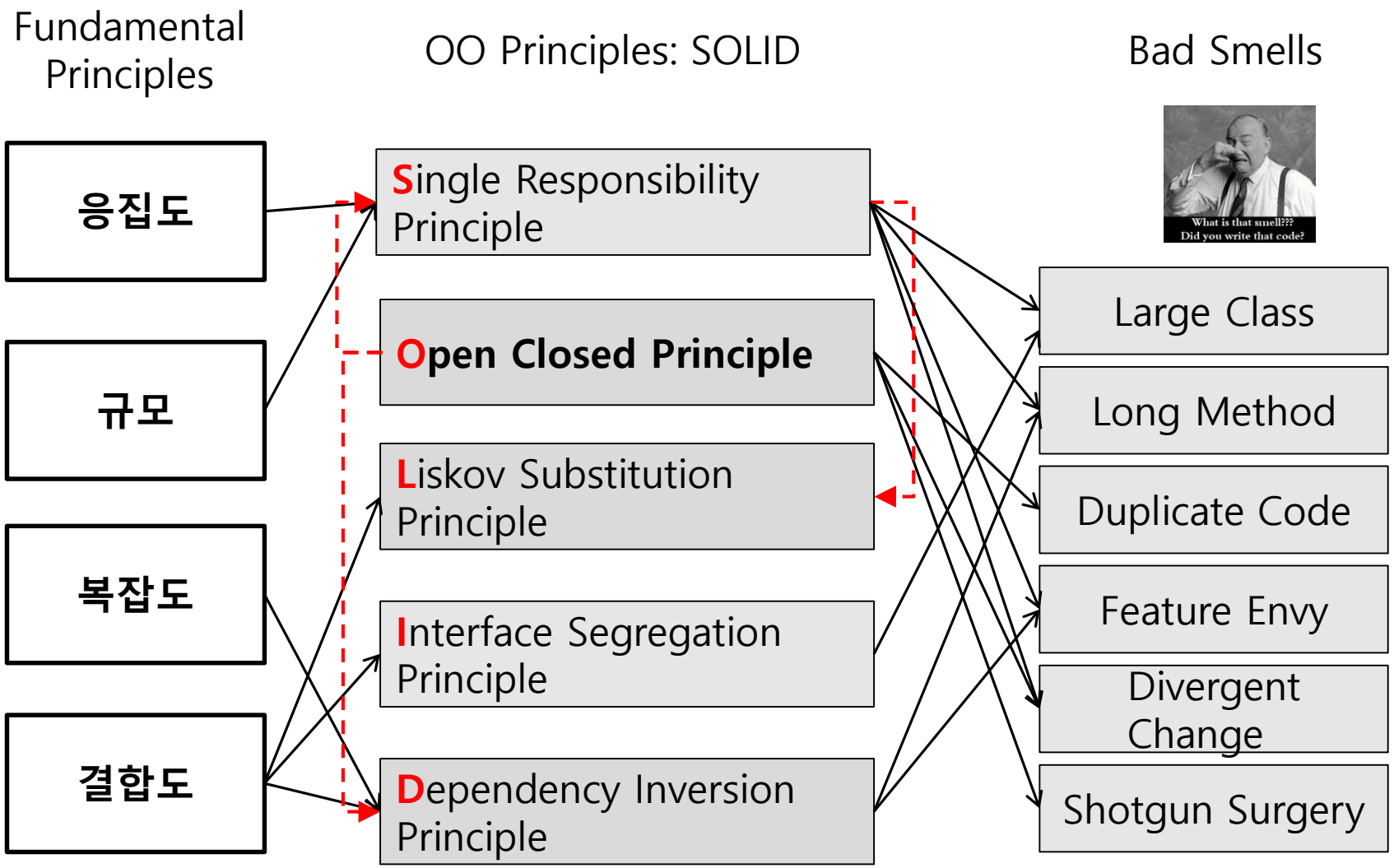
SOLID - Summary



Refactoring Procedure

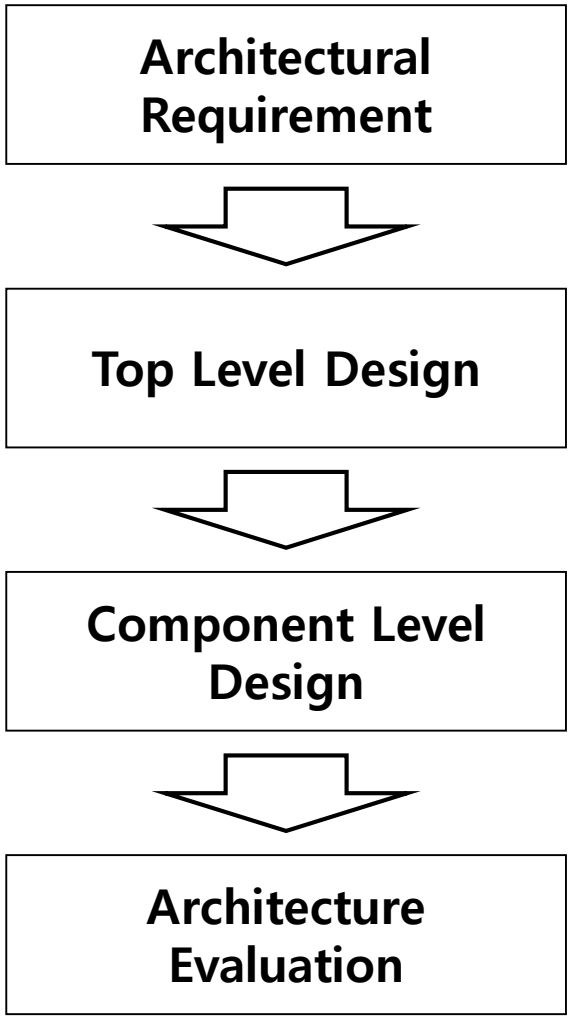


Similar, But Different Principles



ARCHITECTURAL DESIGN PROCESS AND DOCUMENTATION

Architectural Design Document



1. Introduction
2. Project Overview
3. System Overview
4. Architectural Driver

5. Top Level Design

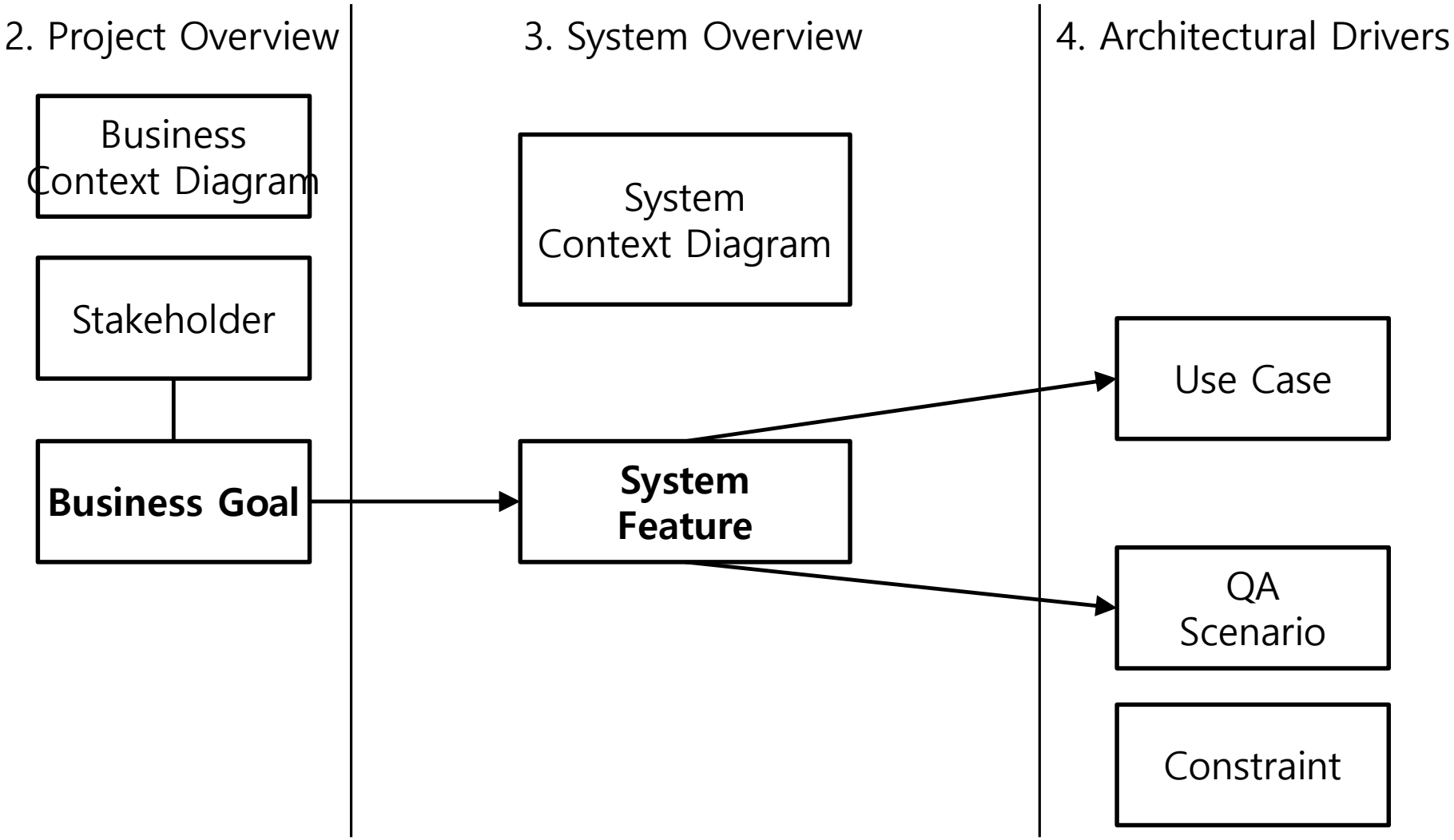
6. Component Level Design

7. Architecture Evaluation

Architecture Requirement

Chapter	Section
2. Project Overview	2.1 Project Background 2.2 Business Context Diagram 2.3 Stakeholder List 2.4 Business Goal List
3. System Overview	3.1 System Context Diagram 3.2 External Entity List 3.3 External Interface List 3.4 System Feature List
4. Architectural Drivers	4.1 Use Case Model 4.2 Quality Attribute Scenario 4.3 Constraint
5. Top Level Design Description	
6. Component Design Description	
7. Architecture Evaluation	

Architecture Requirement - Basic Concepts

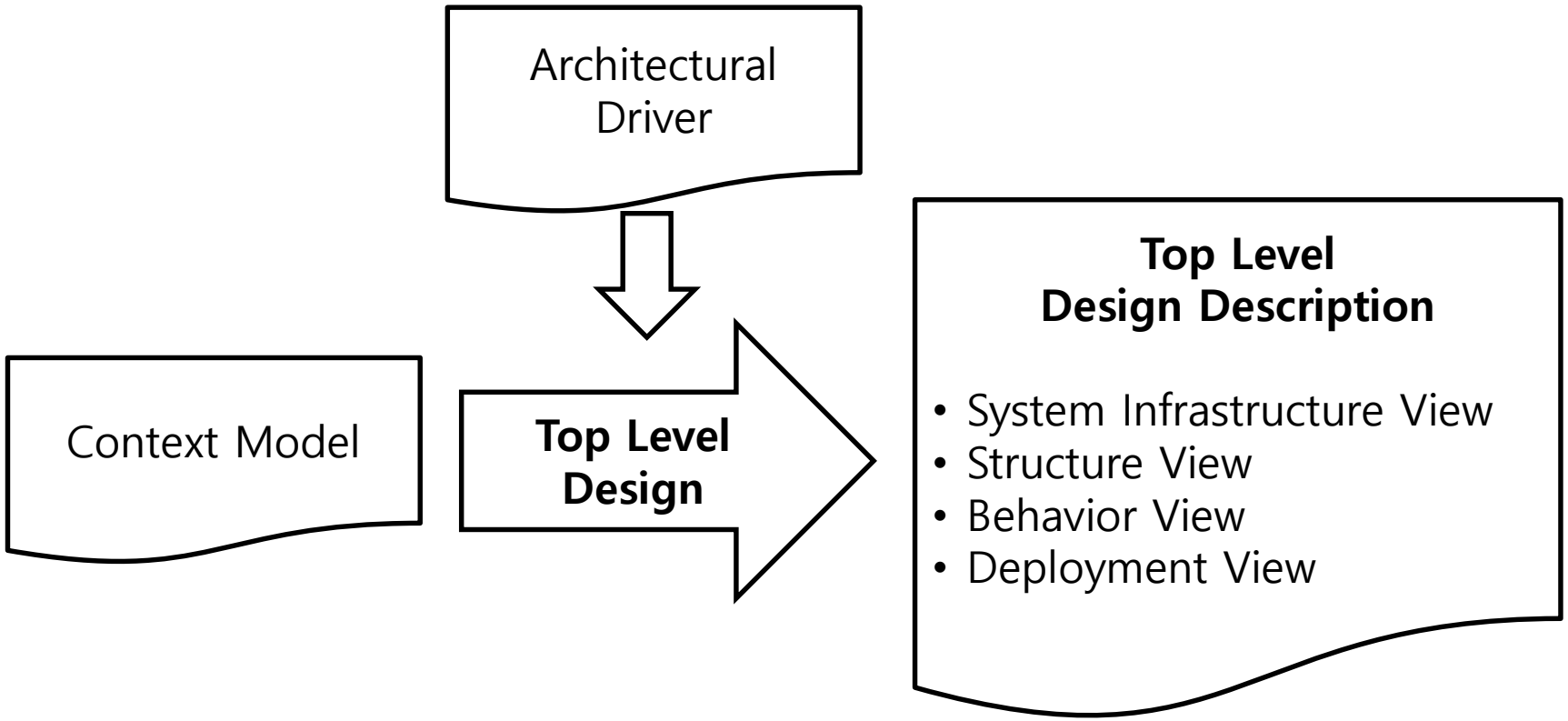


Business Goal – System Feature – UC/QA

- Goal: **Why** do you need a system?
  - Subject: Stakeholder
- Feature: **What** features does the system have to achieve the goal?
  - Subject: System will .. For Stakeholder's Goal
- UC/QA: **How** can the features be provided by the system?
  - Subject: System will do the requirement to satisfy the feature

5. Top Level Design

❖ Using the context as a starting position, start decomposing the system using the architectural drivers



5. Top Level Design Description

5.1 System Infrastructure View
- 5.1.1 System Infrastructure Diagram
- 5.1.2 Node Specification
- 5.1.3 Execution Environment Specification
- 5.1.4 Communication Path Specification

5.2 Structure View
- 5.2.1 Static Structure Model
- 5.2.2 *Component1* Component Specification
- 5.2.3 *Component2* Component Specification
- …

5. Top Level Design Description

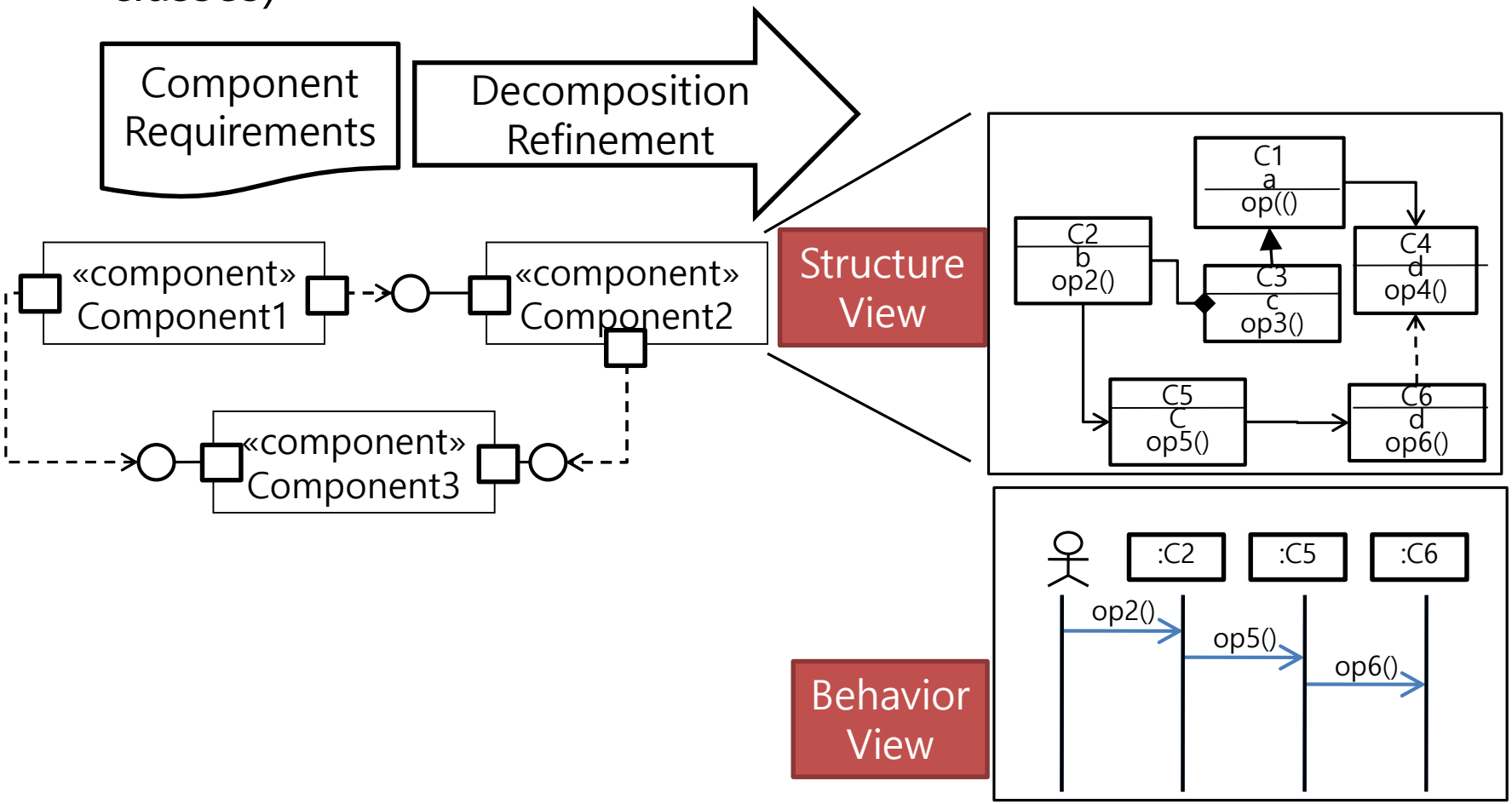
5.3 Behavior View
- 5.3.1 *UC-01 Title* Use Case Behavior Model
- 5.3.2 *UC-02 Title* Use Case Behavior Model
- …

5.4 Deployment View
- 5.4.1 Artifact Definition Model
- 5.4.2 Artifact Deployment Model

5.5 Documenting Design Decisions
- 5.5.1 Design Decision List
- 5.5.2 *DD-01 Title* Description
- 5.5.2 *DD-02 Title* Description
- …

6. Component Level Design Description

❖ Decompose each component into fine-grained elements(i.e., classes)



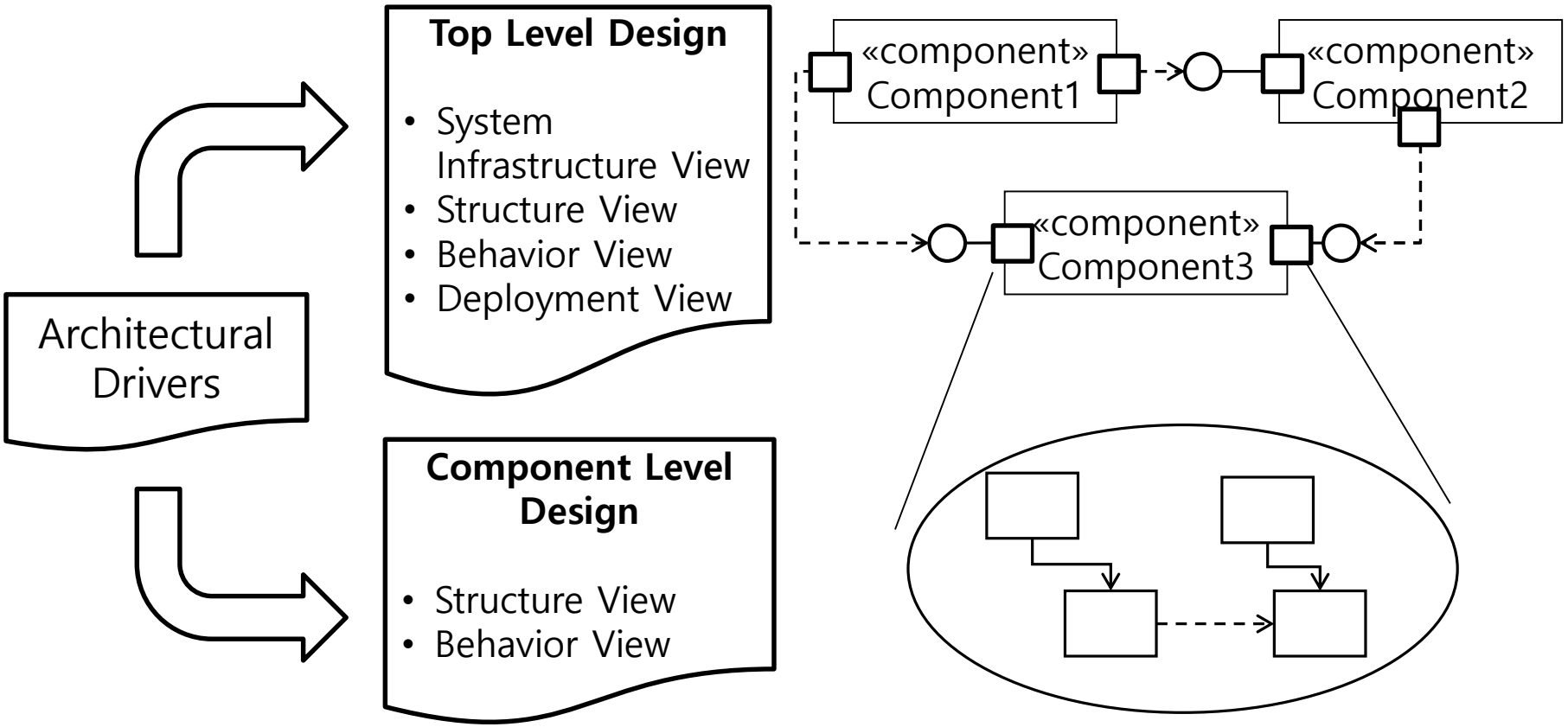
6. Component Level Design Description

6 Component Level Design Description

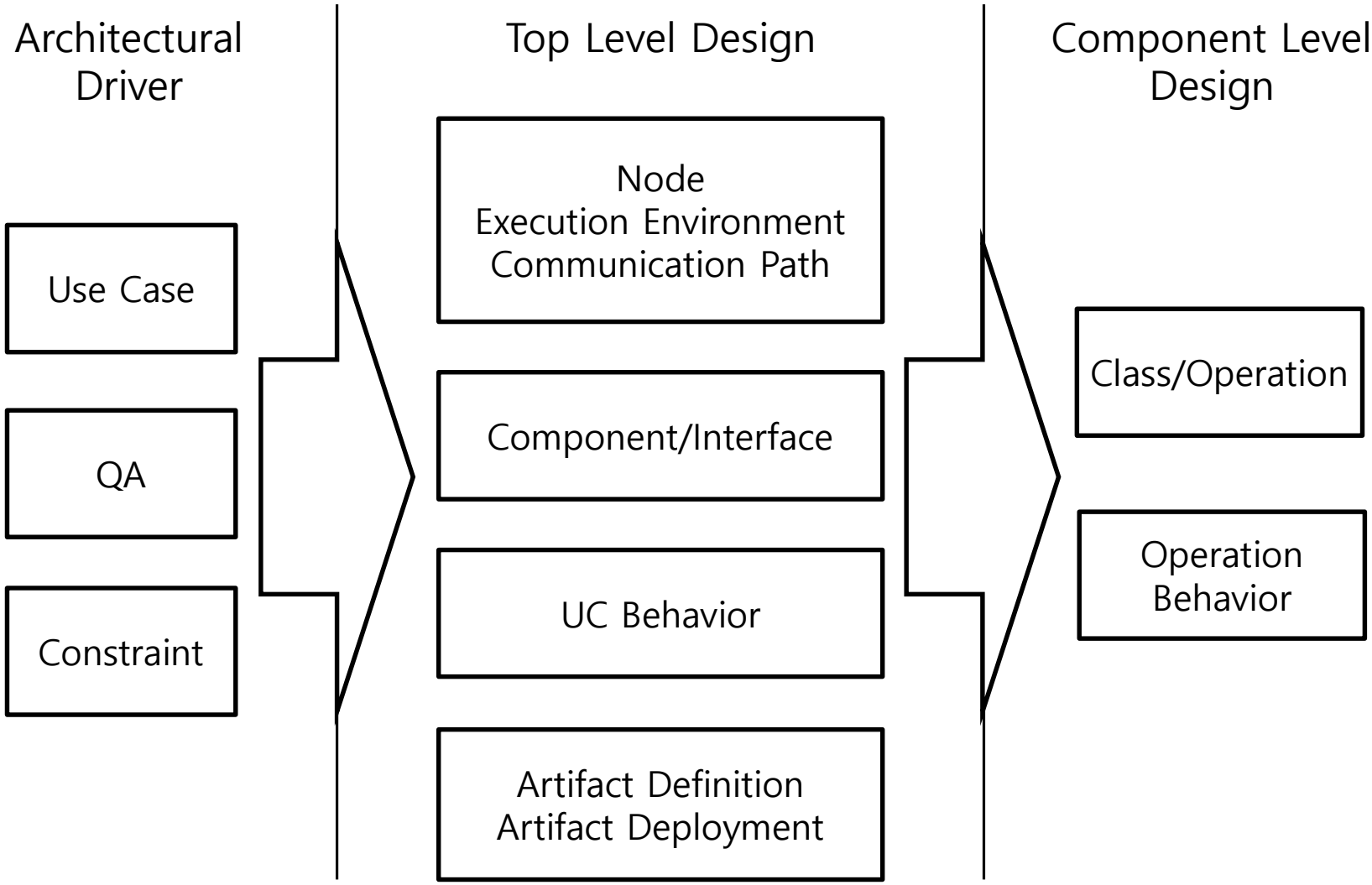
- 6.1 *Component1 Title* Description
  - 6.1.1 Static Structure Diagram
  - 6.1.2 Element List
  - 6.1.3 Design Rationale
- 6.2 *Component2 Title* Description
  - 6.2.1 Static Structure Diagram
  - 6.2.2 Element List
  - 6.2.3 Design Rationale
- …

7. Architecture Evaluation

- Make sure that the architecture you've designed can satisfy all that's expected of it: that is, architectural drivers



Traceability



Summary

- Architectural Design Overview
- Architectural Design Process
  - Architectural Drivers
  - Top Level Design
  - Component Level Design
- Design Techniques
- Design Principles
- Architectural Design Document