Software Architecture

건국대학교 컴퓨터공학과
유준범
jbyoo@konkuk.ac.kr
http://dslab.knkuk.ac.kr

Index

- **Introduction to Software Architecture**

- **Software Architecture Design**
  1. Project Overview
  2. System Overview
  3. ASR Analysis
  4. Architecture Design & Evaluation
  5. Documenting Design with Views
  6. Detailed Component Design (Optional)
  7. Architecture Traceability Summary

Text and References

Introduction to Software Architecture

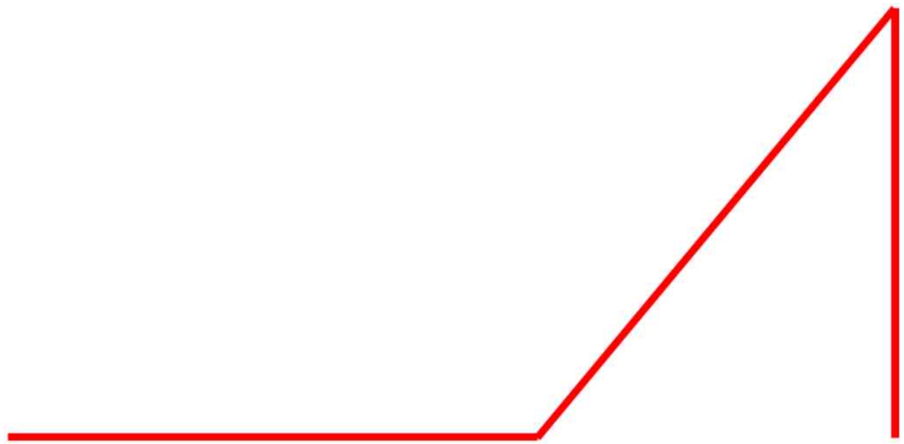
Motivations

- Learning to **design software architectures** in a systematic, predictable, repeatable, and cost-effective way.

- **No silver bullet** in designing software architecture

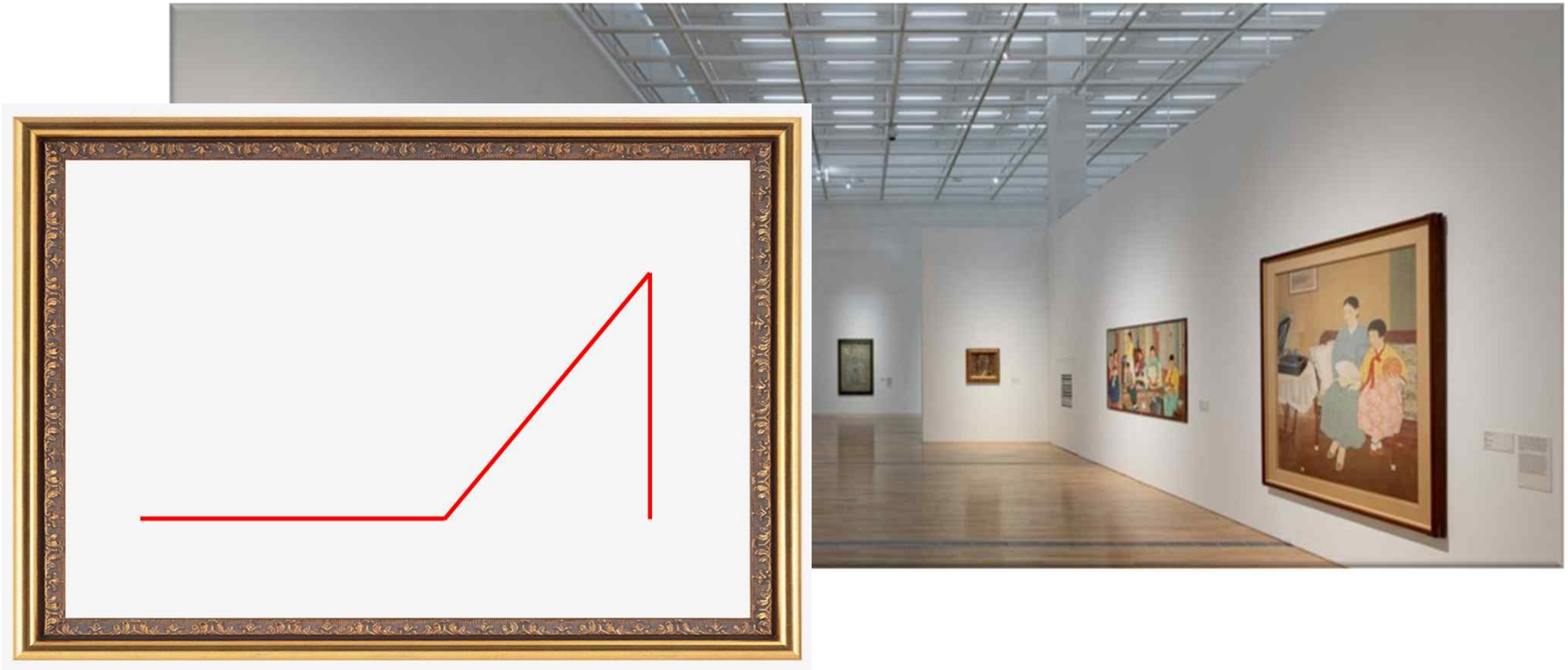
- But everyone can be a better designer
  - by structured methods supported by reusable sets of design help.

What is This?



What is This?

What is This?



작품명: 엄마 구두

작　가: 유O민 (3세)

설　명: 엄마의 7cm 하이힐을 보고
　　　　인상 깊었던 특징을 표현

What is Architecture Design?

Software (System)

Architecture Design

+

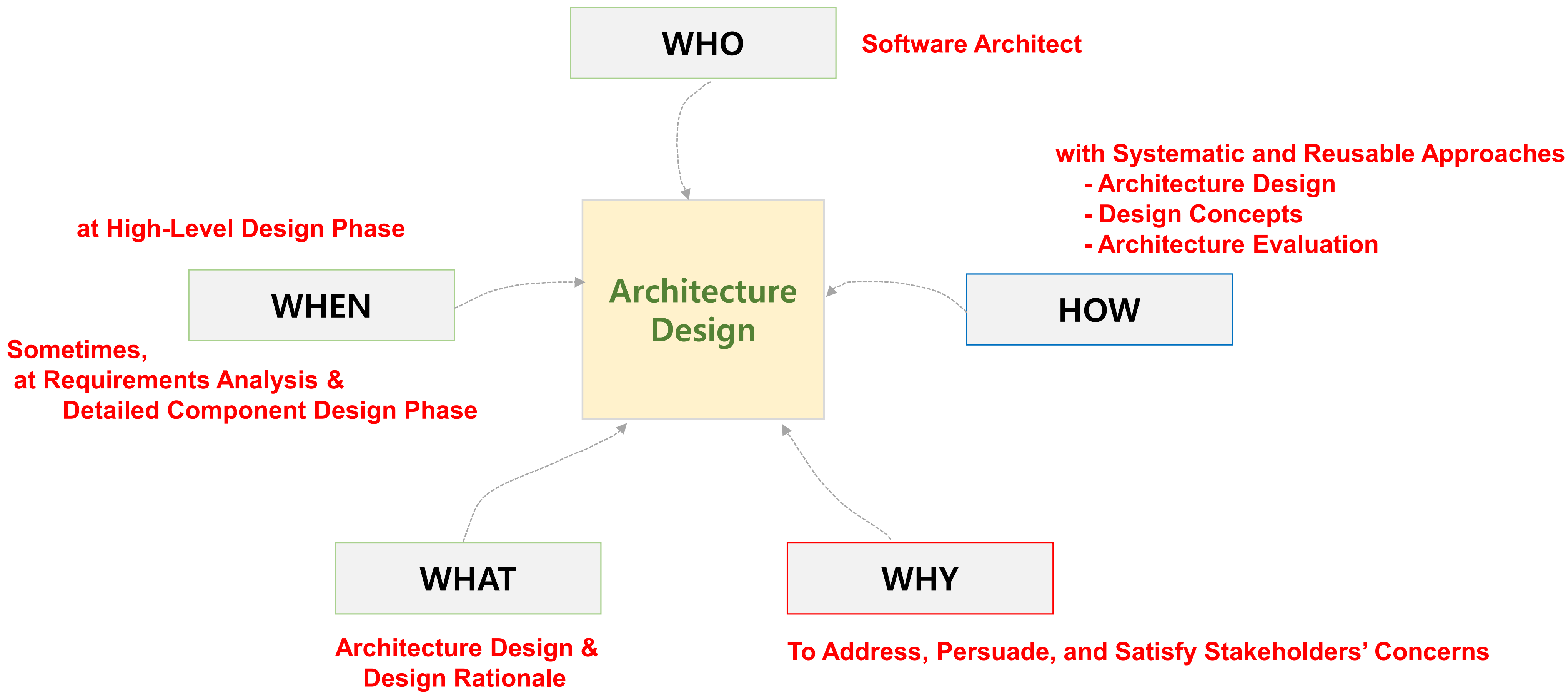
작품명: 엄마 구두

작　가: 유O민 (3세)

설　명: 엄마의 7cm 하이힐을 보고 인상 깊었던 특징을 표현

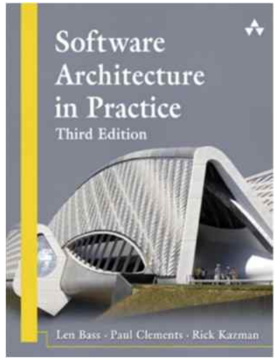
Architecture Description (AD)

The 4W1H of Architecture Design

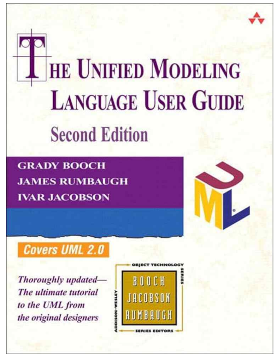


Software Architecture

*"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."*



*"A **software architecture** is the set of **significant decisions** about the **organization** of a software system, the **selection of** the **structural** elements and their **interfaces** by which the system is composed, together with their **behavior** as specified in the collaborations among those elements, the **composition of** these structural and behavioral elements into progressively larger subsystems, and the **architectural style** that guides this organization - these elements and their interfaces, their collaborations, and their composition."*

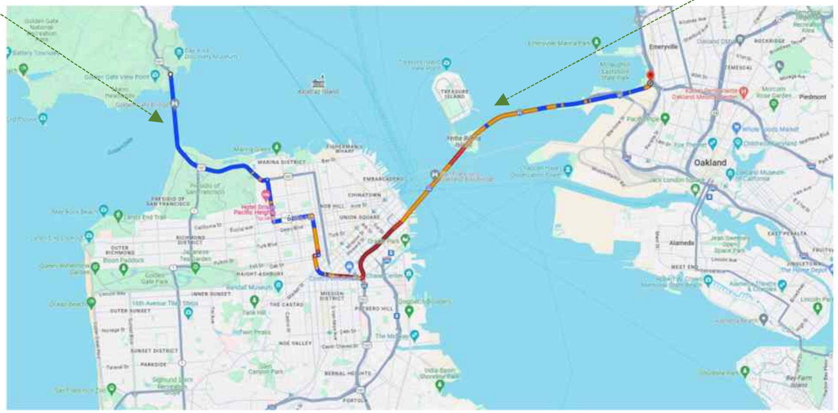


Importance of Software Architecture

Golden Gate Bridge (1937)

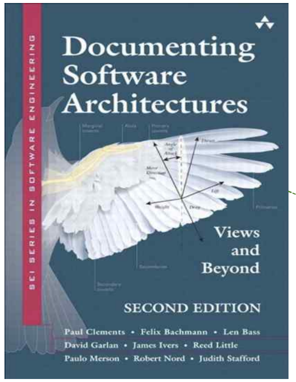
San Francisco - Oakland Bay Bridge (1936)

Collapsed in 1987 on 6.9 earthquake



San Francisco, USA

Software Architecture Life-Cycle Activities



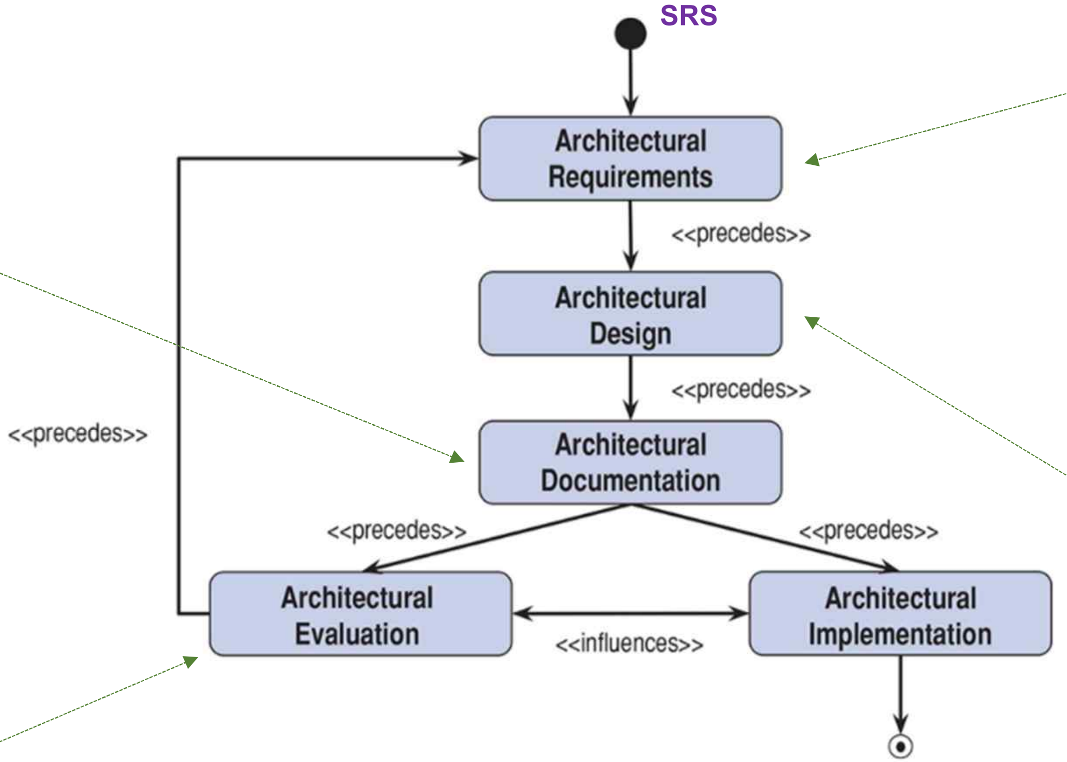
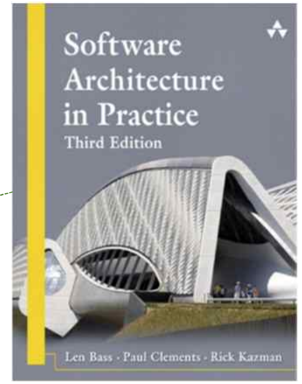
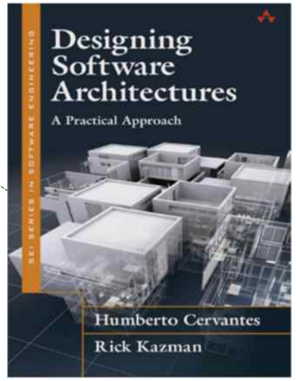
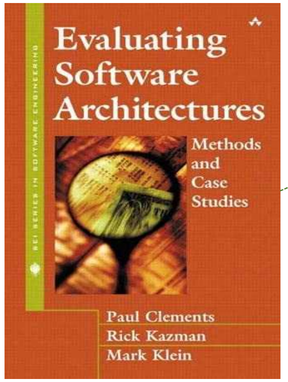


FIGURE 1.1 Software architecture life-cycle activities







The Scope of Our CEP (Comprehensive Evaluating Project)

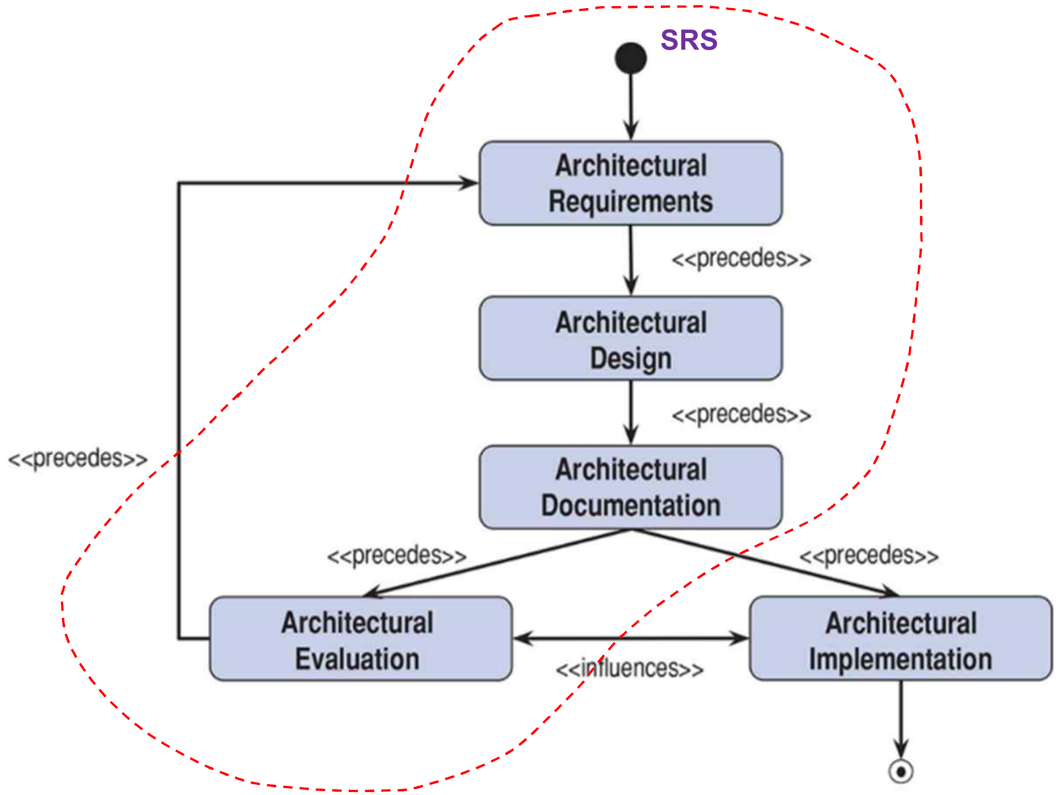


FIGURE 1.1 Software architecture life-cycle activities

Architectural Requirements

- **Architecturally Significant Requirements (ASR)**
  - A few requirements in SRS, that <u>have special importance for the architecture</u>
  - Examples :
    - **Primary functionality** : the most important functionality of the system
    - **QA (Quality Attribute)** : quality attributes such as high performance, high availability, or ease of maintenance
    - **Other design constraints**

- **ASR Analysis**
  - Identifying all ASRs <u>from an SRS</u>
  - Transforming (or Mapping) **<u>ASR into AD</u>** (Architectural Drivers)

- SRS often provides very little information for architects.
  - <u>Architects need to be involved in requirements analysis</u>.
    - Stakeholder analysis
    - Requirements elicitation
    - <u>Keeping traceability starting from stakeholders</u>

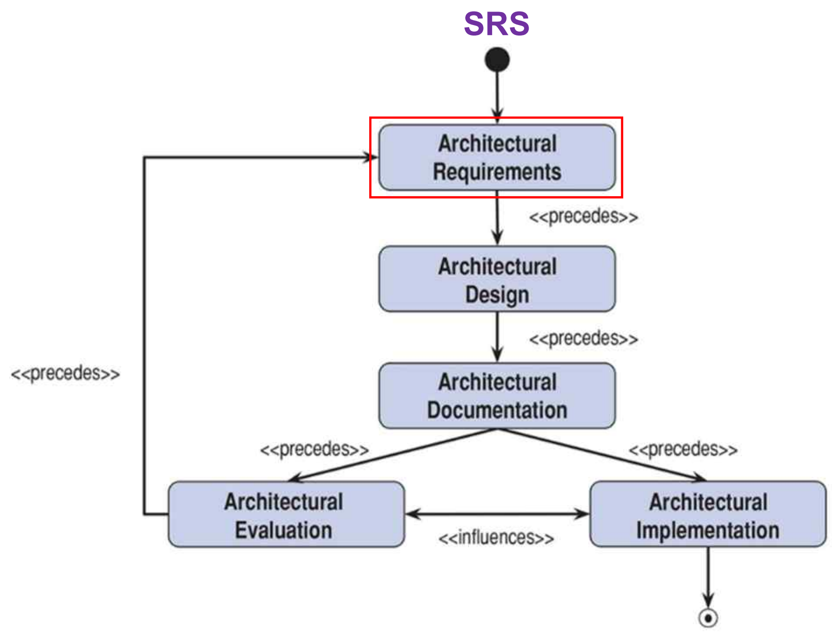


FIGURE 1.1 Software architecture life-cycle activities

Architectural Design

- **The process** of translation from the world of requirements to the world of solutions
  - Producing a set of structures composed of code, frameworks, and components
  - Example :
    - **ADD (Attribute Driven Design) 3.0** with **Design Concepts**

- A good design is one that satisfies all **AD (Architectural Drivers)**.

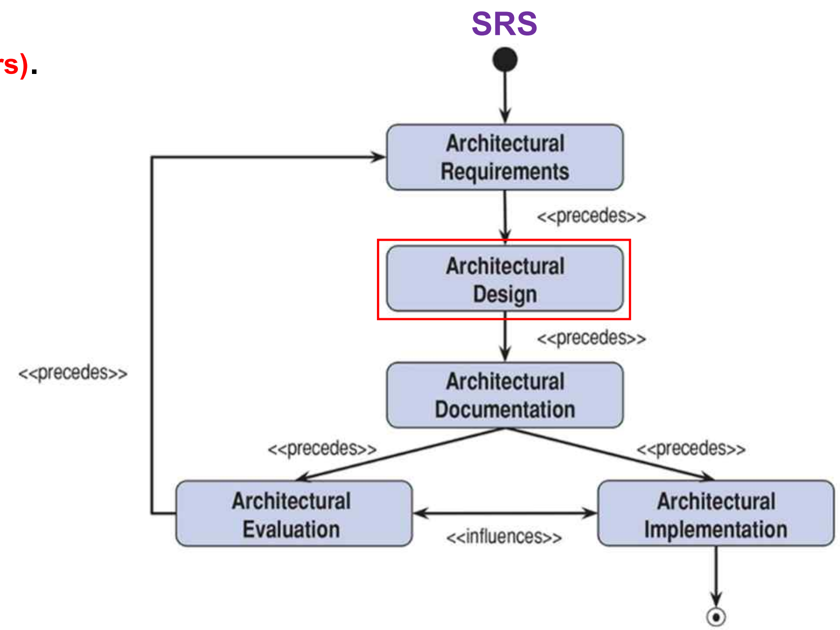


FIGURE 1.1 Software architecture life-cycle activities

Architectural Documentation

- **Preliminary documentation of the structures** should be created as part of architectural design.

- **Architecture Description (AD)**
  - ISO/IEC/IEEE 42010:2011
    “Systems and Software Engineering - Architecture Description”

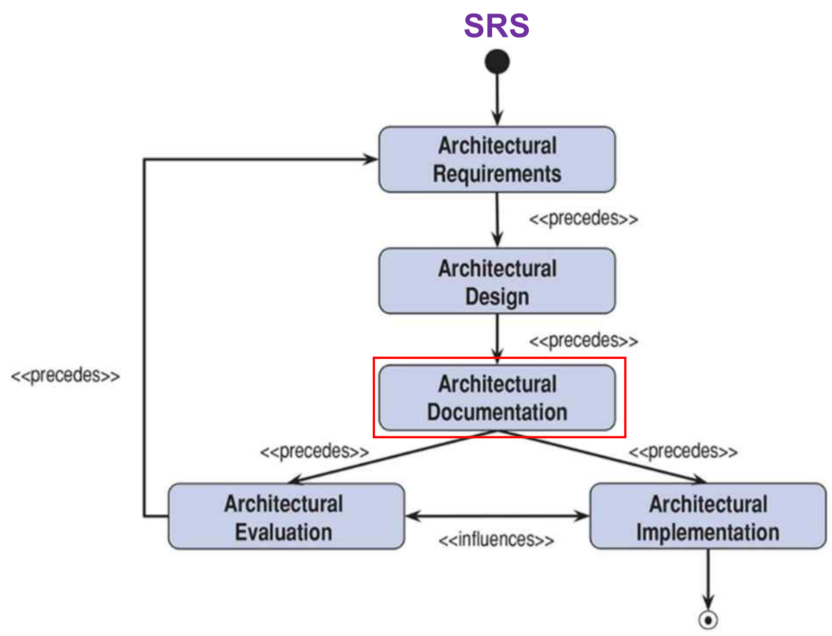


FIGURE 1.1 Software architecture life-cycle activities

Architectural Evaluation

- **Evaluate your architectural design** to ensure that the decision made are **appropriate to address all ASRs**
  - Typically done informally and internally.
  - But for important project, it is advisable to have a formal evaluation done by an external team.

  - Example :
    - **ATAM (Architecture Trade-off Analysis Method)**

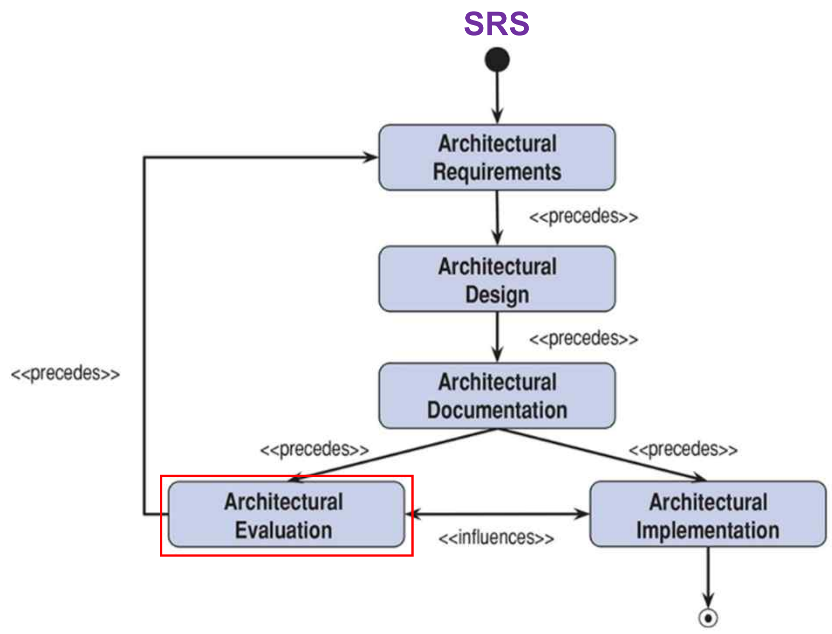


FIGURE 1.1 Software architecture life-cycle activities

Architectural Implementation

- **Implementing** the architecture that you have created and evaluated

- <u>Low-level design</u> and <u>coding</u> are often very closely intertwined.
  - **Low-level design** : Detailed component designs
    - OOD (Object-Oriented Design)
    - SD (Structured Design)
  - **Implementation (Coding)** :
    - OOD → Object-Oriented Programming (C++ / Java)
    - SD → Procedural Programming (C / Fortran)

- **Refactoring**
  - Considering <u>reuse of codes</u> for **Maintainability**
    - Agile
    - Code Review, TDD, CI/CD, Refactoring
    - Design Patterns

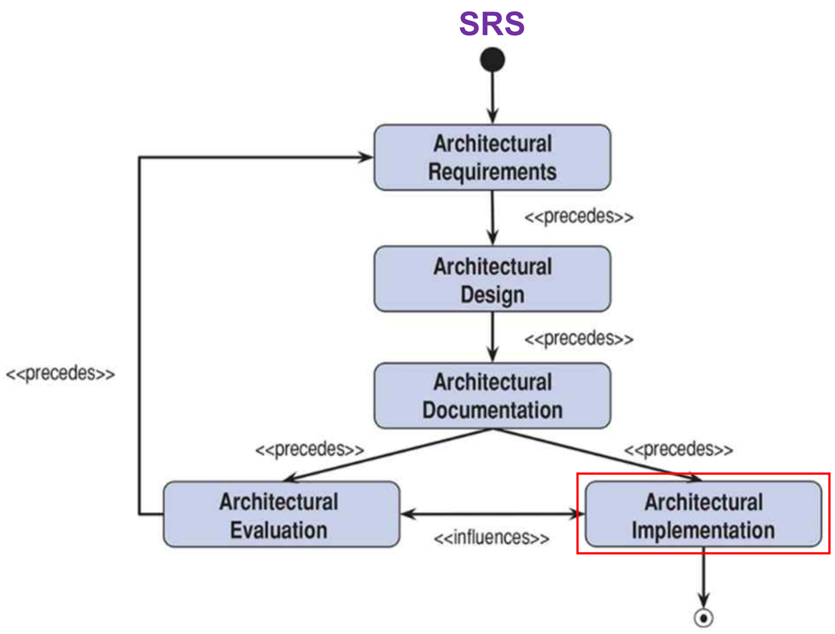
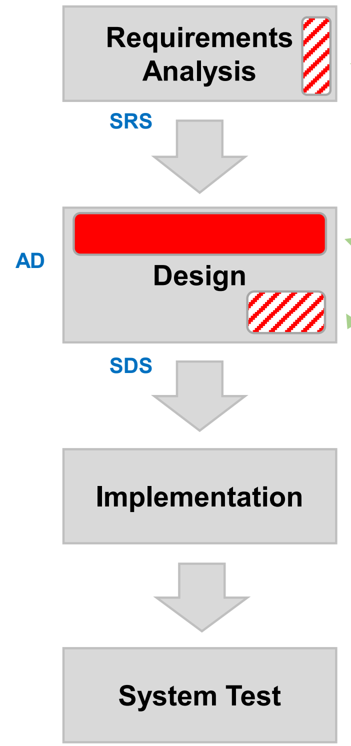


FIGURE 1.1 Software architecture life-cycle activities

CI/CD : Continuous Integration / Continuous Deployment
TDD : Test Driven Development

Scope of Software Architecture Design



- **Requirement Engineering**
  - Stakeholder analysis
  - Identifying user requirements and specifying system requirements
  - Analyzing ASR

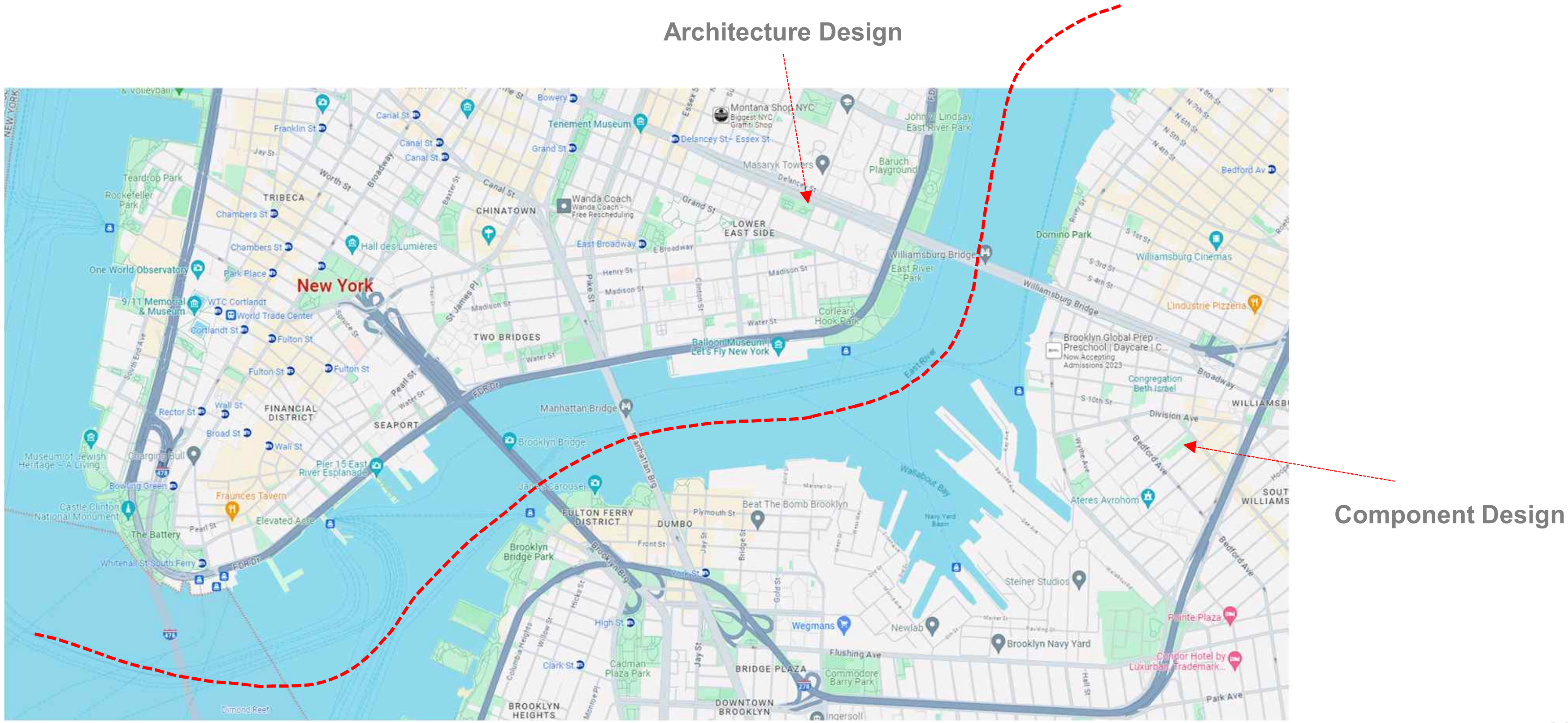
- **Architectural Design**
  - Design of structures that allow architectural drivers to be satisfied
- **Element Interaction Design**
  - Identification of additional elements and their interfaces
- **Deployment of System Elements** (Artifacts) **into Hardware**

- **Element Internals Design (Detailed Component Design)**
  - Interface implementation through OOD/SD

SRS : Software Requirement Specification
SDS : Software Design Specification
AD : Architecture Description

Interactions between Architecture and Component Designs

- **Sometimes** architecture design is affected by component design in reverse direction.
  - Then details of the component design will become the concerns of software architects.



Importance of Architectural Design

- There is a very high cost to a project of not making certain design decisions, or of not making them early enough.
  - Without doing some architectural thinking and some early design work, you cannot confidently predict project cost, schedule, and quality.

- The architecture will influence, but not determine, other decisions that are not in and of themselves design decisions.
  - E.g. Selection of tools, structuring of development environment

- 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 shared resources.

Software Architecture Design in a Nutshell

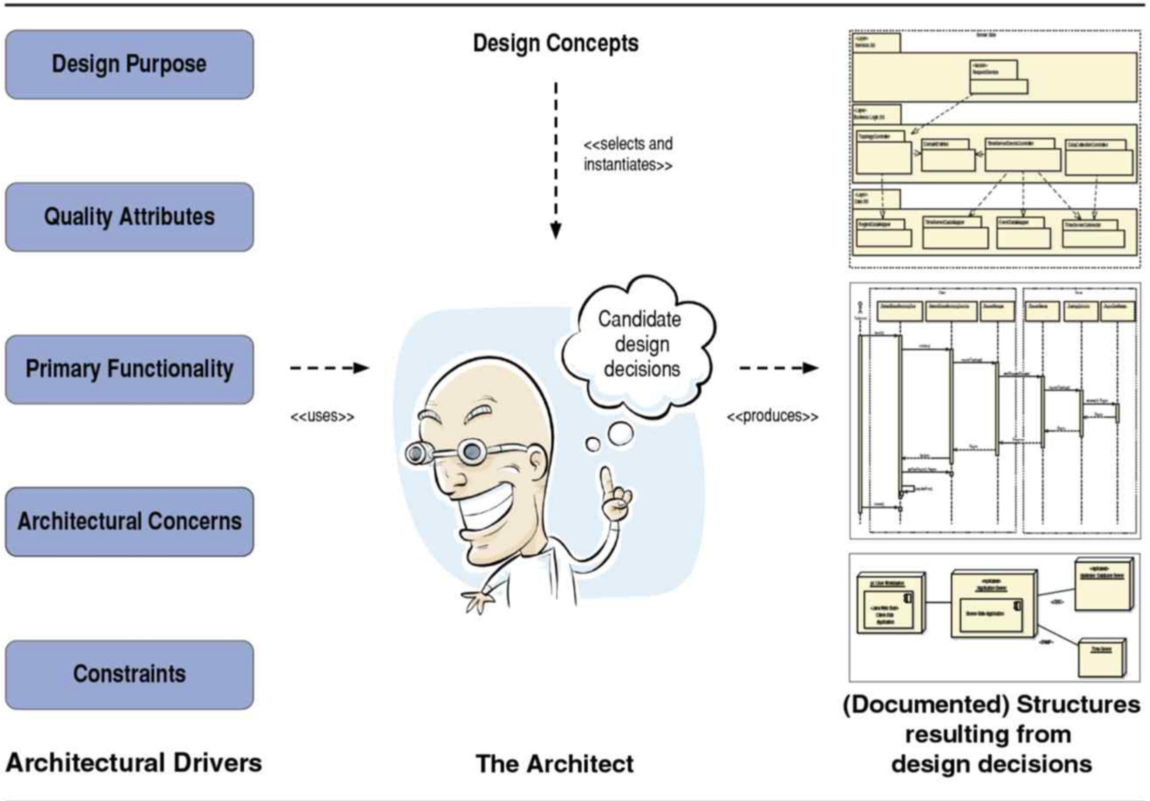
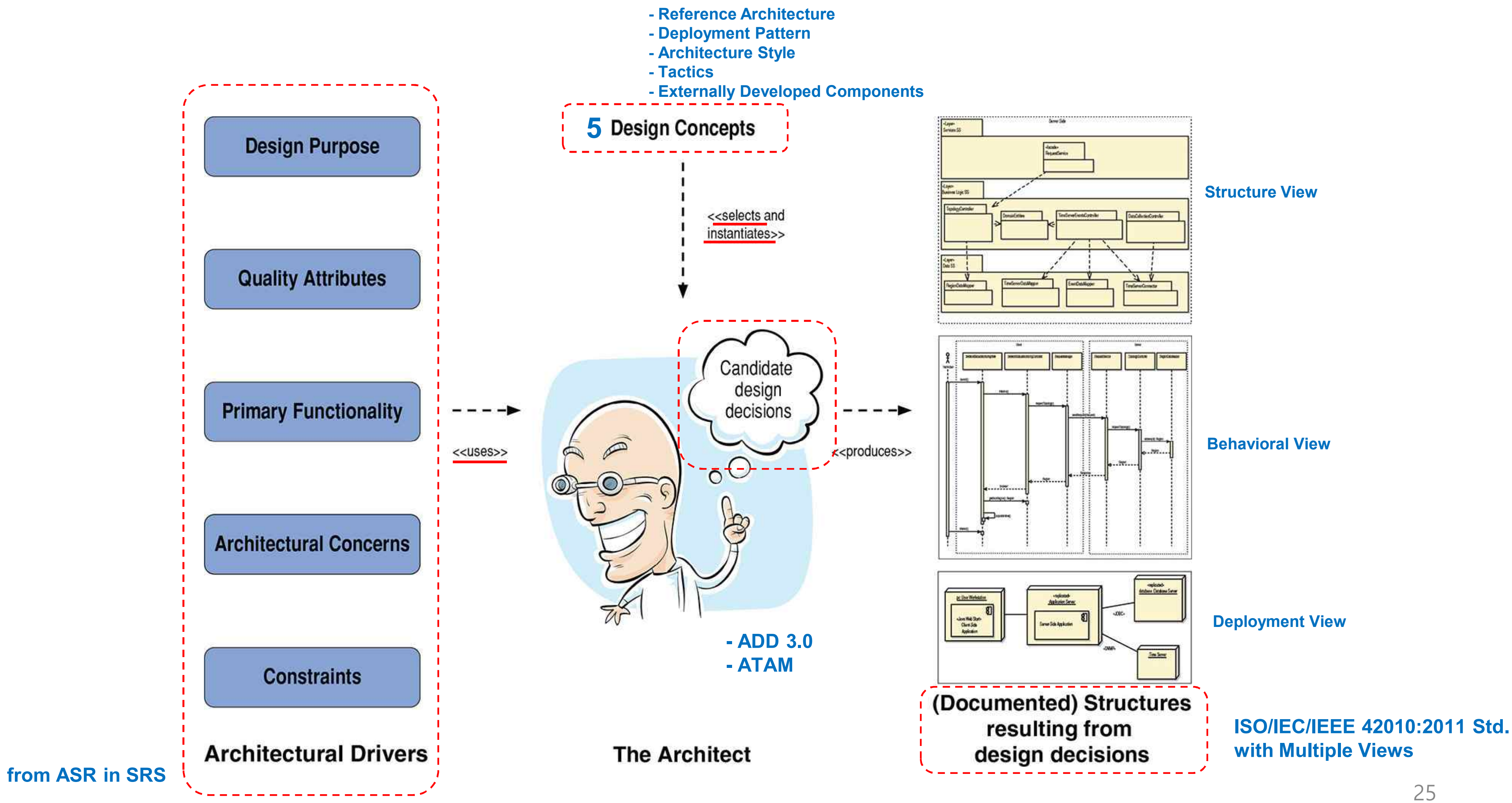


FIGURE 2.1 Overview of the architecture design activity
(Architect image © Brett Lamb | Dreamstime.com)

Software Architecture Design in a Nutshell



Software Architecture Design Approach in a Nutshell

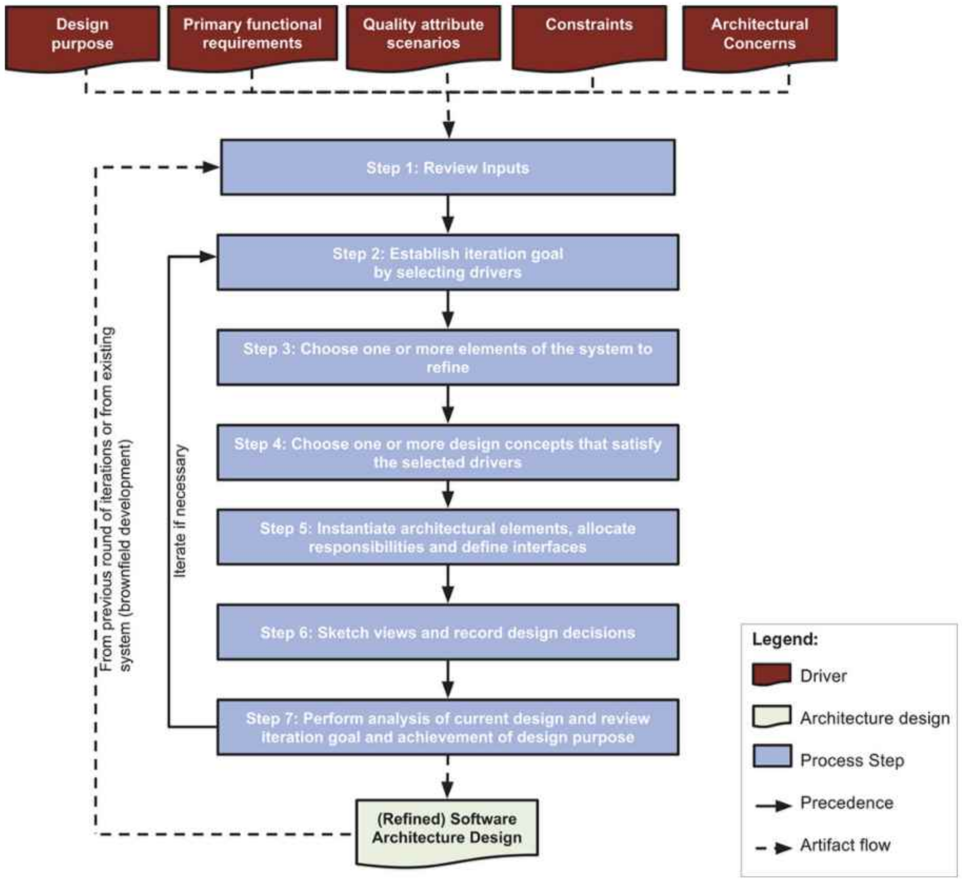


FIGURE 3.1 Steps and artifacts of ADD version 3.0

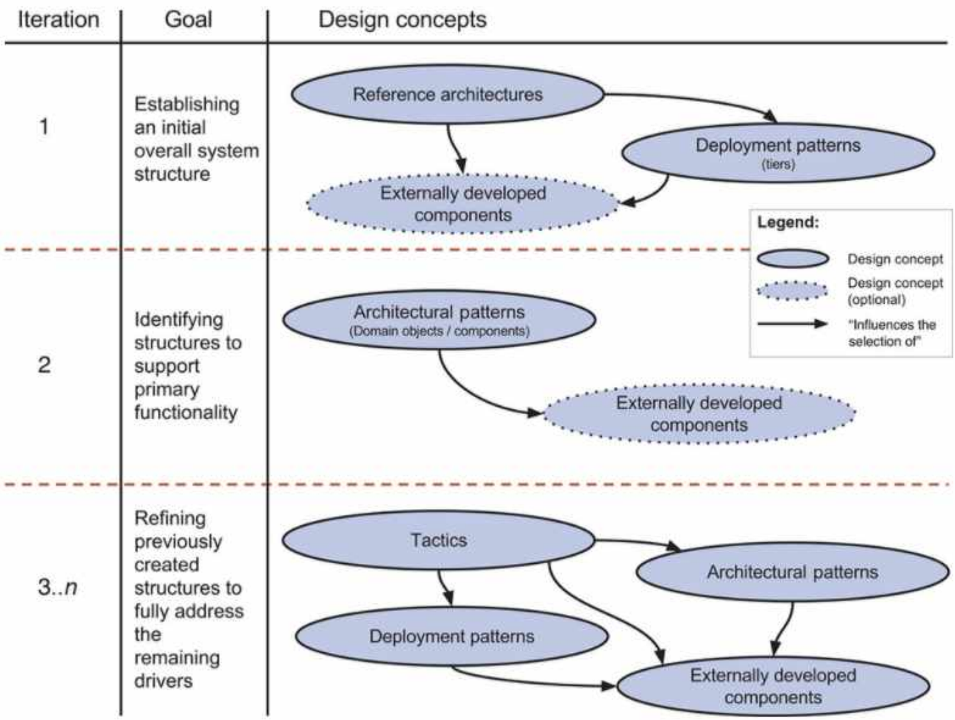
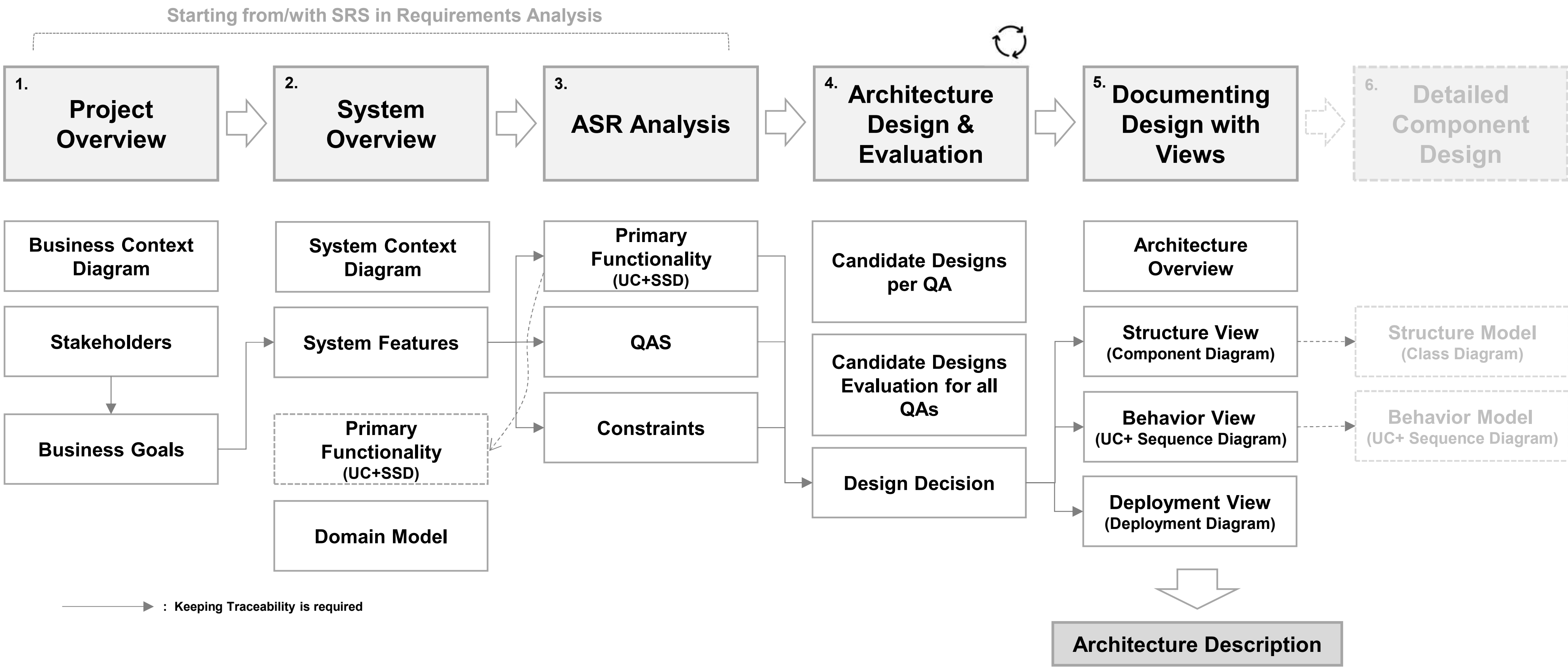
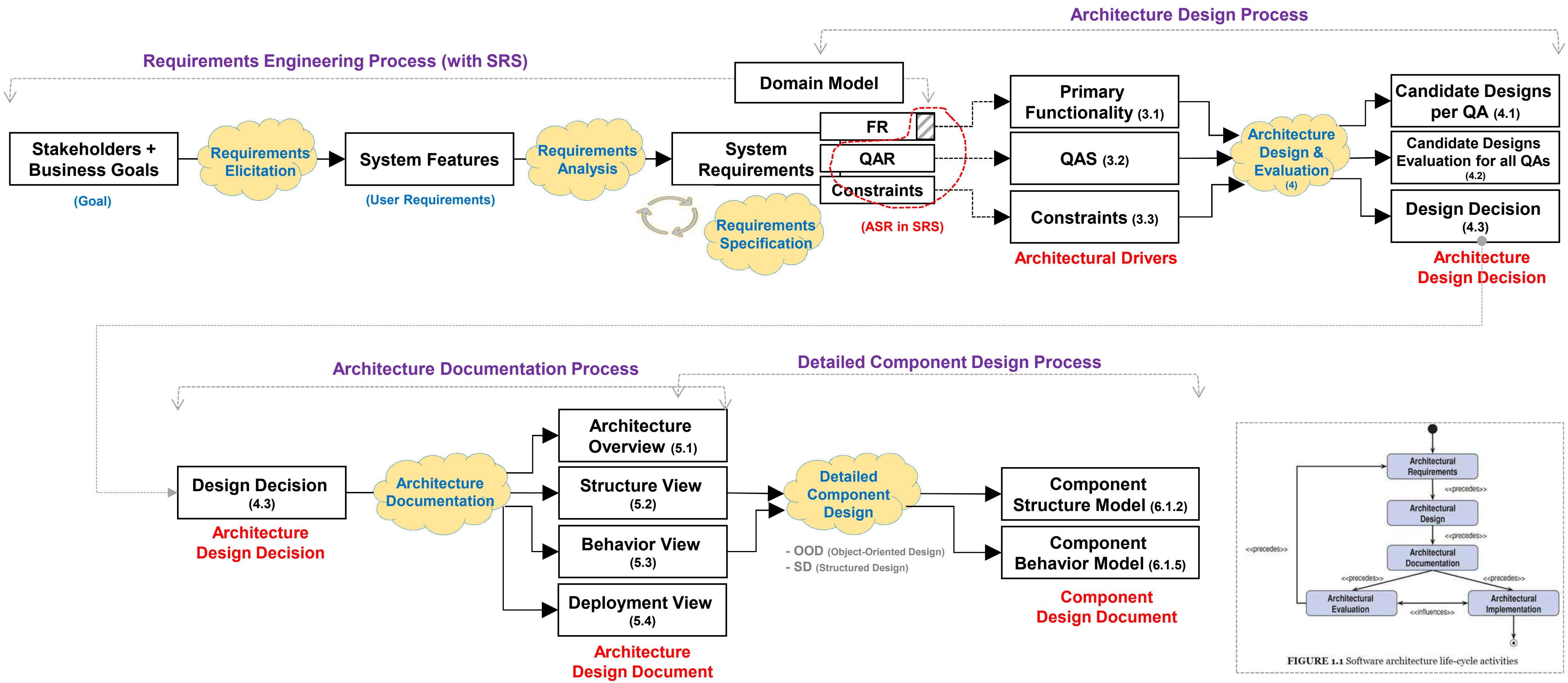


FIGURE 3.2 Design concept selection roadmap for greenfield systems

Our Software Architecture Design Process in CEP



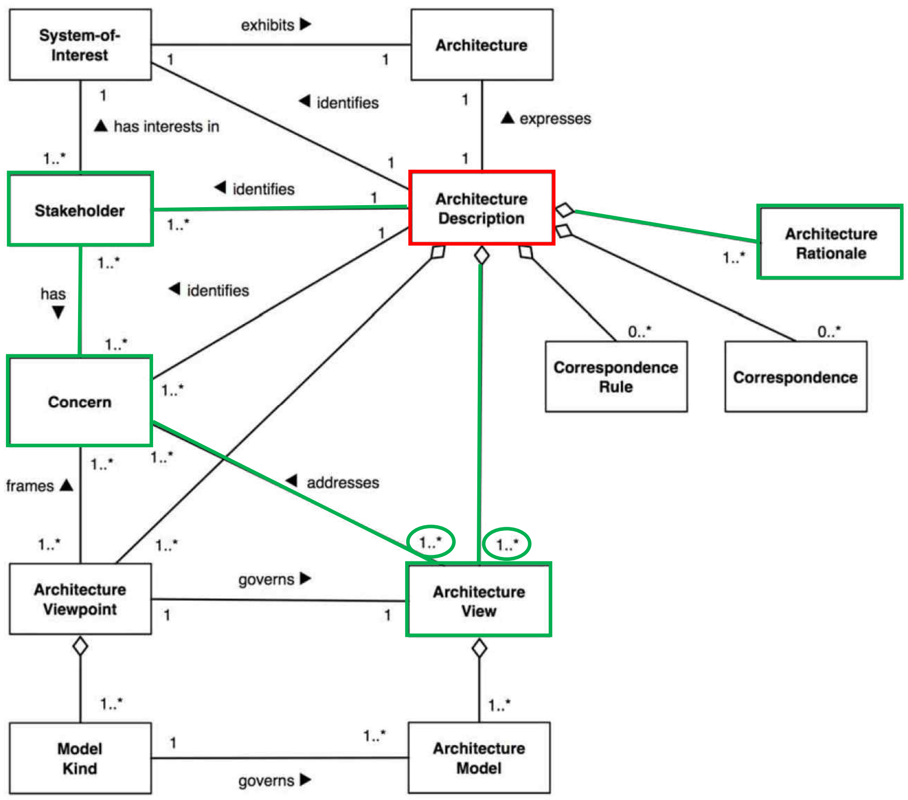
Our Overall Process in CEP



Architecture Description (AD)

- **ISO/IEC/IEEE 42010:2011** "Systems and Software Engineering - Architecture Description"
  - Specifying requirements to be an architectural description (AD)

- AD should demonstrate how an architecture meets the needs of the system's diverse stakeholders.



Organizing Our Architecture Description

1. Project Overview
    - 1.1 Project Background
    - 1.2 Business Context Diagram
    - 1.3 Stakeholders
    - 1.4 Business Goals

2. System Overview
    - 2.1 System Context Diagram
    - 2.2 External Entity
    - 2.3 External Interface
    - 2.4 System Features
    - 2.5 Domain Model
    - 2.6 Assumptions

3. Architectural Drivers
    - 3.1 Primary Functionality
    - 3.2 Quality Attribute Scenario
    - 3.3 Constraints

4. Architecture Design & Evaluation
    - 4.1 Candidate Designs per QA
    - 4.2 Candidate Designs Evaluation for all QAs
    - 4.3 Design Decision

5. Architecture Design Description
    - 5.1 Architecture Overview
    - 5.2 Structure View
    - 5.3 Behavior View
    - 5.4 Deployment View

6. Component Design Description
    - 6.1.2 Component Structure Model
    - 6.1.5 Component Behavior Model

7. Architecture Traceability Summary

What Matters Most in Architecture Design?

- We should be able to check this question with **full traceability**.

추적성 (Traceability)

가장 좋은 (The Best) 방법은 아닐 수 있지만,
현재 소프트웨어공학에서 제공할 수 있는 기법 중,
가장 Feasible 하다고 인정됩니다.

추가적으로, 추적성을 구체적으로 활용해서
"Safety Case"와 같은 Demonstration을 할 수 있어요.

How well does our architecture design address all stakeholders' concerns?

Stakeholders and Concern

- 이해관계자 / 이해당사자
- 주주
- 표준 및 규제 기관

- 고객 (Customer)
- 고객사 (Client)
- 소비자 (Consumer)

- 개발자
- 마케팅 담당자

Stakeholders Requirements Analysis (FR + QA)

QA

- Usability (사용성)
- Reliability (신뢰성)
- Availability (가용성)
- Security (보안성)
- Performance (성능)
- Maintainability (유지보수성)
- Testability
- Modifiability
- Interoperability
- …

Architectural Drivers

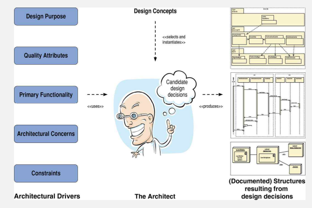
Primary Functionality (Use Case + SSD)

QAS

Constraints

Architecture Design

ADD 3.0 + ATAM



Architecture Description

Design Decision with Multiple Design Views

Design Rationale

UML Diagrams

Software Architecture Design

Software Architecture Design

through a set of Views
(Structure, Behavior, Deployment)

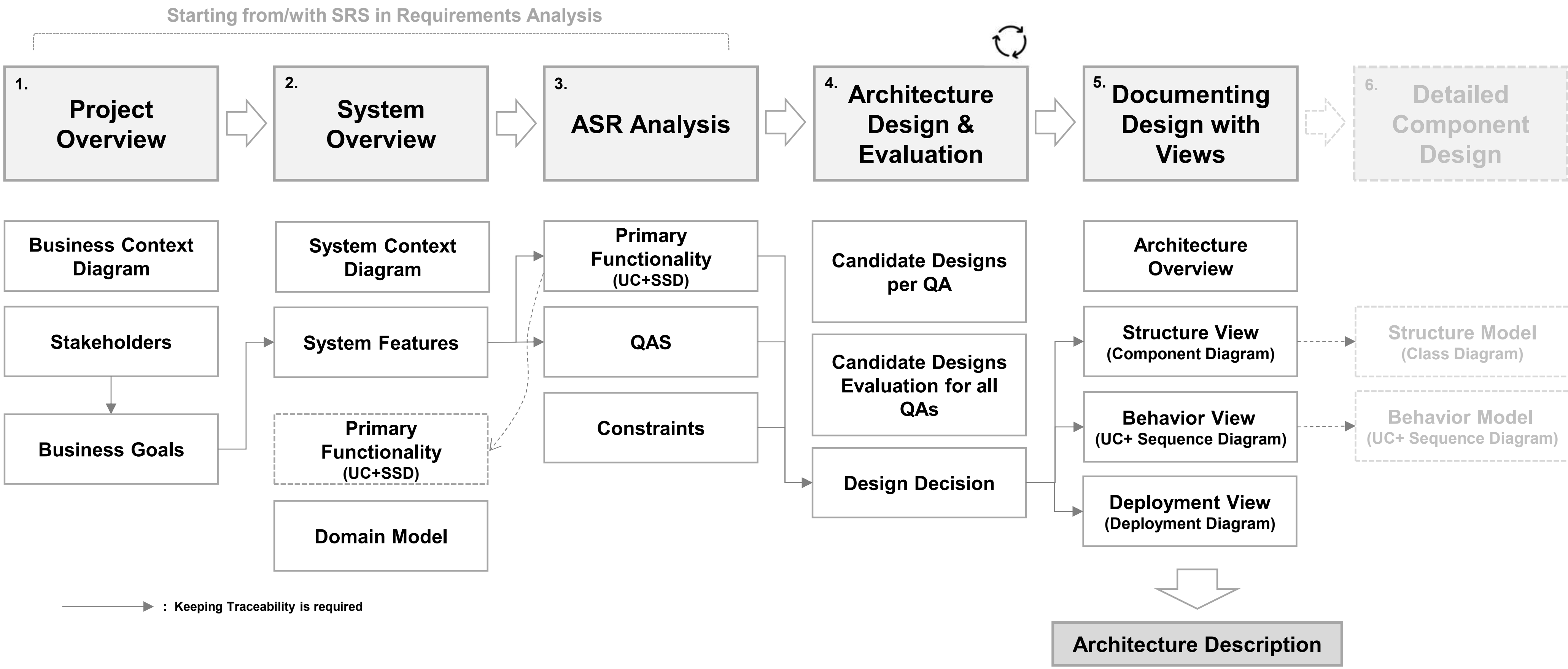
- **The process of creating a <u>specification</u>** of software elements, intended to accomplish <u>goals</u>, subject to <u>constraints</u>.

ASR - Architectural Drivers
(Constraints)

AD (Architecture Description)

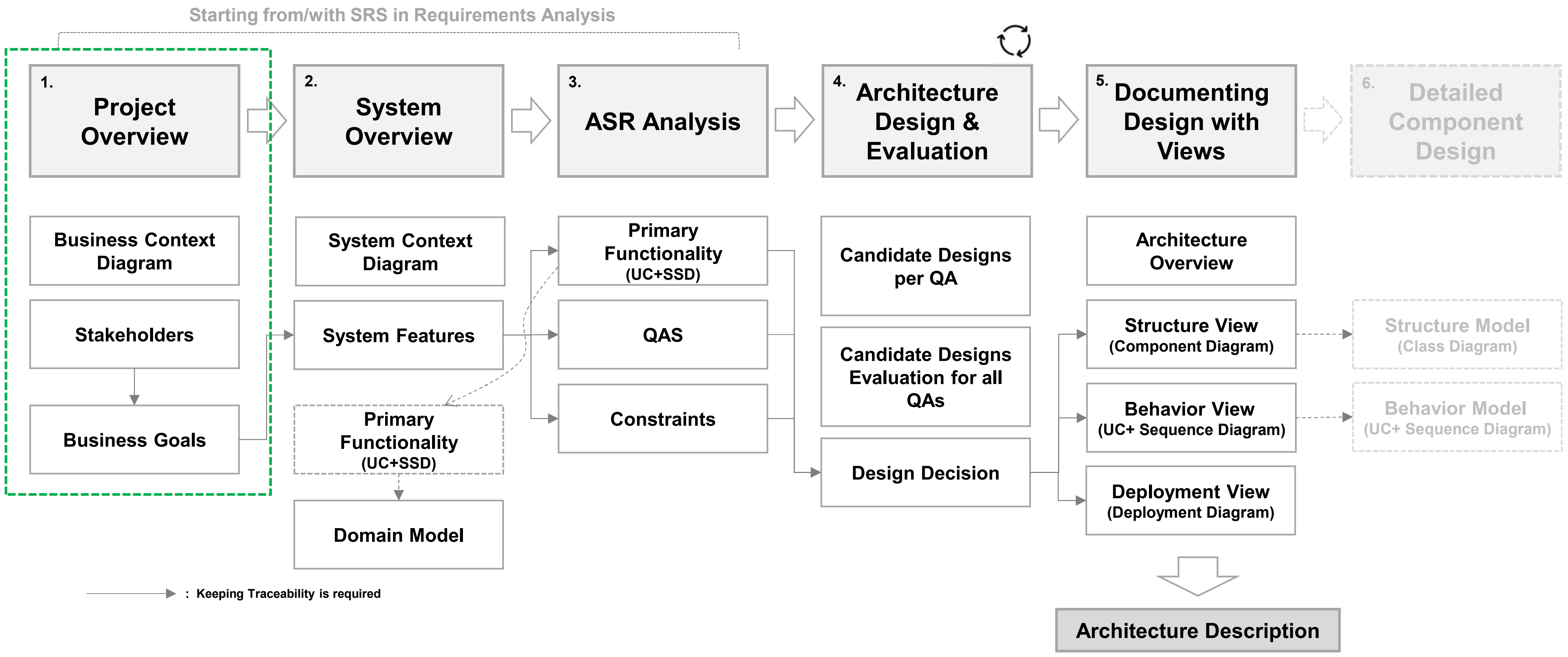
ASR - Architectural Drivers
(Primary Functionality, QAS)

The Software Architecture Design Process in CEP



1. Project Overview

1. Project Overview



Where We are Now in AD

1. Project Overview
   - 1.1 Project Background
   - 1.2 Business Context Diagram
   - 1.3 Stakeholders
   - 1.4 Business Goals
2. System Overview
   - 2.1 System Context Diagram
   - 2.2 External Entity
   - 2.3 External Interface
   - 2.4 System Features
   - 2.5 Domain Model
   - 2.6 Assumptions
3. Architectural Drivers
   - 3.1 Primary Functionality
   - 3.2 Quality Attribute Scenario
   - 3.3 Constraints
4. Architecture Design & Evaluation
   - 4.1 Candidate Designs per QA
   - 4.2 Candidate Designs Evaluation for all QAs
   - 4.3 Design Decision
5. Architecture Design Description
   - 5.1 Architecture Overview
   - 5.2 Structure View
   - 5.3 Behavior View
   - 5.4 Deployment View
6. Component Design Description
   - 6.1.2 Component Structure Model
   - 6.1.5 Component Behavior Model
7. Architecture Evaluation Summary

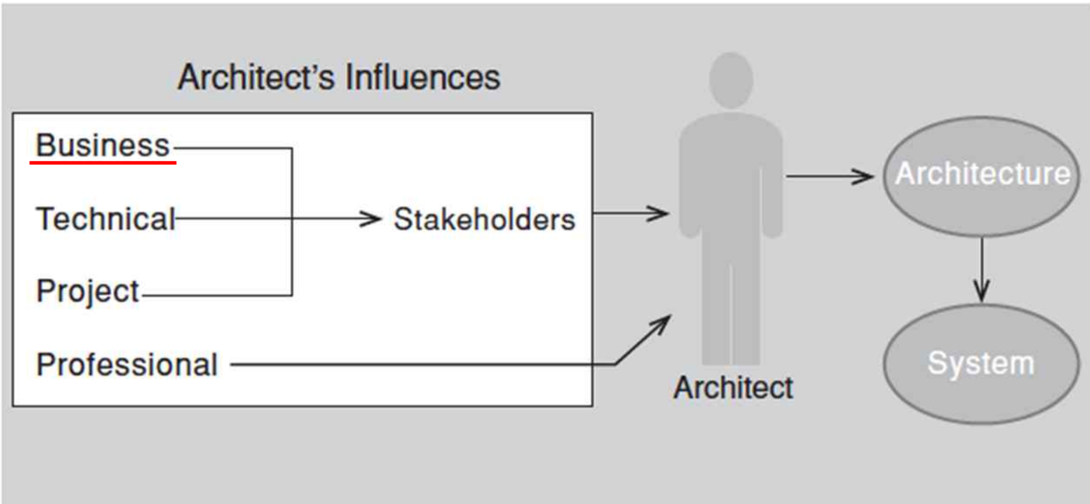
1.1 Project Background

- Describes the project, its purpose, and scope
  - Background information on the system domain to help stakeholders understand the project and the system

1.2 Business Context Diagram

- **Stakeholder Analysis**
  - Identifying all stakeholders of the system and what their goals, which will have a profound influence on the system architecture
  - Example : Business Context Diagram

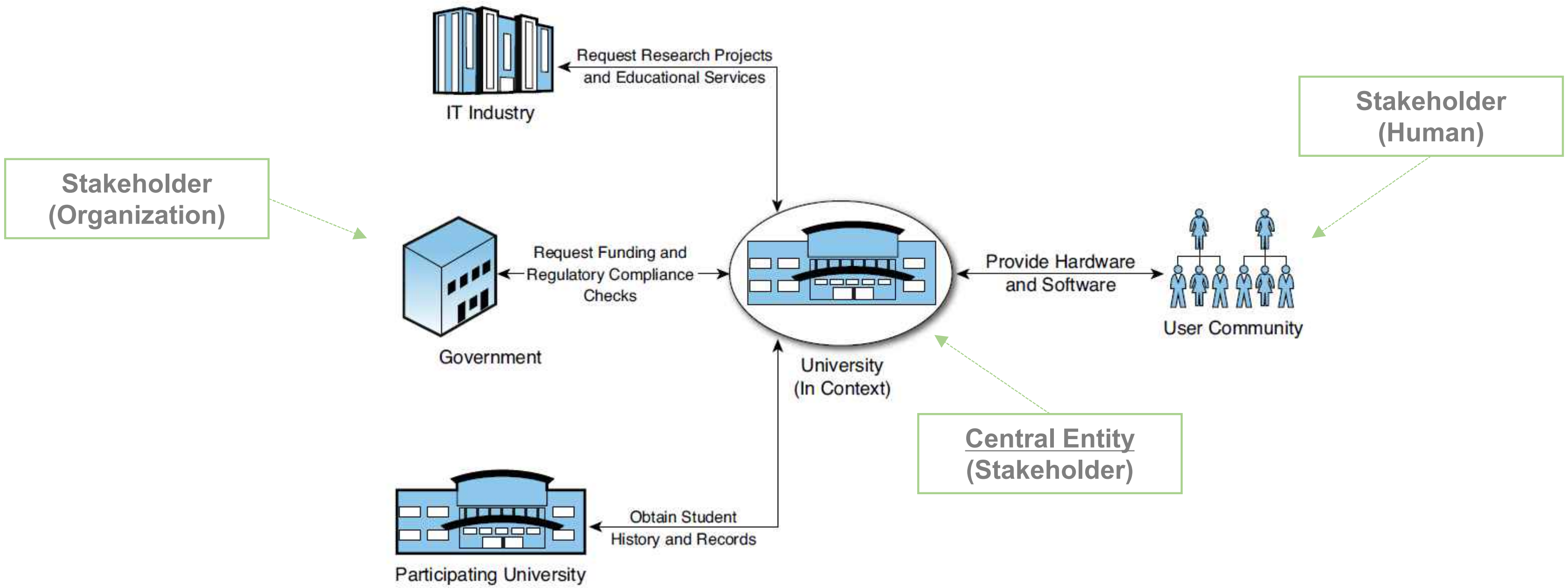
- Systems are created to satisfy the **business goals** of **stakeholders**.



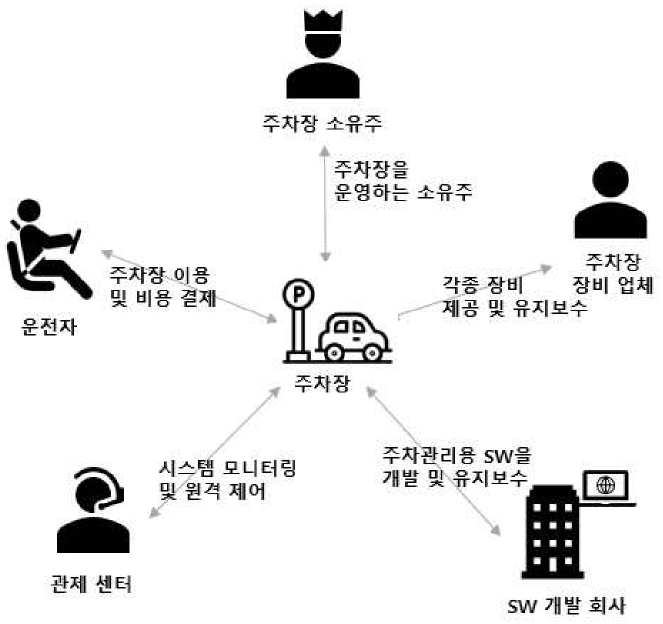
Business Context Diagram

- An **organizational-level view** of
  - How organizations/stakeholders are related to each other
  - What information exchange between them

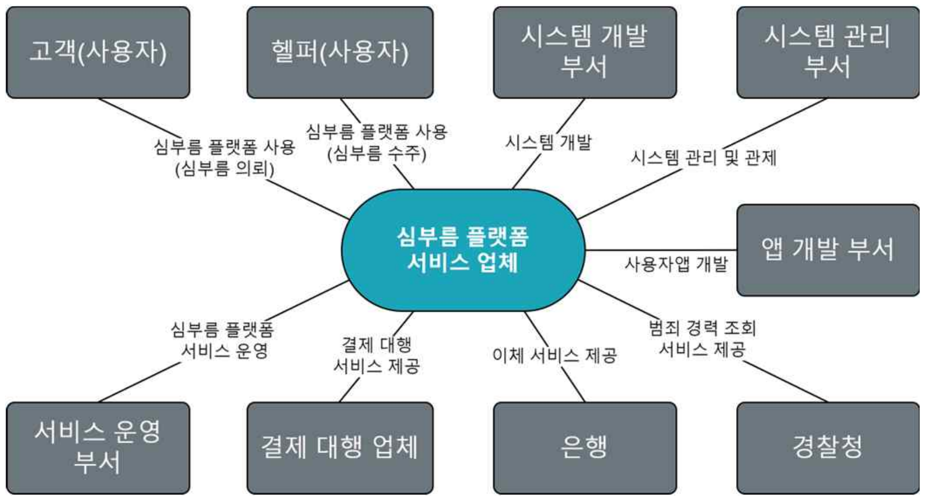
- Example: Building a software for a university



Business Context Diagram : Examples



주차장 관리 시스템



심부름 플랫폼

1.3 Stakeholders

- **Stakeholder** is
  - a person, group, or entity with an interest in or concerns about the realization of the architecture, or
  - a party having a right, share or claim in a system or in its possession of characteristics that meet that party's needs and expectations.
  - All entities that you identified in the business context diagram

Stakeholder List

- Explaining all stakeholders in business context diagram and their concerns/interests

- Example :

Stakeholder	Description
고객(사용자)	**설명:** 심부름 플랫폼 서비스를 이용하여 심부름을 의뢰하려는 사용자 **관심사:** 특정한 업무를 수행하기에 시간적, 거리적 한계가 있을 때, 요금 지불을 통해 직접 수행 불가한 업무를 대행(심부름)할 대리자(헬퍼)를 찾고 싶음. 심부름에 대하여 합리적인 요금을 지불하기 원함. 대행 요청할 업무의 특성(카테고리)에 따라 헬퍼를 쉽고 빠르게 찾기를 원하며, 대리자가 수행하는 심부름의 과정부터 결과까지 편리하게 관리/감독하기를 기대함. 심부름의 결과 수준이 높기를 기대함.

1.4 Business Goals

- **Business goals** are the **primary influencing factors** on the architecture.
  - Should be captured explicitly because they often imply ASRs.

  - No organization builds a system without a reason.

  - Business Goals = Mission Objectives
    - Example:
      - *“What are our ambitions about market share for this product?”*
      - *“How could the architecture contribute to meet them?”*

Category of Business Goals

Category	Goal Examples
Managing product's quality and reputation	System helps improve - Branding, reduce recalls, support certain types of users, quality and testing support. and strategies
Meeting financial objectives	System meets financial objectives through - Revenue generation, business process efficiency, reduced training costs, higher shareholder dividends
Organization's growth and continuity	System promotes growth and continuity through - Market share increase, product line creation and success, international sales, long-term business sustenance
Meeting responsibility to employees	System fulfills responsibilities to employee through - Improved operator safety, reduced workload, and opportunity for learning new development skills
Meeting responsibility to society	System fulfills responsibilities to a society through - Compliance with laws and regulations, those related to ethics, safety, security, privacy, and green computing
Meeting responsibility to country	System fulfills responsibilities to a country through - Compliance with export controls and regulatory conformance

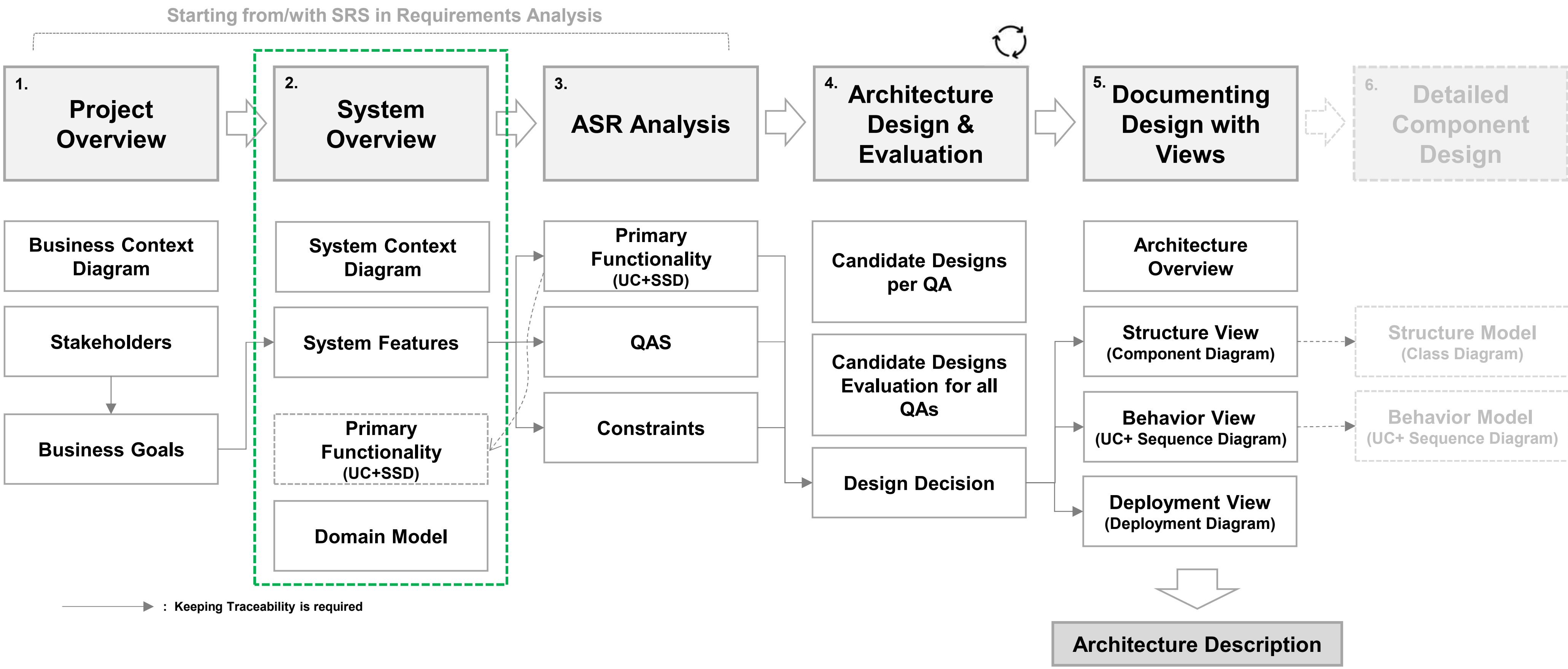
Expressing Business Goals

- All business goals should be expressed clearly in a consistent fashion and contain sufficient information to enable their shared understanding by relevant stakeholders.
  - Expressed for each stakeholder
  - **Traceability** : Stakeholder → Business Goal
- Examples :
  - Owner : *"His family's stock in the company will rise by 5%."*
  - Portfolio manager : *"The company will make the portfolio 30% more profitable."*
  - Project manager : *"Customer satisfaction will rise by 10%."*

Stakeholder	**Business Goal**		
	ID	Statement	Importance
운전자	BG-01	운전자는 입·출차 및 주차 요금 정산 등 전반적인 주차장 이용에서 더 만족스러운 주차 경험을 기대함. - 주차장을 찾고, 입·출차 및 정산하는 전체 시간을 20% 단축하기를 원함.	상

2. System Overview

2. System Overview



Where We are Now in AD

1. Project Overview
    - 1.1 Project Background
    - 1.2 Business Context Diagram
    - 1.3 Stakeholders
    - 1.4 Business Goals

2. System Overview
    - 2.1 System Context Diagram
    - 2.2 External Entity
    - 2.3 External Interface
    - 2.4 System Features
    - 2.5 Domain Model
    - 2.6 Assumptions

3. Architectural Drivers
    - 3.1 Primary Functionality
    - 3.2 Quality Attribute Scenario
    - 3.3 Constraints

4. Architecture Design & Evaluation
    - 4.1 Candidate Designs per QA
    - 4.2 Candidate Designs Evaluation for all QAs
    - 4.3 Design Decision

5. Architecture Design Description
    - 5.1 Architecture Overview
    - 5.2 Structure View
    - 5.3 Behavior View
    - 5.4 Deployment View

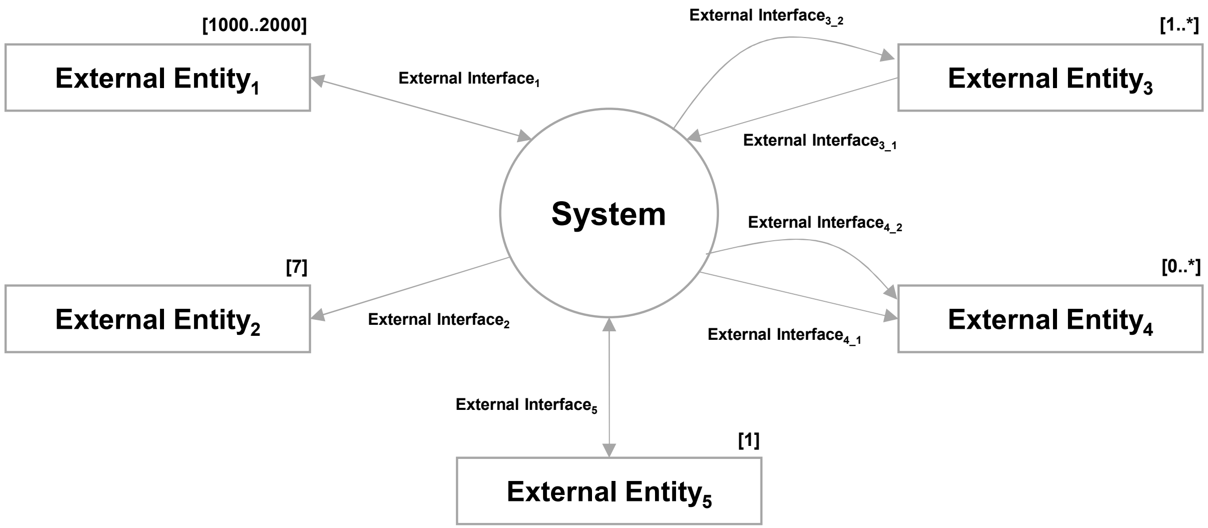
6. Component Design Description
    - 6.1.2 Component Structure Model
    - 6.1.5 Component Behavior Model

7. Architecture Traceability Summary

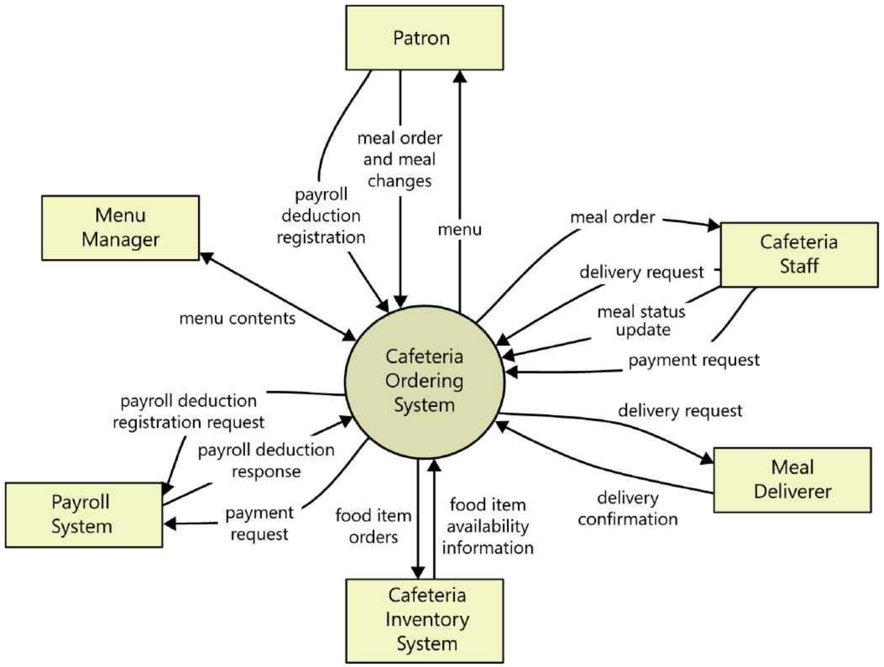
2.1 System Context Diagram

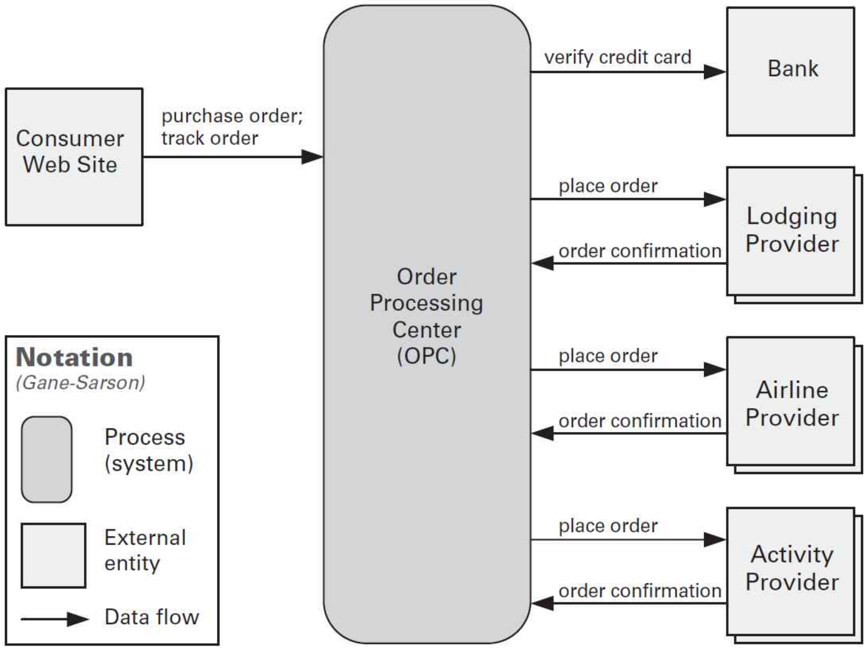
- The system tries to implement the **business goal** within **system context (boundary)**.
  - A good system context model is an essential part of an effective architecture document.

- **System Context Diagram** defines
  - What is in scope? → System Features (2.4)
  - What is out of scope? → External Entities (2.2)
  - How the system related to its environment? → External Interfaces (2.3)



System Context Diagram : Examples





Guidelines for System Context Diagram

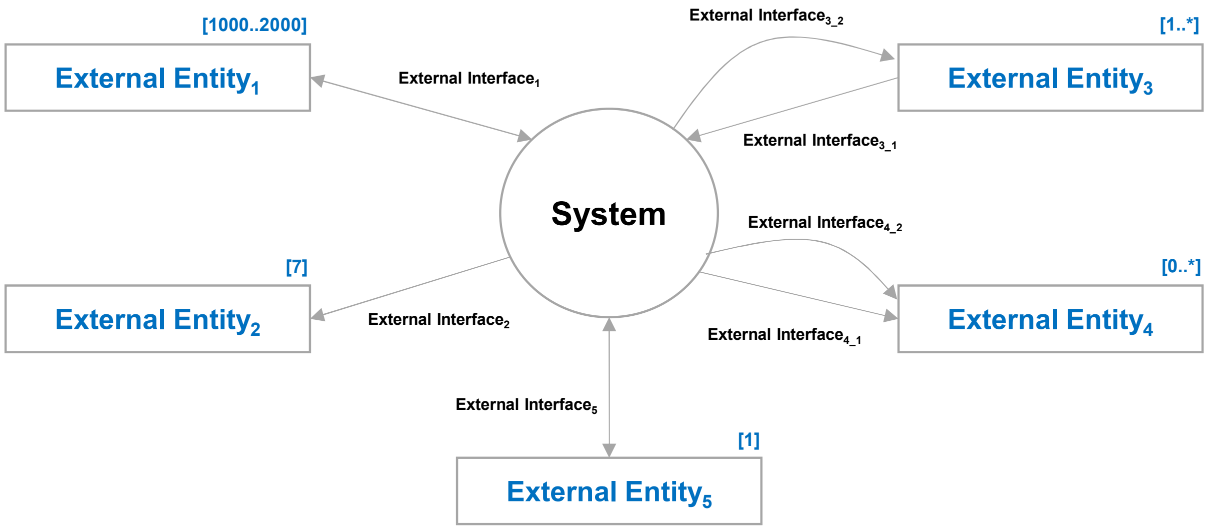
- NOT disclose any architecture detail about the system
  - It just appears as an undecomposed block.
  - In practice, it may show some internal structure of the system being put in context. → **Domain Model (2.5)**

- NOT show any temporal information
  - E.g., order of interactions, flow of data

- NOT show the conditions under which data is transferred, stimuli fired, and messages transmitted
  - Each interface needs to be just "assigned" to one of the system's architecture elements.

2.2 External Entity

- An **external entity** is any person, system, or device with which the system directly interacts.
  - User : User, a class of user, or some other person or role
  - External system : Another system that runs in another organization
  - Internal system : Another system that runs in the same organization as the system being modeled
  - Gateway component : Gateway or other implementation component that has the effect of hiding other systems (internal or external)
  - Data store : Data store that is external to the system (e.g., a shared database, data warehouse)



External Entities Affect System Architecture

- The quality of external entities (e.g., reliability, availability, or performance) may significantly affect the architecture of the system.

  - Example :

A travel booking system exchanges information with many other systems located around the world. Some of these systems may be only intermittently available, because of time zone differences or because they are more liable to failure.

The travel system's interfaces with external systems will therefore need to be carefully designed for **reliable** operation.

- All failed interactions should be automatically retried a configurable number of times.
- These retry attempts should be logged to a database so that operational staff can monitor trends.
- It should be possible to restart very large transfers that fail partway through from the point of failure rather than having to retransmit the whole file.

External Entity : Examples

- All quality attributes that may affect the system architecture shall be described in detail.
  - All assumptions on H/W and systems

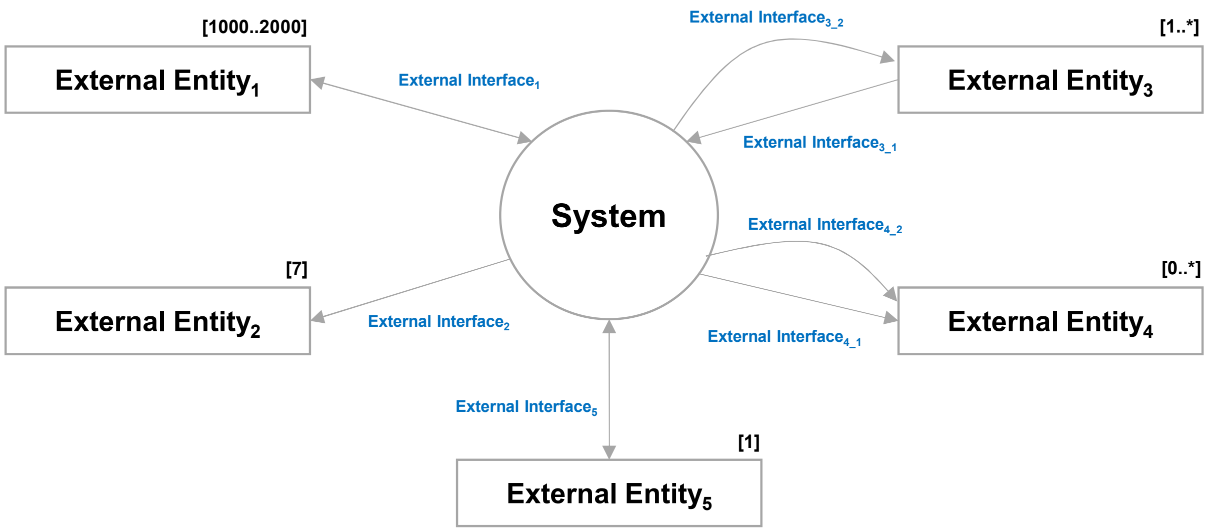
플랫폼 운영자	**유형**: 사용자 **숙련도**: 플랫폼 서비스를 통한 비즈니스 운영에 대한 전문성을 보유하고 있음. 플랫폼 서비스 운영을 위한 시스템 운용이 능숙함. **핵심 기대 사항**: 시스템을 통하여 이용자들의 이용 현황을 파악하고, 관리하기를 원함. 헬퍼 중 범죄 사실이 발생하거나, 불법적인 거래가 발생할 경우 빠른 시간 내에 파악하고 조치하기를 원함.

결제 대행 시스템	**유형**: 시스템 **역할**: 시스템에서 주차 요금 결제 요청 시 각종 카드 및 간편 결제 시스템에 결제 및 지불을 대행해주는 외부 시스템 **관련 Stakeholder**: 결제 대행 업체 **시스템 사양**: Cloud 서비스 운영 가능한 서버 **시스템의 품질 수준**: - 가용성: 99.9% (카드 회사, 간편 결제 회사 별 시스템 점검 시간 존재) - 전자금융거래의 안전성과 신뢰성: 금융감독위원회가 정하는 기준을 준수

입구/출구 안내판	**유형**: 장치 **역할**: 주차장 출입구에 설치되어 시스템이 안내하고자 하는 내용을 LED로 출력 **관련 Stakeholder**: 관제센터, SW개발회사 **장치 사양**: - Control H/W: 아두이노 수준 - LED Display board (문자 표기: 한글/영문/숫자 출력 가능) - Connectivity: Ethernet, Wi-Fi **장치의 품질 수준**: - 요청 수신 0.1초 이내에 안내판에 출력

2.3 External Interface

- **All interfaces** between the system and each external entity should be identified.
  - **Data** provider or consumer : The external entity supplies data directly to the system or receives data directly from it.
  - **Event** provider or consumer : The external entity publishes events that this system wishes to be notified of, or this system publishes events that the external entity wishes to be notified.
  - **Service** provider or consumer : The external entity is requested to perform some action by the system or requests some action of the system, and the system may return data and/or status information in response to the request.



External Interfaces Affect System Architecture

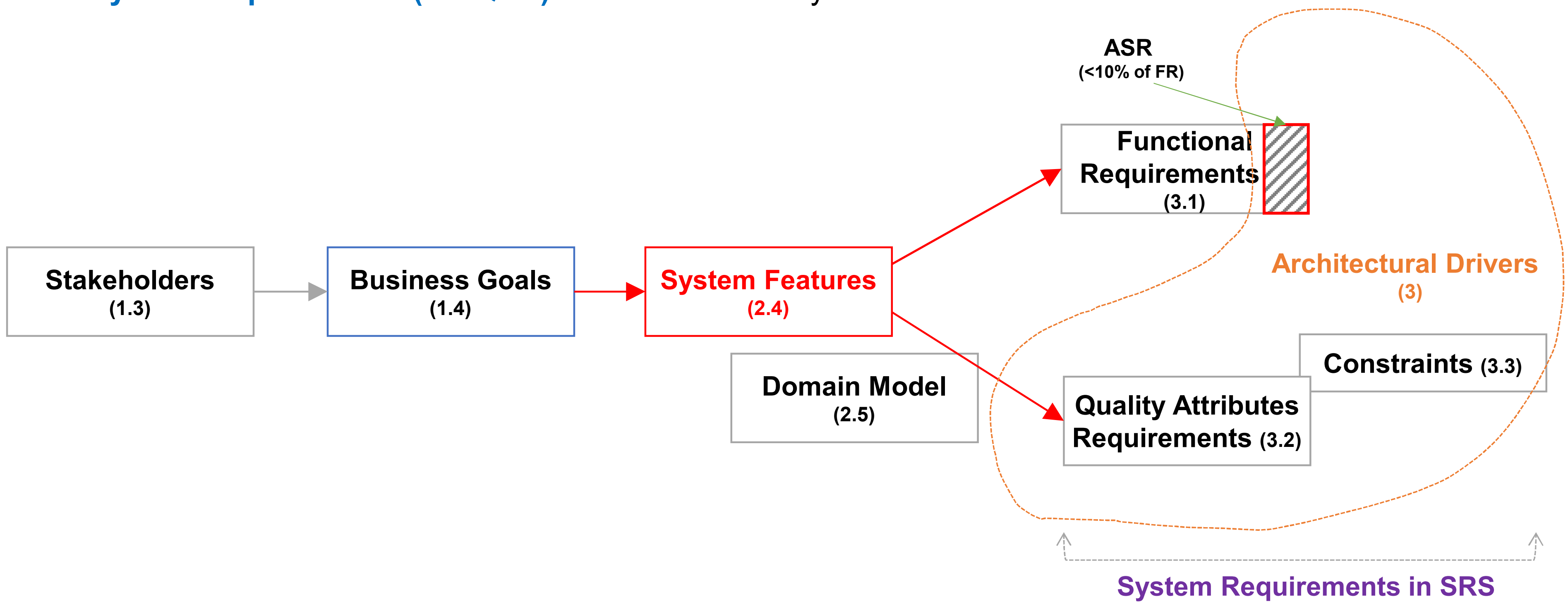
- The quality and characteristics of external interface may have a significant effect on the architecture of the system.
  - **Data** : the content, scope, and meaning of the data to be transferred
  - **Event** : events of interest, their meaning and content, and the volume and timing of their occurrence
  - **Service** : Syntax of the request(name and any parameters), the actions to be taken, any data to be returned, any ack, status, or error information that may be returned, any exception actions to be taken by either side

get parking status	**역할**: 관제 센터 직원이 주차 관리 시스템으로 주차장 운영 현황을 요청 **User interface**: Console (Web UI) **System interface**: HTTPS **특성**: - 사용자 요청 빈도는 일 1회 정도로 낮을 것으로 예상 - 1회 요청 당 10만개 주차장 현황을 조회하므로 시스템 부하 고려 필요

Track errand	**역할**: 고객이 심부름 내역 및 결과(이미지, 동영상)을 조회
	User Interface: Mobile app UI
	System Interface: HTTPS
	특성:
	- 수시 호출 가능함
	- 헬퍼의 DAU(Daily Active User) 22,000명이 1일 심부름 1건 수주한다고 가정하였을 때, 약 22,000건의 심부름 수행 가능
	- 하루 다운로드 22,000 건 * (최대 10장 * 1MB +영상 2건 * 10MB) = 660GB
	  - 평균 60Mbps (약 7.65mb/s) 다운로드 부하
	  - 낮 피크타임을 가정하면 100Mbps 이상 다운로드 부하 가능

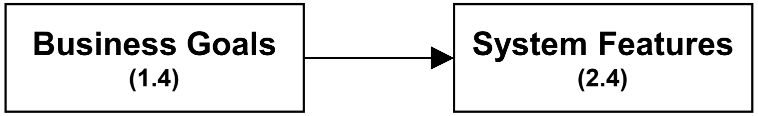
2.4 System Features

- **System features** are the high-level capabilities of the system that are required to deliver by us.
  - **System features** are refined from **Business goals** which we must realize.
  - **System requirements (FR/QAR)** are derived from system features.



Business Goal to System Features

- **Business goals** are refined into **system features**.



- Examples :

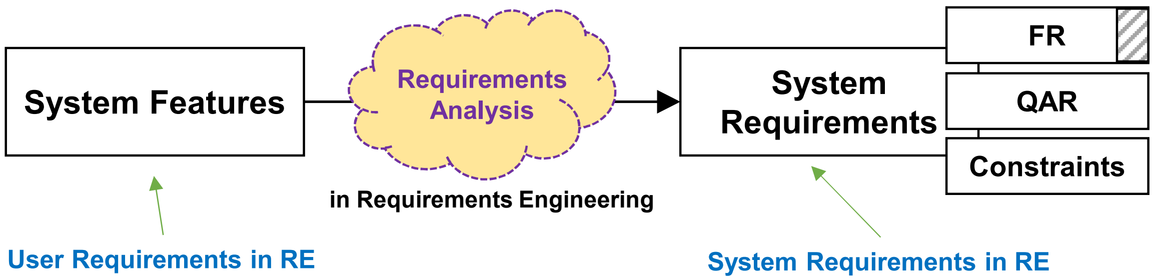
Business Goals	System Features
(I wish to) Expand (our business) by entering new and emerging geographic markets	Support international languages
	Comply with regulations that have an impact on life-critical systems such as fire alarms
(I wish to) Open new sales channels in the form of Value-Added Resellers (VARs)	Support hardware devices from different manufacturers
	Support conversions of nonstandard units used by the different hardware devices

ID	Title	Description	I	Related BG
SF-06	장비 모니터링	주차장의 각종 장비(번호판 카메라, 차단 제어기, 안내판)의 동작 상태를 모니터링 하는 기능	상	BG-03

System Features to System Requirements

- **System Features**
  - Informal statements of capabilities of the system used often for marketing and product-positioning purposes, as a shorthand for a set of behaviors of the system
  - Useful when discussing the system in casual settings
  - Not useful when defining the behavior of the system in precise enough to design, develop, or test the system

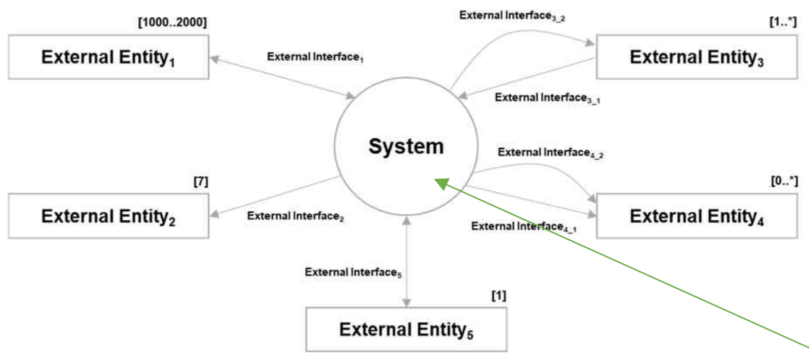
- **System Requirements**
  - Individual statements of conditions and capabilities to which the system must conform
  - Each software requirement is the specification of an externally observable behavior of the system.
  - Detailed and unambiguous requirements that are specific enough to direct the implementation and testing of the system
    - Functional requirements , Non-Functional requirements (Quality attribute requirements, Constraints)



* QAR is not a widely used term. It's usually referred to as QA or Quality Requirements.

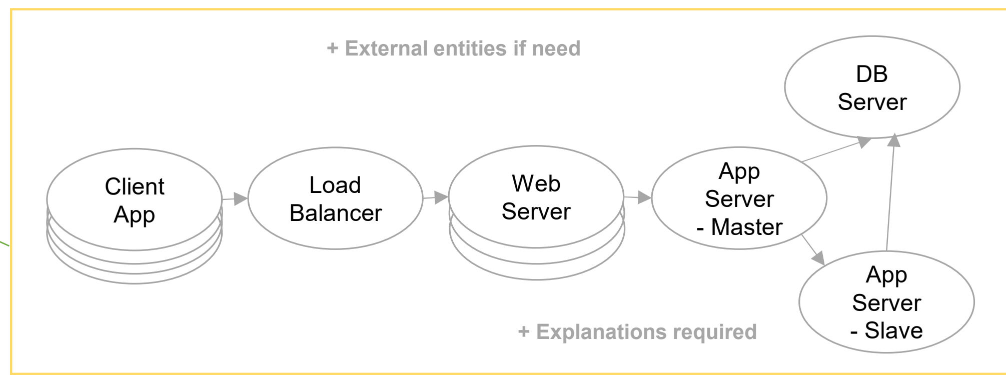
2.5 Domain Model

- **Domain model** is a **conceptual model** for understanding **functional requirements**.
  - Any kind of model is possible: naive diagrams, UML component/deployment diagram
  - Decomposing the system into subsystems and components logically, functionally, or physically
    - To satisfy important functional requirements, i.e., ASR → Primary functionality
  - Playing as a preliminary version of architecture diagram like the domain model in OOAD
    - Based on the information architects have known so far

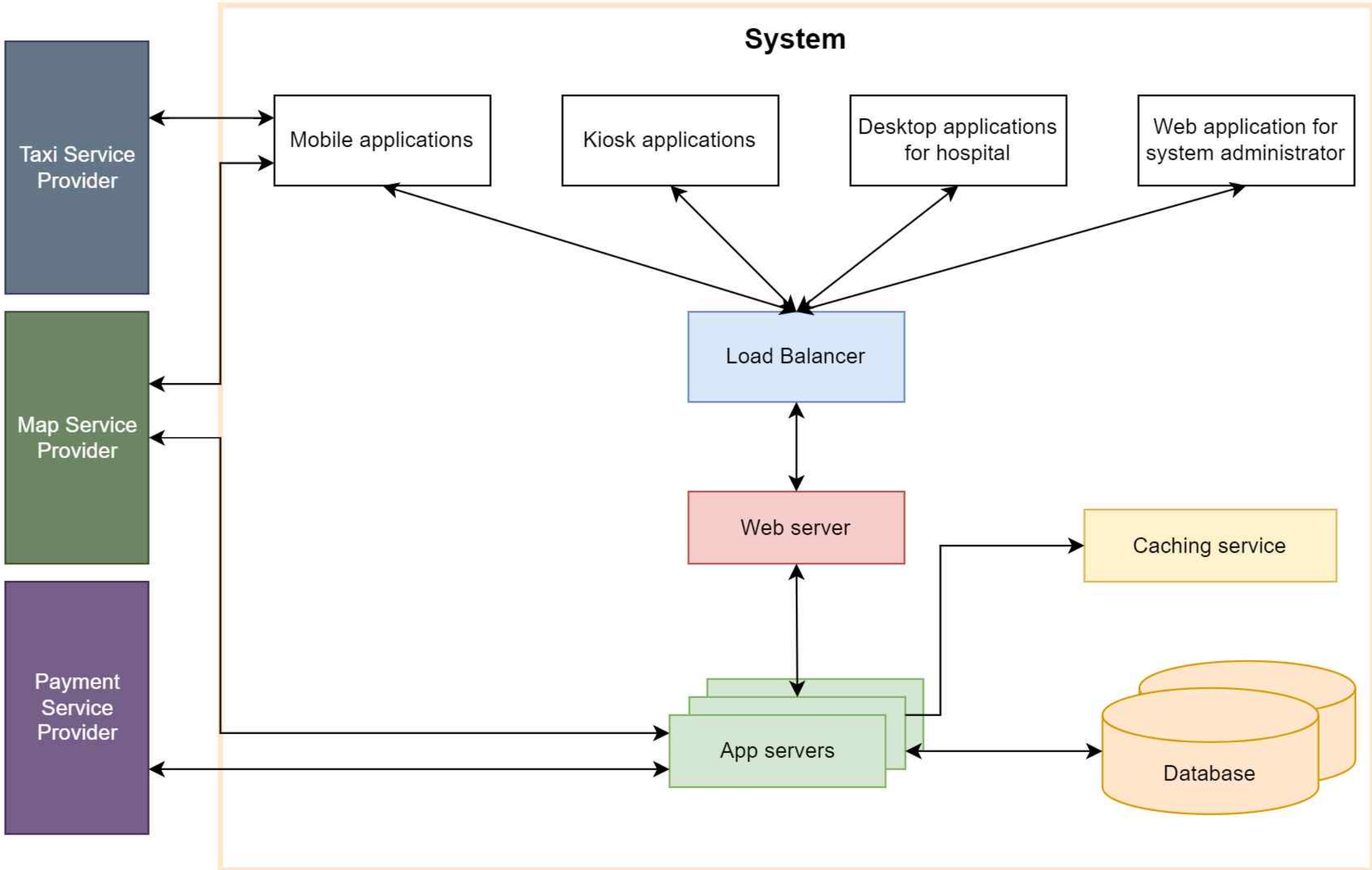


System Context Diagram

Domain Model



Domain Model : An Example



2.6 Assumptions

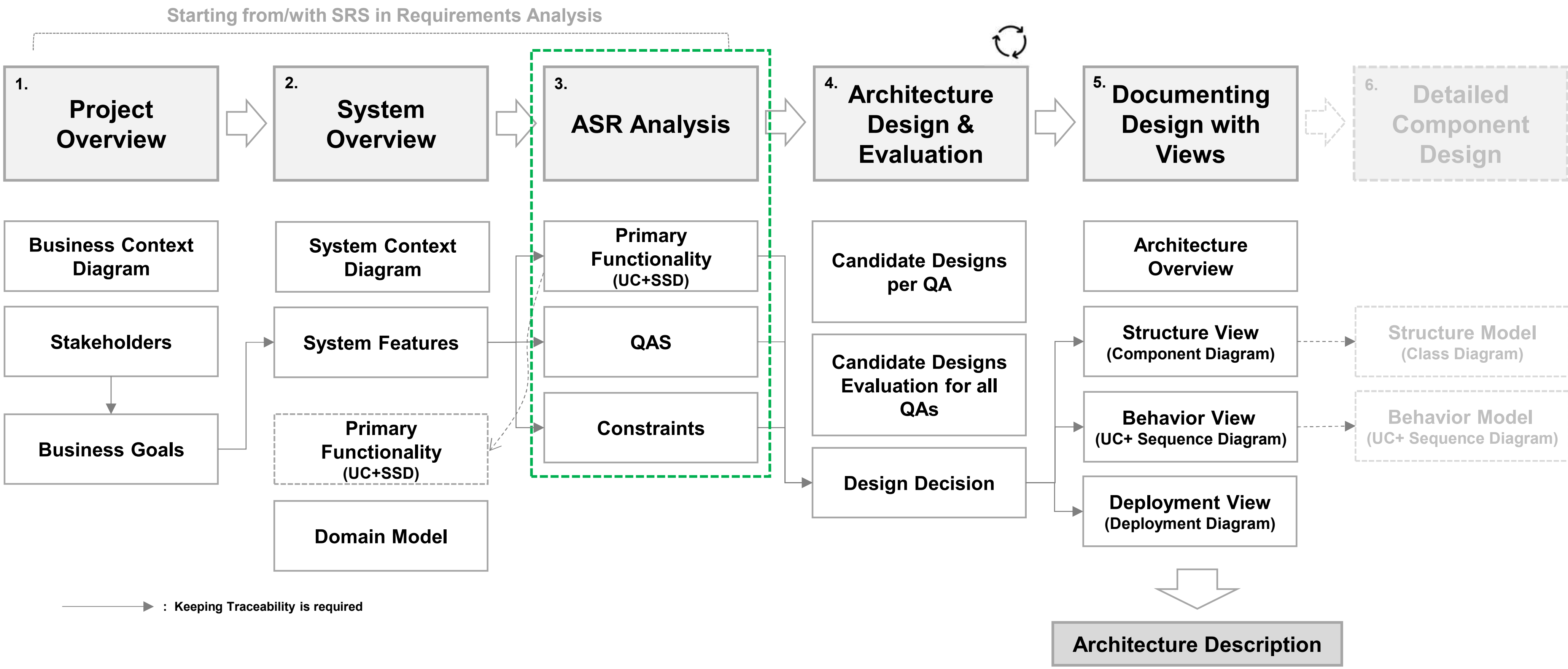
- **Several assumptions** that are not specified in the AD but are necessary for the system architecture design
  - Assuming all functionalities identified in the CEP Guide should be implemented.

  - Example :

1)	헬퍼가 프로필을 등록할 때 성별 외에도 수행 가능한 전문 분야가 있는 경우, 헬퍼 프로필에 기재하도록 하였다. (카테고리 기준, 배달/장보기/설치/운반/청소/돌봄 등)
2)	심부름 수행 도중, 고객이나 헬퍼의 스마트폰이 영원한 파괴/방전/통신불가 상태에 진입하지 않는다고 가정한다. 즉, 심부름 수행 중에 시스템과 스마트폰 사이의 복구 불가능한 통신 두절 상태는 발생하지 않는다.
3)	외부 시스템인 경찰청 시스템, 은행 시스템 모두 영원한 파괴/방전/통신불가 상태에 진입하지 않는다고 가정한다.
4)	사용자가 채팅으로 전달할 수 있는 메시지 길이는 한글 기준 1,000자(최대 3Kbyte) 이하라고 가정한다.
5)	헬퍼가 심부름 진행 현황 보고 용으로 업로드 할 수 있는 미디어 파일의 제한은, 이미지파일 1개당 최대 1MB, 최대 10장 / 영상 파일 최대 10MB 최대 2건으로 제한한다.

3. ASR Analysis

3. ASR Analysis



Where We are Now in AD

1. Project Overview
   - 1.1 Project Background
   - 1.2 Business Context Diagram
   - 1.3 Stakeholders
   - 1.4 Business Goals
2. System Overview
   - 2.1 System Context Diagram
   - 2.2 External Entity
   - 2.3 External Interface
   - 2.4 System Features
   - 2.5 Domain Model
   - 2.6 Assumptions
3. Architectural Drivers
   - 3.1 Primary Functionality
   - 3.2 Quality Attribute Scenario
   - 3.3 Constraints
4. Architecture Design & Evaluation
   - 4.1 Candidate Designs per QA
   - 4.2 Candidate Designs Evaluation for all QAs
   - 4.3 Design Decision
5. Architecture Design Description
   - 5.1 Architecture Overview
   - 5.2 Structure View
   - 5.3 Behavior View
   - 5.4 Deployment View
6. Component Design Description
   - 6.1.2 Component Structure Model
   - 6.1.5 Component Behavior Model
7. Architecture Traceability Summary

Architectural Driver

- **Architectural Drivers (AD)** are the key requirements that are most likely to affect the fundamental structure of the implementation.
    - AD will determine the structure (architecture) of the system.
    - Uncovering the ADs as early as possible is critical to the stable architecture design.
    - ADs are a part of requirements, called by **ASR (Architecturally Significant Requirements)**.

- **5 Architectural Drivers**
    - Primary Functionality
    - Quality Attribute
    - Design Purpose
    - Architectural Concerns
    - Constraints

- **Our ADs** consist of
    - **Primary Functionality**
    - **Quality Attribute Scenario (QAS)**
    - **Constraints**

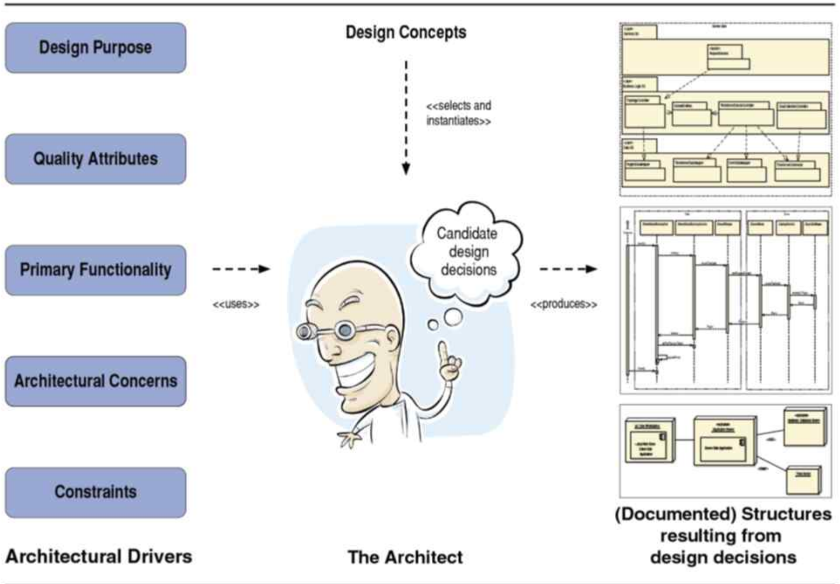


FIGURE 2.1 Overview of the architecture design activity
(Architect image © Brett Lamb I Dreamstime.com)

ASR to Architectural Drivers

- **<u>Mapping</u>** ASR(Architecturally Significant Requirements) in SRS to AD(Architectural Drivers) in AD(Architecture Description)
  - **Functionality → Primary Functionality (Use Case + SSD(System Sequence Diagram))**
  - **Quality Attribute → QAS (Quality Attribute Scenario)**
  - Constraints → Constraints *

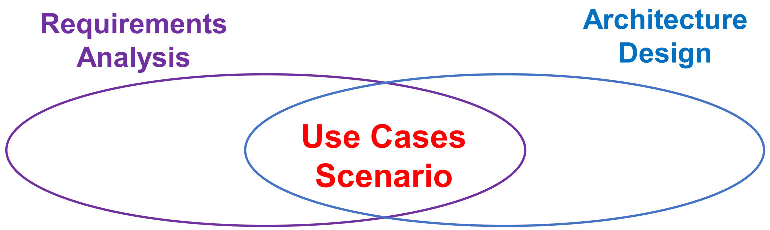
- **Scenario-based requirements analysis** in RE is quite helpful for architecture design.
  - Functional requirements → **(Select & Refine)** → Use Case + SSD
  - Quality attribute requirements → **(QAW)** → QAS

*** It's important, but out of the scope of the CEP for now.**

Use Cases as Scenarios-based Analysis

- **Scenario ≈ Use Case ≈ User Story**

- We can use scenarios in many ways within the architecture definition process.
  - Providing input to architecture definition
  - Evaluating the architecture: Scenarios are a primary input into almost any process of architectural evaluation.
  - Communicating with stakeholders: discussion of a scenario and how the system can meet the situation described is a very useful vehicle for communicating with all types of stakeholders.
  - Finding missing requirements: Another benefit of creating scenarios is that they often reveal what is missing as well as the suitability of what already exists.
  - Driving the testing process: Scenarios help highlight the things that are important to your stakeholders, thus providing a tremendously useful guide for where to focus testing activity.



3.1 Primary Functionality

- **Functionality** is the ability of the system to do the work for which it was intended.
  - Software architecture does not normally influence functionality.
  - Functionality can often be satisfied with any software architecture.

- **Primary functionality** is the functionality that is critical to achieve the business goal.
  - Implying a high level of technical difficulty or requiring the interaction of many architectural elements
  - Approximately 10 percent of use cases (user stories) in SRS are likely to be primary.

- **Functionality** → (Select & Refine) → **Primary Functionality (Use Case + SSD(System Sequence Diagram))**

- Why we need to consider primary functionality when designing an architecture?
  1. May need to plan work assignments
  2. Some quality attributes are directly connected to the primary functionality in the system.

Use Cases

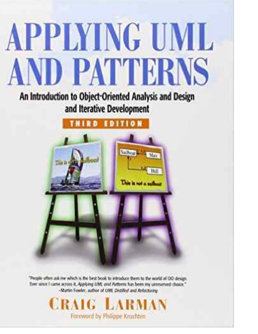
- **Use cases** are **text stories** of some actors using a system to meet goals.
  - A mechanism to capture (analyze) requirements

  - An example (Brief format):
    - **Process Sale**: *"A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items."*

  - Use case is not a diagram, but a text.
    - **Brief**
    - **Causal**
    - **Fully-Dressed**

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Use Case : An Example



Use Case UC1: Process Sale

Scope: NextGen POS application
Level: user goal
Primary Actor: Cashier
Stakeholders and Interests:
– Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
– Salesperson: Wants sales commissions updated.
– Customer: Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.
– Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
– Manager: Wants to be able to quickly perform override operations, and easily debug Cashier problems.
– Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
– Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.
Preconditions: Cashier is identified and authenticated.
Success Guarantee (or Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario (or Basic Flow):

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

*a. At any time, Manager requests an override operation:
   1. System enters Manager-authorized mode.
   2. Manager or Cashier performs one Manager-mode operation. e.g., cash balance change, resume a suspended sale on another register, void a sale, etc.
   3. System reverts to Cashier-authorized mode.

*b. At any time, System fails:
   To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.
   1. Cashier restarts System, logs in, and requests recovery of prior state.
   2. System reconstructs prior state.
      2a. System detects anomalies preventing recovery:
         1. System signals error to the Cashier, records the error, and enters a clean state.
         2. Cashier starts a new sale.

1a. Customer or Manager indicate to resume a suspended sale.
   1. Cashier performs resume operation, and enters the ID to retrieve the sale.
   2. System displays the state of the resumed sale, with subtotal.
      2a. Sale not found.
         1. System signals error to the Cashier.
         2. Cashier probably starts new sale and re-enters all items.
   3. Cashier continues with sale (probably entering more items or handling payment).

2-4a. Customer tells Cashier they have a tax-exempt status (e.g., seniors, native peoples)
   1. Cashier verifies, and then enters tax-exempt status code.
   2. System records status (which it will use during tax calculations)

3a. Invalid item ID (not found in system):
   1. System signals error and rejects entry.
   2. Cashier responds to the error:
      2a. There is a human-readable item ID (e.g., a numeric UPC):
         1. Cashier manually enters the item ID.
         2. System displays description and price.
            2a. Invalid item ID: System signals error. Cashier tries alternate method.
      2b. There is no item ID, but there is a price on the tag:
         1. Cashier asks Manager to perform an override operation.
         2. Managers performs override.
         3. Cashier indicates manual price entry, enters price, and requests standard taxation for this amount (because there is no product information, the tax engine can't otherwise deduce how to tax it)
      2c. Cashier performs <u>Find Product Help</u> to obtain true item ID and price.
      2d. Otherwise, Cashier asks an employee for the true item ID or price, and does either manual ID or manual price entry (see above).

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):
   1. Cashier can enter item category identifier and the quantity.

3c. Item requires manual category and price entry (such as flowers or cards with a price on them):
   1. Cashier enters special manual category code, plus the price.

3-6a: Customer asks Cashier to remove (i.e., void) an item from the purchase:
   This is only legal if the item value is less than the void limit for Cashiers, otherwise a Manager override is needed.
   1. Cashier enters item identifier for removal from sale.
   2. System removes item and displays updated running total.
      2a. Item price exceeds void limit for Cashiers:
         1. System signals error, and suggests Manager override.
         2. Cashier requests Manager override, gets it, and repeats operation.

3-6b. Customer tells Cashier to cancel sale:
   1. Cashier cancels sale on System.

3-6c. Cashier suspends the sale:
   1. System records sale so that it is available for retrieval on any POS register.
   2. System presents a "suspend receipt" that includes the line items, and a sale ID used to retrieve and resume the sale.

4a. The system supplied item price is not wanted (e.g., Customer complained about something and is offered a lower price):
   1. Cashier requests approval from Manager.
   2. Manager performs override operation.
   3. Cashier enters manual override price.
   4. System presents new price.

5a. System detects failure to communicate with external tax calculation system service:
   1. System restarts the service on the POS node, and continues.
      1a. System detects that the service does not restart.
         1. System signals error.
         2. Cashier may manually calculate and enter the tax, or cancel the sale.

5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):
   1. Cashier signals discount request.
   2. Cashier enters Customer identification.
   3. System presents discount total, based on discount rules.

5c. Customer says they have credit in their account, to apply to the sale:
   1. Cashier signals credit request.
   2. Cashier enters Customer identification.
   3. Systems applies credit up to price=0, and reduces remaining credit.

6a. Customer says they intended to pay by cash but don't have enough cash:
   1. Cashier asks for alternate payment method.
      1a. Customer tells Cashier to cancel sale. Cashier cancels sale on System.

7a. Paying by cash:
   1. Cashier enters the cash amount tendered.
   2. System presents the balance due, and releases the cash drawer.
   3. Cashier deposits cash tendered and returns balance in cash to Customer.
   4. System records the cash payment.

7b. Paying by credit:
   1. Customer enters their credit account information.
   2. System displays their payment for verification.
   3. Cashier confirms.
      3a. Cashier cancels payment step:
         1. System reverts to "item entry" mode.
   4. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
      4a. System detects failure to collaborate with external system:
         1. System signals error to Cashier.
         2. Cashier asks Customer for alternate payment.
   5. System receives payment approval, signals approval to Cashier, and releases cash drawer (to insert signed credit payment receipt).
      5a. System receives payment denial:
         1. System signals denial to Cashier.
         2. Cashier asks Customer for alternate payment.
      5b. Timeout waiting for response.
         1. System signals timeout to Cashier.
         2. Cashier may try again, or ask Customer for alternate payment.
   6. System records the credit payment, which includes the payment approval.
   7. System presents credit payment signature input mechanism.
   8. Cashier asks Customer for a credit payment signature. Customer enters signature.
   9. If signature on paper receipt, Cashier places receipt in cash drawer and closes it.

7c. Paying by check…

7d. Paying by debit…

7e. Cashier cancels payment step:
   1. System reverts to "item entry" mode.

7f. Customer presents coupons:
   1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons.
      1a. Coupon entered is not for any purchased item:
         1. System signals error to Cashier.

9a. There are product rebates:
   1. System presents the rebate forms and rebate receipts for each item with a rebate.

9b. Customer requests gift receipt (no prices visible):
   1. Cashier requests gift receipt and System presents it.

9c. Printer out of paper.
   1. If System can detect the fault, will signal the problem.
   2. Cashier replaces paper.
   3. Cashier requests another receipt.

Our Use Case Format

- **The casual format use cases**
  - The system is considered as a black box.
  - No design/implementation details are considered.

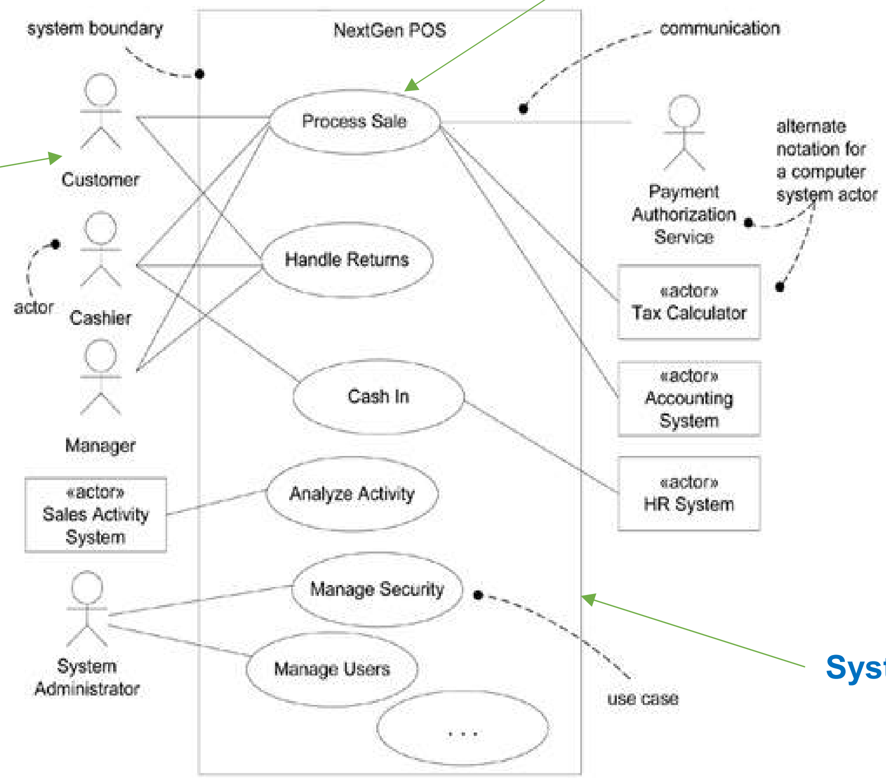
- Example : LMS (Library Management System)

Use Case	1. Make Reservation
Actor	Librarian
Description	A librarian requests LMS to make a reservation for a book.
Stakeholders	User, Librarian, System Manager
Preconditions	Borrower should have an id_card.
Main Scenario	(A) : Actor, (S) : System 1. (A) A librarian requests the reservation of a title 2. (S) Check if a corresponding title exists 3. (S) Check if a corresponding borrower exists 4. (S) Create a reservation information
Alternative Scenario	[Out of date information] 3: (S) If the borrower's information is out of date, request for the update. (A) A librarian updates up-to-date information of the borrower. [Invalid Input] 1~3: If invalid reservation information is entered, indicate an error.

Use Case Diagram

- **Use case diagram** illustrates the name of use cases and actors and the relationships between them.
  - System context diagram
  - A summary of all use cases

Use case



Actor

Something with behavior, such as a person, computer system, or organization

- Primary Actor : having user goals fulfilled through using services of the SuD (System Under Discussion) , *e.g.,* cashier

- Supporting Actor : providing a service to the SuD, *e.g.,* payment authorization service

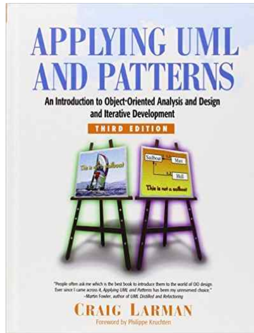
- Offstage Actor : having an interest in the behavior of the use case, but is not primary or supporting, *e.g.,* tax agency

System boundary

General Guidelines for Modeling Use Cases

- Use cases are written in narrative language that can be understood by all stakeholders.
- Use cases are not models for functional decomposition.
- Use case models describe what is needed in a system in terms of functional responses to given stimuli.
- A use case is initiated by an actor, and then goes on to describe a sequence of interactions between actors and the system that, when taken together, model systemic functional requirements.
- Use cases may also include variants of the normal operation that describe error occurrences, detection, handling and recovery, failure modes, or other alternative behaviors.
- Focus on interactions which involve quality attributes such as performance, modifiability and security.
- Include each interaction with all the actors associated with the use case.
  - Each step should be written in active voice with the subject of the system or an associated actor.
  - Each step should describe the behavior of the system or an associated actor, but not both.
  - Each step should describe the interaction clearly.
- Use terms that can be understood by stakeholders. Don’t use technical terms that can be only understood by developers.

System Sequence Diagram

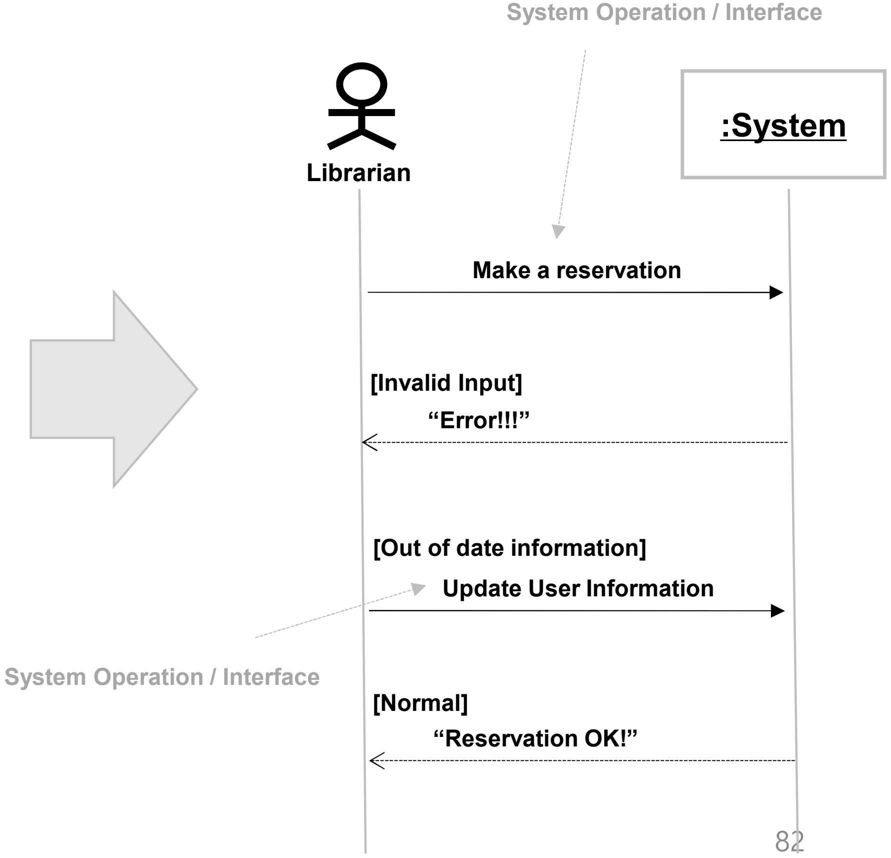


- **Use cases** describe how external actors interact with the software system.
  - During this interaction, an **actor** generates **system events** to a system, usually requesting some **system operation** to handle the event.
- **System sequence diagram (SSD)**
  - A picture that shows the events that external actors generate, their order, and inter-system events, for one particular scenario of a use case.
  - In the **sequence diagram** notation, there are
    - the external actors that interact directly with the system,
    - the system (as a black box), and
    - the system events that the actors generate.
  - Depict system behavior in terms of **what the system does**, not how it does it
  - Used as an input to system design → **System operations / System interfaces**

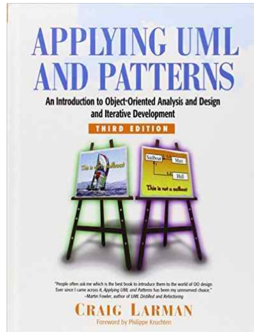
System Sequence Diagram

- One SSD for each use case
  - **The identified system operations/interfaces** will be linked to **behavior views(5.3)**.
  - Keeping traceability is important.

Use Case	1. Make Reservation
Actor	Librarian
Description	A librarian requests LMS to make a reservation for a book.
Stakeholders	User, Librarian, System Manager
Preconditions	Borrower should have an id_card.
Main Scenario	(A) : Actor, (S) : System 1. (A) A librarian requests the reservation of a title 2. (S) Check if a corresponding title exists 3. (S) Check if a corresponding borrower exists 4. (S) Create a reservation information
Alternative Scenario	[Out of date information] 3: (S) If the borrower's information is out of date, request for the update. (A) A librarian updates up-to-date information of the borrower. [Invalid Input] 1~3: If invalid reservation information is entered, indicate an error.

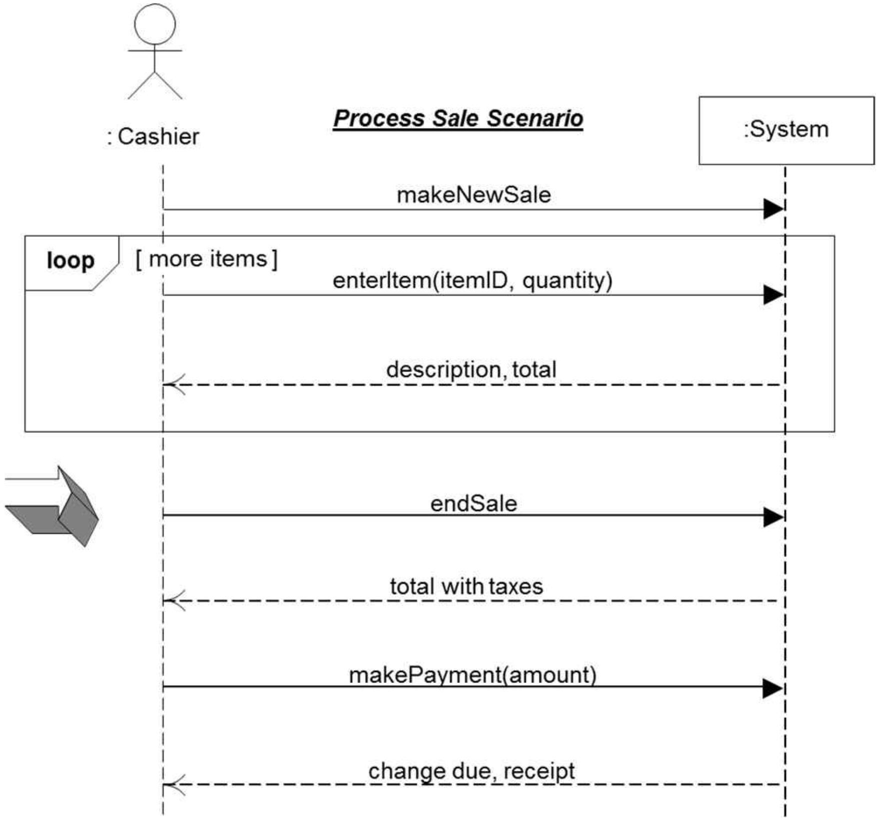


System Sequence Diagram : An Example in OOAD



Simple cash-only *Process Sale* scenario:

1.Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4.System records sale line item and presents item description, price, and running total.
Cashier repeats steps 3-4 until indicates done.
5.System presents total with taxes calculated.
6.Cashier tells Customer the total, and asks for payment.
7.Customer pays and System handles payment.
...



Use Case

System Sequence Diagram

3.2 Quality Attribute Scenario

- **Quality Attribute (Requirement)** is a measurable or testable properties of a system, that used to indicate how well the system satisfies the needs of its **stakeholders**.

- Among ADs, quality attributes are the ones that shape the architecture the most significantly, because,
  - Functionality does not determine architecture.
    - Numerous architectures to satisfy that functionality
    - You could divide up the functionality in any number of ways and assign them to different architectural elements.
    - If functionality were the only thing that mattered, you wouldn't have to divide the system into pieces at all.
  - Instead, we design our systems as structured sets of cooperating architectural elements (layers, components, classes, databases, apps, threads, peers, tiers, and so on) to support a variety of other purposes (i.e., **quality attributes**).

  - Systems are frequently redesigned not because they are functionally deficient, but because
    - They are difficult to maintain, port, or scale. → Maintainability, Portability, Scalability, Modifiability, Extensibility
    - They are too slow. → Performance, Efficiency
    - They have been compromised by hackers. → Security, Confidentiality

Quality Attribute Requirements : Examples

- 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.

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

System Features	QA	Quality Attribute Requirements
Support international languages	*Modifiability*	*A developer should be able to package a version of the system with new language support in 80 person-hours.*
Comply with regulations that have an impact on life-critical systems such as fire alarms	*Performance*	*A life-critical alarm should be reported to the concerned users within 3s of the occurrence of the event that generated the alarm.*
Support hardware devices from different Manufacturers	*Modifiability*	*A field engineer is able to integrate a new field device into the system at runtime with no downtime or side effects.*
Support conversions of nonstandard units used by the different hardware devices	*Modifiability*	*A system administrator configures the system at runtime to handle the units from a newly plugged in field device with no downtime or side effects.*

Describing Quality Attribute Requirements

- Architects require <u>more detailed</u> and <u>unambiguous</u> descriptions of quality attribute requirements.
  - But it is not easy since requirements are driven and written in **natural languages**.

- For examples,
  - *"A system shall be modifiable."*
    → <u>Ambiguous</u>, because every system is modifiable or not with respect to some changes.
  - *"A system shall have high performance."*
    → <u>Ambiguous</u>, because what kind of performance does this refer to? Response time, throughput, or others?

- How to express the qualities unambiguously?
  - Solution is **QAS(Quality Attribute Scenarios)** through **QAW(Quality Attribute Workshop)**, **Utility Tree**, or **Quality Attribute Tree**.

Quality Attributes Scenarios (QAS)

- **QAS (Quality Attribute Scenario)** is a short description of how a system is required to respond to some stimulus.
  - Describing the system's <u>response</u> to some <u>stimulus</u>
  - Specifying the <u>response measure</u> you would like to achieve in response to a specific stimulus

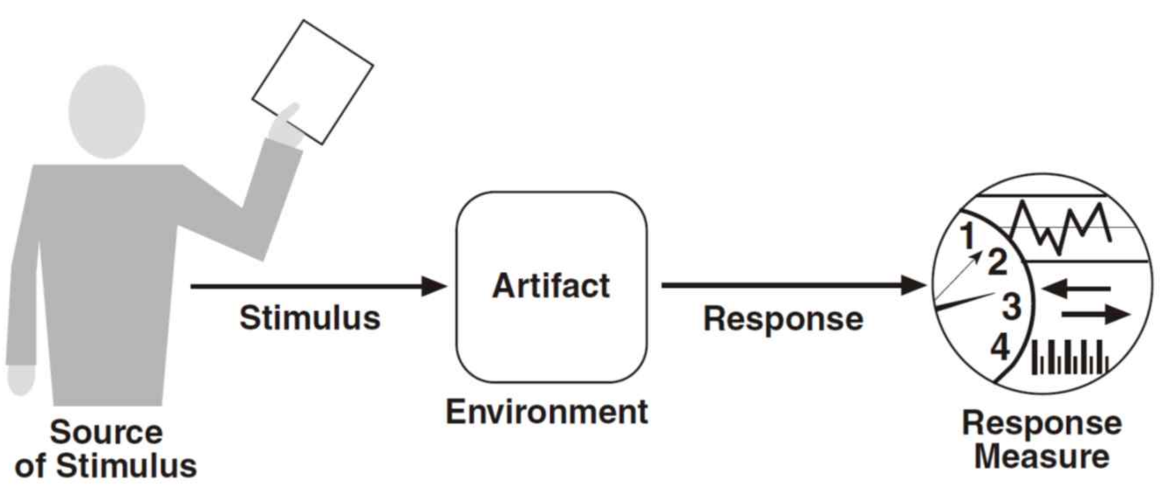
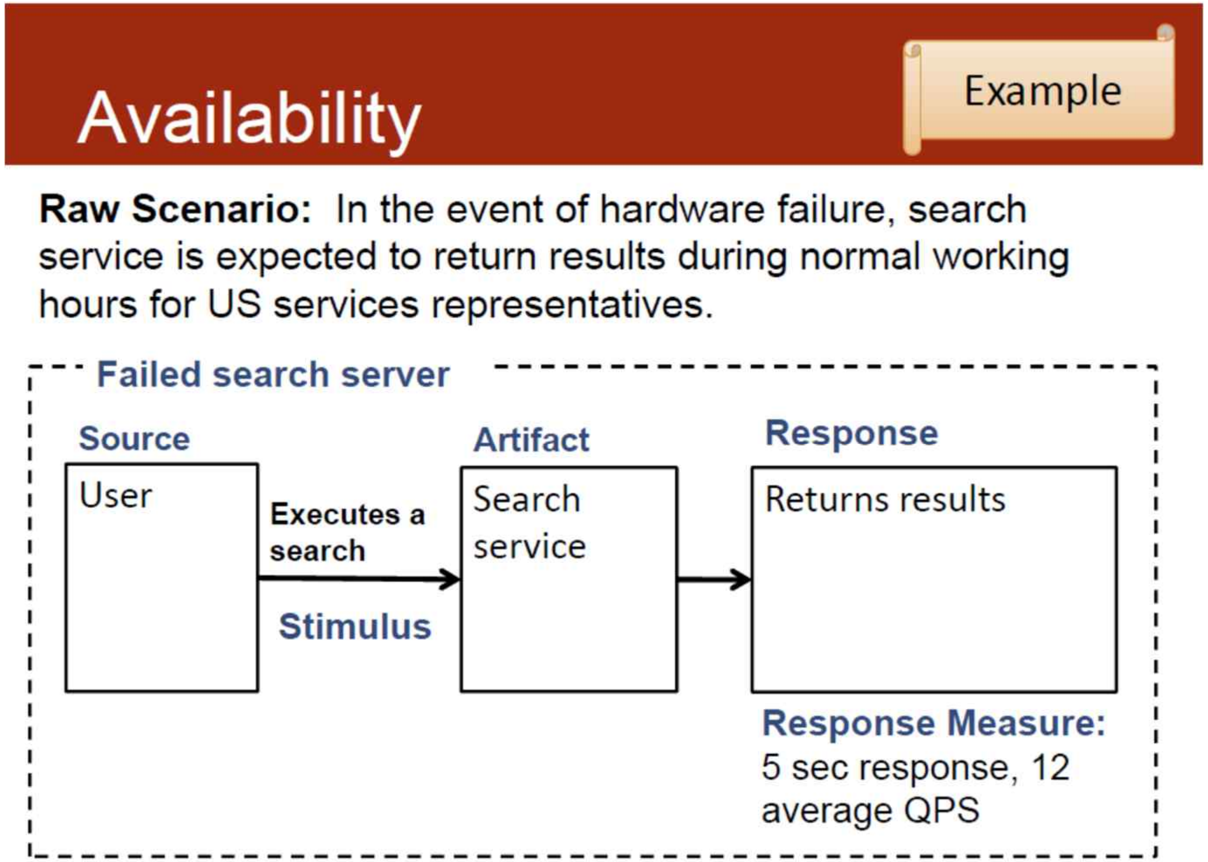


FIGURE 2.2 The six parts of a quality attribute scenario

The QAS Template

Requirement-ID	***QA_###***
Category	관련된 ***Quality Attribute***가 무엇인지 기술함 **(** 예**:** ***Performance, Reliability, Security*** 등**)**
Source	***Stimulus***를 발생시키는 주체가 무엇인지 기술함
Stimulus	시스템에 입력되는 내외부 자극이 무엇인지 기술함
Artifacts	***Stimulus***의 영향을 받는 시스템의 내부 모듈**,** 컴포넌트**,** 혹은 시스템 전체
Environment	해당 ***Stimulus*** 발생시 시스템의 환경**(**운영모드일 수도 있고**,** 그 외 다양한 상황 가능**)**
Response	기술된 ***Environment***에서 ***Artifact***이 ***Stimulus***를 받아들인 후 취하는 ***Action***이 무엇인지 설명함
Response Measure	위의 ***Response***의 정도를 측정하는 단위가 무엇인지 기술함 **(**예**:** 초당 데이터 처리량**,** 반응시간**,** 시간일 경우 ***hour, minute, second*** 여부 등**)**
Priority	***Quality Attribute Tree*** 상에서의 우선순위
Description	위에 기술된 ***Source***부터 ***Response Measure***까지의 내용을 하나의 문장으로 요약해서 기술함

The QAS Example : Availability

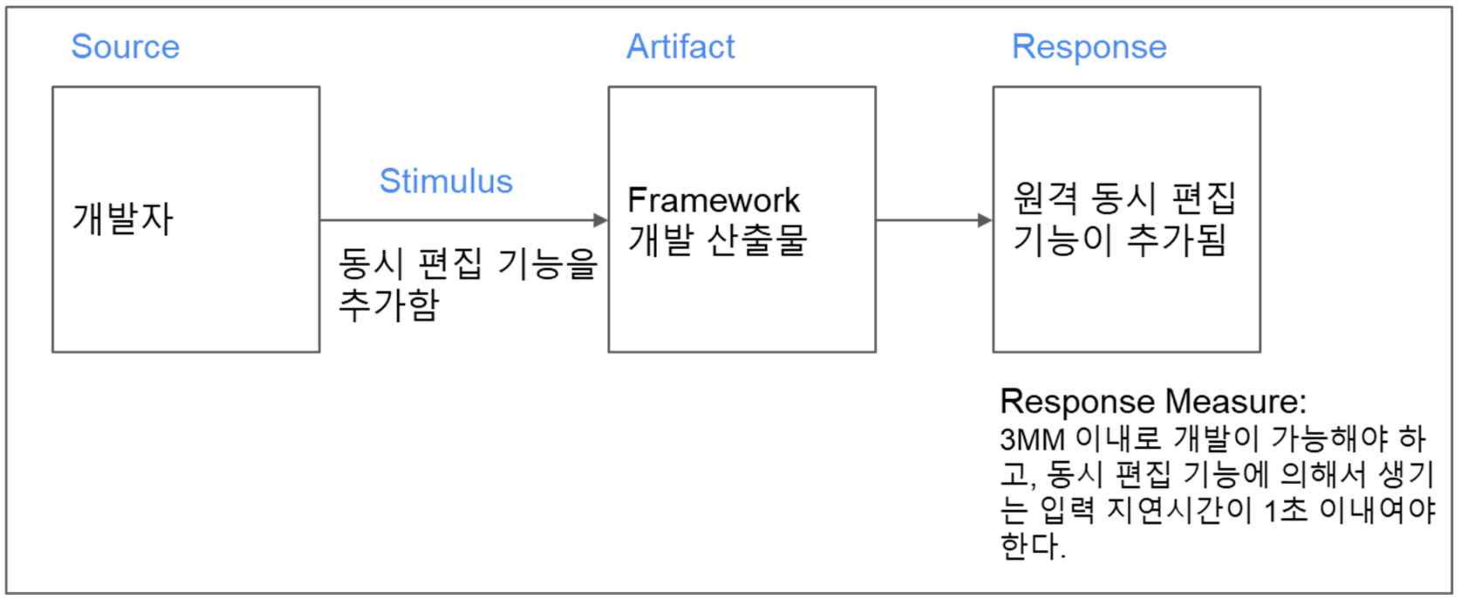


Refined Scenario: In the event of hardware failure, search service is expected to return results within 5 sec, in 12 average QPA (Queries Per Sec)

The QAS Example : Modifiability

Raw scenario: Framework에 원격 동시 편집 기능을 추가하려고 할때 쉽게 추가할 수 있어야 한다 (Modifiability)

Env: Framework가 로컬 편집 기능만 지원



Refined Scenario: Framework에 동시 편집 기능을 추가하려고 할때, 3MM 이내로 개발이 가능해야 하고, 동시 편집 기능에 의해서 생기는 입력 지연시간이 1초 이내여야 한다

The QAS Example : Robustness

Scenario Refinement: Robustness

Raw Scenario : 로봇이 물건을 잘못쌓아도 다시 시작시키면 다음번은 정상적으로 쌓는다.

Source
노동자

Stimulus
정지&리셋 버튼

Artifact
로봇

Environment:
박스가 잘못 쌓임

Response
박스를 다시 쌓는다

Response Measure:
정상보정 시도 회수 < 3회

Refined Scenario :
로봇이 물건을 잘못쌓은 경우, 노동자가 로봇을 정지하고 reset 버튼을 누르면 2회 이내에 박스를 원래 위치에 두고 다시 정상적으로 쌓는다.

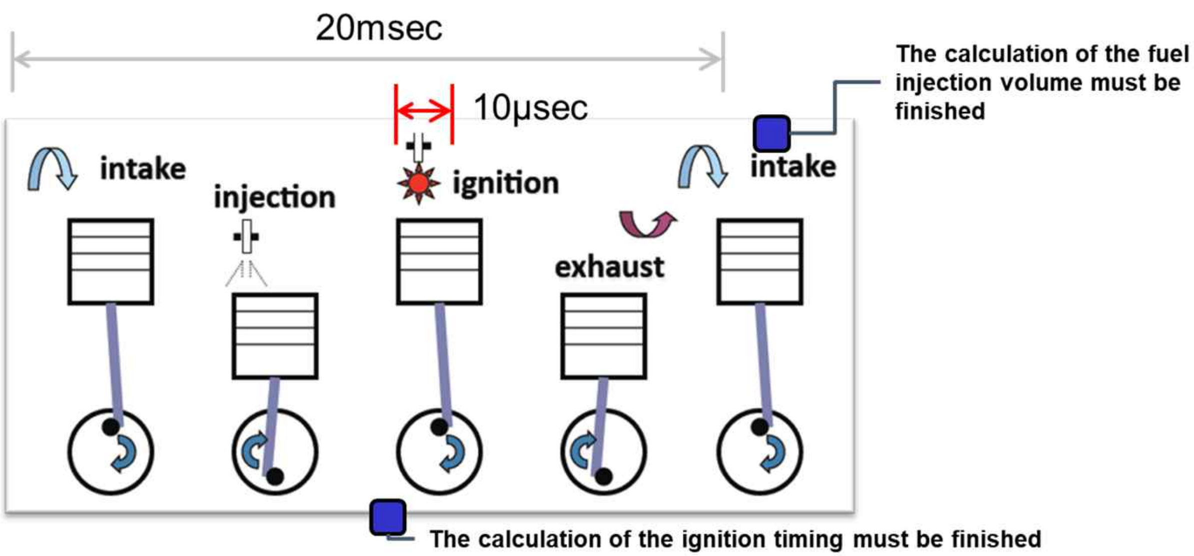
Quality Attribute Tree : Examples

No.	Category	Response Measure	품질속성 요구사항
1	Performance	시스템 동작 Hz 수	1 Core System에서 4 Core System을 지원하는 system으로 변경될 경우, 정상적인 Operation을 수행하는 데 있어서 소모되는 전력이 1 Core System의 경우와 비교하여 35%의 Hz수를 출력하여야 한다.
2		Communication 횟수, Communication 데이터	정상적인 Operation을 수행하는 데 있어서 Core 간 Communication 으로 인해 발생되는 Overhead 증가분은 10% 이내이어야 한다.
3		사용된 Memory 용량	정상적인 Operation을 수행하는 데 있어서 memory 사용량 증가 정도는 70 % 이내이어야 한다.
4		수행 중 각 Core들의 Idle Time	정상적인 Operation을 수행하는 데 있어서 전체 system의 운영 시간 중 각 Core들이 Idle 상태에 머무르는 시간은 15% 이내 이어야 한다.
5		Data Frames/Second	정상적인 Operation하에서 Video Data Decoding 성능은 기존 1 Core System의 3배 정도인 2.5 Data Frames/Second를 만족시켜야 한다.
6		화면의 가로 세로 픽셀 수	정상적인 Operation하에서 Video Decoder가 지원 가능한 출력 화면의 크기는 기존 1 Core System의 경우와 마찬가지로 1280X720 픽셀까지이다 .
7		Bit rate	정상적인 Operation하에서 Video Decoder가 출력하는 화면의 화질을 측정하는 Bit rate는 기존 1 Core System의 경우와 마찬가지로 20Mbps를 지원 가능하여야 한다.
8	Modifiability	수정 컴포넌트 비율	4 Core System에서 향후 그 이상의 개수로 Core 숫자가 늘어날 경우, 4 Core System 기반의 Architecture 상에서 변경되는 Component와 Connector의 비율은 Parallel Node의 숫자에 해당되는 Instance 개수 증가를 제외하고 50% 미만이어야 한다.
9	Functionality	시스템 기능 가용률 :전체 기능 개수 대비 가용 기능 비율	1 Core System에서 4 Core System을 지원하는 system으로 변경될 경우, 정상적인 Operation하에서 Video Decoder가 제공하던 기능 중 가용한 기능은 100%를 만족시켜야 한다.
10		해당사항 없음	Video Decoder의 input data stream 중에 error가 있는 input frame이 입력될 경우, Video Decoder는 해당 frame의 decoding을 수행하지 않고 pass한 후 다음 frame을 읽어 들여야 한다.
11	Portability	시스템 기능 가용률 :전체 기능 개수 대비 가용 기능 비율	Cache memory size 가 32K인 Device에서 16K인 Device로 변경될 경우, 정상적인 Operation하에서 Video Decoder가 제공하던 기능 중 가용한 기능은 100%를 만족시켜야 한다.

Quality Measures Example : Performance

- **Performance requirements**
  - Defining the extent or how well, and under what conditions, a function or task is to be performed

- Example:
  - *"In case of 6,000 rpm and one cycle is 20 msec, timing precision of the ignition should be 10 µsec."*



Quality Measures Example : Availability

- **Availability requirements**
  - Defining the degree to which a system or component is operational and accessible when required for use [IEEE 610].

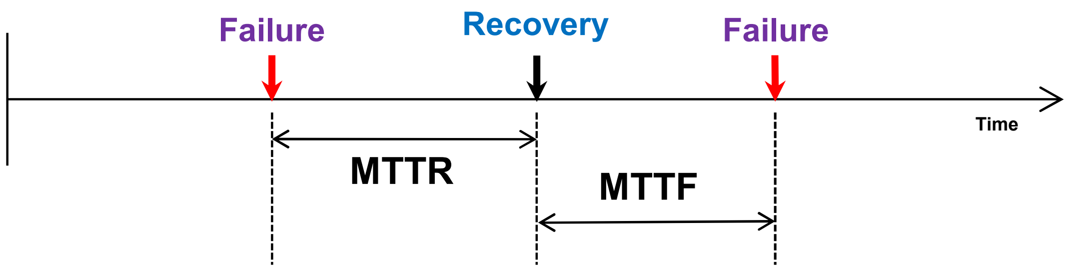
$$\textbf{Availability} = \frac{\textbf{MTTF}}{\textbf{MTTF + MTTR}}$$

MTTF : Mean Time To Failures

MTTR : Mean Time To Repair

MTBF : Mean Time Between Failures
(= MTTR + MTTF)

가용성	연간 장애 시간	주당 장애 시간
98%	7.3일	3시간 22분
99%	3.65일	1시간 41분
99.8%	17시간 30분	20분 10초
99.9%	8시간 45분	10분 5초
99.99%	52분 30초	1분
99.999%	5분25초	6초



QAS Example - Availability

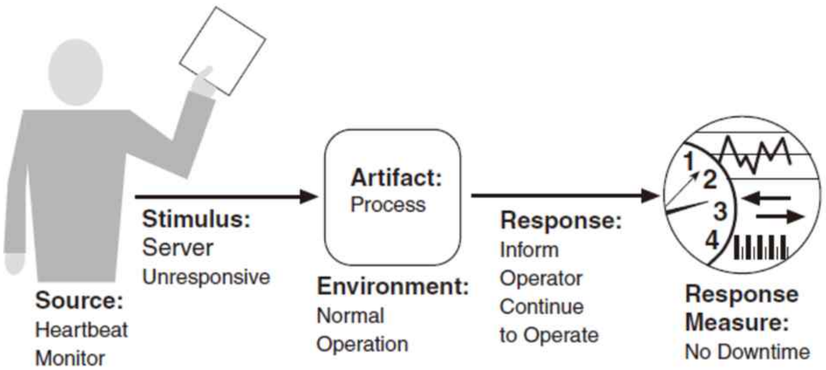


FIGURE 5.3 Sample concrete availability scenario

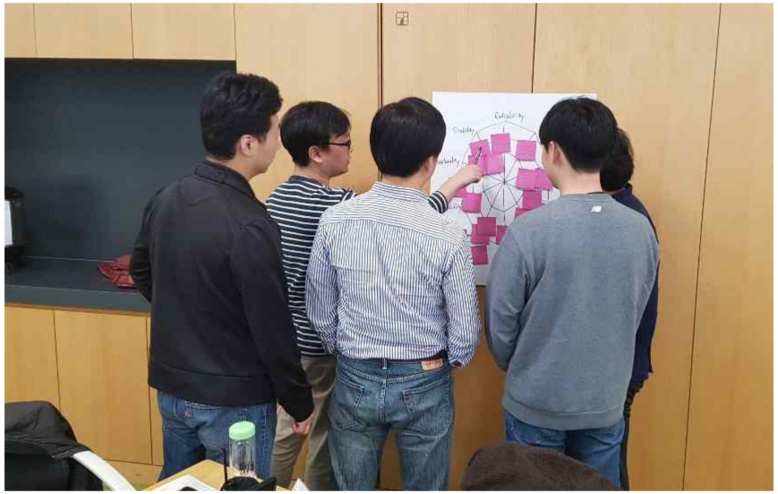
조별 QAS 발표 #1

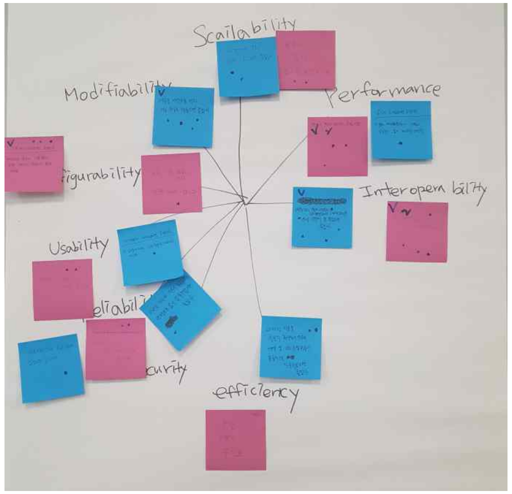
Portion of Scenario	Possible Values
Source	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	Processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	Prevent the fault from becoming a failure Detect the fault: ▪ Log the fault ▪ Notify appropriate entities (people or systems) Recover from the fault: ▪ Disable source of events causing the fault ▪ Be temporarily unavailable while repair is being effected ▪ Fix or mask the fault/failure or contain the damage it causes ▪ Operate in a degraded mode while repair is being effected
Response Measure	Time or time interval when the system must be available Availability percentage (e.g., 99.999%) Time to detect the fault Time to repair the fault Time or time interval in which system can be in degraded mode Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

Quality Attributes Workshop (QAW)

- A **facilitated brainstorming session**
  - A group of system stakeholders cover the bulk of the activities of **eliciting**, **specifying**, **prioritizing**, and **achieving consensus** on **quality attributes**.
  - Output: a set of QASs

- Scenarios should be prioritized (L/M/H).
  - With respect to the success of the system → by the customer
  - With respect to the technical risk associated with the scenario → by the architect





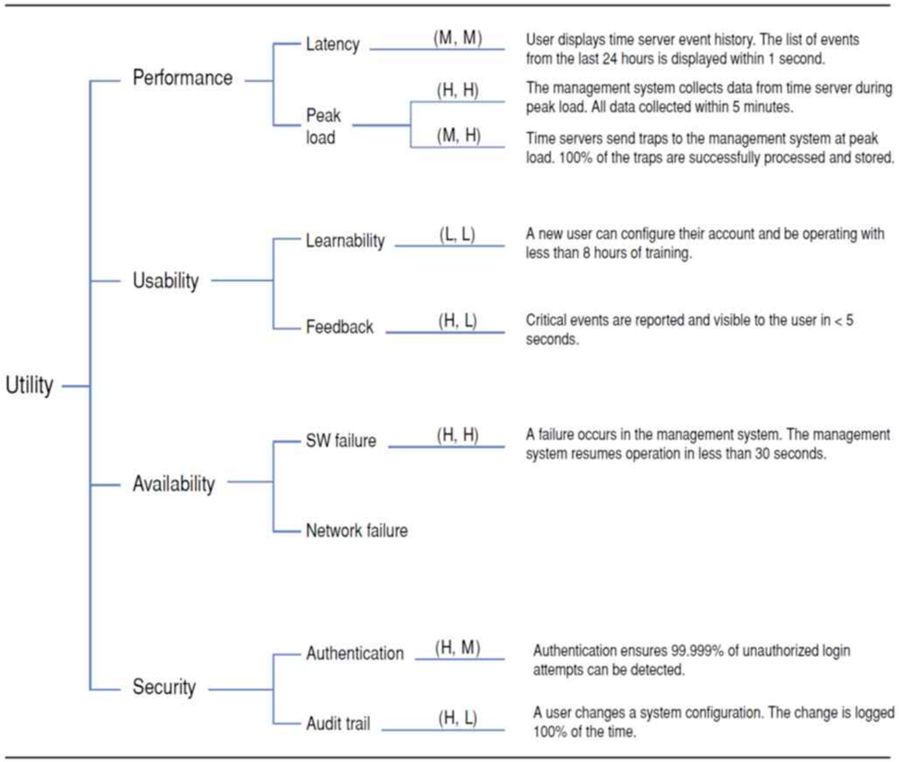
Quality Attributes Workshop (QAW)

- **QAW Steps**
  - QAW Presentation and Introductions
  - Business Goals Presentation
  - Architectural Plan Presentation
  - Identification of Architectural Drivers
  - Scenario Brainstorming
  - Scenario Consolidation
  - Scenario Prioritization
  - Scenario Refinement

- **Mini QAW**
  - Mini-QAW Introduction
  - Introduction to Quality Attributes, Quality Attributes Taxonomy
  - Scenario Brainstorming : “Walk the System Properties Web” activity
  - Raw Scenario Prioritization: Dot Voting
  - Scenario Refinement
  - Review Results with Stakeholders

Utility Tree

- One way to organize your thoughts
  - Useful when no stakeholders are readily available to consult
  - Helps to articulate your quality attribute goals in detail, and then to prioritize them



Software Quality Model : ISO/IEC 9126

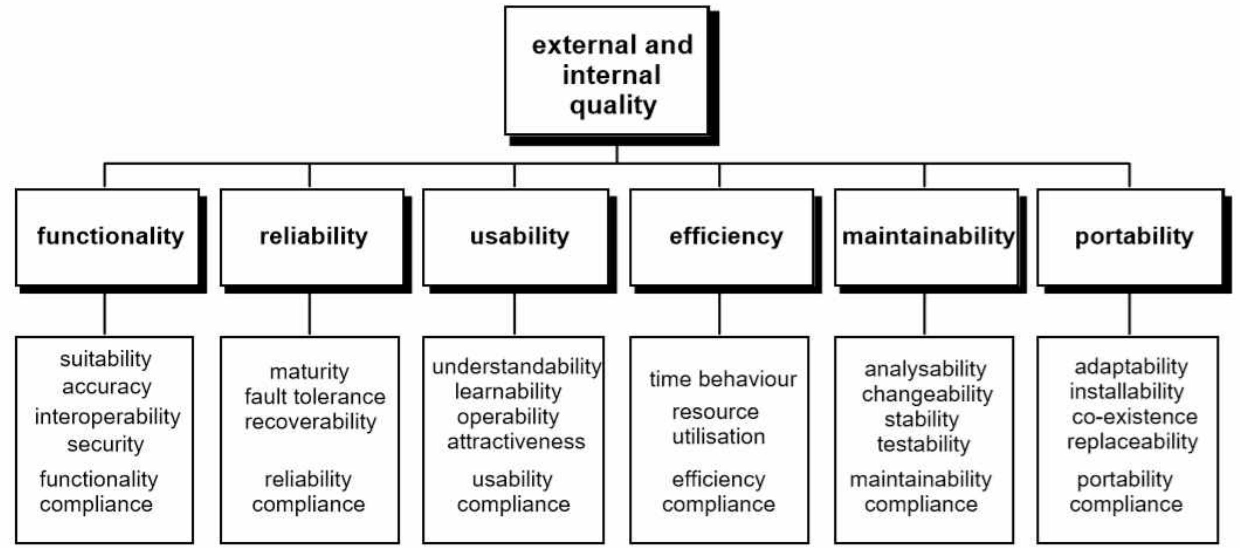


Figure 4 – Quality model for external and internal quality

FINAL DRAFT

INTERNATIONAL STANDARD

ISO/IEC FDIS 9126-1

ISO/IEC JTC 1

Secretariat: **ANSI**

Voting begins on: **2000-01-20**

Voting terminates on: **2000-03-20**

Information technology — Software product quality —

Part 1:
Quality model

Technologies de l'information — Qualité des produits logiciels — Partie 1: Modèle de qualité

Please see the administrative notes on page ii-1

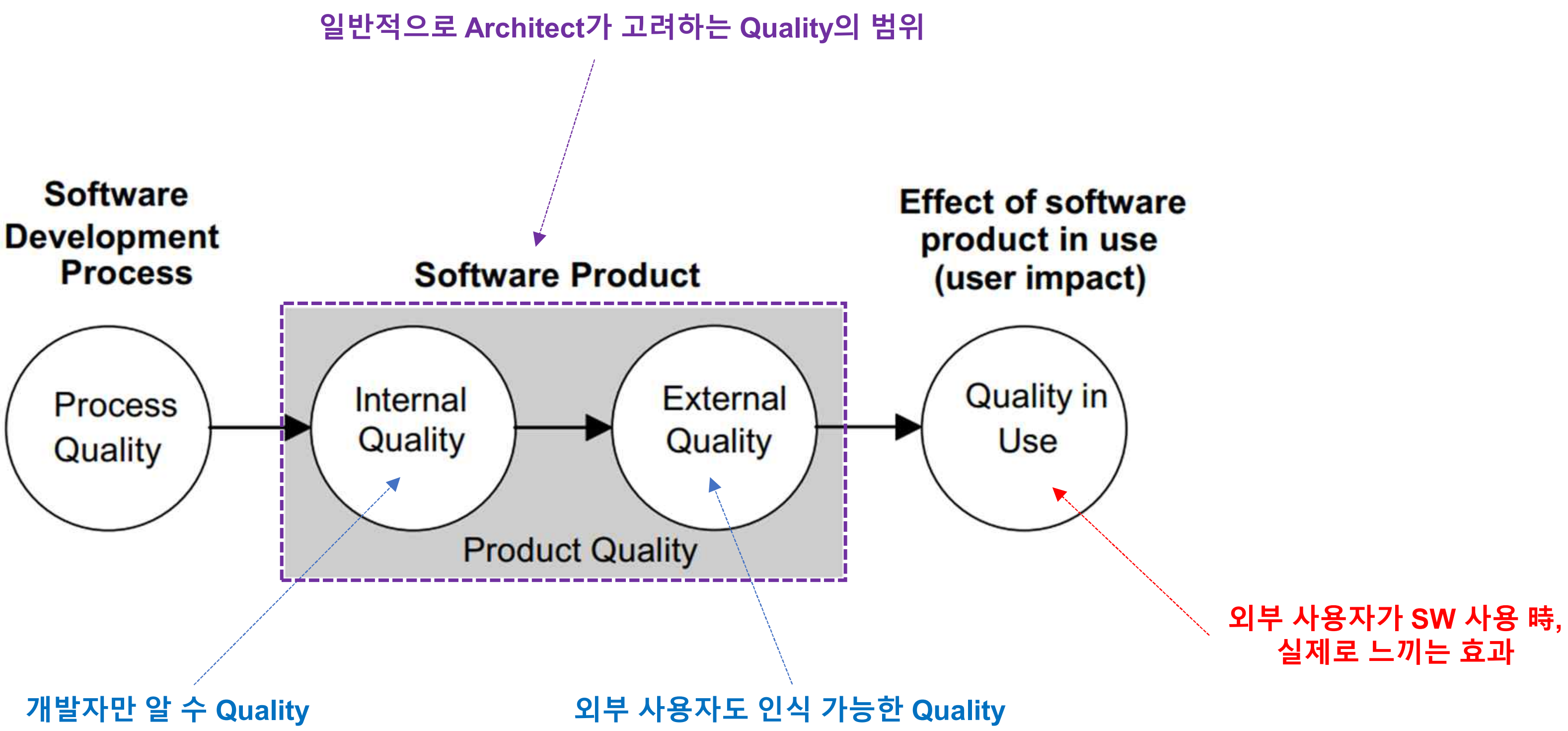
RECIPIENTS OF THIS DOCUMENT ARE INVITED TO SUBMIT, WITH THEIR COMMENTS, NOTIFICATION OF ANY RELEVANT PATENT RIGHTS OF WHICH THEY ARE AWARE AND TO PROVIDE SUPPORTING DOCUMENTATION.

IN ADDITION TO THEIR EVALUATION AS BEING ACCEPTABLE FOR INDUSTRIAL, TECHNOLOGICAL, COMMERCIAL AND USER PURPOSES, DRAFT INTERNATIONAL STANDARDS MAY ON OCCASION HAVE TO BE CONSIDERED IN THE LIGHT OF THEIR POTENTIAL TO BECOME STANDARDS TO WHICH REFERENCE MAY BE MADE IN NATIONAL REGULATIONS.

Reference number
ISO/IEC FDIS 9126-1:2000(E)

© ISO/IEC 2000

Conventional Quality Categories in ISO/IEC 9126



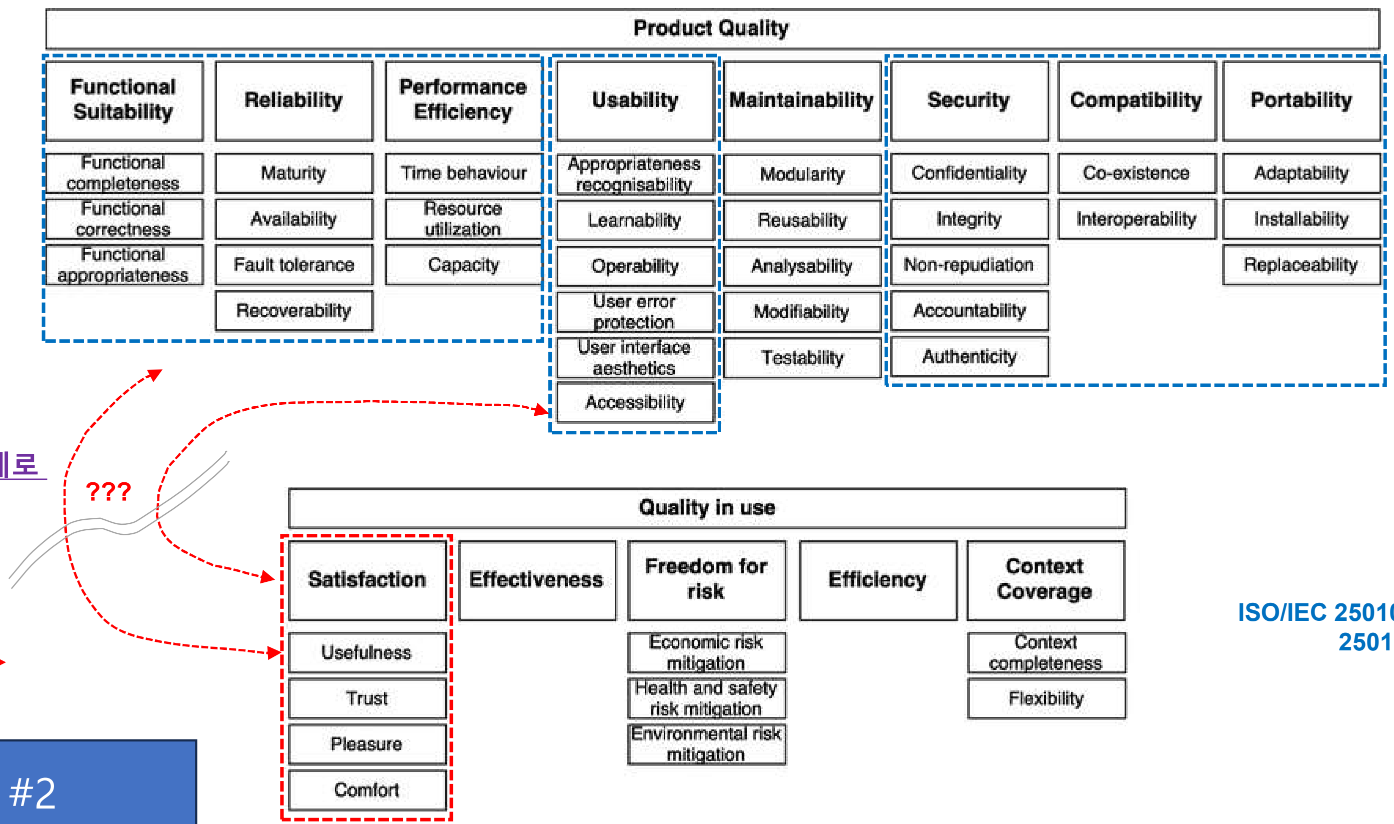
Internal/External Quality가 완벽하게 구별되지 않을 수 있음.

ISO/IEC 25010:2011 SQuaRE – System and Software Quality Model

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

Architecture Design

ISO/IEC 25010:2011
25010:2023



Architecture Design이 실제로 CX와 연결되는지 분석 필요
→ 시작은 QAS부터!!!

ISO/IEC 25010:2011 + 25022:2016
25019:2023

CX
(Customer eXperience)

조별 25010 발표 #2

ISO/IEC 25023:2016 SQuaRE – Product Quality Measurement

- ISO/IEC 25023:2016 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - **Measurement of system and software product quality**
  - Based on ISO/IEC 25010:2011

Characteristics	Sub-characteristics	Measure
Performance Efficiency	Time Behavior	*Mean Response Time*
		Response Time Adequacy
		Mean Turnaround Time
		Turnaround Time Adequacy
		Mean Throughput
	Resource Utilization	*Mean Processor Utilization*
		Mean Memory Utilization
		Mean I/O Devices Utilization
		Bandwidth Utilization
	Capacity	*Transaction Processing Capacity*
		User Access Capacity
		User Access Increase Adequacy

ID	Name	Description	Measurement function
PTb-1-G	**Mean response time**	How long is the mean time taken by the system to respond to a user task or system task?	$X = \sum_{i=1 \text{ to } n} (A_i) / n$ A_i = Time taken by the system to respond to a specific user task or system task at i-th measurement n = Number of responses measured
PTb-2-G	**Response time adequacy**	How well does the system response time meet the specified target?	X = A/B A = Mean response time measured by PTb-1-G B = Target response time specified

INTERNATIONAL STANDARD **ISO/IEC 25023**

First edition
2016-06-15

Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality

Ingénierie des systèmes et du logiciel — Exigences de qualité et évaluation des systèmes et du logiciel (SQuaRE) — Mesurage de la qualité du produit logiciel et du système

Reference number
ISO/IEC 25023:2016(E)

Measures for Product Qualities in ISO/IEC 25010:2011

Characteristics	Sub-characteristics	Measure
Functional Suitability	**Functional Completeness**	*Functional Coverage*
	Functional Correctness	*Functional Correctness*
	Functional Appropriateness	*Functional Appropriateness of Usage Objective*
		Functional Appropriateness of the Systems
Performance Efficiency	**Time Behavior**	*Mean Response Time*
		Response Time Adequacy
		Mean Turnaround Time
		Turnaround Time Adequacy
		Mean Throughput
	Resource Utilization	*Mean Processor Utilization*
		Mean Memory Utilization
		Mean I/O Devices Utilization
		Bandwidth Utilization
	Capacity	*Transaction Processing Capacity*
		User Access Capacity
		User Access Increase Adequacy
Compatibility	**Co-Existence**	*Co-Existence with Other Products*
	Interoperability	*Data Formats Exchangeability*
		Data Exchange Protocol Sufficiency
		External Interface Adequacy

Characteristics	Sub-characteristics	Measure
Usability	**Appropriateness Recognizability**	*Description Completeness*
		Demonstration Coverage
		Entry Point Self-Descriptiveness
	Learnability	*User Guidance Completeness*
		Entry Fields Defaults
		Error Messages Understandability
		Self-Explanatory User Interface
	Operability	*Operational Consistency*
		Message Clarity
		Functional Customizability
		User Interface Customizability
		Monitoring Capacity
		Undo Capacity
		Understandable Categorization of Information
		Appearance Consistency
		Input Device Support
	User Error Protection	*Avoidance of User Operation Error*
		User Entry Error Correction
		User Error Recoverability
	User Interface Aesthetics	*Appearance Aesthetics of User Interfaces*
	Accessibility	*Accessibility for Users with Disabilities*
		Supported Languages Adequacy

Measures for Product Qualities in ISO/IEC 25010:2011

Characteristics	Sub-characteristics	Measure
Reliability	**Maturity**	*Fault Correction*
		Mean Time Between Failure (MTBF)
		Failure Rate
		Test Coverage
	Availability	*System Availability*
		Mean Down Time
	Fault Tolerance	*Failure Avoidance*
		Redundancy of Components
		Mean Fault Notification Time
	Recoverability	*Mean Recovery Time*
		Backup Data Completeness
Security	**Confidentiality**	*Access Controllability*
		Data Encryption Correctness
		Strength of Cryptographic Algorithms
	Integrity	*Data Integrity*
		Internal Data Corruption Prevention
		Buffer Overflow Prevention
	Non-Repudiation	*Digital Signature Usage*
	Accountability	*User Audit Trial Completeness*
		System Log Retention
	Authenticity	*Authentication Mechanism Sufficiency*
		Authentication Rules Conformity

Characteristics	Sub-characteristics	Measure
Maintainability	**Modularity**	*Coupling of Components*
		Cyclomatic Complexity Adequacy
	Reusability	*Reusability Assets*
		Coding Rules Conformity
	Analyzability	*System Log Completeness*
		Diagnosis Function Effectiveness
		Diagnosis Function Sufficiency
	Modifiability	*Modification Efficiency*
		Modification Correctness
		Modification Capability
	Testability	*Test Function Completeness*
		Autonomous Testability
		Test Restartability
Portability	**Adaptability**	*Hardware Environmental Adaptability*
		System Software Environmental Adaptability
		Operational Environmental Adaptability
	Installability	*Installation Time Efficiency*
		Ease of Installation
	Replaceability	*Usage Similarity*
		Product Quality Equivalence
		Functional Inclusiveness
		Data Reusability/Import Capability

Lists of System Quality Attributes (Wikipedia)

Quality attributes [edit]

Notable quality attributes include:

- accessibility
- accountability
- accuracy
- adaptability
- administrability
- affordability
- agility (see Common subsets below)
- auditability
- autonomy [Erl]
- availability
- compatibility
- composability [Erl]
- confidentiality
- configurability
- correctness
- credibility
- customizability
- debuggability
- degradability
- determinability
- demonstrability
- dependability (see Common subsets below)
- deployability
- discoverability [Erl]
- distributability
- durability
- effectiveness
- efficiency
- evolvability
- extensibility
- failure transparency
- fault-tolerance
- fidelity
- flexibility
- inspectability
- installability
- integrity
- interchangeability
- interoperability [Erl]
- learnability
- localizability
- maintainability
- manageability
- mobility
- modifiability
- modularity
- observability
- operability
- orthogonality
- portability
- precision
- predictability
- process capabilities
- producibility
- provability
- recoverability
- redundancy
- relevance
- reliability
- repeatability
- reproducibility
- resilience
- responsiveness
- reusability [Erl]
- robustness
- safety
- scalability
- seamlessness
- self-sustainability
- serviceability (a.k.a. supportability)
- securability (see Common subsets below)
- simplicity
- stability
- standards compliance
- survivability
- sustainability
- tailorability
- testability
- timeliness
- traceability
- transparency
- ubiquity
- understandability
- upgradability
- usability
- vulnerability

Many of these quality attributes can also be applied to data quality.

Tactics

- **Tactics** are the building blocks of design and the raw materials, from which patterns, frameworks, and styles are constructed.
  - **Techniques** that architects have been using for years to manage **quality attribute** response goals
  - Design decisions that influence the control of a quality attribute response
  - Building blocks of architectural patterns

- If architects decides to use a tactics for a quality attribute, then a corresponding architecture should be accompanied.
  - **Availability**
  - **Interoperability**
  - **Modifiability**
  - **Performance**
  - **Security**
  - **Testability**
  - **Usability**

조별 Tactics 발표 #3

Example : Tactics for Availability

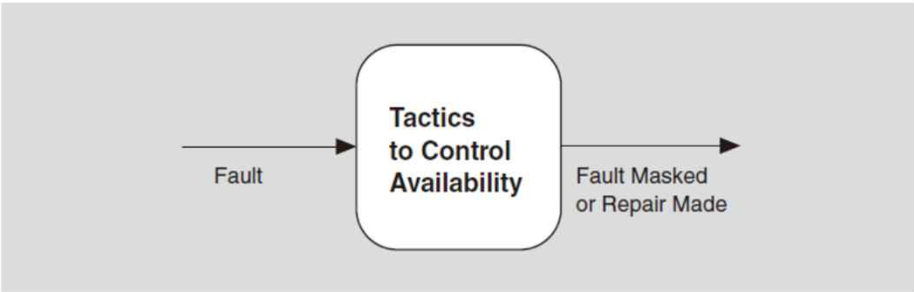
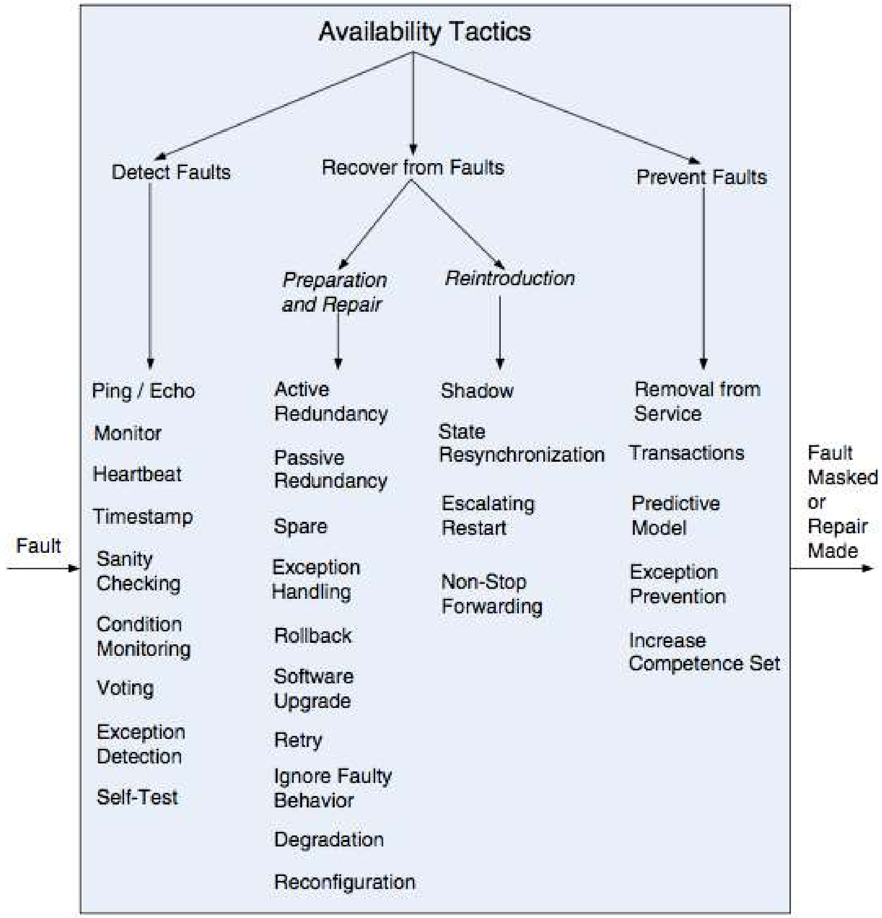


FIGURE 5.4 Goal of availability tactics



Techniques and Principles
> Tactics

3.3 Constraints

- **Constraint**
  - Restrictions on the design or implementation choices available to the developer
  - Can be imposed by external stakeholders and by other systems that interact with the system
  - Should be respected and generally non-negotiable

- **Design Purpose**
- **Architectural Concerns**
- **Constraints**

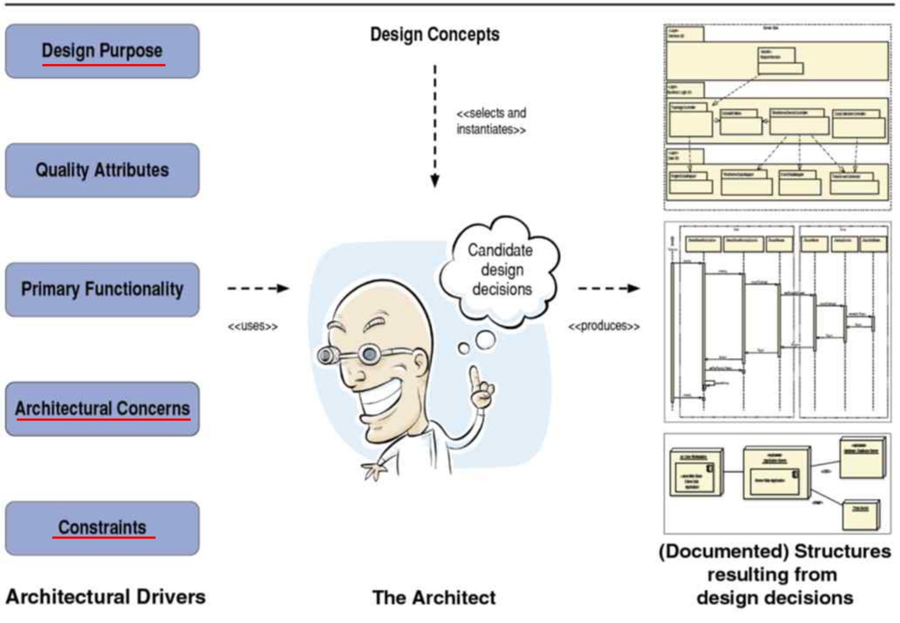


FIGURE 2.1 Overview of the architecture design activity
(Architect image © Brett Lamb I Dreamstime.com)

Design Purpose

- Should be clear about **the purpose of the design** that you want to achieve
  - *"When and why are you doing this architecture design?"*
  - *"Which business goals is the organization most concerned about at this time?"*

  - Examples :
    - You may be doing architecture design as part of a project proposal.
    - You may be doing architecture design as part of the process of creating an exploratory prototype.
    - You may be designing your architecture during development.

Architectural Concerns

- **Additional aspects** that need to be considered as part of architectural design but that are not expressed as traditional requirements.
  - General concerns
    - “Broad” issues that one deals with in creating the architecture
    - Examples: establishing an overall system structure, the allocation of functionality to modules, the allocation of modules to teams, organization of the code base, startup and shutdown, and supporting delivery, deployment, and updates
  - Specific concerns
    - More detailed system-internal issues
    - Examples: exception management, dependency management, configuration, logging, authentication, authorization, caching, and so forth that are common across large numbers of applications
  - Internal requirements
    - Usually not specified explicitly in traditional requirement documents, as customers usually seldom express them. Address aspects that facilitate development, deployment, operation, or maintenance of the system.
    - Called “derived requirements”
  - Issues
    - Results from analysis activities such as design review.
    - May not be present initially.

Constraints

- Decisions over which you have **little or no control** as an architect:
  - Mandated technologies
  - Other systems with which your system needs to interoperate or integrate
  - Laws and standards that must be complied with
  - The abilities and availability of your developers
  - Deadlines that are non-negotiable
  - Backward compatibility with older versions of systems, and so on.

Our Constraints

- A **constraint** is fixed decisions premade 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.
        - Externally imposed limitation on system requirements, design, implementation, or the process used to develop or modify a system

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

- 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.

Business Constraints

- **Business constraints** are indirect constraints on the design space.
  - Not specify that a particular technology is used to design or build a system
  - But impose cost, schedule, regulatory, legal, marketing, and other similar demands that will influence the design of the system

Kind	Description
Cost limitations	How much over what period (time) can be spent on the system or product?
Schedule limitations	What are the delivery schedules? One delivery? Incremental? What functionality must be delivered at what point in time?
Regulatory restrictions and demands	Are there any regulations imposed on the system, product, or organization designing and building the system, or the customer stakeholders' organization?
Legal restrictions and demands	Are there any legal impositions placed on the system, product, or organization designing and building the system, or the customer stakeholders' organization?
Market restrictions and demands	Does the target market impose any restrictions or demands on the system or product, especially if it could prevent entry into another market?
Organizational restrictions and demands	Do any of the organizations involved in the project have policies, processes, resources or lack thereof, or structural issues that could impose restrictions or demands on the design or construction of the system or product?
Logistical issues	Are there logistical issues such as deployment, transportation, supplier/supply chain, and similar that could impact the design of the system?

Technical Constraints

- **Technical constraints** have direct influence on the design.
  - Specific technologies, tools, languages, and databases that must be used or avoided
  - Required development conventions or standards

Kind	Description
Operating system	Are there any constraints to use a particular OS? Are there any constraints to support multiple OSs?
Platform	Are there any constraints to use a particular platform?
Programming languages	Is there a constraint to use a particular programming language?
Peripheral or network hardware	Are there any constraints that specify that particular peripheral devices or network hardware be used?
Commercial products	Is there a constraint that specific commercial hardware and software products be used?
Tools and methods	Are there any constraints that specify that certain tools (e.g., design/programming tools) or technical methods be used?
Protocols, interfaces, standards	Are there any constraints that specify that certain protocols, interfaces, or standards be used or adhered to during development?
Legacy hardware and software	Are there any constraints that indicate that the new system/product must utilize or interact with any legacy hardware or software systems or elements?

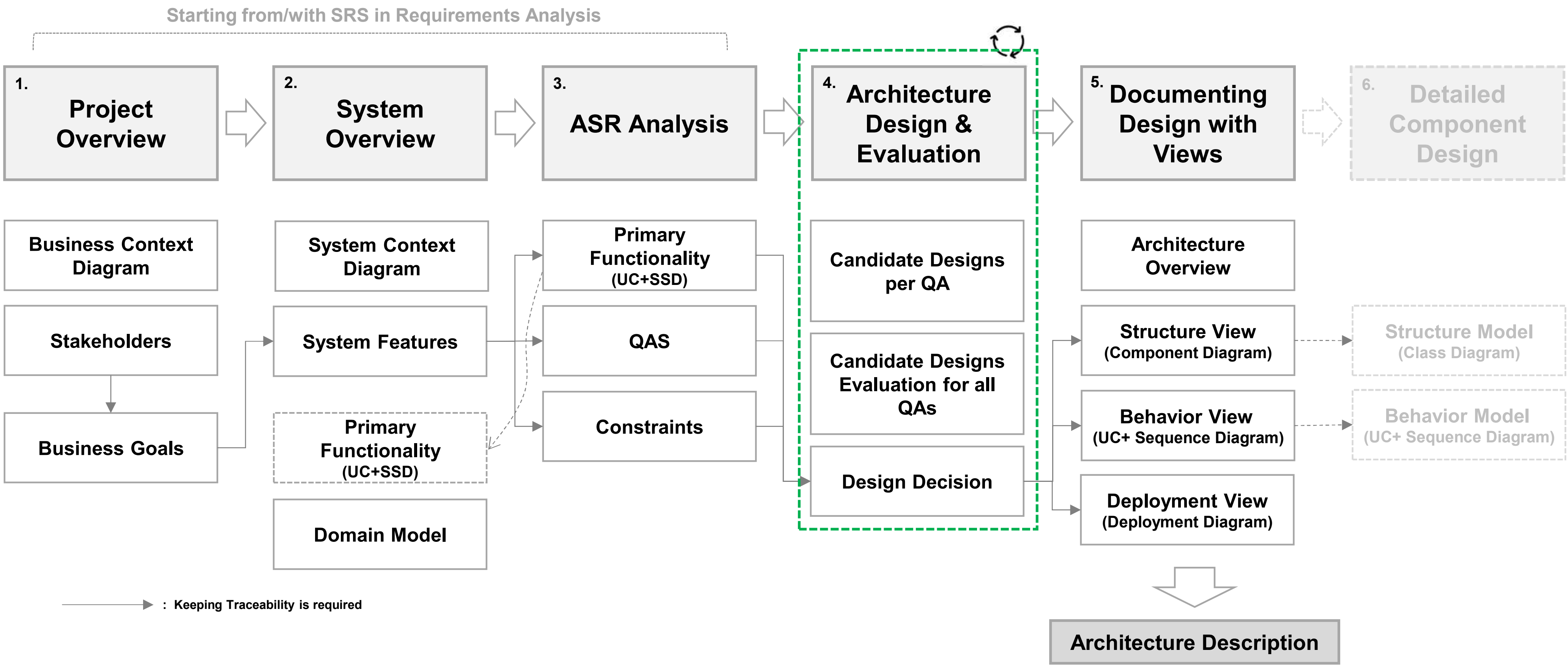
Technical Constraints : Examples

유형	메트릭	허용 최대값				
		근거				
		MISRA	SCR-G	JPL	JSF	HIS
크기	Lines of Code(LOC)	80	200	60	200	50
	Comment Frequency	50%	30%	-	-	-
복잡도	Cyclomatic Complexity(CC)	15	20	-	20	10
	Number of Execution Paths(NPath)	75	-	-	-	80
	Number of Structuring Levels	6	6	-	-	4
결합도 / 모듈화	Number of Parameters	-	8	6	6	5
	Fan In	-	8	-	-	5
	Fan Out	-	10	-	-	7
	Number of Calling Levels	8	-	-	-	4

* MISRA: MISRA Report 5, Software Metrics
* SCR-G: 무기체계 소프트웨어 개발 및 관리 매뉴얼, 소프트웨어 신뢰성/보안성 시험 절차
* JPL: JPL(Jet Propulsion Lab.) Coding Standard for the C
* JSF: Joint Strike Fighter Air Vehicle C++ Coding Standards
* HIS: HIS(Audi, BMW 등 5개 자동차 업체 그룹) Source Code Metrics

4. Architecture Design & Evaluation

4. Architecture Design & Evaluation



Where We are Now in AD

1. Project Overview
    - 1.1 Project Background
    - 1.2 Business Context Diagram
    - 1.3 Stakeholders
    - 1.4 Business Goals

2. System Overview
    - 2.1 System Context Diagram
    - 2.2 External Entity
    - 2.3 External Interface
    - 2.4 System Features
    - 2.5 Domain Model
    - 2.6 Assumptions

3. Architectural Drivers
    - 3.1 Primary Functionality
    - 3.2 Quality Attribute Scenario
    - 3.3 Constraints

4. Architecture Design & Evaluation
    - 4.1 Candidate Designs per QA
    - 4.2 Candidate Designs Evaluation for all QAs
    - 4.3 Design Decision

5. Architecture Design Description
    - 5.1 Architecture Overview
    - 5.2 Structure View
    - 5.3 Behavior View
    - 5.4 Deployment View

6. Component Design Description
    - 6.1.2 Component Structure Model
    - 6.1.5 Component Behavior Model

7. Architecture Traceability Summary

Typical Architecture Design Approach in a Nutshell

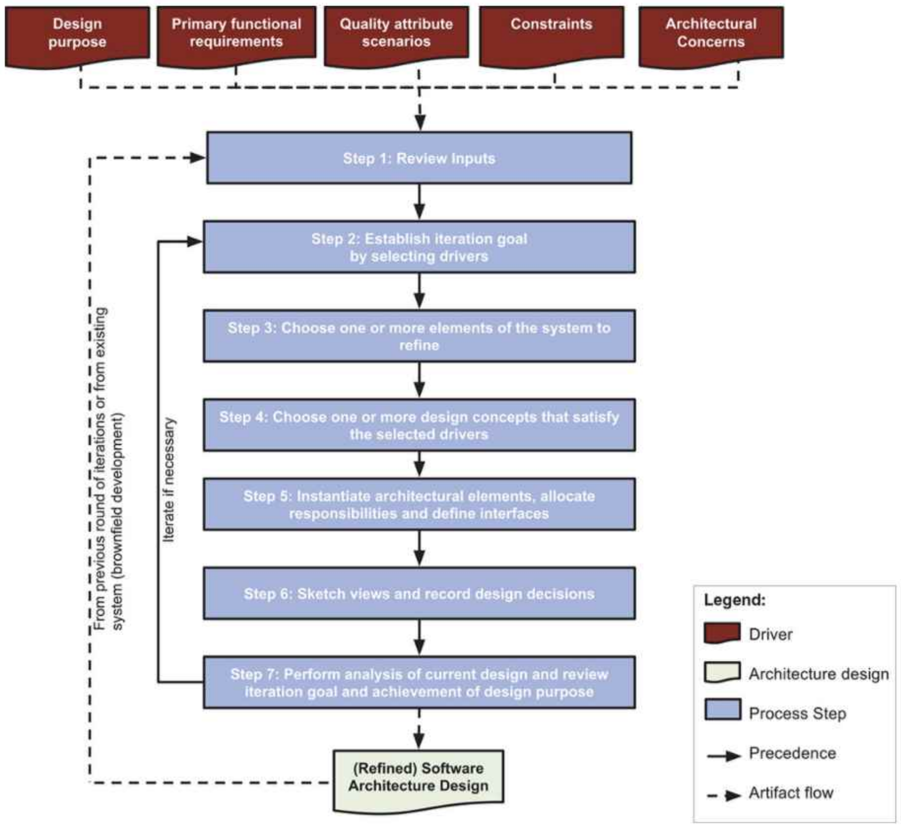


FIGURE 3.1 Steps and artifacts of ADD version 3.0

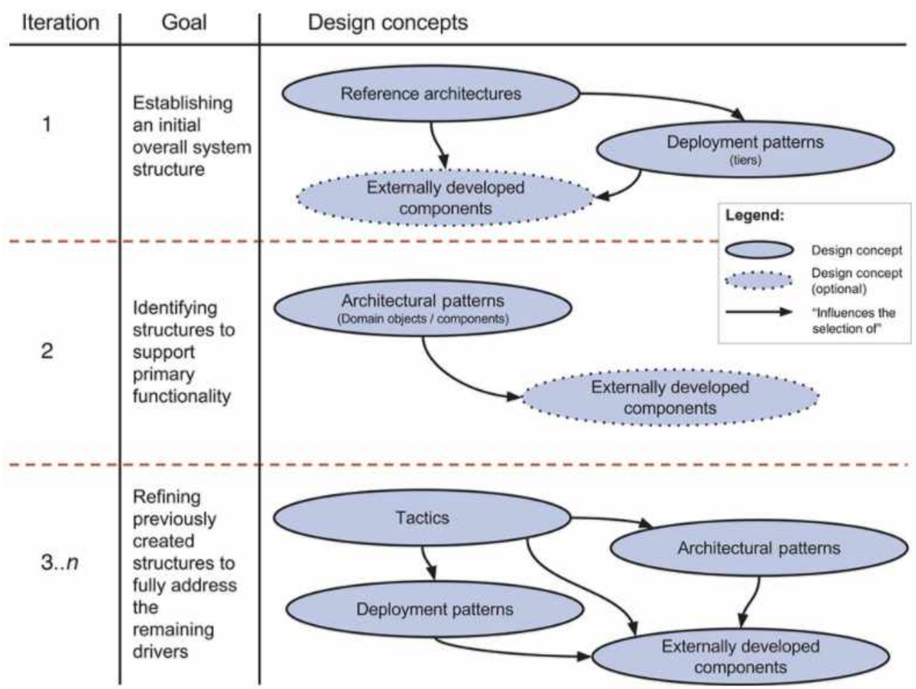
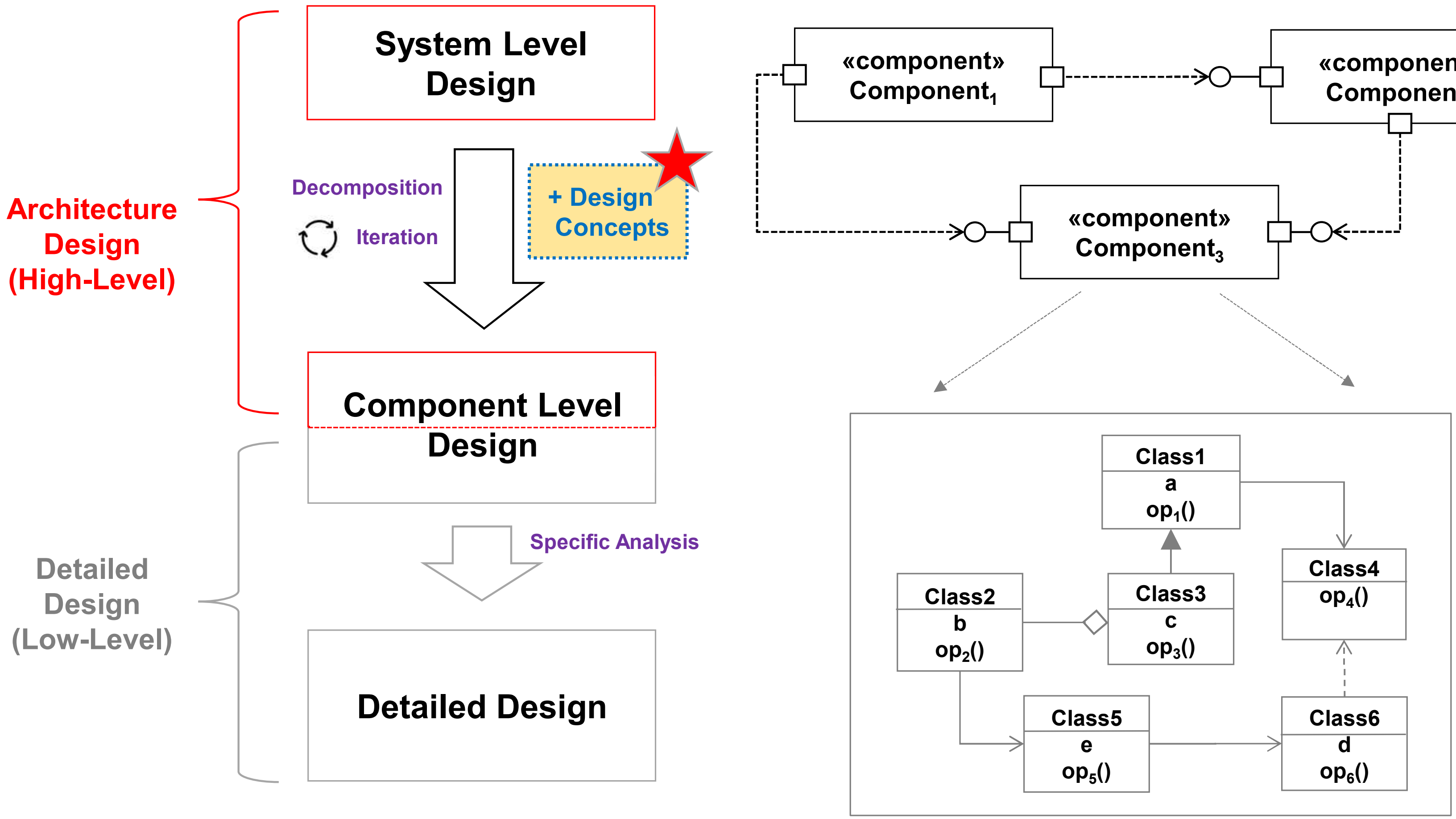


FIGURE 3.2 Design concept selection roadmap for greenfield systems

Our Architecture Design Process



Architecture Design Description

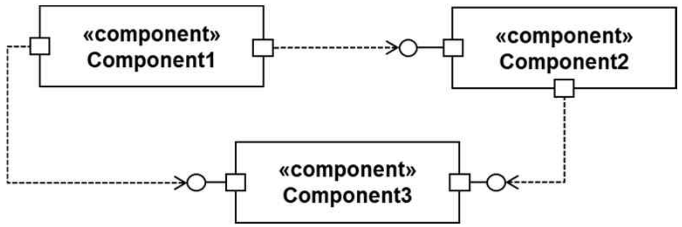
- **Architecture Overview** (5.1)
- **Structure View** (5.2)
- **Behavior View** (5.3)
- **Deployment View** (5.4)

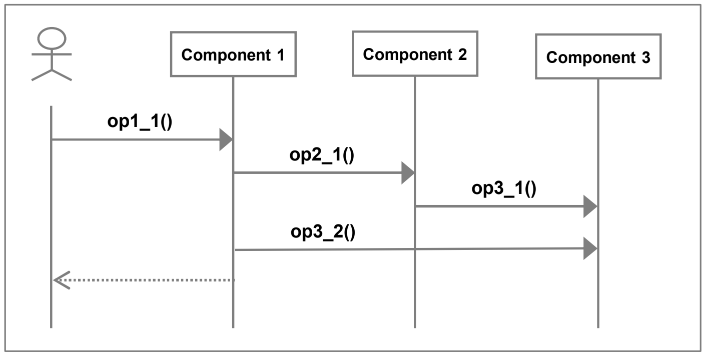
Detailed Component Design Description

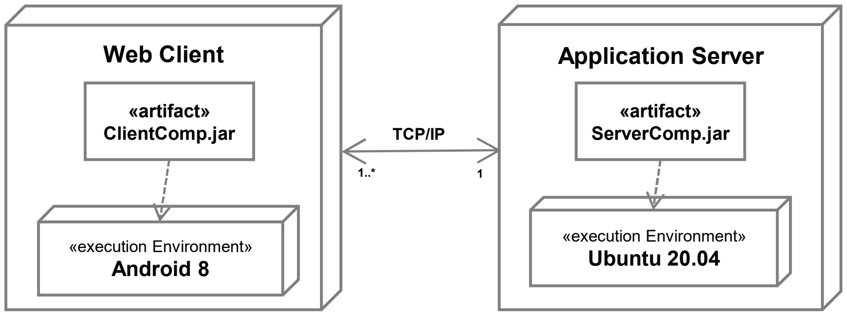
- Component Structure Diagram (6.1.2)
- Component Behavior Diagram (6.1.5)

Our Architecture Design Output : Architecture Description (5.1 ~ 5.4)

- Architectural design is described through **multiple views**.
  - **Architecture Overview (5.1)** ← UML Deployment Diagram (Component / Class Diagram)
  - **Structure View (5.2)** ← UML Component Diagram
  - **Behavior View (5.3)** ← UML Sequence Diagram (+ Use Case)
  - **Deployment View (5.4)** ← UML Deployment Diagram







Design Concepts

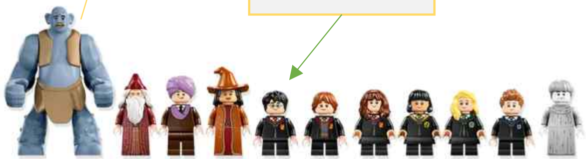
- **Design concepts** are the **building blocks** for creating structures.
  - **Reference Architecture**
  - **Deployment Pattern**
  - **Architectural Style**
  - **Tactics**
  - **Externally Developed Components**

전체 건물의 큰 뼈대(Frame)
→ Reference Architecture
→ Deployment Pattern
→ Architecture Style

부드러운 잘 여닫히는 도어
→ Tactics

Stakeholders' Concern

"적어도 내 공간에서는 머리가 천정에 안 닿았으면 좋겠어!"



Stakeholders

"내 방 문은 언제든지 뛰어 나갈 수 있도록 잘 여닫혀야 해!"

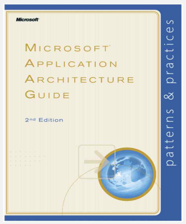
손쉬운 탈부착 가능한 구조물
→ Reference Architecture
→ Deployment Pattern
→ 또는 Tactics

Design Concepts

Reference Architecture

- Blueprint for overall architecture
- Logical structure for specific application types
- Embodying architecture styles

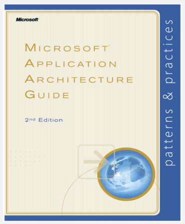
Examples:
- Mobile applications
- Rich client applications
- Rich internet applications
- Service applications
- Web applications



Deployment Pattern

- How to deploy logical into physical
- Physical structure
- Essential for many QAs (Performance, Security, Availability, …)

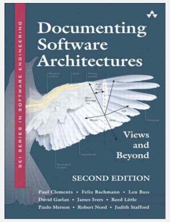
Examples:
- Nondistributed
- Distributed
- Performance
- Reliability
- Security

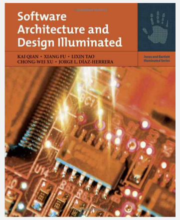


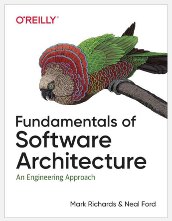
Architecture Style

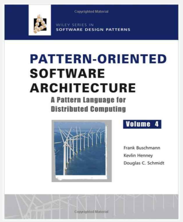
- General and reusable structural layout and its properties
- Not domain-specific
- Logical >> Physical structure

Examples:
- MVC, MVP, MVVM, Layered
- Client-Server, SOA, Microservices
- Pipes and Filters, Blackboard





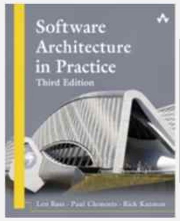




Tactics

- Building blocks of other patterns
- Widely-used techniques to manage QAs by architects
- Quickly evolved

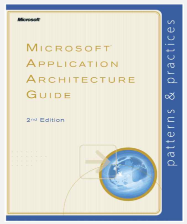
Examples:
- Availability
- Interoperability
- Modifiability
- Performance
- Security
- Testability
- Usability



Externally Developed Components

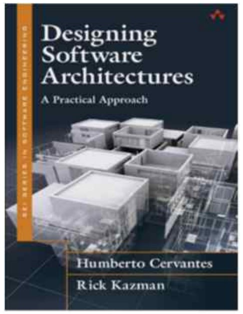
- Generally called COTS software (Commercial Off-The-Shelf)

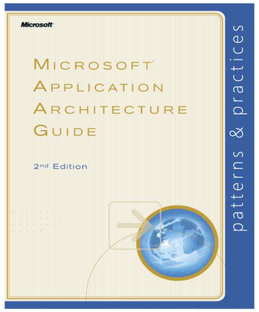
Examples:
- Technical family
- Products (COTS)
- Application framework (Hibernate, Rest, Spring, Swing, etc.)
- Platform (Java, .Net, Google Cloud, etc.)



Design Concepts 1. Reference Architectures

- **Blueprints** that provide an **overall logical structure** for particular types of applications
  - **Mobile applications**
  - **Rich client applications**
  - **Rich internet applications**
  - **Service applications**
  - **Web applications**





- Reference architectures and architectural styles are different.
  - **Architectural styles** (such as "Pipe and Filter" and "Client Server") define types of components and connectors in a specified topology that are useful for structuring an application either logically or physically.
    - Such styles are technology and domain independent.
  - **Reference architectures** provide a structure for applications in specific domains, and they may embody different styles.
    - While architectural styles tend to be popular in academia, reference architectures seem to be preferred by practitioners.

Summary of Application Types

- **Mobile applications**
  - Applications of this type can be developed as thin client or rich client applications. Rich client mobile applications can support disconnected or occasionally connected scenarios. Web or thin client applications support connected scenarios only. The device resources may prove to be a constraint when designing mobile applications.
- **Rich Client applications**
  - Applications of this type are usually developed as stand-alone applications with a graphical user interface that displays data using a range of controls. Rich client applications can be designed for disconnected and occasionally connected scenarios because the applications run on the client machine.
- **Rich Internet applications**
  - Applications of this type can be developed to support multiple platforms and multiple browsers, displaying rich media or graphical content. Rich Internet applications run in a browser sandbox that restricts access to some devices on the client.
- **Service applications**
  - Services expose complex functionality and allow clients to access them from local or remote machine. Service operations are called using messages, based on XML schemas, passed over a transport channel. The goal in this type of application is to achieve loose coupling between the client and the server.
- **Web applications**
  - Applications of this type typically support connected scenarios and can support different browsers running on a range of operating systems and platforms.

Example : Mobile Application Reference Architecture

- A mobile application will normally be structured as a multilayered application consisting of user experience, business, and data layers.

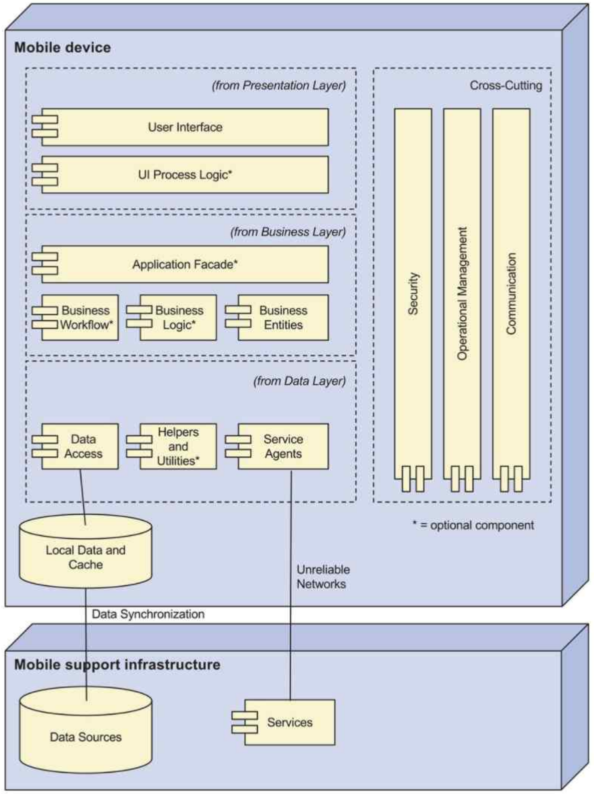


FIGURE A.4 Mobile Application reference architecture (Key: UML)

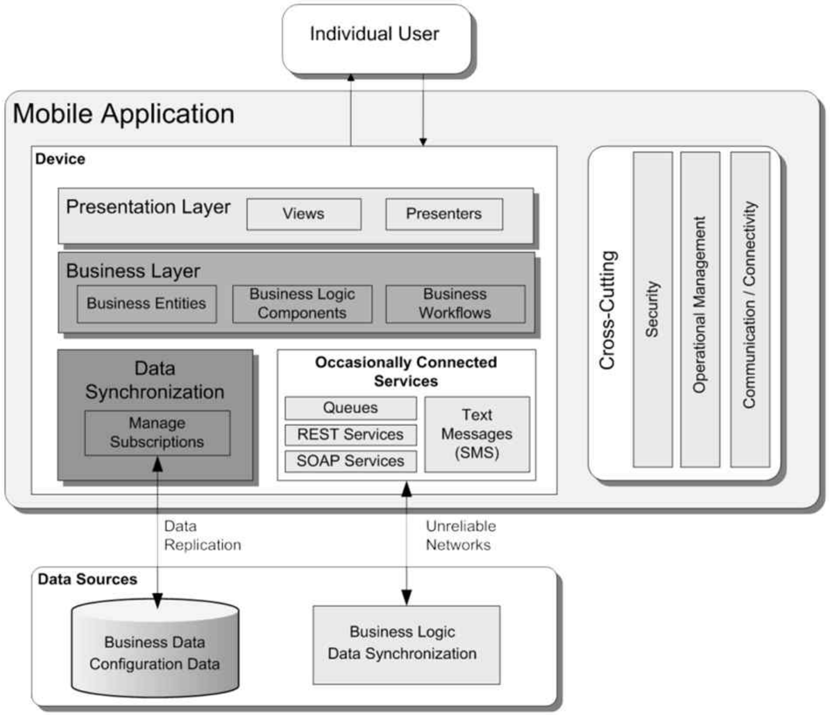
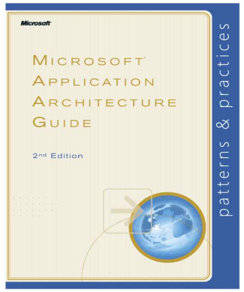


Figure 1 Mobile application archetype



Design Concepts 2. Deployment Patterns

- **Deployment patterns** model **how to physically structure** the system to deploy it.
  - Provide guidance on how to structure the system from a physical standpoint.
  - An initial structure for the system is obtained by **mapping** the **logical** elements that are obtained from reference architectures (and other patterns) **into** the **physical** elements defined by deployment patterns.
  - Good decisions with respect to the deployment of the software system are essential to achieve important quality attributes such as performance, usability, availability, and security.

- Deployment patterns:
  - **Nondistributed**
  - **Distributed**
  - **Performance**
  - **Reliability**
  - **Security**

Example : Nondistributed vs. Distributed

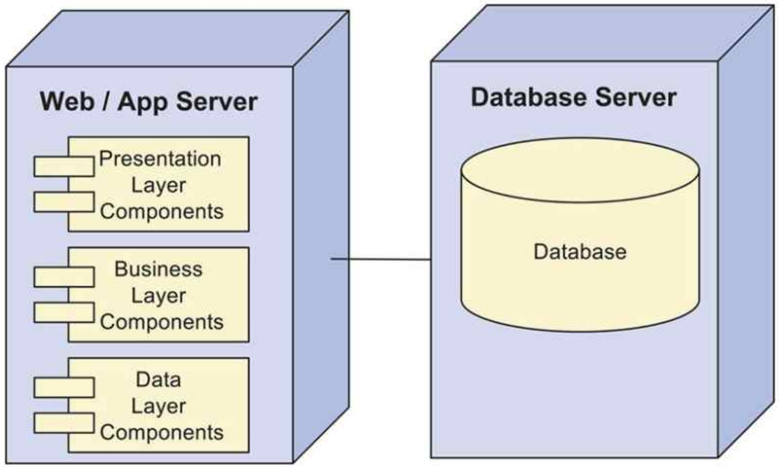


FIGURE A.6 Nondistributed deployment example (Key: UML)

vs.

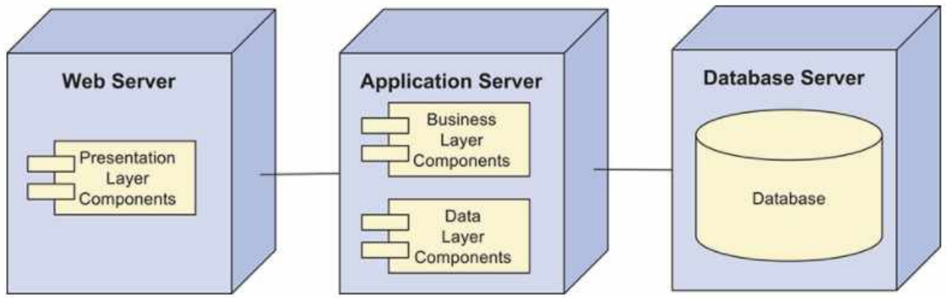


FIGURE A.7 Distributed deployment example (Key: UML)

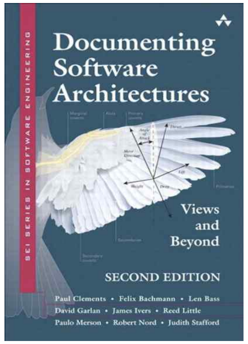
Techniques and Principles
> Deployment Patterns

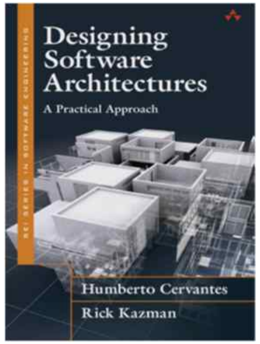
Design Concepts 3. Architectural Design Patterns

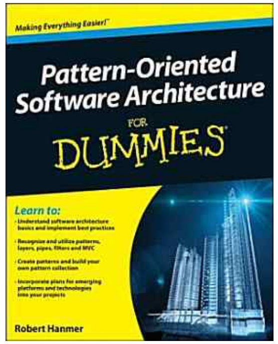
- **Software Architecture Style/Pattern** is a description of <u>general and reusable structural layout</u> and its properties to a commonly occurring structural problems in software architecture.

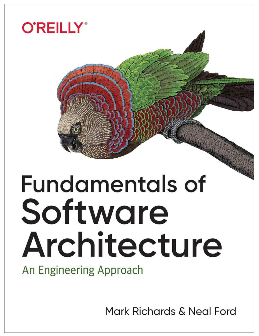
Catalog of architectural patterns [edit]

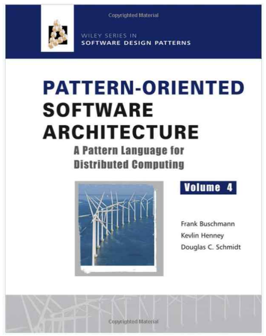
- Multitier architecture
- Model–view–controller
- Domain-driven design
- Blackboard pattern
- Sensor–controller–actuator
- Presentation–abstraction–control
- Component-based
- Monolithic application
- Layered
- Pipes and filters
- Database-centric
- Blackboard
- Rule-based
- Event-driven aka implicit invocation
- Publish-subscribe
- Asynchronous messaging
- Microkernel
- Reflection
- Client-server (multitier architecture exhibits this style)
- Shared nothing architecture
- Space-based architecture
- Object request broker
- Peer-to-peer
- Representational state transfer (REST)
- Service-oriented
- Cloud computing patterns [2]

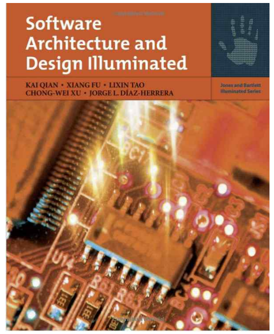








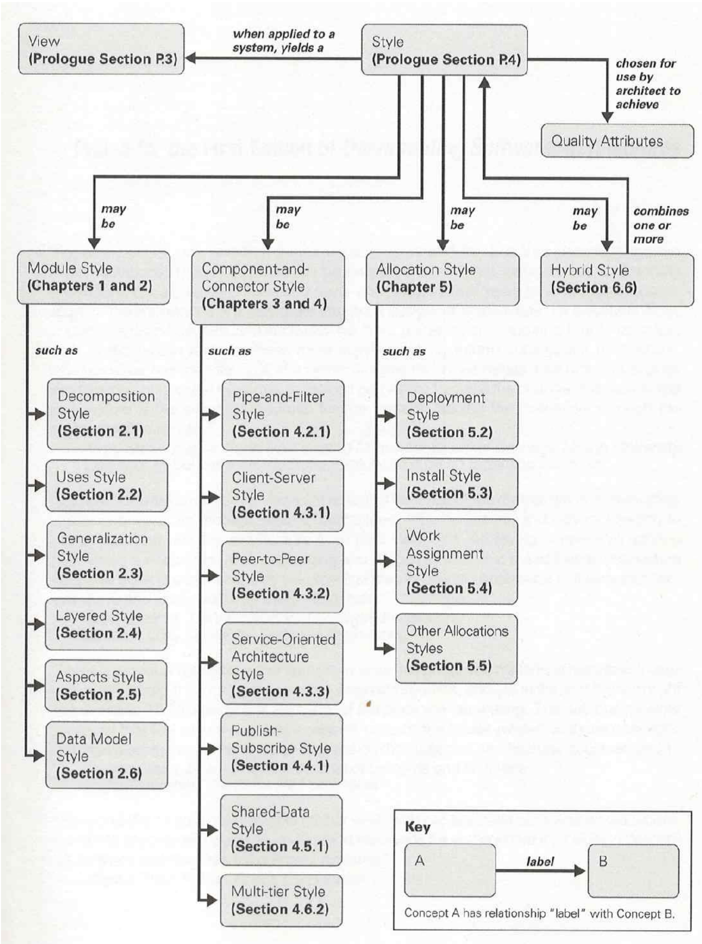


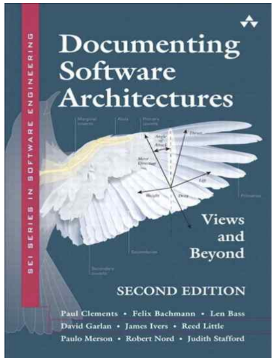


조별 A.+Dpl. Patterns 발표 #4

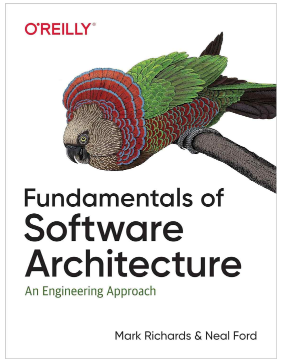
Architectural Design Patterns by CMU

- **Architecture Styles**
  - Module Style
  - Component-and-Connector Style
  - Allocation Style
  - Hybrid Style

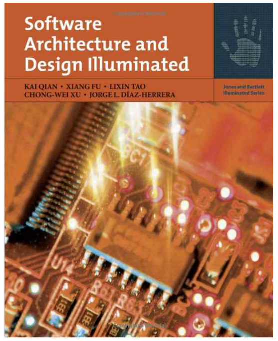




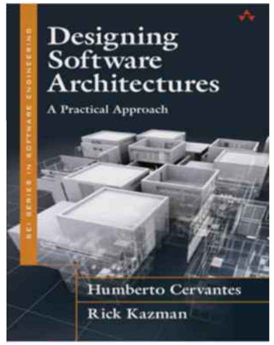
Architectural Design Patterns by Others



Architectural Design Patterns by Others

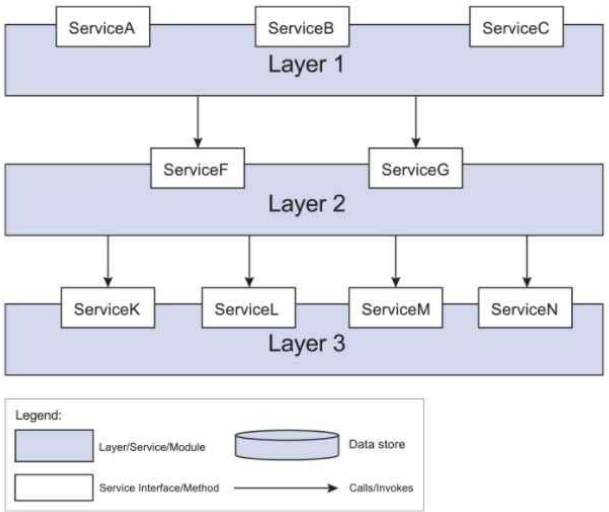


Architectural Design Pattern : Layers, Domain Object



Name	Layers
Problem and context	When transforming a Domain Model (182) into a set of modules that can be allocated to teams, [...] we need to support several concerns: the independent development of the modules, the independent evolution of the modules, the interaction among the modules.
Solution	Define two or more layers for the software under development, where each layer has a distinct and specific responsibility. To make the layering more effective, the interactions between the layers should be highly constrained. The strictest layering, as shown below, allows only unidirectional dependencies and forbids layer-bridging.

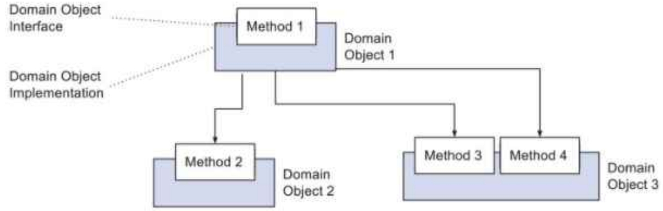
Structure



Consequences and related patterns	Typically, each self-contained and coherent responsibility within a layer is realized as a separate domain object. Domain objects are the containers (modules) that can be developed and evolved independently.

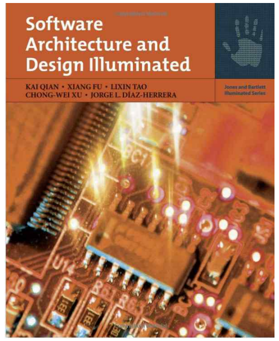
Name	Domain Object
Problem and context	When realizing a Domain Model (182) in terms of Layers (185), a key concern is to decouple self-contained and cohesive application responsibilities.
Solution	Encapsulate each distinct, nontrivial piece of application functionality in a self-contained building block called a domain object.

Structure



Consequences and related patterns	The partitioning of an application's responsibilities into domain objects is based on one or more granularity criteria. There can be different types of domain objects that encapsulate business features, domain concepts, or infrastructure elements. For example, domain objects might be a function such as an income tax calculation or a currency conversion, or a domain concept such as a bank account or a user. Domain objects can also aggregate other domain objects. When designing domain objects, you need to distinguish an Explicit Interface (281), which exports some functionality, from its Encapsulated Implementation (313), which realizes that functionality. The separation of interface and implementation is the key to modularization. It minimizes coupling—each domain object depends only on explicit interfaces, not on encapsulated implementations. This makes it possible to create and evolve a domain object implementation independently from other domain objects.

General Category of Architecture Styles

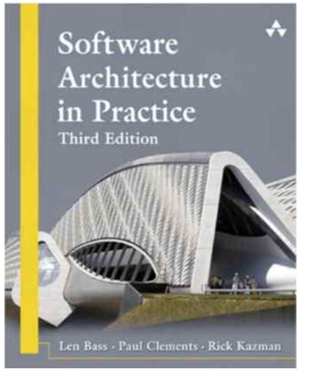


- **Structure**
  - Component-based
  - Monolithic application
  - Layered
  - Pipes and Filters
- **Shared Memory**
  - Data-centric
  - Blackboard
  - Rule-based
- **Messaging**
  - Event-driven
  - Publish-Subscribe
  - Asynchronous messaging
- **Adaptive Systems**
  - Plug-ins
  - Microkernel
  - Reflection
  - Domain specific language
- **Distributed systems**
  - Client-Server (2-tier, 3-tier, n-tier)
  - Peer-to-Peer
  - Object request broker
  - REST (Representational State Transfer)
  - Service-Oriented
  - Microservice
  - Cloud computing patterns

Design Concepts 4. Tactics

- **Tactics** are the building blocks of design and the raw materials, from which patterns, frameworks, and styles are constructed.
    - **Techniques** that architects have been using for years to manage **quality attribute** response goals
    - Design decisions that influence the control of a quality attribute response.
    - Building blocks of architectural patterns

- If architects decides to use a tactics for a quality attribute, then a corresponding architecture should be accompanied.
    - **Availability**
    - **Interoperability**
    - **Modifiability**
    - **Performance**
    - **Security**
    - **Testability**
    - **Usability**



Design Concepts 5. Externally Developed Components

- **Technology families**
  - A technology family represents a group of specific technologies with common functional purposes.
  - Examples: RDBMS, ORM (Object-Oriented to Relational Mapper)
- **Products**
  - A product (or software package) refers to a self-contained functional piece of software that can be integrated into the system that is being designed. Requires only minor configuration or coding. → **COTS**
  - Examples: Oracle, MS SQL Server, MySQL
- **Application frameworks**
  - An application framework (or just framework) is a 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.
  - Examples: Hibernate, Rest, Spring, Swing
- **Platforms**
  - A platform provides a complete infrastructure upon which to build and execute applications.
  - Examples: Java, .Net, Google Cloud

Technology Family: Big Data Domain

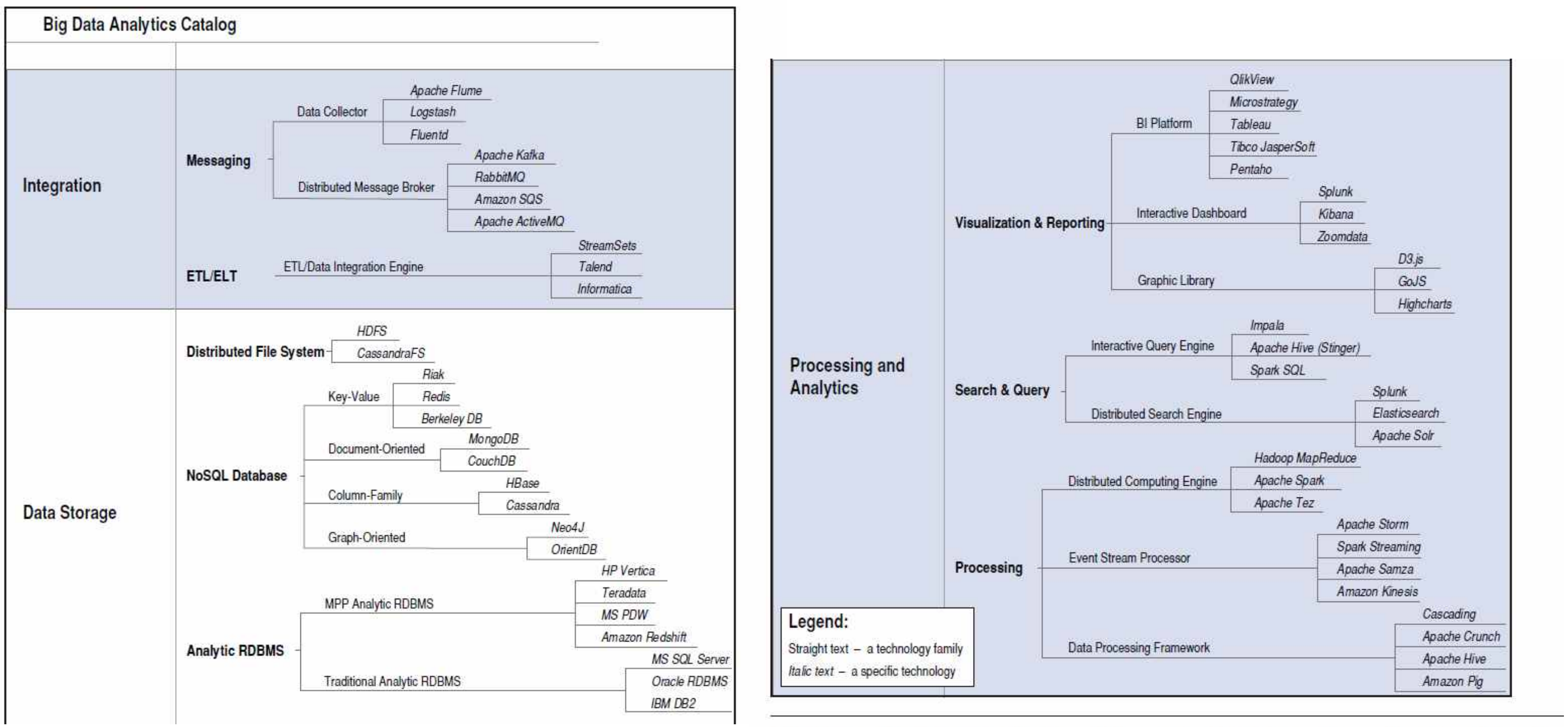
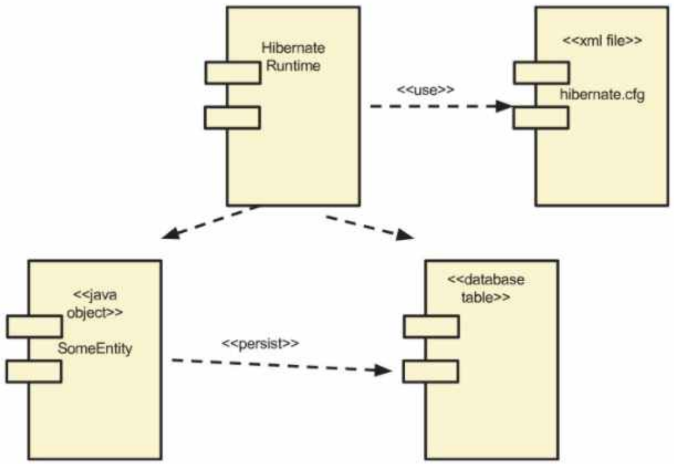


FIGURE 2.10 A technology family tree for the Big Data application domain

Application Framework : Hibernate

Framework Name	Hibernate
Technology family	Object-oriented to relational mapper
Language	Java
URL	http://hibernate.org/
Purpose	Simplify persistence of objects in a relational database.
Overview	Hibernate allows objects to be easily persisted in a relational database (and it supports different database engines). Object-relational mapping rules are described declaratively in an XML file called `hibernate.cfg` or using annotations in the classes whose objects need to be persisted. Hibernate supports transactions and provides a query language called HQL (Hibernate Query Language) that is used to retrieve objects from the database. Hibernate utilizes multilevel caching schemes to improve performance. It also provides mechanisms to allow lazy acquisition of dependent objects to improve performance and reduce resource consumption. These mechanisms are configured declaratively in the configuration files.
Implemented design patterns and tactics	Patterns: ▪ Data Mapper ▪ Resource Cache ▪ Lazy Acquisition Tactics: ▪ Availability: Transactions ▪ Performance: Maintain multiple copies of data (cache)
Benefits	▪ Greatly simplifies the persistence of objects in relational database
Limitations	▪ Complex API ▪ Slower than JDBC (Java Database Connectivity) ▪ Difficult to map to legacy database schemas

Structure



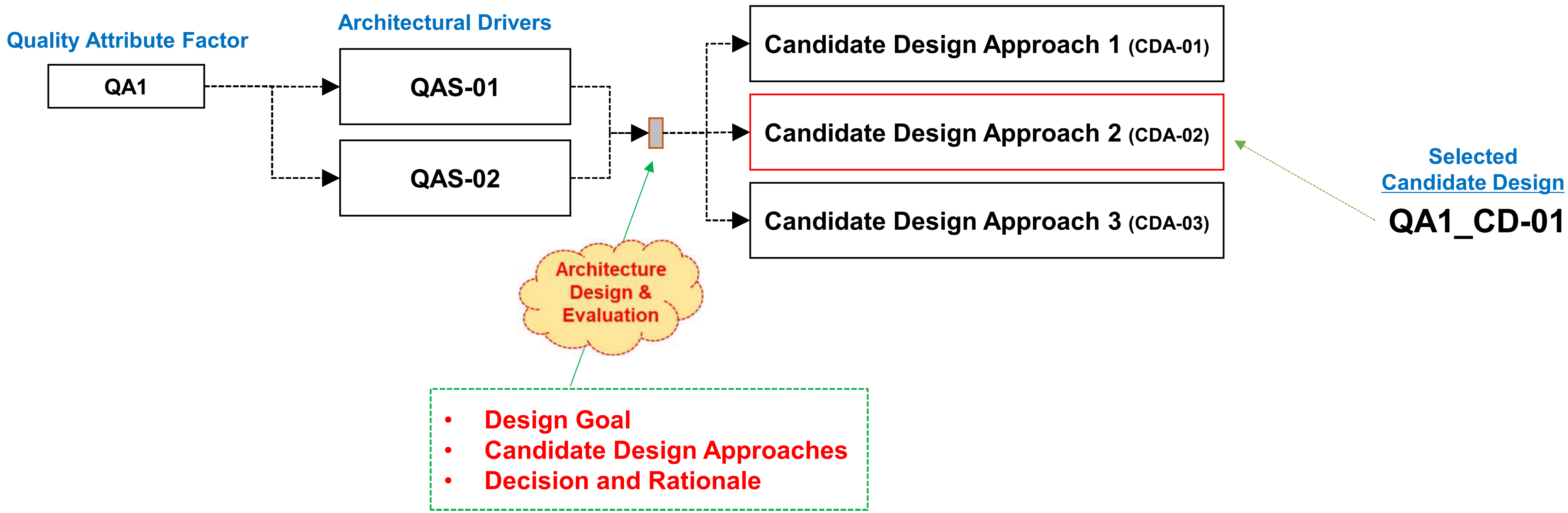
This diagram represents an entity that is persisted to a database by the Hibernate runtime using the information in the configuration file (Key: UML)

4.1 Candidate Designs per QA

- **Why we need to document design decisions?**
  - The process of developing a complex software architecture involves making hundreds of big and small decisions.
    - The results of these decisions are reflected in the views later: the structures with the elements and relations and properties, and the interfaces and behavior of those elements.
  - Understanding the design decisions(i.e., the rationales) is essential for us to acknowledge and improve the design.
    - Most decisions are made in a complex context and almost always involve trade-offs.
  - Just like documenting the architecture helps you design the architecture, documenting the decisions helps you make decisions correctly.

Candidate Design Decision

- **Candidate Design**
  - A candidate of partial architecture design which satisfies with all QASs in a specific QA(Quality Attribute).
  - Proposed, evaluated, and selected by architects for each QA
  - Refining the domain model first is highly recommended.

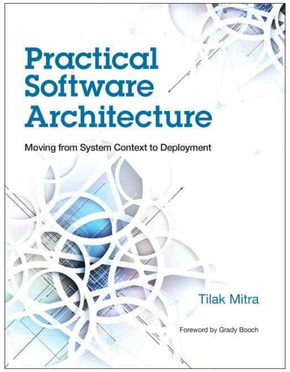


* CDA (Candidate Design Approach) is not a widely used term, but only used in this class.

Design Goal

- Provides a detailed goal of the design decision to achieve a specific QA
    - Stating the architectural design issue being addressed
    - Usually, it is a more elaborated description than the corresponding QAS

Candidate Design Approach (CDA)



- Illustrates with naive figures various design alternatives that have been considered with the objective of solving the problem under consideration.
  - It is okay if some architecture problems have only one alternative and that is the one chosen as the solution (but it is rare.)

- Each design approach is described in detail along with its pros and cons.

CDA ID	**Title** of the approach
Candidate Design Approach (CDA) Description	• Present and describe the design with diagrams • **Describe design concepts applied**. They include reference architectures, architectural styles/patterns, architectural tactics, principles. • Use naive/UML diagrams or View models
Pros	Discuss architectural drivers promoted by the design alternative
Cons	Discuss architectural drivers inhibited by the design alternative

Decision and Rationale

- Describe any design trade-offs relevant to the design decisions in terms of ADs.
  - Describes the rationale behind choosing the solution among the various alternatives, substantiated by a list of architecture design principles that the solution complies with, along with a potential list of principles that may be in noncompliance (substantiated by an explanation for the deviations).

- Select one candidate design approach for the QA.

QA Name		Analysis	**Candidate Design Approach (CDA) #1 (Selected)**	…	Candidate Design Approach #n
ID	Title				
QAS-01		Pros	(+) Description		
		Cons	(-)		
QAS-02		Pros	(++)		
		Cons	(-)		

Candidate Design :

QA	QAS	CD	Description
QA1: Performance	QAS-01 QAS-02	**QA1_CD-01 (+ Title)**	

4.2 Candidate Designs Evaluation for All QAs

- **Architecture Evaluation**
  - Use any approach, technique, and method such as ATAM (Architecture Trade-off Analysis Method)
  - Evaluate all candidate designs (CD) with respect to all QA/QASs together, and select a set of CDs

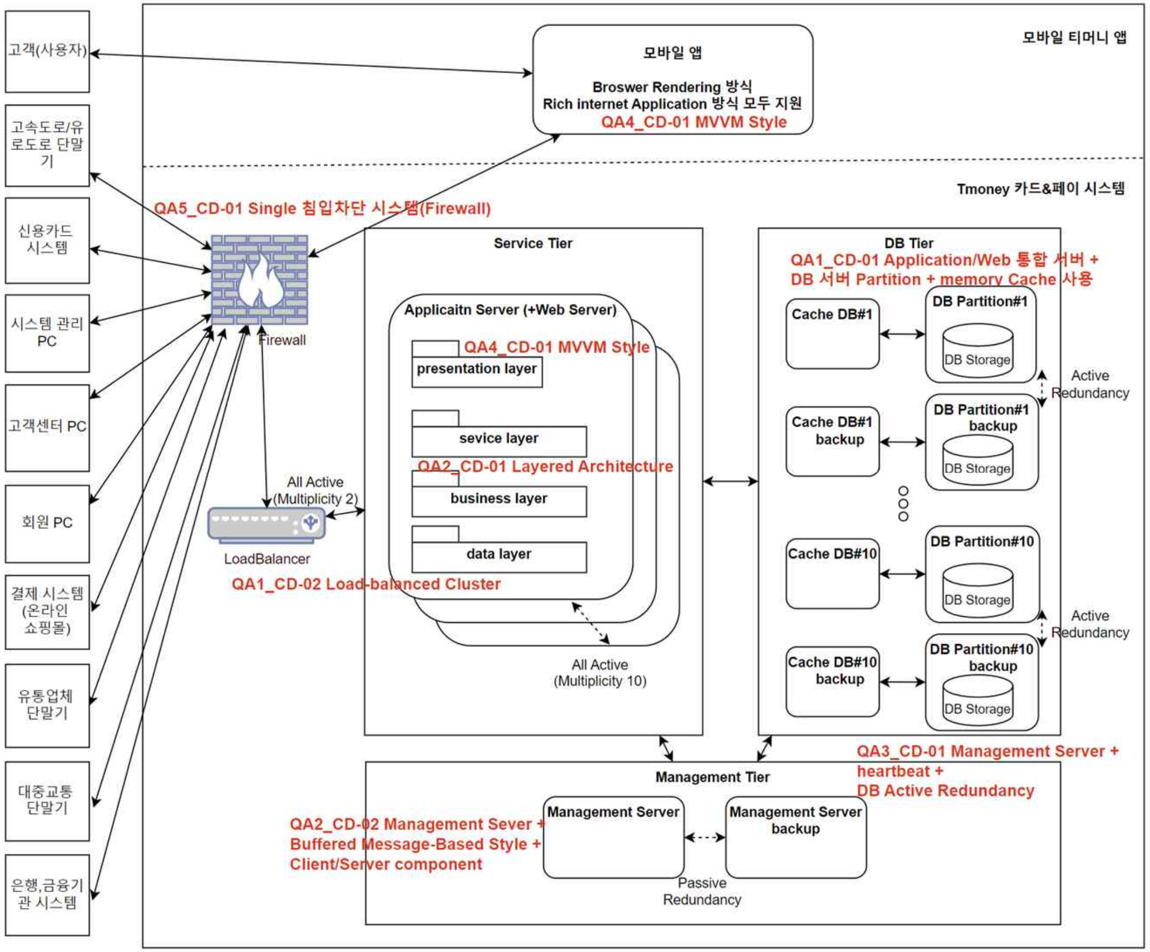
QA	QAS	Analysis	Candidate Design (CD) #1 QA1_CD-01 + Title	QA1_CD-2 + Title	…	QA5_CD1 + Title
QA1 Performance	QAS-01	Pros	(+) Description	(+)		(++)
		Cons	(-)	(--)		(--)
	QAS-02	Pros	(++)	(+)		(++)
		Cons	(-)	(-)		(--)
QA2	QAS-03					
QA3	QAS-04			- (NA)		
QA4						
QA5						

4.3 Design Decision

- Collect all the selected CDs to complete the final design decision (DD).
  - All design decisions that are considered architecturally important to satisfy the business, technical, and engineering goals are captured and summarized.
  - The entire DD is described through a **naive picture**(s). → An upgrade version of **Domain Model (2.5)**

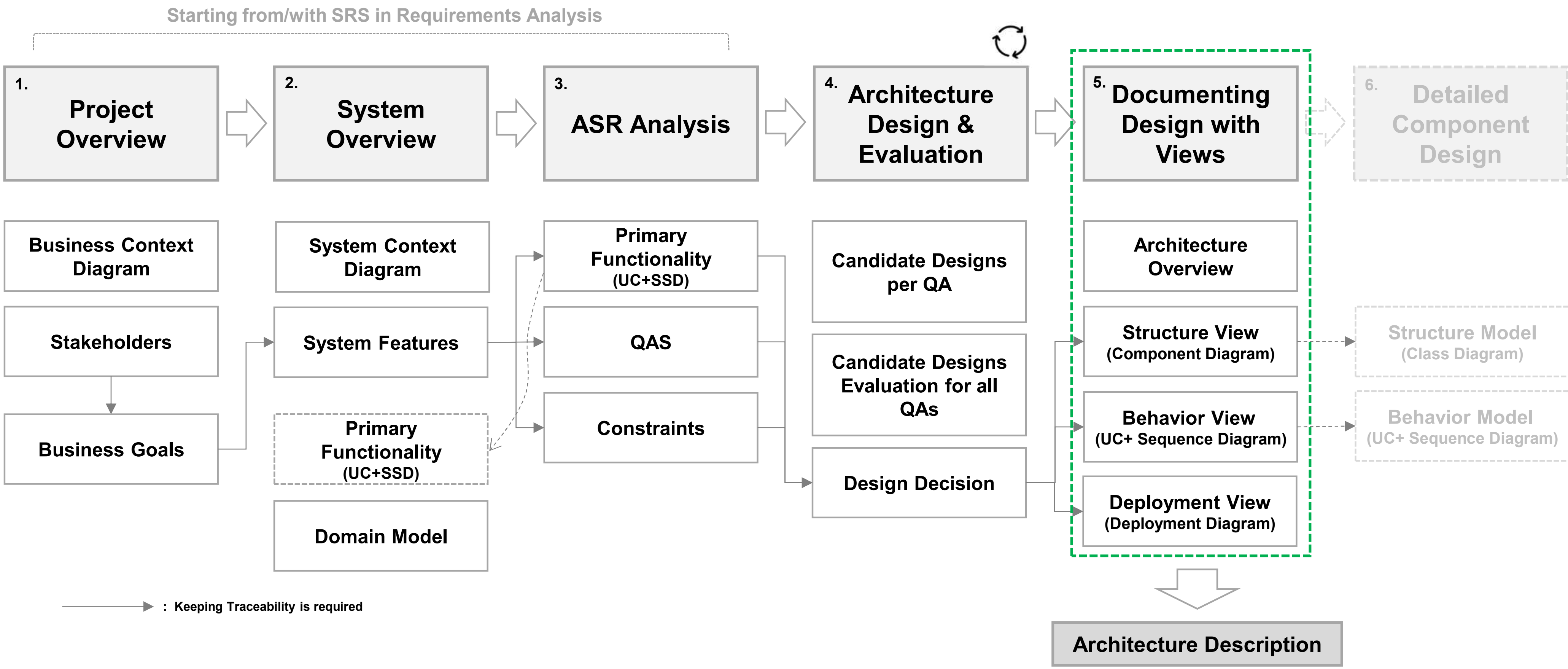
- **Details of the entire DD** will be explained through **three views** in Section 5.
  - **5.1 Architecture Overview** ← A UML Deployment version of the Domain Model
  - **5.2 Structure View**
  - **5.3 Behavior View**
  - **5.4 Deployment View**

Design Decision : An Example



5. Documenting Design with Views

5. Documenting Design with Views



Where We are Now in AD

1. Project Overview
    - 1.1 Project Background
    - 1.2 Business Context Diagram
    - 1.3 Stakeholders
    - 1.4 Business Goals

2. System Overview
    - 2.1 System Context Diagram
    - 2.2 External Entity
    - 2.3 External Interface
    - 2.4 System Features
    - 2.5 Domain Model
    - 2.6 Assumptions

3. Architectural Drivers
    - 3.1 Primary Functionality
    - 3.2 Quality Attribute Scenario
    - 3.3 Constraints

4. Architecture Design & Evaluation
    - 4.1 Candidate Designs per QA
    - 4.2 Candidate Designs Evaluation for all QAs
    - 4.3 Design Decision

5. Architecture Design Description
    - **5.1 Architecture Overview**
    - **5.2 Structure View**
    - **5.3 Behavior View**
    - **5.4 Deployment View**

6. Component Design Description
    - 6.1.2 Component Structure Model
    - 6.1.5 Component Behavior Model

7. Architecture Traceability Summary

Our Architecture Design Process (Revisited)

Architecture Design (High-Level)

System Level Design

Decomposition

Iteration

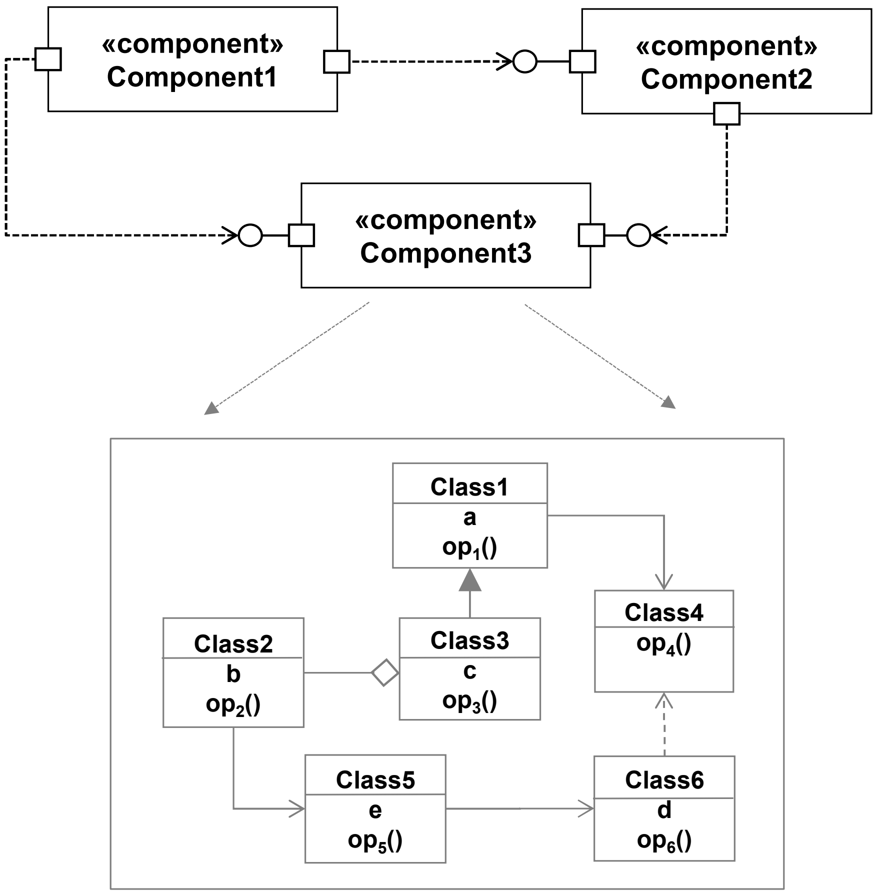
+ Design Concepts

Component Level Design

Detailed Design (Low-Level)

Specific Analysis

Detailed Design



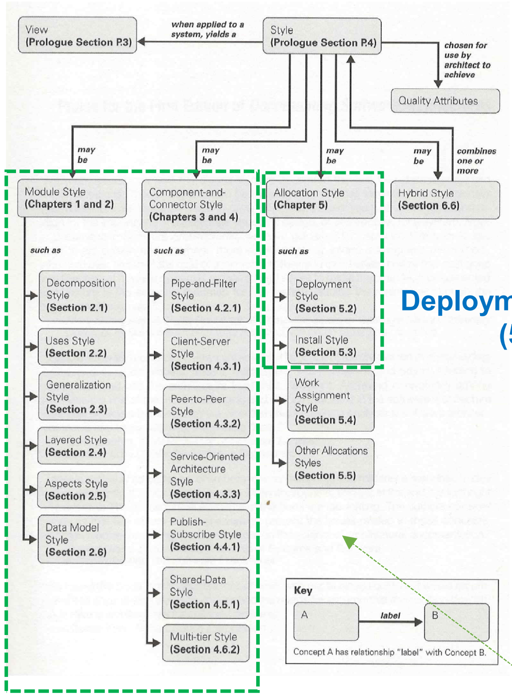
Architecture Design Description

- Architecture Overview (5.1)
- Structure View (5.2)
- Behavior View (5.3)
- Deployment View (5.4)

Detailed Component Design Description

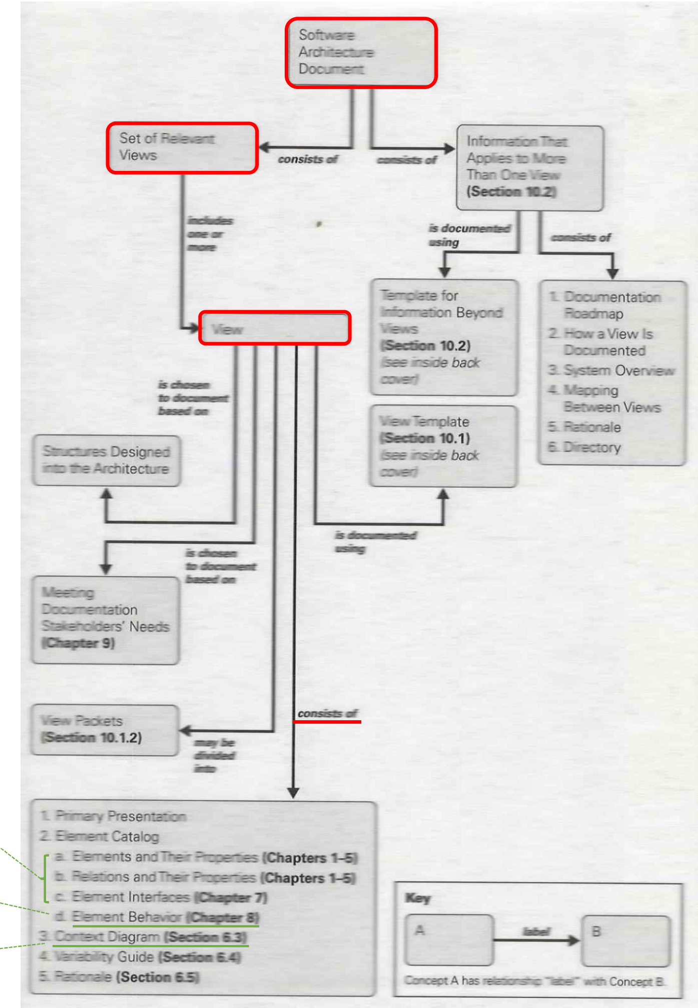
- Component Structure Diagram (6.1.2)
- Component Behavior Diagram (6.1.5)

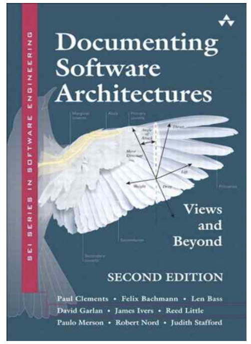
Architecture Design View Styles by CMU and Ours



Deployment Views (5.4)

Structure Views (5.2)





Behavior Views (5.3)

Domain Model (2.5 → 4.3)

Architecture Overview (5.1)

Our Architecture Design Views

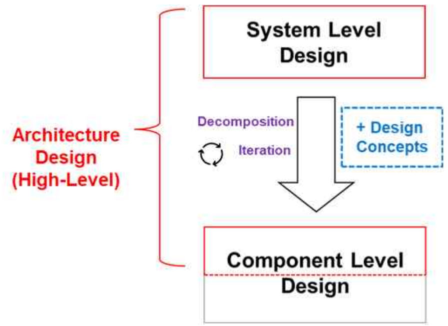
- **Architecture Overview (5.1)**
  - **Architecture Overview Diagram** : Sketching overall architecture design with UML Deployment(+Component) Diagram
    - An official version of **domain models** developed through sections 2.5 and 4.3

- **Structure View (5.2)**
  - **Static Structure Model** : Describing static structures with UML Component Diagram
  - Component Specification : Specifying all interfaces of components

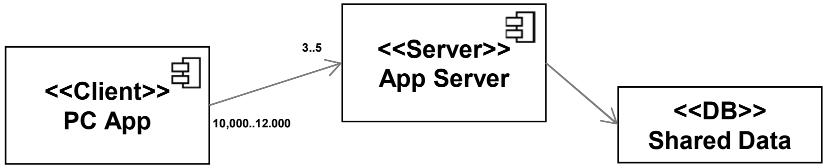
- **Behavior View (5.3)**
  - **Behavior Model** : Specifying the interactions of systems and components to satisfy the system's behavior with UML Sequence Diagram (+Use Case)

- **Deployment View (5.4)**
  - **Deployment Model** : Mapping software units to elements of an environment in which the software executes with UML Deployment Diagram

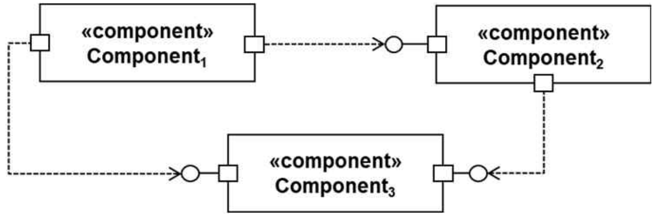
Documenting Architecture Design with Our Views



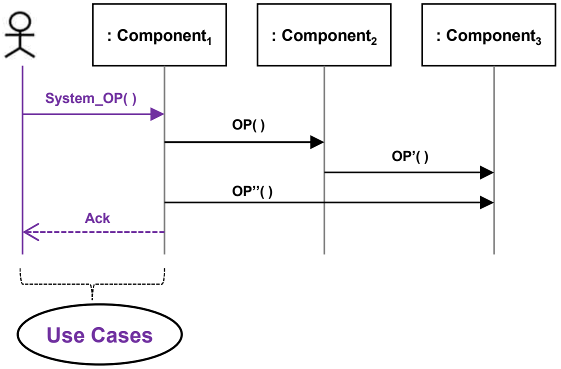
Architecture Overview (5.1)



Structure View (5.2)

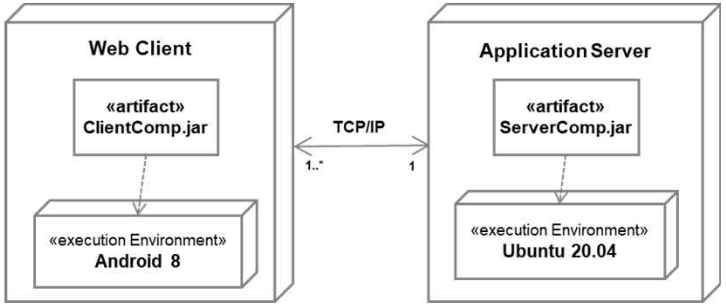


Behavior View (5.2)



form Primary Functionality (3.1)

Deployment View (5.2)



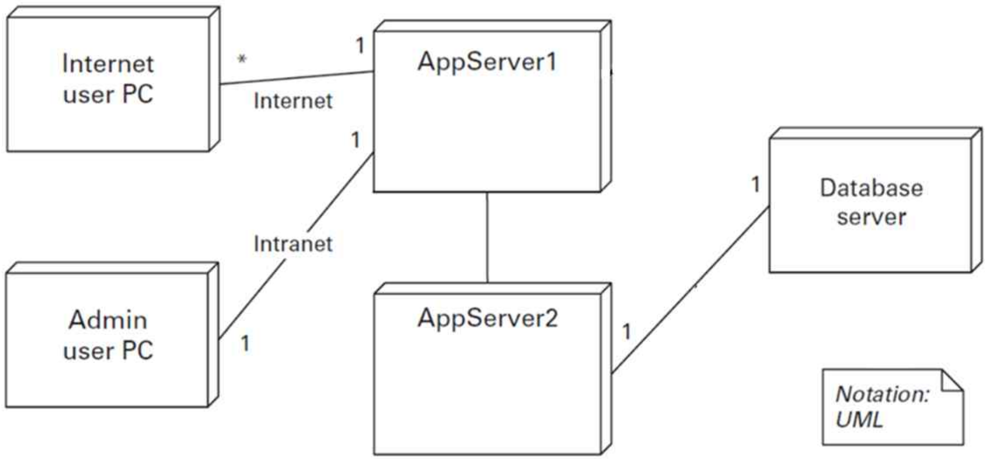
5.1 Architecture Overview

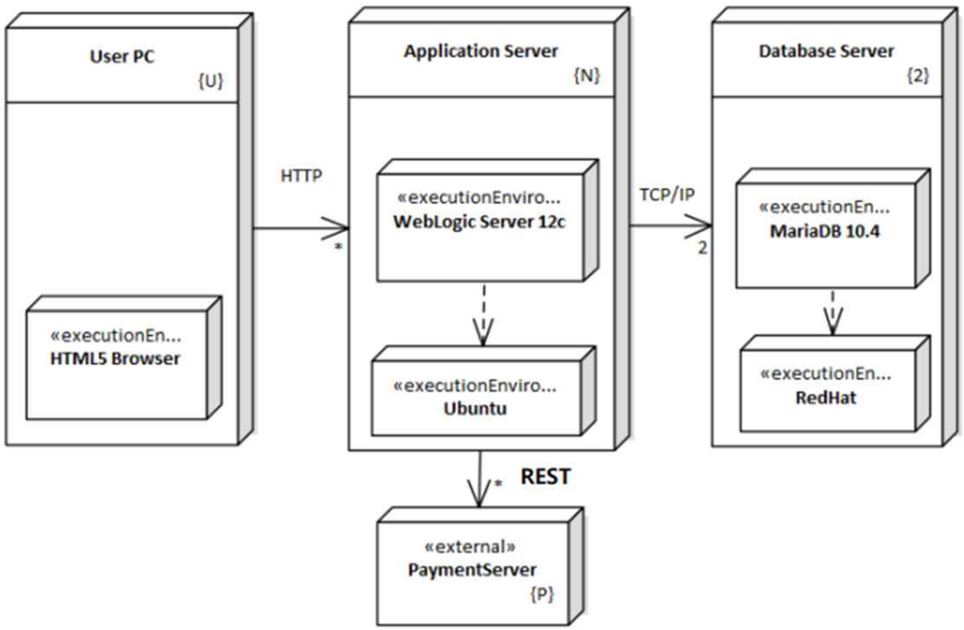
- An official version of **domain models** described **with UML**
  - Describing the overall architecture captured in the domain model (2.5) and the Design Decision (4.3) with the UML deployment diagram
  - Similar with “Infrastructure Diagram”
  - Detailed description will be specified with Deployment View (5.4).

- A high-level representation of system architecture
  - In case of large systems, describing the physical infrastructure in detail (Node, Execution Environment, Communication Path)

Node	Deployment target which represents computational resource upon which artifacts may be deployed for execution - Placement and scope of key system infrastructure elements (node, networks, sensors, workstations, etc.)
Execution Environment	A (software) node that offers an execution environment for specific types of artifacts - The choice of specific technology to implement the components
Communication Path	Association between two deployment targets, through which they can exchange signals and messages

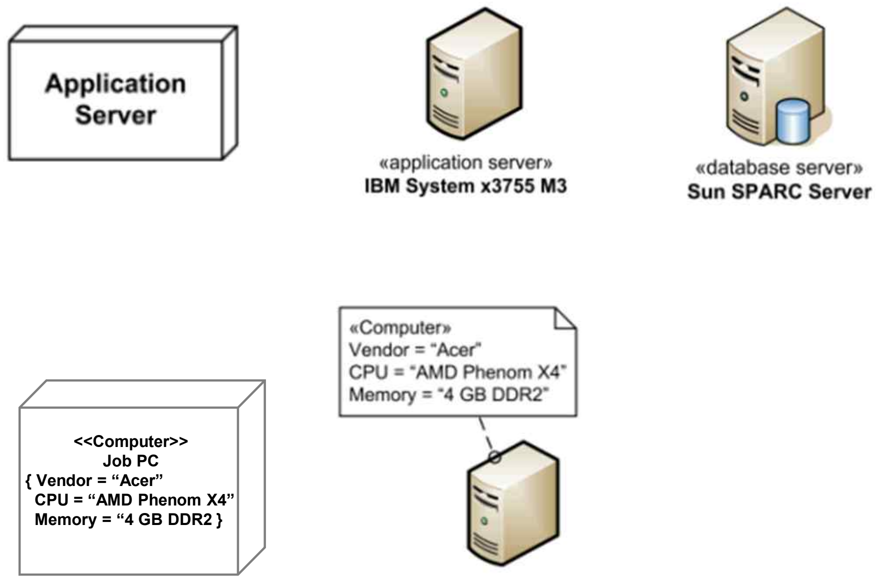
Architecture Overview Diagram





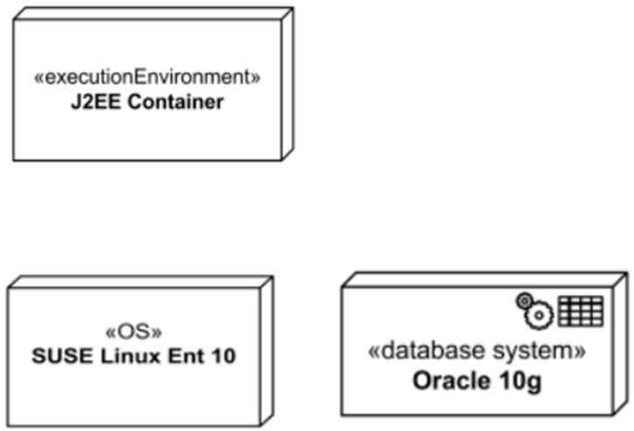
Architecture Overview Diagram : Node

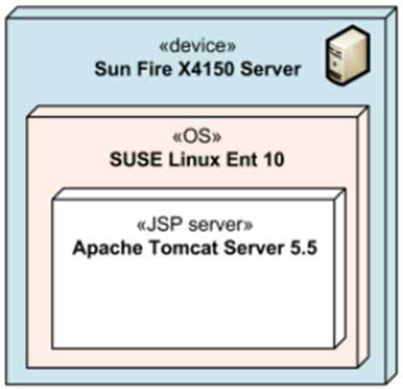
- **Node** is a <u>deployment target</u> which represents computing resource.
  - Examples of node stereotypes :
    - «application server», «client workstation», «mobile device», «embedded device»



Architecture Overview Diagram : Execution Environment

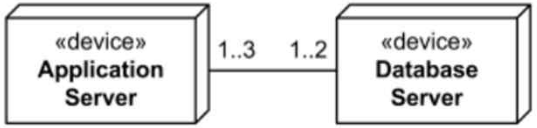
- A (software) node offers an **execution environment** for specific types of artifacts (executables)
  - Example stereotypes of execution environment :
    - «OS», «workflow engine», «database system», «J2EE container», «web server», «web browser», etc.



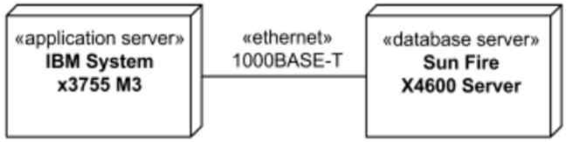


Architecture Overview Diagram : Communication Path

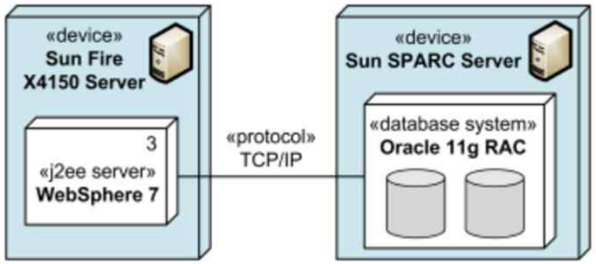
- An **association** between two deployment targets, through which they can exchange signals and messages
  - Communication path between several application servers and database server



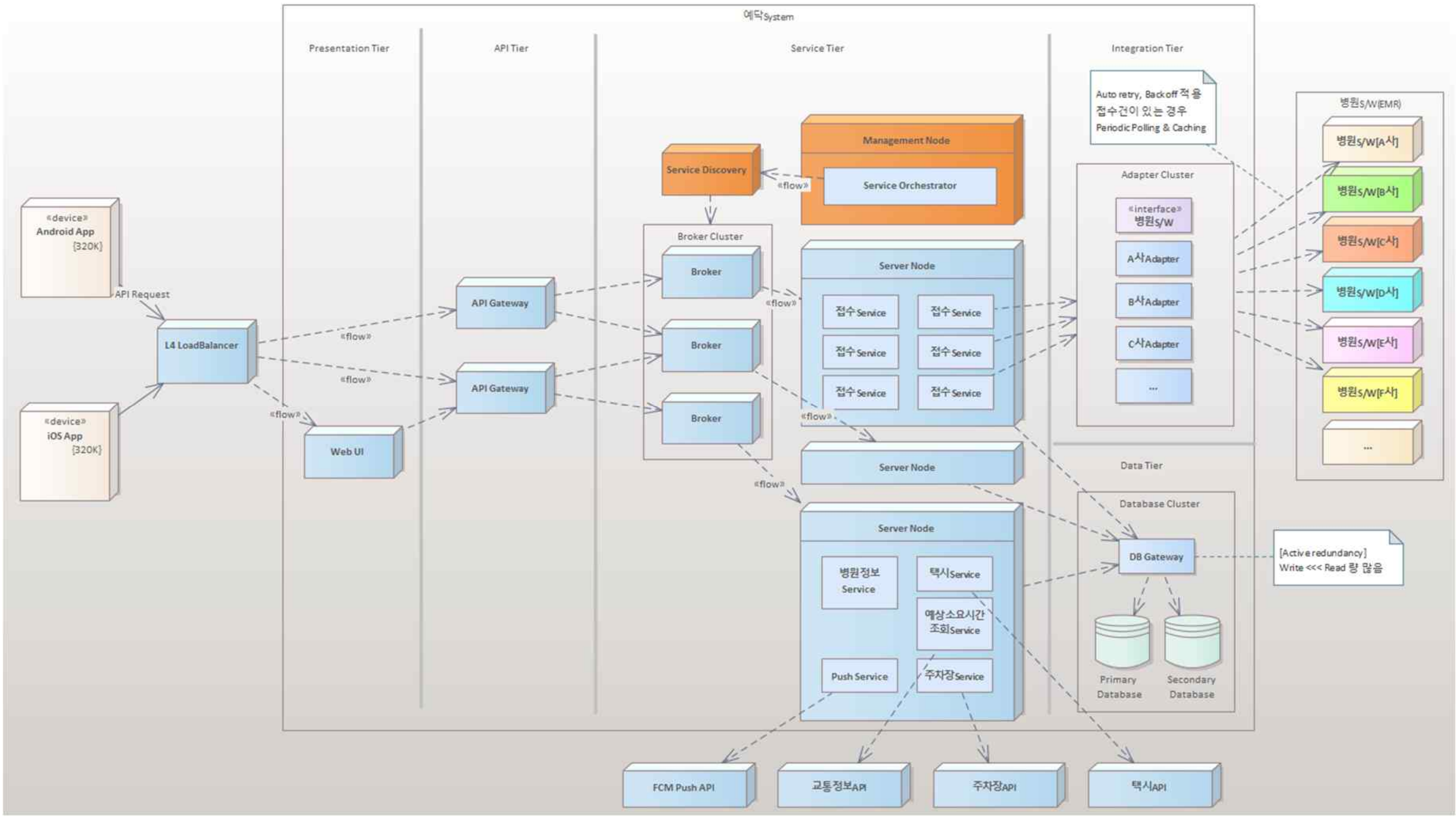
  - Gigabit Ethernet as communication path between application and database servers



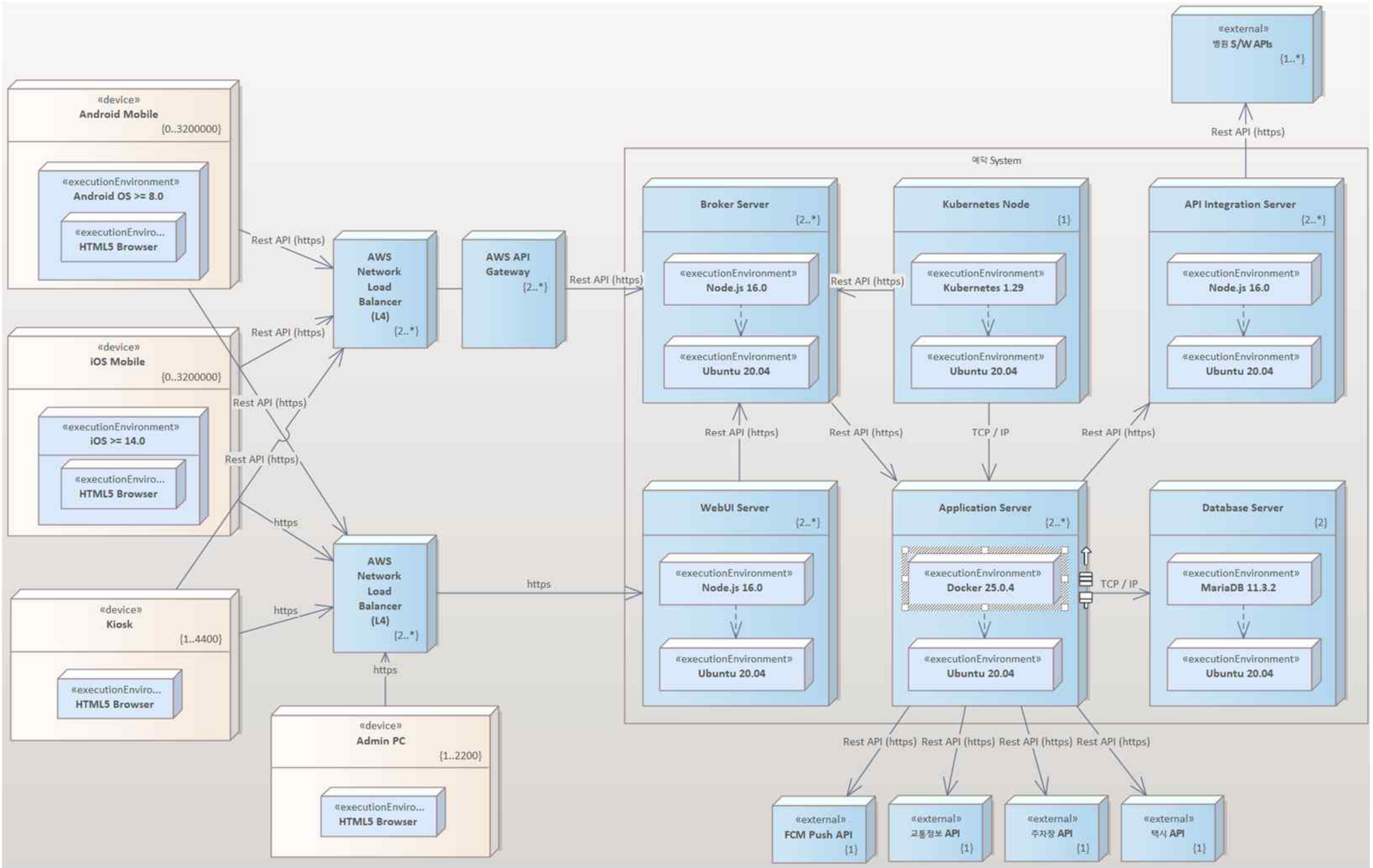
  - TCP/IP protocol as communication path between J2EE server and database system



Architecture Overview Diagram : An Example

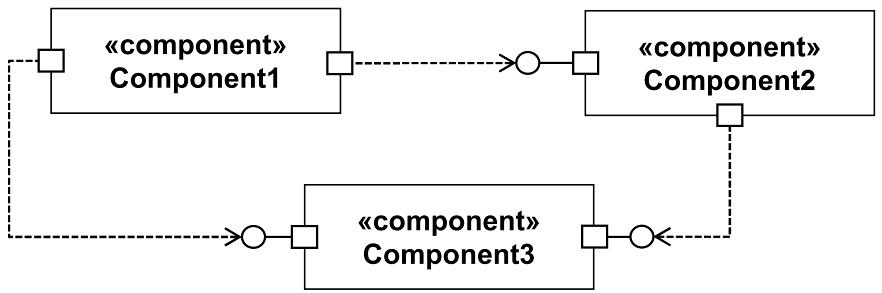


Architecture Overview Diagram : An Example

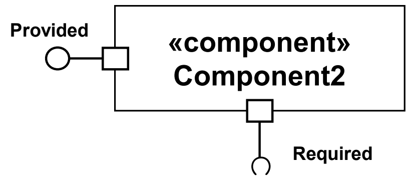


5.2 Structure View

- **Structure View**
  - **Static Structure Model** : Describing static structures with UML Component Diagram
  - **Component Specification** : Specifying all interfaces of components



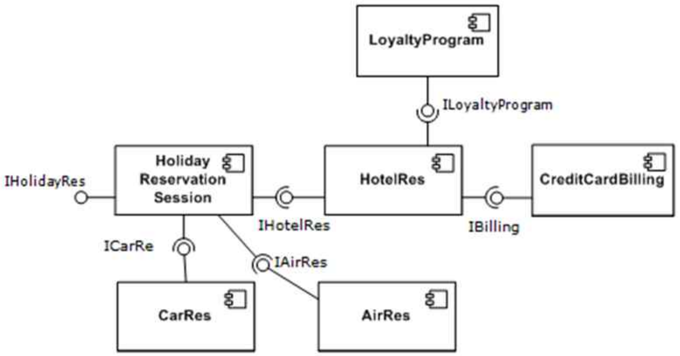
Static Structure Model (5.2.1)

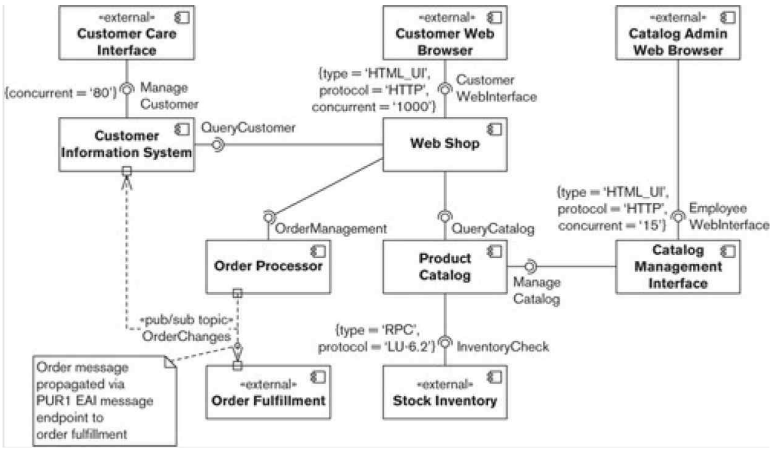


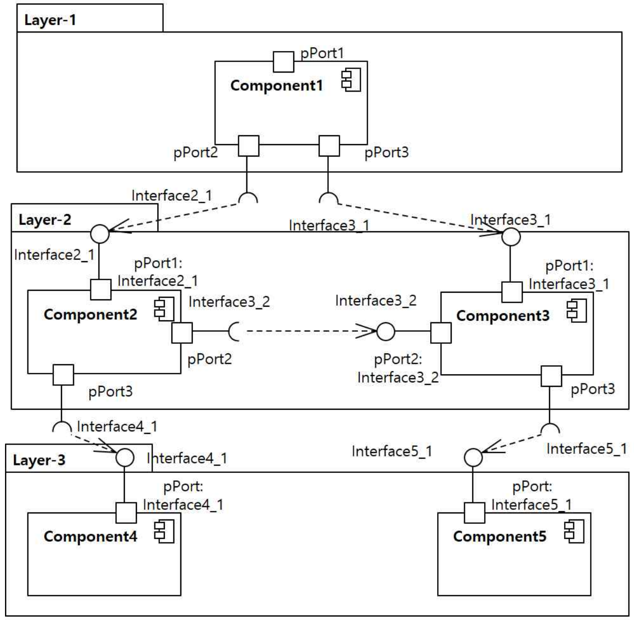
Component Specification (5.2.2)

Static Structure Diagram

- Describe **components** that implement the functionalities and QAs
  - Develop one static structure diagram for each node in the architecture overview diagram







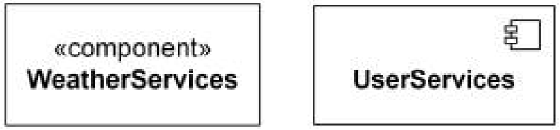
Static Structure Diagram – Element List

- Describe each element in the static structure diagram

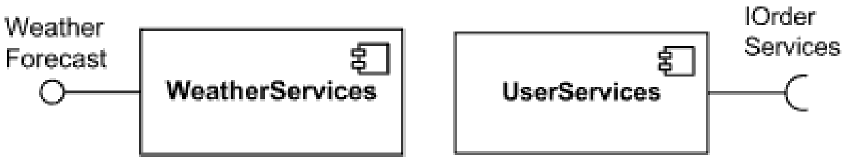
Element Name	Responsibility	Relevant ADs
Layer-1		
Layer-2		
Layer-3		
Component1		
…		
Component5		

Static Structure Diagram – Component

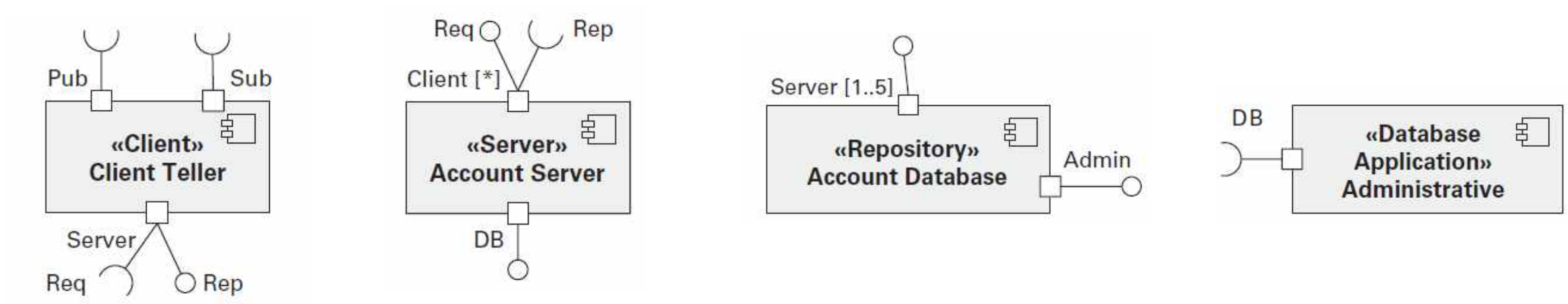
- A **component** is a well-defined functional part of the system which
  - Has particular responsibilities and



  - Exposes well-defined interfaces(Provided/Required) that allow it to be connected to other elements.



- Stereotypes are used to denote the type of the view-specific component.

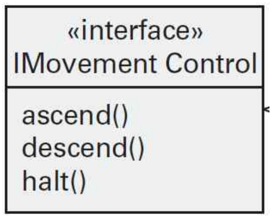


Static Structure Diagram – Interface

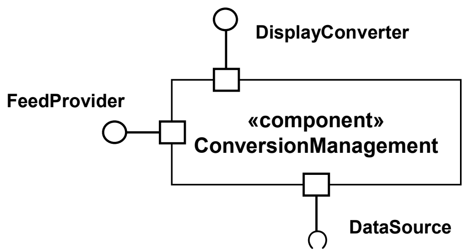
- An **interface** is a well-defined mechanism by which the functions of an element can be accessed by other elements.
  - **Provided interfaces**
    - Interface that the component realizes (provided services)
    - Other components and classes interact with a component through its provided interfaces.
  - **Required interfaces**
    - Interface that the component needs to function (expected services)
    - The component needs another class or component that realizes that interface to function.

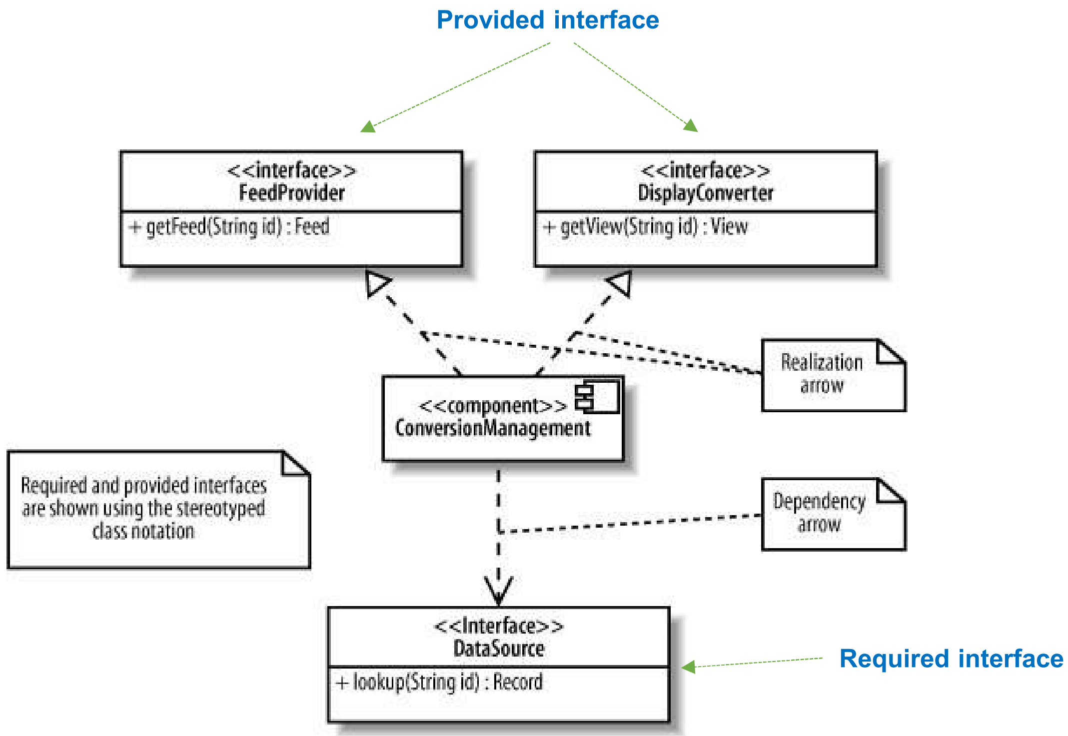
Static Structure Diagram – Interface

- An interface is defined by the inputs, outputs, and semantics of each operation offered, and the nature of the interaction needed to invoke the operation.



- Stereotype notation for interfaces :

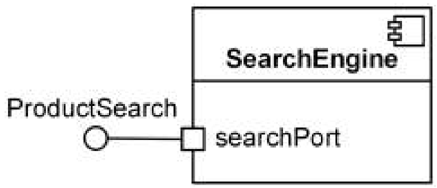


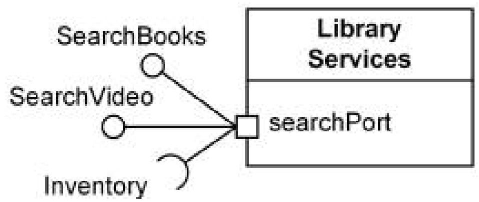


Static Structure Diagram – Port

- **Ports** represent interaction points through which a component communicates with other components and its environment.

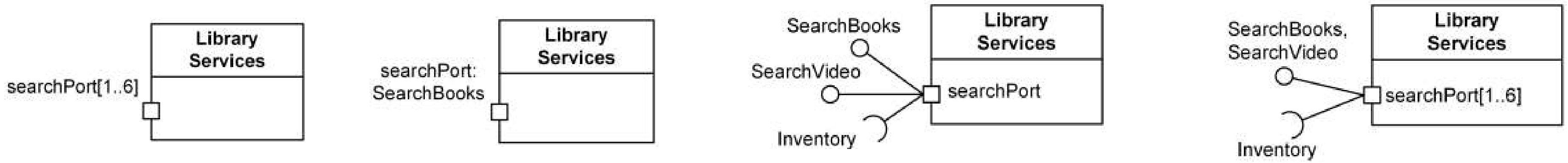
- Component interactions take a variety of forms :
  - Function or method calls
  - Remote procedure calls
  - Web service requests
  - Data streams, shared memory, and message passing



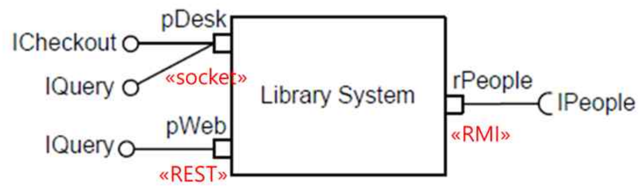


Static Structure Diagram – Port

- Various notations for ports

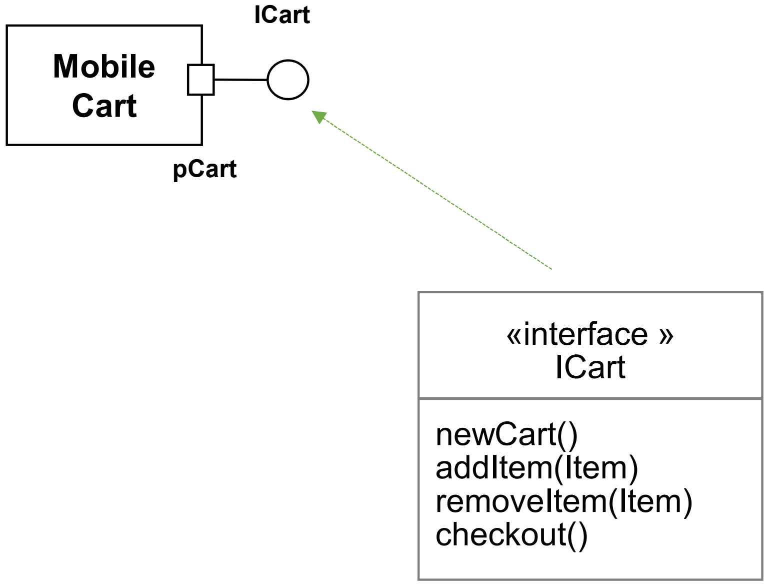


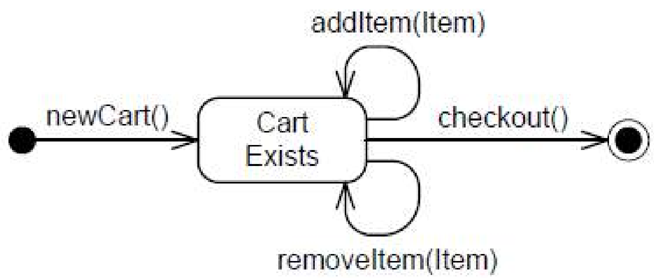
- Multiple ports with stereotypes are used for
  - Cohesive set of interfaces
  - Communication protocols
  - Reduced coupling



Static Structure Diagram – Port

- The port behavior can be specified with the UML State (Statechart) Diagram.

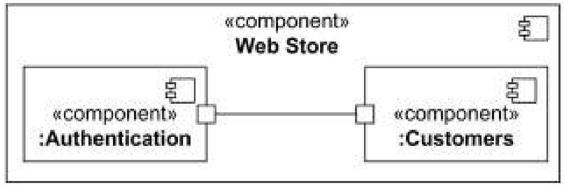


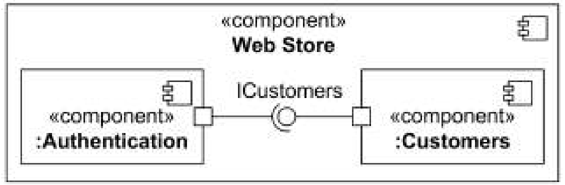


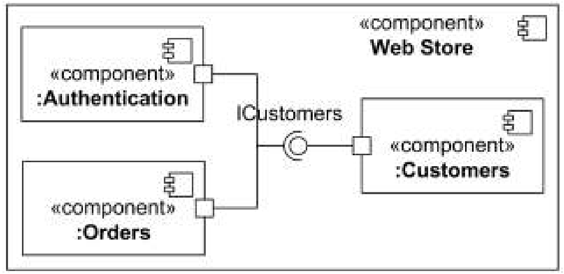
UML Statechart Diagram

The UML Composite Structure Diagram - Connectors

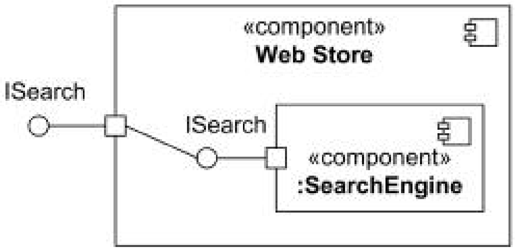
- **Assembly connector** defines that one component provides the services that another component requires.
  - It must only be defined from a required interface to a provided interface.
  - An assembly connector is notated by a “ball-and-socket” connection.

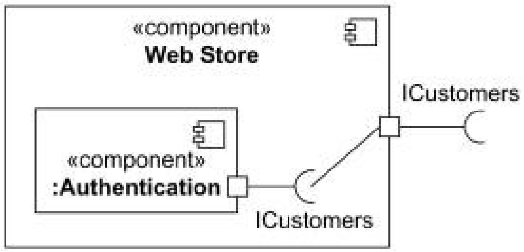






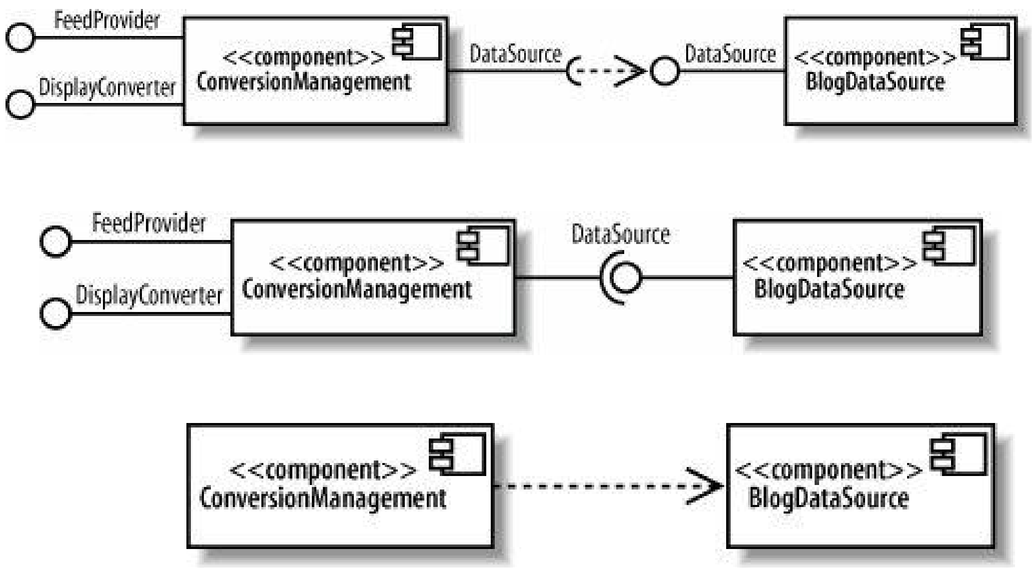
- **Delegation connector** links the external contract of a component to the internal realization.
  - Represents the forwarding of signals
  - It must only be defined between used interfaces or ports <u>of the same kind</u>.



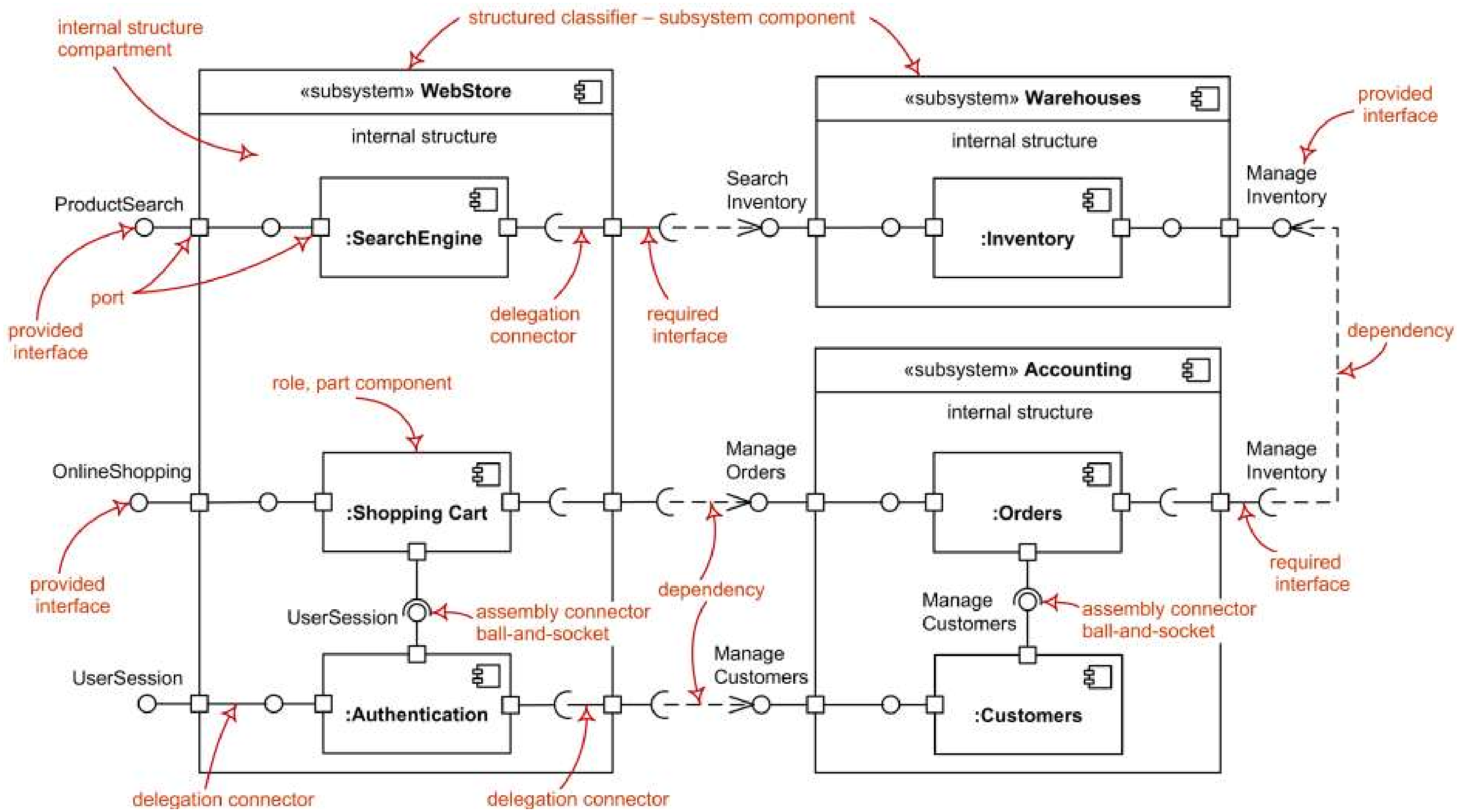


Static Structure Diagram – Components Working Together

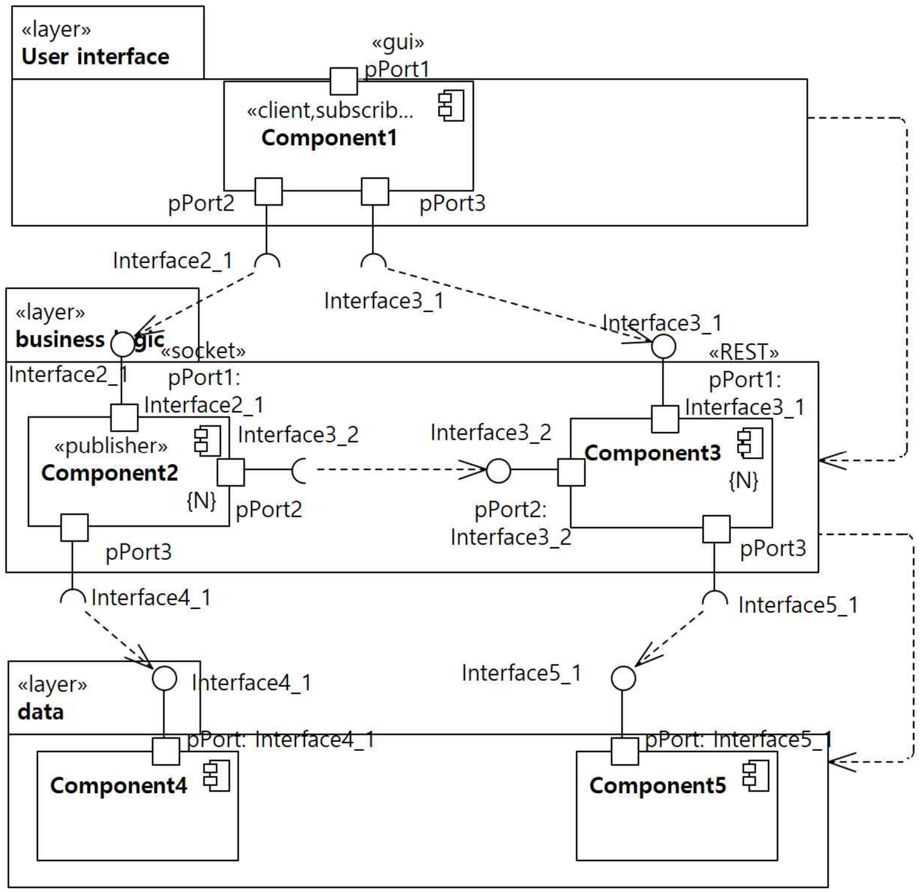
- If a component has a required interface, then it needs another class or component in the system that provides it.
  - At a higher-level view, this is a dependency relation between the components.

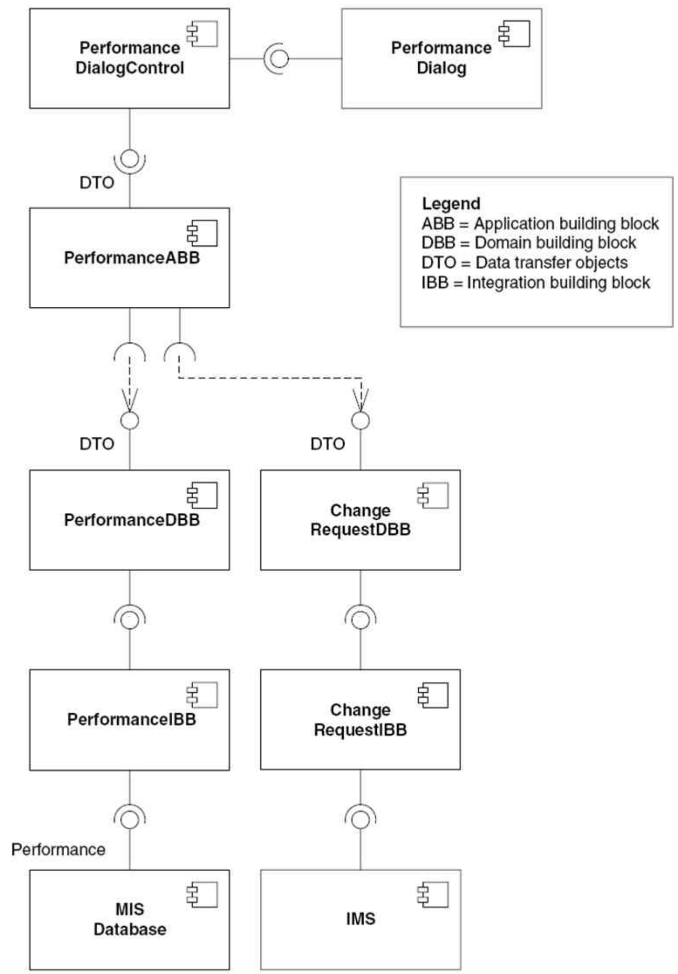


Static Structure Diagram : Examples

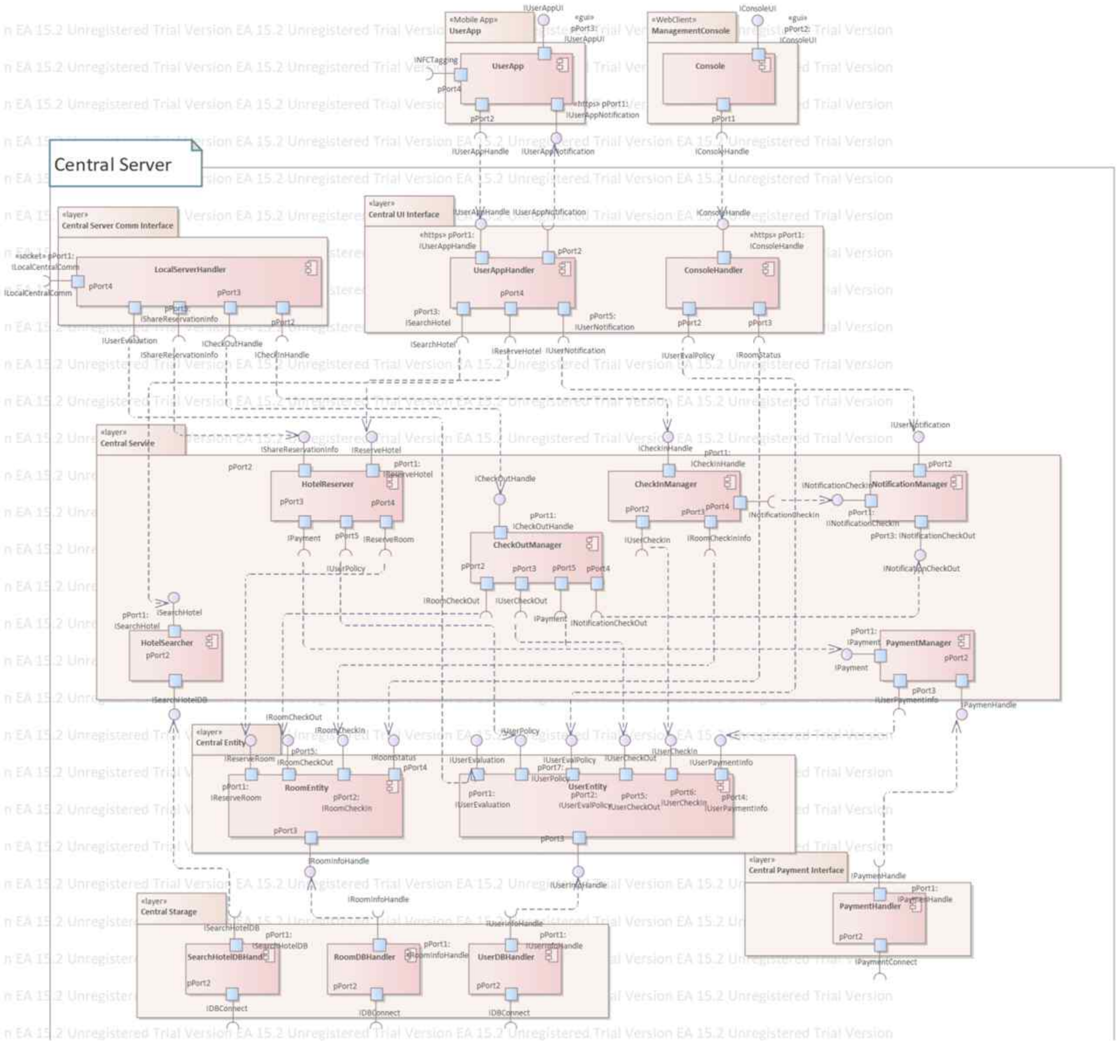


Static Structure Diagram : Examples



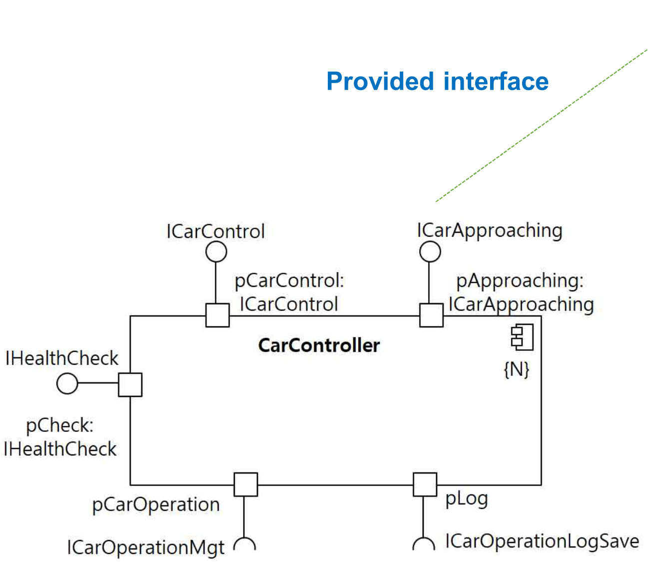


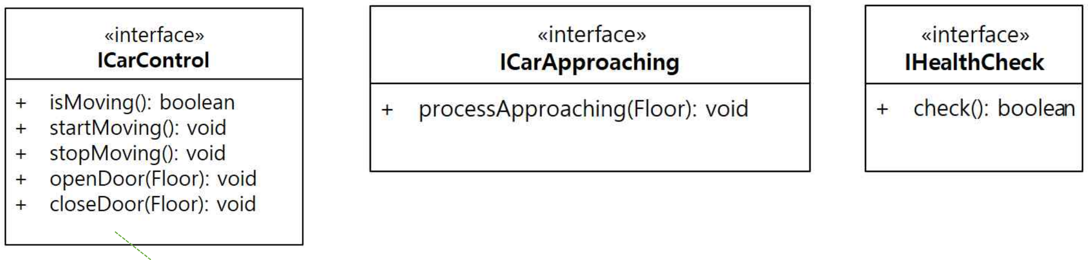
Static Structure Diagram : Examples



Component Specification

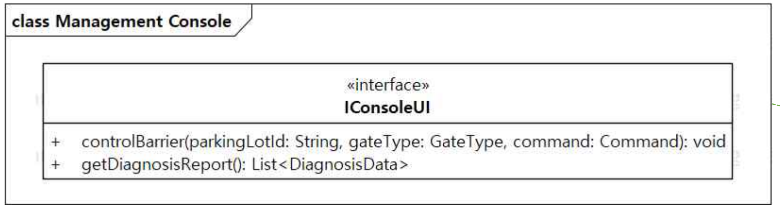
- Specifies **all provided interfaces** of components and **all evident operations** for each interface
  - The required interfaces are specified by other providing components.



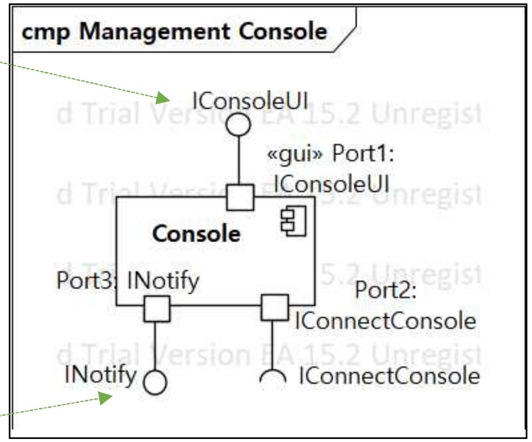


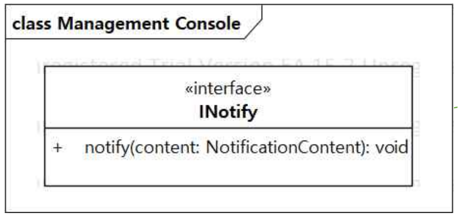
Operation	Responsibility
isMoving( )	…
startMoving( )	
stopMoving( )	
openDoor( )	
closeDoor( )	

Component Specification : An Example



Operation	Responsibility
controlBarrier()	Console에서 차단기 원격 제어를 요청하는 operation
getDiagnosisReport()	Console에서 장비 동작 상태 (진단 결과 요약 정보)를 요청하는 operation





Interface Name	Kind	Responsibility
IConsoleUI	Provided	Console 화면의 UI를 담당하는 interface
INotify	Provided	Console로 장비 고장 알림을 전달하는 interface
IConnectConsole	Required	Console과 Main Server 사이의 통신을 담당하는 interface로서, Console로 받은 요청을 Main Server로 전달

Operation	Responsibility
Notify()	Console UI로 알림 내용을 출력하는 operation

Component Design Principles

- **Component Design Principles**
  - **Cohesion**
    - To what extent are the functions provided by an element strongly related to each other?
  - **Coupling**
    - How strong are the element interrelationships? To what extent do changes in one element affect others?
  - **Extensibility**
    - Will the architecture be easy to extend to allow the system to perform new functions in the future?
  - **Functional Flexibility**
    - How amenable is the system to supporting changes to the functions already provided?
  - **Separation of Concerns**
    - To what extent is common processing performed in only one place?
  - **Consistency**
    - Are mechanisms and design decisions applied consistently throughout the architecture?

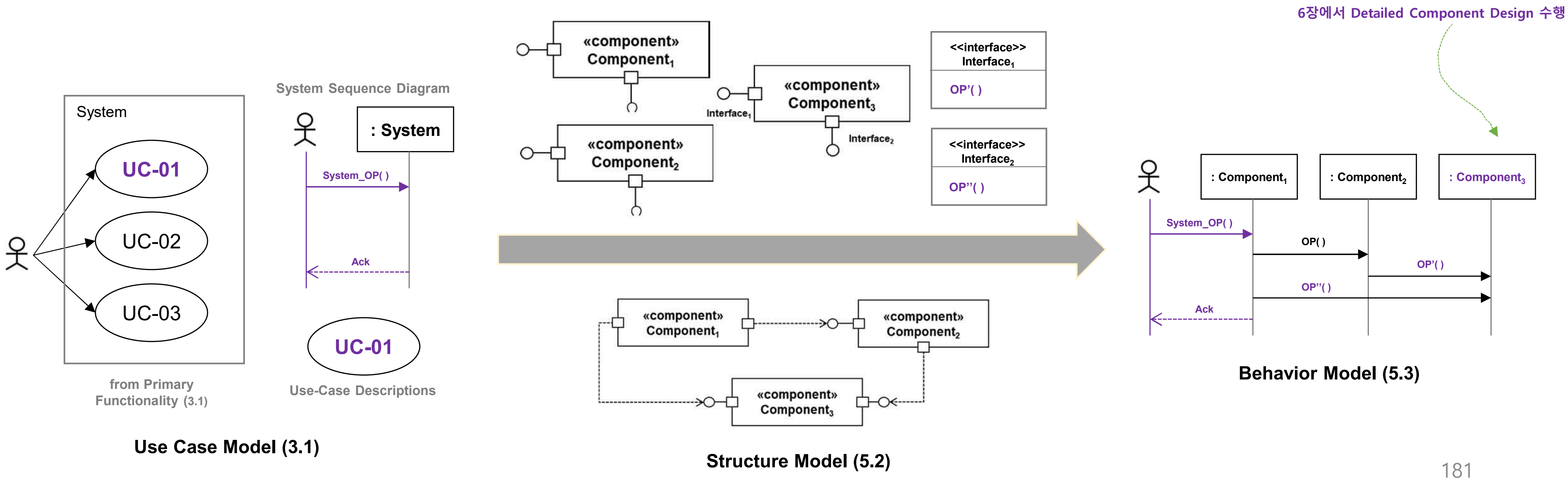
조별 Design Principles 발표 #5

Component Interface Design Principles

- **Component Interface Design Principles**
  - **Separate Interface**
    - ISP(Interface Segregation Principle)
  - **Use Abstract Name**
    - Use outcome-revealing name
    - Use implementation-free name
  - **Make Interface Abstract**
    - Data Abstraction: introduce parameter object, preserve whole object, introduce abstract data type
    - Functional Abstraction: introduce facade function
    - Implementation abstraction: encapsulate collection, replace parameter with method, replace parameter with explicit method, parameterize method
  - **Minimize Dependency**
    - DIP(Dependency Inversion Principle)
    - Law of Demeter; Hide delegation

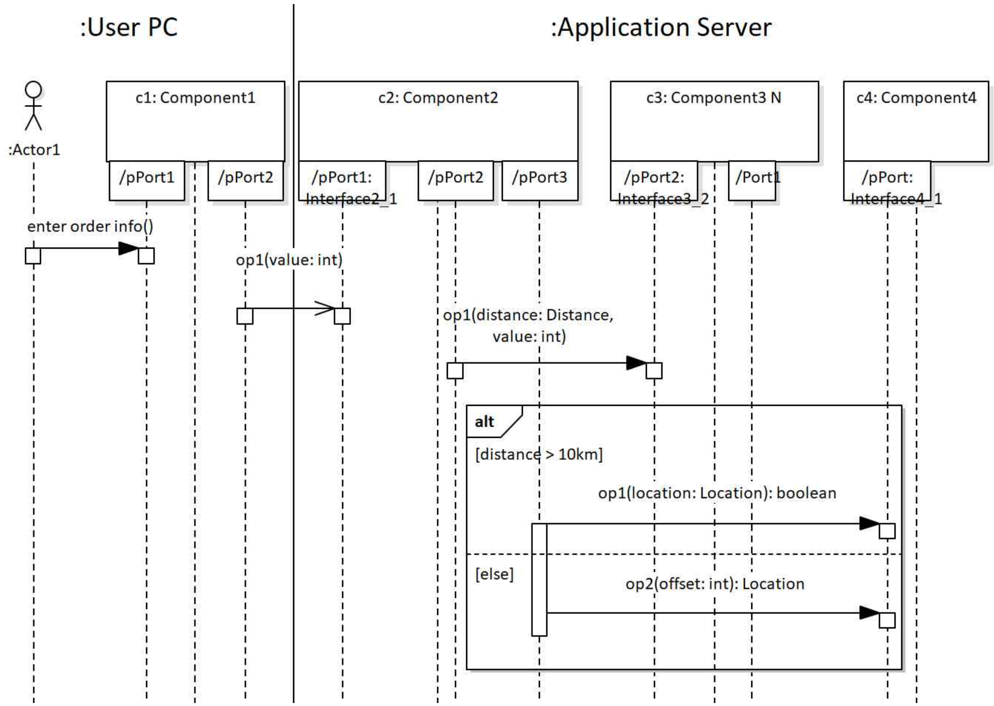
5.3 Behavior View

- **Behavior View**
  - **Behavior Model** : Specifying the interactions of systems and components to satisfy the system's behavior
  - For each use case (3.1) marked as ASR, analyze interactions among system components through the UML **Sequence Diagram**.
    - Starting from the SSD and its system operations/interfaces

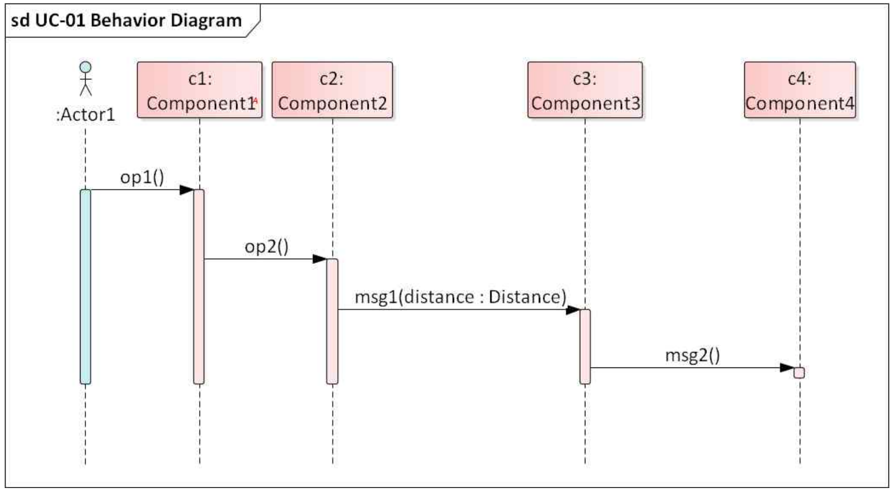


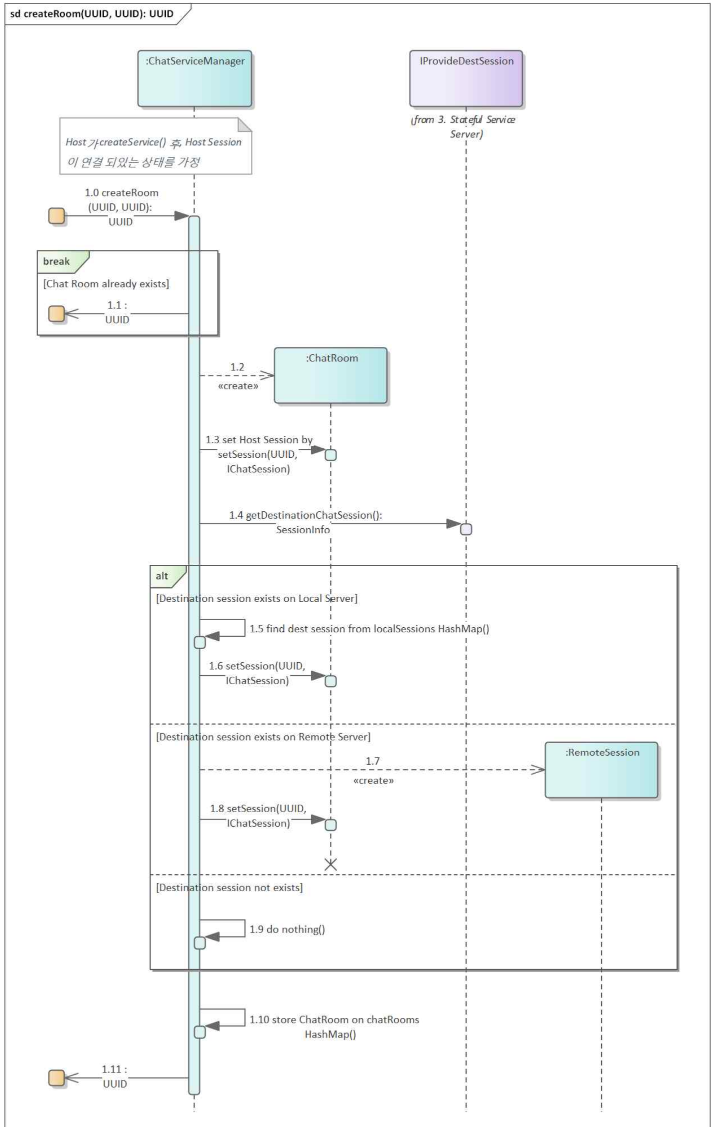
Behavior Diagram

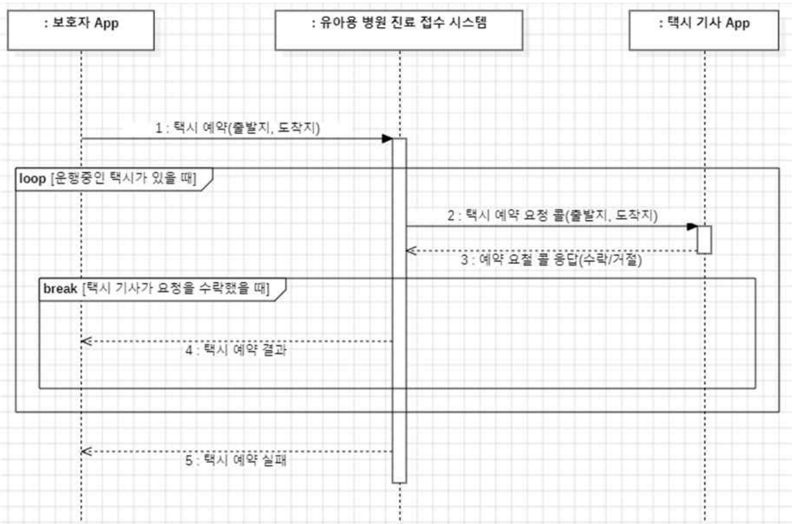
- **The UML Sequence Diagram**
  - Describing <u>interactions</u> among component instances of <u>static structure model</u> through <u>**ports**</u>
    - But ports can be omitted if they seem irrelevant or not important.
  - Should <u>correspond exactly</u> to the use cases and system sequence diagrams from (3.1)



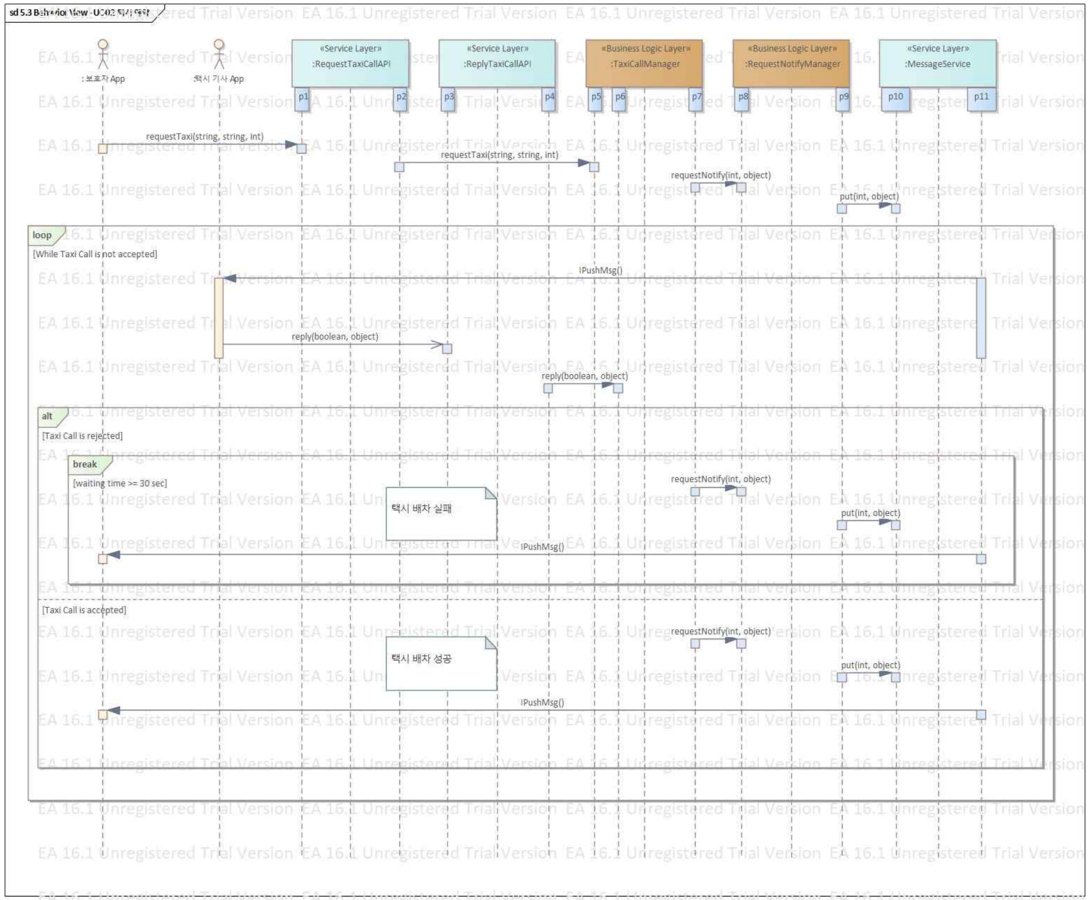
Behavior Diagram : Examples





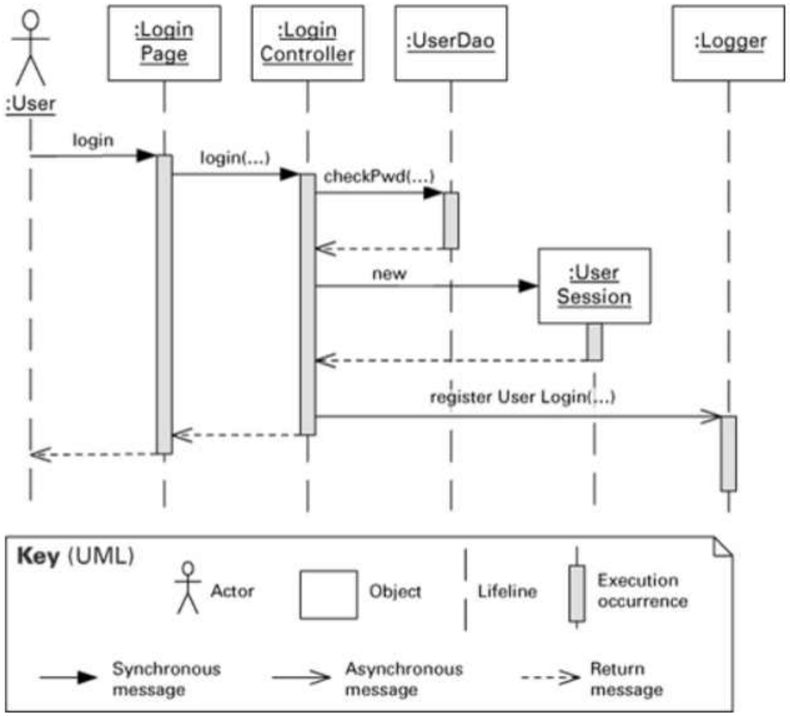


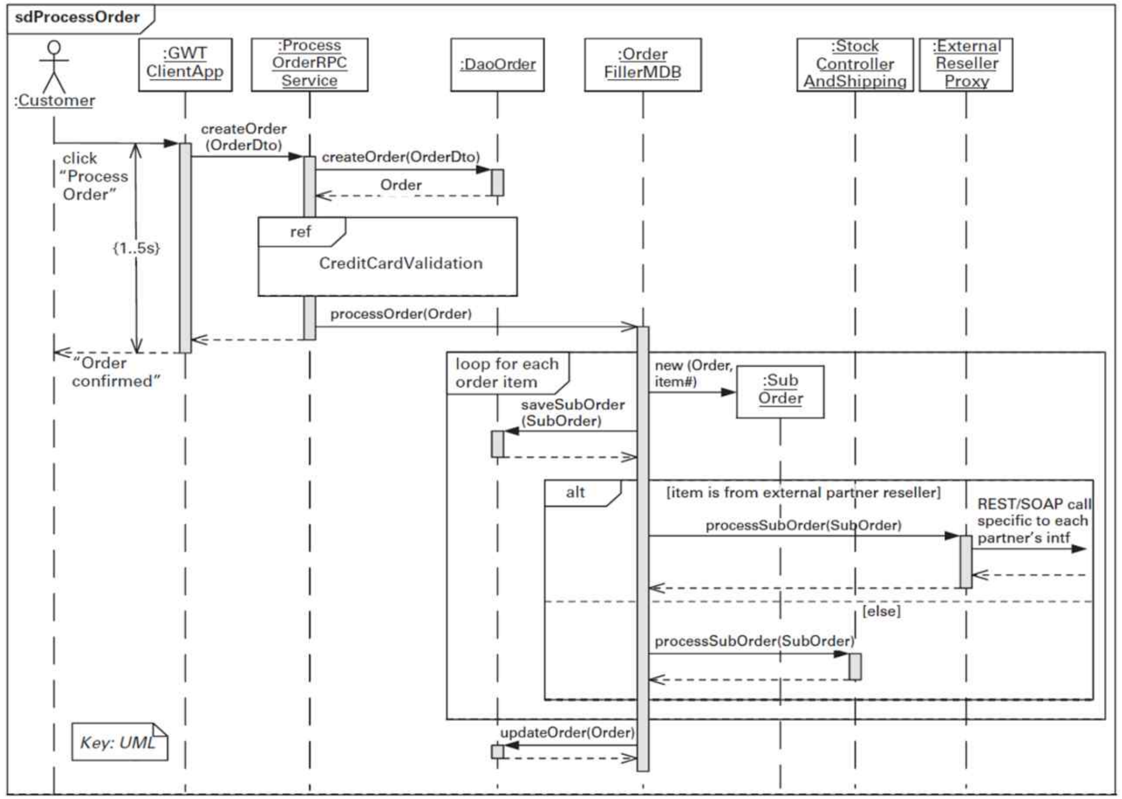
: 보호자 App
: 유아용 병원 진료 접수 시스템
: 택시 기사 App
1 : 택시 예약(출발지, 도착지)
loop [운행중인 택시가 있을 때]
2 : 택시 예약 요청 콜(출발지, 도착지)
3 : 예약 요청 콜 응답(수락/거절)
break [택시 기사가 요청을 수락했을 때]
4 : 택시 예약 결과
5 : 택시 예약 실패



: 보호자 App
택시 기사 App
«Service Layer»
:RequestTaxiCallAPI
«Service Layer»
:ReplyTaxiCallAPI
«Business Logic Layer»
:TaxiCallManager
«Business Logic Layer»
:RequestNotifyManager
«Service Layer»
:MessageService
p1
p2
p3
p4
p5
p6
p7
p8
p9
p10
p11
requestTaxi(string, string, int)
requestTaxi(string, string, int)
requestNotify(int, object)
put(int, object)
loop
[While Taxi Call is not accepted]
IPushMsg()
reply(boolean, object)
reply(boolean, object)
alt
[Taxi Call is rejected]
break
[waiting time >= 30 sec]
택시 배차 실패
requestNotify(int, object)
put(int, object)
IPushMsg()
[Taxi Call is accepted]
택시 배차 성공
requestNotify(int, object)
put(int, object)
IPushMsg()
EA 16.1 Unregistered Trial Version

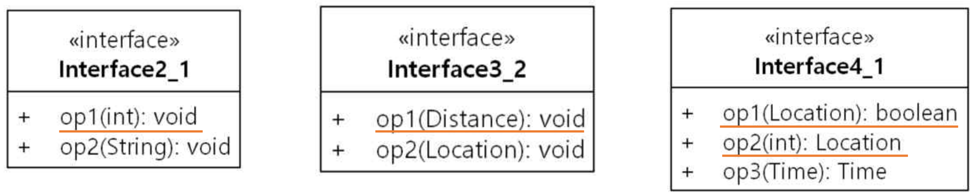
Behavior Diagram : Examples

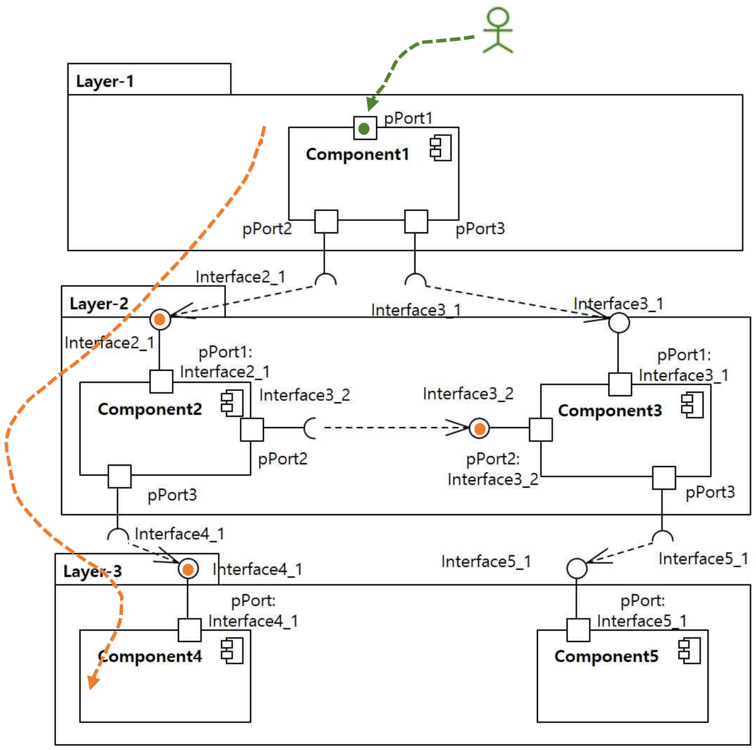




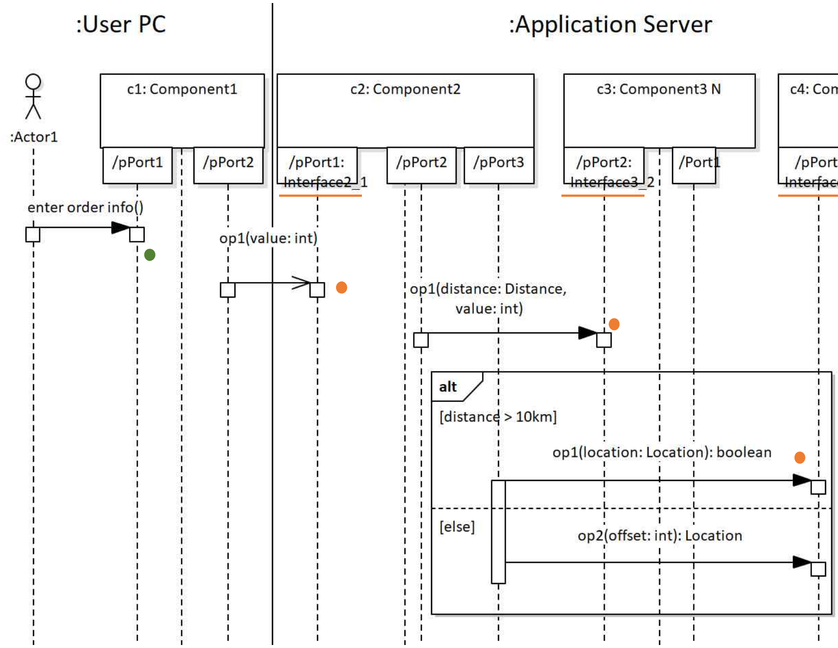
Consistency between Structure and Behavior Views

Structure View



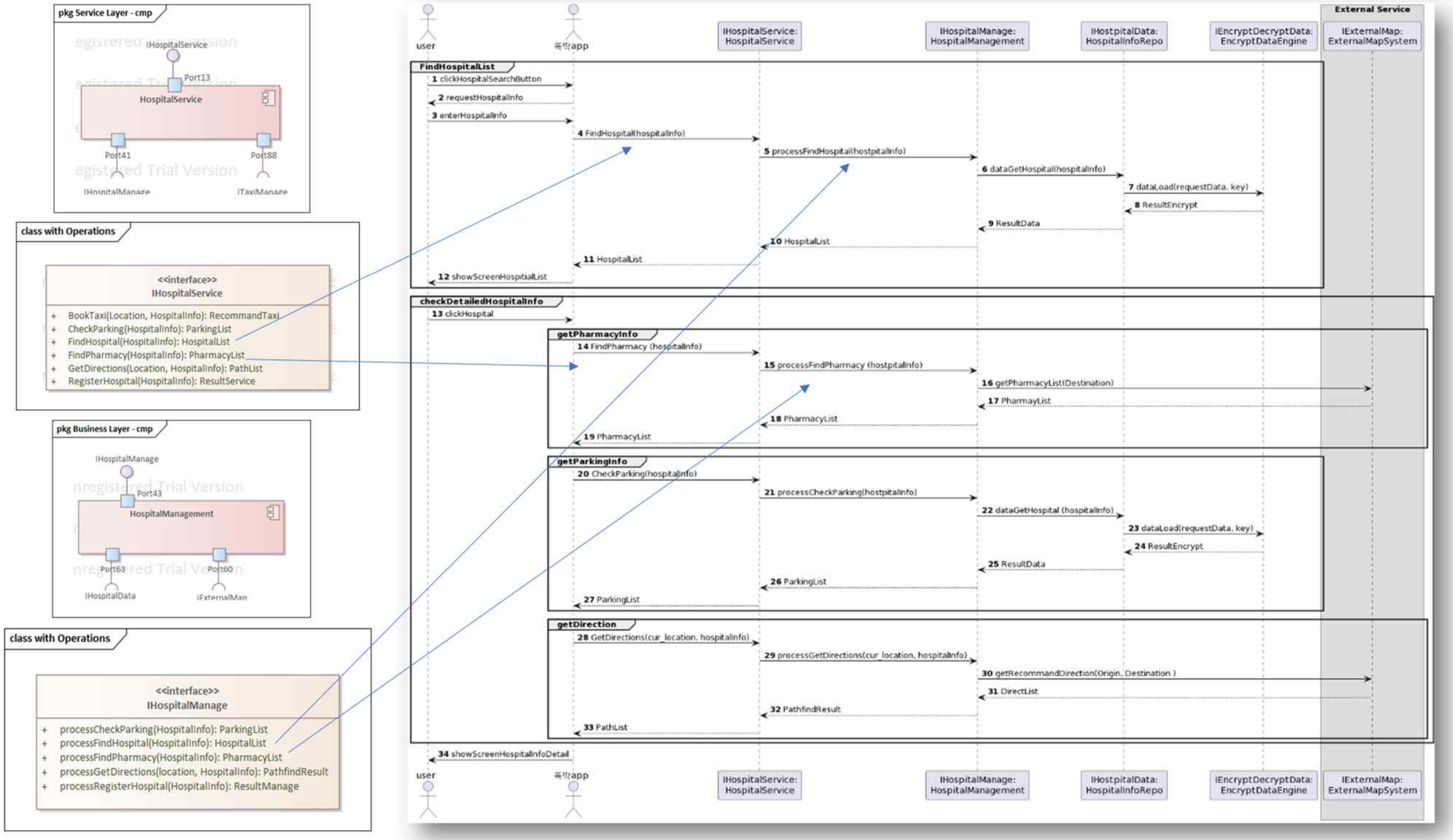


Structure View



Behavior View

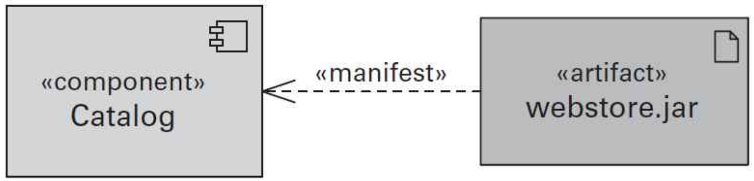
Consistency between Structure and Behavior Views : An Example



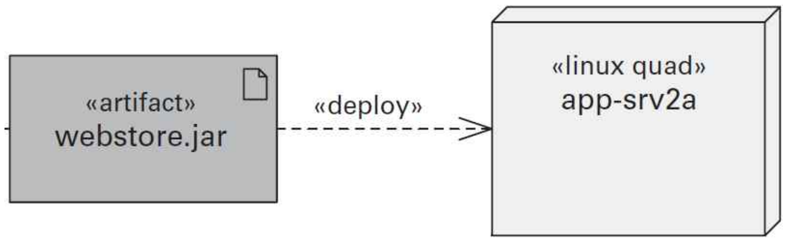
5.4 Deployment View

- One or more components are **manifested** by an artifact, and then the artifacts are **deployed** to its execution environment.

  - **Artifact Definition Model**

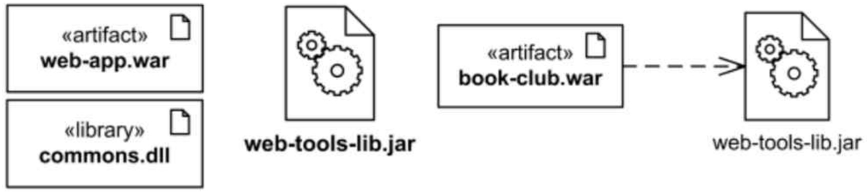


  - **Artifact Deployment Model**

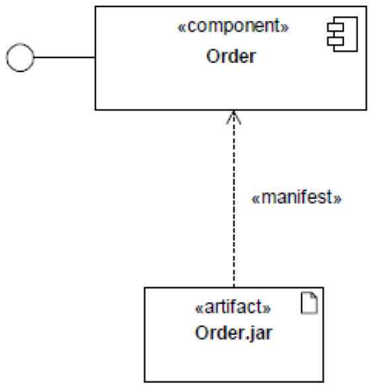


Artifact Definition Model

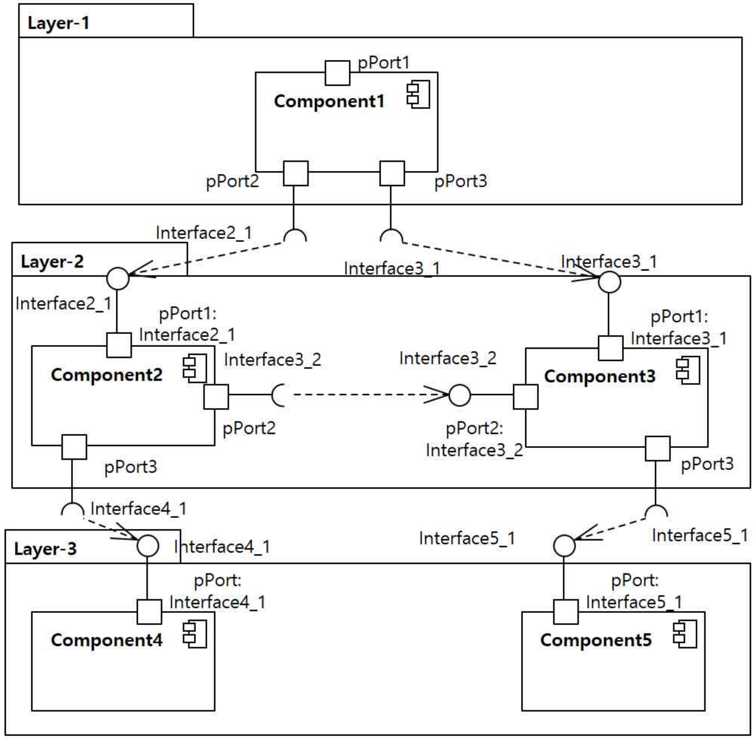
- **Artifact**
  - Physical packaging of components
  - A physical implementation unit of components



- **Artifact Definition Model** describes how the physical artifacts maps to logical components.
  - **<<manifest>>** relationship between an **Artifact** and a **Component**

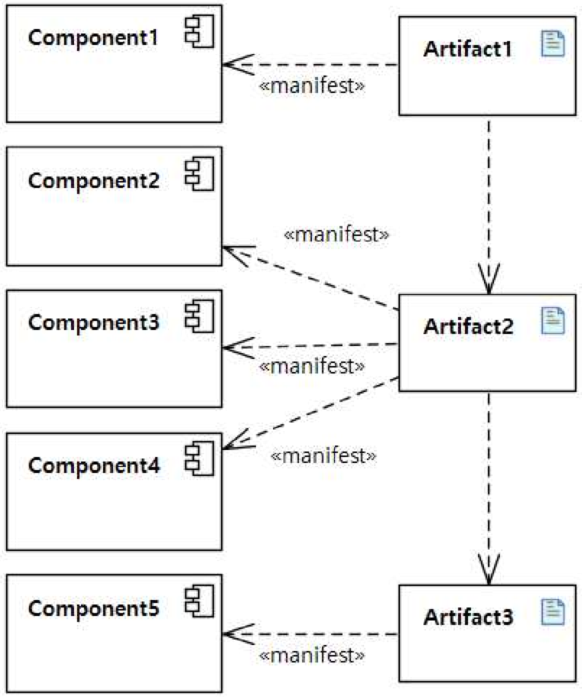


Artifact Definition Model vs. Static Structure Model



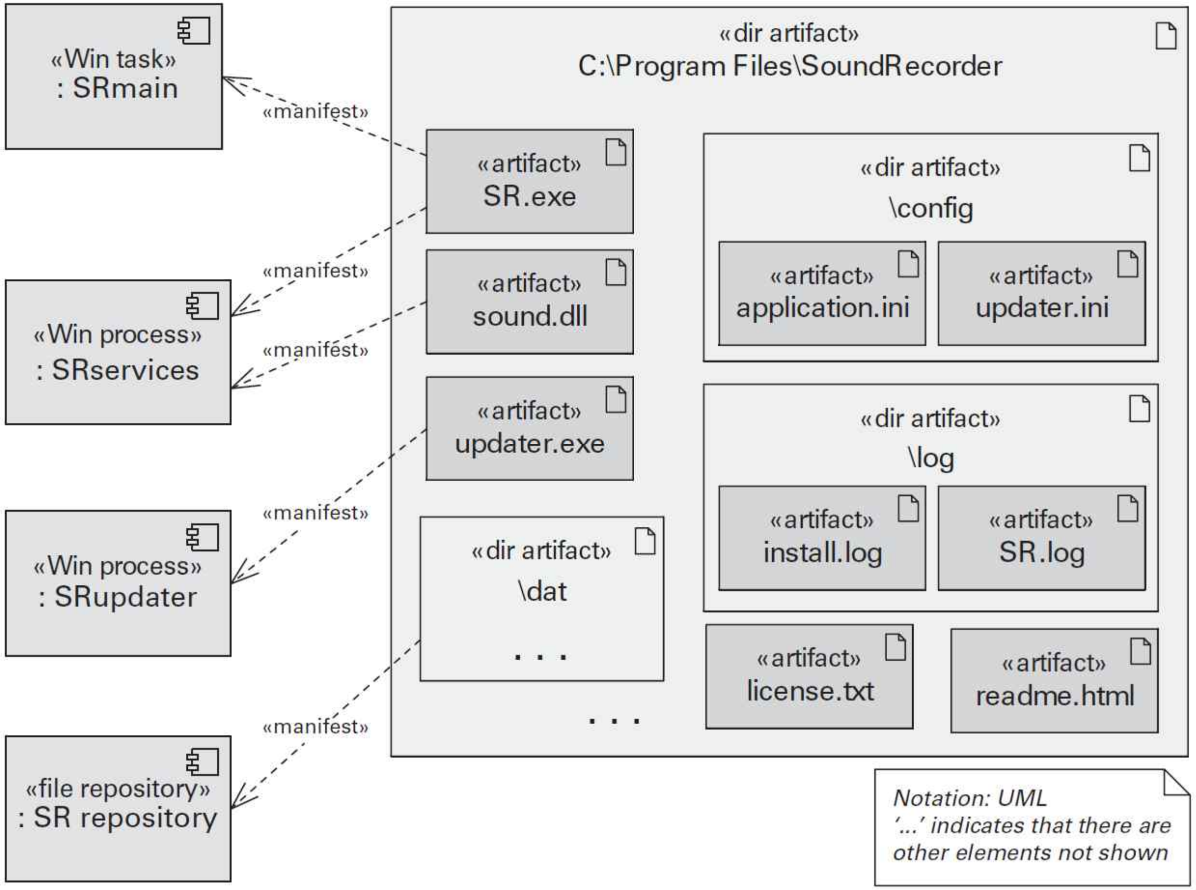
Static Structure Model
(Structure View)

vs.

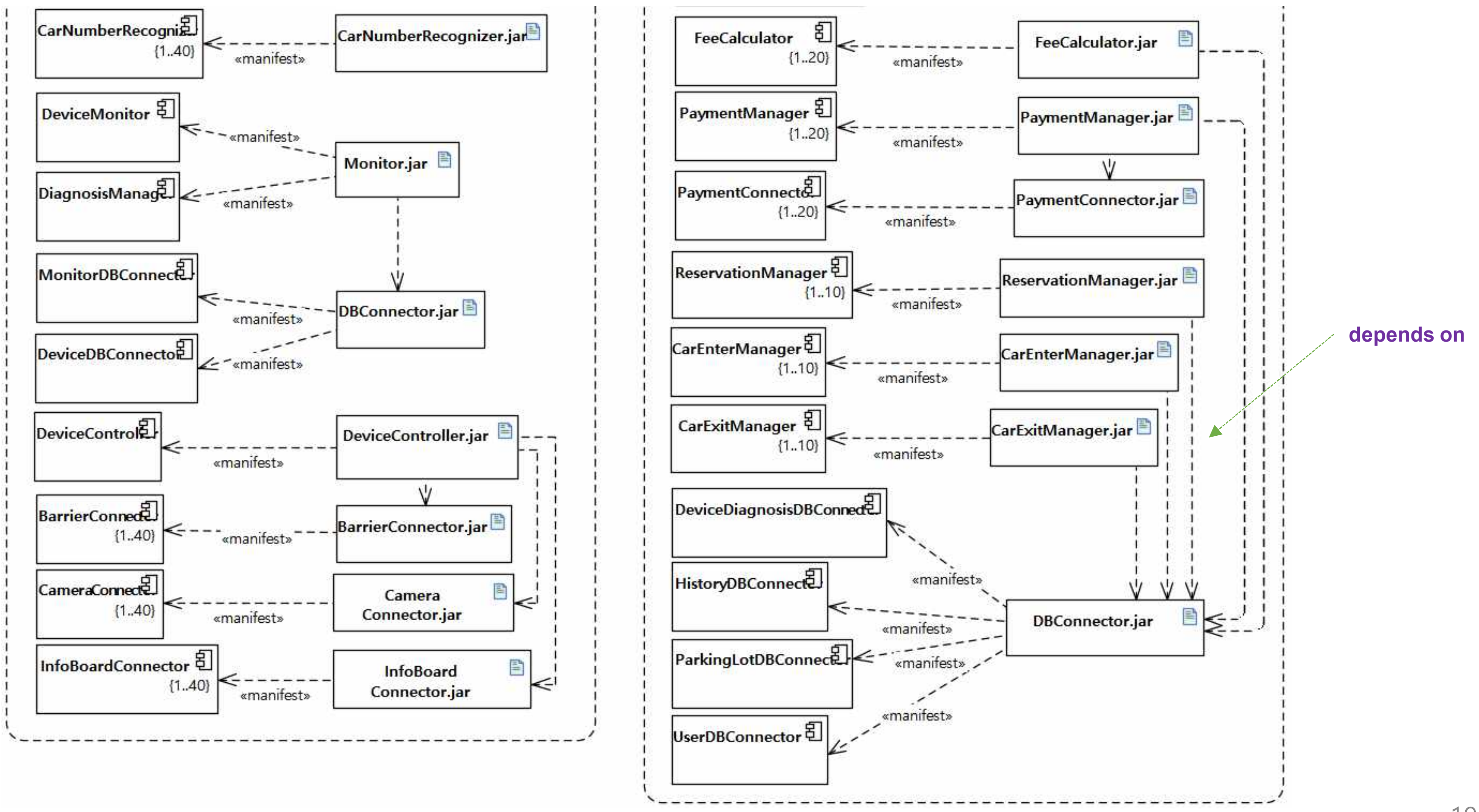


Artifact Definition Model
(Deployment View)

Artifact Definition Diagram - Examples

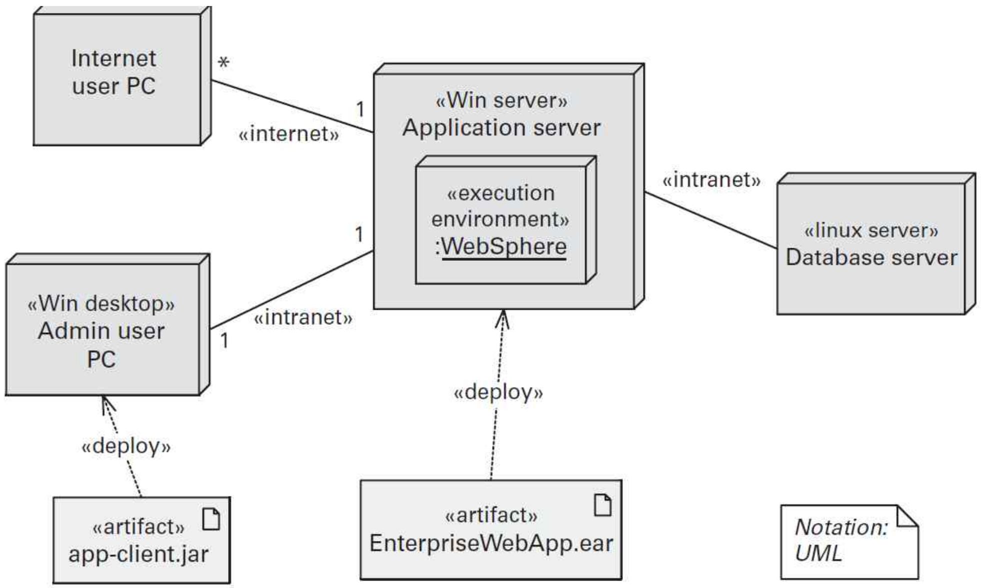


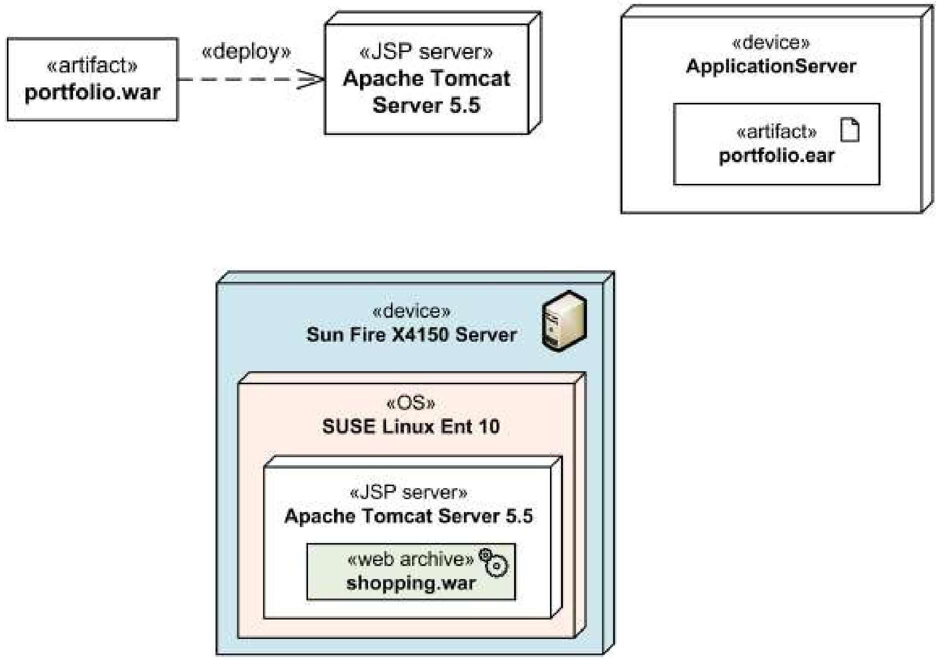
Artifact Definition Diagram - Examples



Artifact Deployment Model

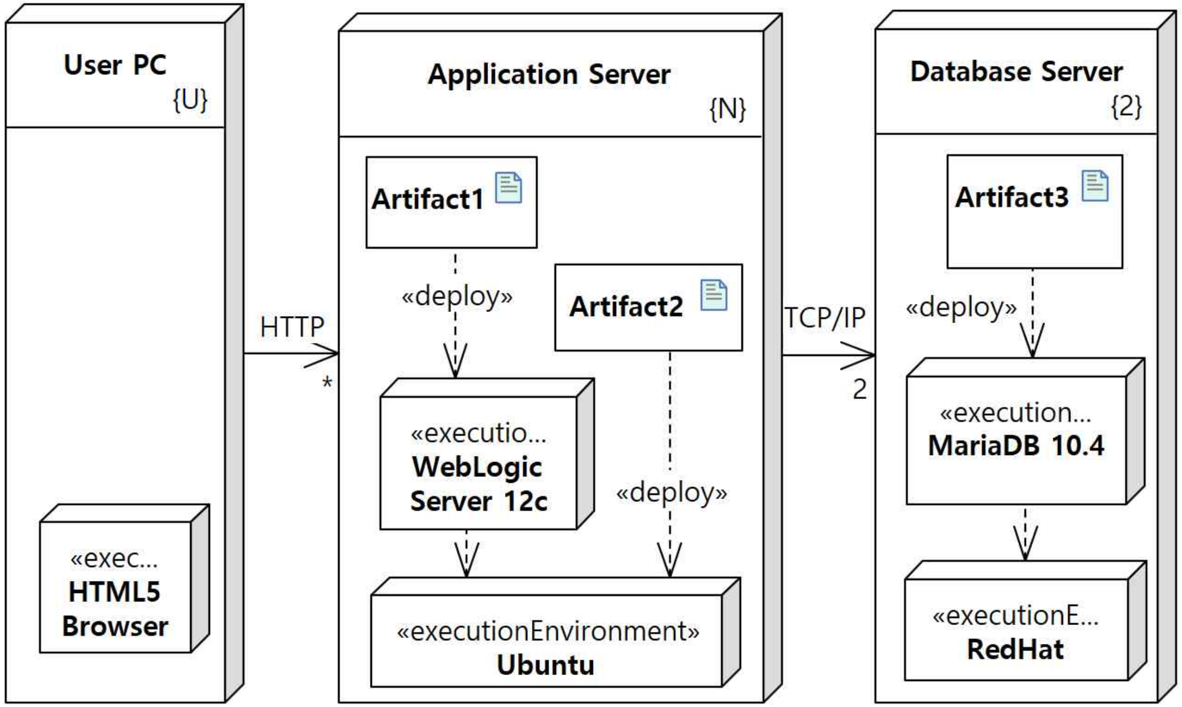
- The **distribution of artifacts** on a set of nodes so that they can be installed, configured, and hosted on physical nodes.



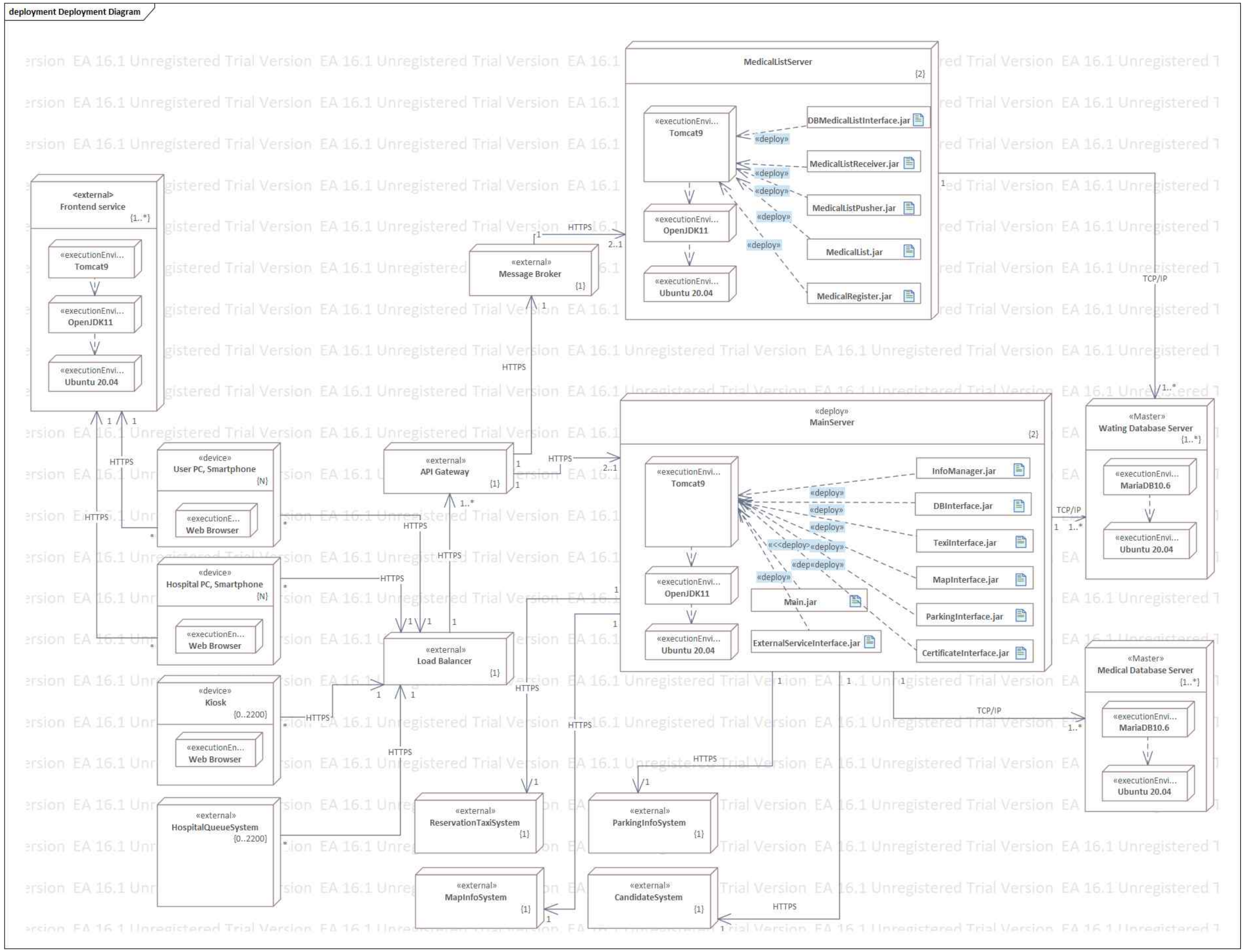


JAR : Java Archive (독립 실행)
WAR : Web Archive (JSP 서버 필요)
EAR : Enterprise Archive (Java Enterprise Edition 서버 필요)

Artifact Deployment Diagram - Examples

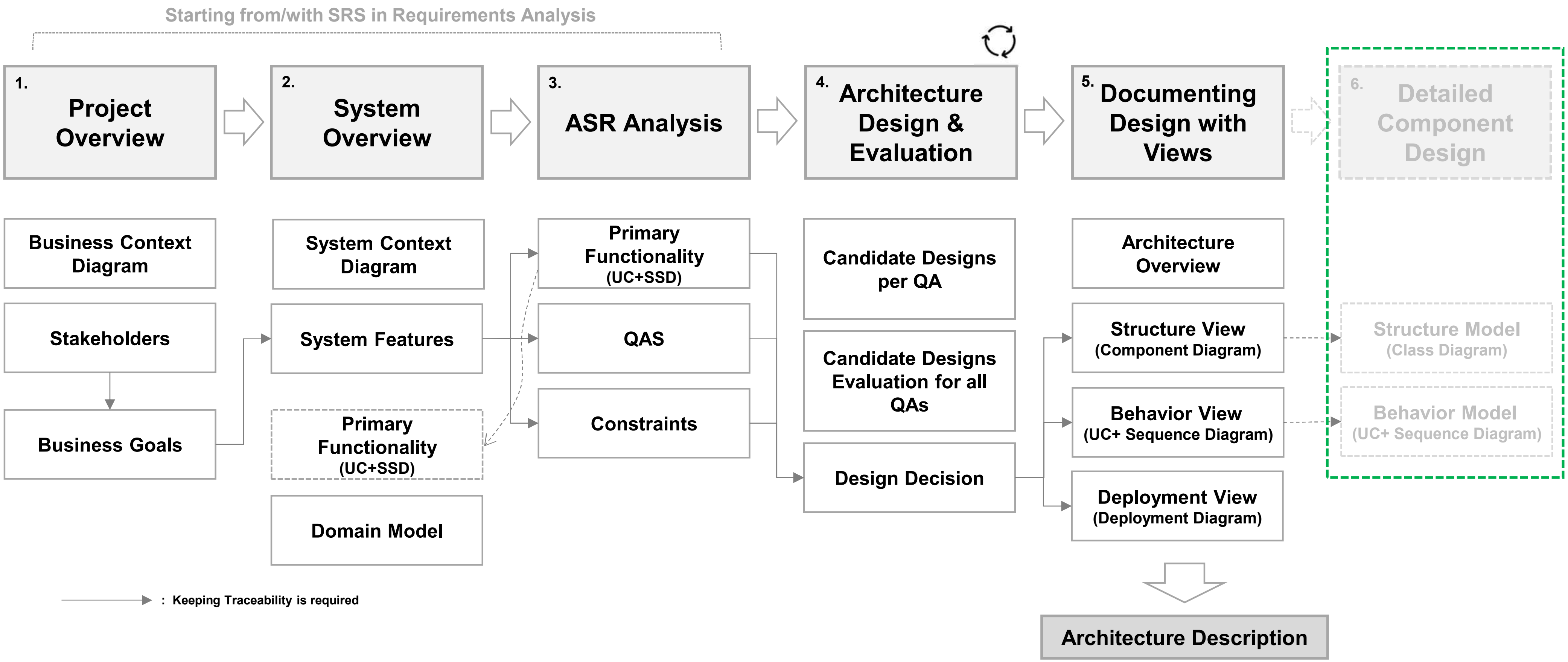


Artifact Deployment Diagram - Examples



6. Detailed Component Design

6. Detailed Component Design



Where We are Now in AD

1. Project Overview
    - 1.1 Project Background
    - 1.2 Business Context Diagram
    - 1.3 Stakeholders
    - 1.4 Business Goals

2. System Overview
    - 2.1 System Context Diagram
    - 2.2 External Entity
    - 2.3 External Interface
    - 2.4 System Features
    - 2.5 Domain Model
    - 2.6 Assumptions

3. Architectural Drivers
    - 3.1 Primary Functionality
    - 3.2 Quality Attribute Scenario
    - 3.3 Constraints

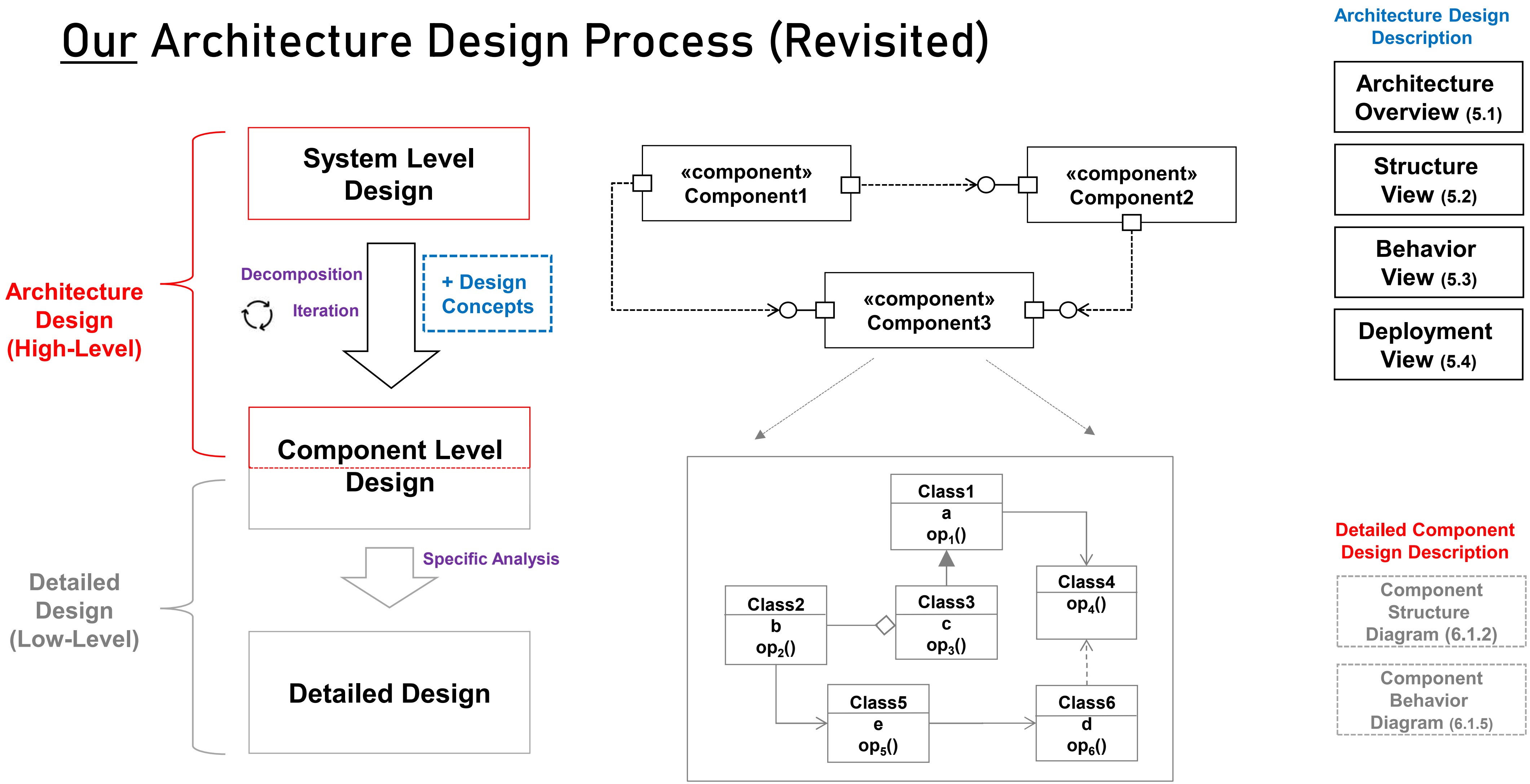
4. Architecture Design & Evaluation
    - 4.1 Candidate Designs per QA
    - 4.2 Candidate Designs Evaluation for all QAs
    - 4.3 Design Decision

5. Architecture Design Description
    - 5.1 Architecture Overview
    - 5.2 Structure View
    - 5.3 Behavior View
    - 5.4 Deployment View

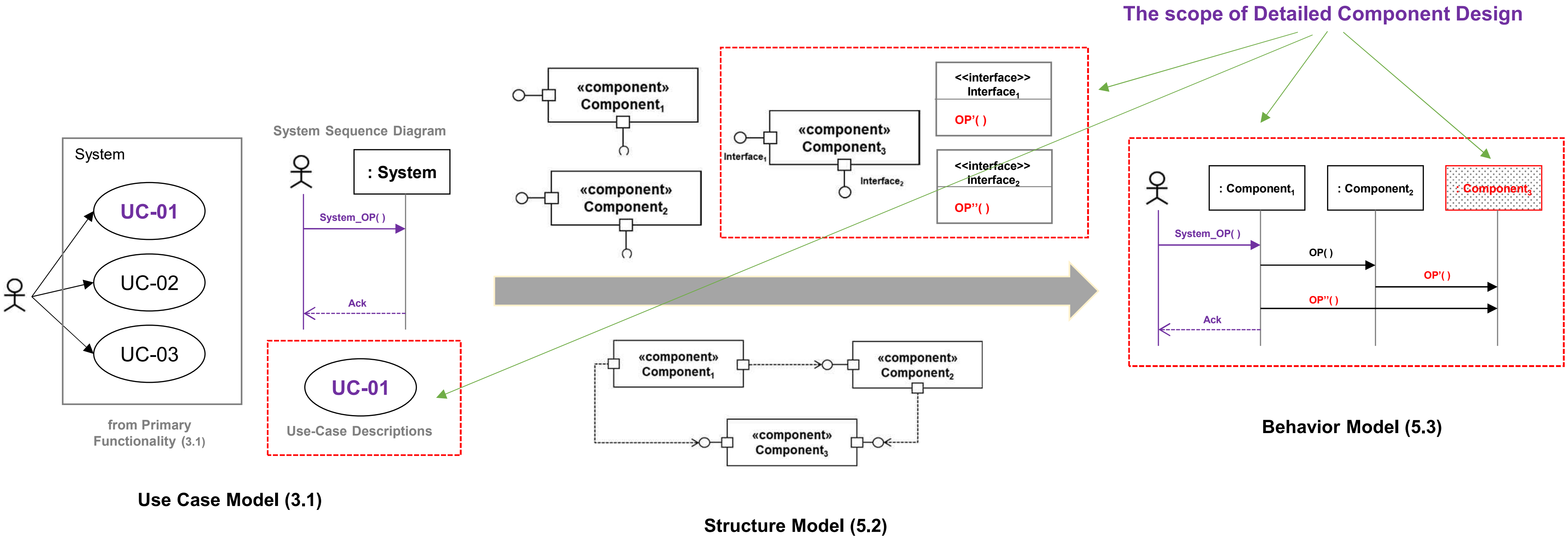
6. Component Design Description
- **6.1.2 Component Structure Model**
- **6.1.5 Component Behavior Model**

7. Architecture Traceability Summary

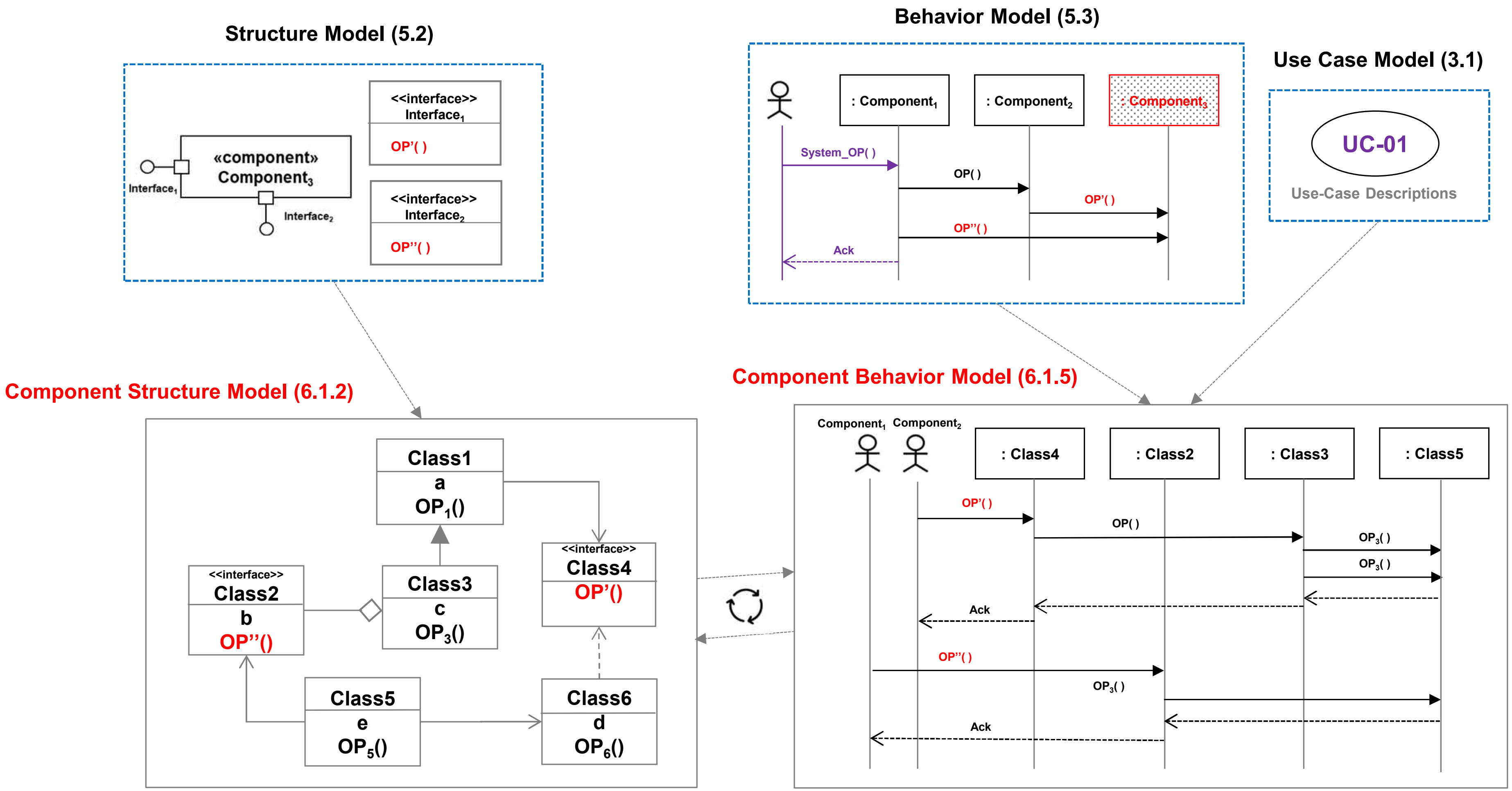
Our Architecture Design Process (Revisited)



The Scope of Detailed Component Design from Behavior View

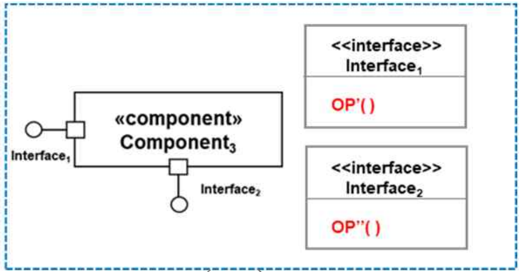


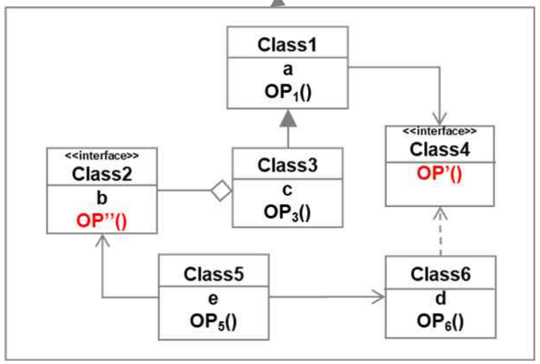
Detailed Component Design



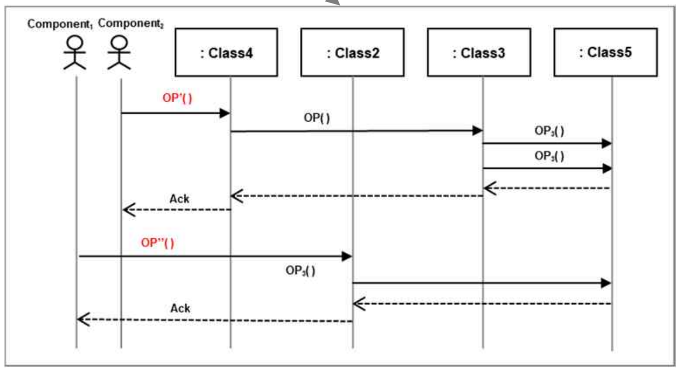
Detailed Component Design Description

- For each component,
  - **Component Structure Model**
    - Static Structure Diagram : **UML Class Diagram**
    - Element list
    - Design rationale
  - **Component Behavioral Model**
    - Component Behavior Diagram : **UML Sequence Diagram**





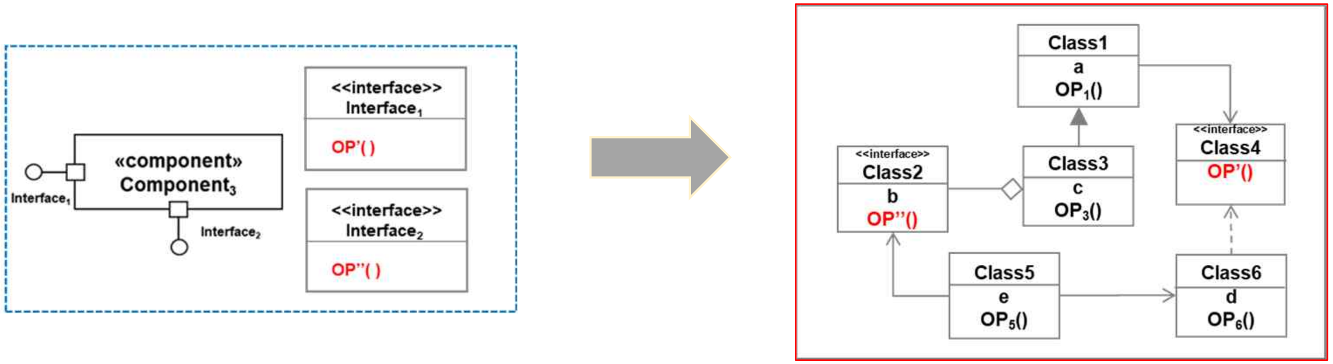
Static Structure Diagram



Component Behavior Diagram

Component Structure Model

- Represents the decomposition of the component
  - **UML Class Diagram**
  - A list of all elements
  - **Design Rationale** : Explains specific component decomposition techniques **(e.g., SOLID, Design Patterns)** to promote QAs



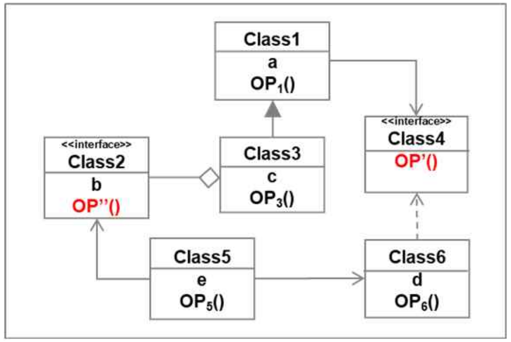
Static Structure Diagram

- For all provided interfaces of the component, behavior models need to be analyzed.
  - Component Behavior Diagram → **UML Sequence Diagram**

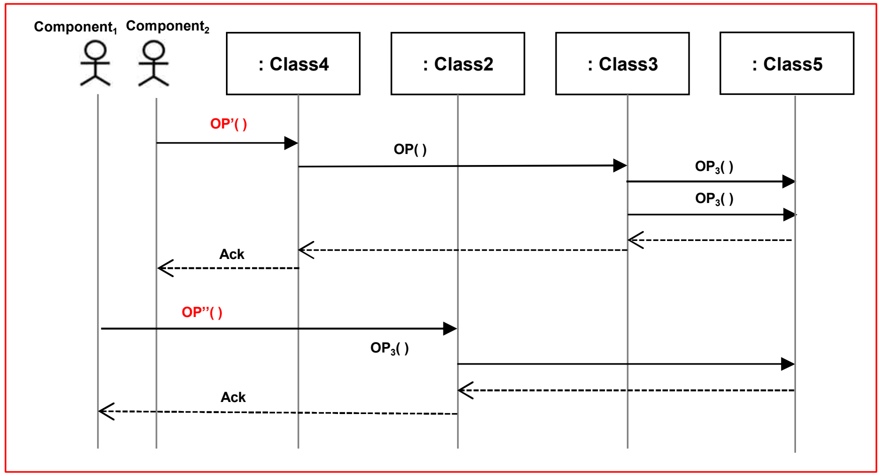
Component Behavior Model

- Describes how each operation of the provided interface can be realized with the collaboration of class instances

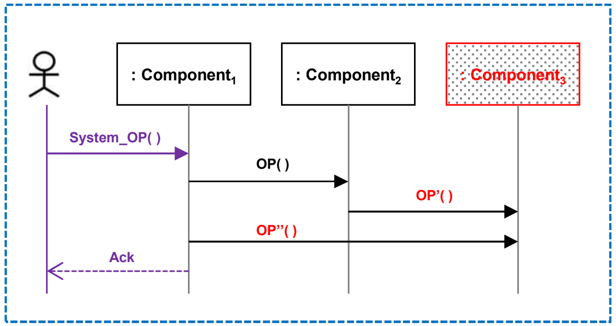
Static Structure Diagram (6.1.2)



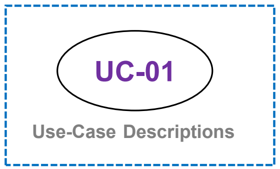
Component Behavior Diagram



Behavior Model (5.3)

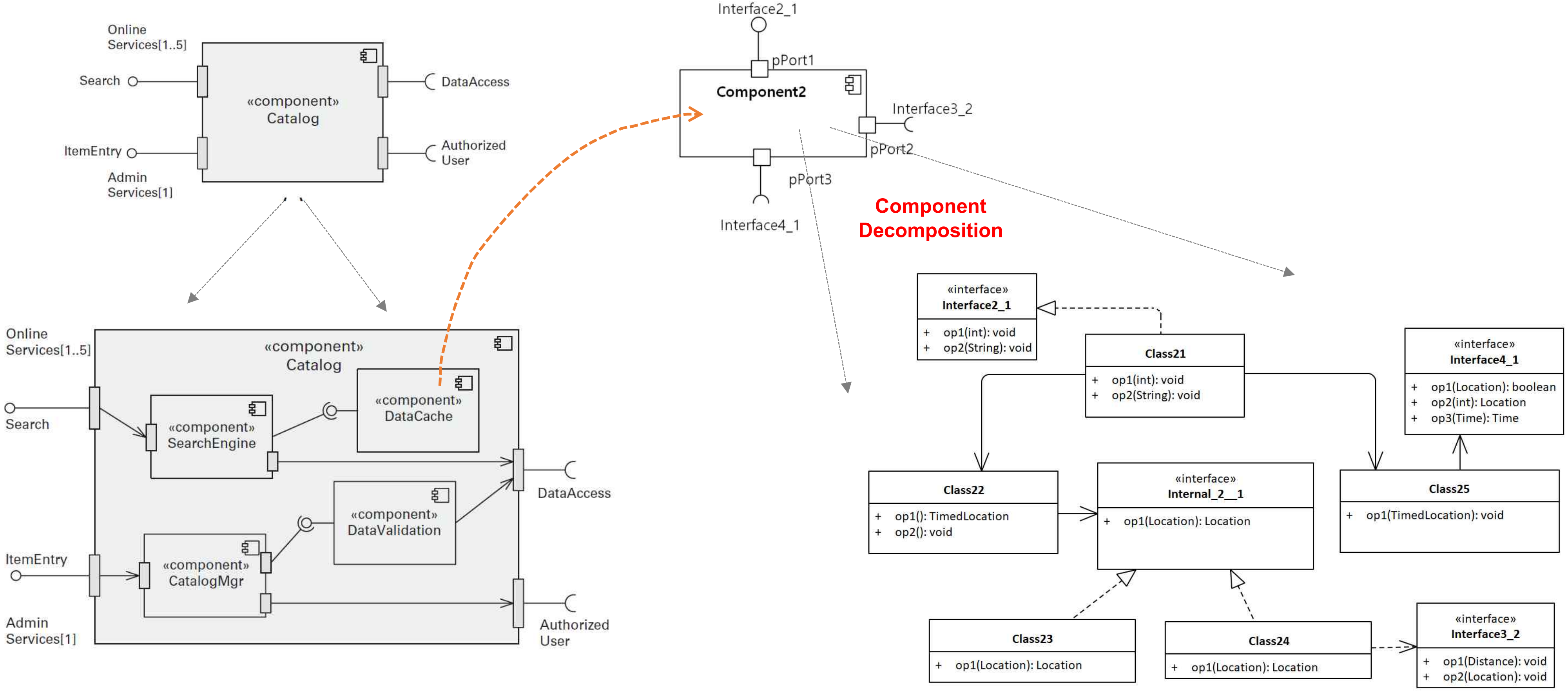


Use Case Model (3.1)

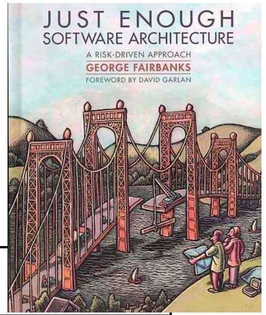


Component Decomposition

- A component consists of several fine-grained elements including smaller components and/or classes.
  - Use component decomposition strategies



Component Decomposition Strategies

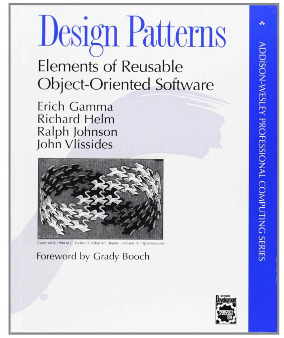


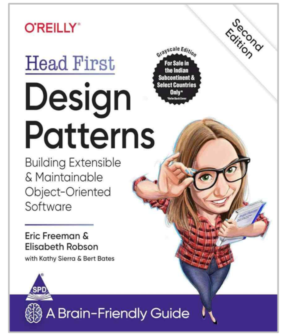
Decomposition Strategy	Description
Functionality	• Decomposing a system based on functionality is perhaps the most obvious strategy. • You can invent the required functionality and clump together with related functions.
Archetypes	• Archetypes / core types are salient types from the domain, such as a Contact, Advertisement, User, or Email • Characteristics of an archetype include having an independent existence and having few mandatory associations to other types
Patterns	• A system can be decomposed so that its components are elements defined by an **architectural patterns**, **design patterns**, GRAPS, and **design principles (SOLID)**. • Choosing an architectural pattern is highly effective at achieving quality attribute goals because each style has known qualities that it promotes.
Achievement of certain QA	• For example, to support **Modifiability (→Maintenainability)**, impact of any one change is localized.
Build-versus-buy decisions	• Some modules may be bought in the commercial marketplace, reused intact from a previous project, or obtained as open-source software.
Product line implementation	• It is essential to distinguish between common components, used in every or most products, and variable components, which differ across products.
Team allocation	• To allow implementation of different responsibilities in parallel, separate components that can be allocated to different teams should be defined

Design Patterns of GoF (Gang of Four)

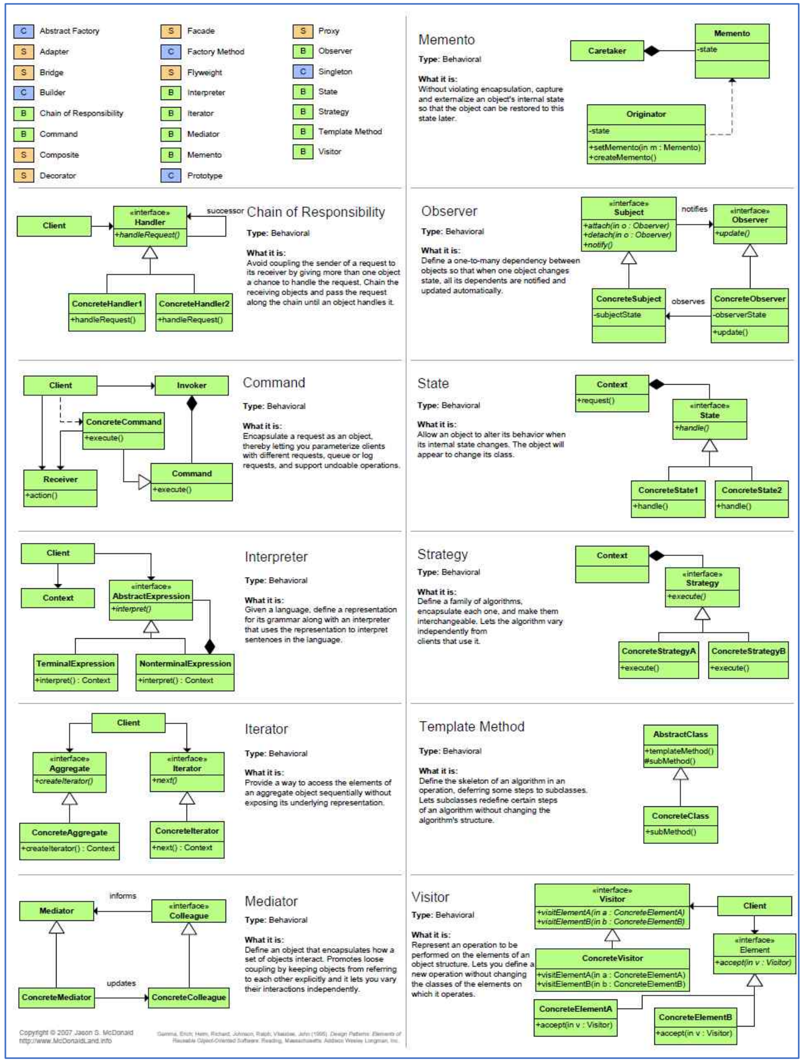
The GoF launched the software patterns movement, but many others have made significant contributions, including Ward Cunningham, Kent Beck, Jim Coplien, Grady Booch, Bruce Anderson, Richard Gabriel, Doug Lea, Peter Coad, and Doug Schmidt, to name just a few.

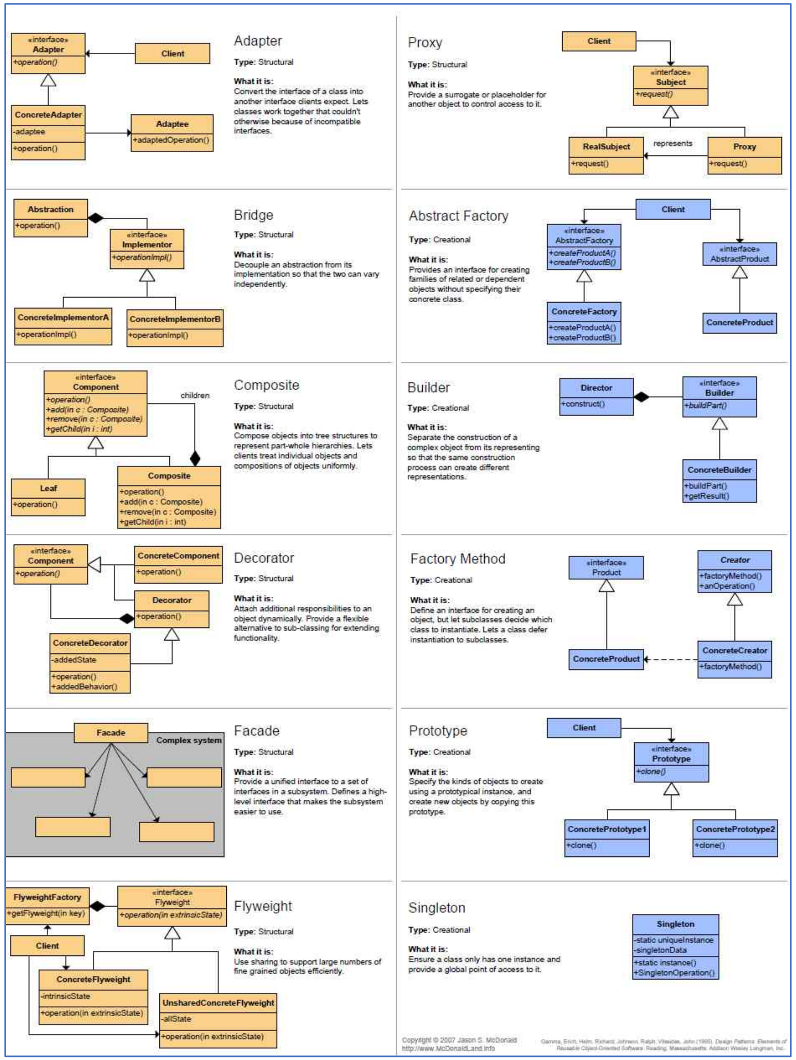
*John Vlissides passed away in 2005. A great loss to the Design Patterns community.



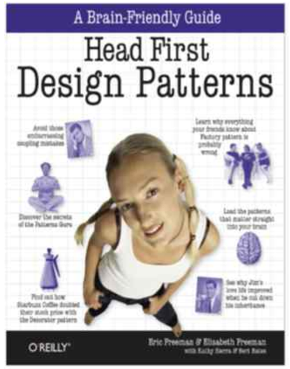


23 Design Patterns of GoF





Object-Oriented Design Principles in HFDP



Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

Design Principle

Program to an interface, not an implementation.

Design Principle

Favor composition over inheritance.

Design Principle

A class should have only one reason to change.

Design Principle

Depend upon abstractions. Do not depend upon concrete classes.

Design Principle

Strive for loosely coupled designs between objects that interact.

Design Principle

Classes should be open for extension, but closed for modification.

The Hollywood Principle

Don't call us, we'll call you.

Design Principle

Principle of Least Knowledge - talk only to your immediate friends.

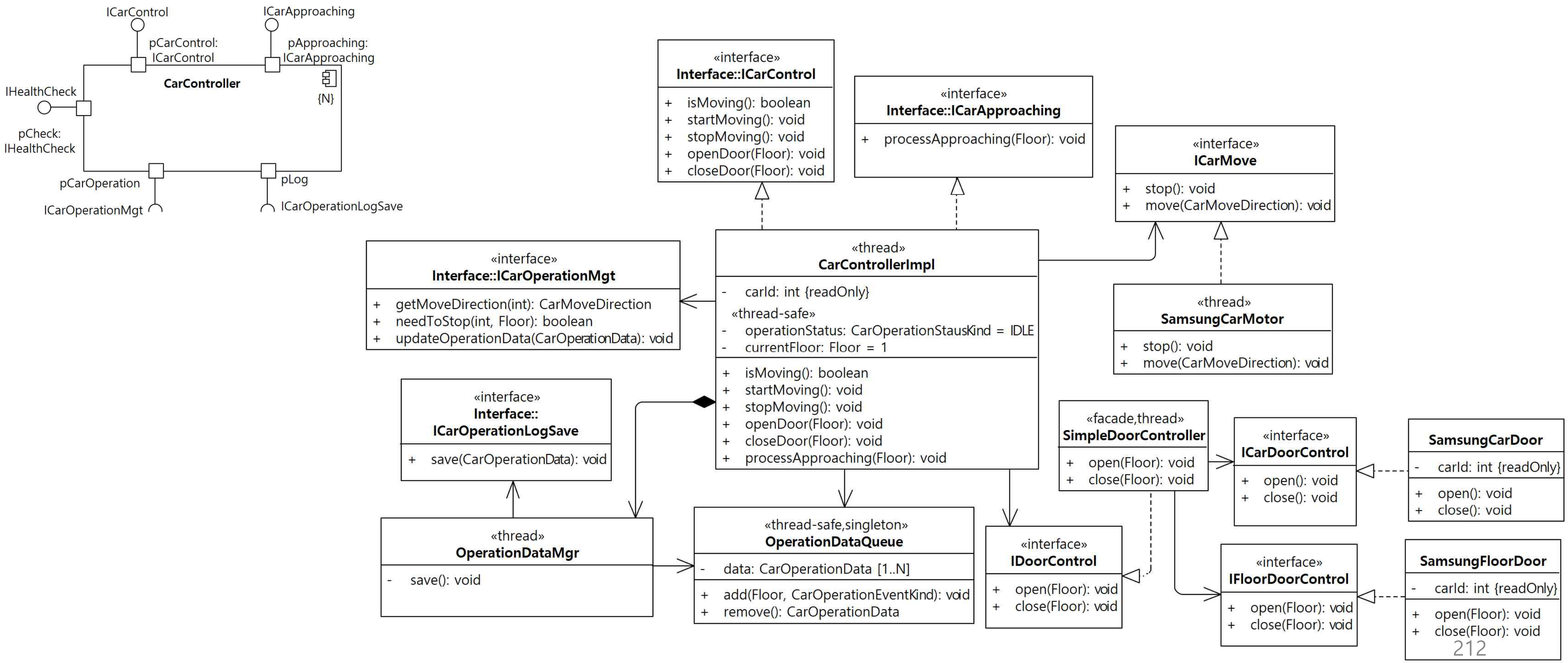
Object-Oriented Design Principles - SOLID

- **SOLID** : 5 basic principles object-oriented design for maintainable and extensible systems
  - **SRP : S**ingle **R**esponsibility **P**rinciple
  - **OCP : O**pen **C**losed **P**rinciple
  - **LSP : L**iskov **S**ubstitution **P**rinciple
  - **ISP : I**nterface **S**egregation **P**rinciple
  - **DIP : D**ependency **I**nversion **P**rinciple

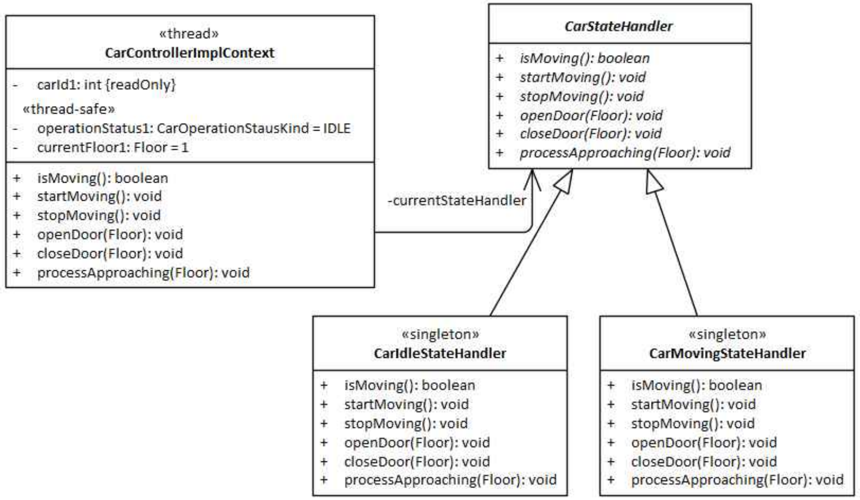
Name	Full Name	Description	Ways to Apply with
SRP	Single Responsibility Principle	A module should have one, and only one, reason to change.	Separate the module into multiple ones for each reason.
ISP	Interface Segregation Principle	Client should not be affected by the interface it does not use.	Make 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 modules 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.

Component Structure Model : Examples

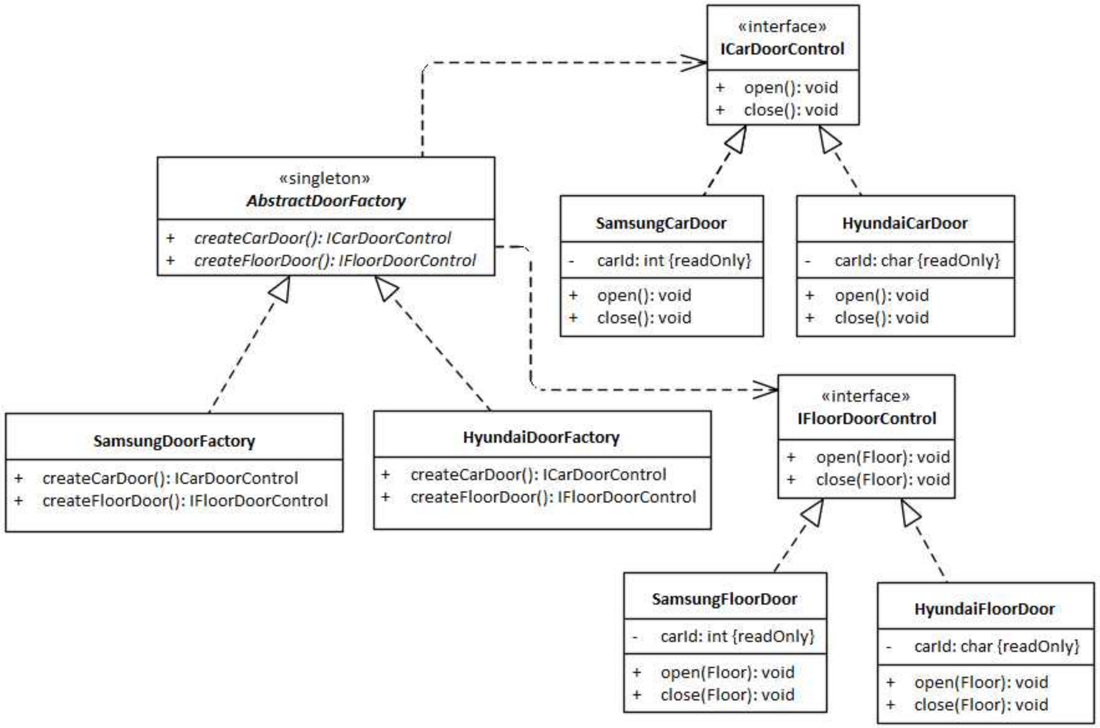
- **CarController** Component



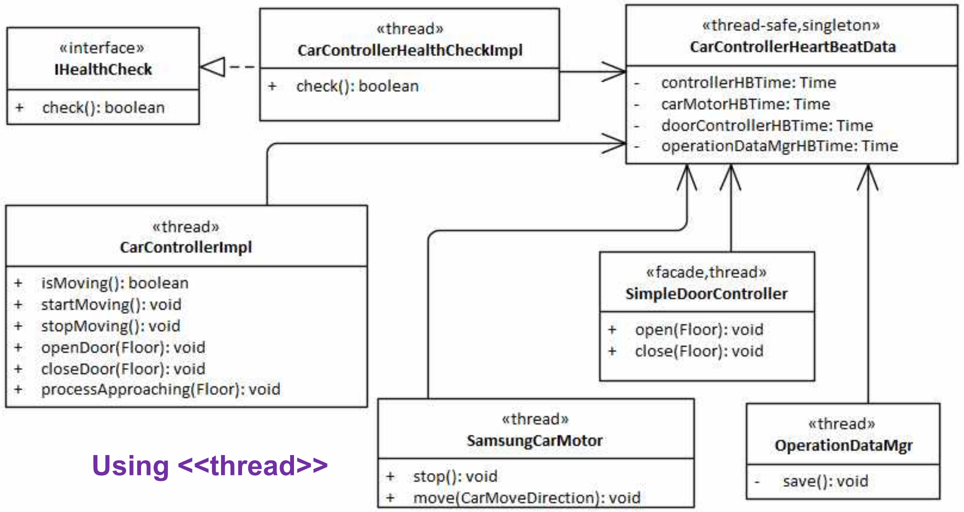
Component Structure Model : Examples



State Pattern

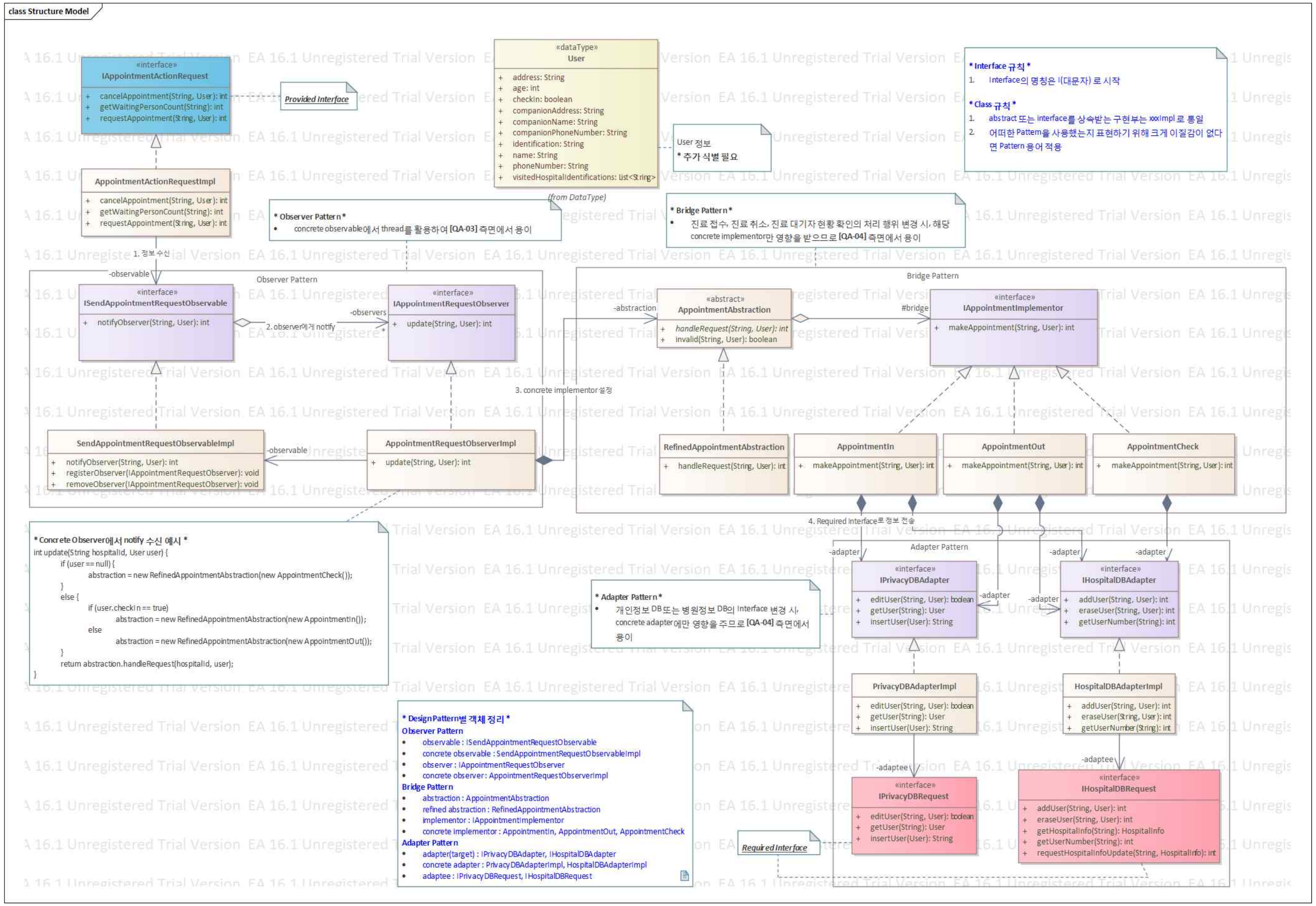


Abstract Factory Pattern

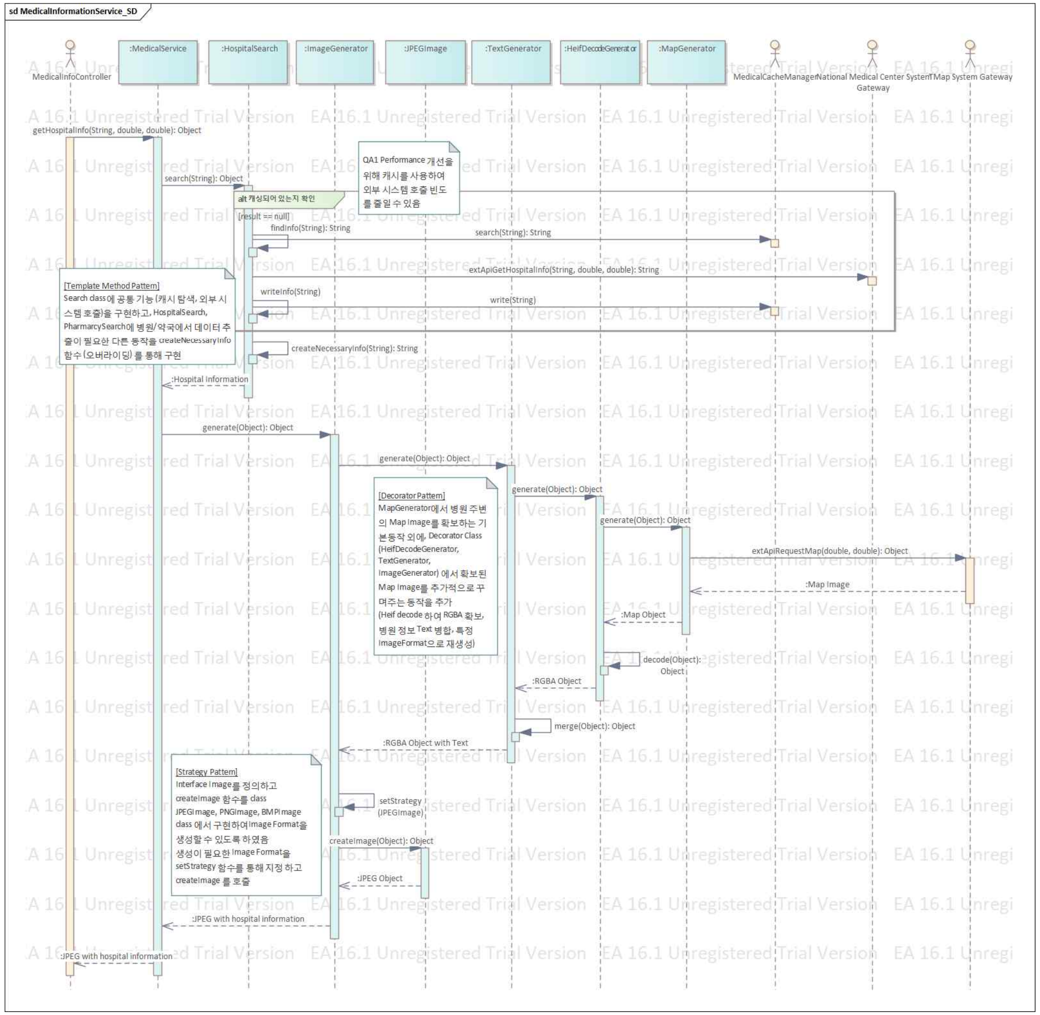


Using <<thread>>

Component Structure Model : Examples



Component Behavior Model : Examples



Component Structure Model – Element List

- Describe each element comprising the component with its responsibility.

Element Name	Responsibility
Class21	
Class22	
Internal_2_1	
Class23	
Class24	
Class25	

Component Structure Model – Design Rationale

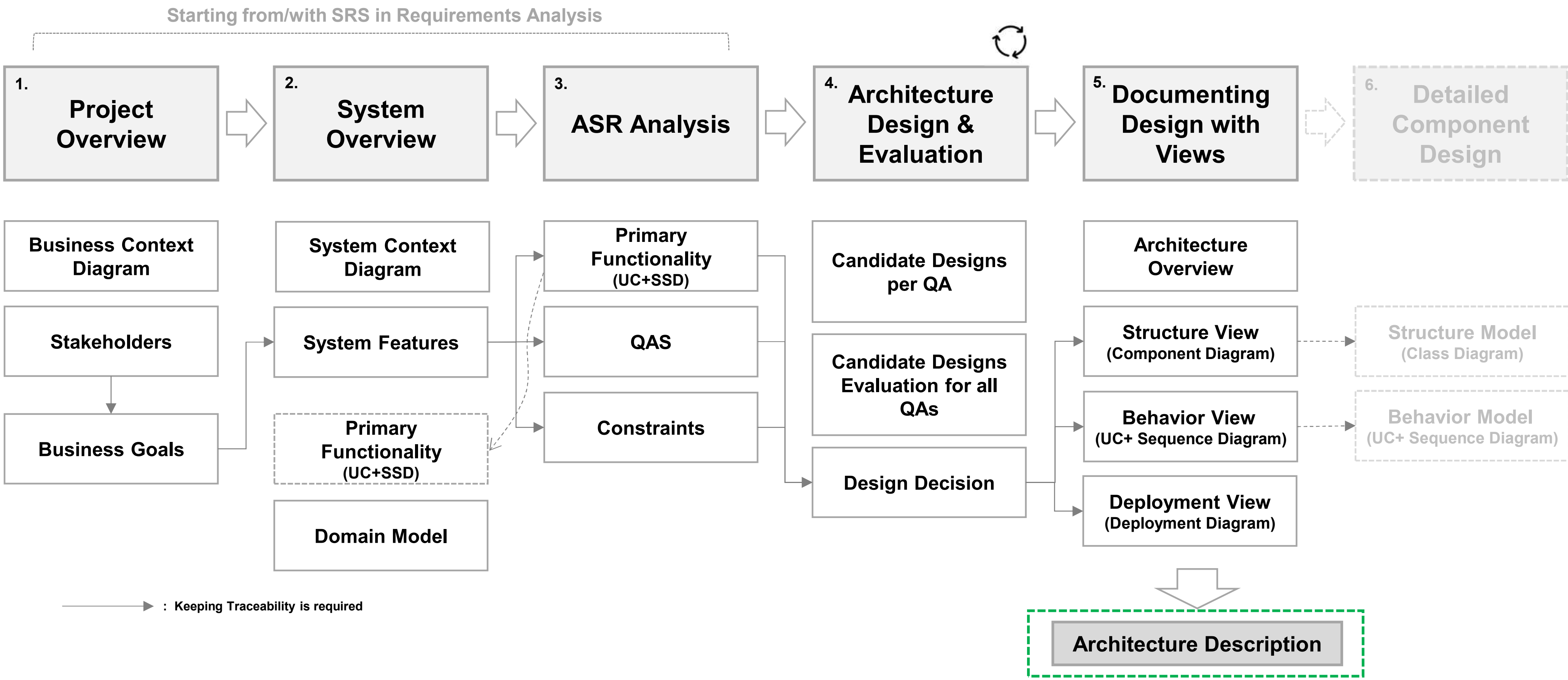
- Describe the **rationale** for the decomposition.
  - Explain your specific component decomposition strategies in detail
    - Design patterns, OO design principles (SOLID)

  - If possible, relate your component design decisions to the **quality requirement** by describing how each quality requirement are promoted by the decomposition.
    - Not all QA/QAS are relevant to the specific detailed component design at the class/object level.

QA	Relevant Elements	Description
QA1 (Performance)		
QAS-03		

7. Architecture Traceability Summary

7. Architecture Traceability Summary

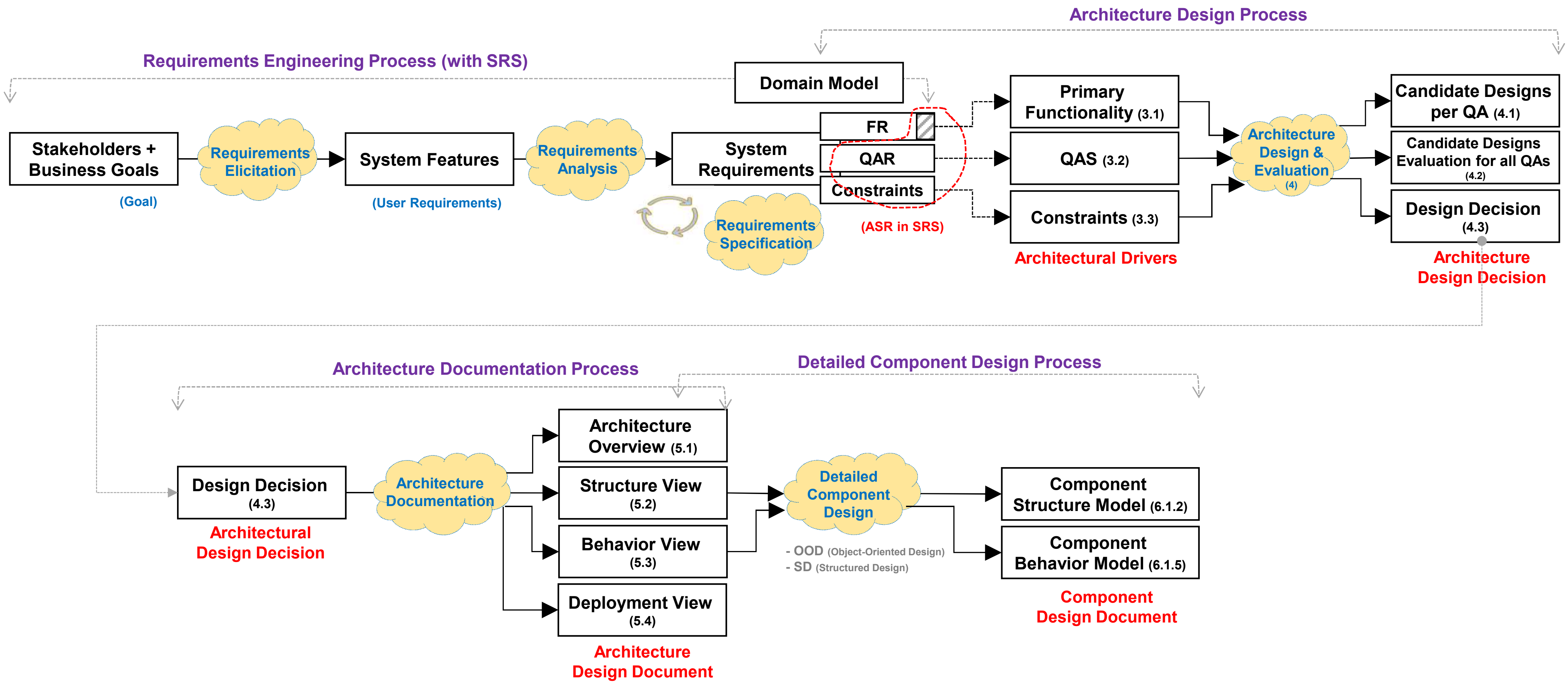


Where We are Now in AD

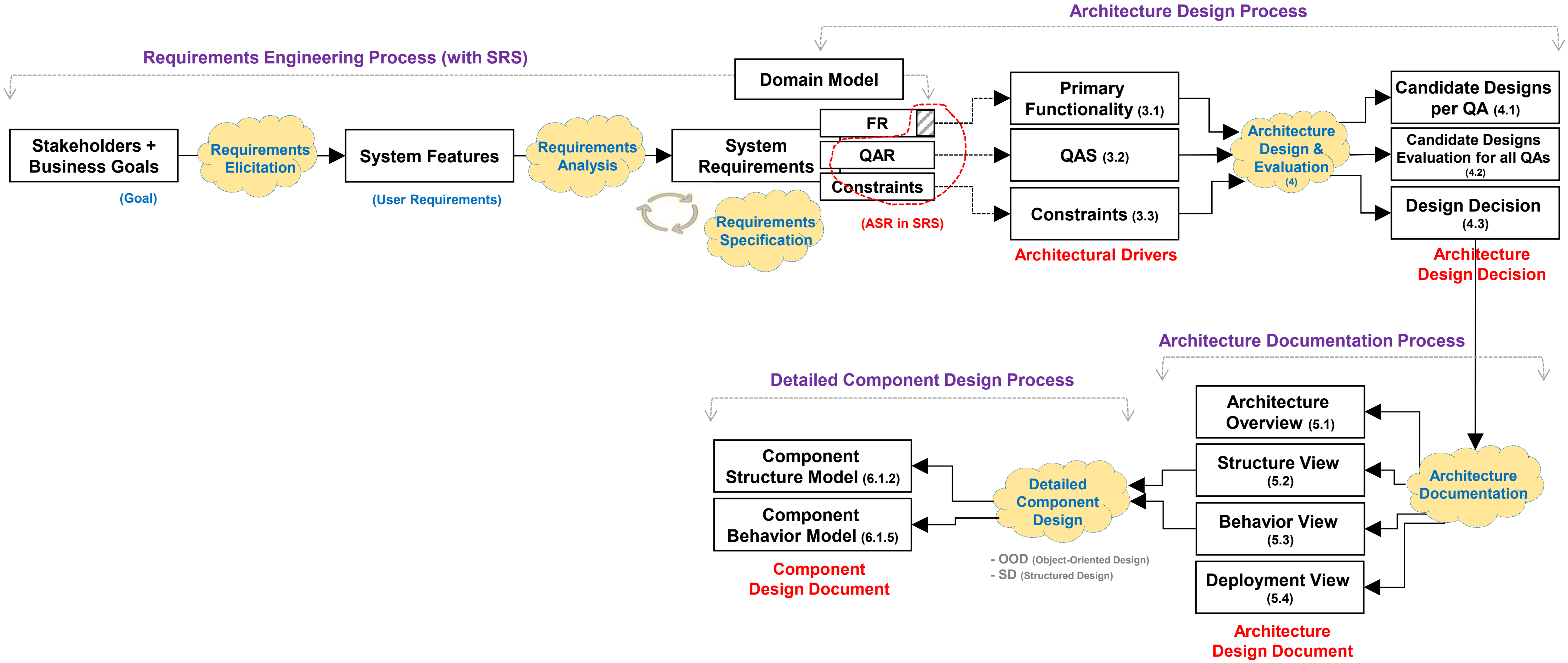
1. Project Overview
    - 1.1 Project Background
    - 1.2 Business Context Diagram
    - 1.3 Stakeholders
    - 1.4 Business Goals
2. System Overview
    - 2.1 System Context Diagram
    - 2.2 External Entity
    - 2.3 External Interface
    - 2.4 System Features
    - 2.5 Domain Model
    - 2.6 Assumptions
3. Architectural Drivers
    - 3.1 Primary Functionality
    - 3.2 Quality Attribute Scenario
    - 3.3 Constraints
4. Architecture Design & Evaluation
    - 4.1 Candidate Designs per QA
    - 4.2 Candidate Designs Evaluation for all QAs
    - 4.3 Design Decision
5. Architecture Design Description
    - 5.1 Architecture Overview
    - 5.2 Structure View
    - 5.3 Behavior View
    - 5.4 Deployment View
6. Component Design Description
    - 6.1.2 Component Structure Model
    - 6.1.5 Component Behavior Model

7. Architecture Traceability Summary

The Overall CEP Process



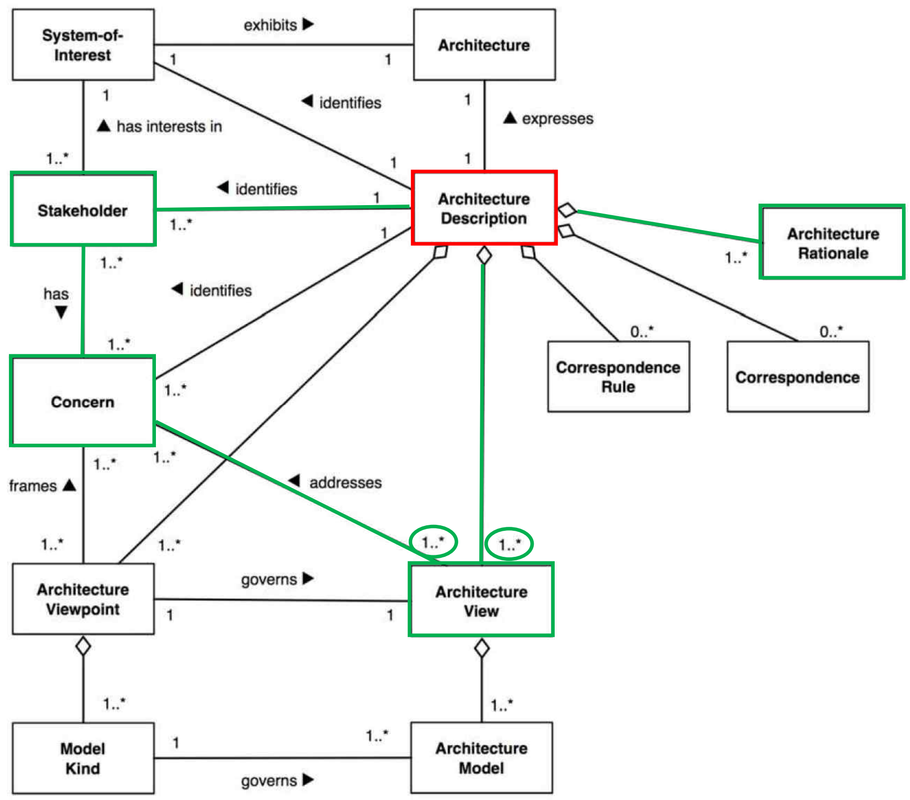
The Overall CEP Process



Architecture Traceability

- **ISO/IEC/IEEE 42010:2011** “Systems and Software Engineering - Architecture Description

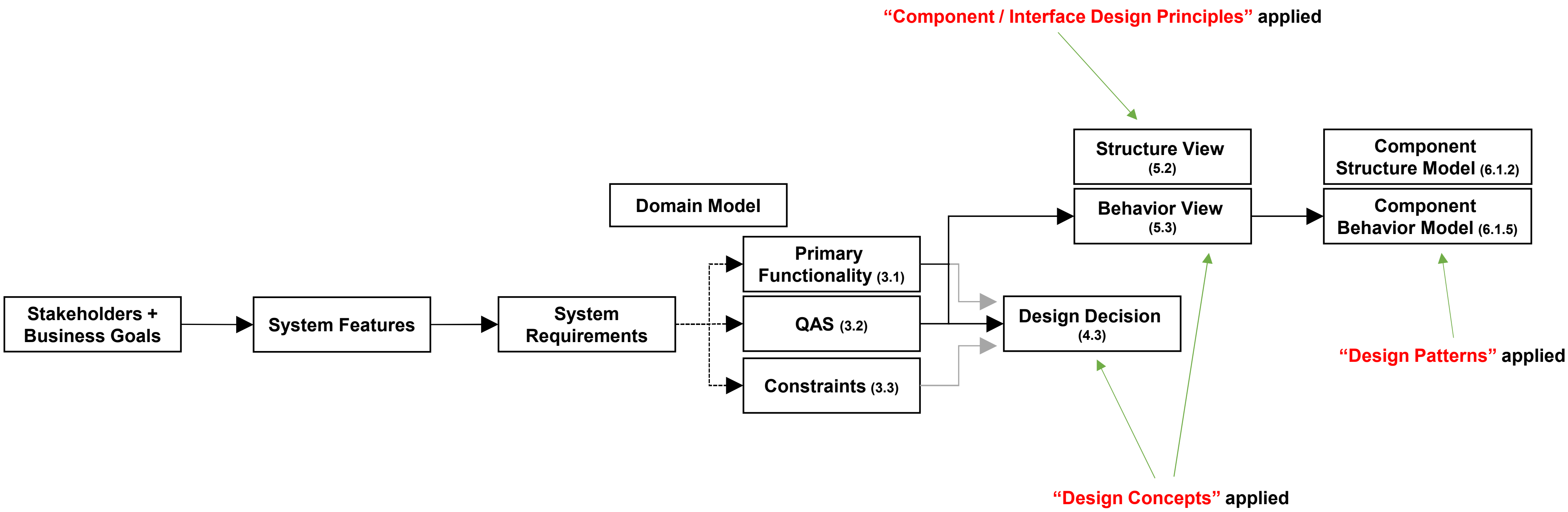
- AD should demonstrate how an architecture meets the needs of the system’s diverse stakeholders.



- Architecture traceability starts from Stakeholders.
    - Stakeholders
    - Concerns
    - Architecture Views
    - Architecture Rationale

7.1 Architecture Traceability Graph

- **A full-scale graph** tracing from stakeholders up to components (and classes)
  - Any notation (graph or table) is possible.
  - Every individual item in the graph should be traceable bidirectionally.



7.2 Summary of Traceability Items

- Explains all elements which take part in the architecture traceability briefly and clearly

Traceability Items		Description
ID	Title	

7.3 Safety Case

- Demonstrate reasonably that your claim is successfully satisfied by your architecture design
  - Examples of claims :
    - “*Customers will not wait more than 5 minutes.*”
    - “*The system will not expose any customer information.*”

- Choose a claim and demonstrate their satisfaction with your traceability, reasonably.