Software Engineering

(3) System Analysis Design Principles

Sokendai / National Institute of Informatics

Fuyuki Ishikawa / 石川　冬樹

f-ishikawa@nii.ac.jp / @fyufyu

http://research.nii.ac.jp/~f-ishikawa/

大学共同利用機関法人 情報・システム研究機構
国立情報学研究所
National Institute of Informatics

TOC

- <u>Quality Characteristics of Software Systems</u>
- System Analysis
- Design Principles

Quality?

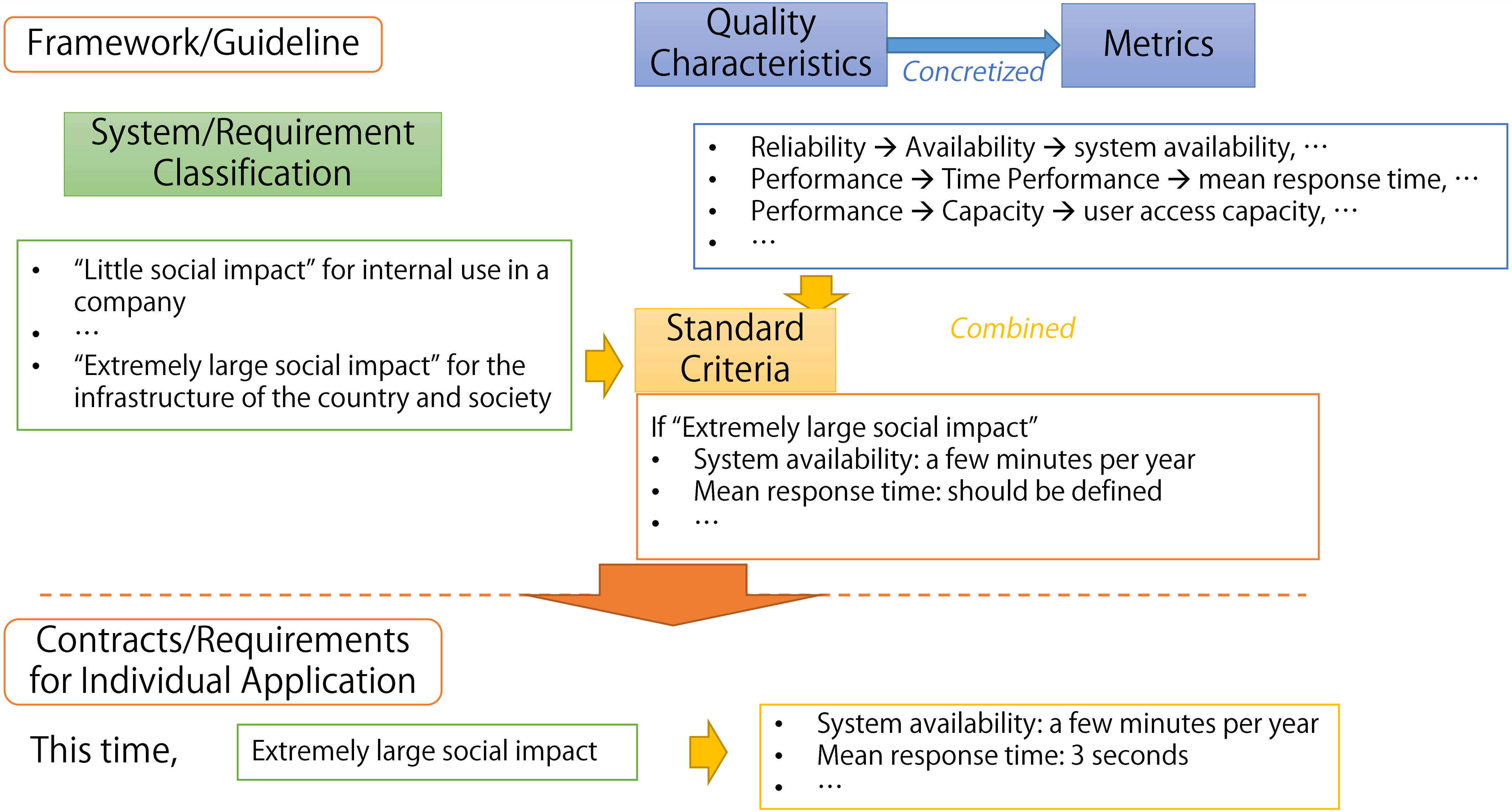
- Definition of the term “Quality”

> degree to which a set of inherent characteristics of an object fulfils requirements
>
> ISO 9000 for quality management systems

> degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value
>
> ISO/IEC 25010 for systems and software quality requirements and evaluation

- Multiple characteristics (aspects/viewpoints)
- Relative to needs or requirements
- Primarily deal with non-functional requirements

Decision Making of Requirements for Quality



Framework/Guideline

Quality Characteristics

Concretized

Metrics

System/Requirement Classification

- Reliability → Availability → system availability, …
- Performance → Time Performance → mean response time, …
- Performance → Capacity → user access capacity, …
- …

- “Little social impact” for internal use in a company
- …
- “Extremely large social impact” for the infrastructure of the country and society

Standard Criteria

Combined

If “Extremely large social impact”
- System availability: a few minutes per year
- Mean response time: should be defined
- …

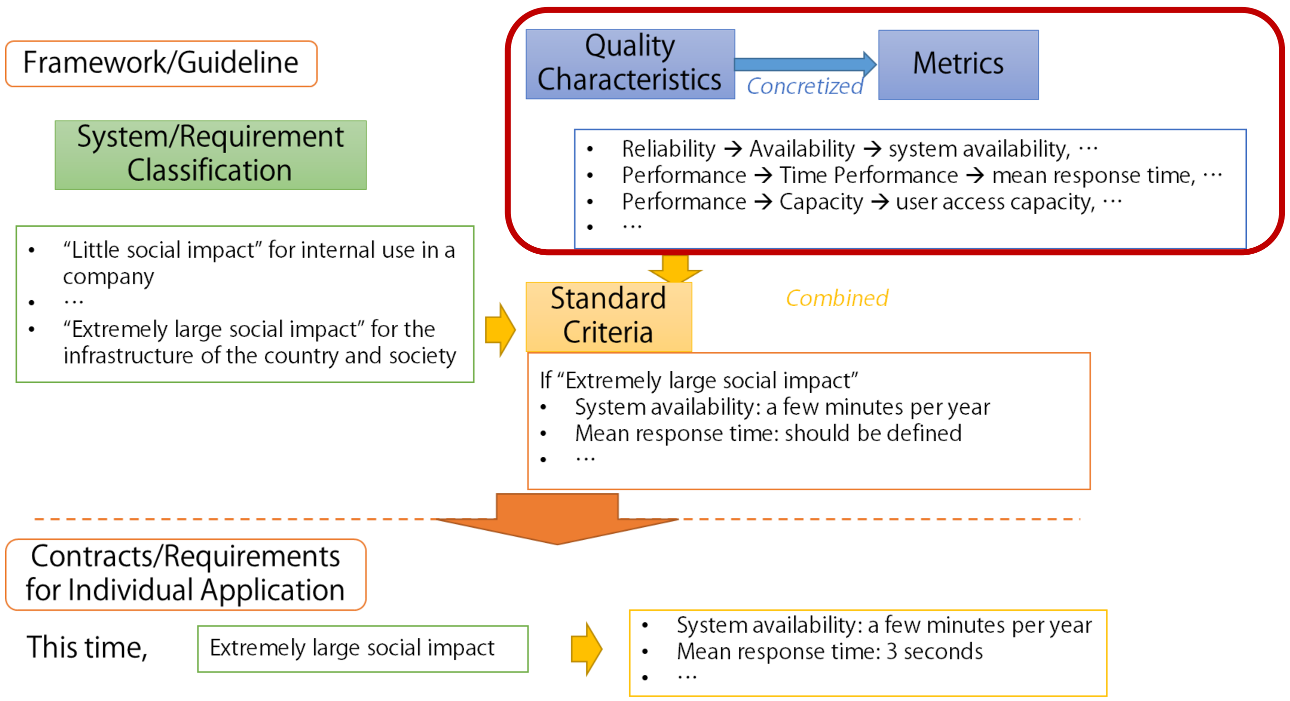
Contracts/Requirements for Individual Application

This time, Extremely large social impact

- System availability: a few minutes per year
- Mean response time: 3 seconds
- …

SQuaRE (ISO 250XX Series): Overview

- Provides “quality models”
  - SQuaRE: Systems and software Quality Requirements and Evaluation



Framework/Guideline

System/Requirement Classification

- “Little social impact” for internal use in a company
- …
- “Extremely large social impact” for the infrastructure of the country and society

Quality Characteristics → Metrics

Concretized

- Reliability → Availability → system availability, …
- Performance → Time Performance → mean response time, …
- Performance → Capacity → user access capacity, …
- …

Standard Criteria

Combined

If “Extremely large social impact”

- System availability: a few minutes per year
- Mean response time: should be defined
- …

Contracts/Requirements for Individual Application

This time, Extremely large social impact

- System availability: a few minutes per year
- Mean response time: 3 seconds
- …

This part

SQuaRE (ISO 250XX Series): Classification of Quality

- Quality of Product
  - Performance, availability, security, etc. (as mentioned)
  - Divided into internal/external, i.e., what are inside development teams and what are observable from users
- Quality in Use
  - e.g., How effective is the actual use, in specific contexts?
- Quality of Process
  - e.g., how well documentation is systematically done

SQuaRE (ISO 250XX Series): Product Quality

Functional Suitability
- Functional Completeness
- Functional Correctness
- Functional Appropriateness

Performance Efficiency
- Time Behavior
- Resource Utilization
- Capacity

Security
- Confidentiality
- Integrity
- Non-repudiation
- Authenticity
- Accountability

Compatibility
- Co-existence
- Interoperability

Usability
- Appropriateness Recognizability
- Learnability
- Operability
- User Error Protection
- User Interface Aesthetics
- Accessibility

Reliability
- Maturity
- Availability
- Fault Tolerance
- Recoverability

Maintainability
- Modularity
- Reusability
- Analyzability
- Modifiability
- Testability

Portability
- Adaptability
- Installability
- Replaceability

SQuaRE (ISO 250XX Series): Quality in Use

Satisfaction
- Usefulness
- Trust
- Pleasure
- Comfort

Effectiveness
- Effectiveness

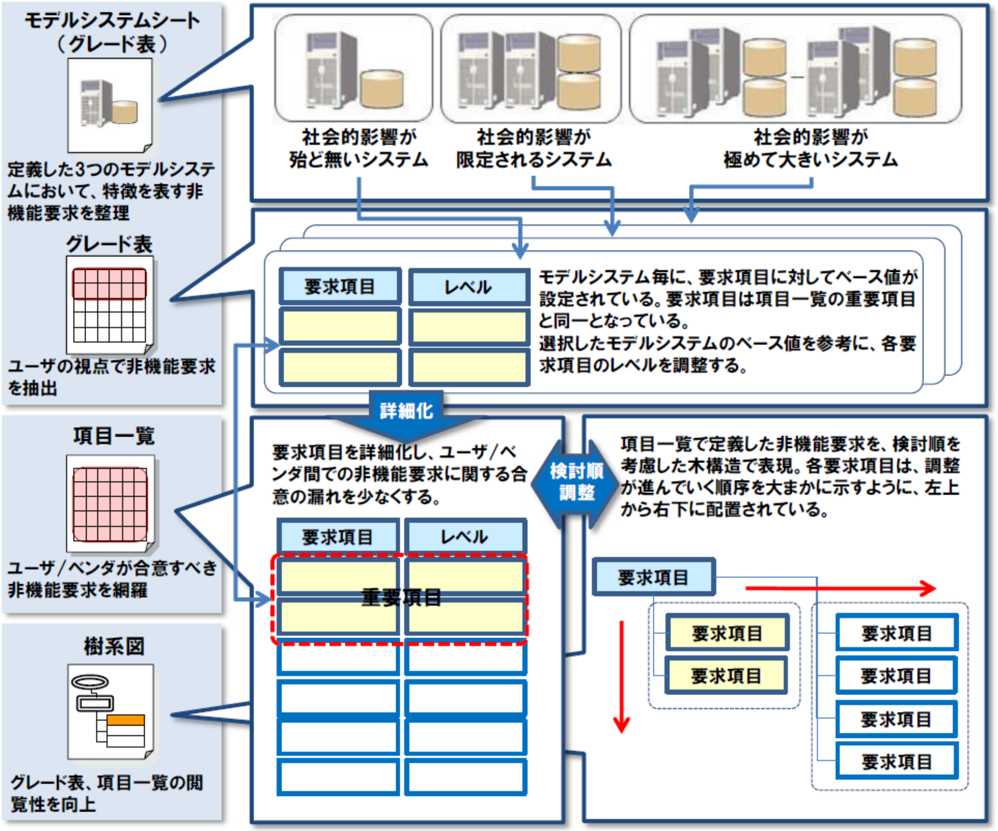
Efficiency
- Efficiency

Freedom from risk
- Economic risk mitigation
- Health and safety risk mitigation
- Environmental risk mitigation

Context Coverage
- Context completeness
- Flexibility

Ref: 非機能要求グレード (Guideline in Japan)

- "System Type → Level of Non-Functional Requirements" (from IPA, the initial ver. in 2010)



定義した3つのモデルシステムにおいて、特徴を表す非機能要求を整理

ユーザの視点で非機能要求を抽出

ユーザ／ベンダが合意すべき非機能要求を網羅

グレード表、項目一覧の閲覧性を向上

モデルシステム毎に、要求項目に対してベース値が設定されている。要求項目は項目一覧の重要項目と同一となっている。
選択したモデルシステムのベース値を参考に、各要求項目のレベルを調整する。

要求項目を詳細化し、ユーザ／ベンダ間での非機能要求に関する合意の漏れを少なくする。

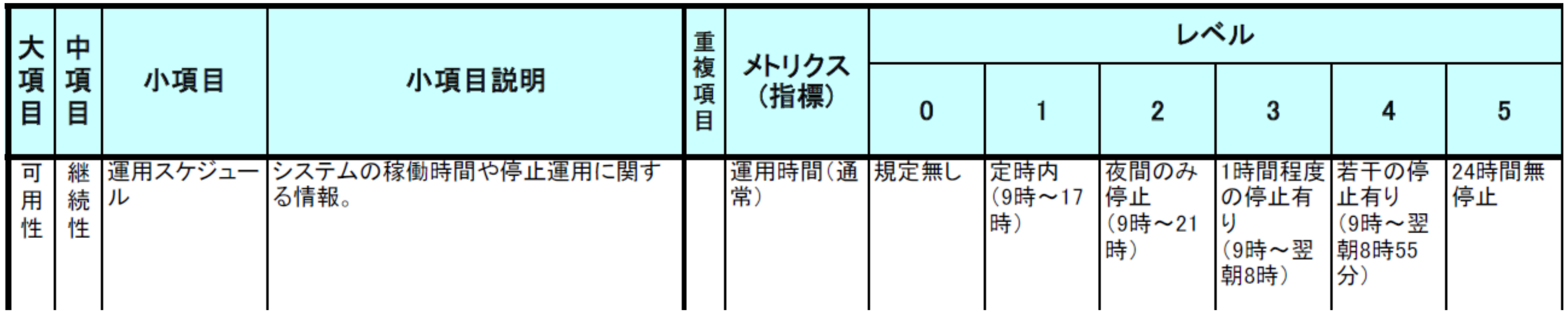
項目一覧で定義した非機能要求を、検討順を考慮した木構造で表現。各要求項目は、調整が進んでいく順序を大まかに示すように、左上から右下に配置されている。

図 1.4.2.1　非機能要求グレードの概要と全体イメージ

[非機能要求グレード2018 利用ガイド解説編より https://www.ipa.go.jp/sec/softwareengineering/std/ent03-b.html]

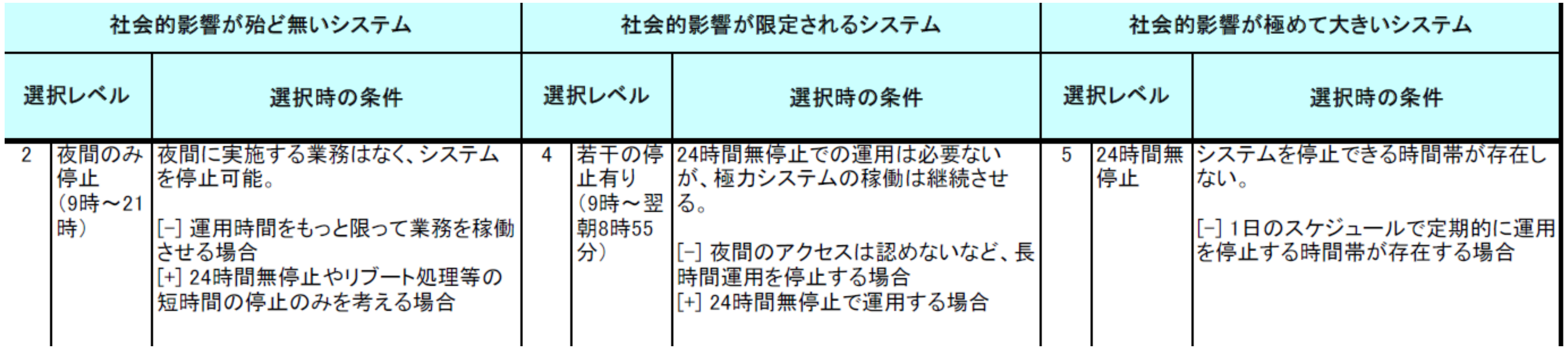
Ref: 非機能要求グレード (Guideline in Japan)

- Quality characteristics and their levels



大項目	中項目	小項目	小項目説明	重複項目	メトリクス（指標）	レベル					
						0	1	2	3	4	5
可用性	継続性	運用スケジュール	システムの稼働時間や停止運用に関する情報。		運用時間（通常）	規定無し	定時内（9時～17時）	夜間のみ停止（9時～21時）	1時間程度の停止有り（9時～翌朝8時）	若干の停止有り（9時～翌朝8時55分）	24時間無停止

- Level selection according to system types



社会的影響が殆ど無いシステム			社会的影響が限定されるシステム			社会的影響が極めて大きいシステム		
選択レベル		選択時の条件	選択レベル		選択時の条件	選択レベル		選択時の条件
2	夜間のみ停止（9時～21時）	夜間に実施する業務はなく、システムを停止可能。 [-] 運用時間をもっと限って業務を稼働させる場合 [+] 24時間無停止やリブート処理等の短時間の停止のみを考える場合	4	若干の停止有り（9時～翌朝8時55分）	24時間無停止での運用は必要ないが、極力システムの稼働は継続させる。 [-] 夜間のアクセスは認めないなど、長時間運用を停止する場合 [+] 24時間無停止で運用する場合	5	24時間無停止	システムを停止できる時間帯が存在しない。 [-] 1日のスケジュールで定期的に運用を停止する時間帯が存在する場合

[非機能要求グレード2018 グレード表より
https://www.ipa.go.jp/sec/softwareengineering/std/ent03-b.html]

TOC

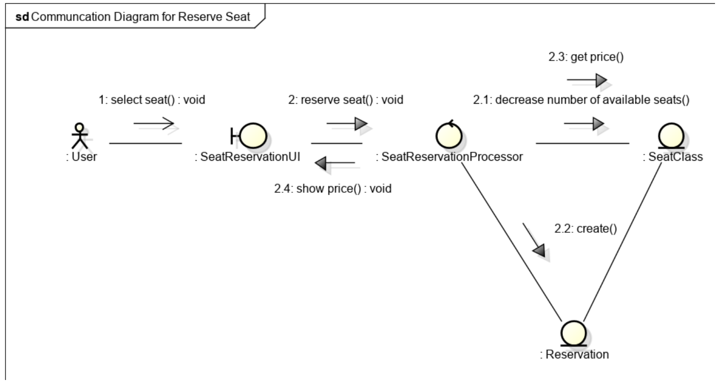
- Quality Characteristics of Software Systems
- System Analysis
- Design Principles

System Analysis

- System Analysis
  - Construct system models about “how to realize the functionality” based on domain and requirements models
  - Investigate abstract essences and do not define the implementation detail, guiding the following design activities
- Robustness Analysis
  - One of popular methods for system analysis
  - Make requirements, specifically use cases, robust by examining their realization

System Analysis: Typical Procedure

1. Analyze and define the flow to realize each use case
   - Define classes with three roles of boundary, control, and entity (this is the idea of robustness analysis)
   - Analyze interactions in each use case



2. Integrate definitions obtained for each use case into those of the whole system
   - Organize elements and relationships, typically in class diagrams

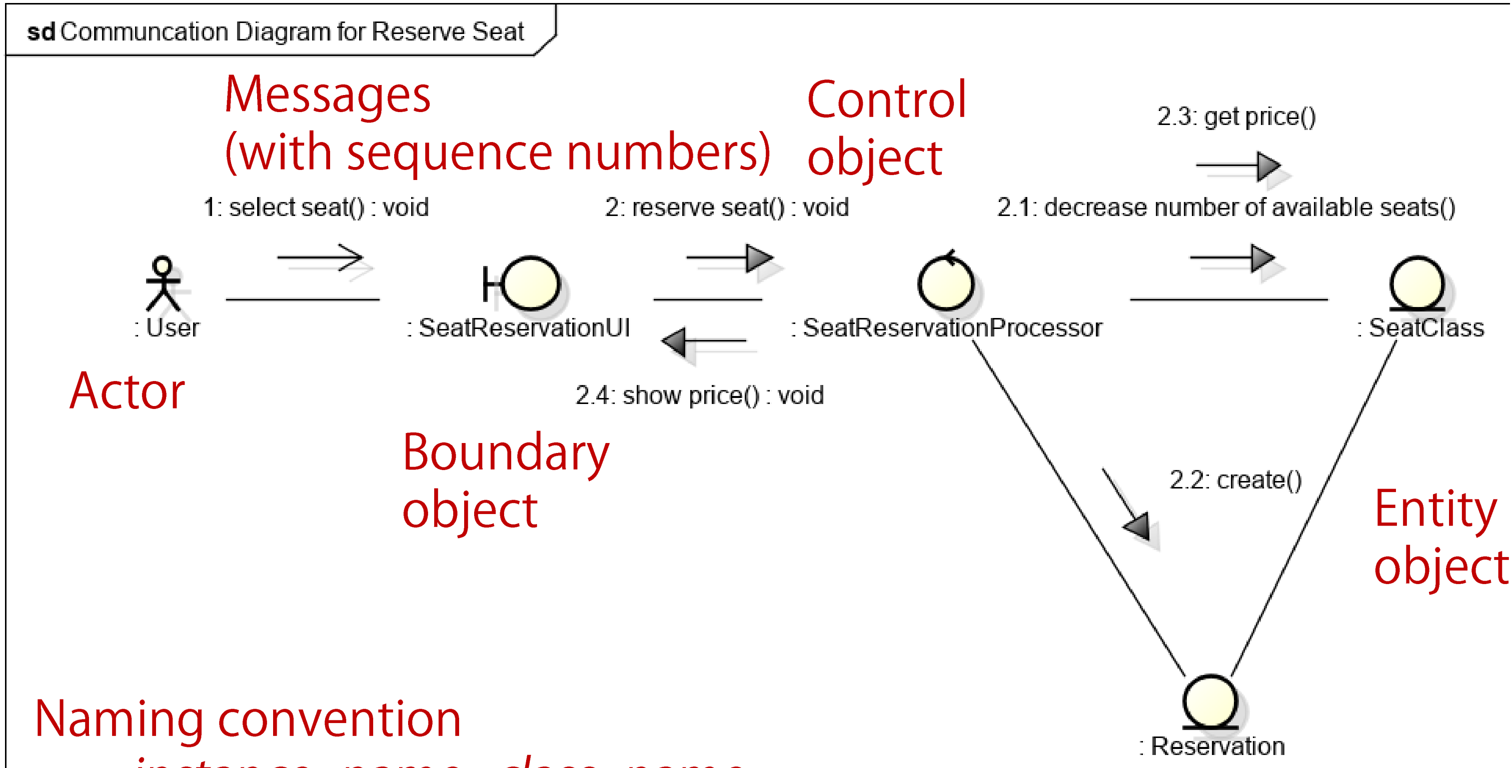
Robustness Analysis: Three Roles of Classes

- Define how to realize each use case in an abstract way with the following three elements
  - Boundary: class that serves as the interface between actors and the system
  - Control: class that works on entities or other controls classes given stimulus from boundaries
  - Entity: class that maintains information in a durable way

Demonstration

Example of Robustness Analysis

- Use case for “reserve seat”
  - Using three symbols for objects in the communication diagram



Messages
(with sequence numbers)

Control
object

Actor

Boundary
object

Entity
objects

Naming convention

instance_name : *class_name*

(instance_name may be omitted unless there are multiple objects of the same class)

Robustness Analysis: Principles

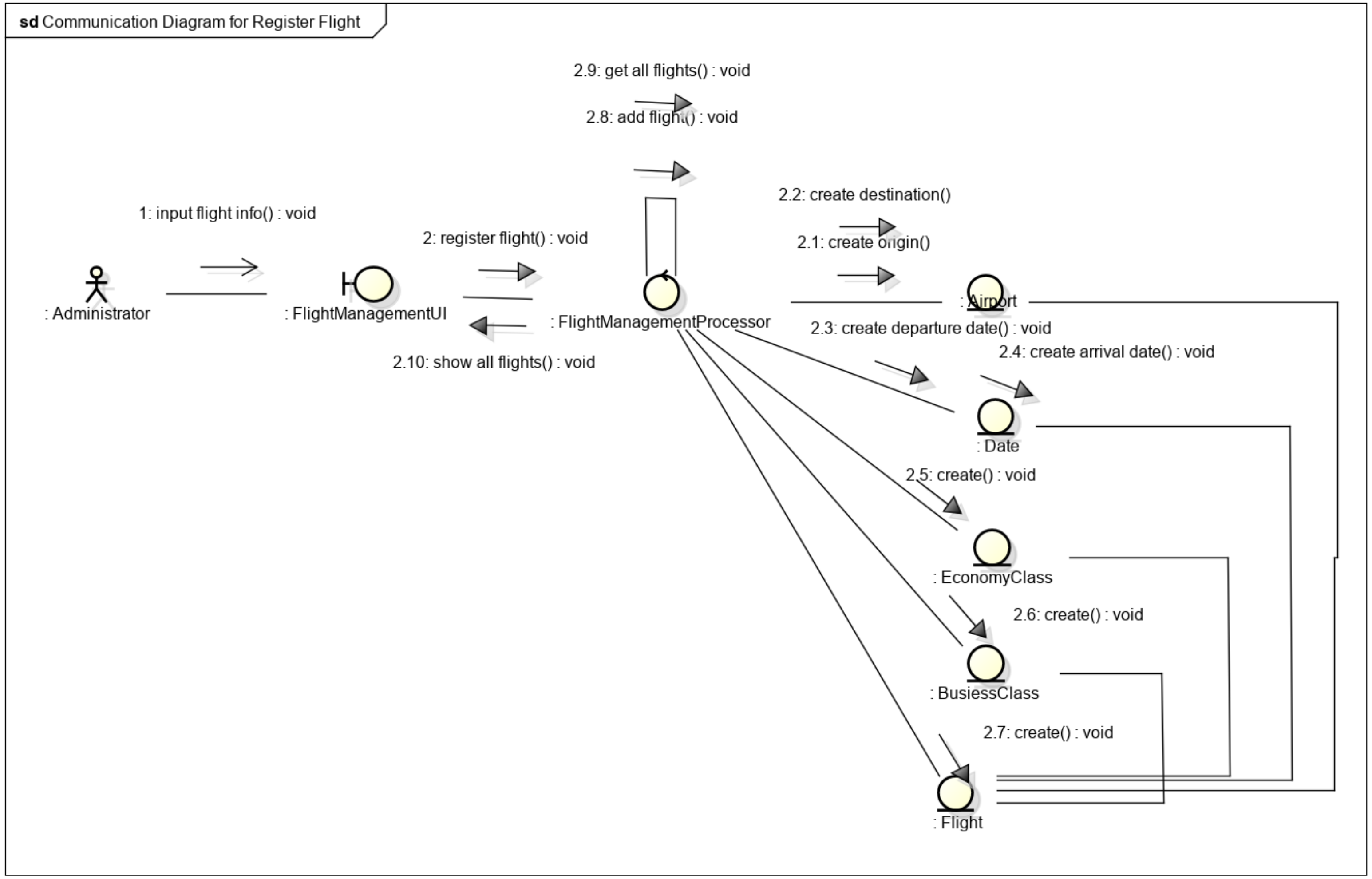
- High abstraction level
  - Do not care conventions of programming languages (e.g., naming rules, object creation, etc.)
  - Do not define data types
  - May include naïve behavior in terms of implementation (e.g., mutual invocations between two objects, without using "return" value, generally not desirable at the program level)

System Analysis (Cont’d)

- Robustness analysis was the first step, for each use case
  - We got what classes (modules/functionalities/responsibilities) are necessary in the form of exchanges messages
- Merge the class definitions obtained for each use case
  - Entity classes are generally common in multiple use cases and should match with the concepts obtained in the domain analysis
  - Abstraction level is still high
    - e.g., no design on “which object has a pointer to another” or “how each object is initialized” (unless it is the core of the use case)

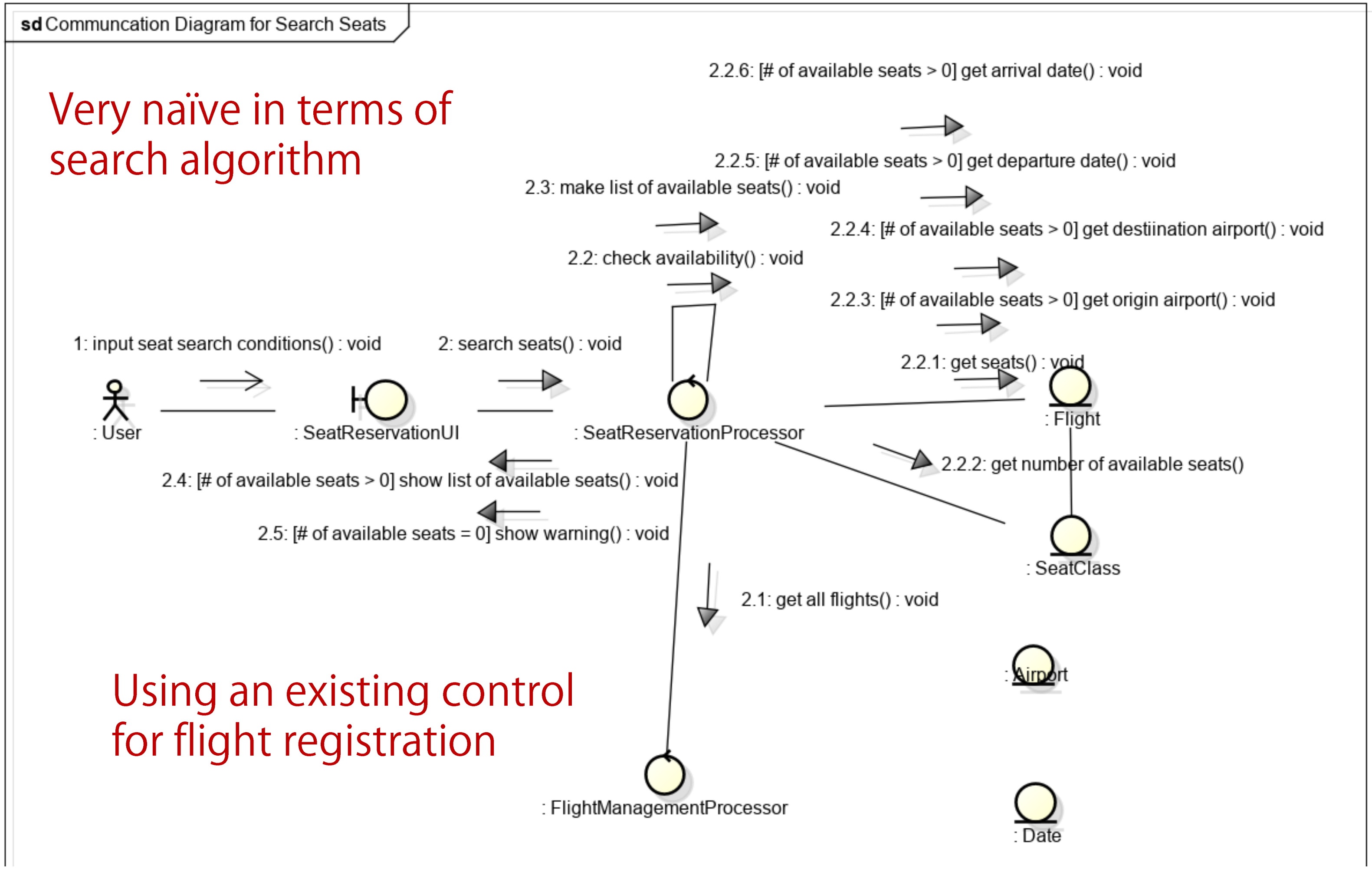
Other Examples of Robustness Analysis (1)

- Use case of “register flight”


sd Communication Diagram for Register Flight
2.9: get all flights() : void
2.8: add flight() : void
2.2: create destination()
1: input flight info() : void
2: register flight() : void
2.1: create origin()
: Administrator
: FlightManagementUI
: FlightManagementProcessor
: Airport
2.3: create departure date() : void
2.4: create arrival date() : void
2.10: show all flights() : void
: Date
2.5: create() : void
: EconomyClass
2.6: create() : void
: BusiessClass
2.7: create() : void
: Flight

Other Examples of Robustness Analysis (2)

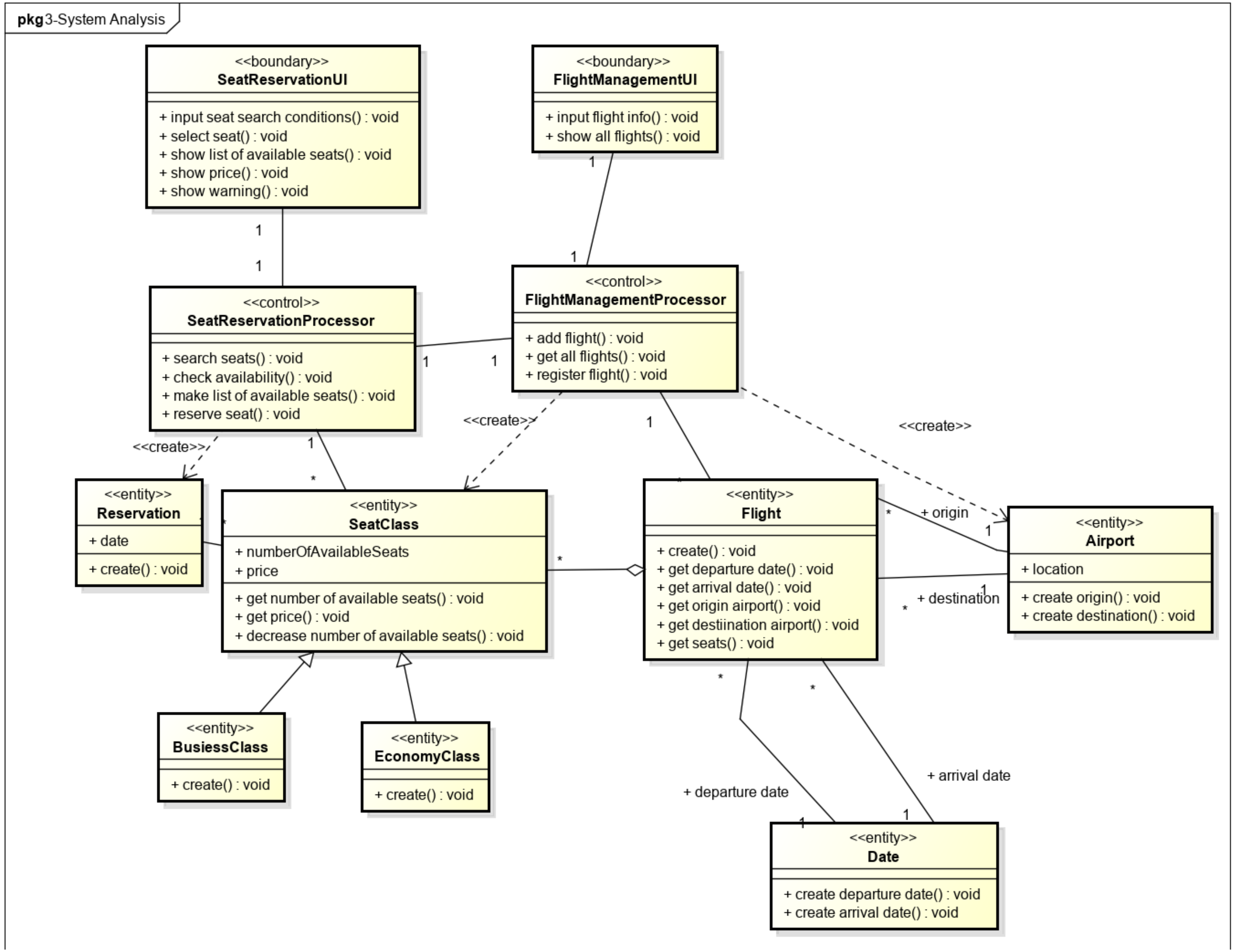
- Use case of “search seats”


sd Communcation Diagram for Search Seats
2.2.6: [# of available seats > 0] get arrival date() : void
2.2.5: [# of available seats > 0] get departure date() : void
2.3: make list of available seats() : void
2.2.4: [# of available seats > 0] get destiination airport() : void
2.2: check availability() : void
2.2.3: [# of available seats > 0] get origin airport() : void
1: input seat search conditions() : void
2: search seats() : void
2.2.1: get seats() : void
: User
: SeatReservationUI
: SeatReservationProcessor
: Flight
2.2.2: get number of available seats()
2.4: [# of available seats > 0] show list of available seats() : void
2.5: [# of available seats = 0] show warning() : void
: SeatClass
2.1: get all flights() : void
: Airport
: FlightManagementProcessor
: Date

Very naïve in terms of search algorithm

Using an existing control for flight registration

System Analysis (Integrated Class Diagram)



TOC

- Quality Characteristics of Software Systems
- System Analysis
- Design Principles

Design

- Design
  - Defines “How” to realize the requirements
  - Needs to reflect the non-functional requirements
  - Deals with the whole system (architecture) or individual parts (components)

Design Principles

- Encapsulation
- Information Hiding
- Abstraction
- Modularization
- Divide-and-Conquer
- Consideration of Cohesion and Coupling
- Separation of Concerns

Extended from [Buschmann et al., Pattern-Oriented Software Architecture, Wiley, 1996]

Encapsulation and Information Hiding

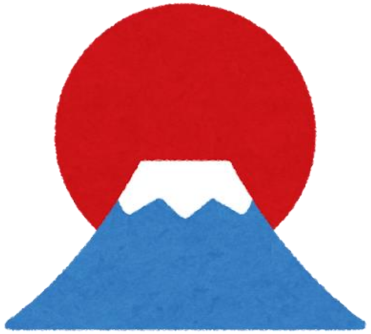
- Encapsulation（カプセル化）
  - Define an abstract element by grouping the structure and behavior and design the interface for access
  - Contributes to information hiding, abstraction, modifiability, and reusability
- Information Hiding（情報隠蔽）
  - Hide the implementation detail from the client
  - Clients do not need to know the inside
  - Providers can modify the inside without affecting clients

Abstraction

- Abstraction（抽象化）
  - One Definition: an abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer. [Booch, 1991]
  - Examples often discussed

メトロネットワーク
Metro Network

[cited from https://www.tokyometro.jp/station/]



from いらすとや

Coupling

- Coupling（結合度）
  - The upper side is considered “bad” with more dependencies between modules

Content	A module uses and alters data in another
Control	A modules communicate with another by passing a control flag to affect the behavior
Common	Two modules communicate through global data
Stamp	Two modules communicate by passing a shared structure (including redundancy)
Data	Two modules communicate by passing the minimum required data

Cohesion

- Cohesion（凝集度）
  - The upper side is considered “bad” with less relationships between elements in a module

Coincidental	Elements are unrelated
Logical	Elements have similar activities (e.g., all I/O actions)
Temporal	Elements perform in similar timing (e.g., initialization)
Procedural	Elements work sequentially (on different data) (e.g., a part split from a flow chart)
Communicational	Elements work on the same input
Informational	Elements cover all the functions on one data structure
Functional	Elements are related with each other to perform one function

Separation of Concerns

- Separation of Concerns（関心事の分離）
  - Different or unrelated obligations should be separated e.g., by allocation to different components
- Separation of Interface and Implementation

➡ Minimization of what each role of developers needs to know, as well as what changes affect

Summary

- Quality
  - Systematically investigated with characteristics or different aspects, often derived from standards/guidelines
- System Analysis / Robustness Analysis
  - Examine “how” in implementation-independent abstract models and validate requirements
- Design Principles
  - Focusing on decisions of roles/responsibilities, or boundaries, for maintainability including changeability, testability, etc.