

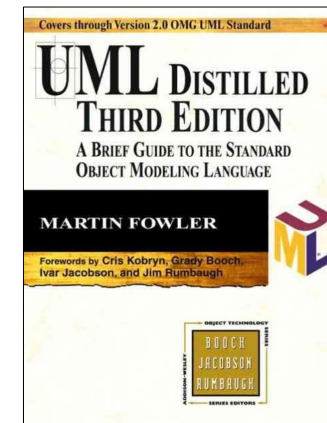
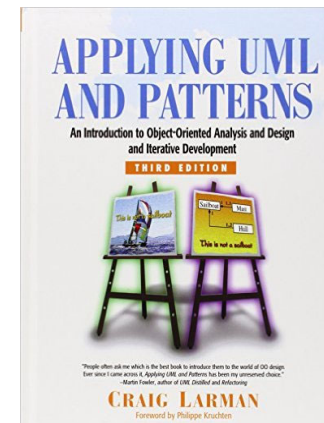
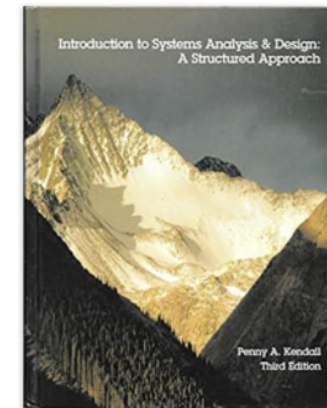
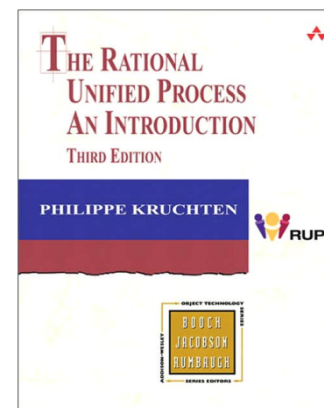
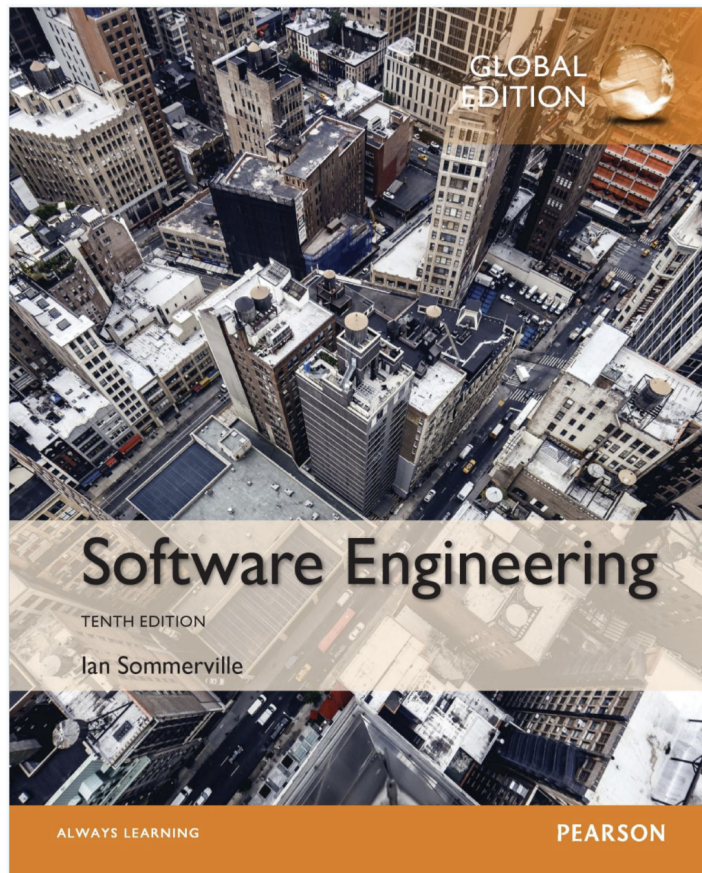
Software Engineering

JUNBEOM YOO

KONKUK University

<http://dslab.konkuk.ac.kr>

Text and References



Contents

1. **An Introduction to Software Engineering**
2. **Software Development Process**
3. **Agile Software Development**
4. **Requirements Engineering**
5. **System Modeling**
 - **Structured Analysis and Structured Design (SASD)**
6. **Architectural Design**
7. **Design and Implementation**
 - **An Introduction to UML**
 - **Object-Oriented Analysis and Design (OOAD)**
8. **Software Testing**
9. **Software Evolution**

Lists of Homework/Activities

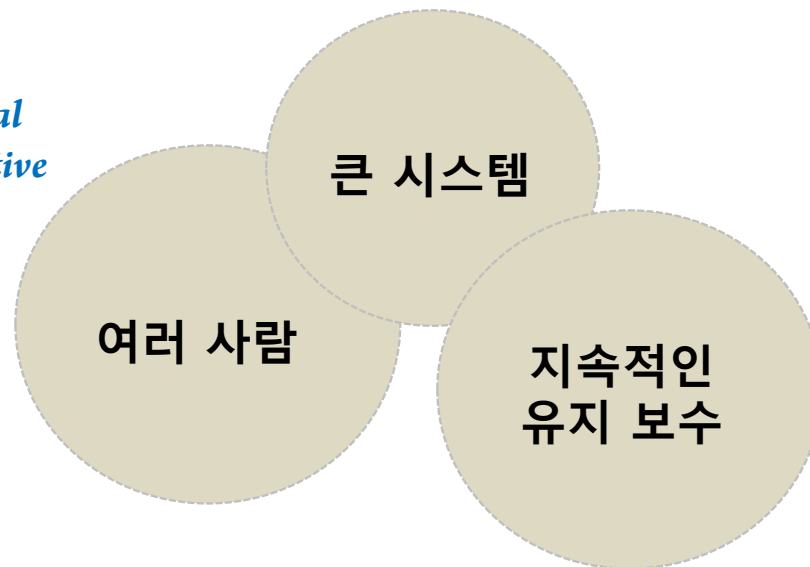
1. An Introduction to Software Engineering - Homework #1
2. Software Development Process - Homework #2
3. Agile Software Development - Homework #3
4. Requirements Engineering - Homework #4
5. System Modeling - Homework #5
 - Structured Analysis and Structured Design (SASD) - Homework #6 #7
6. Architectural Design - Homework #8
7. Design and Implementation - Homework #9
 - An Introduction to UML
 - Object-Oriented Analysis and Design (OOAD) - Homework #10
8. Software Testing - Homework #11
9. Software Evolution - Homework #12

1. An Introduction to Software Engineering

Software Engineering

- **Software engineering** is concerned with **theories**, **methods** and **tools** for professional and cost-effective software development.
 - More and more systems are software-controlled.
 - The economies of all developed nations are dependent on software.
 - Software costs often dominate computer system (hardware) costs.
 - Software costs is more to maintain than to develop.
 - For systems with a long life, maintenance costs may be several times development costs.

- *Professional*
- *Cost-Effective*



- *Large Systems*
- *Team Development*
- *Reuse*

Types of Software Products

- **Generic Software**

- Stand-alone systems that are marketed and sold to any customer who wishes to buy them
 - Most PC software : graphics programs, project management tools, CAD software
 - Software for specific markets such as appointments systems for dentists
- Software requirements specification is owned by the software developer.
 - Decisions on software change are made by the developer.

- **Customized Software**

- Software that is commissioned by a specific customer to meet their own needs
 - Embedded control systems, air traffic control software, traffic monitoring systems
- Software specification is owned by the customer for the software.
 - Customers make decisions on software changes that are required.

- **Question:**

- 은행에서 Teller가 사용하는 “은행시스템”은 어떤 SW 인가요?

Essential Attributes of Good Software Products

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because <u>software change is inevitable</u> in changing business environments.
Dependability	Software dependability includes a range of characteristics including <u>reliability</u> , <u>security</u> and <u>safety</u> . Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of <u>system resources</u> such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be <u>acceptable</u> to the type of users for which it is designed. This means that it must be <u>understandable</u> , <u>usable</u> and <u>compatible</u> with other systems that they use.

Software Engineering

- **Software engineering** is an *engineering discipline* that is concerned with *all aspects of software production* from the early stages of system specification through to maintaining the system after it has gone into use.

- “*Engineering discipline*”
 - Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints

- “*All aspects of software production*”
 - Not just technical process of development, but also project management and the development of tools, methods, etc. to support software production.

Typical Activities in Software Engineering

- **Software Specification**
 - Customers and engineers define the software to produce and the constraints on its operation.
 - **Requirements Engineering**

- **Software Development**
 - The software is designed and programmed.
 - **Architecture Design, Detailed Design and Implementation**
 - **CTIP** (Continuous Test and Integration Platform)

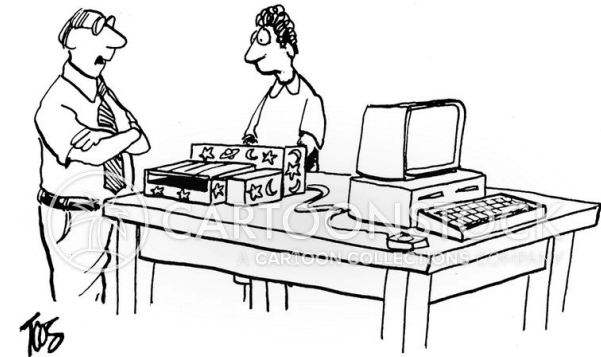
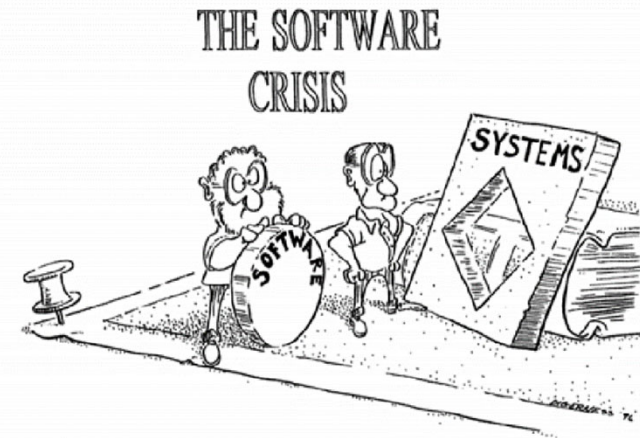
- **Software Validation**
 - The software is checked to ensure that it is what the customer requires.
 - **Software V&V** (Verification & Validation) , **Testing**

- **Software Evolution**
 - The software is modified to reflect changing customer and market requirements.
 - **Software Maintenance**

Software Project Failures

- **Software projects failures (The Software Crisis)** due to
 - **Increasing system complexity**
 - **Larger** and even more **complex** and **new** systems are required.
 - Systems must be built and delivered more **quickly**.
 - **Not use software engineering methods**
 - **New software engineering techniques** help us to build **larger**, more **complex** systems, the demands **change**.
 - But many companies **do not use** software engineering.

- **A solution** to overcome software project failures is to adopt **software engineering**.



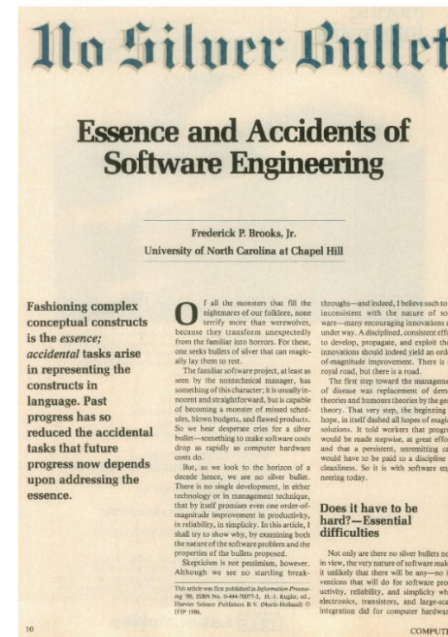
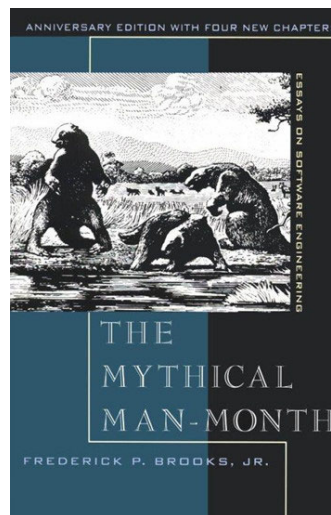
"I'm a software engineer, so I can confirm it works by magic."

Software Engineering Diversity

- **No silver bullet** for software engineering
 - There are many different types of software system.
 - There is **no universal set of software techniques** applicable to all of these.
- The software engineering methods and tools used depend on
 - the **type** of application being developed, the **requirements** of the customer, and the background of the **development team**



#Q r#v#lqj ch#r iwz duh#nqj lqhhu#lqj #ghyha#sp hqw# rxog#
 surgx.fh#lqj#rughu0r.i0p djq.lwgh#p suryhp hqw#wr#
 surj udp p lqj #surgx.fwly|w| # lwk.lq43# hduv1#
 Iuhgulfn#E#urrrnv#< ; 9##



Software Application Types

Type	Features
Stand-alone applications	Application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to network
Interactive transaction-based applications	Applications that execute on a remote computer and are accessed by users from their own PCs or terminals, including web applications such as e-commerce applications (\approx Web-based System)
Embedded control systems	Software control systems that control and manage hardware devices
Batch processing systems	Business systems that are designed to process large numbers of individual inputs to create corresponding outputs in large batches
Entertainment systems	Systems that are primarily for personal use and which are intended to entertain the user
Systems for modelling and simulation	Systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects (= M&S)
Data collection systems	Systems that collect data from their environment using a set of sensors and send that data to other systems for processing.
Systems of systems	Systems that are composed of a number of other software systems (\approx CPS)

Fundamentals of Software Engineering

- **Fundamental principles** applicable to all types of software system, irrespective of the development techniques used:
 - *“Systems should be developed using a managed and understood development process. Of course, different processes are used for different types of software.”*
 - *SLDC (Software Development Life-Cycle) , Development Process , Agile, Dev/Ops*
 - *“Dependability and performance are important for all types of systems.”*
 - *Software Quality*
 - *“Understanding and managing the software specification and requirements are important.”*
 - *Requirements Engineering*
 - *“Where appropriate, you should reuse software that has already been developed rather than write new software.”*
 - *Software Reuse , Open-Source Software*

A Newcomer : Web-based Software Engineering

- The **Web** is now a **platform** for running various application.
 - **Web services** allow application functionality to be accessed over the web.
 - **Cloud computing** enables applications run remotely on the '*cloud*'.

- **Web-based systems**
 - **Complex distributed systems**
 - The fundamental principles of software engineering are applicable to web-based systems in the same ways.
 - **Software reuse**
 - When building these systems, you think about how you can assemble them from pre-existing software components and systems.
 - **Incremental and agile development**
 - Web-based systems should be developed and delivered incrementally.
 - **Service-oriented systems**
 - Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services.
 - **Rich interfaces**
 - Interface development technologies such as AJAX and HTML5 have emerged that support the creation of rich interfaces within a web browser.

10 FAQs about Software Engineering

Question & Answer
<p><i>What is software?</i> Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.</p>
<p><i>What are the attributes of good software?</i> Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.</p>
<p><i>What is software engineering?</i> Software engineering is an engineering discipline that is concerned with all aspects of software production.</p>
<p><i>What are the fundamental software engineering activities?</i> Software specification, software development, software validation and software evolution.</p>
<p><i>What is the difference between software engineering and computer science?</i> Computer science focuses on theory and fundamentals. software engineering is concerned with the practicalities of developing and delivering useful software.</p>

10 FAQs about Software Engineering

Question & Answer
<p><i>What is the difference between software engineering and system engineering?</i> System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.</p>
<p><i>What are the key challenges facing software engineering?</i> Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.</p>
<p><i>What are the costs of software engineering?</i> Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.</p>
<p><i>What are the best software engineering techniques and methods?</i> While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. You can't, therefore, say that one method is better than another.</p>
<p><i>What differences has the web made to software engineering?</i> The web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development led to the advances in programming languages and software reuse.</p>

Homework / Activity #1

- 다음 논문을 읽고, 본인의 의견을 피력하세요.

	과거 (1960s)	현재 (2020s)	미래 (2040s)
The Software Crisis	SW개발 과제가 모두 실패함.		
Causes			
Solutions			

No Silver Bullet: Software Engineering Reloaded

Steven Fraser and Dennis Mancl

A celebratory panel took place at the 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications in Montreal. The occasion was the 20th anniversary of Fred Brooks' paper "No Silver Bullet: Essence and Accidents of Software Engineering." The paper appeared in the April 1987 *Computer*, reprinted from the *Proceedings of the IFIP 10th World Computer Congress* (North-Holland, 1986). The panelist positions appear in the *OOPSLA 2007 Conference Companion* (ACM Press).

Steve Fraser as impresario introduced the panel, which included Fred Brooks (Univ. of North Carolina at Chapel Hill), David Parnas (Univ. of Limerick), Linda Northrop (Software Engineering Inst.), Aki Namioka (Cisco Systems), Dave Thomas (Bedarra Research), Ricardo Lopez (Qualcomm), and Martin Fowler (ThoughtWorks).

Steve started by polling the audience: "How many of you have read the paper?" About three-quarters raised their hands. "I remember that it came out on the day of my doctoral defense. My thesis supervisor said it was a good thing that I didn't say anything that disagreed with Fred."

Opening statements

The first panelist to speak was Fred Brooks, who is widely recognized for his 1975 book *The Mythical Man-Month* (Addison-Wesley), a collection of essays on software project management. The book was based on Fred's experience as the project manager for the development of IBM's System/360 family of computers and then the OS/360 operating system and compilers. Fred recapped "No Silver Bullet," suggesting that software challenges are either essential or accidental. The premise of the paper was that unless the remaining accidental complexity is 90 percent of all the remaining complexity, shrinking all accidental complexity to zero still would not result in an order-of-magnitude improvement. Fred suggested that useful solutions must address inherent complexity—observing that object-oriented techniques have come closest to achieving this goal.

Next up was David Parnas, whose collected papers were published in *Software Fundamentals* (Addison-Wesley) in 2001. In particular, he's known for his *Communications of the ACM* papers "On the Criteria to Be Used in Decomposing Systems into Modules" (May 1972) and "Software Aspects of Strategic Defense Systems" (Dec.



Samples from SE Undergraduate (KU 2021)

	과거(1960s)	현재(2020s)	미래(2040s)
The software crisis	SW 개발 과제가 모두 실패함.	다양하고 복잡해진 SW 개발 실패. => No Silver Bullet (다양한 software system을 정통하는 하나의 SW engineering 기법은 없음)	빠르게 향상되는 컴퓨팅 속도를 SW가 따라가지 못할 수 있음.
Causes	1950년대 컴퓨터 하드웨어의 급속한 발전으로 한 사람이 개발하던 SW를 여러 사람이 협업하기 시작함. 그 과정에서 software crisis 발생.	메인프레임 컴퓨터에서 벗어나 다양한 용도의 소형화된 컴퓨터가 개발됨. 이제는 개발하는 어플리케이션의 타입, 개발팀의 규모에 따라 SW 엔지니어링 set이 다르게 적용되어야 하는데 기존의 단일 솔루션으로 SW engineering을 진행하고자 하여 software crisis 발생.	무어의 법칙이 깨진지는 오래지만, 18개월마다 컴퓨팅 요소의 성능이 2배로 좋아진다는 말이 있을 정도로 컴퓨팅 속도는 빨라지고 있음. SW 개발이 컴퓨팅 속도를 따라가지 못해 software crisis가 발생할 가능성이 있음.
Solutions	특정 순서를 따라 개발하면 일정 수준 이상의 퀄리티를 보장하는 waterfall model이 등장.	어플리케이션의 타입, 고객의 요구, 개발팀의 규모에 따라 SW 시스템이 다양해짐. 이에 따라 서로 다른 SW engineering 기법들을 적재적소에 적용시켜 SW 개발 과제를 해결.	이미 다중 프로세서, 양자 컴퓨터 등 더 빠르게 연산을 수행하는 컴퓨터들이 연구되거나 사용되고 있고, 더욱 다양한 플랫폼에서 컴퓨터가 활약을 하는 가운데, 이에 상응하는 SW 엔지니어링 기법이 개발되어야 할 것이라 예측.

참고문헌

Boehm, B. (2006, May). A view of 20th and 21st century software engineering. In Proceedings of the 28th international conference on Software engineering (pp. 12-29).

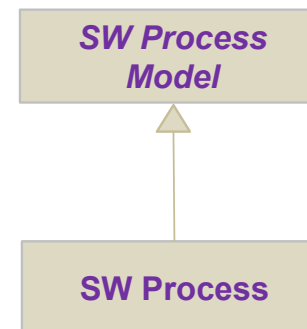
Randell, B. (1996). The 1968/69 nato software engineering reports. History of software engineering, 37.

Fraser, S., & Mancl, D. (2008). No silver bullet: Software engineering reloaded. IEEE Software, 25(1), 91-94.

2. Software Development Process

Software Process

- **Software process** is **a structured set of activities** required to develop a software system.
- Many different software processes but all involve:
 - **Specification**: defining what the system should do
 - **Design and implementation**: defining the organization of the system and implementing the system
 - **Validation**: checking that it does what the customer wants
 - **Evolution**: changing the system in response to changing customer needs.
- **Software process model** is **an abstract representation of a process**, presenting a description of a process from some perspectives.
 - Waterfall
 - Incremental
 - Evolutionary
 - Spiral
 - CBD (Component-Based Development)
 - Iterative - Agile
 - Iterative - RUP (Rational Unified Process)



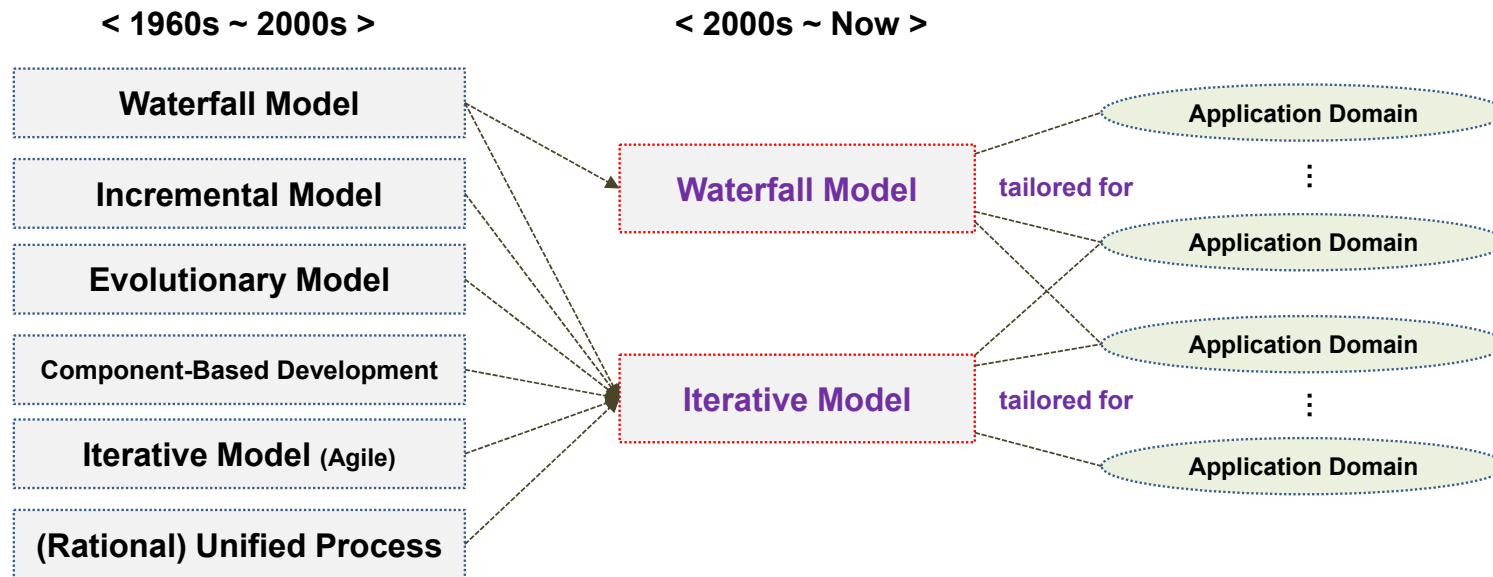
Software Process Model

Software Process Model

- **Software (Development) Process models**

- Defining a distinct set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software, systematically.
- Defining **Who** is doing **What**, **When** to do it, **How** to reach a certain goal.

= **SDLC (SW Development Life-Cycle)** models (**SW생명주기모델**)

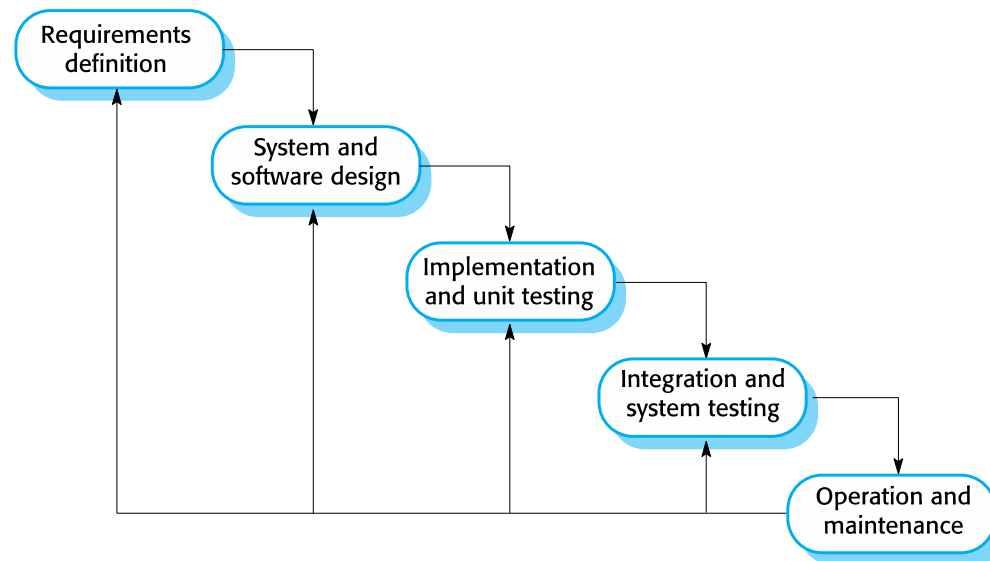


Typical SDLC Models

- Widely used **SDLC** (SW Development Life-Cycle) models:
 - **Waterfall**
 - **Incremental**
 - **Evolutionary**
 - **Spiral**
 - **CBD** (Component-Based Development)
 - **Iterative - Agile**
 - **Iterative - RUP** (Rational Unified Process)

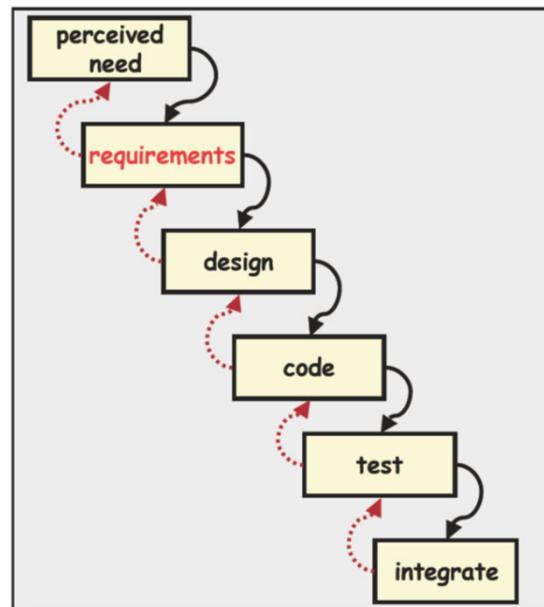
The Waterfall Model

- A **classic** software development life-cycle model proposed in 1960s
 - Suggests a systematic and sequential approach to software development
 - Has distinct/separated phases
 - In principle, a phase must be complete before moving onto the next phase.
 - Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.

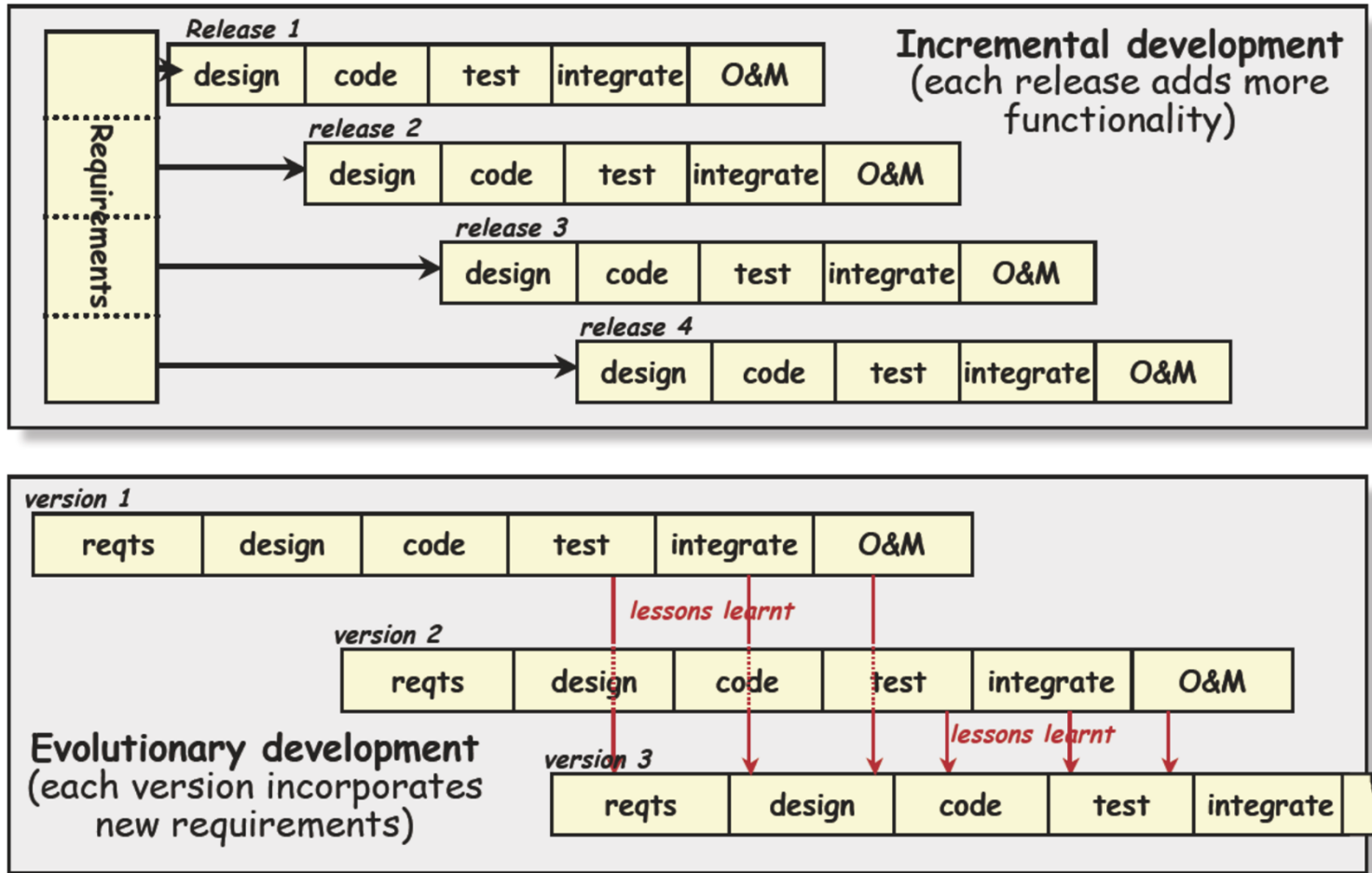


The Waterfall Model

- The waterfall model is **useful** in situations where,
 - Requirements are fixed early.
 - Work can/should proceed to completion in a linear manner.
 - **Large systems engineering projects** where a system is developed at several sites
- Only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.



The Incremental and Evolutionary Model

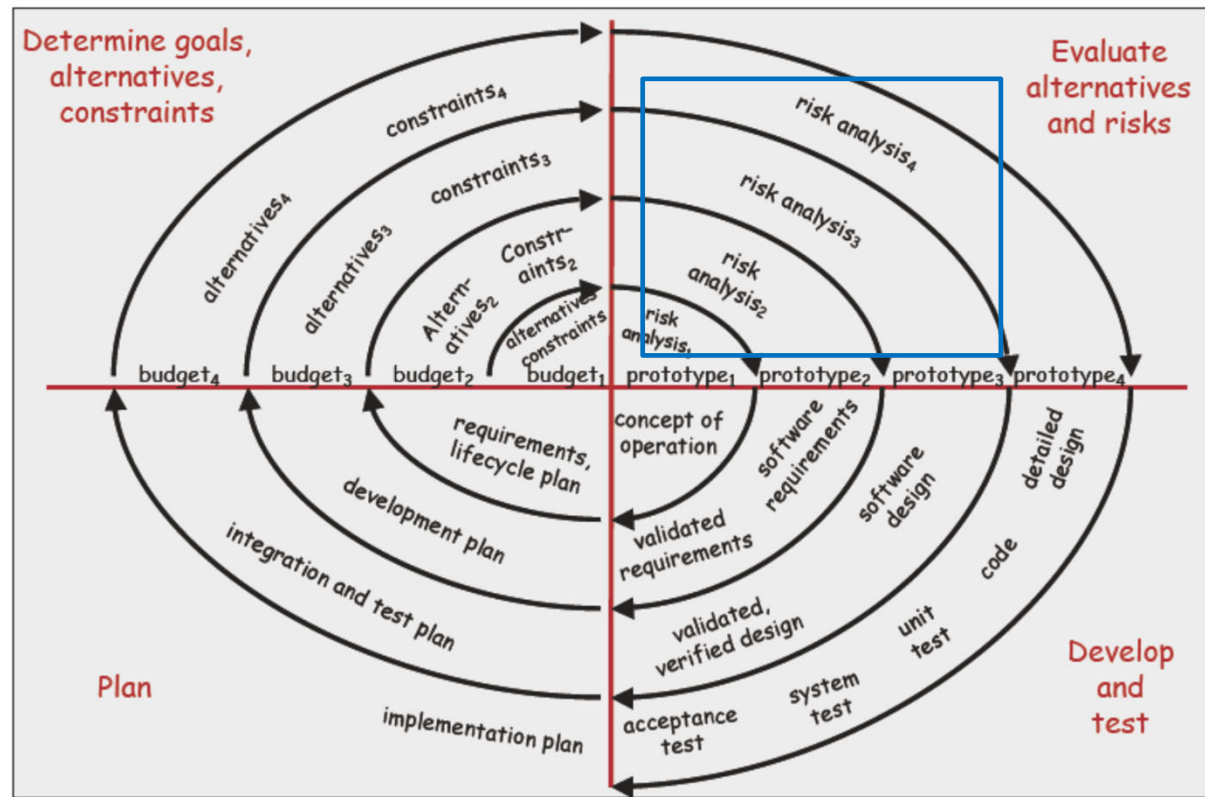


The Incremental and Evolutionary Model

- Often called “**Incremental and Evolutionary Development**”
- A number of increments are developed **in parallel**.
 - Each increment is developed independent with each other, and integrated later.
→ Incremental Development
 - The last version is the final one to deliver.
→ Evolutionary Development
 - More rapid delivery and deployment of useful software to the customer is possible.
- The process is **not visible**.
 - Many increments are developed concurrently.
 - Documentations are not easy.
- System structure **tends to degrade** as new increments are added.
 - Regular change tends to corrupt its structure.
 - Incorporating further software changes becomes increasingly difficult and costly.

The Spiral Model

- An iterative version of the waterfall model with “*risk analysis*” added



CBD (Component-Based Development)

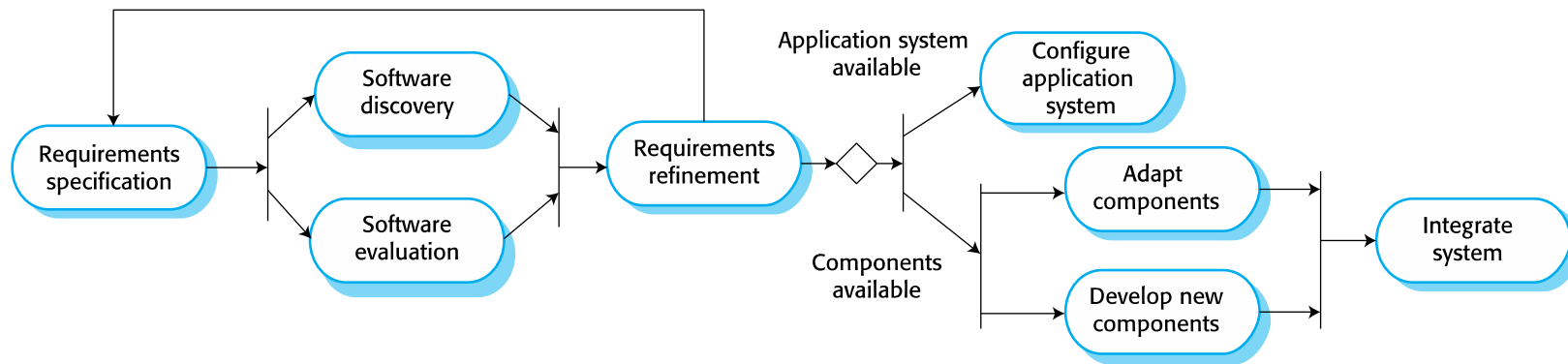
- Based on **software reuse**
 - Systems are integrated from existing components or application systems.
 - Using **COTS** (Commercial-off-the-shelf) systems/components.
 - Reused elements should be **configured** to adapt their behaviour and functionality.

- Reuse is now conceptually **the standard approach** for building many types of business systems.

- **Types of reusable software components:**
 - **Stand-alone application systems (COTS):**
 - Configured for use in a particular environment
 - **Collections of objects:**
 - Developed as a package to be integrated with **component frameworks** such as **.NET** or **J2EE**
 - **Web services:**
 - Developed according to service standards and which are available for remote invocation

CBD

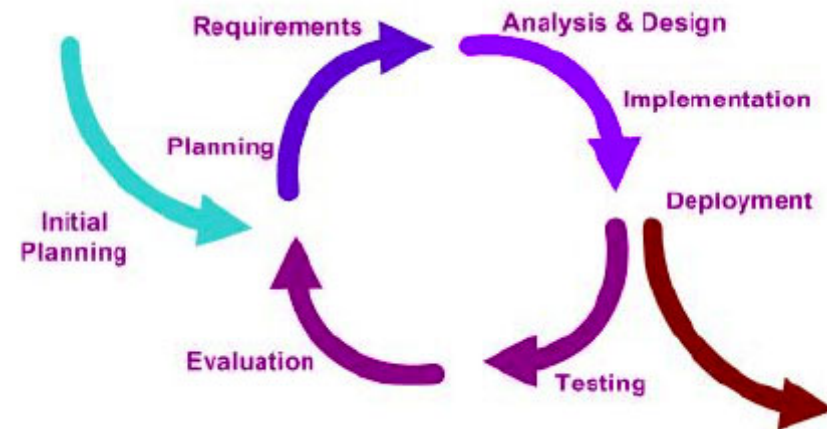
- Reuse-oriented software engineering process



- Advantages :
 - Reduced costs and risks as less software is developed from scratch.
 - Faster delivery and deployment of systems are possible.
- Disadvantages :
 - Requirements compromises are inevitable, so system may not meet real needs of users.
 - Loss of control over evolution of reused system elements

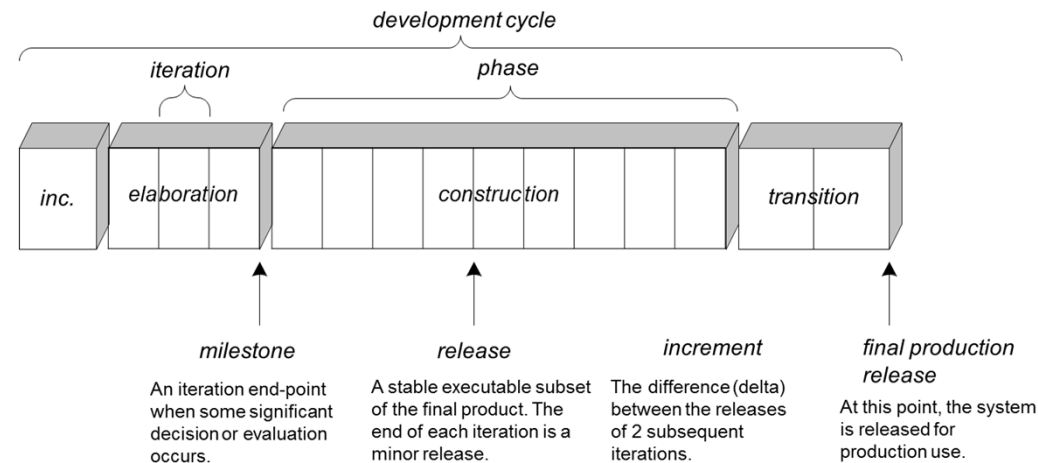
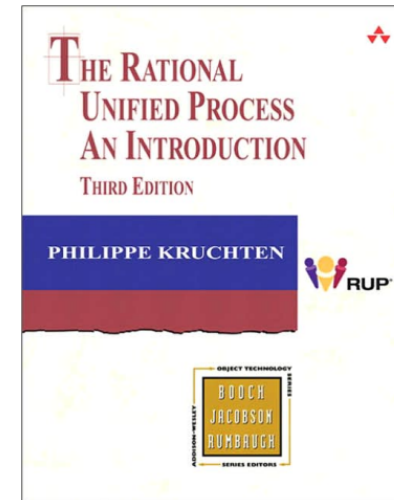
The Iterative Model - Agile

- **Agile development** is an **umbrella term** for a group of methodologies weighting rapid prototyping and rapid development experiences.
 - Lightweight in terms of documentation and process specification
 - Example: **XP** (eXtreme Programming) , **TFD** (Test First Development)
- Agile methods attributes
 - **Iterative** (several cycles)
 - **Incremental** (not delivering the product at once)
 - Actively involve **users** to establish requirements
- **Agile Manifesto**
 - **Individual** over processes and tools
 - **Working software** over documentation
 - **Customer collaboration** over contract negotiation
 - **Responding to change** over following a plan



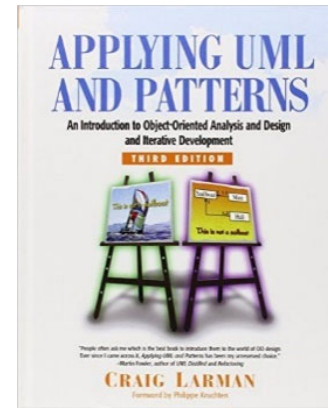
The Iterative Model - RUP

- **Rational Unified Process (RUP) or UP**
 - A software development approach that is
 - **Iterative (Incremental, Evolutionary)**
 - Each iteration includes a **small waterfall cycle (3~4 weeks)**.
 - **Risk-driven / Client-driven / Architecture-centric**
 - **Use-Case-driven**
 - A Well-defined and well-structured software engineering process
 - 4 Phases and 9 Disciplines
 - A de-facto industry standard for developing OO software



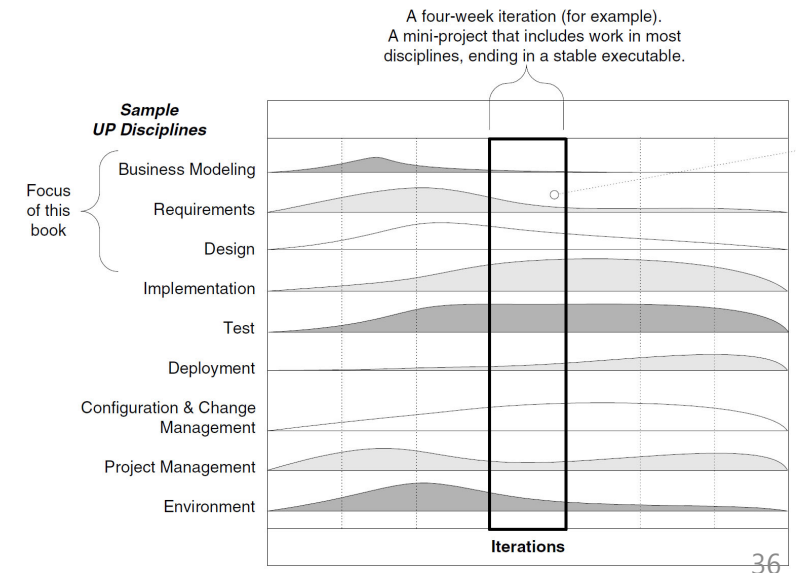
(Rational) Unified Process

- The **UP** encourages a combination of **risk-driven** and **client-driven iterative planning**.
 - To identify and drive down the **high risks (architecturally)**, and
 - To build **visible features that clients care** most about.
- Risk-driven iterative development** includes more specifically the practice of **architecture-centric iterative development**.
 - Early iterations in elaboration phase focus on building, testing, and stabilizing the core architecture.



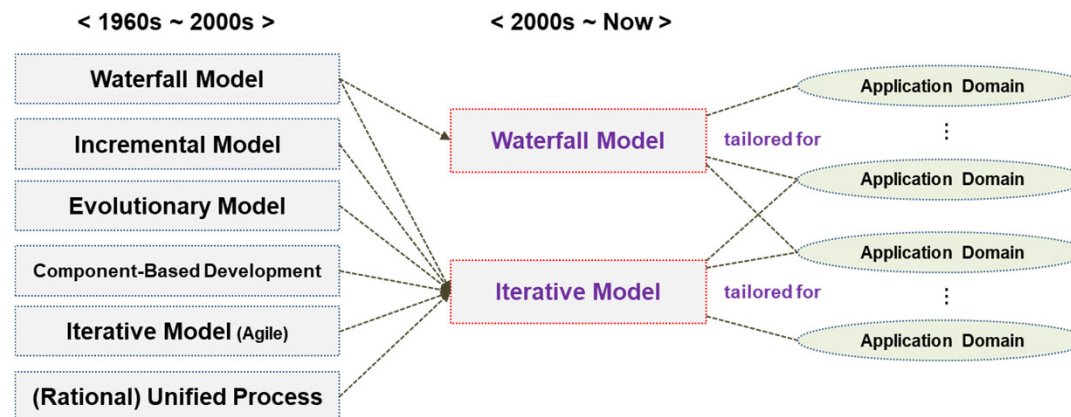
Sample Unified Process Artifacts and Timing (s-start; r-refine)

Discipline	Artifact	Iteration →			
		Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model (code, html, ...)		s	r	r



Waterfall vs. Iterative

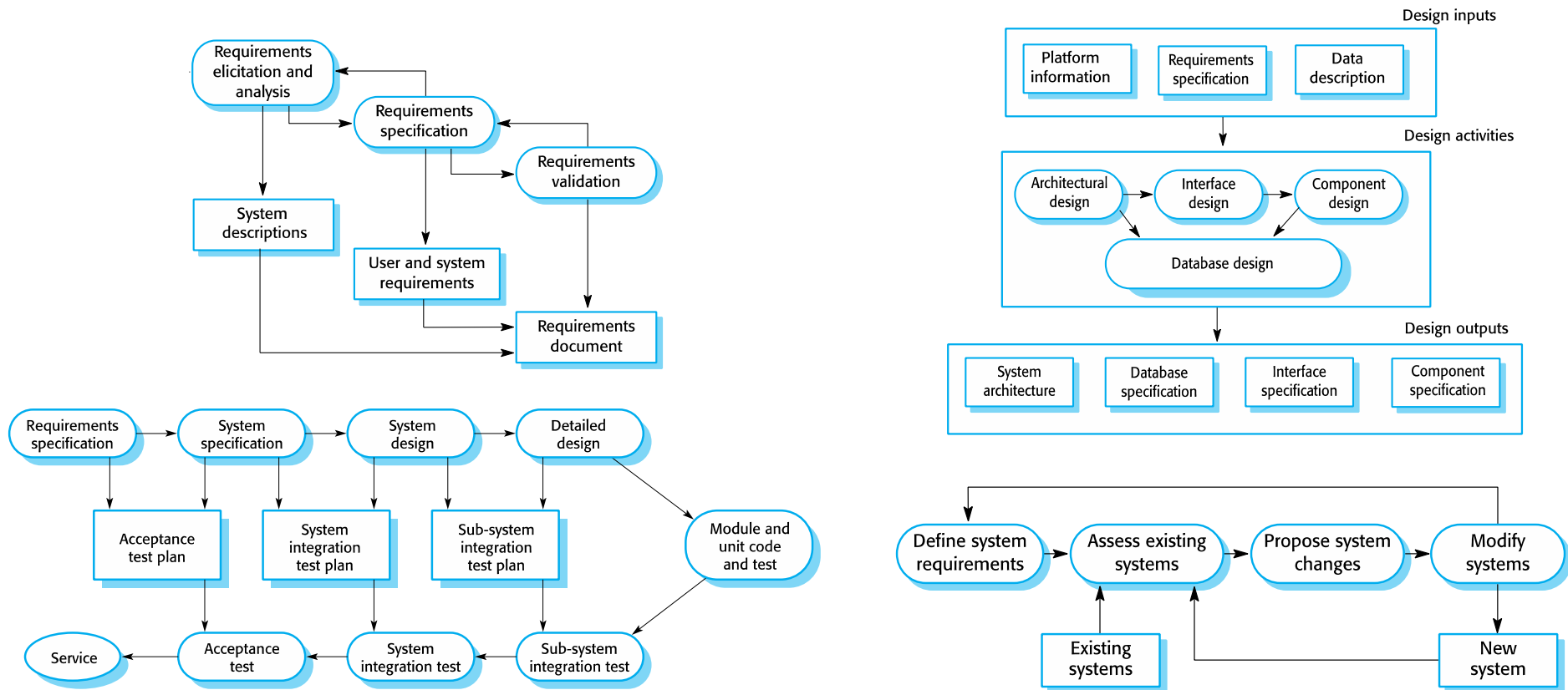
- The Waterfall process (= Plan-driven)
 - All process activities are planned in advance.
 - Progress is measured against this plan.
- The Iterative process (= Agile, UP)
 - Planning is incremental and iterative.
 - Easier to change the process to reflect changing customer requirements
- There are no right or wrong software development processes.
 - In practice, most practical processes include elements of both waterfall and iterative approaches.



Process Activities

Process Activities

- The **4 basic process activities** of **specification**, **development**, **validation** and **evolution** are organized differently in different development processes.



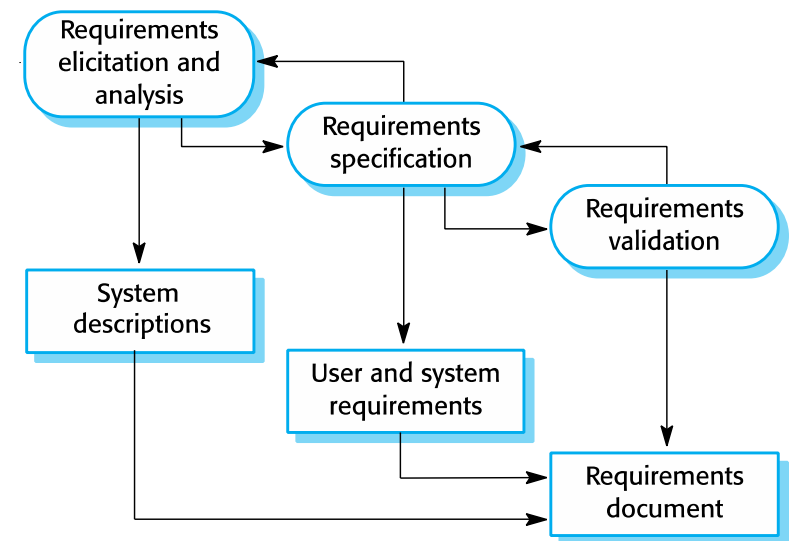
1. Requirements Engineering Process

- **RE (Requirements Engineering)**

- The process of establishing what services are required and the constraints on the system's operation and development
 - **What services:** functional requirements (**FR**)
 - **Constraints:** non-functional (quality) requirements (**NFR**)

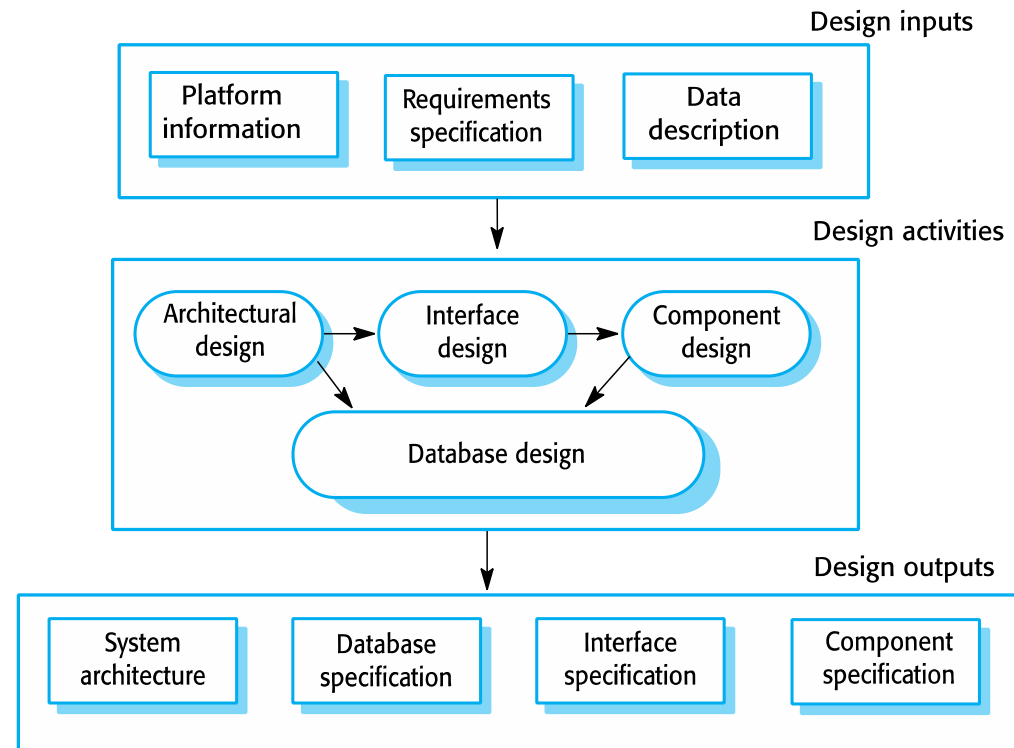
- Requirements engineering process

- Requirements elicitation and analysis
 - What do the system stakeholders require or expect from the system?
- Requirements specification
 - Defining the requirements in detail
- Requirements validation
 - Checking the validity of the requirements



2. Software Design and Implementation

- The process of converting the system specification into an executable system
 - **Software design**: Design a software structure that realizes the specification
 - **Implementation**: Translate this structure into an executable program



Design Activities

- **Architectural design**
 - Identify the overall structure of the system, the principal components (subsystems or modules), their relationships, and how they are distributed
 - **AD (Architecture Description): ISO/IEC/IEEE 42010:2011 - Systems and Software Engineering - Architecture Description**

- **Interface design**
 - Define the interfaces between system components

- **Component selection and design**
 - Search for reusable components. If unavailable, you design how it will operate

- **Database design**
 - Design the system data structures and how these are to be represented in a database

Implementation Activities

- The software is implemented either by developing programs or by configuring application system.
 - **Programming**
 - An individual activity with no standard process
 - Clean code + Refactoring + Unit Testing
 - **Debugging**
 - An activity of finding (locating) program faults and correcting these faults
 - ≠ **Testing** : An activity of detecting program faults

- **Design and implementation are interleaved activities** for most types of software system.

3. Software Validation

- **Verification and Validation (V&V)** intends to show that a system conforms to its specification (→ Verification) and meets the requirements of the system customer (→Validation).
 - Involves **(static) code checking, review** and system **(dynamic) testing**.
 - **Testing** is the most commonly used V & V activity.
 - **IEEE 1012-2016 - IEEE Standard for System, Software, and Hardware Verification and Validation**

Software Testing

- **Stages of testing**

- **Component (Unit) Testing**

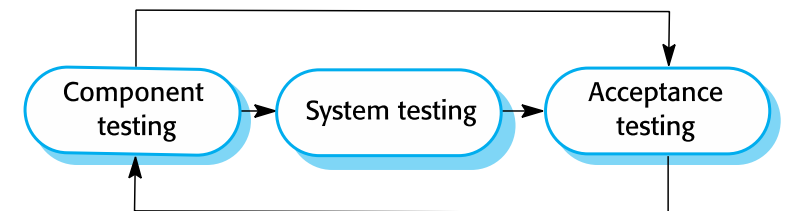
- Unit testing / Module testing
 - Individual components are tested independently.
 - Components may be functions or objects or coherent groupings of these entities.

- **System Testing**

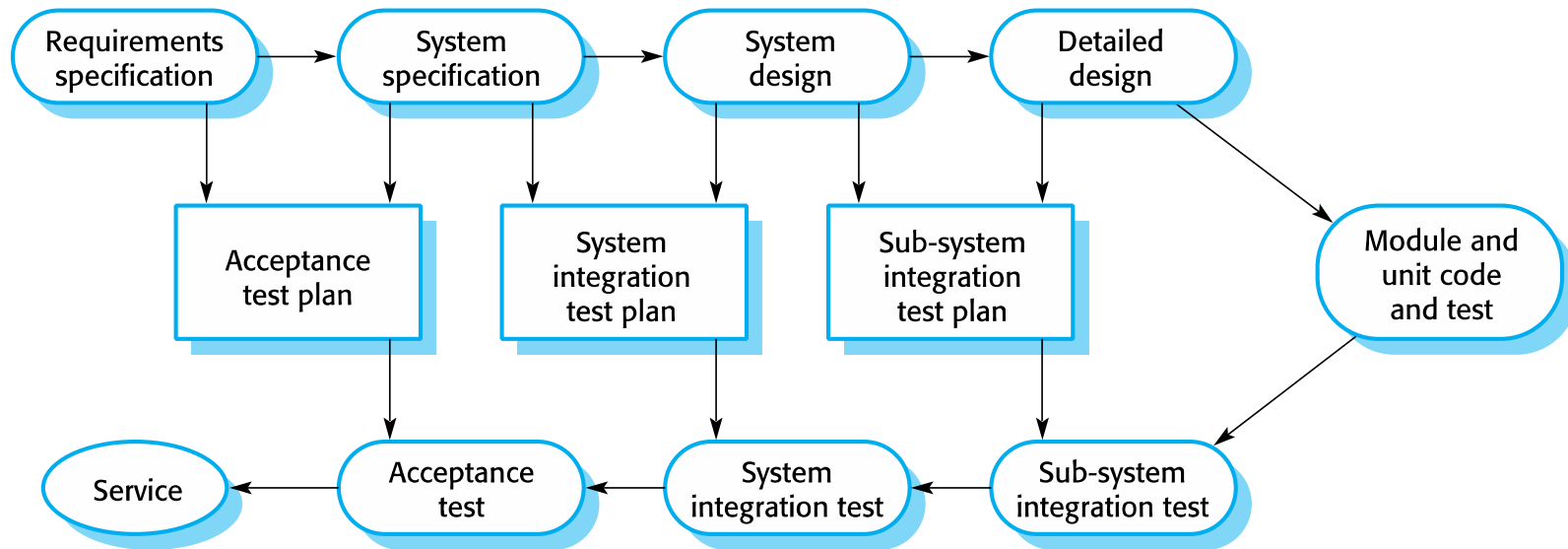
- + Integration Testing
 - Testing of the system as a whole.
 - Testing of emergent properties is particularly important.

- **Acceptance Testing**

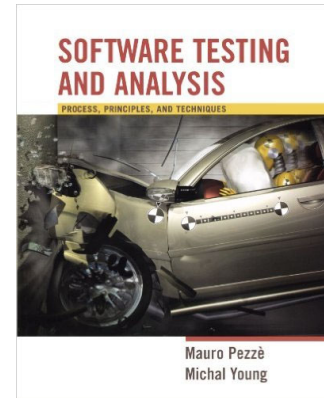
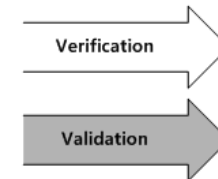
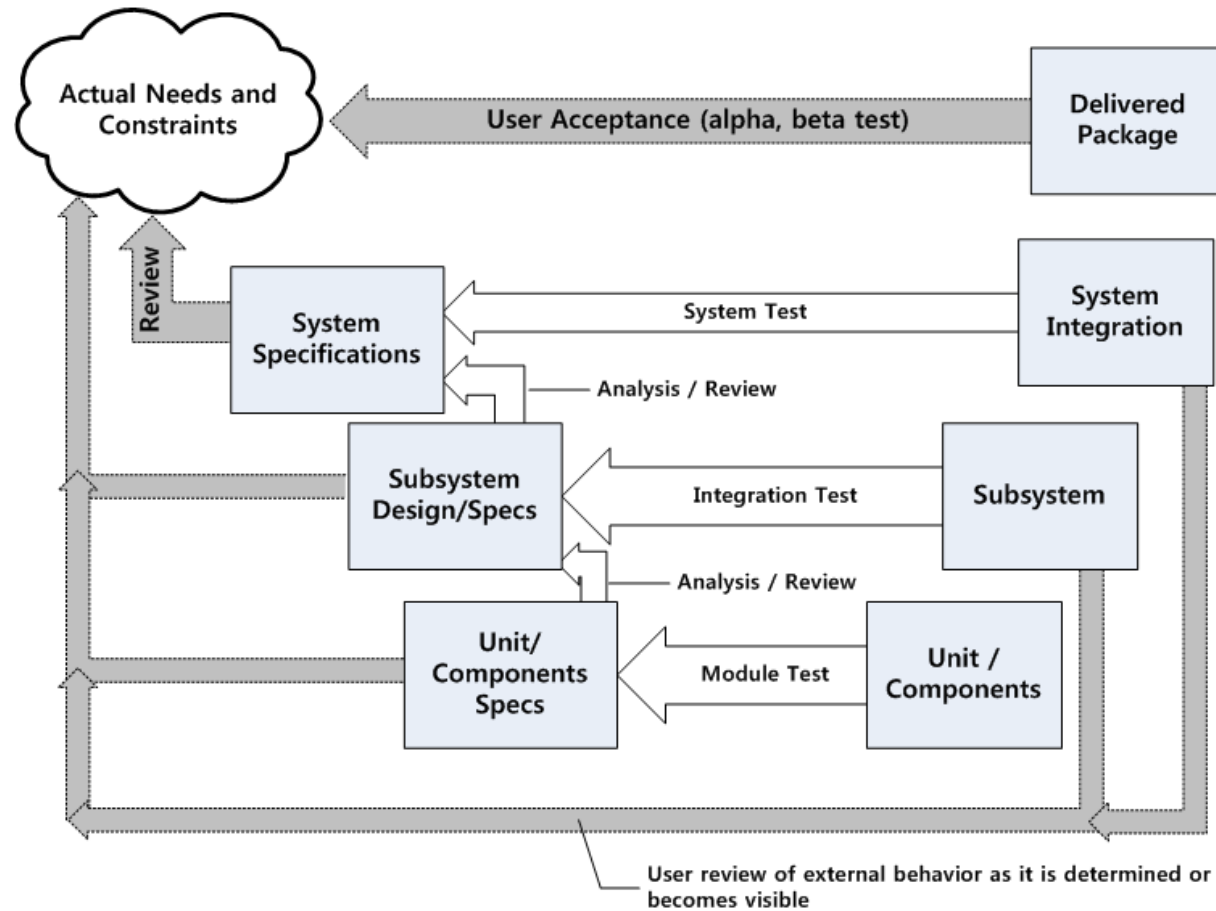
- Testing with customer data to check that the system meets the customer's needs.
 - Validation activity



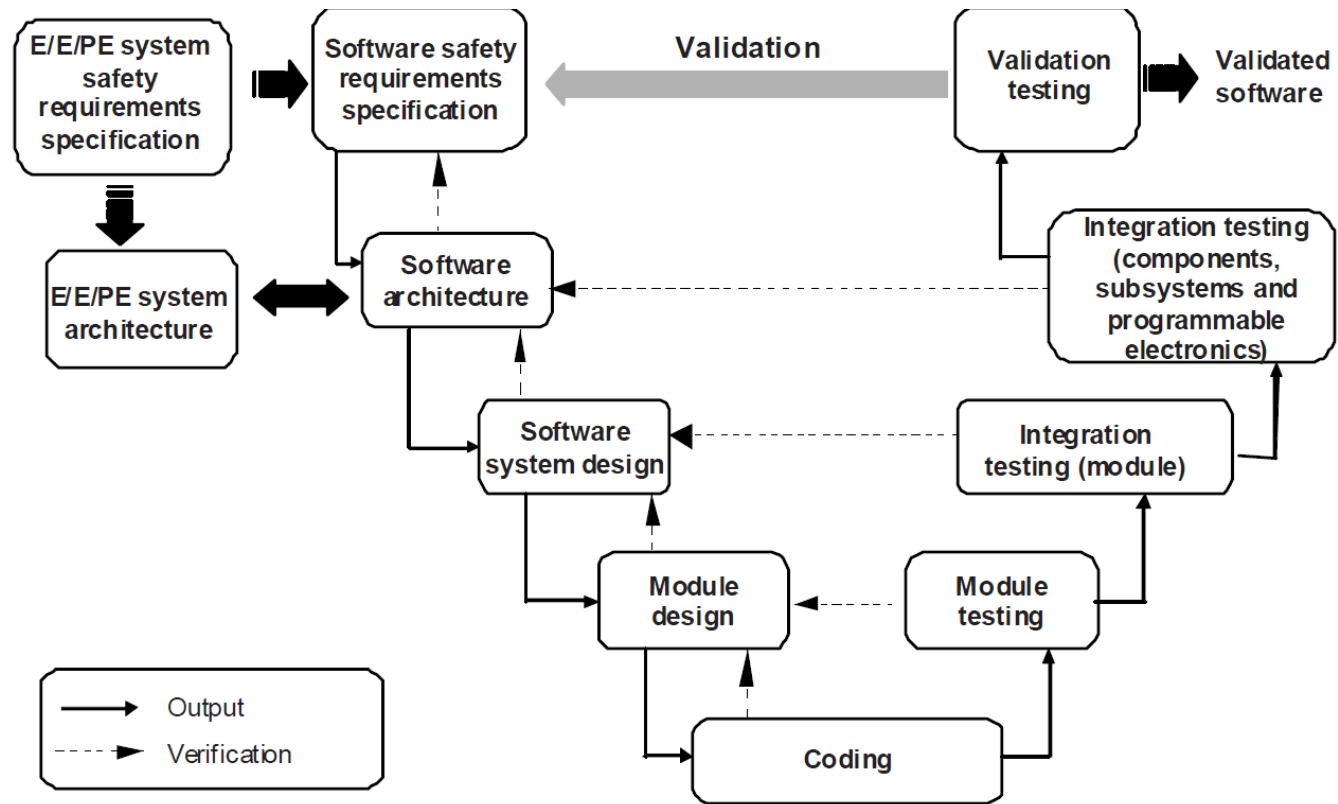
V-Model of Software Testing



V-Model of Software Testing

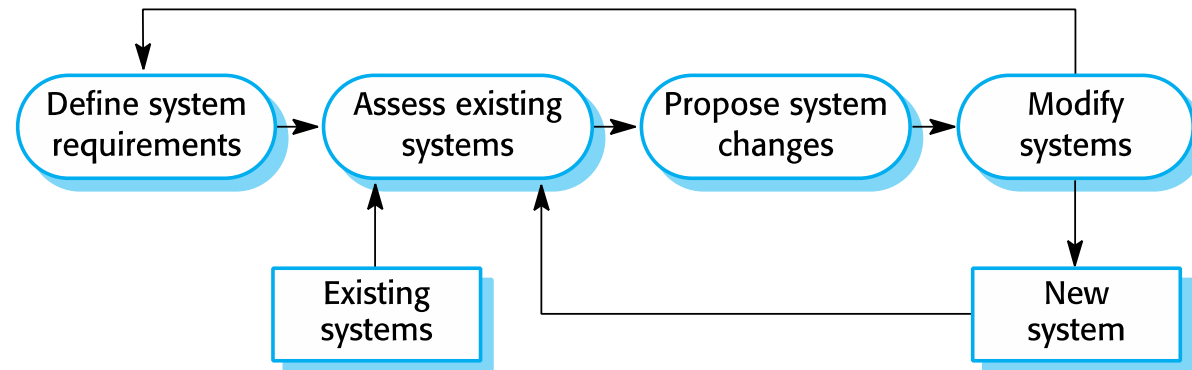


V-Model of Software Testing from IEC 61508



4. Software Evolution

- Software must also **evolve and change**, as requirements change through changing business circumstances.
 - Software is inherently flexible and can change.
 - **Maintenance**
 - **S3M (SW Maintenance Maturity Model)**



Process Improvement

Process Improvement

- **Process improvement**

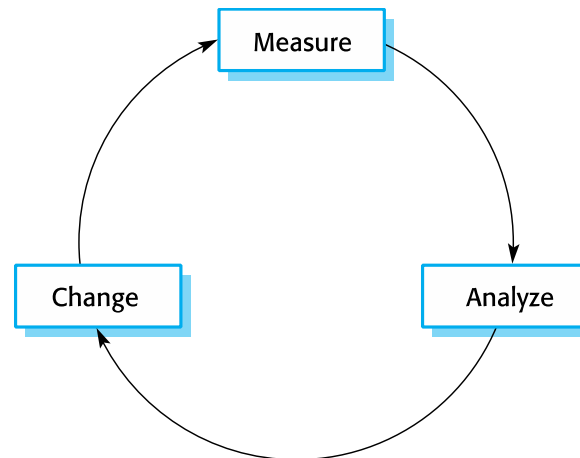
- Understanding existing processes and **changing these processes** to increase product quality and/or reduce costs and development time.

- A way of **enhancing the quality** of their software and **reducing costs**

- **The level of process maturity**, such as **CMMi**, reflects the extent to which good technical and management practice has been adopted in **organizational** software development processes.

- Activities of process improvement

- Analysis
 - Change
 - Measurement



Process Improvement Activities

- **Process analysis**
 - The current process is assessed, and process weaknesses and bottlenecks are identified.
 - Process models (process maps) that describe the process may be developed.

- **Process change**
 - Process changes are proposed to address some of the identified process weaknesses.
 - These are introduced and the cycle resumes to collect data about the effectiveness of the changes.

- **Process measurement**
 - Measure one or more attributes of the software process or product
 - These measurements forms a baseline that helps you decide if process improvements have been effective.

Process Measurement

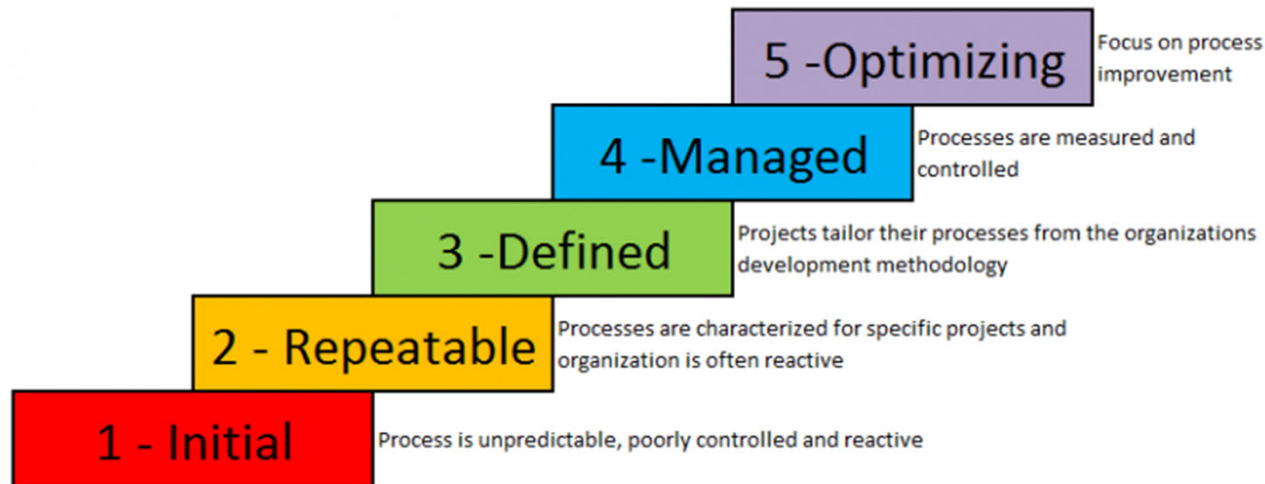
- Wherever possible, **quantitative process data** should be collected.
 - However, organizations often do not have clearly defined process standards.
 - It is very difficult as we don't know what to measure.
 - A process should be defined **before** any measurement is possible.

- The organizational objectives should drive the process improvements.

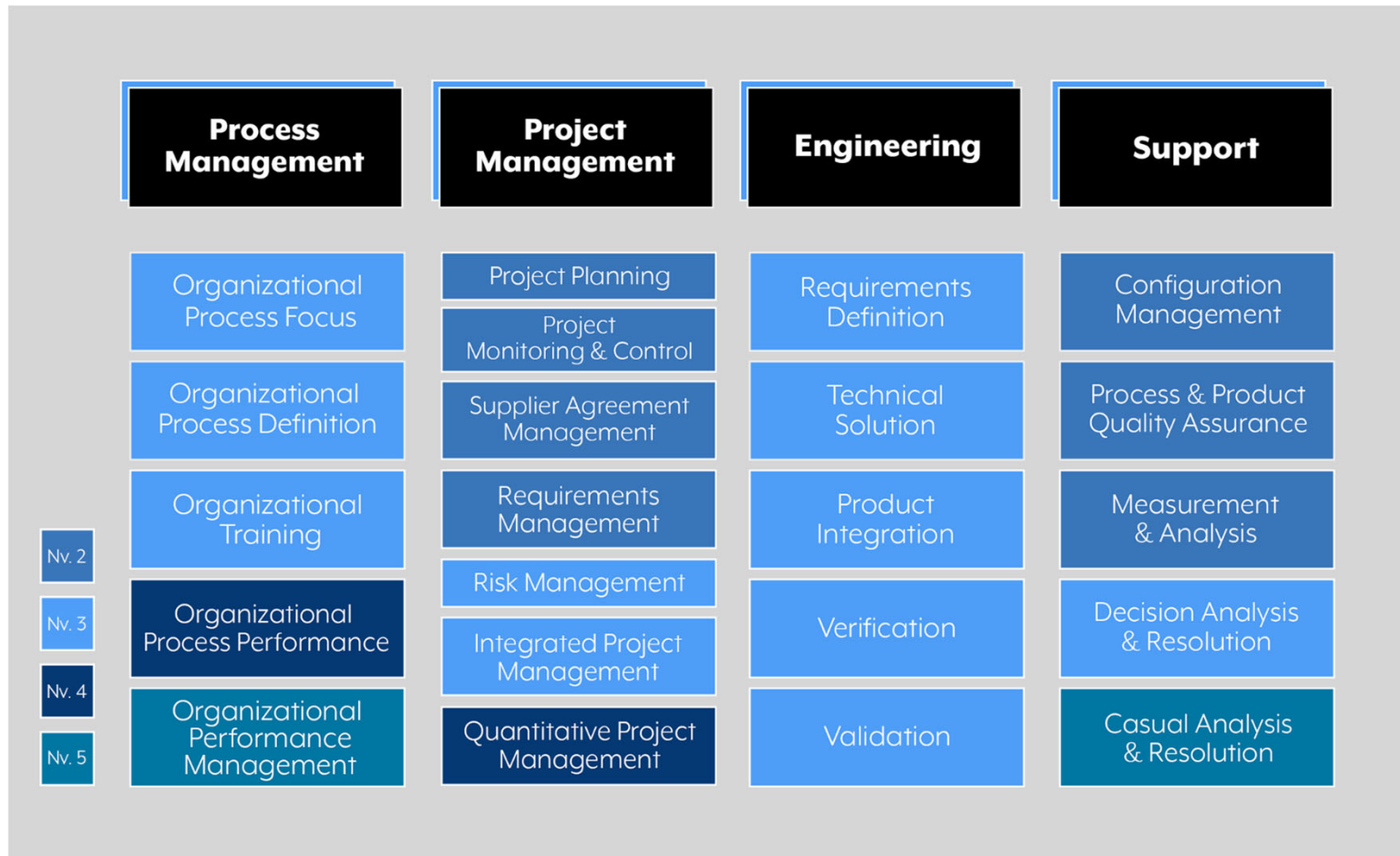
- **Examples of process metrics**
 - Time taken for process activities to be completed
 - calendar time, effort to complete an activity or process
 - Resources required for processes or activities
 - total effort in person-days
 - Number of occurrences of a particular event
 - number of defects discovered

The SEI CMMi

- **CMMi** (Capability Maturity Model Integrated) of SEI (Software Engineering Institute) in CMU
 1. **Initial** : Essentially **uncontrolled**
 2. **Repeatable** : **Product (Project) management** procedures are defined and used.
 3. **Defined** : **Process management** procedures and strategies are defined and used.
 4. **Managed** : **Quality management** strategies are defined and used.
 5. **Optimizing** : **Process improvement** strategies are defined and used.



22 Processes in CMMi



22 Processes in CMMi 1.3

Capability Maturity Model Integration® for Development Version 1.3 (CMMI-DEV-V1.3®)												
Level	Process	Goal	Practice	Subpractice	Work Product	Concept	Detailed	Modular	Info Map	Usable	Evidence	Total
1 - Initial	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2 - Managed	7	15	54	231	231	100%	100%	100%	100%	100%	100%	100%
3 - Defined	11	26	86	411	411	100%	100%	100%	100%	100%	100%	100%
4 - Quant. Mgd.	2	3	12	66	66	100%	100%	100%	100%	100%	100%	100%
5 - Optimizing	2	5	15	71	71	100%	100%	100%	100%	100%	100%	100%
Total	22	49	167	779	779	100%	100%	100%	100%	100%	100%	100%
1 - Initial	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Requirements Management (REQM)	Manage Requirements	Understand Requirements	Establish criteria for distinguishing appropriate requirements providers	Lists of criteria for distinguishing appropriate requirements providers	1	2	4	8	16	32	100%
				Establish objective criteria for the evaluation and acceptance of requirements	Criteria for evaluation and acceptance of requirements	1	2	4	8	16	32	100%
				Analyze requirements to ensure that established criteria are met	Results of analyses against criteria	1	2	4	8	16	32	100%
				Reach an understanding of requirements with requirements providers so that project participants can commit to them	A set of approved requirements	1	2	4	8	16	32	100%
			Obtain Commitment to Requirements	Assess the impact of requirements on existing commitments	Requirements impact assessments	1	2	4	8	16	32	100%
				Negotiate and record commitments	Documented commitments to requirements and requirements changes	1	2	4	8	16	32	100%
			Manage Requirements Changes	Document all requirements and requirements changes that are given to or generated by the project	Requirements change requests	1	2	4	8	16	32	100%
				Maintain a requirements change history, including the rationale for changes	Requirements change impact reports	1	2	4	8	16	32	100%
			Maintain Bidirectional Traceability of Requirements	Evaluate the impact of requirement changes from the standpoint of relevant stakeholders	Requirements status	1	2	4	8	16	32	100%
				Make requirements and change data available to the project	Requirements database	1	2	4	8	16	32	100%
				Maintain requirements traceability to ensure that the source of lower level (i.e., derived) requirements is documented	Requirements traceability matrix	1	2	4	8	16	32	100%
			Ensure Alignment Between Project Work and Requirements	Maintain requirements traceability from a requirement to its derived requirements and allocation to work products	Requirements tracking system	1	2	4	8	16	32	100%
				Generate a requirements traceability matrix	Requirements traceability report	1	2	4	8	16	32	100%
				Review project plans, activities, and work products for consistency with requirements and changes made to them	Documentation of inconsistencies between requirements and project plans and work products, including sources and conditions	1	2	4	8	16	32	100%
	Identify the source of the inconsistency (if any)	Sources of inconsistency		1	2	4	8	16	32	100%		
	Establish Estimates	Estimate the Scope of the Project		Identify any changes that should be made to plans and work products resulting from changes to the requirements baseline	Changes to resolve inconsistencies	1	2	4	8	16	32	100%
				Initiate any necessary corrective actions	Corrective actions	1	2	4	8	16	32	100%
				Develop a WBS	WBS	1	2	4	8	16	32	100%
				Define the work packages in sufficient detail so that estimates of project tasks, responsibilities, and schedule can be specified	Task descriptions	1	2	4	8	16	32	100%
		Establish Estimates of Work Product and Task Attributes		Identify products and product components to be externally acquired	Product and component list	1	2	4	8	16	32	100%
				Identify work products to be reused	Reusable work product list	1	2	4	8	16	32	100%
				Determine the technical approach for the project	Technical approach	1	2	4	8	16	32	100%
Use appropriate methods to determine the attributes of the work products and tasks to be used to estimate resource requirements				Estimating models and inputs	1	2	4	8	16	32	100%	
			Estimate the attributes of work products and tasks	Attribute estimates	1	2	4	8	16	32	100%	

CMMI 2.1 (SE/SW/IPPD/SS) Process Evaluation Checklist

CMMI SM (SE/SW/IPPD/SS) Process Evaluation Checklist										
Level	Process Area	Specific Goal	Specific Practice	Concept	Detailed	Modular	Formatted	Usable	Evidence	Total
1 - Initial	Requirements Management	Manage Requirements	Obtain an Understanding of Requirements	1	2	4	8	16	32	100%
			Obtain Commitment to Requirements	1	2	4	8	16	32	100%
Manage Requirements Changes			1	2	4	8	16	32	100%	
Maintain Bidirectional Traceability of Requirements			1	2	4	8	16	32	100%	
Identify Inconsistencies Between Project Work and Requirements			1	2	4	8	16	32	100%	
Project Planning		Establish Estimates	Estimate the Scope of the Project	1	2	4	8	16	32	100%
			Establish Estimates of Work Product and Task Attributes	1	2	4	8	16	32	100%
			Define Project Life Cycle	1	2	4	8	16	32	100%
			Determine Estimates of Effort and Cost	1	2	4	8	16	32	100%
			Establish the Budget and Schedule	1	2	4	8	16	32	100%
	Develop a Project Plan	Identify Project Risks	1	2	4	8	16	32	100%	
		Plan for Data Management	1	2	4	8	16	32	100%	
		Plan for Project Resources	1	2	4	8	16	32	100%	
		Plan for Needed Knowledge and Skills	1	2	4	8	16	32	100%	
		Plan Stakeholder Involvement	1	2	4	8	16	32	100%	
Project Monitoring and Control	Obtain Commitment to the Plan	Establish the Project Plan	1	2	4	8	16	32	100%	
		Review Plans that Affect the Project	1	2	4	8	16	32	100%	
		Reconcile Work and Resource Levels	1	2	4	8	16	32	100%	
		Obtain Plan Commitment	1	2	4	8	16	32	100%	
		Monitor Project Planning Parameters	1	2	4	8	16	32	100%	
	Monitor Project Against Plan	Monitor Commitments	1	2	4	8	16	32	100%	
		Monitor Project Risks	1	2	4	8	16	32	100%	
		Monitor Data Management	1	2	4	8	16	32	100%	
		Monitor Stakeholder Involvement	1	2	4	8	16	32	100%	
		Conduct Progress Reviews	1	2	4	8	16	32	100%	
Supplier Agreement Management	Manage Corrective Action to Closure	Conduct Milestone Reviews	1	2	4	8	16	32	100%	
		Analyze Issues	1	2	4	8	16	32	100%	
		Take Correction Action	1	2	4	8	16	32	100%	
		Manage Corrective Action	1	2	4	8	16	32	100%	
		Establish Supplier Agreements	1	2	4	8	16	32	100%	
	Satisfy Supplier Agreements	Determine Acquisition Type	1	2	4	8	16	32	100%	
		Select Suppliers	1	2	4	8	16	32	100%	
		Establish Supplier Agreements	1	2	4	8	16	32	100%	
		Review COTS Products	1	2	4	8	16	32	100%	
		Execute the Supplier Agreement	1	2	4	8	16	32	100%	
Measurement and Analysis	Accept the Acquired Product	Transition Products	1	2	4	8	16	32	100%	
		Align Measurement and Analysis Activities	1	2	4	8	16	32	100%	
		Establish Measurement Objectives	1	2	4	8	16	32	100%	
		Specify Measures	1	2	4	8	16	32	100%	
		Specify Data Collection and Storage Procedures	1	2	4	8	16	32	100%	
	Provide Measurement Results	Specify Analysis Procedures	1	2	4	8	16	32	100%	
		Collect Measurement Data	1	2	4	8	16	32	100%	
		Analyze Measurement Data	1	2	4	8	16	32	100%	
		Store Data and Results	1	2	4	8	16	32	100%	
		Communicate Results	1	2	4	8	16	32	100%	
Process and Product Quality Assurance	Transition Products	Objectively Evaluate Processes	1	2	4	8	16	32	100%	
		Objectively Evaluate Work Products and Services	1	2	4	8	16	32	100%	
		Communicate and Ensure Resolution of Noncompliance Issues	1	2	4	8	16	32	100%	
		Establish Records	1	2	4	8	16	32	100%	
		Identify Configuration Items	1	2	4	8	16	32	100%	
	Establish Baselines	Establish a Configuration Management System	1	2	4	8	16	32	100%	
		Create or Release Baselines	1	2	4	8	16	32	100%	
		Track and Control Changes	1	2	4	8	16	32	100%	
		Track Change Requests	1	2	4	8	16	32	100%	
		Control Configuration Items	1	2	4	8	16	32	100%	
Establish Integrity	Establish Configuration Management Records	1	2	4	8	16	32	100%		
	Perform Configuration Audits	1	2	4	8	16	32	100%		
	CMMI Level 2 Score	100%	100%	100%	100%	100%	100%	100%		

	Requirements Development	Develop Customer Requirements	Collect Stakeholder Needs	1	2	4	8	16	32	100%
			Elicit Needs	1	2	4	8	16	32	100%
			Develop the Customer Requirements	1	2	4	8	16	32	100%
		Develop Product Requirements	Establish Product and Product-Component Requirements	1	2	4	8	16	32	100%
			Allocate Product Component Requirements	1	2	4	8	16	32	100%
			Identify Interface Requirements	1	2	4	8	16	32	100%
		Analyze and Validate Requirements	Establish Operational Concepts and Scenarios	1	2	4	8	16	32	100%
			Establish a Definition of Required Functionality	1	2	4	8	16	32	100%
			Analyze Requirements	1	2	4	8	16	32	100%
			Analyze Requirements to Achieve Balance	1	2	4	8	16	32	100%
			Validate Requirements	1	2	4	8	16	32	100%
		Technical Solution	Select Product-Component Solutions	Develop Alternative Solutions and Selection Criteria	1	2	4	8	16	32
	Develop Detailed Alternative Solutions and Selection Criteria			1	2	4	8	16	32	100%
	Evolve Operational Concepts and Scenarios			1	2	4	8	16	32	100%
	Select Product-Component Solutions			1	2	4	8	16	32	100%
	Develop the Design		Design the Product or Product Component	1	2	4	8	16	32	100%
			Establish a Technical Data Package	1	2	4	8	16	32	100%
			Establish Interface Descriptions	1	2	4	8	16	32	100%
			Design Interfaces Using Criteria	1	2	4	8	16	32	100%
	Implement the Product Design		Perform Make, Buy, or Reuse Analyses	1	2	4	8	16	32	100%
			Implement the Design	1	2	4	8	16	32	100%
			Develop Product Support Documentation	1	2	4	8	16	32	100%
	Product Integration	Prepare for Product Integration	Determine Integration Sequence	1	2	4	8	16	32	100%
			Establish the Product Integration Environment	1	2	4	8	16	32	100%
			Establish Product Integration Procedures and Criteria	1	2	4	8	16	32	100%
		Ensure Interface Compatibility	Review Interface Descriptions for Completeness	1	2	4	8	16	32	100%
			Manage Interfaces	1	2	4	8	16	32	100%
		Assemble Product Components and Deliver the Product	Confirm Readiness of Product Components for Integration	1	2	4	8	16	32	100%
			Assemble Product Components	1	2	4	8	16	32	100%
	Verification	Prepare for Verification	Evaluate Assembled Product Components	1	2	4	8	16	32	100%
			Package and Deliver the Product or Product Component	1	2	4	8	16	32	100%
			Select Work Products for Verification	1	2	4	8	16	32	100%
Perform Peer Reviews		Establish the Verification Environment	1	2	4	8	16	32	100%	
		Establish Verification Procedures and Criteria	1	2	4	8	16	32	100%	
		Prepare for Peer Reviews	1	2	4	8	16	32	100%	
Verify Selected Work Products		Conduct Peer Reviews	1	2	4	8	16	32	100%	
		Analyze Peer Review Data	1	2	4	8	16	32	100%	
		Perform Verification	1	2	4	8	16	32	100%	
		Analyze Verification Results and Identify Corrective Action	1	2	4	8	16	32	100%	
Prepare for Validation	Select Products for Validation	1	2	4	8	16	32	100%		
	Establish the Validation Environment	1	2	4	8	16	32	100%		

3 - Defined	Validation	Establish Validation Procedures and Criteria	1	2	4	8	16	32	100%
		Validate Product or Process	1	2	4	8	16	32	100%
		Analyze Validation Results	1	2	4	8	16	32	100%
	Organizational Process Focus	Determine Process Improvement Opportunities	1	2	4	8	16	32	100%
		Plan and Implement Process-Improvement Activities	1	2	4	8	16	32	100%
		Establish Organizational Process Needs	1	2	4	8	16	32	100%
		Appraise the Organization's Processes	1	2	4	8	16	32	100%
		Identify the Organization's Process Improvements	1	2	4	8	16	32	100%
		Establish Process Action Plans	1	2	4	8	16	32	100%
		Implement Process Action Plans	1	2	4	8	16	32	100%
	Organizational Process Definition	Establish Standard Processes	1	2	4	8	16	32	100%
		Establish Life-Cycle Model Descriptions	1	2	4	8	16	32	100%
		Establish Tailoring Criteria and Guidelines	1	2	4	8	16	32	100%
		Establish the Organization's Measurement Repository	1	2	4	8	16	32	100%
		Establish the Organization's Process Asset Library	1	2	4	8	16	32	100%
	Organizational Training	Establish an Organizational Training Capability	1	2	4	8	16	32	100%
		Provide Necessary Training	1	2	4	8	16	32	100%
		Establish the Strategic Training Needs	1	2	4	8	16	32	100%
		Determine Which Training Needs Are the Responsibility of Organization	1	2	4	8	16	32	100%
		Establish an Organizational Training Tactical Plan	1	2	4	8	16	32	100%
		Establish Training Capability	1	2	4	8	16	32	100%
		Deliver Training	1	2	4	8	16	32	100%
	Integrated Project Management for IPPD	Use the Project's Defined Process	1	2	4	8	16	32	100%
		Coordinate and Collaborate with Relevant Stakeholders	1	2	4	8	16	32	100%
		Use the Project's Shared Vision for IPPD	1	2	4	8	16	32	100%
		Organize Integrated Teams for IPPD	1	2	4	8	16	32	100%
		Use Organizational Process Assets for Planning Project Activities	1	2	4	8	16	32	100%
		Integrate Plans	1	2	4	8	16	32	100%
		Manage the Project Using the Integrated Plans	1	2	4	8	16	32	100%
		Contribute to the Organizational Process Assets	1	2	4	8	16	32	100%
		Manage Stakeholder Involvement	1	2	4	8	16	32	100%
		Resolve Coordination Issues	1	2	4	8	16	32	100%
	Risk Management	Prepare for Risk Management	1	2	4	8	16	32	100%
		Identify and Analyze Risks	1	2	4	8	16	32	100%
		Mitigate Risks	1	2	4	8	16	32	100%
		Determine Risk Sources and Categories	1	2	4	8	16	32	100%
		Define Risk Parameters	1	2	4	8	16	32	100%
		Establish a Risk Management Strategy	1	2	4	8	16	32	100%
		Identify Risks	1	2	4	8	16	32	100%
	Integrated Teaming	Establish Team Composition	1	2	4	8	16	32	100%
		Govern Team Operation	1	2	4	8	16	32	100%
		Identify Team Tasks	1	2	4	8	16	32	100%
		Identify Needed Knowledge and Skills	1	2	4	8	16	32	100%
		Assign Appropriate Team Members	1	2	4	8	16	32	100%
		Establish a Shared Vision	1	2	4	8	16	32	100%
	Integrated Supplier Management	Analyze and Select Sources of Products	1	2	4	8	16	32	100%
		Coordinate Work with Suppliers	1	2	4	8	16	32	100%
		Evaluate and Determine Sources of Products	1	2	4	8	16	32	100%
Monitor Selected Supplier Work Products		1	2	4	8	16	32	100%	
Decision Analysis and Resolution	Evaluate Alternatives	1	2	4	8	16	32	100%	
	Establish Guidelines for Decision Analysis	1	2	4	8	16	32	100%	
	Establish Evaluation Criteria	1	2	4	8	16	32	100%	
	Identify Alternative Solutions	1	2	4	8	16	32	100%	
	Select Evaluation Methods	1	2	4	8	16	32	100%	
	Evaluate Alternatives	1	2	4	8	16	32	100%	
Organizational Environment for Integration	Provide IPPD Infrastructure	1	2	4	8	16	32	100%	
	Manage People for Integration	1	2	4	8	16	32	100%	
	Establish an Integrated Work Environment	1	2	4	8	16	32	100%	
	Identify IPPD-Unique Skill Requirements	1	2	4	8	16	32	100%	
	Establish Leadership Mechanisms	1	2	4	8	16	32	100%	
CMMI Level 3 Score			100%	100%	100%	100%	100%	100%	100%

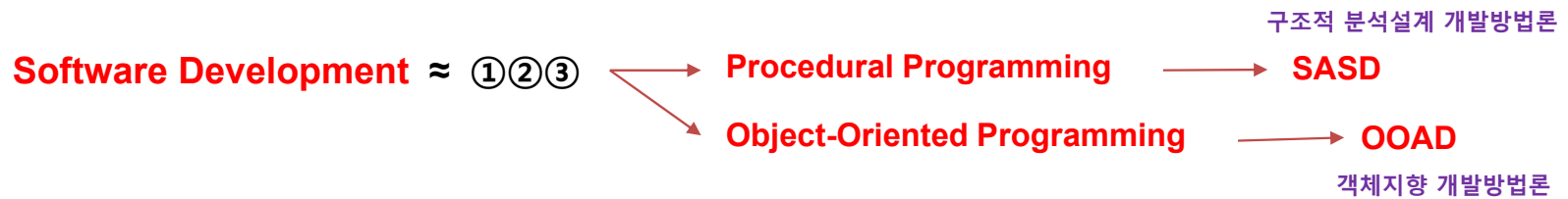
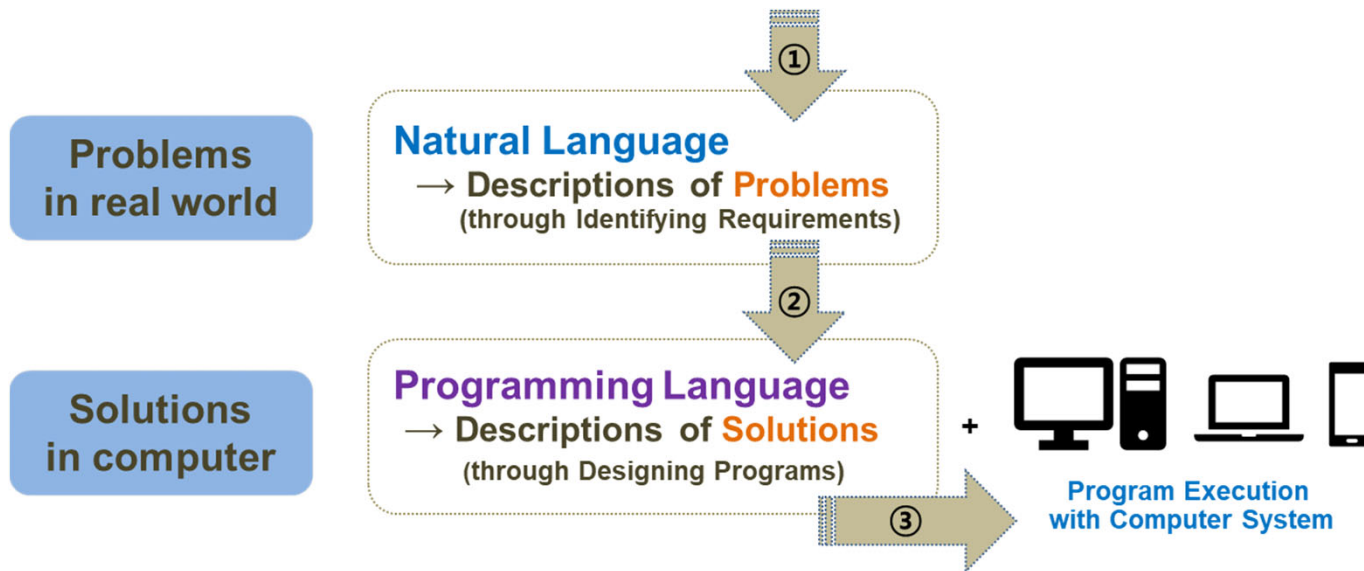
4 - Quantitatively Managed	Organizational Process Performance	Establish Performance Baselines and Models	Select Processes	1	2	4	8	16	32	100%
			Establish Process Performance Measures	1	2	4	8	16	32	100%
			Establish Quality and Process-Performance Objectives	1	2	4	8	16	32	100%
			Establish Process Performance Baselines	1	2	4	8	16	32	100%
			Establish Process Performance Models	1	2	4	8	16	32	100%
	Quantitative Project Management	Quantitatively Manage the Project	Establish the Project's Objectives	1	2	4	8	16	32	100%
			Compose the Defined Process	1	2	4	8	16	32	100%
			Select the Subprocesses that Will Be Statistically Managed	1	2	4	8	16	32	100%
		Statistically Manage Subprocess Performance	Manage Project Performance	1	2	4	8	16	32	100%
			Select Measures and Analytic Techniques	1	2	4	8	16	32	100%
			Apply Statistical Methods to Understand Variation	1	2	4	8	16	32	100%
			Monitor Performance of the Selected Subprocesses	1	2	4	8	16	32	100%
			Record Statistical Management Data	1	2	4	8	16	32	100%
			CMMI Level 4 Score	100%	100%	100%	100%	100%	100%	100%
5 - Optimizing	Organizational Innovation and Deployment	Select Improvements	Collect and Analyze Improvement Proposals	1	2	4	8	16	32	100%
			Identify and Analyze Innovations	1	2	4	8	16	32	100%
			Pilot Improvements	1	2	4	8	16	32	100%
		Deploy Improvements	Select Improvements for Deployment	1	2	4	8	16	32	100%
			Plan the Deployment	1	2	4	8	16	32	100%
	Manage the Deployment		1	2	4	8	16	32	100%	
	Causal Analysis and Resolution	Determine Causes of Defects	Measure Improvement Effects	1	2	4	8	16	32	100%
			Select Defect Data for Analysis	1	2	4	8	16	32	100%
		Address Causes of Defects	Analyze Causes	1	2	4	8	16	32	100%
			Implement the Action Proposals	1	2	4	8	16	32	100%
			Evaluate the Effect of Changes	1	2	4	8	16	32	100%
	CMMI Level 5 Score	100%	100%	100%	100%	100%	100%	100%		
	CMMI Level 2, 3, 4, & 5 Score				100%	100%	100%	100%	100%	100%

sm Capability Maturity Model Integration and CMMI are service marks of Carnegie Mellon University.

SW Development Methodology

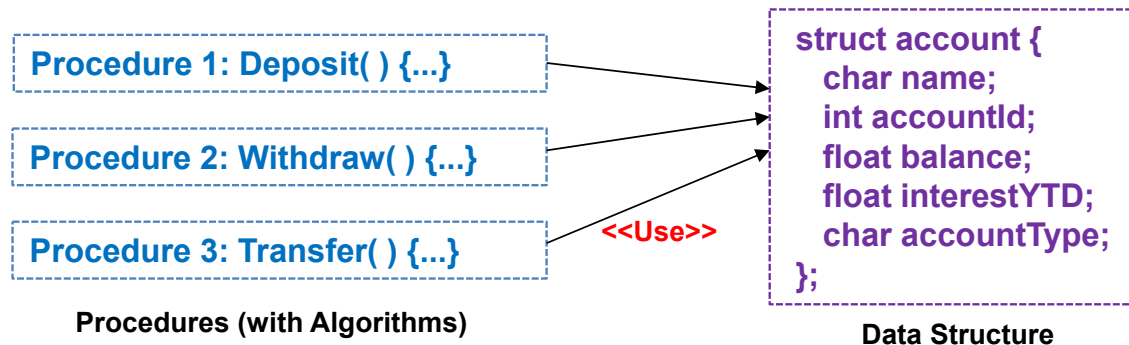
Software Development Methodology

- Software Development \approx Solving Problems with Software working on a Computer



Procedural Programming

- A program is organized with **procedures**.
 - **Procedure/Function**
 - building-block of procedural programs
 - **statements** changing values of **variables**
 - Focusing on data structures, algorithms, and sequencing of steps
 - **Algorithm** : a set of instructions for solving a problem (Control-centric)
 - **Data structure** : a construct used to organize data in a specific way (Data-centric)
 - Most computer languages (from FORTRAN to c) are procedural programming languages.



A Procedural Program

```

*polynomial.c - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

#include <stdio.h>
#include <stdlib.h>

int p1Coef, p1Degree, p2Coef, p2Degree;

//다항식 하나를 list로 보고, 다항식에 들어가는 각 항을 해당 list의 node로 본다.

typedef struct Term {
    int coeff;//각 항의 계수
    int degree;//각 항의 차수
    struct Term *next;//next term을 알 수 있는, 단순연결리스트
} Term;

typedef struct polynomial {
    Term *head;
    Term *tail;
}Polynomial;

void addTerm(Polynomial *p, int coeff, int degree) {
    //임의의 term이 자신의 다음에 오는 next term만 알 수 있으므로 새로 입력되는 항은 앞에 덧붙인다
    Term* termNew;
    Term* temp;

    termNew = (Term *)malloc(sizeof(Term));
    termNew->coeff = coeff;
    termNew->degree = degree;
    termNew->next = NULL;

    if (p->head == NULL) {
        p->head = termNew;
    }
    else {
        temp = p->head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = termNew;
    }
}

void tSort(Polynomial *p) {
    int temp;
    //다항식을 입력받은 후 차수가 높은 것부터 낮은 것까지 정렬한다
    Term *term = (Term *)malloc(sizeof(Term));
    term = p->head;
    while ((term->next != NULL) {
        if (p->head->degree > p->head->next->degree) {

```

```

*polynomial.c - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

int main() {

    Polynomial *p1 = (Polynomial *)malloc(sizeof(Polynomial));
    Polynomial *p2 = (Polynomial *)malloc(sizeof(Polynomial));
    Polynomial *p3 = (Polynomial *)malloc(sizeof(Polynomial));

    while (1) {
        printf("*****\n");
        printf("첫 번째 다항식에 들어갈 각 항의 계수와 지수를 순서대로 입력하세요\n");
        printf("계수에 0을 입력할 경우 더이상 입력 받지 않습니다\n");
        printf("*****\n");

        p1->head = NULL;
        do {
            scanf_s("%d %d", &p1Coef, &p1Degree);
            if (p1Coef != 0) {
                addTerm(p1, p1Coef, p1Degree);
            }
        } while (p1Coef != 0);

        printf("*****\n");
        printf("두 번째 다항식에 들어갈 각 항의 계수와 지수를 순서대로 입력하세요\n");
        printf("계수에 0을 입력할 경우 더이상 입력 받지 않습니다\n");
        printf("*****\n");

        p2->head = NULL;
        do {
            scanf_s("%d %d", &p2Coef, &p2Degree);
            if (p2Coef != 0) {
                addTerm(p2, p2Coef, p2Degree);
            }
        } while (p2Coef != 0);

        printf("*****\n");
        printf("A(x)=");
        printPoly(p1);
        printf("B(x)=");
        printPoly(p2);

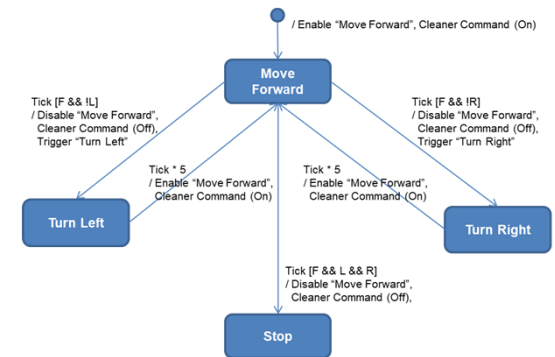
        p3->head = NULL;
        addP(p1, p2, p3);

        printf("*****\n");
        printf("C(x)=");
        printPoly(p3);
    }
}

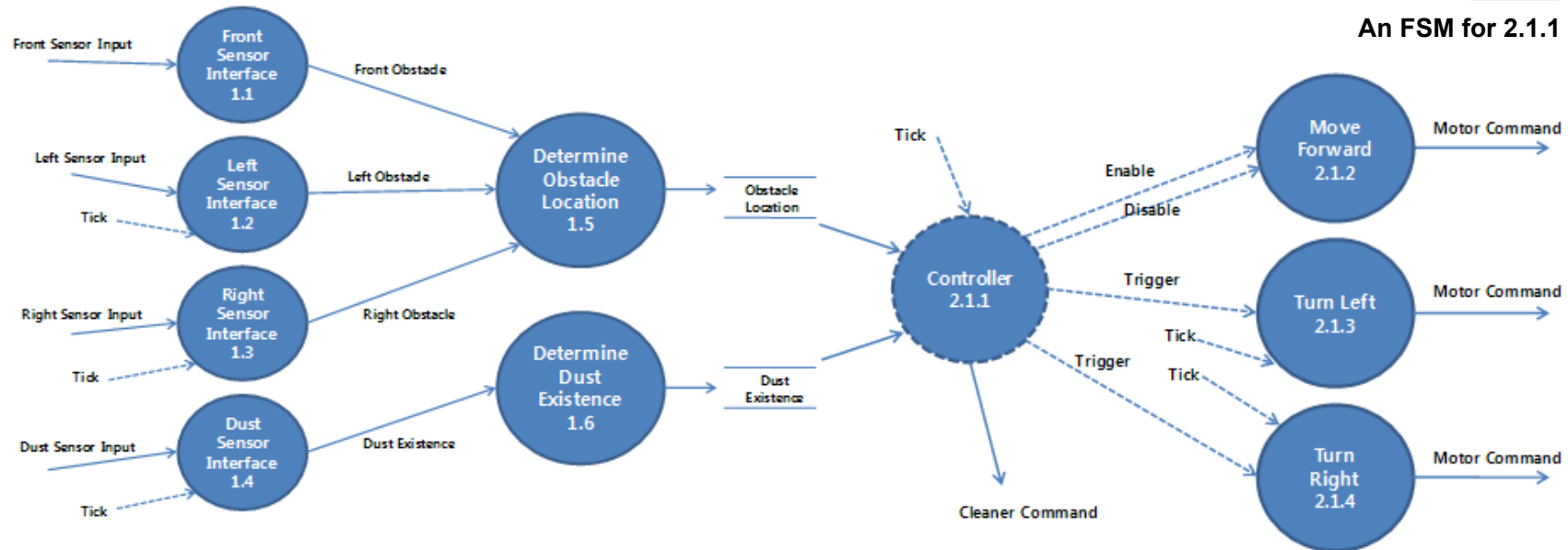
```


SASD for Procedural Programming

- **SASD** (Structured Analysis and Structured Design, 구조적분석설계 개발방법론)
 - A traditional software development methodology for procedural programs
 - **Top-Down Divide and Conquer**
 - Divide large, complex problems into smaller, more easily handled ones
 - Functional view of the problem using **DFD** (Data Flow Diagram)

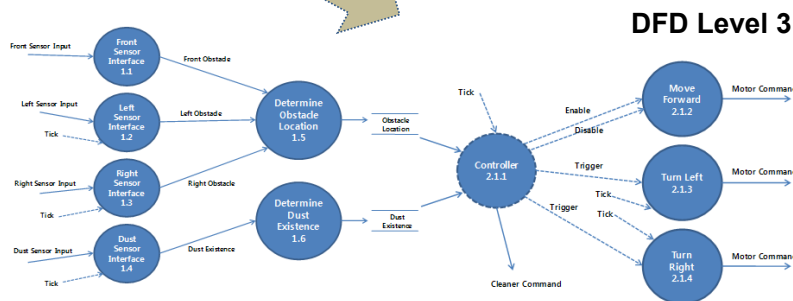
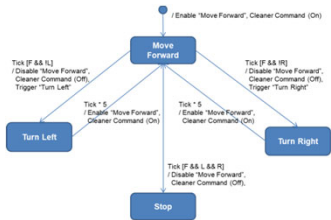
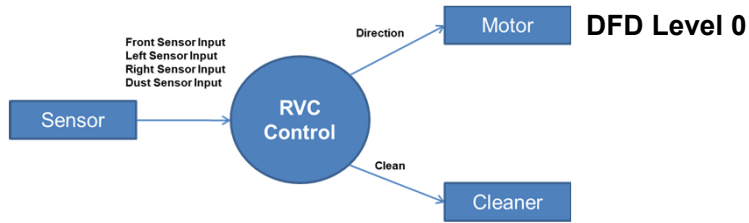


An FSM for 2.1.1 Controller



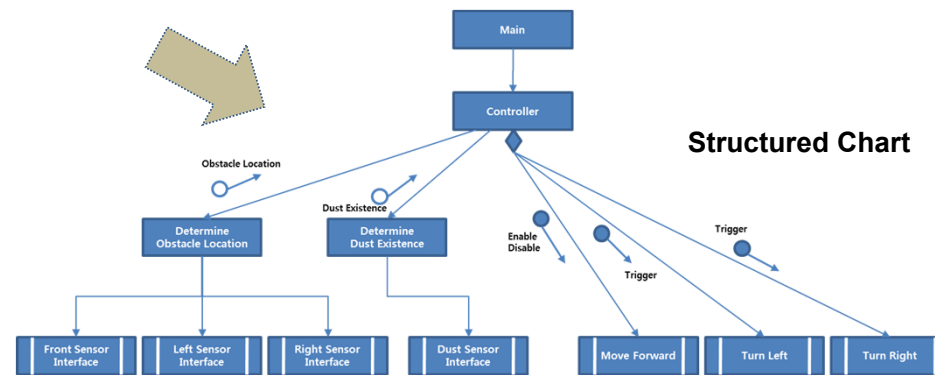
A level 3 DFD for RVC Control

An SASD Example - RVC Control



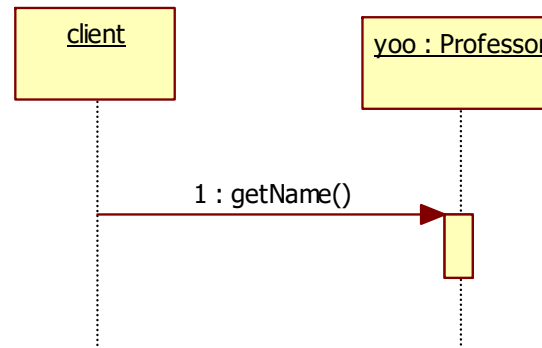
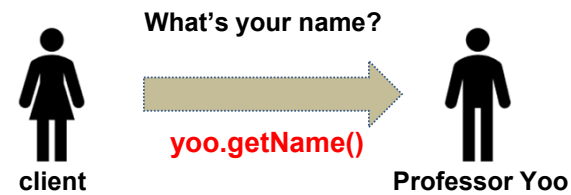
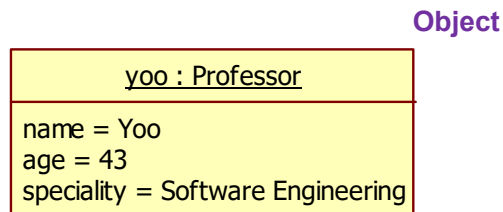
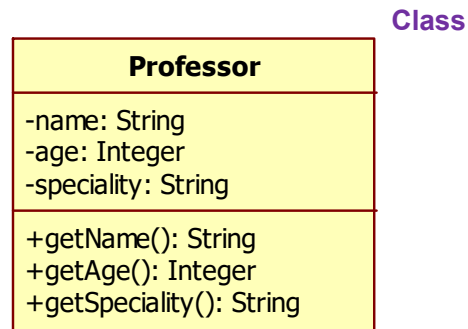
Structured Analysis

Structured Design



Object-Oriented Programming

- A program is organized with **objects**.
 - **Providing system functionalities** through **object communications**
 - **Object** : consisting of **data** and **operations**
 - **Object communication** : an object **calls** an operation of other objects with its data
 - No explicit data flow, but only **communication sequences** among objects



An Object-Oriented Program

```

Factories.java - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
import java.util.Scanner;

public class Factorials {
    static int fac;

    public static int recursiveFactorial(int n) {
        if (n == 0) return 1;
        else return n*recursiveFactorial(n-1);
    }

    public static void main(String[] args) {
        while(true) {
            System.out.println("계산하고자 하는 factorial 값을 입력하세요.");
            Scanner keyboard = new Scanner(System.in);
            if(keyboard.hasNextInt()) {
                fac = keyboard.nextInt();
            }else {
                System.out.println("잘못된 입력입니다! 프로그램을 종료합니다.");
                break;
            }

            System.out.print( fac + "! = ");

            if(fac == 0) {
                System.out.println("0! = 1");
            }else {
                int i = fac;
                while(i > 1) {
                    System.out.print(i + "*");
                    i--;
                }

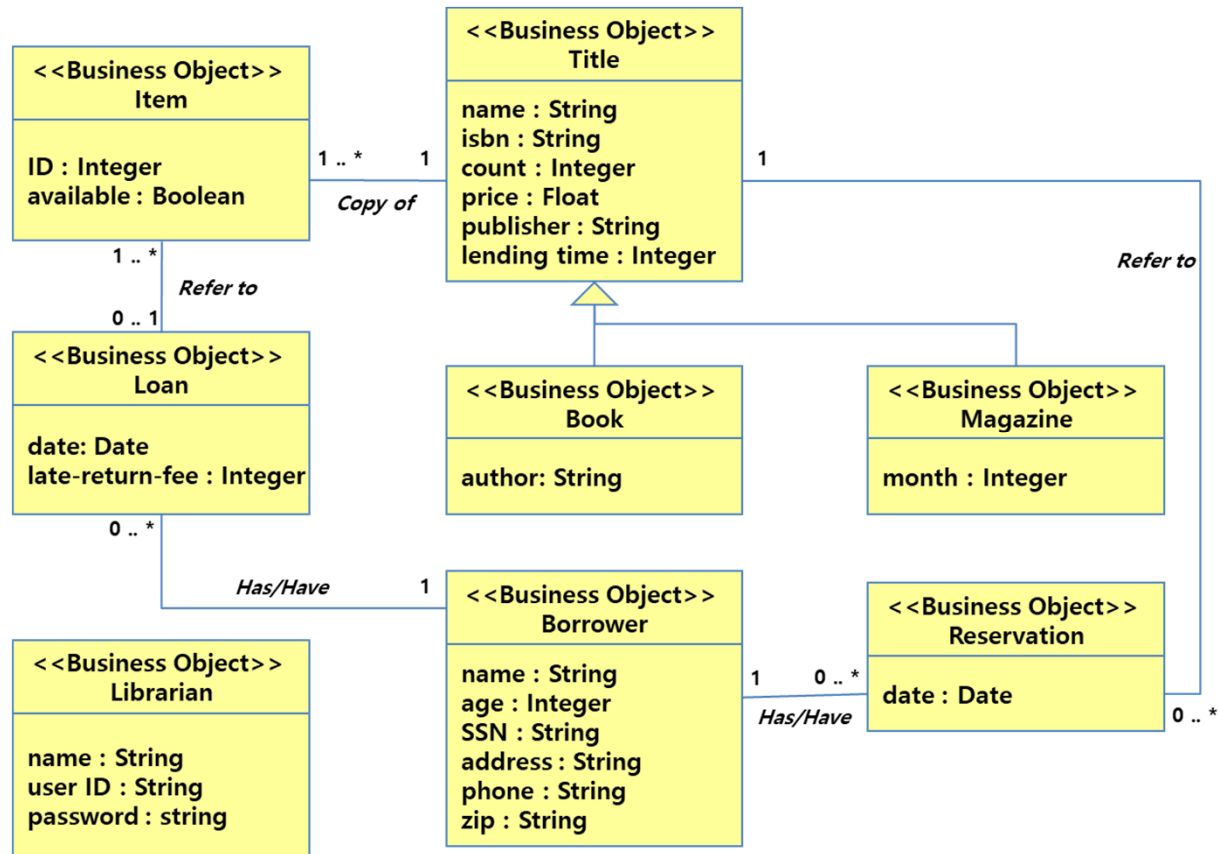
                System.out.println("1 = " + recursiveFactorial(fac) + "\n");
            }
        }
    }
}
Ln 1, Col 1    100%    Windows (CRLF)    UTF-8

```

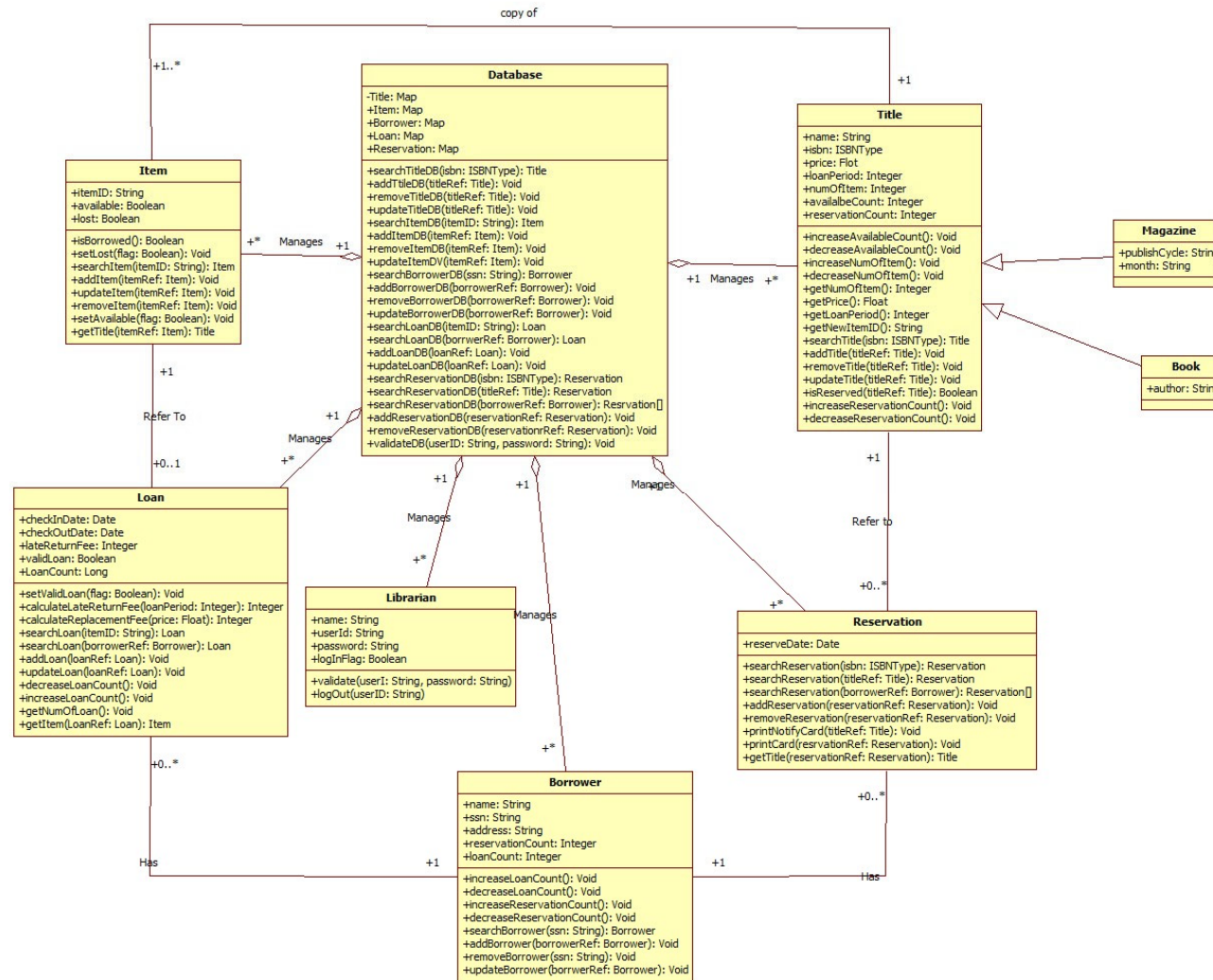
OOAD for Object-Oriented Programming

- **OOAD** (Object-Oriented Analysis and Design, AKA 객체지향개발방법론)
 - *“Identifying your requirements and creating a domain model, and then add methods to the appropriate classes and define the messaging between the objects in order to fulfill the requirements”*
 - **Object-Oriented Analysis (OOA)**
 - Discover the domain concepts/objects (**Domain Model**)
 - Identify requirements (**Use-Case Model**)
 - **Object-Oriented Design (OOD)**
 - Define software objects (Static model → **Class Diagram**)
 - Define how they collaborate to fulfill the requirements (Dynamic model → **Sequence Diagram**)
 - Various development process models are available.
 - **Waterfall**
 - **UP (Iterative)**

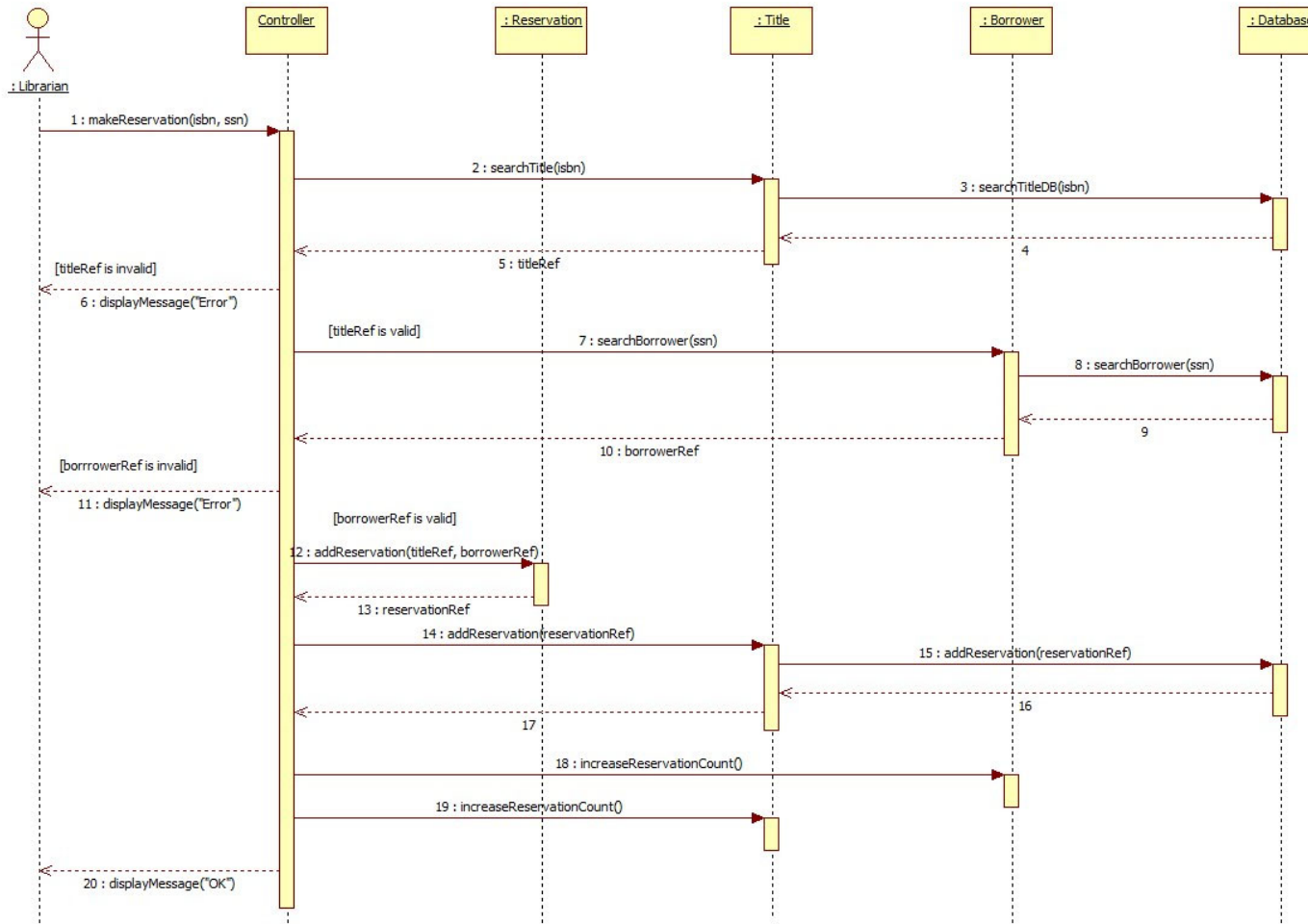
OOA - Domain Model



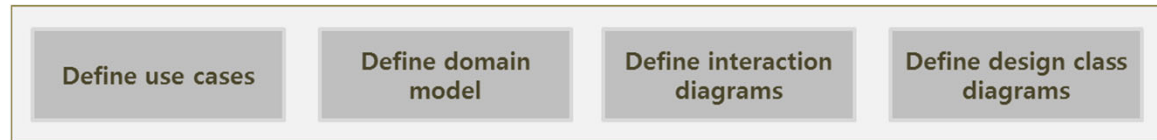
OOD Static Model - Class Diagram



OOD Dynamic Model - Sequence Diagram



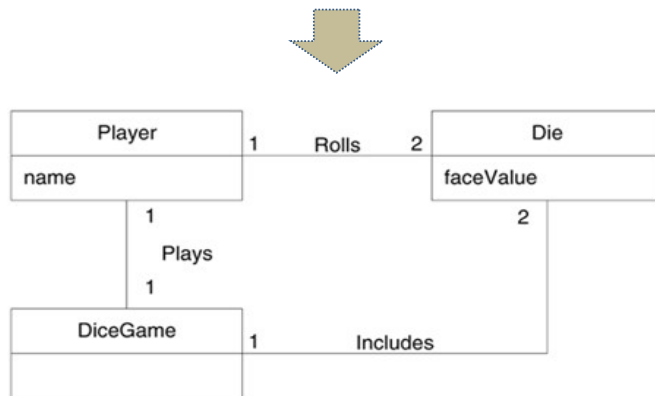
An OOAD Example - Dice Game



OOA

Use Case : Play a Dice Game

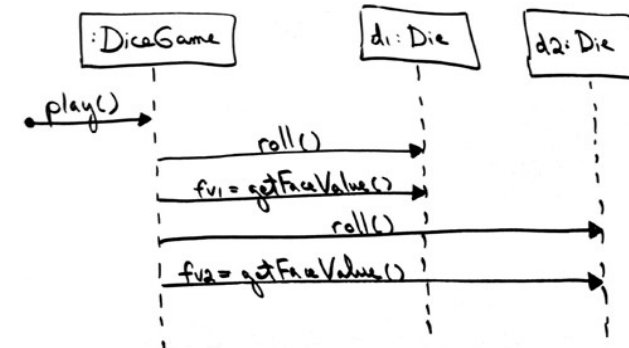
- Player requests to roll the dice.
- System presents results.
- If the dice's face value totals seven, player wins; otherwise, player loses.



Domain Model

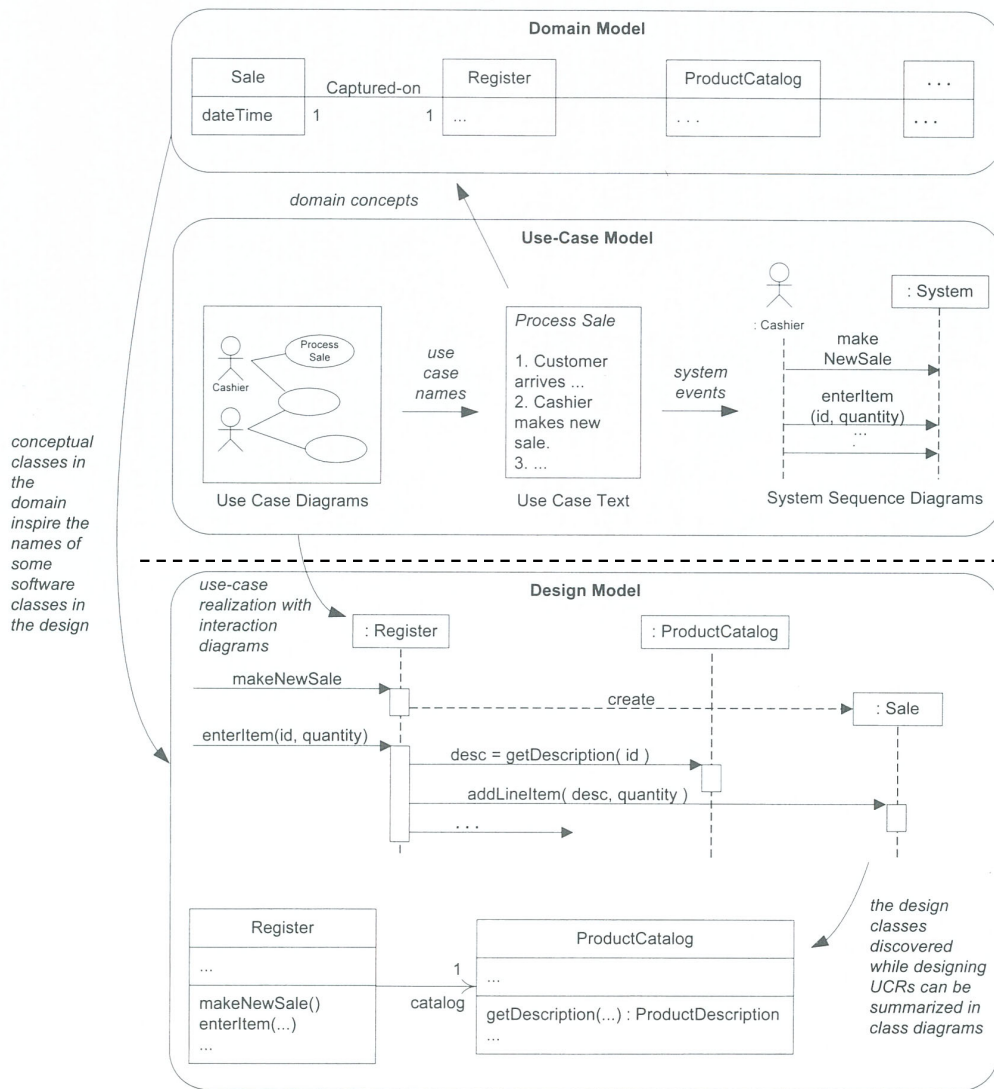
OOD

Interaction Diagram



Design Class Diagram

Sample Unified Process Artifact Relationships



conceptual classes in the domain inspire the names of some software classes in the design

Domain Model

Use-Case Model

OOA
OOD

Sequence Diagrams

Class Diagram

OO Implementation

Homework / Activity #2

- CMMi의 적용에 대한 국내 신문 기사를 찾아보세요.
- 본인이 최근에 수행한 (팀)프로젝트를 대상으로, CMMi 평가를 Rough하게 진행하고, 그 내용을 간단하게 정리하세요.
 - 제공되는 CMMi 2.1 Checklist를 기준으로, 각 항목에 대해서 “상/중/하”로 평가합니다.

CMMI sm (SE/SW/IPPD/SS) Process Evaluation Checklist											
Level	Process Area	Specific Goal	Specific Practice	Concept	Detailed	Modular	Formatted	Usable	Evidence	Total	
1 - Initial	Requirements Management	Manage Requirements	Obtain an Understanding of Requirements	1	2	4	8	16	32	100%	
			Obtain Commitment to Requirements	1	2	4	8	16	32	100%	
			Manage Requirements Changes	1	2	4	8	16	32	100%	
			Maintain Bidirectional Traceability of Requirements	1	2	4	8	16	32	100%	
			Identify Inconsistencies Between Project Work and Requirements	1	2	4	8	16	32	100%	
	Project Planning	Establish Estimates	Estimate the Scope of the Project	1	2	4	8	16	32	100%	
			Establish Estimates of Work Product and Task Attributes	1	2	4	8	16	32	100%	
			Define Project Life Cycle	1	2						
		Develop a Project Plan	Determine Estimates of Effort and Cost	1	2						
			Establish the Budget and Schedule	1	2						
			Identify Project Risks	1	2						
			Plan for Data Management	1	2						
			Plan for Project Resources	1	2						
			Plan for Needed Knowledge and Skills	1	2						
		Obtain Commitment to the Plan	Plan Stakeholder Involvement	1	2						
			Establish the Project Plan	1	2						
			Review Plans that Affect the Project	1	2						
			Reconcile Work and Resource Levels	1	2						
			Obtain Plan Commitment	1	2						
			Monitor Project Planning Parameters	1	2						
	Project Monitoring and Control	Monitor Project Against Plan	Monitor Commitments	1	2						
			Monitor Project Risks	1	2						
			Monitor Data Management	1	2	4	8	16	32	100%	
			Monitor Stakeholder Involvement	1	2	4	8	16	32	100%	
			Conduct Progress Reviews	1	2	4	8	16	32	100%	
		Manage Corrective Action to Closure	Conduct Milestone Reviews	1	2	4	8	16	32	100%	
			Analyze Issues	1	2	4	8	16	32	100%	
			Take Correction Action	1	2	4	8	16	32	100%	
			Manage Corrective Action	1	2	4	8	16	32	100%	
			Determine Acquisition Type	1	2	4	8	16	32	100%	

각 항목별 정성 평가
(상 / 중 / 하)

Samples from SE Undergraduate (KU 2021)

넥스트칩, 'CMMI V2.0' 인증 "차량용 반도체 강화"

기사입력 2020.10.12. 오후 2:47 | 최종수정 2020.10.12. 오후 2:50 | 기사유형 스크린 | 본문듣기 · 설정



[이데일리 강경래 기자] 넥스트칩이 'CMMI V2.0' 레벨3 인증을 취득하면서 자동차 반도체 분야에서 경쟁력을 한층 강화했다고 12일 밝혔다.

CMMI(능력성숙도통합모델, Capability Maturity Model Integration)는 미국 카네기멜론 대학 부설 소프트웨어공학연구소(SEI)에서 개발한 소프트웨어와 시스템 엔지니어링 프로세스 능력 평가 모델이다. 이는 자동차 시장에서 비즈니스를 영위하기 위한 핵심 인증 중 하나로 꼽힌다.

넥스트칩 측은 "급변하는 비즈니스 환경과 다양한 산업계 요구사항을 반영하기 위해 2018년에 기존 V1.3 CMMI를 개정해 V2.0 기준을 만들었다"며 "넥스트칩이 국내 업계 최초로 V2.0 기준으로 심사를 진행해 인증을 받았다"고 설명했다.

이번 레벨3 인증 취득은 국제 표준 기반으로 정해진 공정에 따라 체계적으로 제품 개발을 진행하고 관리했음을 의미한다. 이를 통해 자동차에 들어가는데 있어 신뢰성과 품질을 확보했음을 의미하기 때문에 레벨3 인증은 자동차 반도체 분야에서 의미가 있다.

넥스트칩은 앞서 지난 3월 소프트웨어 개발에 특화된 프로세스 모델인 'A-SPICE' 레벨 3 인증도 취득했다. 이를 통해 완성차 업체에서 요구하는 소프트웨어 필수 요구사항을 만족시키며 자동차 반도체 분야에서의 경쟁력을 한층 높일 수 있었다.

넥스트칩 관계자는 "이번 V2.0 기준 레벨3 인증을 통해 자동차 반도체 시장에서의 기술 경쟁력을 다시 한번 확인할 수 있었다"며 "국내외 자동차 시장에서 더 활발하게 적용될 것으로 기대한다"고 말했다.

한화시스템, 연구개발 역량평가에서 최고 등급 받아

기사입력 2020.12.02. 오후 4:24 | 기사유형 스크린 | 본문듣기 · 설정

10 2

요약문 가 댓글



[서울경제] 한화시스템(272210)이 국제적인 연구개발(R&D) 역량평가 기준인 'CMMI(Capability Maturity Model Integration)' 최신 2.0 버전에서 최고 등급인 레벨 5 인증을 획득했다고 2일 밝혔다.

CMMI는 미국 카네기멜론 대학의 소프트웨어 공학연구소가 미국 국방부 의뢰를 받아 개발한 세계적인 소프트웨어 시스템 품질관리 역량평가 인증 모델이다. 해당 인증은 소프트웨어 개발 및 프로젝트 관리에 대한 조직의 역량을 종합적으로 평가한다.

한화시스템은 기존 CMMI 1.3 버전에서 최고 단계인 레벨 5를 지난 2009년부터 5회 연속 인증해왔으며, 2020년 CMMI 인증 만료시점에 맞춰 새롭게 개정된 2.0 버전 인증작업을 진행해왔다.

CMMI 2.0 버전은 무기체계 연구개발 과제뿐만 아니라, 응용연구 시험개발 등 국방 연구개발 전체 분야의 사업적 기술적 요건을 충족해야 인증을 받을 수 있다. 한화시스템은 CMMI 기반으로 연구개발 과정을 개선 발전시켜왔으며, 전략적 관리체계를 내재화하여 업무 효율과 시스템 개선 및 혁신활동에 크게 기여해왔다.

2. [KAI, '항공 소프트웨어 개발역량' 세계 최고 입증]

항공기개발 전 분야에서 CMMI 최고단계 레벨 5 획득



한국항공우주산업(이하 KAI)이 항공기 개발 전 분야에서 CMMI(Capability Maturity Model Integration) 최고 단계인 레벨 5를 획득하며 항공 소프트웨어(SW) 개발역량이 세계 최고 수준임을 다시 한 번 입증했다.

한국항공우주산업(이하 KAI)이 항공 소프트웨어(SW) 개발역량이 세계 최고 수준임을 다시 한 번 입증했다. 항공기 개발 전 분야에서 CMMI(Capability Maturity Model Integration) 최고 단계인 레벨 5를 획득한 것.

KAI는 2014년 항공전자 부문의 CMMI 레벨 5 인증을 획득한데 이어 올해 체계중합(S), 항공전자, 비행제어, 시험평가 등 항공기 개발 전 분야로 대상을 확대하여 CMMI의 레벨 5를 인증 받았다.

CMMI는 SW와 시스템공학(SE) 분야의 개발역량을 평가하는 대표적인 국제 기준이다. 미국 국방성이 우수 SW와 시스템 개발업체를 객관적인 기준으로 선정하기 위해 카네기멜론대학 SW공학 연구소(SE, SW Engine

3. Agile Software Development

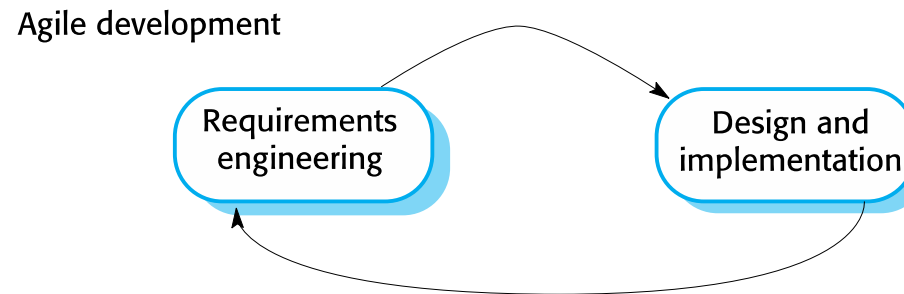
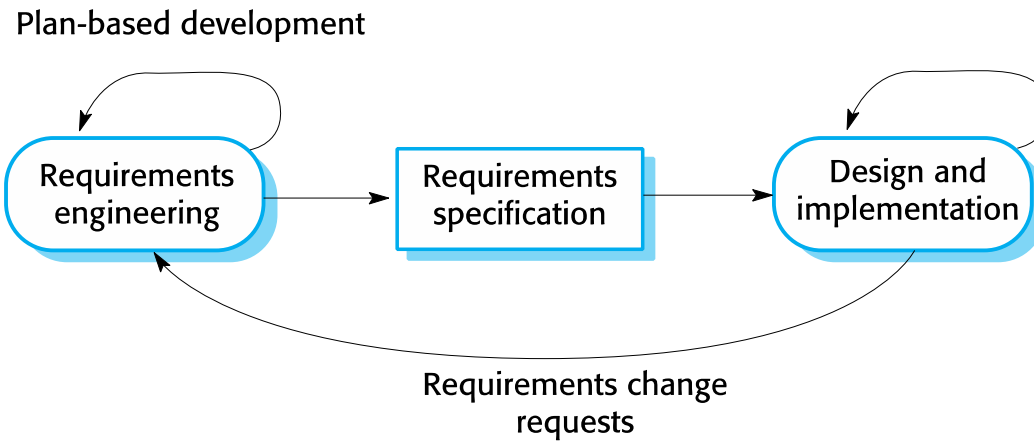
Rapid Software Development

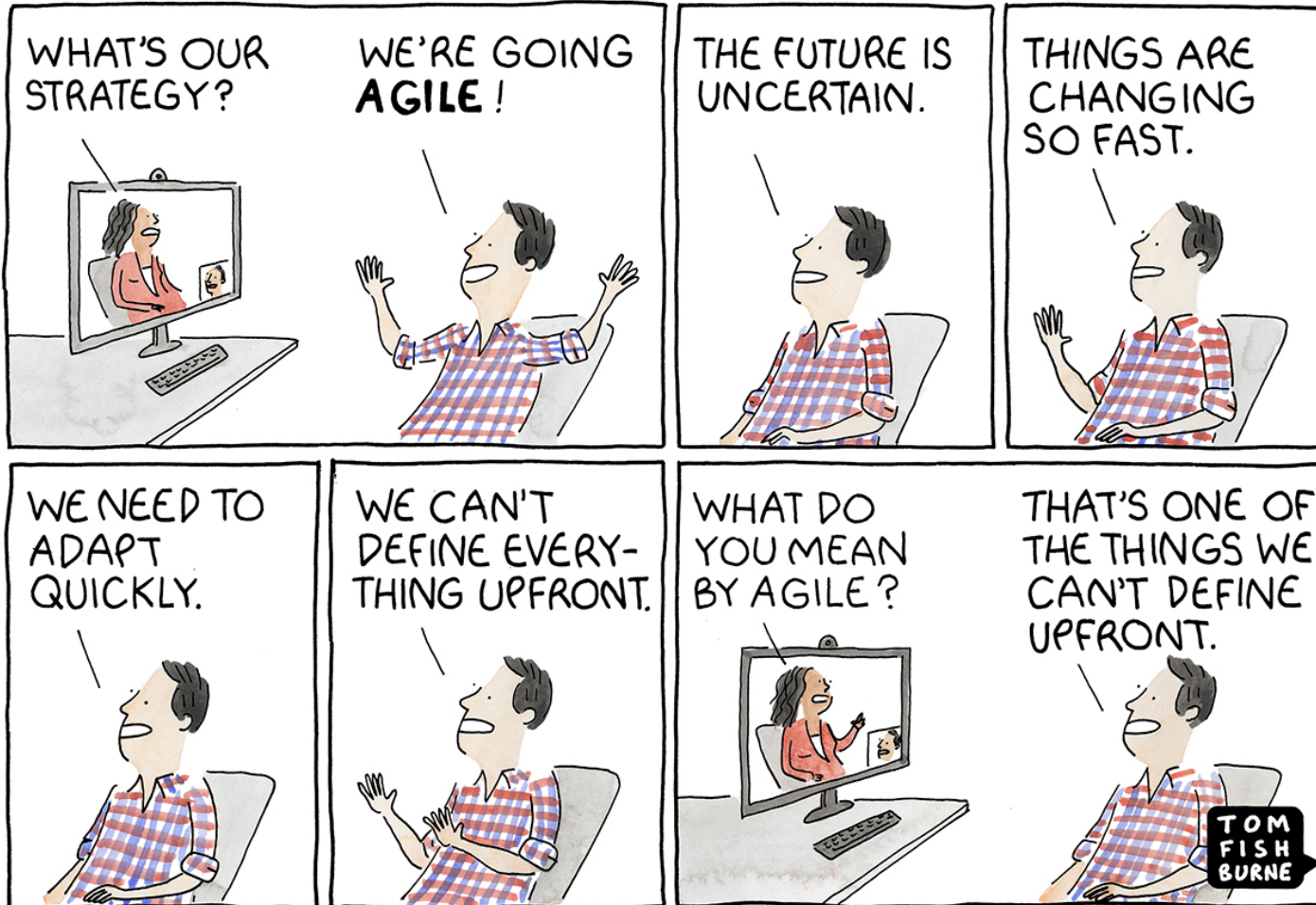
- **Rapid development and delivery** is now often the most important requirement for software systems.
 - Software must **evolve quickly** to reflect changing business needs.
 - **Plan-driven development** does **not** meet these business needs.

- **Agile development methods** emerged in the late 1990s to radically reduce the delivery time for working software systems.

- **Features of Agile development**
 - The system is developed as **a series of versions or increments** with **stakeholders involved** in version specification and evaluation.
 - **Frequent delivery** of new versions for evaluation
 - Extensive tool support (e.g., **automated testing tools**)
 - **Minimal documentation** to focus on working code

Plan-Driven (Waterfall) vs. Agile Development





© marketoonist.com

Agile Methods

Agile Methods

- Motivation
 - Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s (**Waterfall**)
 - To reduce overheads in the software process and to be able to respond quickly to changing requirements without excessive rework

- **Agile methods**
 - Focus on the code rather than the design
 - Based on an iterative approach to software development
 - Intend to deliver working software quickly and evolve this quickly to meet changing requirements

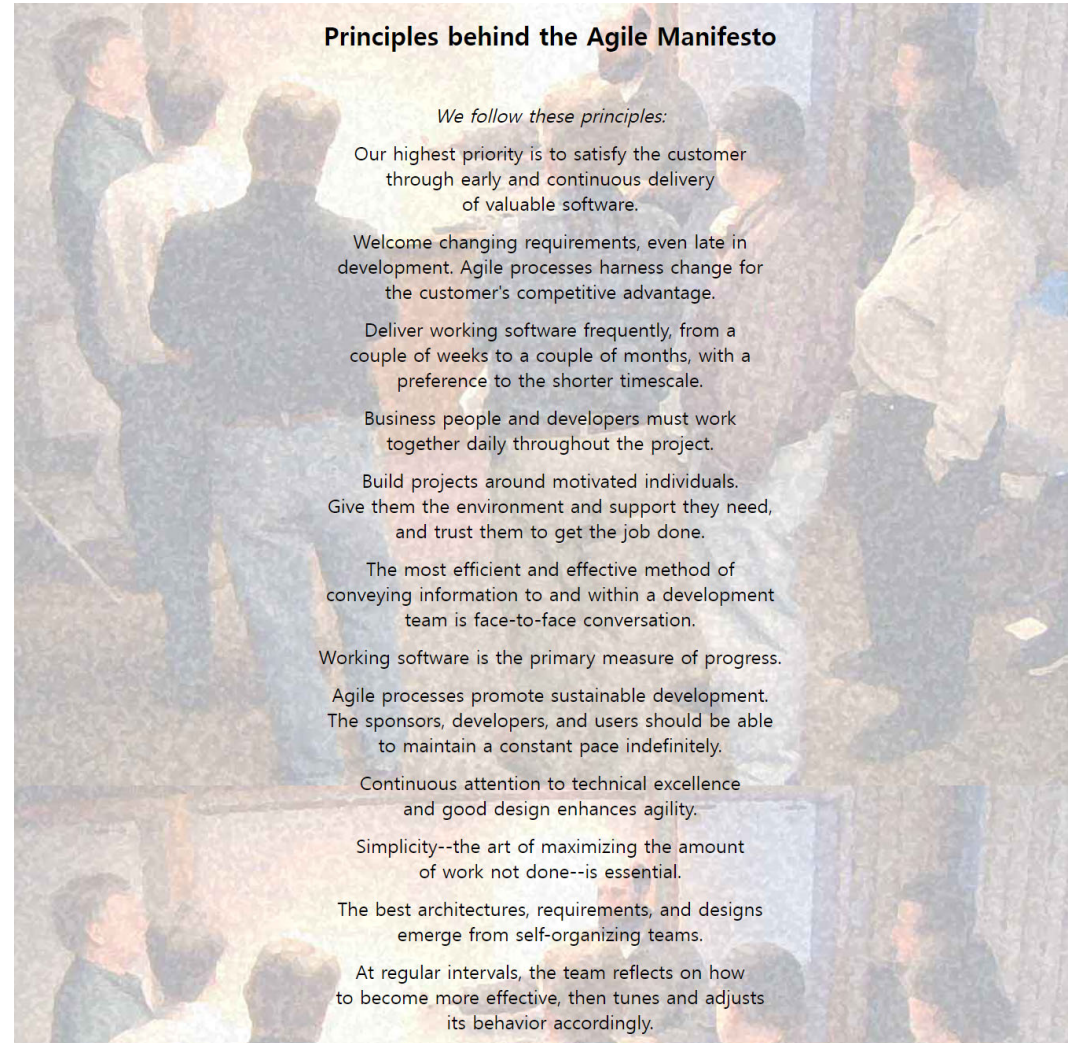
- **Two types of Agile methods**
 - **Agile Development Techniques**
 - **Agile Project Management**

Agile Manifesto

- *“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value.”*



Principles of Agile Methods



Principles behind the Agile Manifesto

We follow these principles:

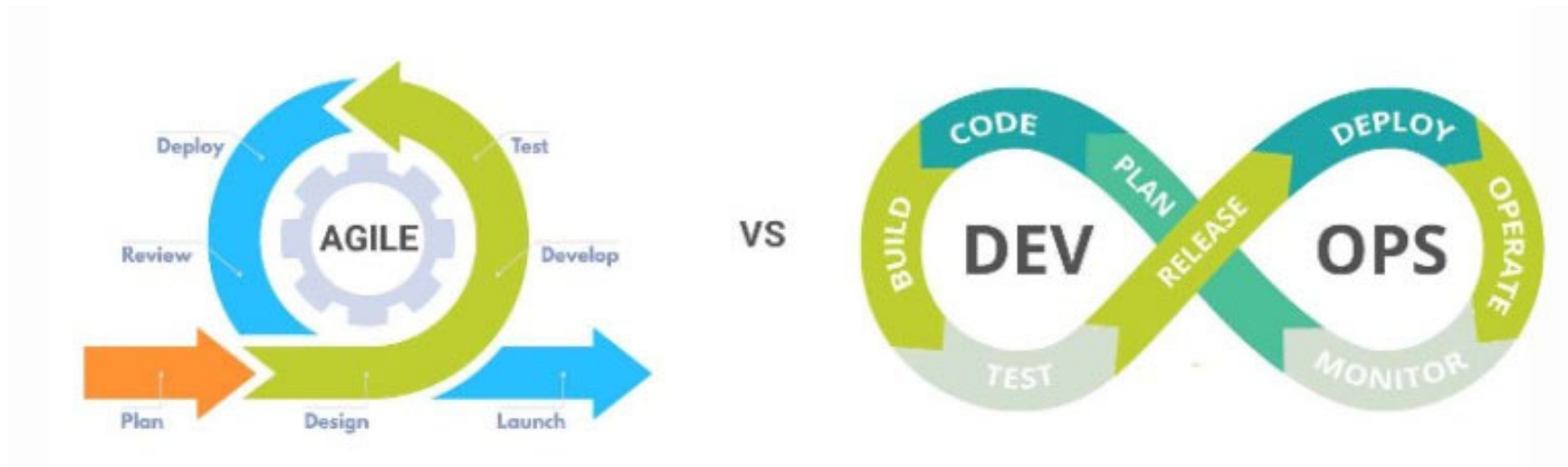
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity--the art of maximizing the amount of work not done--is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Applicability of Agile Method

- Development of **small** or **medium-sized** product for sale
 - Almost all software products and apps are now developed using an agile approach.

- Custom system development within an organization where,
 - Clear commitment from **customers** to become involved in the development process.
 - Few external rules and regulations that affect the software

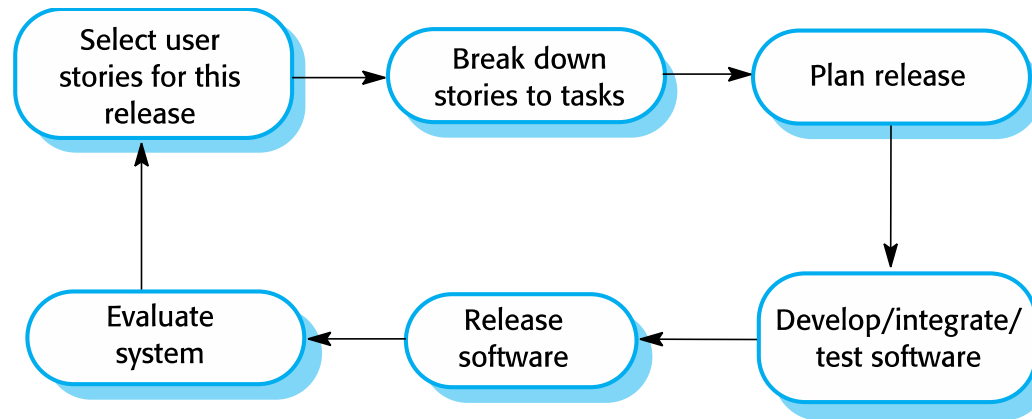
Agile vs. DevOps



Agile Development Techniques

Extreme Programming

- **Extreme Programming (XP)** takes an '*extreme*' approach to iterative development.
 - New versions may be built several times per day.
 - Increments are delivered to customers every 2 weeks.
 - All tests must be run for every build and the build is only accepted if tests run successfully.



XP Principles

- **The XP principles**

- Incremental development is supported through small, frequent system releases.
- **Customer involvement** means full-time customer engagement with the team.
- Collective ownership through pair programming
- Change supported through regular system releases
- Maintaining simplicity through constant refactoring

XP Practices

Principle or Practice	Description
<u>Incremental planning</u>	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'.
<u>Small releases</u>	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
<u>Simple design</u>	Enough design is carried out to meet the current requirements and no more.
<u>Test-first development</u>	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
<u>Refactoring</u>	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.
<u>Pair programming</u>	Developers work in pairs, checking each other's work and providing the support to always do a good job.
<u>Collective ownership</u>	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
<u>Continuous integration</u>	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
<u>Sustainable pace</u>	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
<u>On-site customer</u>	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

XP in Practice

- The XP method itself is not widely used now, since
 - Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.

- However, XP practices are widely used in other development methods.
 1. User stories for specification
 2. Refactoring
 3. Test-first development (TFD)
 4. Pair programming

1. User Stories for Requirements

- User requirements are expressed as **user stories** or **scenarios**.
 - Written on cards and the development team break them down into implementation tasks. **Tasks** are the basis of schedule and cost estimates.
 - **Customer or user is part of the XP team** and is responsible for making decisions on requirements.
 - The customer chooses the stories for inclusion in the next release.

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either 'current medication', 'new medication' or 'formulary'.

If you select 'current medication', you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, 'new medication', the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose 'formulary', you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click 'OK' or 'Change'. If you click 'OK', your prescription will be recorded on the audit database. If you click 'Change', you reenter the 'Prescribing medication' process.

User Story

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

Tasks

2. Refactoring

“나중에 고치기 쉽게 설계하라.”
 “나중에 쉽게 고칠 수 있는 자리를 미리 만들어 놔라.”

- Conventional wisdom in software engineering is to design for change.
 - It is worth spending time and effort **anticipating changes** as this reduces costs later in the life cycle.
- XP, however, claims that this is not worthwhile as changes cannot be reliably anticipated.
- XP proposes **constant code improvement (Refactoring)** to make changes easier when they must be implemented.
 - Changes are easier to make because the code is well-structured and clear.
 - Examples of refactoring
 - Re-organization of a class hierarchy to remove duplicate code.
 - Tidying up and renaming attributes and methods to make them easier to understand.
 - The replacement of inline code with calls to methods that have been included in a program library.
 - Types/levels of refactoring
 - Architecture > Design > Code > Data

3. Test-First Development

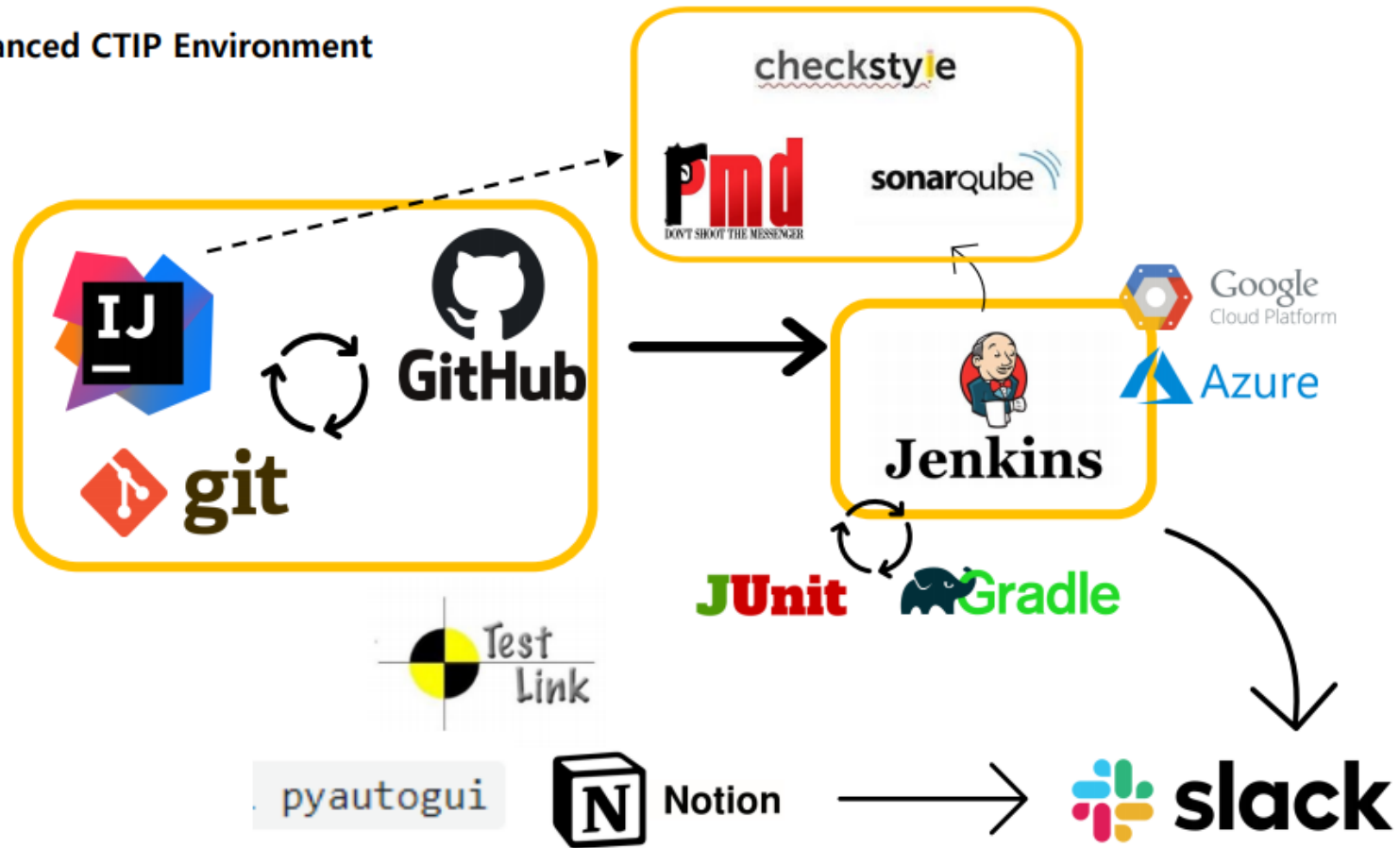
- **TFD (Test-First Development)**
 - Testing is central to XP.
 - *“The program should be tested after every change has been made.”*

- Difficulties in TFD
 - Programmers prefer programming to testing and sometimes they take short cuts when writing tests.
 - Some tests can be very difficult to write incrementally.
 - It is difficult to judge the completeness of a set of (a lot of) tests.

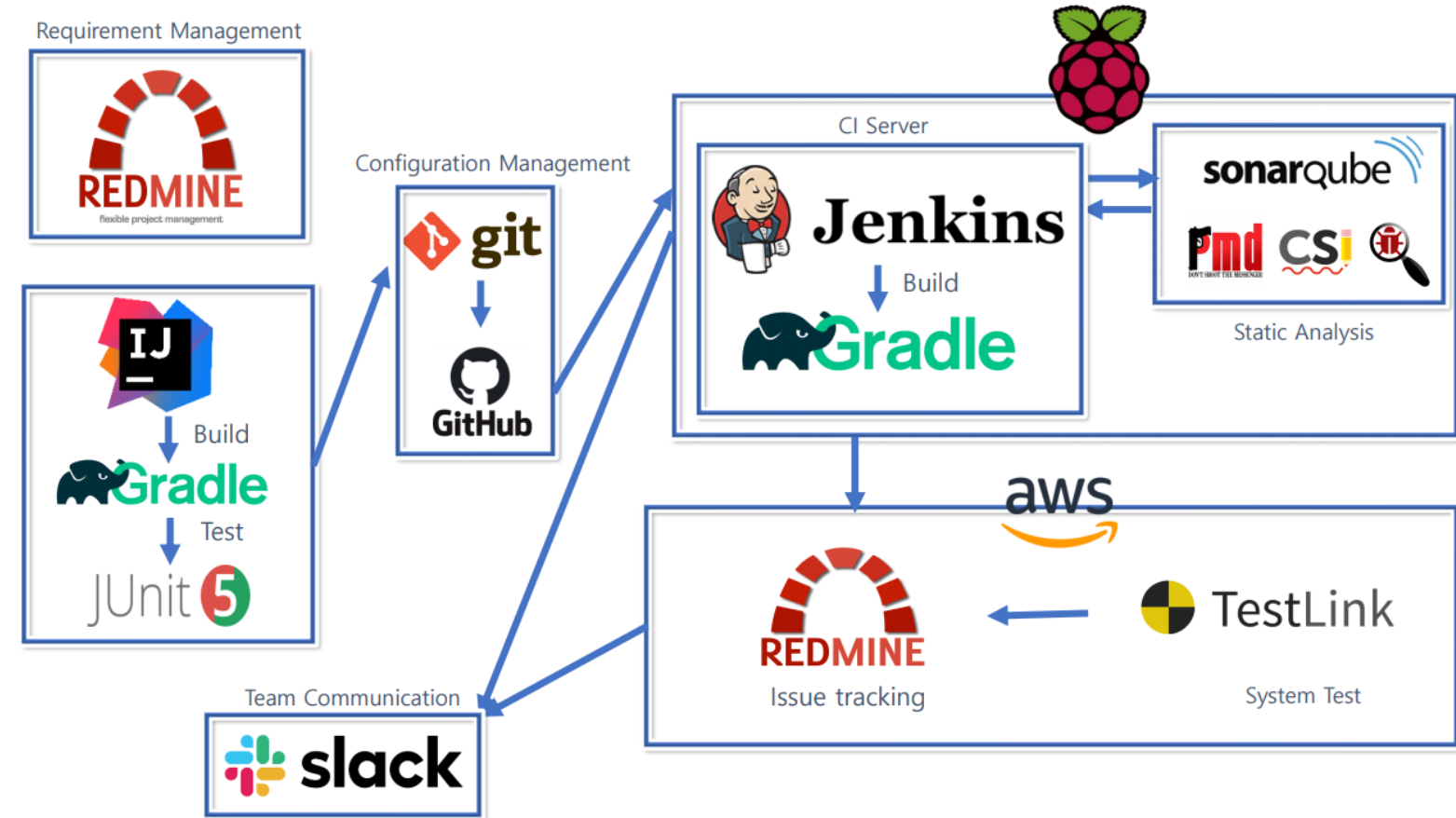
- **Features of the XP testing**
 - Test-First development
 - Incremental test development from scenarios
 - User involvement in test development and validation
 - Automated test harnesses are used to run all component tests each time that a new release is built.
(**CTIP: Continuous Testing and Integration Platform**)

CTIP Examples (KU 2021)

Advanced CTIP Environment

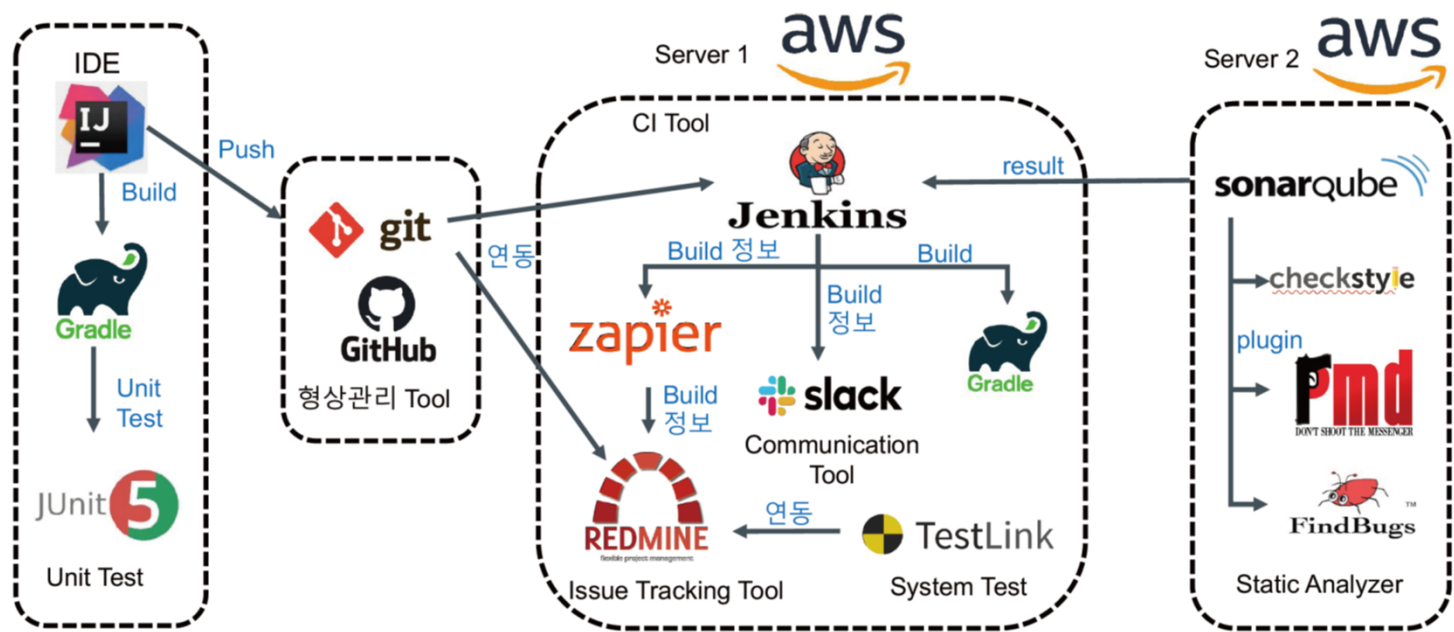


CTIP Examples (KU 2021)



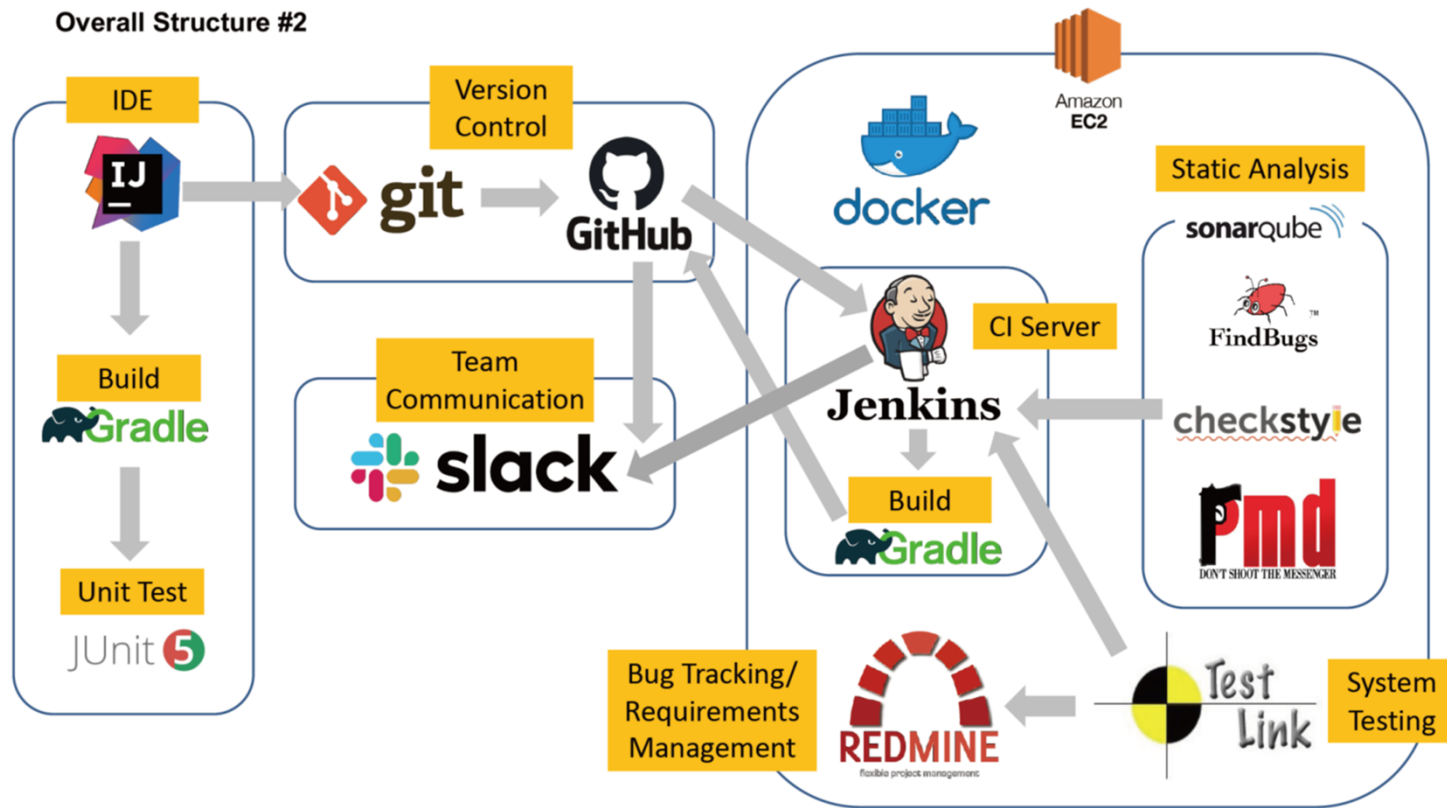
CTIP Examples (KU 2021)

Part 1, CTIP 환경 구성도

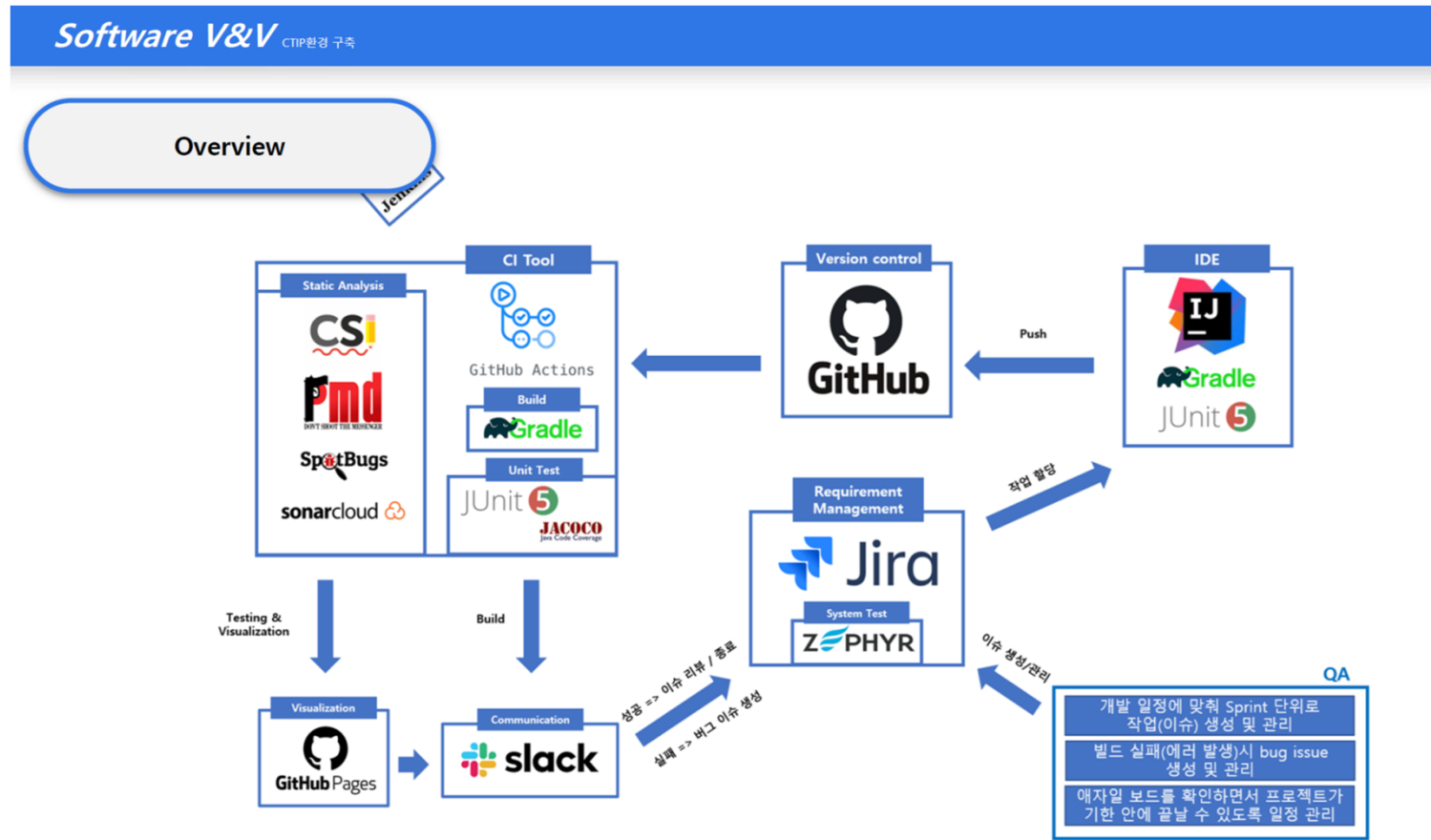


©Saebyeol Yu, Saebyeol's PowerPoint

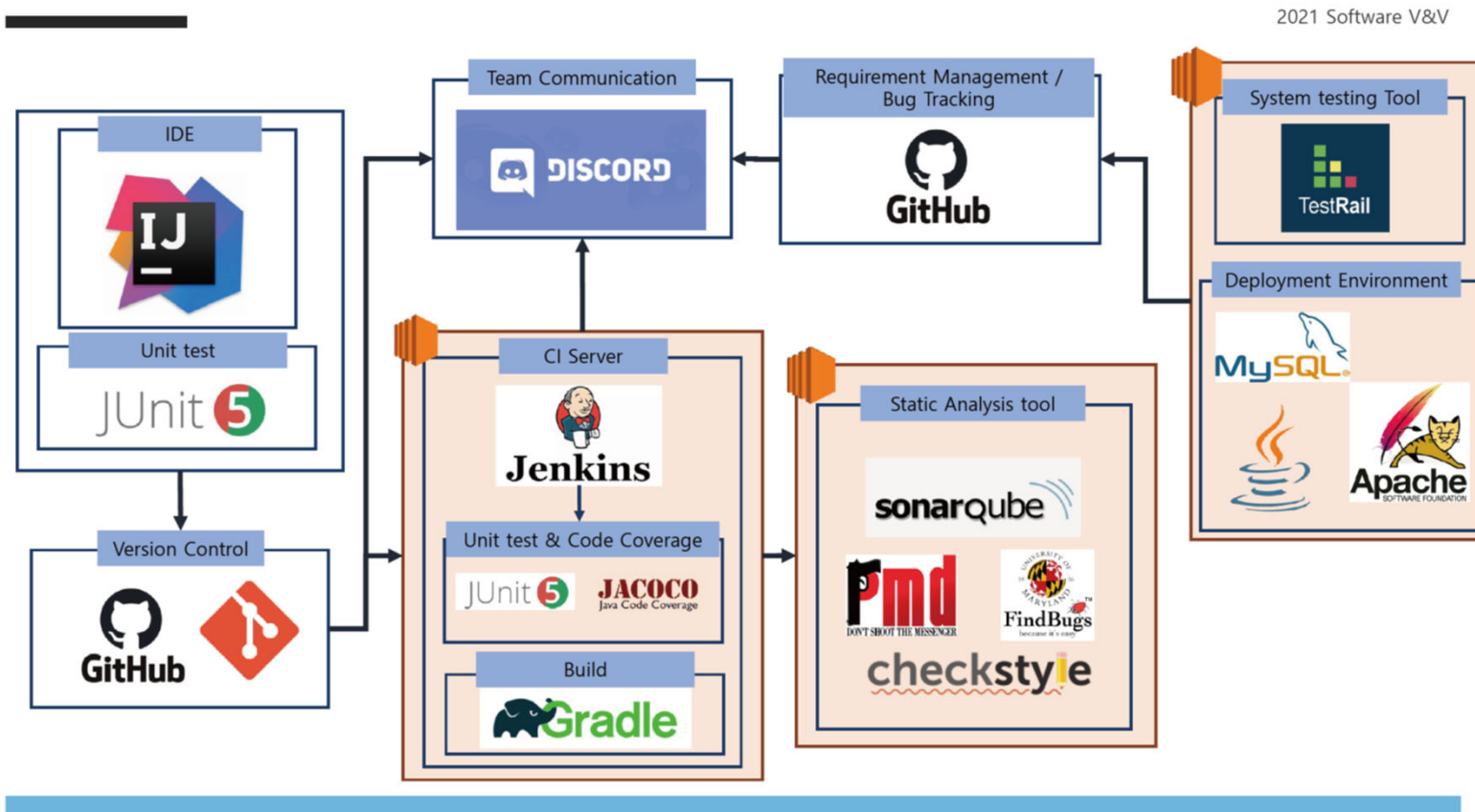
CTIP Examples (KU 2021)



CTIP Examples (KU 2021)



CTIP Examples (KU 2021)



Test-Driven Development

- **TDD (Test-Driven Development)**
 - *“Writing tests before code clarifies the requirements to be implemented.”*
 - Tests are written as programs rather than data so that they can be executed automatically.
 - The test includes a check that it has executed correctly.

- **Automated test execution environment** is mandatory.
 - All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.

Customer Involvement

- **The customer is a part of the team in XP.**
 - Help develop acceptance tests for the stories that are to be implemented in the next release of the system.
 - Writes tests as development proceeds.
 - All new code is therefore validated to ensure that it is what the customer needs.
 - **test case** ≠ test data

- However, customers have limited time available and so cannot work full-time with the development team.
 - They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test Case Description for 'Dose Checking'

Test 4: Dose checking

Input:

1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:

1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:

OK or error message indicating that the dose is outside the safe range.

Test Automation

- **Test automation**

- *“Tests are written as executable components before the task is implemented.”*
- Automated test framework is required.
- Each testing component should
 - Be stand-alone (independent),
 - Simulate the submission of input to be tested, and
 - Check that the result meets the output specification.

- **Automated test framework**

- A system that makes it easy to write executable tests and submit a set of tests for execution
- Examples:
 - A series of **xUnit** (e.g., **JUnit**)
- As testing is automated, a set of tests is always ready to test quickly.
 - Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately, as CTIP.

4. Pair Programming

- **Pair programming**
 - *“Involves programmers working in pairs, developing code together.”*
 - Programmers sit together at the same computer to develop the software.
 - Pairs are created dynamically so that all team members work with each other during the development process.
 - The sharing of knowledge that happens during pair programming is very important, as it reduces the overall risks to a project when team members leave.
 - Advantages:
 - Helps develop **common ownership** of code and spreads knowledge across the team.
 - Serves as an **informal review** process as each line of code is looked at by more than 1 person.
 - Encourages **refactoring** as the whole team can benefit from improving the system code.
- **Pair programming is not necessarily inefficient.**
 - Some evidence suggests that a pair working together is more efficient than 2 programmers working separately.

Agile Project Management

Agile Project Management

- The principal responsibility of **software project managers** is to manage the project so that the software is delivered on time and within the planned budget for the project.
- **The standard approach** to project management is the **plan-driven**.
 - Managers draw up a plan for the project showing
 - What should be delivered,
 - When it should be delivered, and
 - Who will work on the development of the project deliverables.
- **Agile project management** requires a different approach.
 - Should be adapted to incremental development and the practices used in agile (development) methods.
 - **Scrum**

Scrum

- **Scrum**

- An agile method that [focuses on managing iterative development](#) rather than specific agile practices.
- The name of **a short daily meeting**
 - All team members share information, describe their progress since the last meeting, problems that have arisen, and what is planned for the following day.
 - Everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

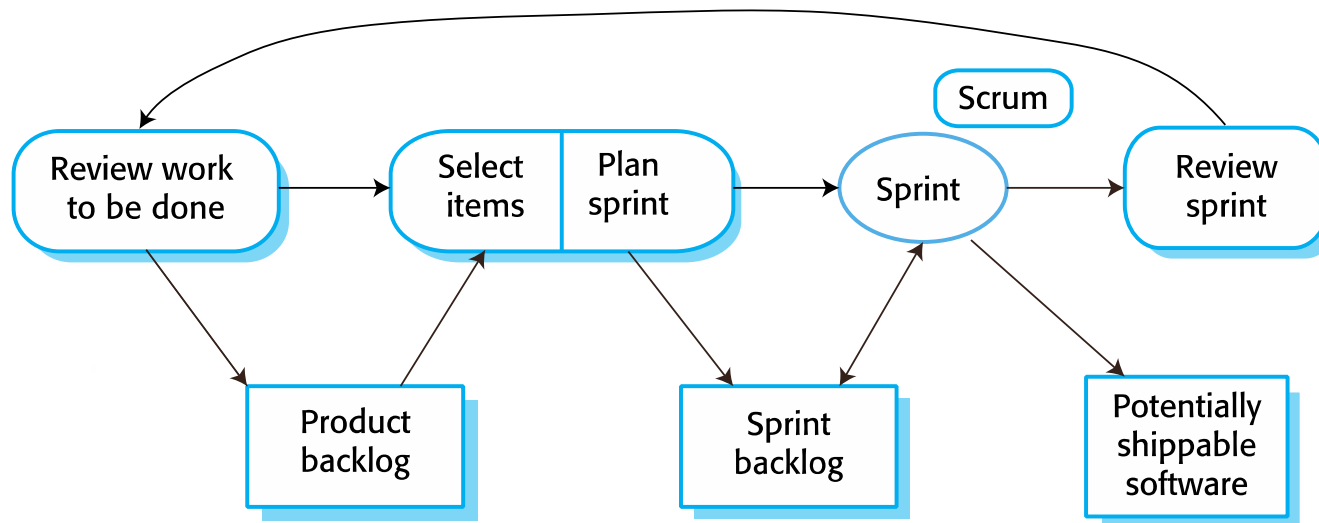
- **3 phases** in Scrum

- **Initial phase**
 - An outline planning phase, where you establish the general objectives for the project and design the software architecture.
- **A series of **sprint** cycles**
 - Each cycle develops an increment of the system. (2~4 weeks for each sprint)
- **Project closure phase**
 - Wraps up the project, completes required documentation, and assesses the lessons learned from the project.

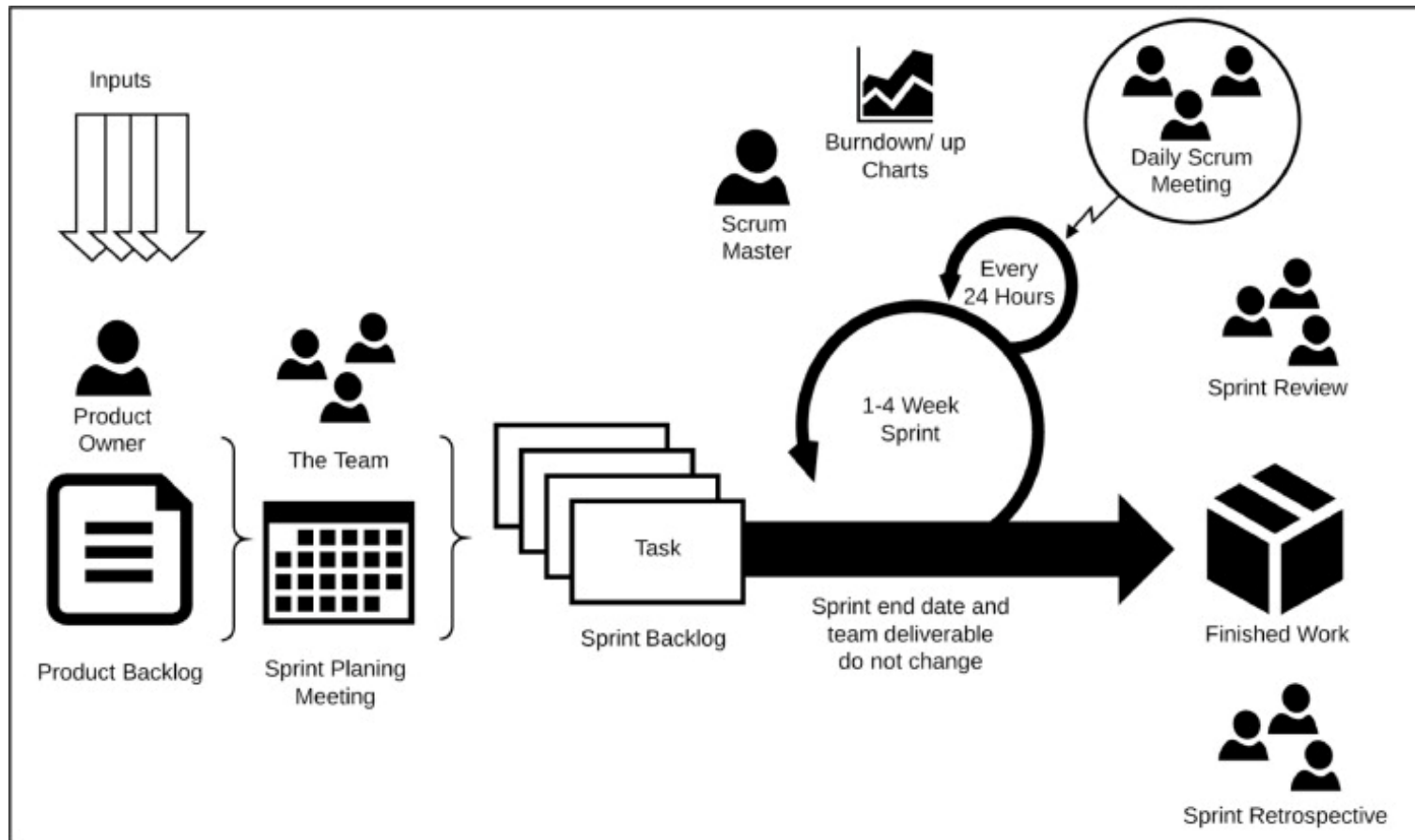
Scrum Terminology

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is <u>a list of 'to do' items</u> which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
Scrum	<u>A daily meeting</u> of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
Scrum Master	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	<u>An estimate of how much product backlog effort</u> that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

The Scrum Sprint Cycles



The Scrum Sprint Cycles



The Scrum Sprint Cycles

- Sprints are fixed length, normally **2~4 weeks**.
- The starting point for planning is the **product backlog**, which is the list of work to be done on the project.
- The selection phase involves all project teams who work with the customer to select the features and functionality from the product backlog to be developed during the sprint (**sprint backlog**).
 - Once these are agreed, the team organize themselves to develop the software.
 - During this stage, the team is isolated from the customer and the organization, with all communications channelled through the so-called 'Scrum master'.
 - The role of the Scrum master is to protect the development team from external distractions.
- At the end of the sprint, the work done is reviewed and presented to stakeholders.
 - The next sprint cycle then begins.

Homework / Activity #3

- **Agile Development Techniques** 사용에 대한 신문기사(사내)를 찾아 보세요.
- **Agile Project Management** 적용에 대한 신문기사(사내)를 찾아 보세요.

Samples from SE Undergraduate (KU 2021)

1. The agile team has gone home. Now what?

Guest

The agile team has gone home. Now what?

Alok Uniyal, Infosys

September 11, 2021 8:40 AM

f t in

[Agile](#) may be synonymous with software development, but it is equally about people. Because the purpose of agile sprints is to incorporate feedback at quick intervals to deliver what customers want. And the agile process itself works best with close collaboration between developer and stakeholder groups, also bringing together development and IT operations teams when used in conjunction with [DevOps](#). So it is no surprise that there is a strong correlation between an enterprise's growth and its agile capability: 6 of the top 7 agile levers by impact were people-related. (My team published some [recent research](#) on this.)

A core principle of agile transformation is to use face-to-face interaction, which went right out the window when the pandemic hit. However, the use of agile has actually increased over the past year while almost everyone around the world was working remotely. Reconciling these seemingly opposite shifts makes for an interesting challenge for enterprises. But it is not an impossible one.

Workforce and workspace levers

Using agile virtual workspaces along with digital collaboration platforms to support remote but collective and cohesive work has been a big driver of success. At my company, we conducted a [study](#) of our own employees right before and just into the pandemic, and it showed that when 3 or more early agile sprints were conducted on-premises with workers coming to the office, it paved the way for the asynchronous communication and remote work that followed. At the same time, using digitized visual Kanban dashboards along with other

[통신] 통신공룡 KT의 변신... '하이브리드 애자일' 실험 뿌리 내린다

B2면 TOP | 기사입력 2020.09.07. 오전 4:04 | 기사원문 | 스크림 | 본문듣기 · 설정

한때 '통신공룡'으로 불리던 KT가 완전히 달라졌다. 3~4년 전부터 디지털 네이티브 기업들과 경쟁하는가 싶더니 올해 구현모 사장(CEO) 취임 이후부터는 아예 '종합 IT서비스 기업' '인공지능(AI) 컴퍼니'를 자처하고 나섰다. 자유로운 의사소통과 토론문화 등이 정착되면서 KT 내부에서는 IT나 게임기업 혹은 스타트업처럼 일하는 것 같다는 반응도 나온다. KT는 어떻게 거대 조직에 '민첩성'과 '유연성'이라는 DNA를 이식할 수 있었을까. 비결은 '하이브리드 애자일'이다.

애자일은 '빠르고 유연하게' 소프트웨어(SW)와 서비스를 개발하는 것을 말한다. 사전 계획이나 문서화 작업보다는 프로그래밍 과정에 초점을 두는 개발 방식으로, 구글 아마존 같은 IT기업이나 테크 스타트업에서 각광받아 왔다. KT는 전 세계 기업 75%가 채택한 스크럼(Scrum) 방법론을 발전시켜 KT만의 '하이브리드 애자일'을 정립했고, 이를 전사 차원으로 확장해 잇단 성공 사례를 만들어냈다.

하이브리드 애자일은 '어떻게 하면 빠르게 개발할 수 있을까'를 고민한 끝에 나온 결과물이다. KT는 기존의 통신 서비스는 물론 시와 빅데이터, 클라우드 등 미래 기술도 동시에 개발해야 하는 회사다. 게다가 기존 전통기업처럼 아웃소싱을 기반으로 한 개발이 많기 때문에 자체 개발하는 스타트업이나 디지털 기업이 사용하는 애자일 방법론을 적용하기가 쉽지 않았다. 기존 스크럼 애자일만으로는 국내 기업에 부과된 법규와 의무 등 이른바 컴플라이언스를 준수하기 어렵다는 점도 문제였다.

오훈용 KT IT부문 플랫폼IT서비스단장은 프로젝트에 따라 레고 블록을 조립하듯 '맞춤형 애자일 매뉴얼'을 만들었다고 설명했다. 오 단장은 "스크럼 표준을 준수하면서 자체 개발에는 '애자일 와우(Agile-wow)' 방법론을, 아웃소싱 개발에는 '애자일 키(Agile-key)' 방법론을 루트랙으로 적용하는 등 유연하게 대응한 것이 하이브리드 애자일의 성공 비결"이라며 "이런 특별한 경험과 노하우를 나누고 함께 혁신하고자 앞으로 고객사에까지 하이브리드 애자일 방법론을 확장할 계획"이라고 말했다.

4. Requirements Engineering

Requirements Engineering

- The **process** of establishing the services that a customer requires from a system and the constraints under which it operates and is developed.
 - **System requirements**: descriptions of the system services and constraints that are generated during the requirements engineering process

- **Requirements**
 - Range from a high-level abstract statements of a service or of system constraints to a detailed mathematical functional specification.
 - Statements of services → **Functional requirements (FR)**
 - System constraint → **Non-functional requirements (NFR)**

Types of Requirement

- **User requirements**

- Statements in natural language and diagrams of the services the system provides and its operational constraints
- Elicited/Discovered from stakeholders
- Defined for customers

- **System requirements**

- A structured document setting out detailed descriptions of the system's functions, services and operational constraints
- Defines what should be implemented
- Specified for developers

User and System Requirements

User requirements definition

- 1.** The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

- 1.1** On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
- 1.2** The system shall generate the report for printing after 17.30 on the last working day of the month.
- 1.3** A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
- 1.4** If drugs are available in different dose units (e.g. 10mg, 20mg, etc) separate reports shall be created for each dose unit.
- 1.5** Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

System Stakeholders

- **Any person or organization** who is **affected by** the system in some way and so who has a legitimate **interest**
- **Typical stakeholders:**

User	Concerned with the features and functionality of the new system
Designer	Want to build a perfect system, or reuse existing code
System Analyst	Want to “get the requirements right”
Training and User Support	Want to make sure the new system is usable and manageable
Business Analyst	Want to make sure “we are doing better than the competition”
Technical Author	Will prepare user manuals and other documentation for the new system
Project Manager	Wants to complete the project on time, within budget, with all objectives met.
Customer	Wants to get best value for money invested

Agile Methods and Requirements

- Many agile methods argue that
 - “*Producing detailed system requirements is a waste of time as requirements change so quickly.*”
 - The requirements document is therefore always out of date.

- Agile methods usually use incremental requirements engineering and may express requirements as **user stories**.
 - This is practical for business systems.
 - This is often problematic for systems that require pre-delivery analysis (e.g., critical systems) or systems developed by several teams.

Functional and Non-Functional Requirements

Functional and Non-Functional Requirements

- **Functional requirements**

- Statements of **services** the system should provide
 - How the system should react to particular inputs.
 - How the system should behave in particular situations.
- May state what the system should not do.

- **Non-functional requirements**

- **Constraints** on the services or functions offered by the system such as
 - timing constraints, constraints on the development process, standards, etc.
- Often apply to **the system as a whole** rather than individual features or services.

- **Domain requirements**

- Constraints on the system from the domain of operation

Functional Requirements

- Describing functionality or system services depends on the type of software, expected users and the type of system where the software is used.
 - **Functional User Requirements** may be high-level statements of what the system should do.
 - **Functional System Requirements** should describe the system services (user requirements) in detail.

- An example of Mentcare System
 - Functional user requirement : *“A user shall be able to search the appointments lists for all clinics.”*
 - Functional system requirement : *“The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.”*

Requirements Imprecision

- Problems arise when functional requirements are not precisely stated.
 - Ambiguous requirements may be interpreted in different ways by developers and users.

- For example, the term ‘search’ in the requirement :

“A user shall be able to search the appointments lists for all clinics.”

- User intention : *Search for a patient name across all appointments in all clinics.*
- Developer interpretation : *Search for a patient name in an individual clinic. User chooses a clinic then search.*

Requirements Completeness and Consistency

- In principle, requirements should be both **complete and consistent (C&C)**.
- **Complete**
 - They should include descriptions of all facilities required.
- **Consistent**
 - There should be no conflicts or contradictions in the descriptions of the system facilities.
- In practice, it is impossible to produce a complete and consistent requirements document.

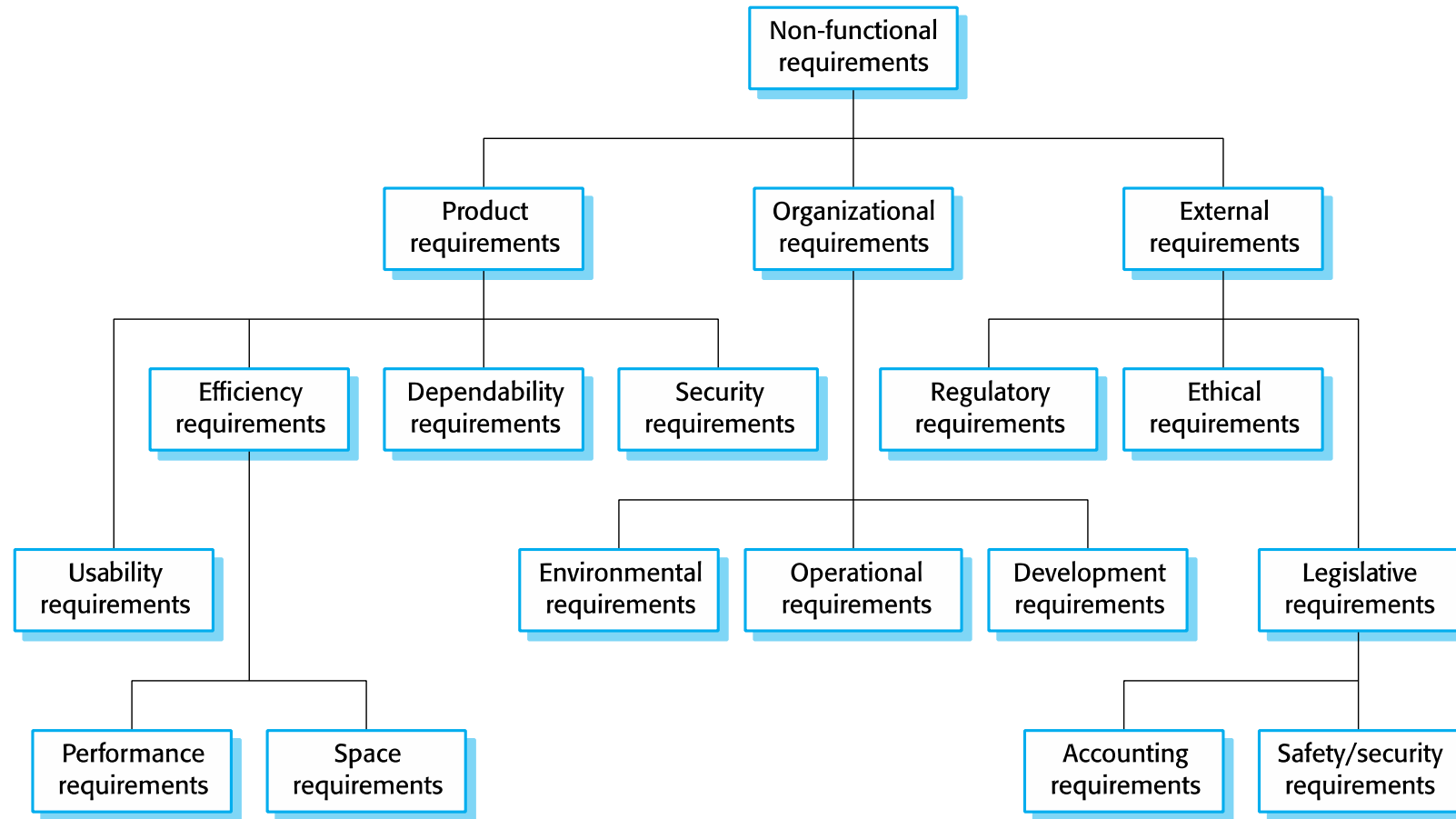
Non-Functional Requirements

- Define system **properties** and **constraints**
 - **Properties**: reliability, response time and storage requirements, I/O device capability, system representations, etc.
 - Quality Attribute Requirements
 - **Constraints**: mandating a particular IDE, programming languages or development methods, or standards compliance (IEC 61508, ISO 26262, IEEE 829,830,1012,1016,12207,25010, etc.)

- Non-functional requirements may be more critical than functional requirements.
 - If these are not met, the system may be useless.

- Non-functional requirements may **affect** the overall architecture of a system.
 - **Generate a number of related functional requirements** that define system services that are required.

Types of Non-Functional Requirements



Non-functional Classifications

- **Product requirements**

- Requirements which specify that the delivered product must behave in a particular way
- E.g., execution speed, reliability, etc.
- *“The Mentcare system shall be available to all clinics during normal working hours (Mon-Fri, 0830-17.30). Downtime within normal working hours shall not exceed five seconds in any one day.”*

- **Organizational requirements**

- Requirements which are a consequence of organisational policies and procedures
- E.g., process standards used, implementation requirements, etc.
- *“Users of the Mentcare system shall authenticate themselves using their health authority identity card.”*

- **External requirements**

- Requirements which arise from factors which are external to the system and its development process
- E.g., interoperability requirements, legislative requirements, etc.
- *“The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.”*

Quality Attributes

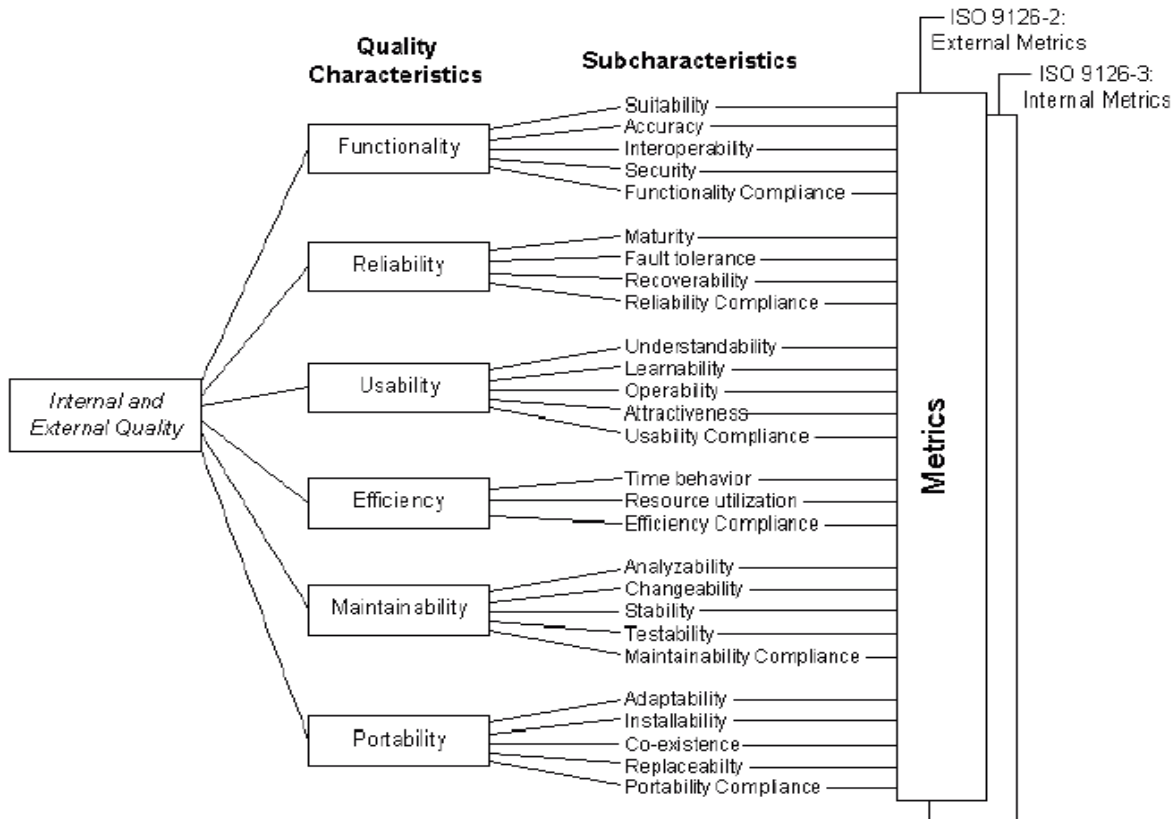
- **Measurable or testable properties of a system**
 - Used to indicate **how well the system satisfies** the needs of its stakeholders
 - Availability, configurability, modifiability, performance, reliability, reusability, security, portability, maintainability, efficiency, usability, many others
 - **Emergent properties** : not a measure of software in isolation
 - Measures the relationship between software and its application domain
 - Cannot measure this until you place the software into its environment
 - Quality will be different in different environments

- Software quality is all about **fitness to purpose** of **stakeholders**.
 - *“Does it do what is needed?”*
 - *“Does it do it in the way that its users need it to?”*
 - *“Does it do it reliably enough? fast enough? safely enough? securely enough?”*
 - *“Will it be affordable? will it be ready when its users need it?”*
 - *“Can it be changed as the needs change?”*

Quality Attributes : Taxonomies

- **-ilities**
 - understandability, usability, modifiability, interoperability, reliability, portability, maintainability, scalability, configurability, customizability, adaptability, variability, volatility, traceability, ...
- **-ities**
 - security, simplicity, clarity, ubiquity, integrity, modularity, ...
- **-ness**
 - user-friendliness, robustness, timeliness, responsiveness, correctness, completeness, conciseness, cohesiveness, ...
- **others**
 - performance, efficiency, accuracy, precision, cost, development time, low coupling, ...

ISO/IEC 9126



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

ISO 9126-1 : Information Technology

- Software Product Quality - Part 1: Quality Model

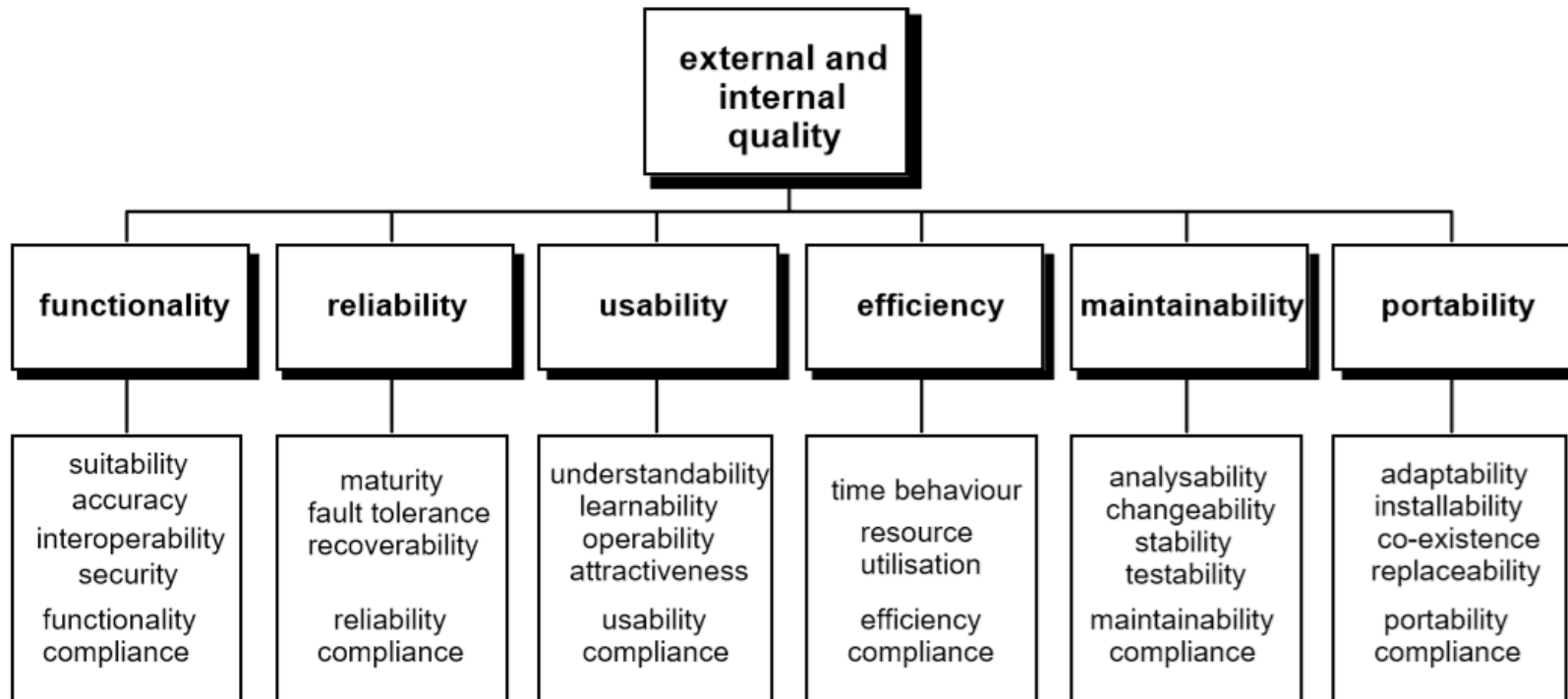


Figure 4 – Quality model for external and internal quality

ISO/IEC 25010

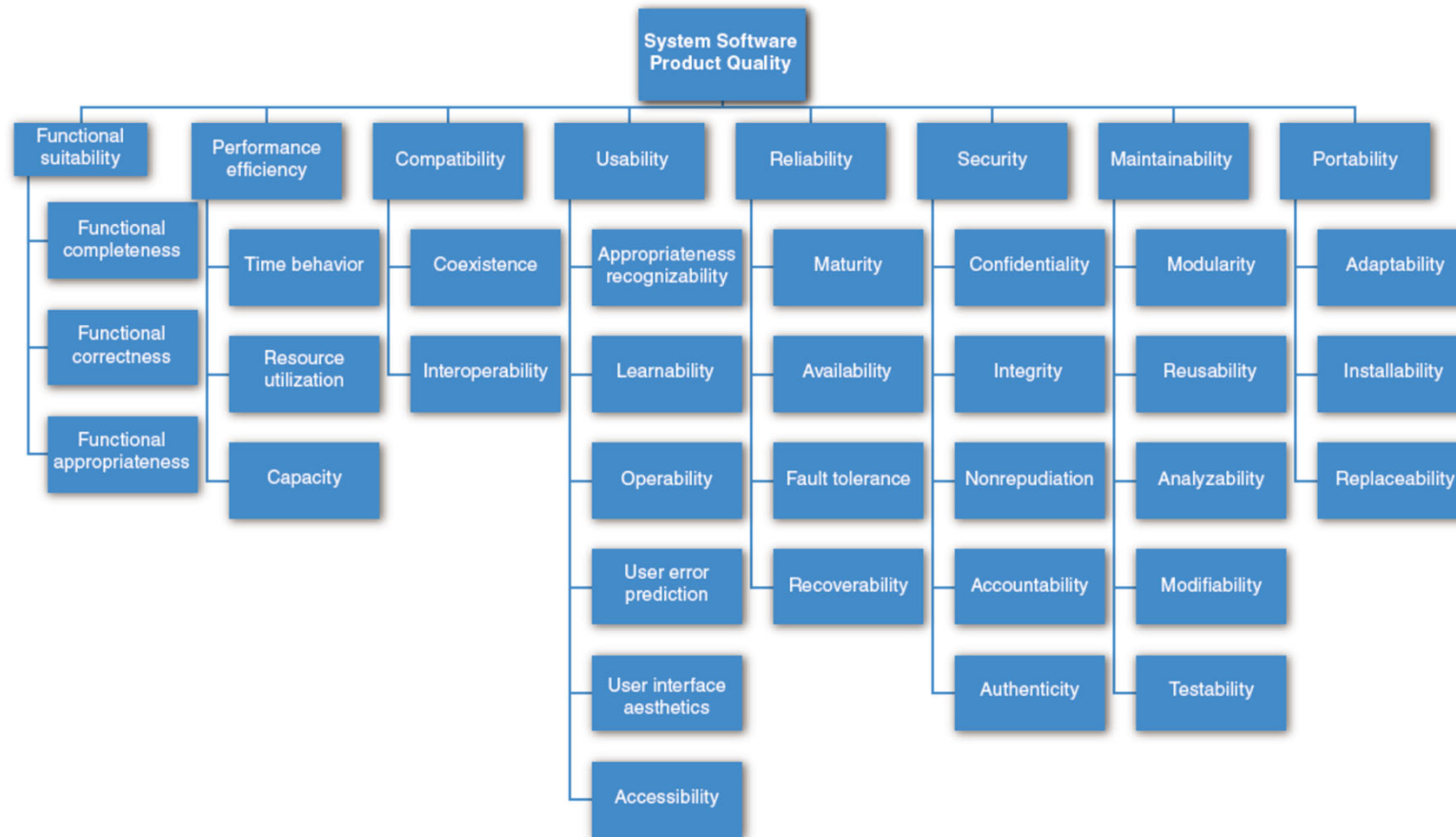


FIGURE 12.1 The ISO/IEC FCD 25010 product quality standard

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

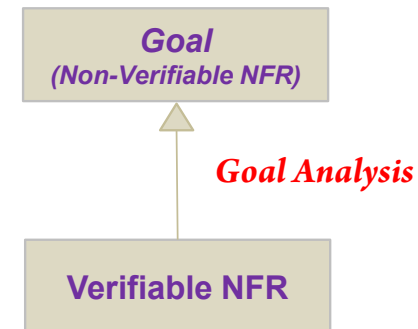
Many of these quality attributes can also be applied to data quality.

Goals and Requirements

- **Non-functional requirements** may be **very difficult to state precisely**.
 - Imprecise requirements may be difficult to verify.
 - Goals are helpful to developers as they convey the intentions of the system users.

- **Goal**
 - A general intention of the user such as ‘ease of use’
 - Often NFR (Quality Attributes)

- **Verifiable Non-Functional Requirement**
 - A statement using some measure that can be objectively tested



Example : Goal and Non-Functional Requirements

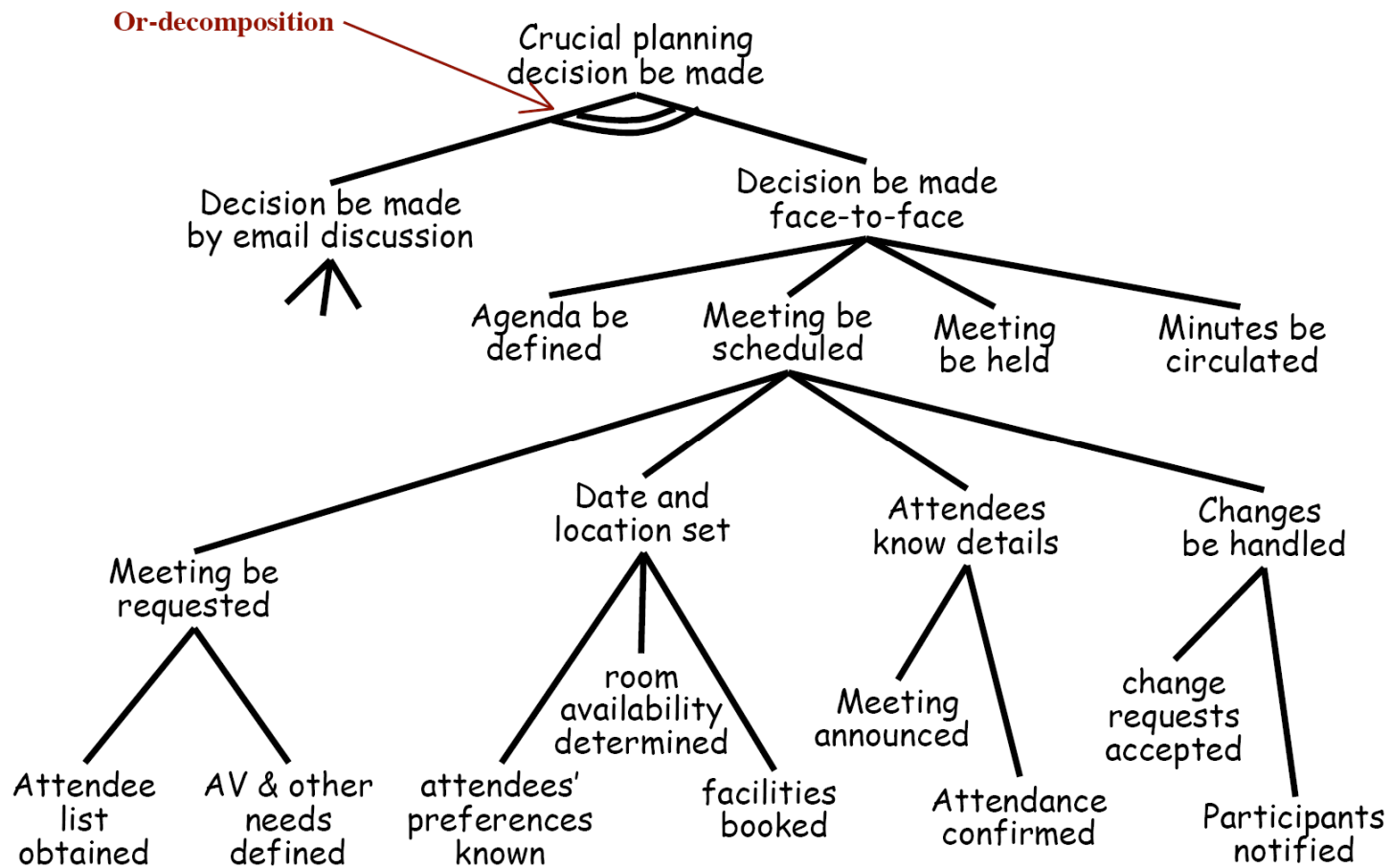
- Quality factor: **Usability**
- Goal:
 - *“The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.”*
- Verifiable non-functional requirement
 - *“Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.”*

Goal Analysis

- **Goal Analysis**
 - Focus on *why* a system is required
 - Express the ‘why’ as a set of stakeholder **goals**
 - **Goal refinement** to arrive at **specific requirements**
 - Document, organize and classify goals
 - **Goal evolution**
 - Refine, elaborate, and operationalize goals
 - **Goal hierarchies** show **refinements** and **alternatives**

- **Goal model** visualizes goal analysis
 - **(Hard) Goal**
 - Describe functions that must be carried out.
 - FR
 - **Soft Goal**
 - Cannot really be fully satisfied such as quality.
 - Accuracy, Performance, Security, etc.
 - NFR (Quality)

Example - Goal Elaboration



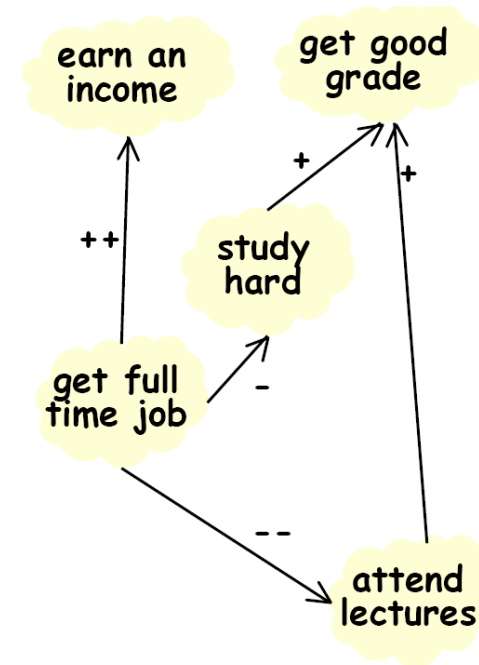
Goal Elaboration

- **Goal Elaboration**

- “**Why**” questions explore **higher** goals (context)
- “**How**” questions explore **lower** goals (operations)
- “**How else**” questions explore **alternatives**

- Relationships between goals

- One goal **helps** achieve another (+)
- One goal **hurts** achievement of another (-)
- One goal **makes** another (++)
 - Achievement of goal A guarantees achievement of goal B
- One goal **breaks** another (--)
- Achievement of goal A prevents achievement of goal B



Soft Goals

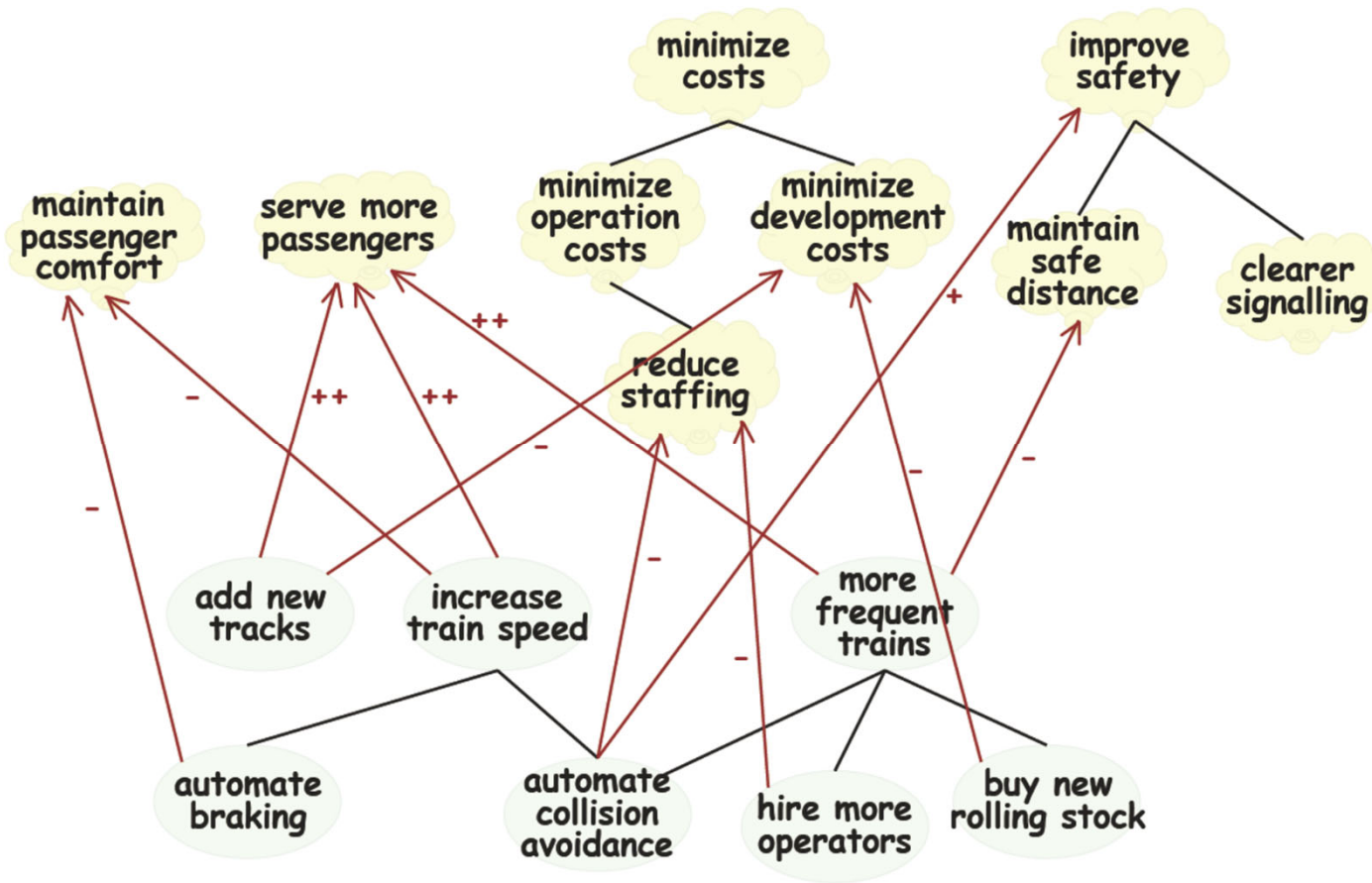
- **Soft Goals:** Goals can never be fully satisfied.
 - E.g., “*system should be easy to use*” , “*access should be secure*”
 - Also known as **NFR**(Non-Functional Requirements) or **Quality attributes/requirements**
 - We have to look for things that contribute to satisfying soft goals.

- Example: a **train system**, we identified 3 soft goals.



Soft Goals as Selection Criteria

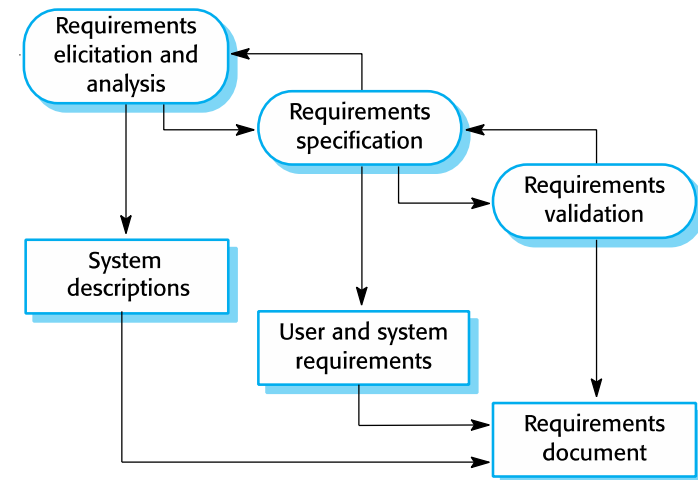
- Goal Analysis



Requirements Engineering Processes

Requirements Engineering Processes

- The **RE process varies widely** depending on
 - the application domain
 - the software development process applied
 - the people/organization developing the requirements
- **4 common activities** common to all processes:
 1. **Requirements elicitation & analysis**
 2. **Requirements specification**
 3. **Requirements validation**
 4. **Requirements change management**



- In practice, RE is an iterative activity in which these processes are interleaved.

1. Requirements Elicitation

- Called **requirements elicitation** or **requirements discovery**.
 - Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.

- **Difficulties in requirements elicitation:**
 - Stakeholders don't know what they really want.
 - Stakeholders express requirements in their own terms.
 - Different stakeholders may have conflicting requirements.
 - Organizational and political factors may influence the system requirements.
 - The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

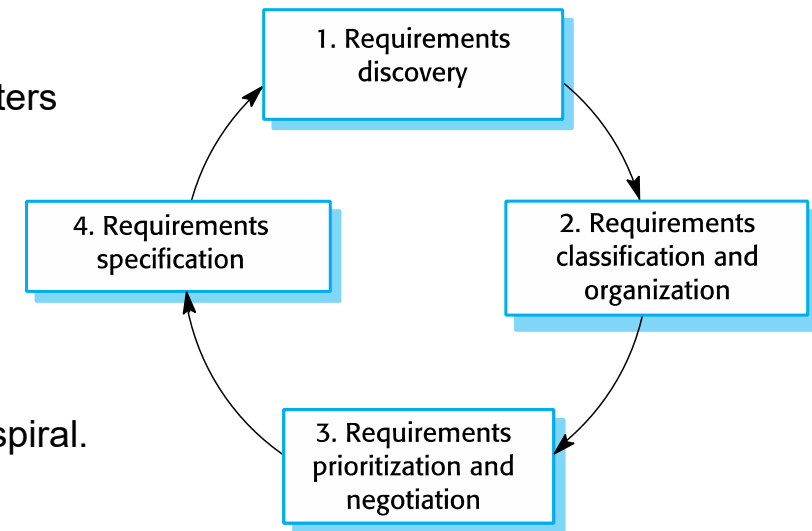
Process Activities in Requirements Elicitation

- **Requirements discovery**
 - Interacting with stakeholders to discover their requirements
 - Domain requirements are also discovered at this stage.

- **Requirements classification and organization**
 - Groups related requirements and organises them into coherent clusters

- **Prioritization and negotiation**
 - Prioritizing requirements and resolving requirements conflicts

- **Requirements specification**
 - Requirements are documented and input into the next round of the spiral.



Requirements Discovery

- Techniques for **requirements discovery**:
 1. [Requirements workshop](#)
 2. Brainstorming
 3. Storyboards (Use-Case scenario)
 4. Interviews
 5. Questionnaires
 6. Role playing
 7. Prototypes
 8. Customer requirement specification review

2. Requirements Specification

- The process of writing down the user and system requirements in a requirements document.
 - **User requirements** have to be understandable by end-users and customers who do not have a technical background.
 - **System requirements** are more detailed requirements and may include more technical information.

- The requirements may be part of a contract for the system development.
 - Requirements should state what the system should do, and the design should describe how it does this.
 - In practice, requirements and design are often inseparable.

Ways of Writing a System Requirements Specification

Notation	Description
Natural language	The requirements are written using numbered sentences in natural language . Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template . Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification .

Natural Language Specification

- Requirements are written as **natural language sentences** supplemented by **diagrams and tables**.
 - Used for writing requirements because it is expressive, intuitive and universal.
- Difficulties in writing requirements in natural languages
 - **Lack of clarity**
 - Precision is difficult without making the document difficult to read.
 - **Requirements confusion**
 - Functional and non-functional requirements tend to be mixed-up.
 - **Requirements amalgamation**
 - Several different requirements may be expressed together.
- **Guidelines :**
 - Invent a standard format and use it for all requirements.
 - Use language in a consistent way.
 - Use shall for mandatory requirements, should for desirable requirements.
 - Use text highlighting to identify key parts of the requirement.
 - Avoid the use of computer jargon.
 - Include an explanation (rationale) of why a requirement is necessary.

Insulin Pump: Natural Language Specification

- 3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes.
(Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.)
- 3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1.
(A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)

Structured Specification

- An approach to writing requirements where the freedom of the requirements writer is limited, and requirements are [written in a standard way](#).
 - Works well for some types of requirements such as requirements for embedded control system.
- Examples:
 - **Form-based specification**
 - **Tabular specification**
 - **Use-Case (Description Table)**

Condition	Action
Sugar level falling ($r2 < r1$)	CompDose = 0
Sugar level stable ($r2 = r1$)	CompDose = 0
Sugar level increasing and rate of increase decreasing ($(r2 - r1) < (r1 - r0)$)	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ($(r2 - r1) \geq (r1 - r0)$)	CompDose = round $((r2 - r1)/4)$ If rounded result = 0 then CompDose = MinimumDose

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading ($r2$); the previous two readings ($r0$ and $r1$).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

Action

CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requirements

Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition

The insulin reservoir contains at least the maximum allowed single dose of insulin.

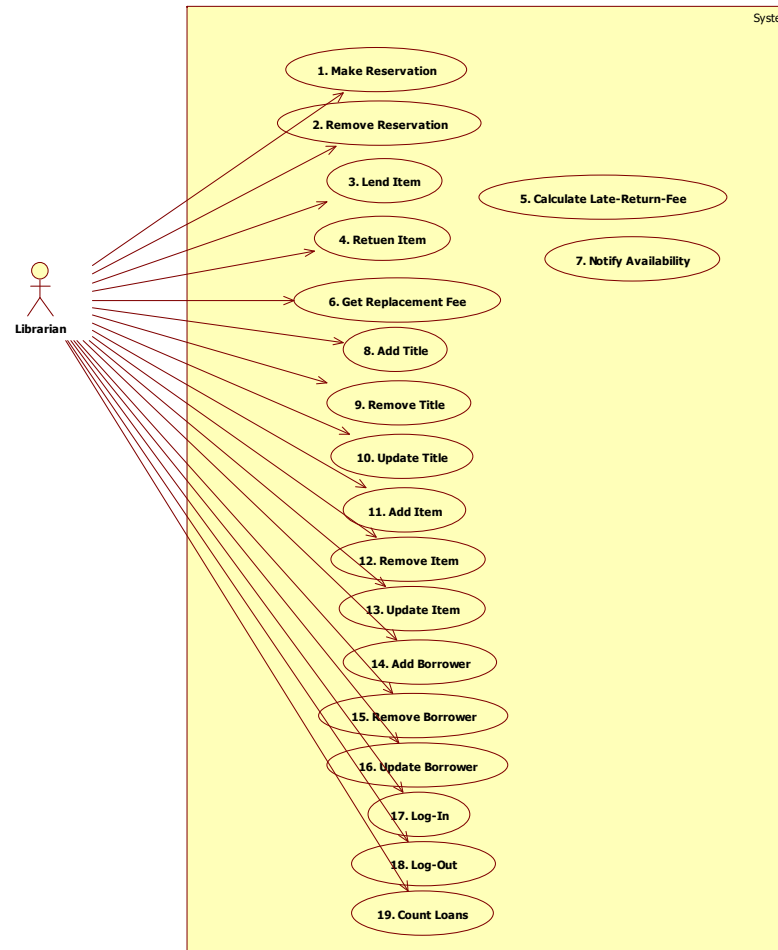
Post-condition $r0$ is replaced by $r1$ then $r1$ is replaced by $r2$.

Side effects None.

Use Cases

- Use-cases are a kind of scenario that are included in the **UML**.
 - Identify the actors in an interaction and which describe the interaction itself
- A set of use cases should describe all possible interactions with the system.
 - High-level graphical model (**UML Use-Case Diagram**) is used to summarize all use-cases.
 - **UML Sequence Diagrams** may be used to add detail to use-cases by showing the sequence of event processing in the system.

Library Management System: Use-Case Diagram



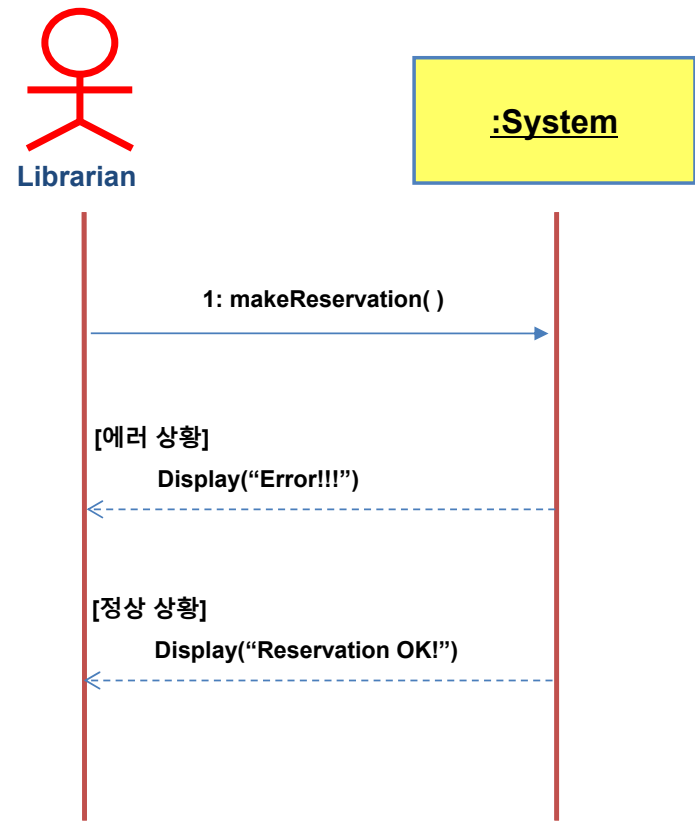
Library Management System: Use-Case Description

Use Case	1. Make Reservation
Actor	Librarian
Purpose	(As in the business use case)
Overview	(As in the business use case)
Type	Primary and Essential
Cross Reference	System Functions: R1.1, R3.1 Use Case: "Add Borrower"
Pre-Requisites	Borrower should have an id_card.
Typical Courses of Events	(A) : Actor, (S) : System 1. (A) A librarian requests the reservation of title 2. (S) Check if a corresponding title exists 3. (S) Check if a corresponding borrower exists 4. (S) If the borrower does not exist, invoke "Add Borrower" 5. (S) Create reservation information
Alternative Courses of Events	N/A
Exceptional Courses of Events	Line 1: If invalid reservation information is entered, indicate an error.

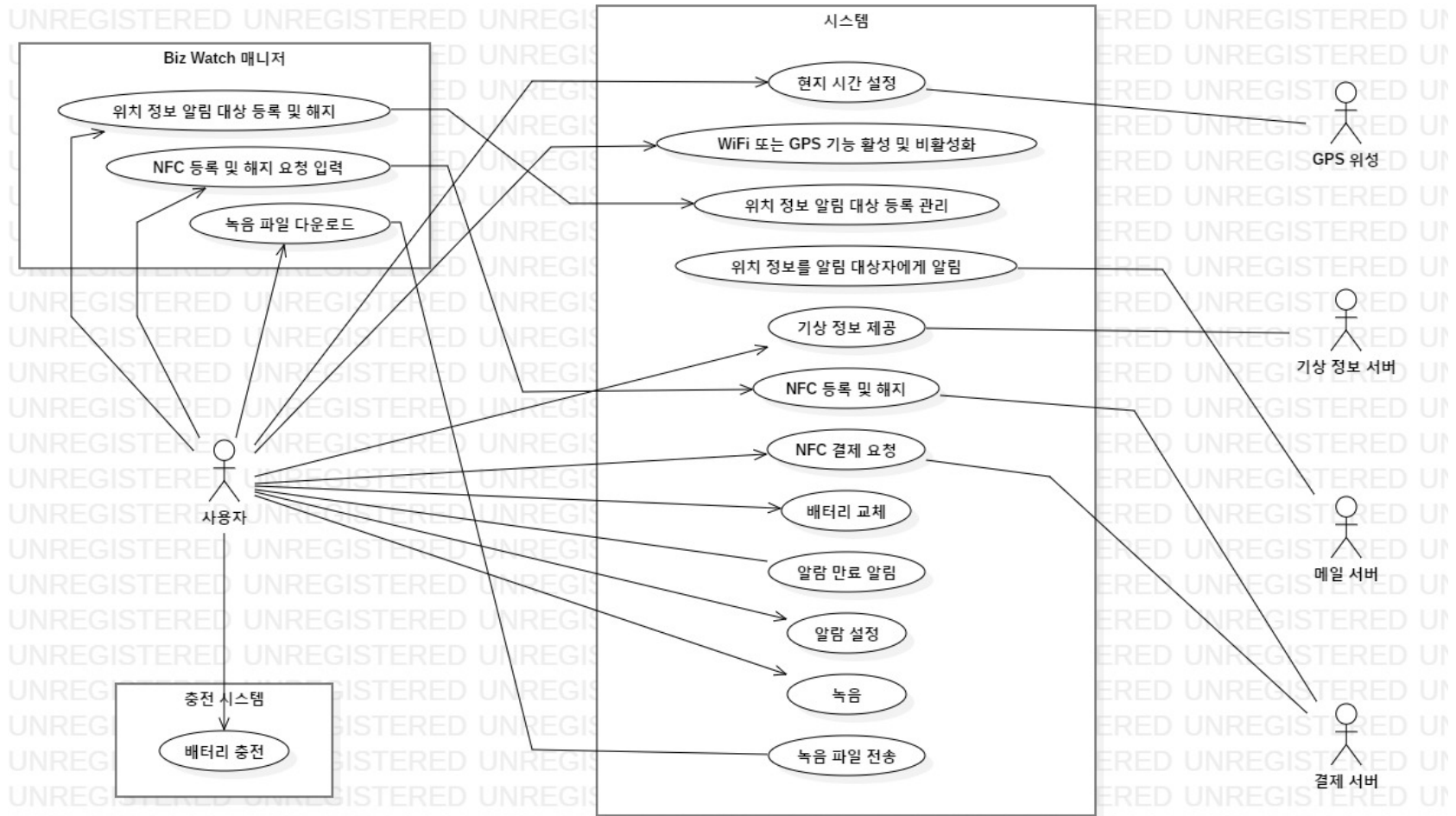
Library Management System: System Sequence Diagram

USE CASE: 1. Make Reservation

1. (A) A librarian requests the reservation of title.
2. (S) Check if corresponding title exist.
3. (S) Check if corresponding borrower exist.
4. (S) If the borrower does not exist, invoke "Add Borrower".
(→ connect to the other Use Case)
5. (S) Create reservation information.



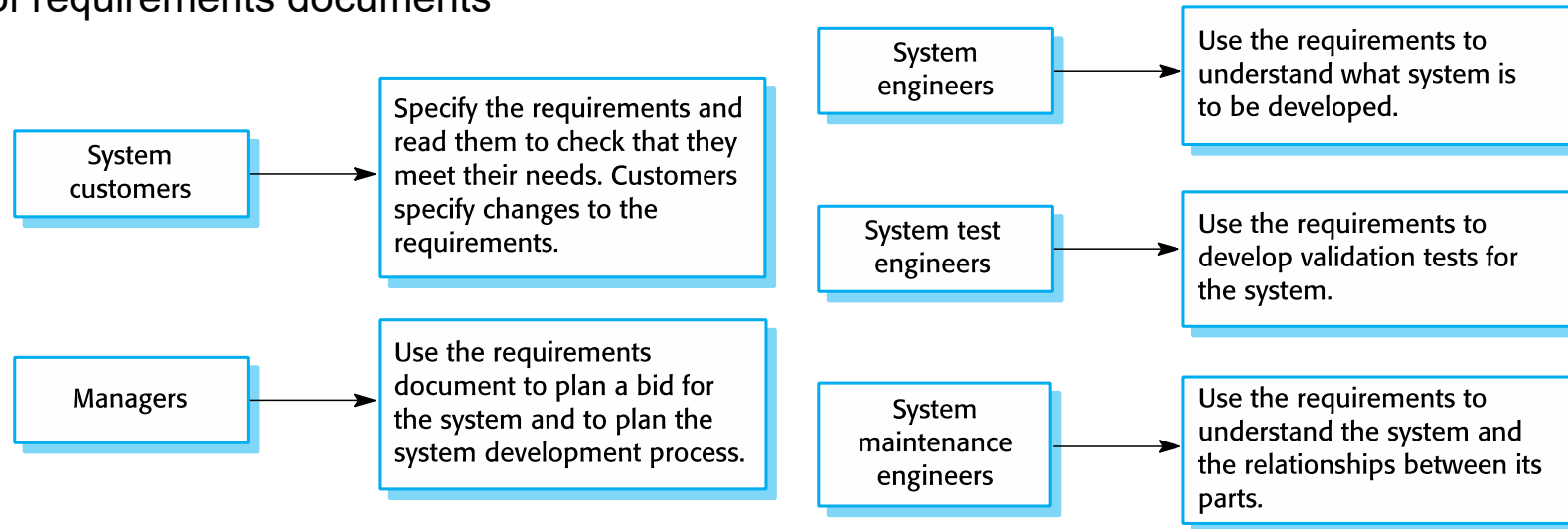
Advanced Smart Watch: Use-Case Diagram



The Software Requirements Document

- The software requirements document is **an official statement** of what is required of the system developers.
 - Should include both a definition of user requirements and a specification of the system requirements.
 - It is NOT a design document.
 - As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

Users of requirements documents



Features for Good Specifications

Features	Considerations
Valid (Correct)	<ul style="list-style-type: none"> - Expresses the real needs of the stakeholders (customers, users,...) - Does not contain anything that is not “required”
Unambiguous	<ul style="list-style-type: none"> - Every statement can be read in exactly one way
Complete	<ul style="list-style-type: none"> - All the things the system must do and all the things it must not do! - Conceptual Completeness <ul style="list-style-type: none"> • E.g., responses to all classes of input - Structural Completeness <ul style="list-style-type: none"> • E.g., no TBDs!!!
Understandable (Clear)	<ul style="list-style-type: none"> - E.g., by non-computer specialists
Consistent	<ul style="list-style-type: none"> - Doesn't contradict itself - Uses all terms consistently
Ranked	<ul style="list-style-type: none"> - Indicates relative importance / stability of each requirement
Verifiable	<ul style="list-style-type: none"> - A process exists to test satisfaction of each requirement
Modifiable	<ul style="list-style-type: none"> - Can be changed without difficulty <ul style="list-style-type: none"> • Good structure and cross-referencing
Traceable	<ul style="list-style-type: none"> - Origin of each requirement is clear - Labels each requirement for future referencing

SRS Contents

- **Software Requirements Specification** should address:
 - **Functionality**
 - What is the software supposed to do?
 - **External interfaces**
 - How does the software interact with people, the system's hardware, other hardware, and other software?
 - What assumptions can be made about these external entities?
 - **Required performance**
 - What is the speed, availability, response time, recovery time of various software functions, and so on?
 - **Quality attributes**
 - What are the portability, correctness, maintainability, security, and other considerations?
 - **Design constraints imposed on an implementation**
 - Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) and so on?

Typical Mistakes in SRS

Mistakes	Description
Noise	text that carries no relevant information to any feature of the problem
Silence	a feature that is not covered by any text
Over-Specification	text that describes a detailed design decision, rather than the problem
Contradiction	text that defines a single feature in a number of incompatible ways
Ambiguity	text that can be interpreted in at least two different ways
Forward Reference	text that refers to a terms or features yet to be defined
Wishful Thinking	text that defines a feature that cannot possibly be verified
Requirements on Users	Cannot require users to do certain things, can only assume that they will
Jigsaw Puzzles	Distributing key information across a document and then cross-referencing
Duck Speak Requirements	Requirements that are only there to conform to standards
Unnecessary Invention of Terminology	e.g., 'user input presentation function'
Inconsistent Terminology	Inventing and then changing terminology
Putting the onus on the developers	i.e., making the reader work hard to decipher the intent
Writing for the hostile reader	There are fewer of these than friendly readers

Requirements Document Variability

- Information in requirements document depends on the type of system and the approach to development used.
 - If systems are developed incrementally, it will typically have less detail in the requirements document.

- **Requirements documents standards** have been designed.
 - **IEEE standards 830-1998**
 - Mostly applicable to the requirements for large systems engineering projects

SRS Standard: IEEE Std 830-1998

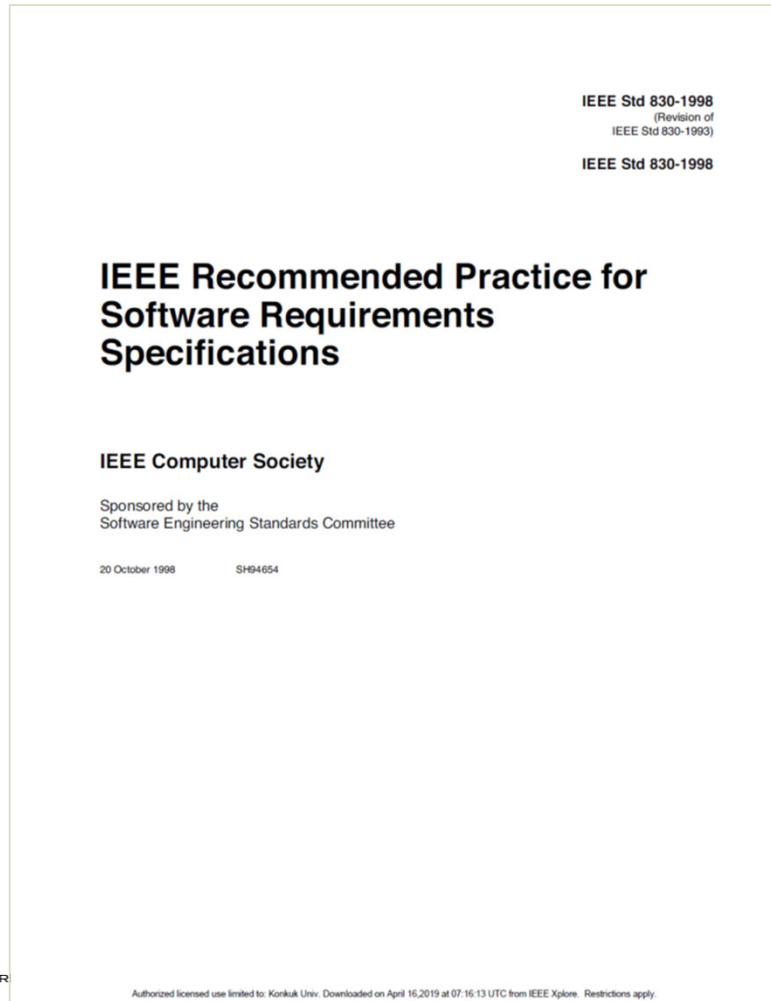


Table of Contents	
1.	Introduction
1.1	Purpose
1.2	Scope
1.3	Definitions, acronyms, and abbreviations
1.4	References
1.5	Overview
2.	Overall description
2.1	Product perspective
2.2	Product functions
2.3	User characteristics
2.4	Constraints
2.5	Assumptions and dependencies
3.	Specific requirements (See 5.3.1 through 5.3.8 for explanations of possible specific requirements. See also Annex A for several different ways of organizing this section of the SRS.)
	Appendixes
	Index

Figure 1 – Prototype SRS outline

SRS Templates: IEEE Std 830-1998

A.1 Template of SRS Section 3 organized by mode: Version 1

- 3. Specific requirements
 - 3.1 External interface requirements
 - 3.1.1 User interfaces
 - 3.1.2 Hardware interfaces
 - 3.1.3 Software interfaces
 - 3.1.4 Communications interfaces
 - 3.2 Functional requirements
 - 3.2.1 Mode 1
 - 3.2.1.1 Functional requirement 1.1
 - .
 - .
 - .
 - 3.2.1.*n* Functional requirement 1.*n*
 - 3.2.2 Mode 2
 - .
 - .
 - .
 - 3.2.*m* Mode *m*
 - 3.2.*m*.1 Functional requirement *m*.1
 - .
 - .
 - .
 - 3.2.*m*.*n* Functional requirement *m*.*n*
 - 3.3 Performance requirements
 - 3.4 Design constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

A.2 Template of SRS Section 3 organized by mode: Version 2

- 3. Specific requirements
 - 3.1. Functional requirements
 - 3.1.1 Mode 1
 - 3.1.1.1 External interfaces
 - 3.1.1.1.1 User interfaces
 - 3.1.1.1.2 Hardware interfaces
 - 3.1.1.1.3 Software interfaces
 - 3.1.1.1.4 Communications interfaces
 - 3.1.1.2 Functional requirements
 - 3.1.1.2.1 Functional requirement 1
 - .
 - .
 - .
 - 3.1.1.2.*n* Functional requirement *n*
 - 3.1.1.3 Performance
 - 3.1.2 Mode 2
 - .
 - .
 - .
 - 3.1.*m* Mode *m*
 - 3.2 Design constraints
 - 3.3 Software system attributes
 - 3.4 Other requirements

SRS Templates: IEEE Std 830-1998

A.3 Template of SRS Section 3 organized by user class

- 3. Specific requirements
 - 3.1 External interface requirements
 - 3.1.1 User interfaces
 - 3.1.2 Hardware interfaces
 - 3.1.3 Software interfaces
 - 3.1.4 Communications interfaces
 - 3.2 Functional requirements
 - 3.2.1 User class 1
 - 3.2.1.1 Functional requirement 1.1
 - .
 - .
 - 3.2.1.*n* Functional requirement 1.*n*
 - 3.2.2 User class 2
 - .
 - .
 - 3.2.*m* User class *m*
 - 3.2.*m*.1 Functional requirement *m*.1
 - .
 - .
 - 3.2.*m*.*n* Functional requirement *m*.*n*
 - 3.3 Performance requirements
 - 3.4 Design constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

A.4 Template of SRS Section 3 organized by object

- 3. Specific requirements
 - 3.1 External interface requirements
 - 3.1.1 User interfaces
 - 3.1.2 Hardware interfaces
 - 3.1.3 Software interfaces
 - 3.1.4 Communications interfaces
 - 3.2 Classes/Objects
 - 3.2.1 Class/Object 1
 - 3.2.1.1 Attributes (direct or inherited)
 - 3.2.1.1.1 Attribute 1
 - .
 - .
 - 3.2.1.1.*n* Attribute *n*
 - 3.2.1.2 Functions (services, methods, direct or inherited)
 - 3.2.1.2.1 Functional requirement 1.1
 - .
 - .
 - 3.2.1.2.*m* Functional requirement 1.*m*
 - 3.2.1.3 Messages (communications received or sent)
 - 3.2.2 Class/Object 2
 - .
 - .
 - 3.2.*p* Class/Object *p*
 - 3.3 Performance requirements
 - 3.4 Design constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

SRS Templates: IEEE Std 830-1998

A.5 Template of SRS Section 3 organized by feature

- 3. Specific requirements
 - 3.1 External interface requirements
 - 3.1.1 User interfaces
 - 3.1.2 Hardware interfaces
 - 3.1.3 Software interfaces
 - 3.1.4 Communications interfaces
 - 3.2 System features
 - 3.2.1 System Feature 1
 - 3.2.1.1 Introduction/Purpose of feature
 - 3.2.1.2 Stimulus/Response sequence
 - 3.2.1.3 Associated functional requirements
 - 3.2.1.3.1 Functional requirement 1
 - .
 - .
 - 3.2.1.3.n Functional requirement *n*
 - 3.2.2 System feature 2
 - .
 - .
 - 3.2.m System feature *m*
 - .
 - .
 - 3.3 Performance requirements
 - 3.4 Design constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

A.6 Template of SRS Section 3 organized by stimulus

- 3. Specific requirements
 - 3.1 External interface requirements
 - 3.1.1 User interfaces
 - 3.1.2 Hardware interfaces
 - 3.1.3 Software interfaces
 - 3.1.4 Communications interfaces
 - 3.2 Functional requirements
 - 3.2.1 Stimulus 1
 - 3.2.1.1 Functional requirement 1.1
 - .
 - .
 - 3.2.1.n Functional requirement 1.n
 - 3.2.2 Stimulus 2
 - .
 - .
 - 3.2.m Stimulus m
 - 3.2.m.1 Functional requirement *m.1*
 - .
 - .
 - 3.2.m.n Functional requirement *m.n*
 - 3.3 Performance requirements
 - 3.4 Design constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

SRS Templates: IEEE Std 830-1998

A.7 Template of SRS Section 3 organized by functional hierarchy

- 3. Specific requirements
 - 3.1 External interface requirements
 - 3.1.1 User interfaces
 - 3.1.2 Hardware interfaces
 - 3.1.3 Software interfaces
 - 3.1.4 Communications interfaces
 - 3.2 Functional requirements
 - 3.2.1 Information flows
 - 3.2.1.1 Data flow diagram 1
 - 3.2.1.1.1 Data entities
 - 3.2.1.1.2 Pertinent processes
 - 3.2.1.1.3 Topology
 - 3.2.1.2 Data flow diagram 2
 - 3.2.1.2.1 Data entities
 - 3.2.1.2.2 Pertinent processes
 - 3.2.1.2.3 Topology
 - 3.3 Performance requirements
 - 3.4 Design constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

- 3.2.1.n.1 Data entities
- 3.2.1.n.2 Pertinent processes
- 3.2.1.n.3 Topology
- 3.2.2 Process descriptions
 - 3.2.2.1 Process 1
 - 3.2.2.1.1 Input data entities
 - 3.2.2.1.2 Algorithm or formula of process
 - 3.2.2.1.3 Affected data entities
 - 3.2.2.2 Process 2
 - 3.2.2.2.1 Input data entities
 - 3.2.2.2.2 Algorithm or formula of process
 - 3.2.2.2.3 Affected data entities
 - .
 - .
 - 3.2.2.m Process *m*
 - 3.2.2.m.1 Input data entities
 - 3.2.2.m.2 Algorithm or formula of process
 - 3.2.2.m.3 Affected data entities
 - 3.2.3 Data construct specifications
 - 3.2.3.1 Construct 1
 - 3.2.3.1.1 Record type
 - 3.2.3.1.2 Constituent fields
 - 3.2.3.2 Construct 2
 - 3.2.3.2.1 Record type
 - 3.2.3.2.2 Constituent fields
 - .
 - .
 - 3.2.3.p Construct *p*
 - 3.2.3.p.1 Record type
 - 3.2.3.p.2 Constituent fields
 - 3.2.4 Data dictionary
 - 3.2.4.1 Data element 1
 - 3.2.4.1.1 Name
 - 3.2.4.1.2 Representation
 - 3.2.4.1.3 Units/Format
 - 3.2.4.1.4 Precision/Accuracy
 - 3.2.4.1.5 Range
 - 3.2.4.2 Data element 2
 - 3.2.4.2.1 Name
 - 3.2.4.2.2 Representation
 - 3.2.4.2.3 Units/Format
 - 3.2.4.2.4 Precision/Accuracy
 - 3.2.4.2.5 Range
 - .
 - .
 - 3.2.4.q Data element *q*
 - 3.2.4.q.1 Name
 - 3.2.4.q.2 Representation
 - 3.2.4.q.3 Units/Format
 - 3.2.4.q.4 Precision/Accuracy
 - 3.2.4.q.5 Range

SRS Templates: IEEE Std 830-1998

A.8 Template of SRS Section 3 showing multiple organizations

- 3. Specific requirements
 - 3.1 External interface requirements
 - 3.1.1 User interfaces
 - 3.1.2 Hardware interfaces
 - 3.1.3 Software interfaces
 - 3.1.4 Communications interfaces
 - 3.2 Functional requirements
 - 3.2.1 User class 1
 - 3.2.1.1 Feature 1.1
 - 3.2.1.1.1 Introduction/Purpose of feature
 - 3.2.1.1.2 Stimulus/Response sequence
 - 3.2.1.1.3 Associated functional requirements
 - 3.2.1.2 Feature 1.2
 - 3.2.1.2.1 Introduction/Purpose of feature
 - 3.2.1.2.2 Stimulus/Response sequence
 - 3.2.1.2.3 Associated functional requirements
 - .
 - .
 - .
 - 3.2.1.*m* Feature 1.*m*
 - 3.2.1.*m*.1 Introduction/Purpose of feature
 - 3.2.1.*m*.2 Stimulus/Response sequence
 - 3.2.1.*m*.3 Associated functional requirements
 - 3.2.2 User class 2
 - .
 - .
 - .
 - 3.2.*n* User class *n*
 - .
 - .
 - .
 - 3.3 Performance requirements
 - 3.4 Design constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

3. Requirements Validation

- Concerned with **demonstrating** that the requirements define the system that the customer really wants.
 - Requirements error costs are high, so validation is very important
 - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

- Requirements checking
 - **Validity** : Does the system provide the functions which best support the customer's needs?
 - **Consistency** : Are there any requirements conflicts?
 - **Completeness** : Are all functions required by the customer included?
 - **Realism** : Can the requirements be implemented given available budget and technology
 - **Verifiability** : Can the requirements be checked?

Requirements Validation Techniques

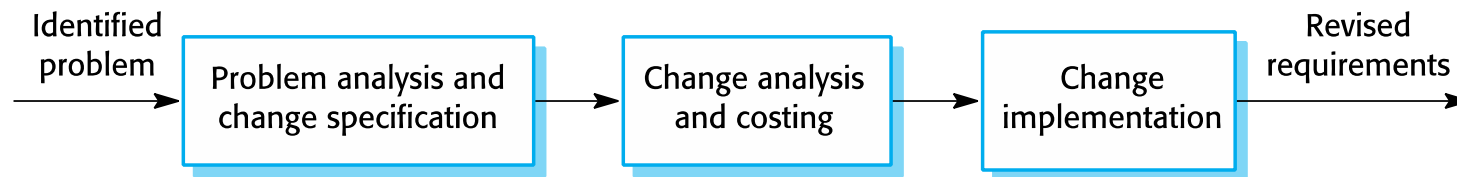
- **Requirements reviews**
 - Systematic manual analysis of the requirements

- **Prototyping**
 - Using an executable model of the system to check requirements

- **Test-case generation**
 - Developing tests for requirements to check testability

4. Requirements Change Management

- **Requirements change management** is the process of managing changing requirements during **the requirements engineering process** and **system development**, and even **after delivery**
 - We need to keep track of individual requirements and **maintain links** between dependent requirements so that you can assess the impact of requirements changes.
 - Need to establish a **formal process for making change proposals** and linking these to system requirements.
 - Decides if a requirements change should be accepted or not.



- **Requirement change management tools** start traceability analysis from requirements to code and TC.
 - CTIP (Continuous Testing and Integration Platform) is useful.

Homework / Activity #4

- 본인이 최근에 수행한 프로젝트 하나를 대상으로, ISO/IEC 9126₍₂₅₀₁₀₎이 제시하는 Software Quality 중 이 프로젝트에 해당되는 항목을 찾고 자세히 분석하세요.
 - 해당되는 항목에 대한 Quality Requirements를 작성하세요.

프로젝트 개요		간단한 설명
Quality Attributes		QA1, QA2, QA3, ... , Responsiveness (예)
QA Reqmts	QA1	Quality Requirements
	QA2	
	QA3	
	...	
	Responsiveness	“이 시스템은 5초 내에 무조건 반응해야 한다.”

5. System Modeling

System Modeling

- **System modeling** is the process of **developing abstract models of a system**, with each model presenting **a different view or perspective** of that system.
 - Helping analysts to understand the functionality of the system
 - Helping analysts to communicate with customers
 - Mostly based on notations in the **Unified Modeling Language (UML)**

- **System perspectives (Views)**
 - **External perspective**: modeling the context or environment of the system
 - **Interaction perspective**: modeling the interactions between a system and its environment, or between the components of a system
 - **Structural perspective**: modeling the organization of a system or the structure of the data processed by the system
 - **Behavioral perspective**: modeling the dynamic behavior of the system and how it responds to events

Use of Graphical Models - UML

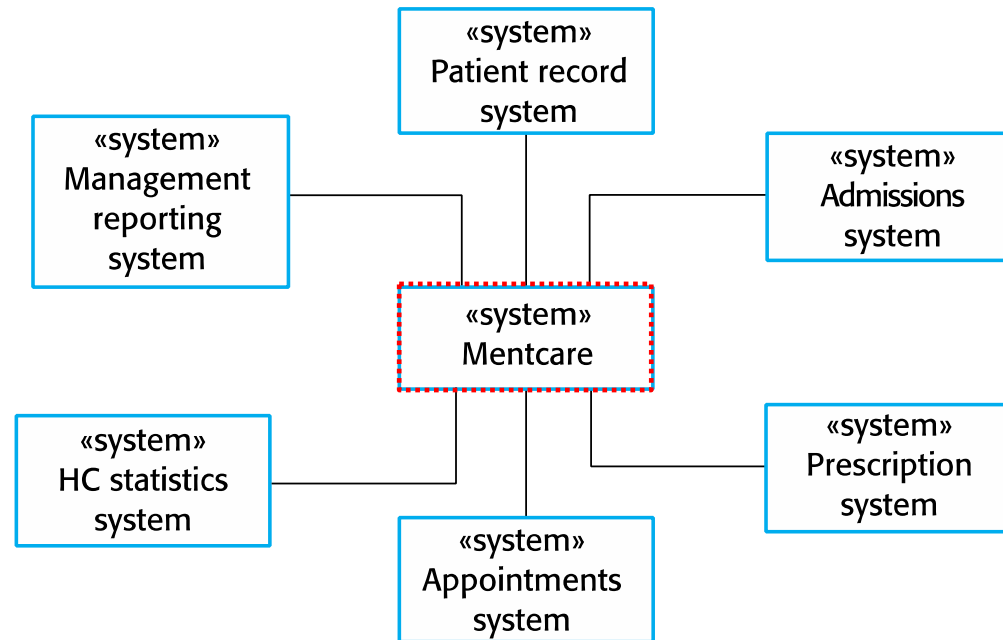
- **UML diagrams** used for system modeling:
 - **Use case diagram** : showing the interactions between a system and its environment
 - **Sequence diagram** : showing interactions between external actors and the system, or between system components
 - **Class diagram** : showing the object classes and the associations between these classes
 - **State (Statechart) diagram** : showing how the system reacts to internal and external events
 - **Activity diagram** : showing the activities involved in a process or in data processing

External Models

Context Models

- **Context models** illustrate the **operational context** (boundary) of a system.
 - **External perspective**
 - Show what lies outside the system boundaries
 - **Architecture models** show the system and its relationship with other systems.

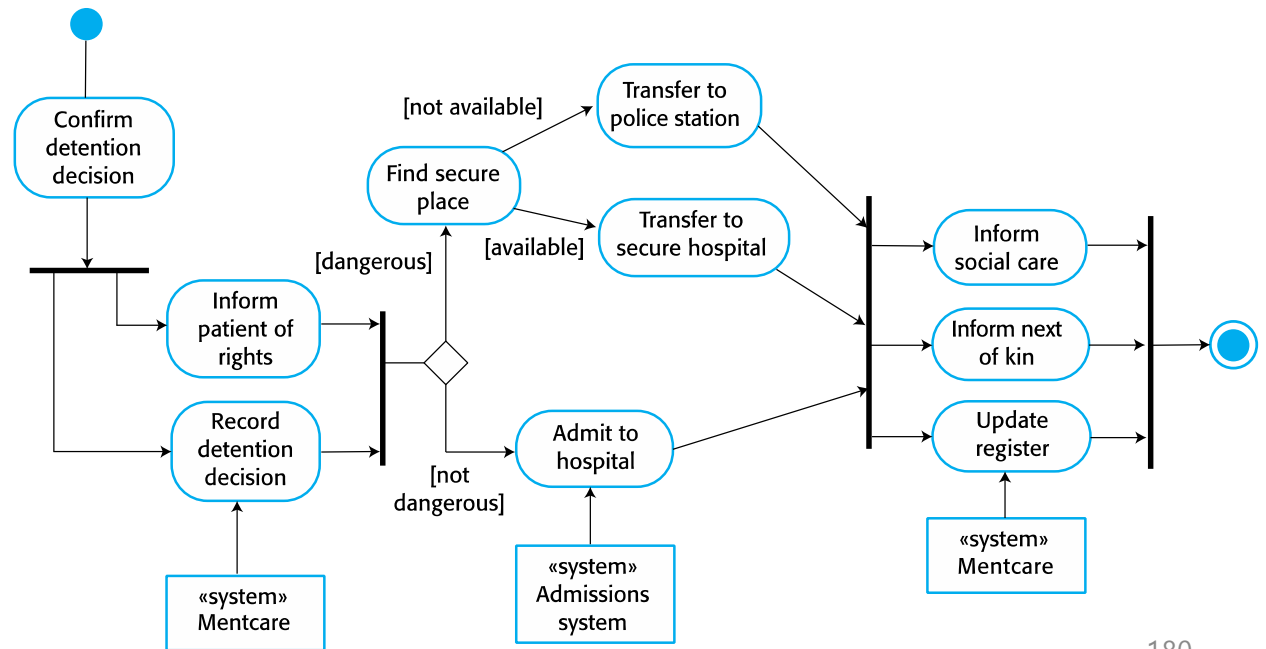
- Example: Mentcare System



Process Models

- **Process models** reveal **how the system is used in business processes**.
 - Show how the other systems will be used in business environment.
 - **UML activity diagrams** may be used to define business process models.
 - System level > Component interaction level

- Example : Involuntary Detention (강제구금)



Interaction Models

Interaction Models

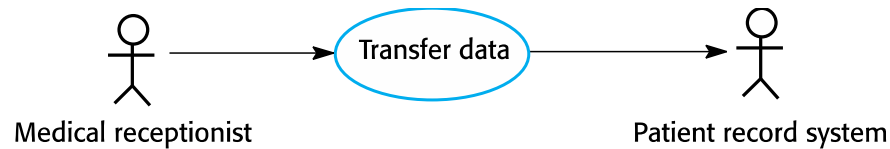
- **Interaction models**
 - Modeling **user interaction**
 - Helps to identify user requirements.
 - Modeling **system-to-system interaction**
 - Highlights the communication problems that may arise.
 - Modeling **component interaction**
 - Helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

- **UML Use-Case diagram** and **UML Sequence diagram** are often used.

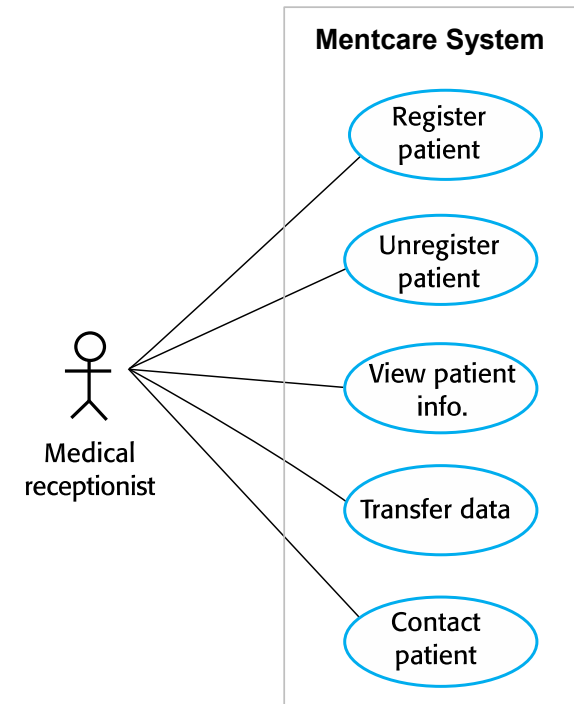
Use Case Modeling

- **Use case** represents a discrete task that involves external interaction with a system.
 - Use case is a **text scenario**.
 - Represents a discrete task that involves external interaction with a system.
 - Actors in a use case may be people or other systems.
 - **Use case diagrams** provide an overview of all use cases.

• Example : “Transfer Data” use-case in Mentcare System

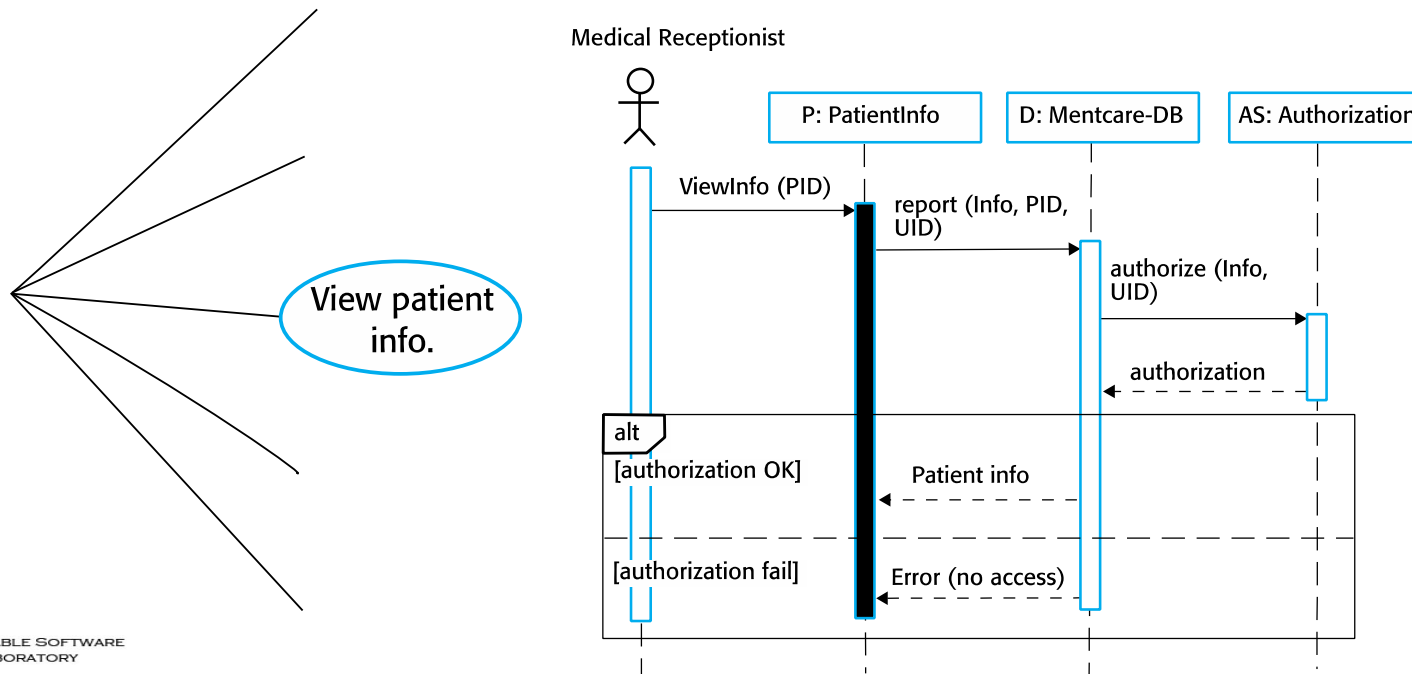


MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.



Sequence Diagrams

- **Sequence diagrams** show the **sequence of interactions** that take place during a particular use case.
 - The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
 - Interactions between objects are indicated by annotated arrows.
- Example : “View Patient Information” use case in Mentcare System



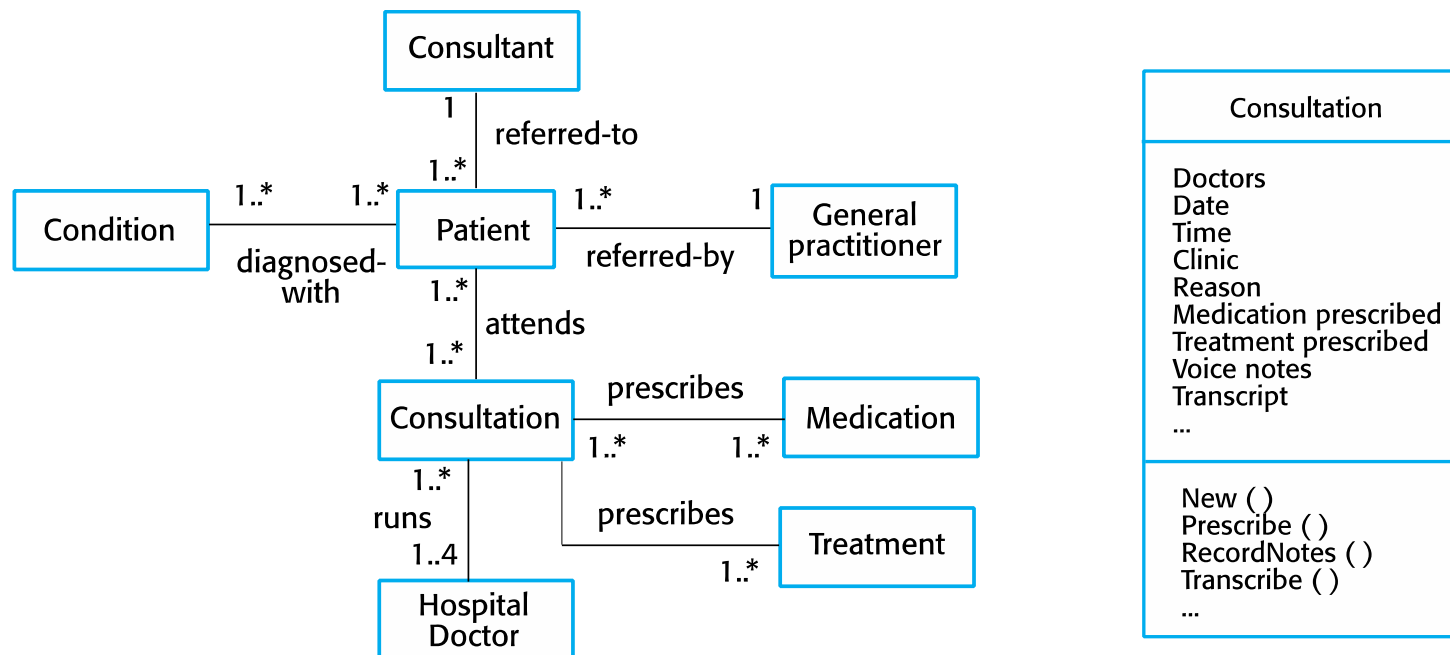
Structural Models

Structural Models

- **Structural models**
 - Represent the organization of a system in terms of the components that make up that system and their relationships.
 - **Static models** : show the structure of the system design
 - **Class diagram**
 - **Dynamic models** : show the organization (structure) of the system when it is executing (*i.e.*, dynamics)
 - **Object diagram, Component diagram, Composite structure diagram**
- Structural models are developed/created when you are designing the **system architecture**.

Class Diagrams

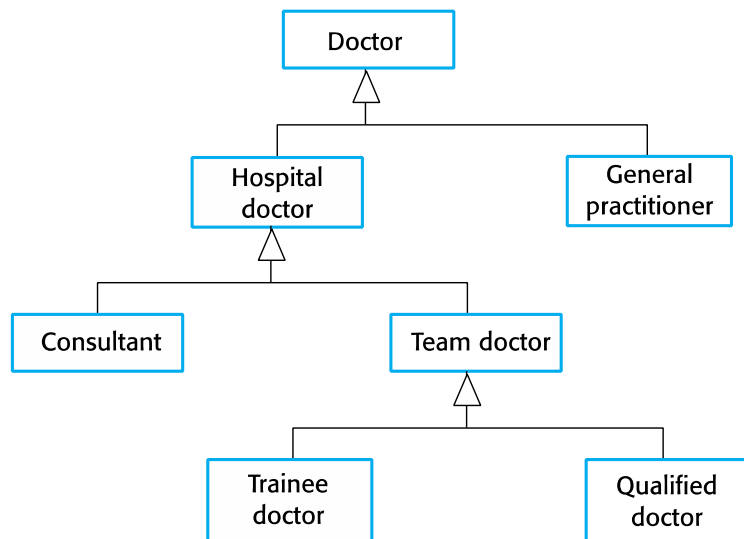
- **Class diagrams** show the classes in a system and the associations between these classes.
 - **(Object) Class**: a general definition of one kind of system object
 - **Association**: a link between classes indicating relationship between them
 - Used when developing an object-oriented system model.



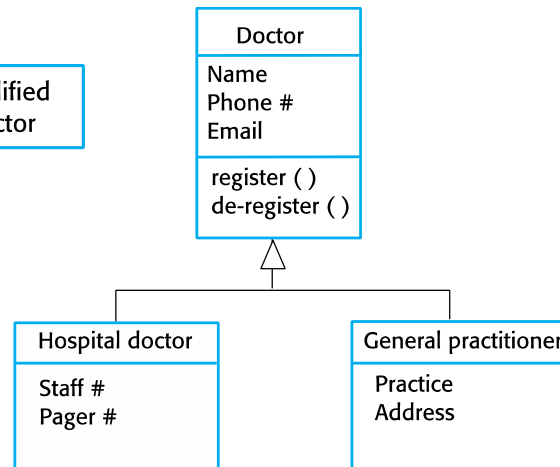
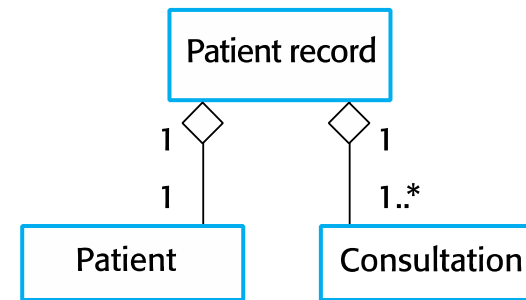
Classes and Associations in the MHC-PMS

Generalization and Aggregation in Class Diagram

- Generalization (Inheritance)



- Shared Aggregation / Composition



Behavioral Models

Behavioral Models

- **Behavioral models**

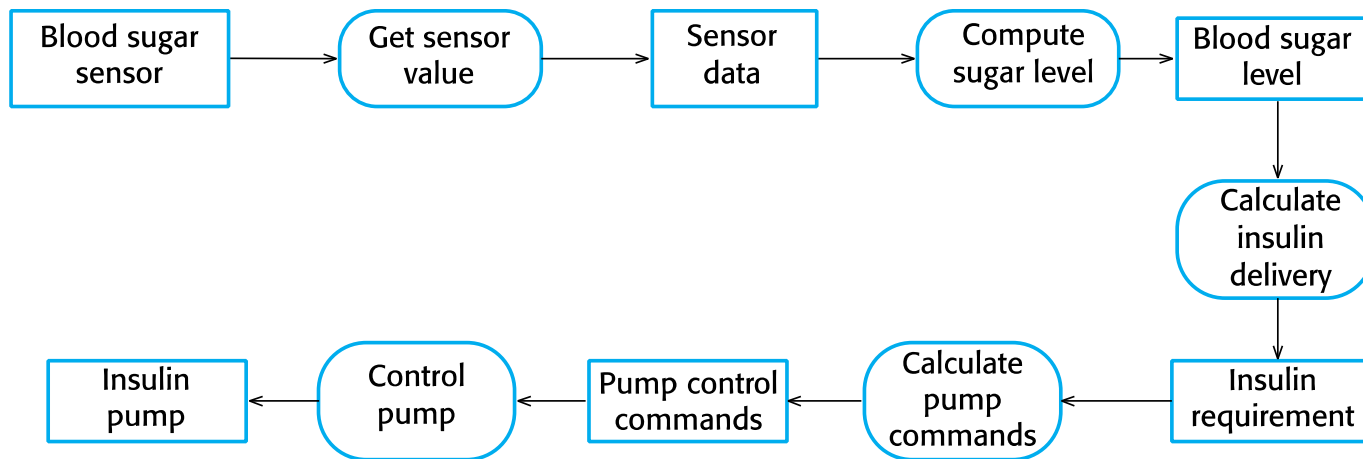
- Model dynamic behavior of a system as it is executing.
- Show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
 - **Data**: Some data arrives and has to be processed by the system.
 - **Events** : Some event happens and triggers system processing. Events may have associated data.

- Behavioral models

- **Data-driven model**
- **Event-driven model (State machine model)**

Data-Driven Models

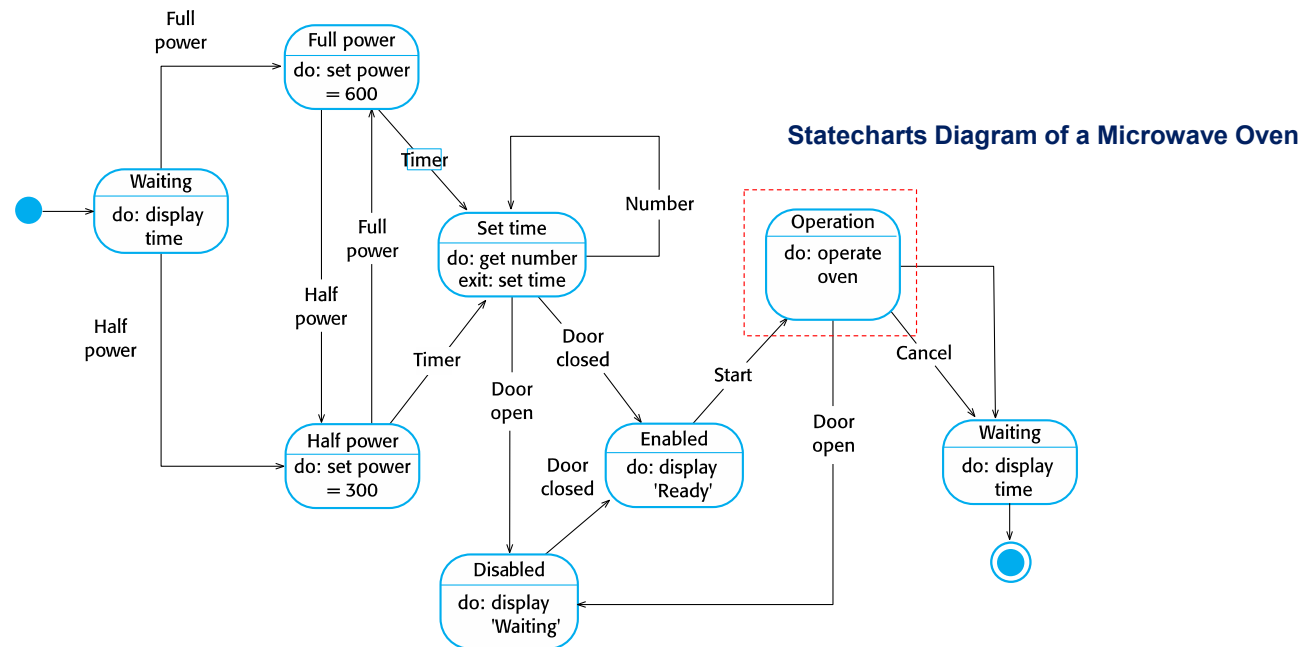
- Many business systems are **data-processing systems** that are primarily **driven by data**.
 - Controlled by the data input to the system, with relatively **little external event** processing.
 - Show the sequence of actions involved in processing input data and generating an associated output.
- **Data flow diagram (DFD)** and **UML Activity diagram** are also used.



An Activity Model of the Insulin Pump's Operation

Event-Driven Models

- Real-time systems are often **event-driven** with minimal data processing.
- **Event-driven modeling** shows how a system responds to external and internal events.
 - Based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.
 - Modeled well with **FSM (Finite State Machine)**.

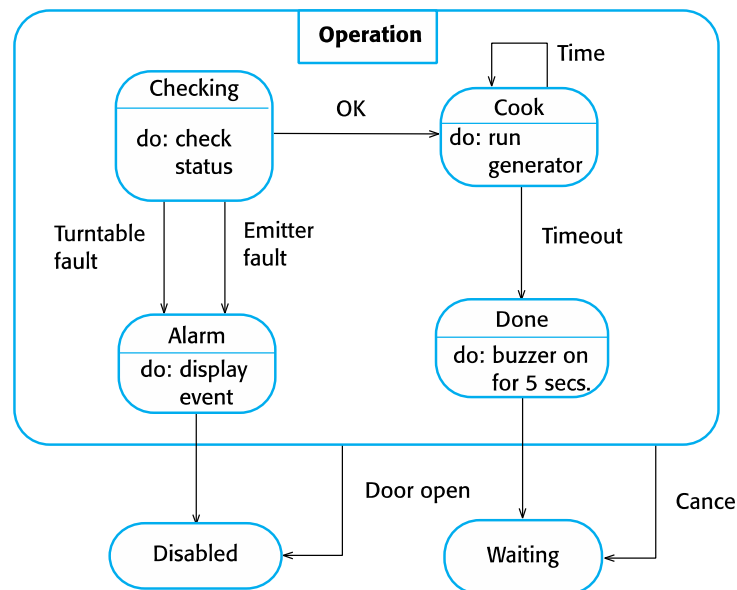


State Machine Models

- **State Machine models**

- Model the behaviour of the system in response to external and internal events.
- Show the system's responses to stimuli.
 - **System states** : nodes
 - **Events** : arcs between these nodes.
 - **Transitions** : When an event occurs, the system moves from one state to another.

- **UML Statecharts diagram**



Microwave Oven - Operations State

Model-Driven Engineering

Model-Driven Engineering

- **Model-driven engineering (MDE)**

- An approach to software development where models rather than programs are the principal outputs of the development process.
- The **programs** executing on a hardware/software platform are generated automatically from the models.
 - Software engineers no longer should be concerned with programming language details or the specifics of execution platforms.

- **MDE** is still at an early stage of development.

- Advantages
 - Allows systems to be considered at higher levels of abstraction
 - Generating code automatically means that it is cheaper to adapt systems to new platforms.
- Disadvantages
 - Models for abstraction are not necessarily right for implementation.
 - Savings from generating code may be outweighed by the costs of developing translators for new platforms.

Model-Driven Architecture

- **Model-driven architecture (MDA)** is a model-focused approach to software design and implementation.
 - The precursor of more general MDEs
 - Models at different levels of abstraction are created.
 - From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.
 - **CIM (Computation Independent Model)**
 - **PIM (Platform Independent Model)**
 - **PSM (Platform Specific Models)**
 - Often use a subset of UML models to describe a system

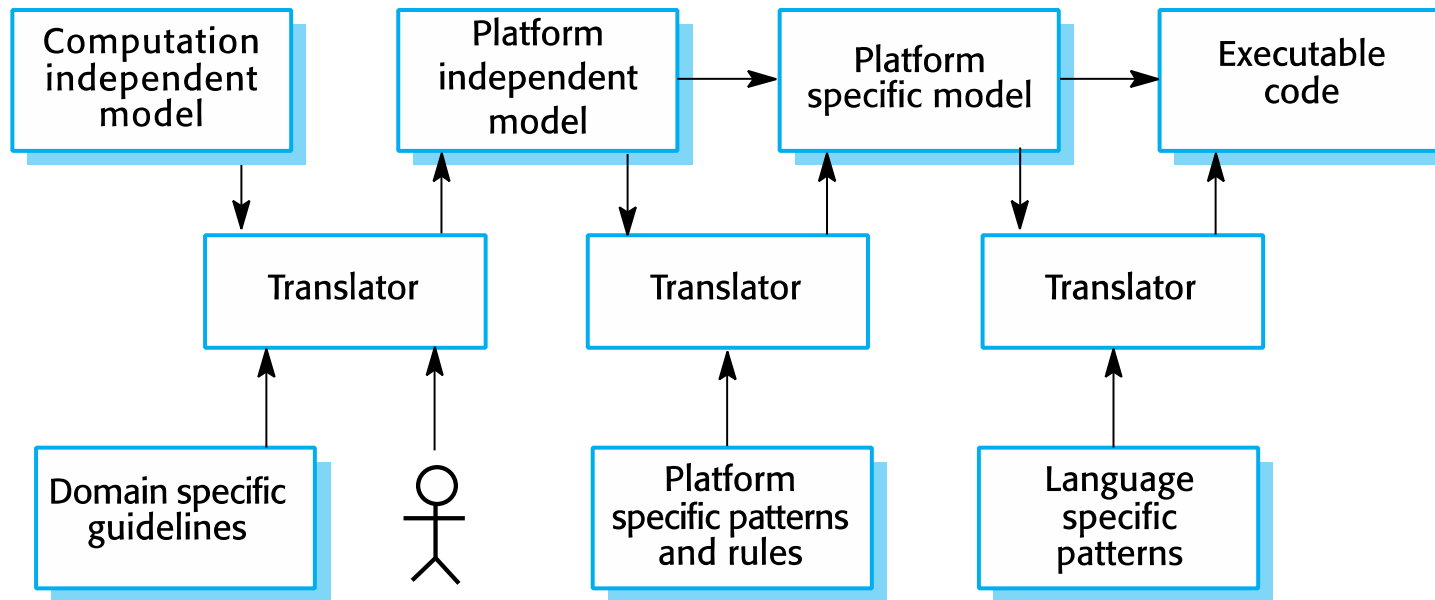
Types of Models in MDA

- **Computation Independent Model (CIM)**
 - Models the important domain abstractions used in a system
 - CIMs are sometimes called domain models.

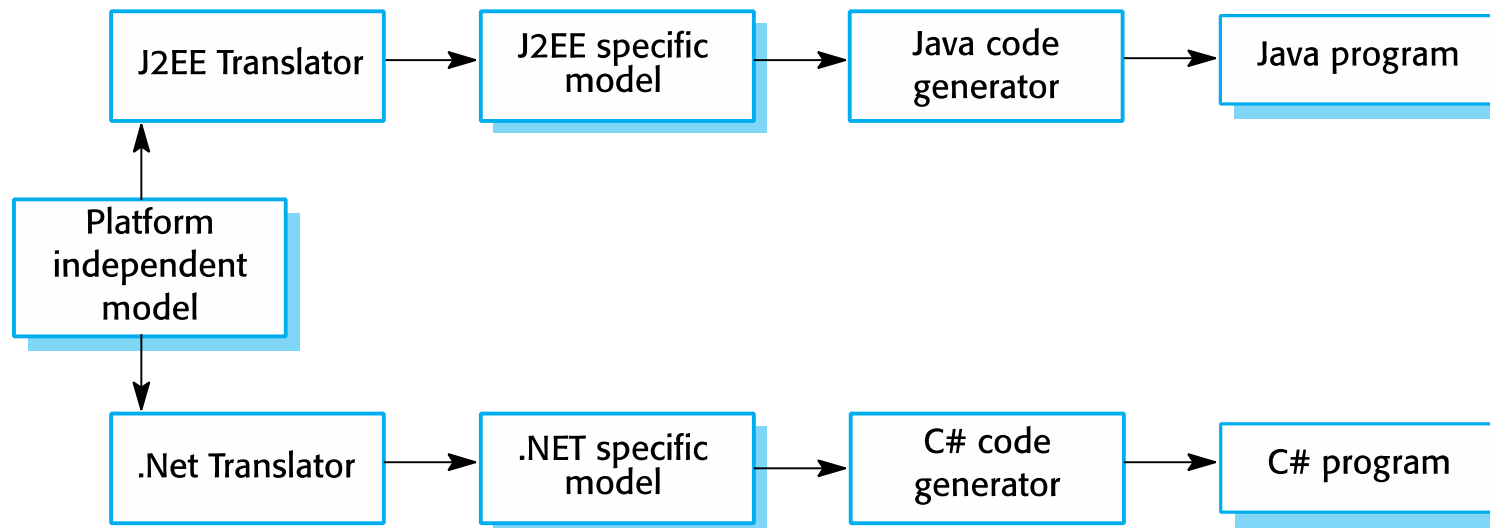
- **Platform Independent Model (PIM)**
 - Models the operation of the system without reference to its implementation.
 - PIMs are usually described using UML models that show the static system structure and how they respond to external and internal events.

- **Platform Specific Models (PSM)**
 - Transformations of the platform-independent model into a separate PSM for each application platform
 - In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

MDA Transformations



Multiple Platform-Specific Models



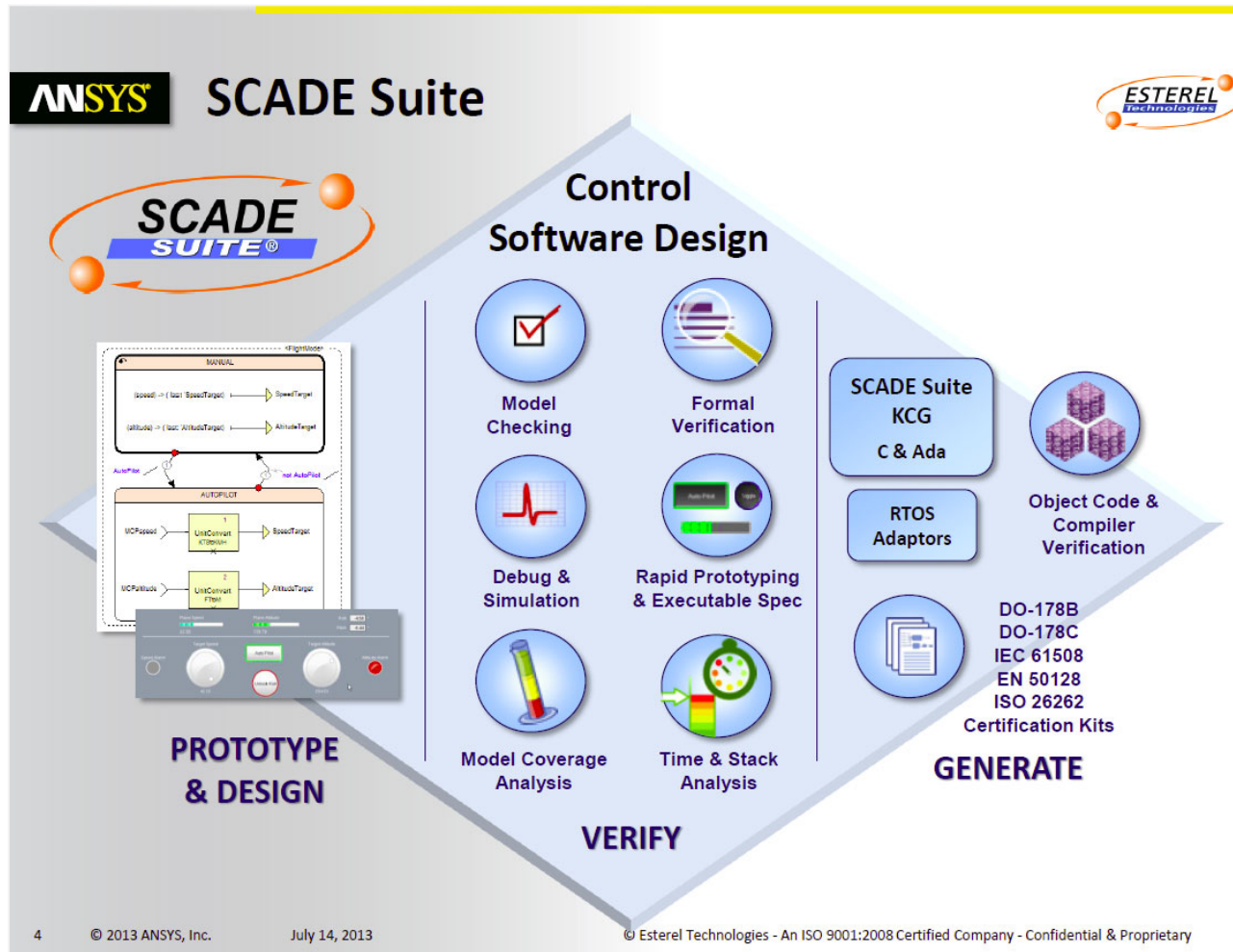
Adoption of MDA

- Limitations on adopting MDE/MDA
 - **Specialized tool support** is required to convert models from one level to another
 - There is limited tool availability and organizations may require tool adaptation and customization to their environment

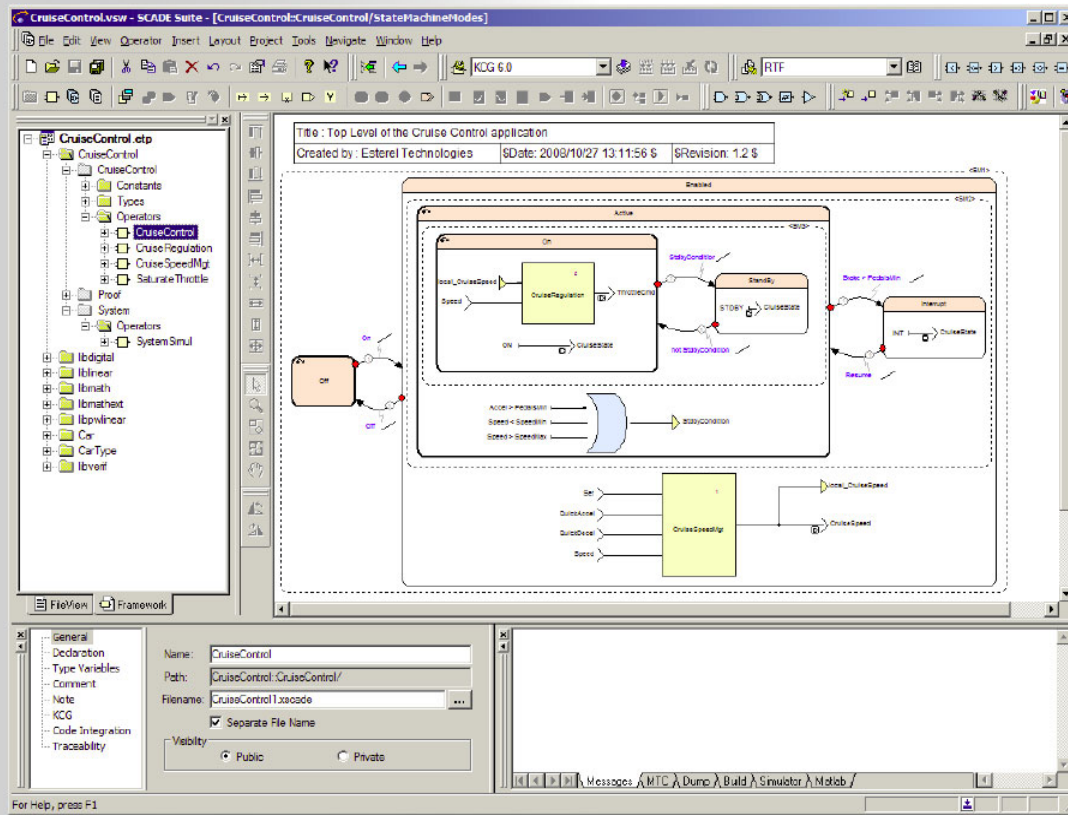
- Models are a good way of **facilitating discussions** about a software design.
 - However, the abstractions that are useful for discussions may not be the right abstractions for implementation.
 - For most complex systems, **implementation is not the major problem** – requirements engineering, security and dependability, integration with legacy systems and testing are all more significant.

- The arguments for platform-independence are only valid for large, long-lifetime systems.
 - For most software products and information systems, the savings from the use of MDA are likely to be **outweighed by the costs of its introduction and tooling**.

MDE Example : SCADE



ANSYS SCADE Suite Editor





SCADE Suite Libraries



libpwnlinear

pwnlinear package

lut package

liblinear

linear package

libdigital

digital package

truthtables package

filters package

Discrete filters (+ normalized versions)

Transfer functions

libmath

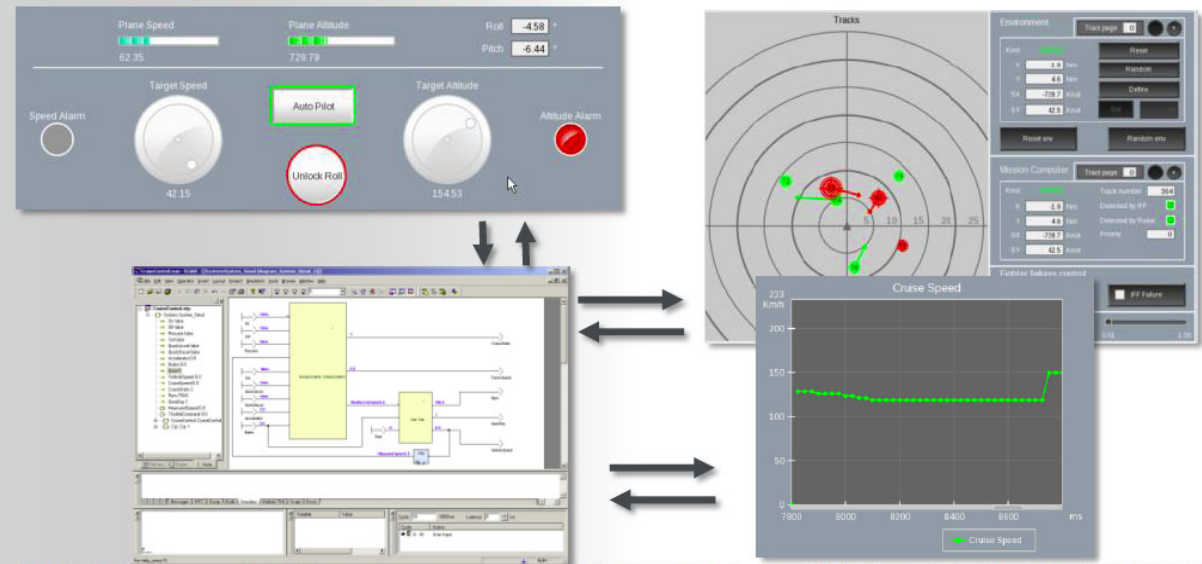
math package

vect package

ANSYS[®] SCADE Rapid Prototyper



- **Graphical panels** for quick & comfortable model debug & simulation
 - Features a library of interactive & display **predefined widgets**
 - Library of widgets can be user-customized/augmented



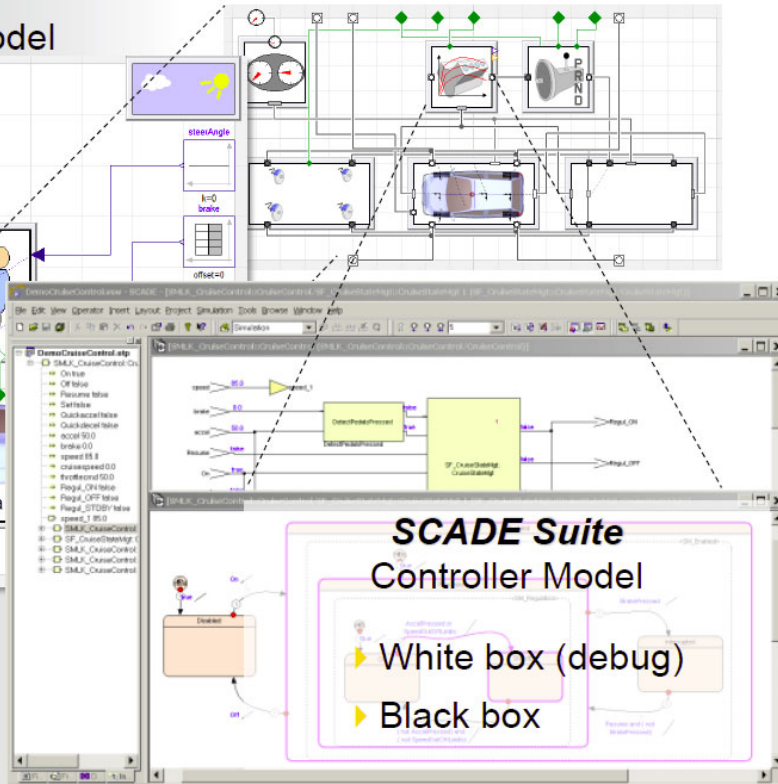
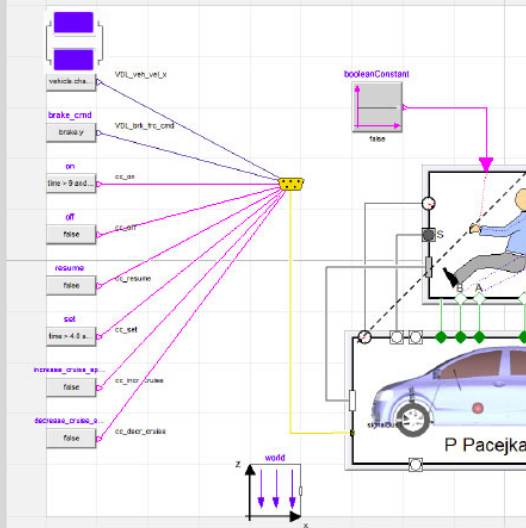
20 © 2013 ANSYS, Inc. July 14, 2013 © Esterel Technologies - An ISO 9001:2008 Certified Company - Confidential & Proprietary



Modelica/SCADE Suite Co-simulation



Modelica Car multi-physics Model

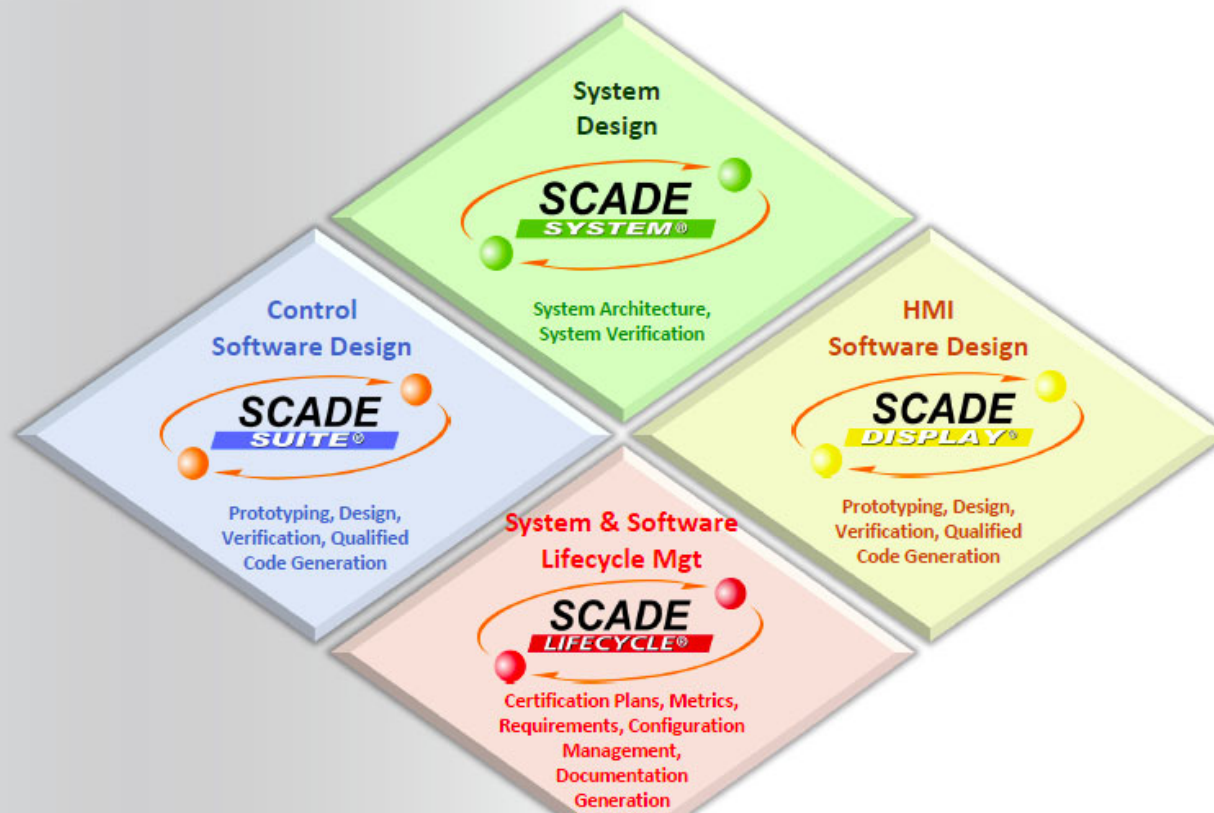


DYMOLA (Dassault Systèmes) and SCADE Co-simulation is a result of CESAR Project

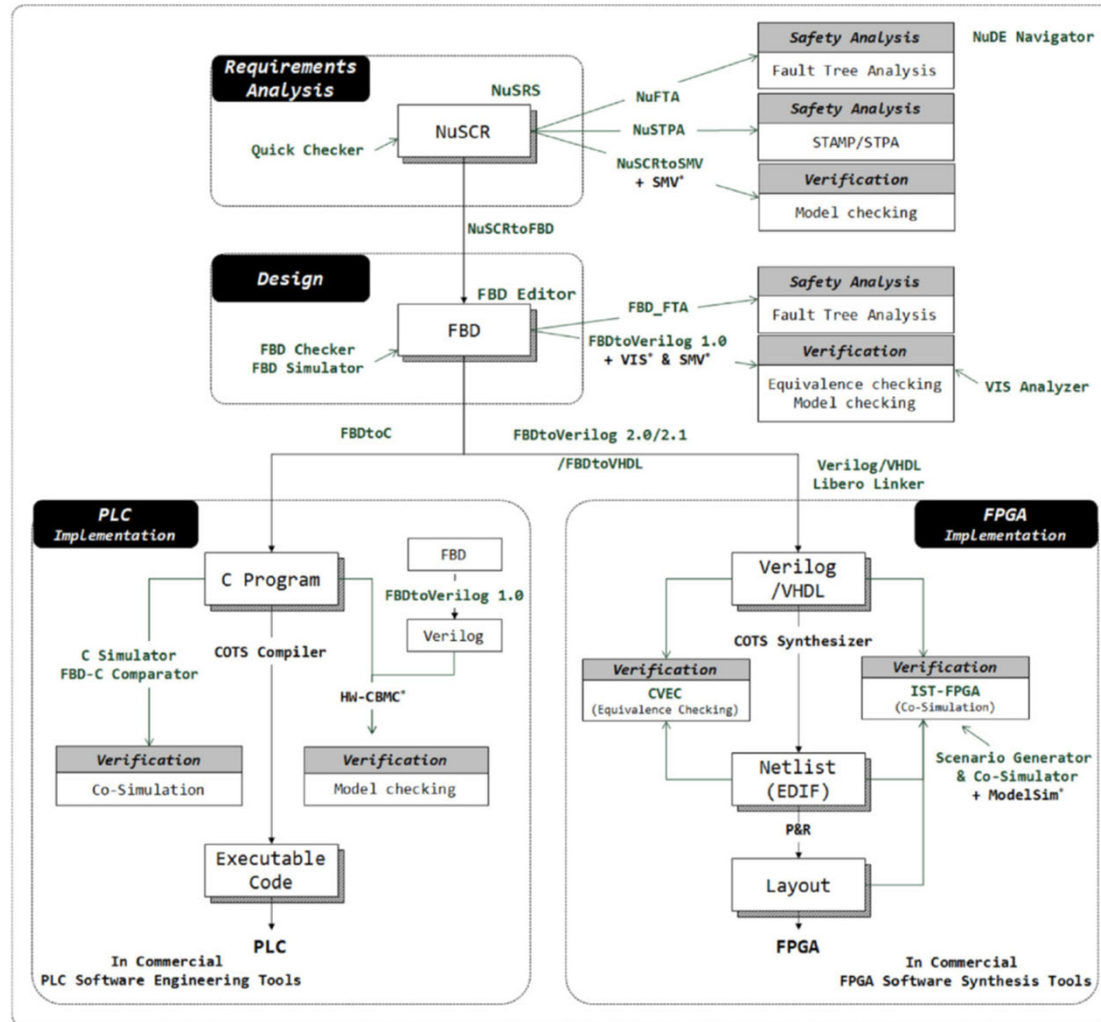


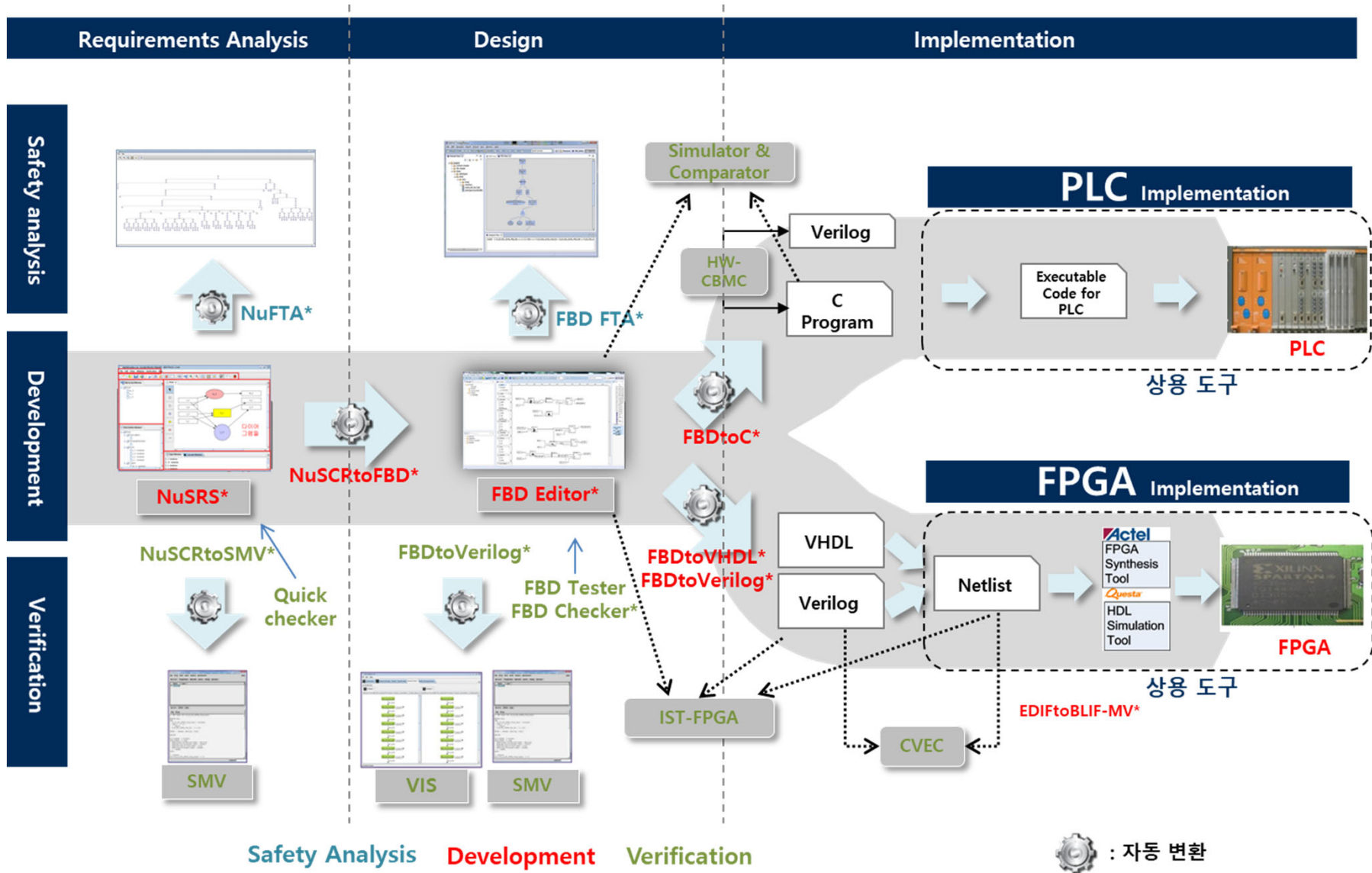


SCADE Product Family



MDE Example : NuDE





Safety Analysis Development Verification

: 자동 변환

NuSRS - Eclipse Platform

File Edit Navigate Search Project NuDE NuSRS Run Window Help

Quick Access Resource FBD_Editor NuDE 0.9 NuSRS

Common Navigator

- New_Nude
 - FBDtoC
 - FBDtoVerilog
 - NuSCRtoFBD
 - NuSRS
 - (NuSRS) RPS BP (20130716).xml

Hierarchy Window

- Root
 - g_BP
 - g_LO_SG1_LEVEL
 - g_VAR_OVER_PWR
 - g_HI_LOG_POWER
 - g_LO_PZR_PRESS
 - g_SG1_LO_FLOW
 - g_HI_LOCAL_POWER

Description Window

- g_VAR_OVER_PWR
 - Description
 - 가변 과충력 트립 (자동비출/상승)
 - TemplateNumber
 - Input
 - f_VAR_OVER_PWR_PV : 0.30
 - f_VAR_OVER_PWR_Manu_Test
 - f_VAR_OVER_PWR_MT_Query
 - f_VAR_OVER_PWR_Trip_Status
 - f_VAR_OVER_PWR_Ptrp_Status
 - f_VAR_OVER_PWR_Ptrp_Err
 - Output
 - f_VAR_OVER_PWR_Val_Out
 - f_VAR_OVER_PWR_Ptrp_SP
 - f_VAR_OVER_PWR_Ptrp_Out
 - th_VAR_OVER_PWR_Trip_Log
 - th_VAR_OVER_PWR_Ptrp_Log
 - f_VAR_OVER_PWR_PV_Err
 - f_VAR_OVER_PWR_Trip_Out

Diagram Window

Type Window

```

f_Mod_Err: boolean
f_VAR_OVER_PWR_Chan_Err: boolean
f_VAR_OVER_PWR_MT_Query: boolean
f_VAR_OVER_PWR_Manu_Test: 0.30000
f_VAR_OVER_PWR_Op_By_Inv: boolean
f_VAR_OVER_PWR_PV: 0.30000
f_VAR_OVER_PWR_PV_Err: boolean
f_VAR_OVER_PWR_Ptrp_Out: boolean
f_VAR_OVER_PWR_Ptrp_SP: 0.30000
f_VAR_OVER_PWR_Ptrp_Status: boolean
f_VAR_OVER_PWR_Ptrp_Out: boolean
f_VAR_OVER_PWR_Trip_Status: boolean
f_VAR_OVER_PWR_Trip_SP: 0.30000
f_VAR_OVER_PWR_Trip_Out: boolean

```

0 items selected

NuSRS – NuSCR Modeling Environment

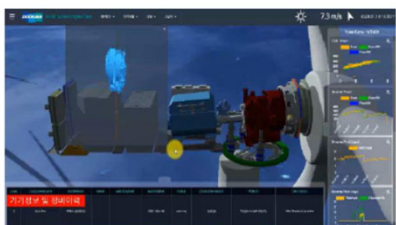
Homework / Activity #5

- “Digital Twin (디지털 트윈)”의 정의를 살펴 보고, 국내외 적용 사례를 찾아 보세요.

Samples from SE Undergraduate (KU 2021)

- 두산중공업

디지털 트윈이 적용된 풍력 발전기의 실시간 장비·환경 데이터는 가상의 쌍둥이 모델과 시스템 담당자에게 전송된다. 설비 담당자는 이를 기반으로 쌍둥이 모델에서 다양한 사건 시뮬레이션을 진행할 수 있으며 개선된 모델링이 나오면 이를 장비로 보내 시스템을 업그레이드할 수 있는 구조다. 이를 통해 먼 거리에 있는 풍력 발전 설비에 직접 방문하지 않고도 다양한 데이터 수집이 가능해지며 실제와 같은 환경 데이터를 기반으로 설비에 대한 더 높은 운영 가시성을 확보할 수 있게 된다.



- 한국남동발전

한국남동발전은 국내 발전사 최초로 디지털 트윈을 활용한 신재생에너지 설비 점검 시스템을 개발해 본격 운용에 들어갔다. 이를 통해 남동발전은 인공지능(AI) 기반 기술을 활용한 자율비행 드론으로 사진 및 영상 자료를 촬영하고 이를 실제 설비를 가상화한 3D 모델에 일치시킨 후 설비상태를 관리할 수 있는 디지털트윈 기술을 접목한 시스템으로 풍력 및 태양광설비에 대한 점검·이력을 관리할 수 있게 됐다. 지금까지는 망원경이나, 로프 액세스, 크레인 등 장비로 풍력설비 블레이드를 점검하거나 열화상 카메라로 태양광 패널을 점검해 이력 관리를 해왔다.

4.2.1 General Electric의 'Predix' [9]

GE는 2016년 세계 최초의 산업용 클라우드 기반 오픈 플랫폼인 '프레딕스(Predix)'를 공개했다. 프레딕스 공개 직후 지금까지 수만 명의 소프트웨어 개발자가 프레딕스 플랫폼을 통해 수백 개의 산업용 애플리케이션 생태계를 구축했으며, 특히 개발사인 GE 역시 프레딕스를 활용해 2017년 기준으로 80만 개에 달하는 디지털 트윈을 개발하며 디지털 트윈 확산에 앞장서고 있다.

4.2.3 GE 헬스케어의 "Command Center" [10]

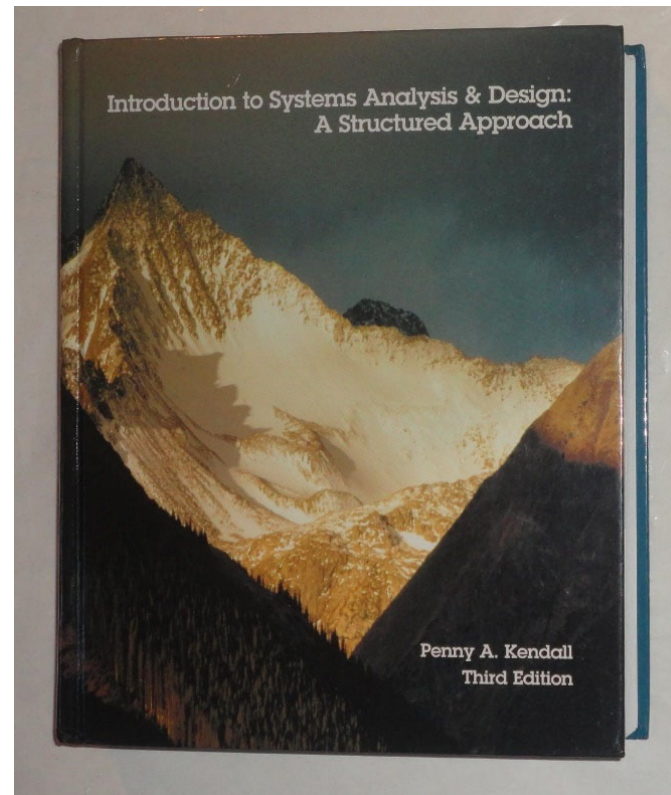
미국 존스홉킨스병원은 GE 헬스케어와 함께 '디지털 트윈', '커맨드센터(Command Center)'를 구축했다. 디지털 트윈은 병원이 가지고 있는 데이터로 가상의 병원을 만들고 운영을 예측하는데 활용된다. 해당 모델을 통해 실시간으로 병원 의료 서비스의 수요를 분석하고 예측할 수 있다. 예측 결과에 따라 적절한 프로그램을 수행할 수 있도록 있게 한다.

GE 헬스케어의 커맨드센터는 전 세계 300개 이상의 병원에 도입돼 있다. 최적의 환자 치료를 지원하기 위해 24시간 내내 AI 기술을 활용해 성과를 모니터링하고 개선 프로그램을 선보인다. 또한 신속대응팀이 상주해 일선에 있는 의료진을 지원한다. 해당 대응팀은 간호-임상-수술 신속대응팀 등 여러 가지 형태로 이루어져 있으며, 디지털 트윈 시뮬레이션을 토대로 상황을 인지하고 위험을 예측해 조치한다. 더조인트 커미션 저널(The Joint Commission Journal)의 관련 논문에 따르면, 이 시스템을 통해 존스홉킨스병원의 연간 환자 수용력은 2% 증가했고 응급실 대기 시간은 35% 감소했다.

An Introduction to Structured Analysis and Structured Design (SASD)

References

- Modern Structured Analysis, Edward Yourdon, 1989.
- Introduction to System Analysis and Design: a Structured Approach, Penny A. Kendall, 1996.



Structured Analysis

- **Structured analysis** [Kendall 1996]
 - A set of **techniques** and **graphical** tools
 - Allowing the analysts to develop a new kind of system specification that are easily understandable to the users.
 - Data/Functional modeling: **DFD**, **ERD**
 - State-oriented modeling: **STD (FSM)**

- Analysts attempt to divide large, complex problems into smaller, more easily handled ones.
 - **Top-Down Divide and Conquer** approach

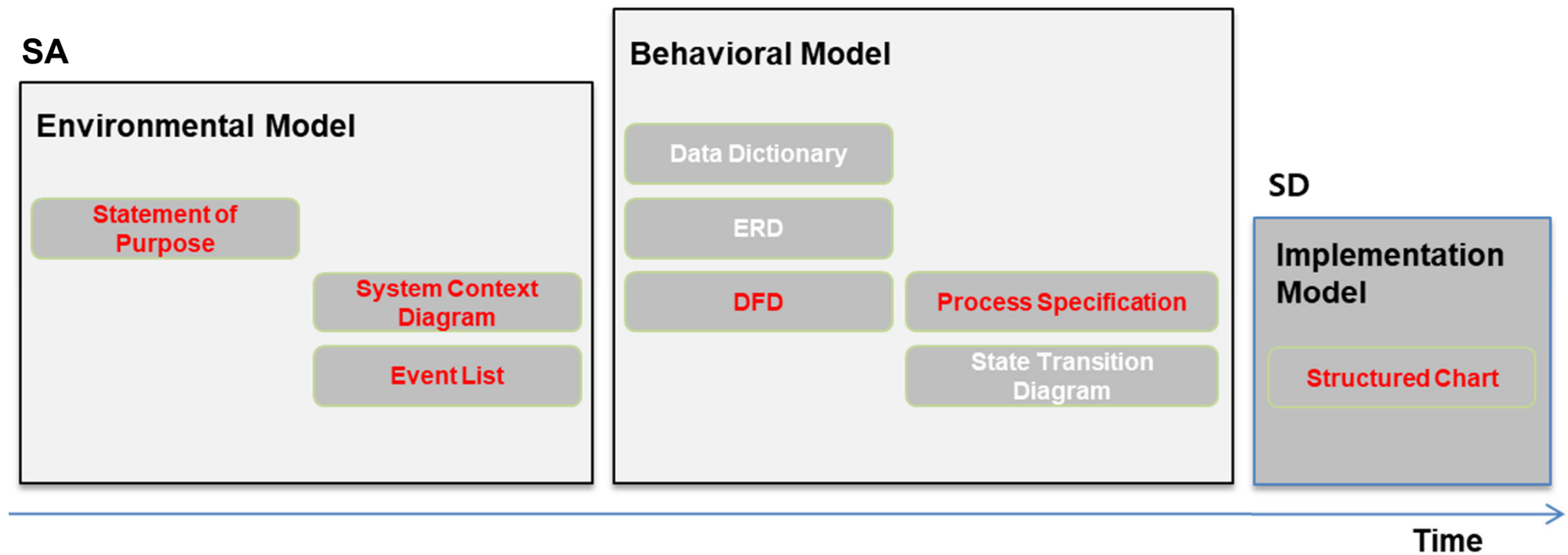
History of SASD (Structured Analysis and Structured Design)

- Developed in the late 1970s by DeMarco, Yourdon and Constantine after the emergence of structured programming.
- IBM incorporated SASD into their development cycle in the late 1970s and early 1980s.
- Yourdon published the book “Modern Structured Analysis” in 1989.
- The availability of CASE tools in 1990s enabled analysts to develop and modify the graphical SASD models.



For more cartoons, visit: www.briscoe.org To subscribe: Send an e-mail to join-smallworld@briscoe.org
© 2001 Briscoe <http://www.briscoe.org/>

An Overview of SASD



Structured Analysis (SA) - An Example of RVC SW Controller



Statement of Purpose

- A clear and concise textual description of the purpose for the system to develop
 - Should be deliberately vague.
 - Intended for top level management, user management and others who are not directly involved in the system.

Statement of Purpose - RVC Example

- **User Requirements (Business Requirements)**
 - **PFR (Preliminary Functional Requirements) 로 작성**



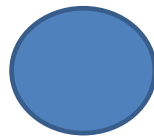
Robot Vacuum Cleaner (RVC) SW Controller

- An RVC automatically cleans and mops household surface.
- It goes straight forward while cleaning.
- If its sensors found an obstacle, it stops cleaning, turns aside left or right, and goes forward with cleaning.
- If there are obstacles in both front, left and right, it move backward and turn aside left or right, and goes forward.
- If it detects dust, power up the cleaning for a while.
- We do not consider the detail design and implementation on HW controls.
- We only focus on the automatic cleaning function.

System Context Diagram

- A special case of **DFD** (Data Flow Diagram)
 - **DFD Level 0**
 - Highlights the boundary between the system and outside world.
 - Highlights the people, organizations and outside systems that interact with the system under development.

- Notation :



Process : represents the proposed system

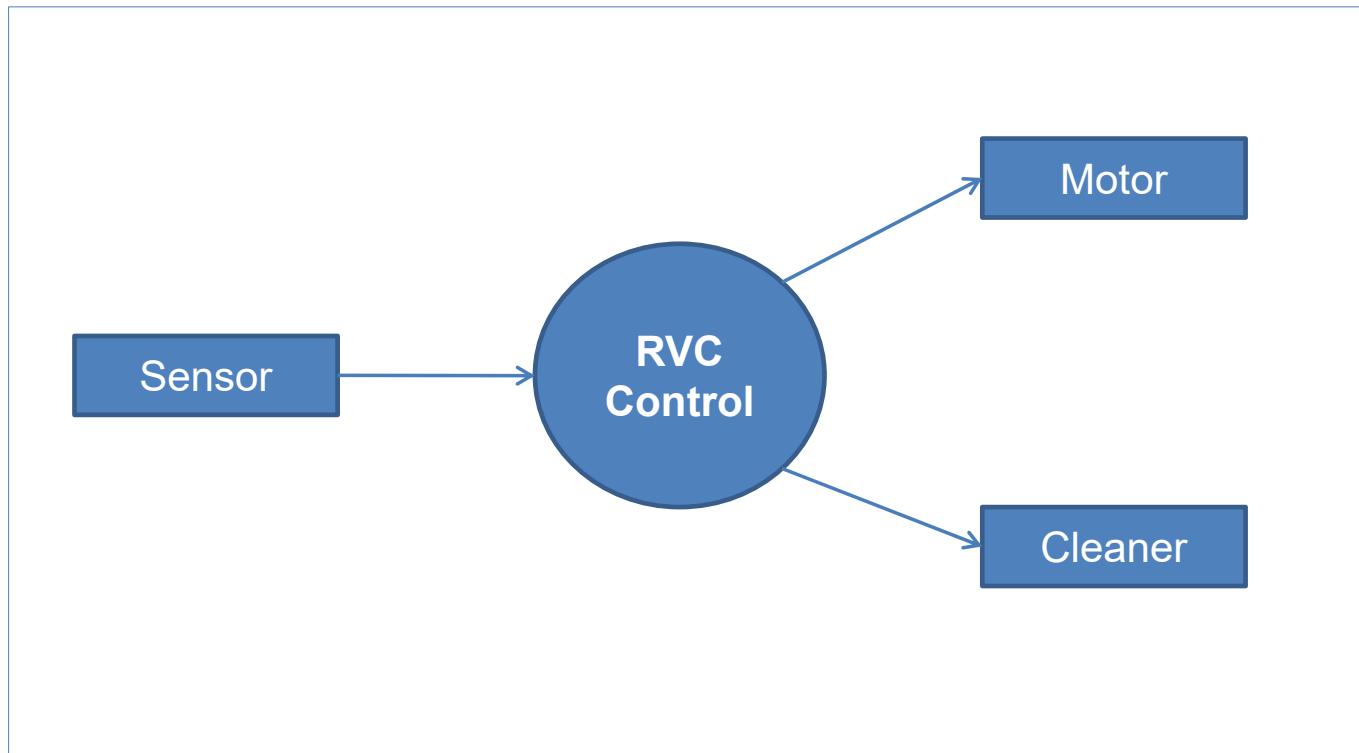


Terminator : represents the external entities



Flow : represents the in/out data flows

System Context Diagram – RVC Example



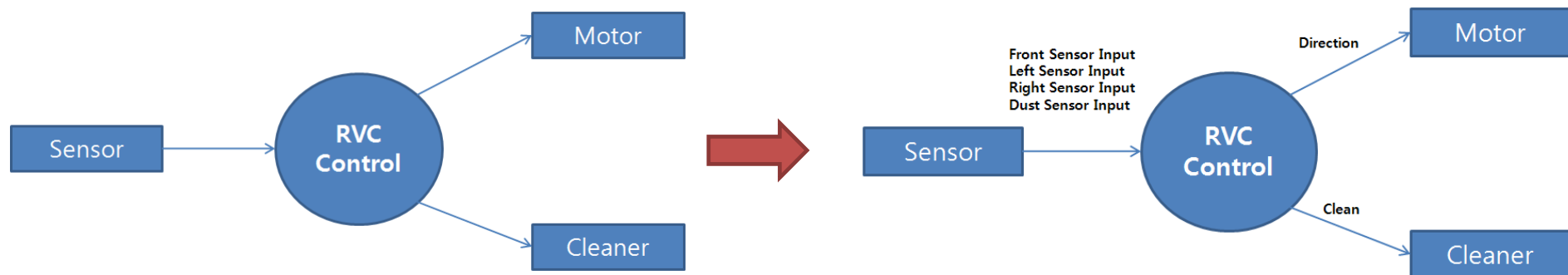
Event List

- A list of the event/stimuli outside of the system to which it must respond.
 - Used to describe the context diagram in detail.

- Types of inputs events
 - **Flow-oriented event** : triggered by incoming data
 - **Temporal event** : triggered by internal clock
 - **Control event** : triggered by an external unpredictable event

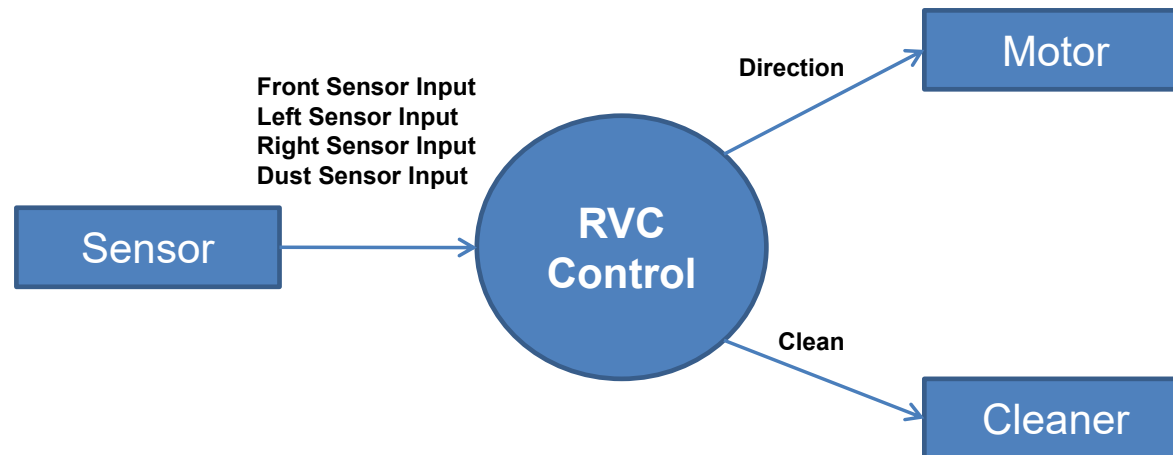
Event List – RVC Example

Input/ Output Event	Description
Front Sensor Input	Detects obstacles in front of the RVC
Left Sensor Input	Detects obstacles in the left side of the RVC periodically
Right Sensor Input	Detects obstacles in the right side of the RVC periodically
Dust Sensor Input	Detects dust on the floor periodically
Direction	Direction commands to the motor (go forward / go backward / turn left with an angle / turn right with an angle)
Clean	Turn off / Turn on / Power-Up



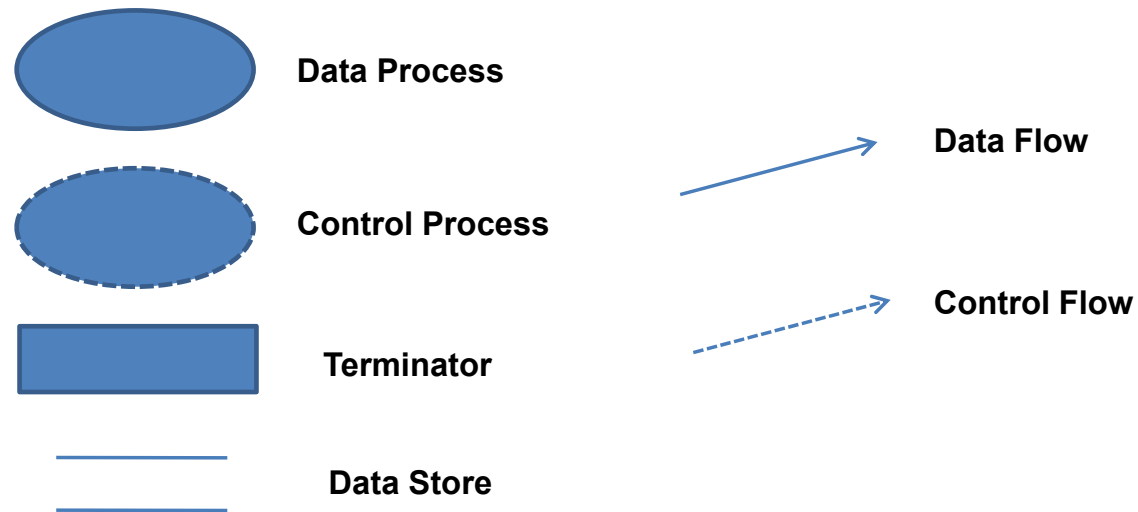
Context Diagram for RVC

System Context Diagram – RVC Example

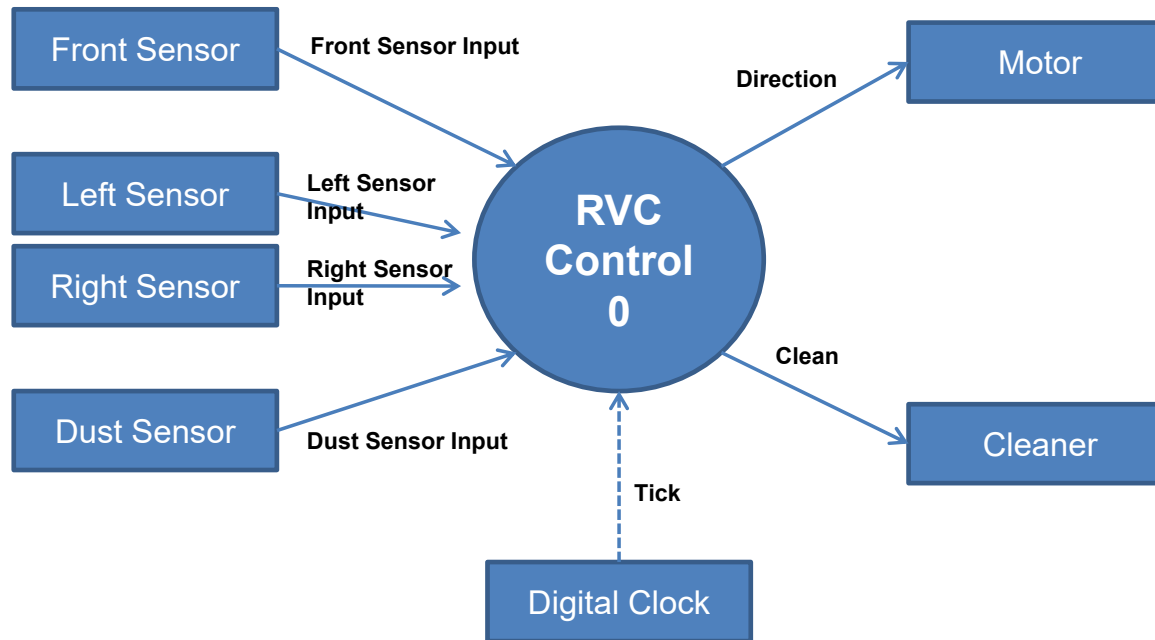


Data Flow Diagram (DFD)

- Provides a means for **functional decomposition**.
 - Composed of hierarchies(levels) of DFDs.
- **Notation** (A kind of CDFD)



DFD Level 0 – RVC Example

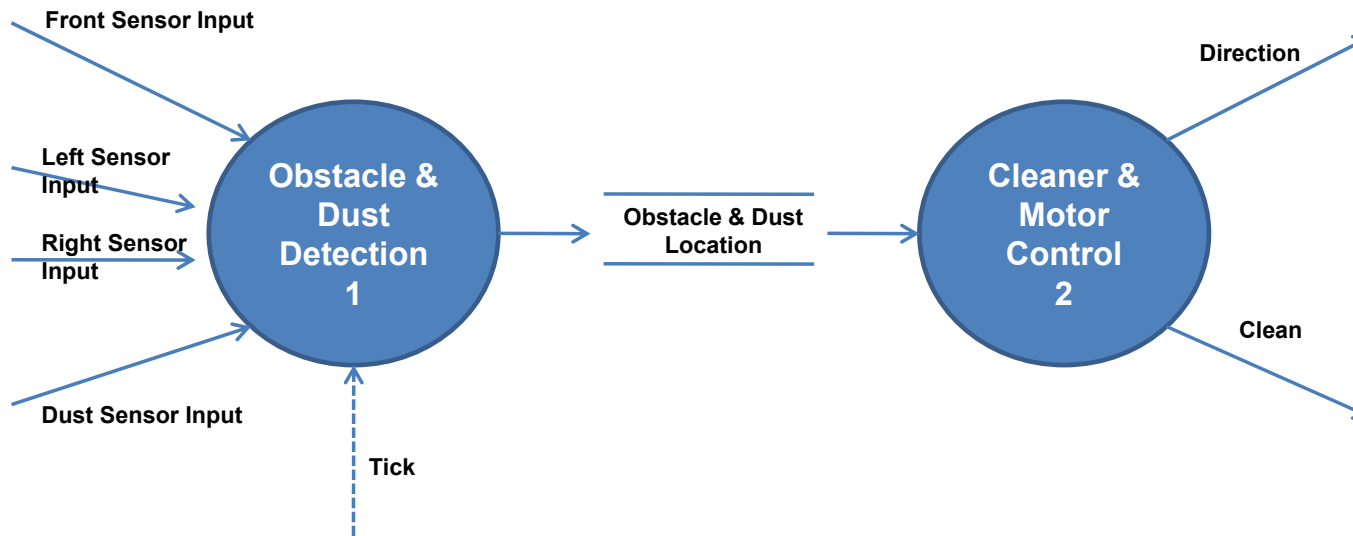


DFD Level 0 – RVC Example

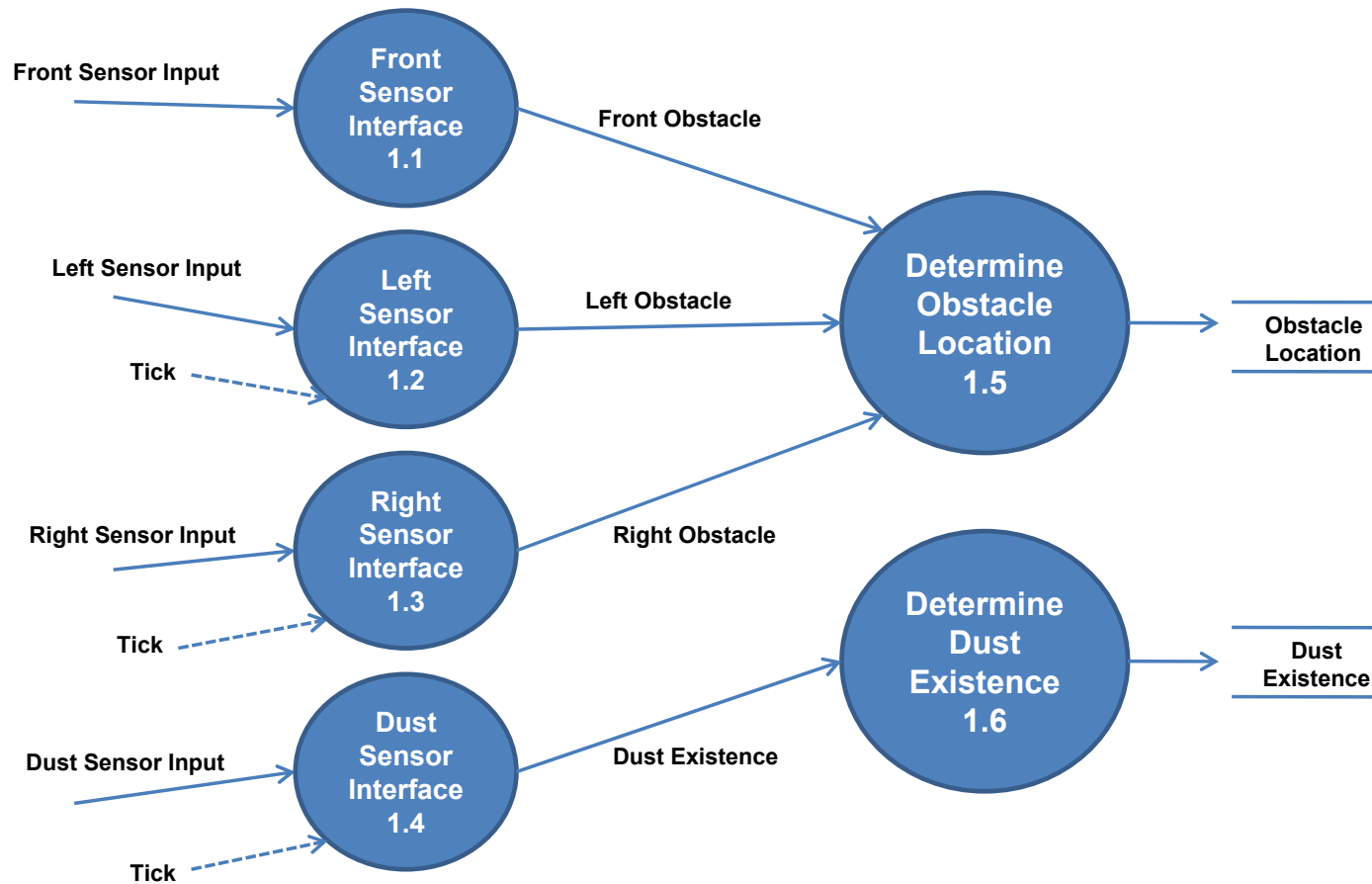
(A kind of) Data Dictionary

Input/ Output Event	Description	Format / Type
Front Sensor Input	Detects obstacles in front of the RVC	True / False , Interrupt
Left Sensor Input	Detects obstacles in the left side of the RVC periodically	True / False , Periodic
Right Sensor Input	Detects obstacles in the right side of the RVC periodically	True / False , Periodic
Dust Sensor Input	Detects dust on the floor periodically	True / False , Periodic
Direction	Direction commands to the motor (go forward / turn left with an angle / turn right with an angle)	Forward / Backward / Left / Right
Clean	Turn off / Turn on / Power-Up	On / Off / Up

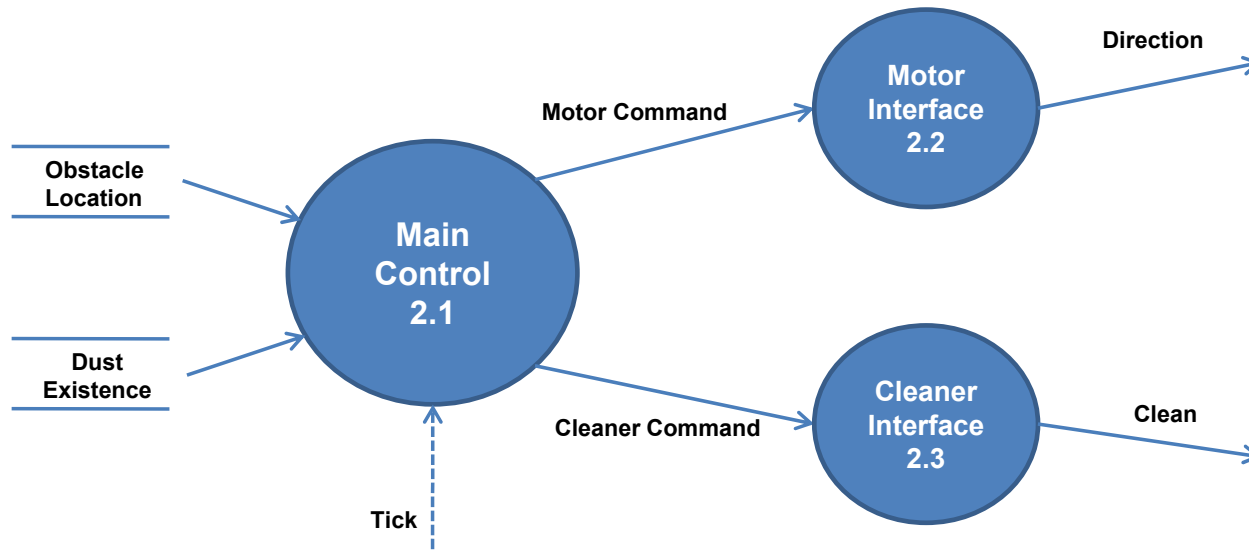
DFD Level 1 – RVC Example



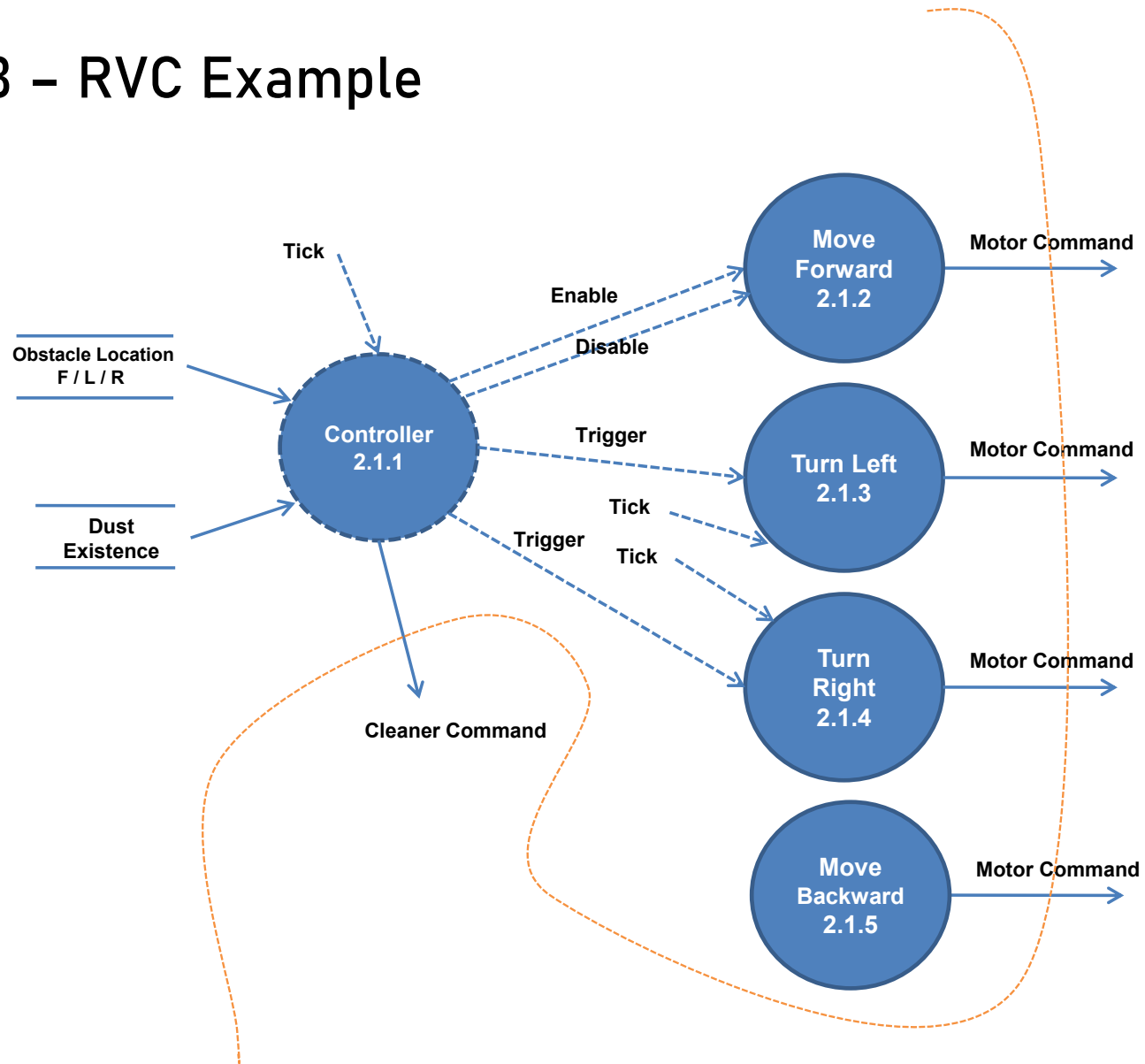
DFD Level 2 – RVC Example



DFD Level 2 – RVC Example

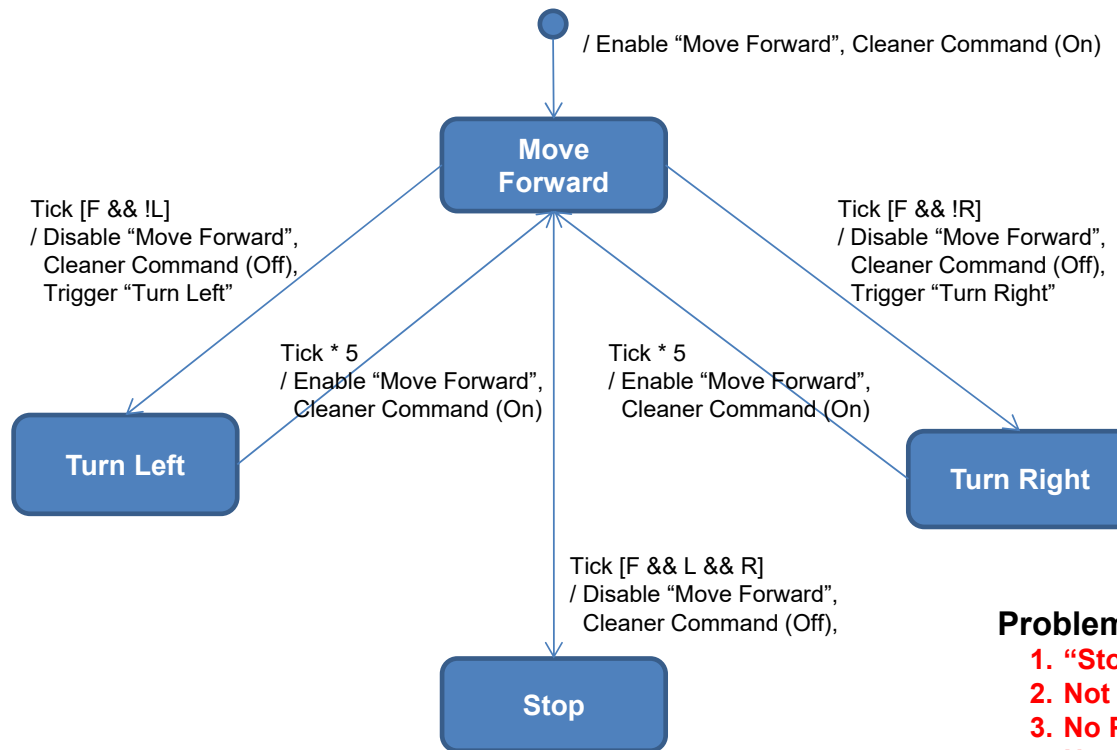


DFD Level 3 – RVC Example



DFD Level 4 – RVC Example

FSM for Controller 2.1.1

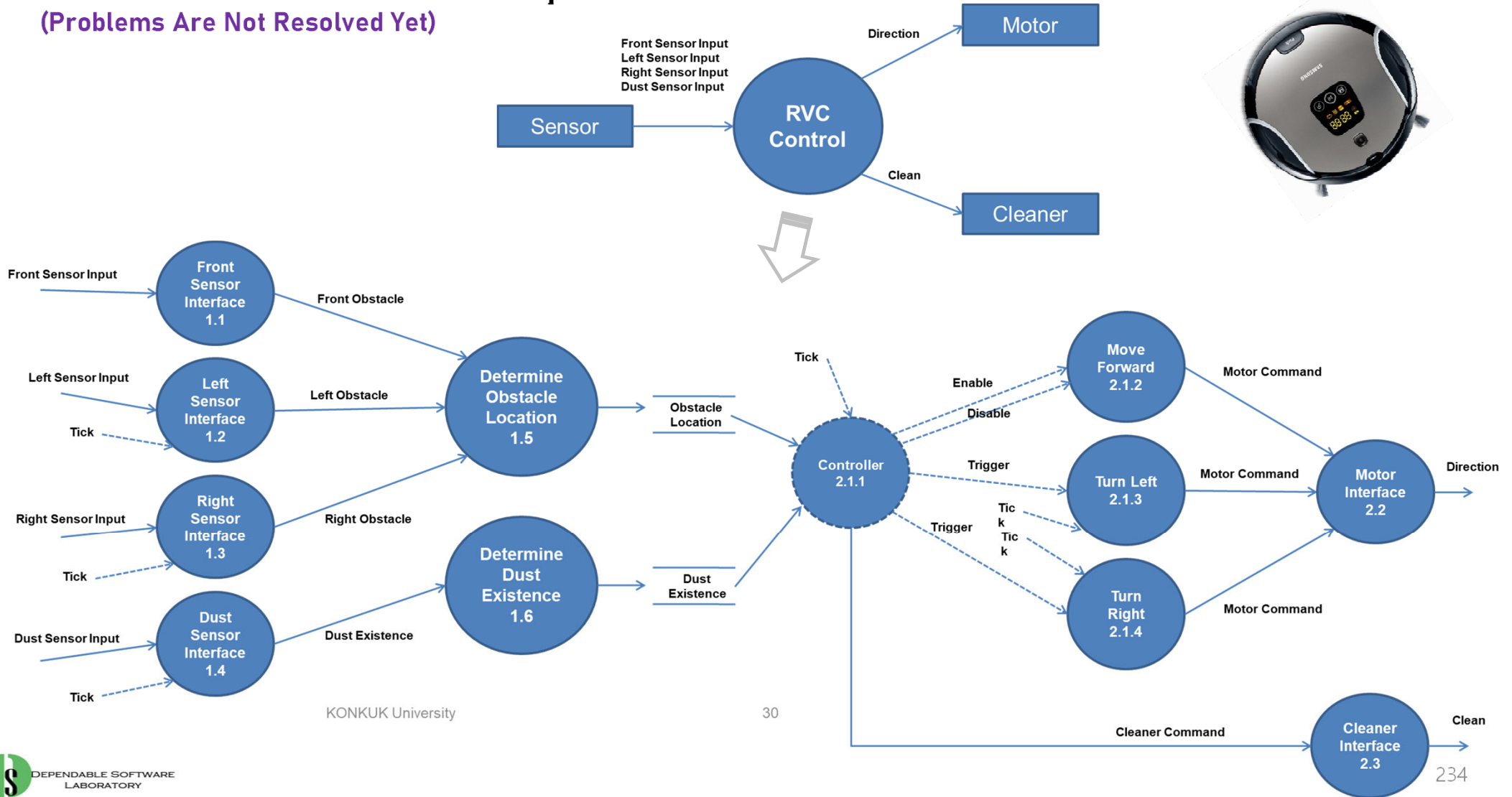


Problems in this model:

1. "Stop" state (deadlock)
2. Not consider "Dust"
3. No Priority for left/right turn
4. Not consider "Backward"

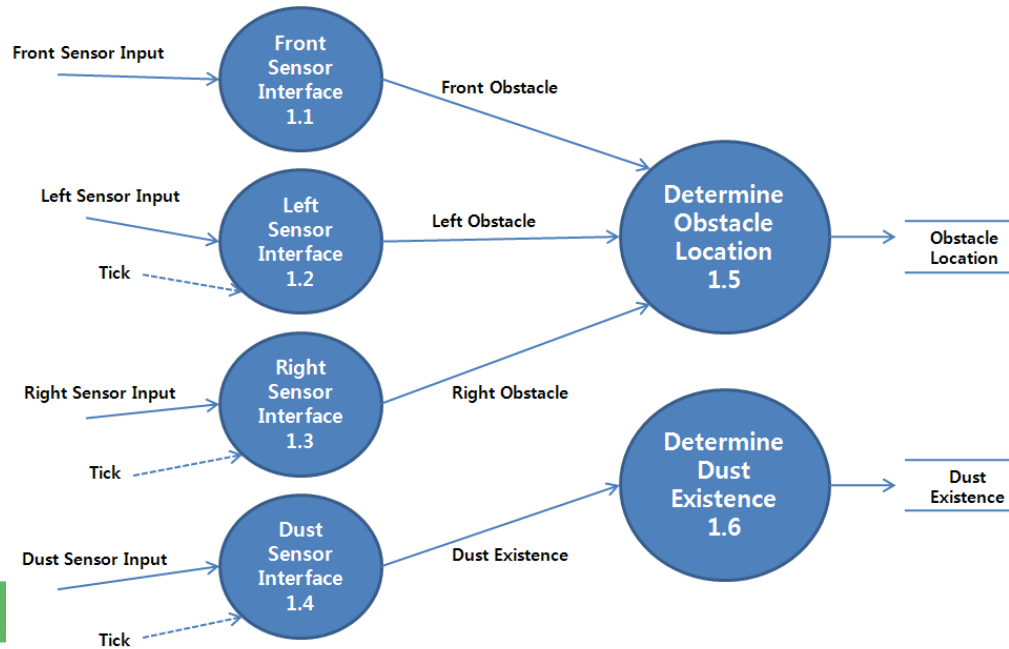
DFD Overview – RVC Example

(Problems Are Not Resolved Yet)



Process Specification

- Shows **process details** which are implied but not shown in a DFD.
 - Specifying the input, output, and algorithm of a module in a DFD
 - Normally written in pseudo-code or **table format**
 - Specifying all (upper/lower) processes in DFD hierarchies
- Example : Left Sensor Interface



Ref. No.	1.2
Name	Left Sensor Interface
Input	Left Sensor Input (+Data structure if possible) , Tick
Output	Left Obstacle (+Data structure)
Process Description	“Left Sensor Input” process reads an analog value of the left sensor periodically, converts it into a digital value such as True/False, and assigns it into output variable “Left Obstacle.”

Data Dictionary

- Defines data elements to avoid different interpretations.
 - Often used in a simple form like below

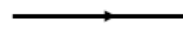
• Example :

Input/ Output Event	Description	Format / Type
Front Sensor Input	Detects obstacles in front of the RVC	True / False , Interrupt
Left Sensor Input	Detects obstacles in the left side of the RVC periodically	True / False , Periodic
Right Sensor Input	Detects obstacles in the right side of the RVC periodically	True / False , Periodic
Dust Sensor Input	Detects dust on the floor periodically	True / False , Periodic
Direction	Direction commands to the motor (go forward / turn left with an angle / turn right with an angle)	Forward / Left / Right / Stop
Clean	Turn off / Turn on / Power-Up	On / Off / Up
Front Obstacle		
Left Obstacle		
Right Obstacle		
Dust Existence		
Obstacle Location		
Dust Existence (2)		

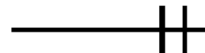
Entity Relationship Diagram (ERD)

- A graphical representation of the **data layout** of a system at a high level of abstraction
 - Defines data elements and their inter-relationships in the system.
 - Similar with the class diagram in UML.

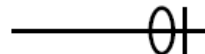
- Notation (Original)



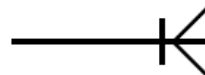
Associated Object



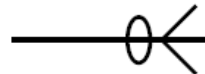
Cardinality – Exactly one



Cardinality – Zero or one

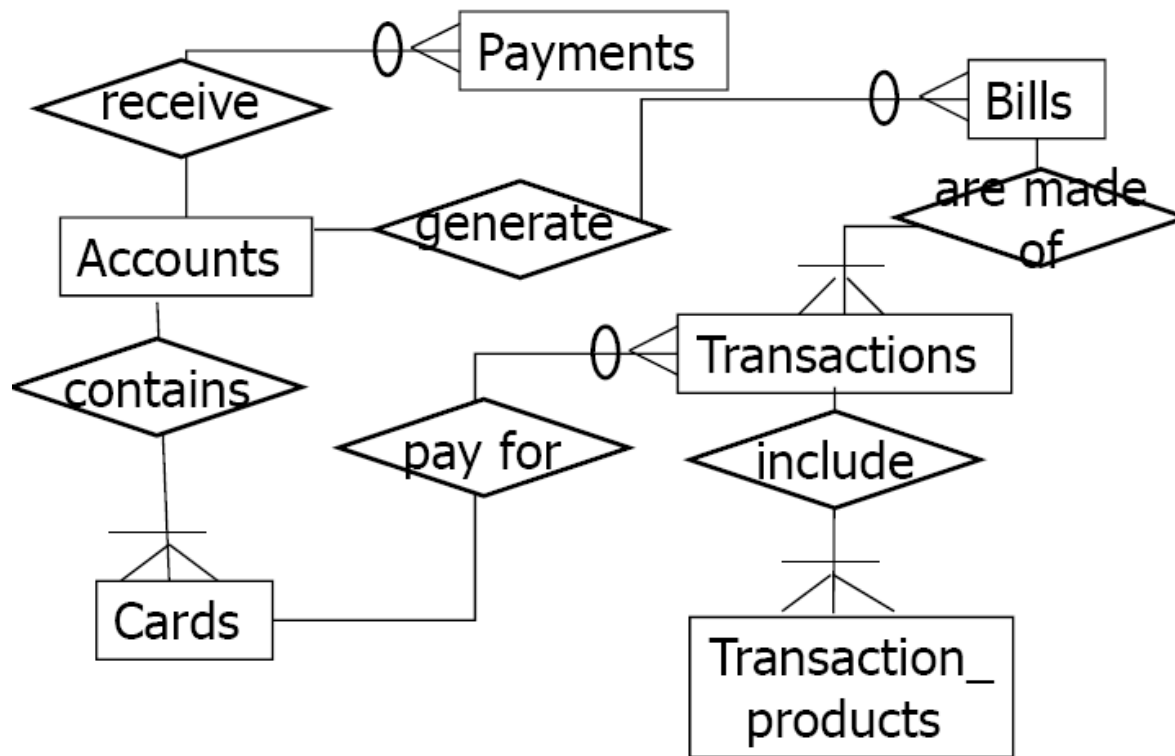


Cardinality – Mandatory Many

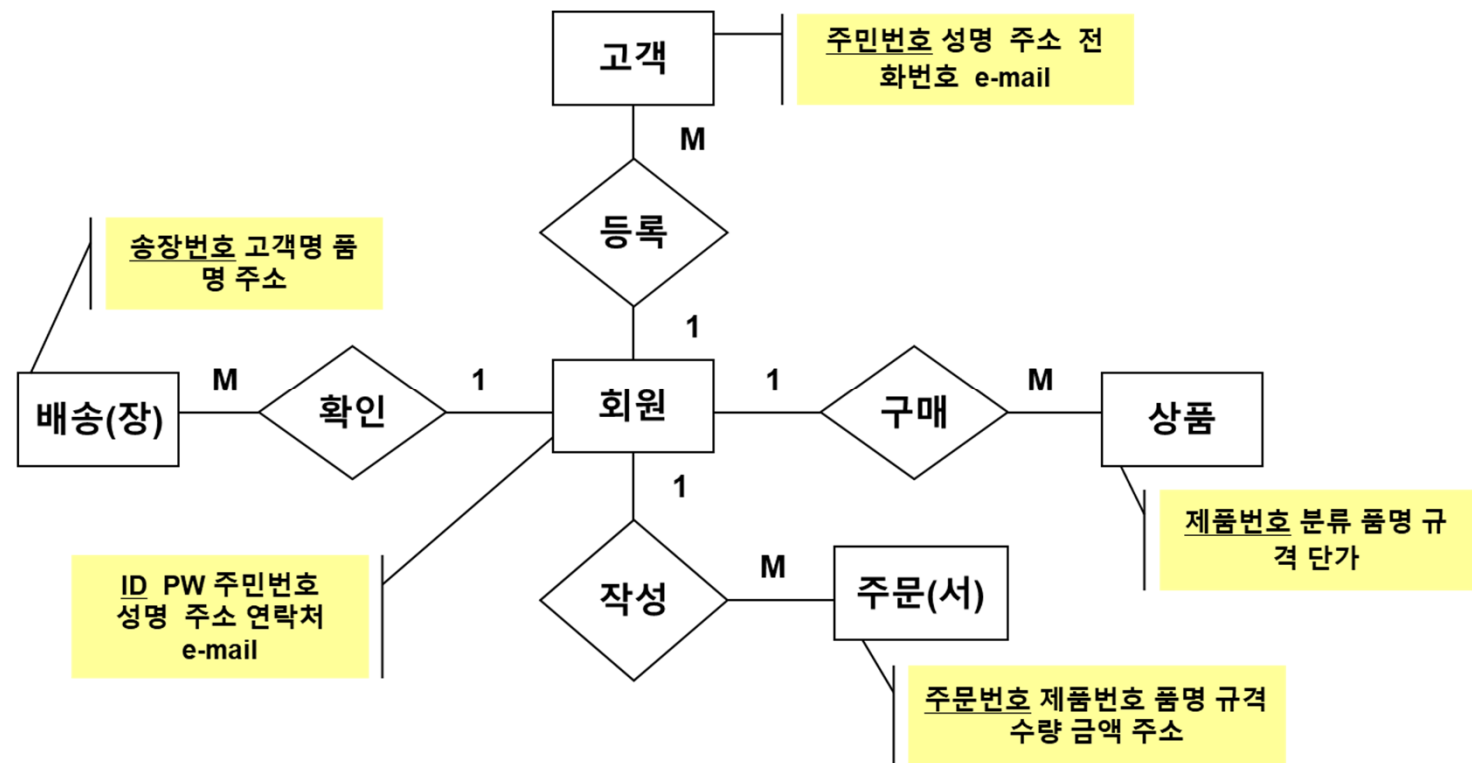
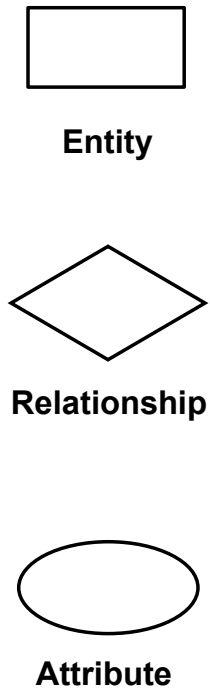


Cardinality – Optional Many

Entity Relationship Diagram - Example



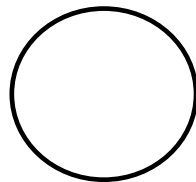
Entity Relationship Diagram - Example



State Transition Diagram

- Shows **the time ordering between processes**.
 - More primitive than the Statecharts Diagram in UML
 - Different from the State transition diagram (FSM) used in DFD
 - Similar with the **UML Activity Diagram**
 - Not widely used now.

- Notation:

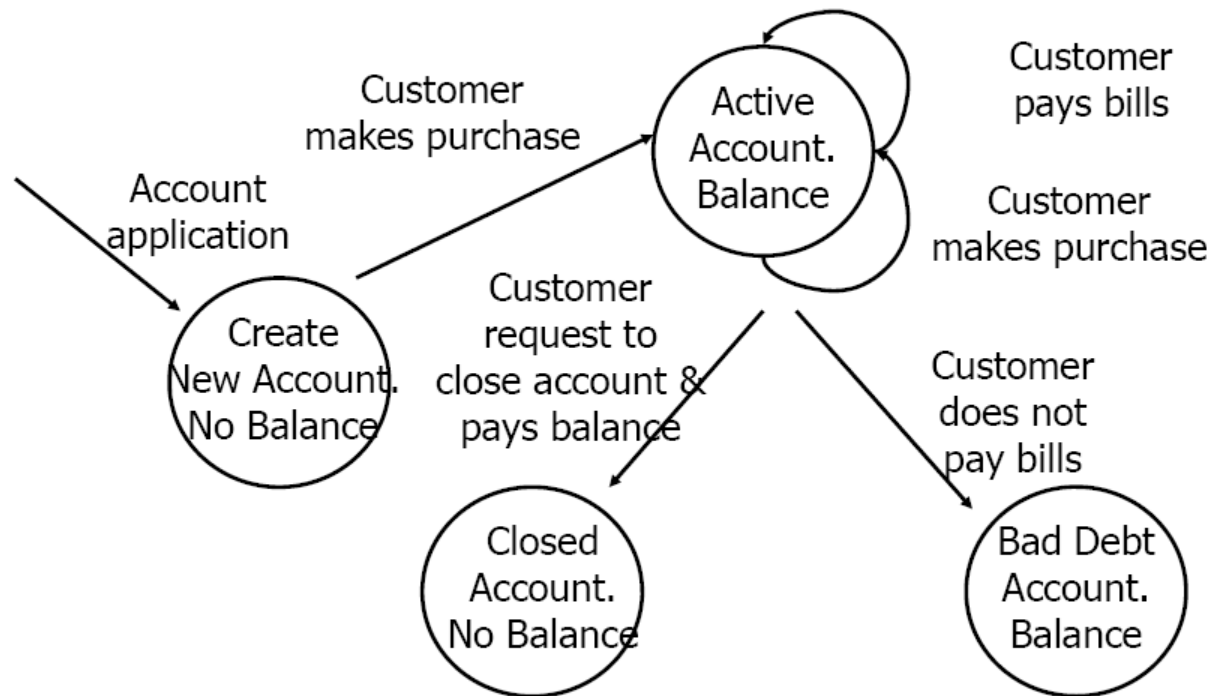


Objects
(Process)

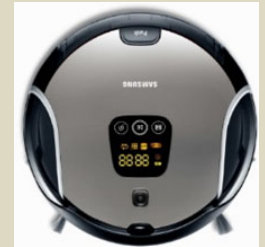


Transitions

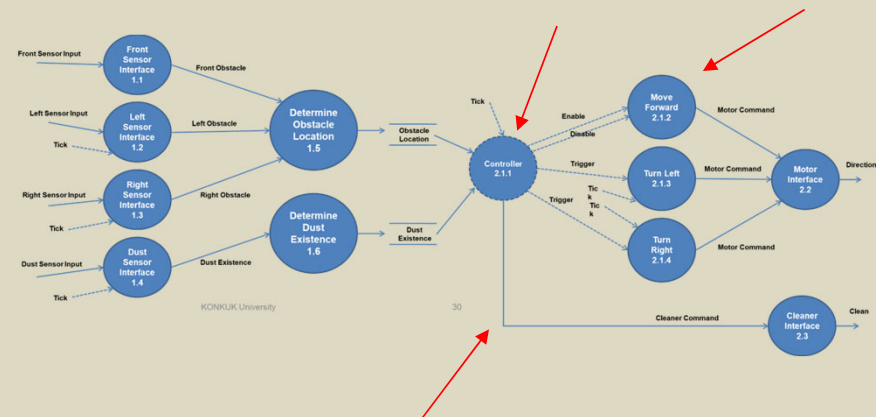
State Transition Diagram - Example



Homework / Activity #6



- **Structured Analysis for the RVC Control Software**
 - Complete the analysis for the RVC Controller in more details.
 - Resolve the problems identified
 - “Stop” state (deadlock)
 - Not consider “Dust”
 - No Priority for left/right turn
 - Not consider “Backward”
 - Inconsistent design of Cleaner Interface with Motor Interface
 - Complete full versions of process specifications and data dictionary
 - System Context Diagram
 - A hierarchy of DFDs + FSM
 - Process Specifications
 - Data Dictionary
 - PPT 로 작성하세요.



Homework #6

- SA 기법을 활용해서 분석한 RVC Control SW에 대한 요구사항명세서(SRS)를 IEEE Std 830-1998를 준수하여 문서로 완성하세요.
 - 적절한 Templates을 사용하세요.

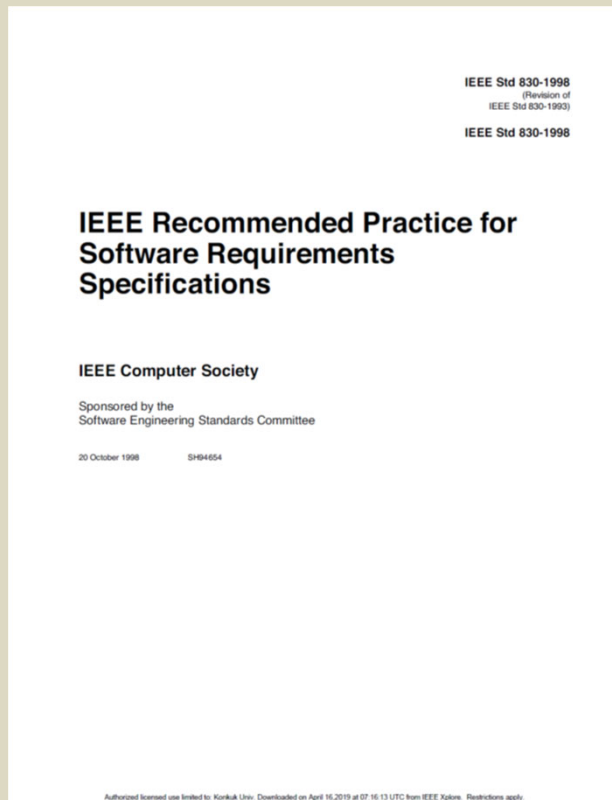


Table of Contents
1. Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, acronyms, and abbreviations
1.4 References
1.5 Overview
2. Overall description
2.1 Product perspective
2.2 Product functions
2.3 User characteristics
2.4 Constraints
2.5 Assumptions and dependencies
3. Specific requirements (See 5.3.1 through 5.3.8 for explanations of possible specific requirements. See also Annex A for several different ways of organizing this section of the SRS.)
Appendixes
Index

Figure 1 – Prototype SRS outline

Structured Design (SD) - An Example of RVC SW Controller



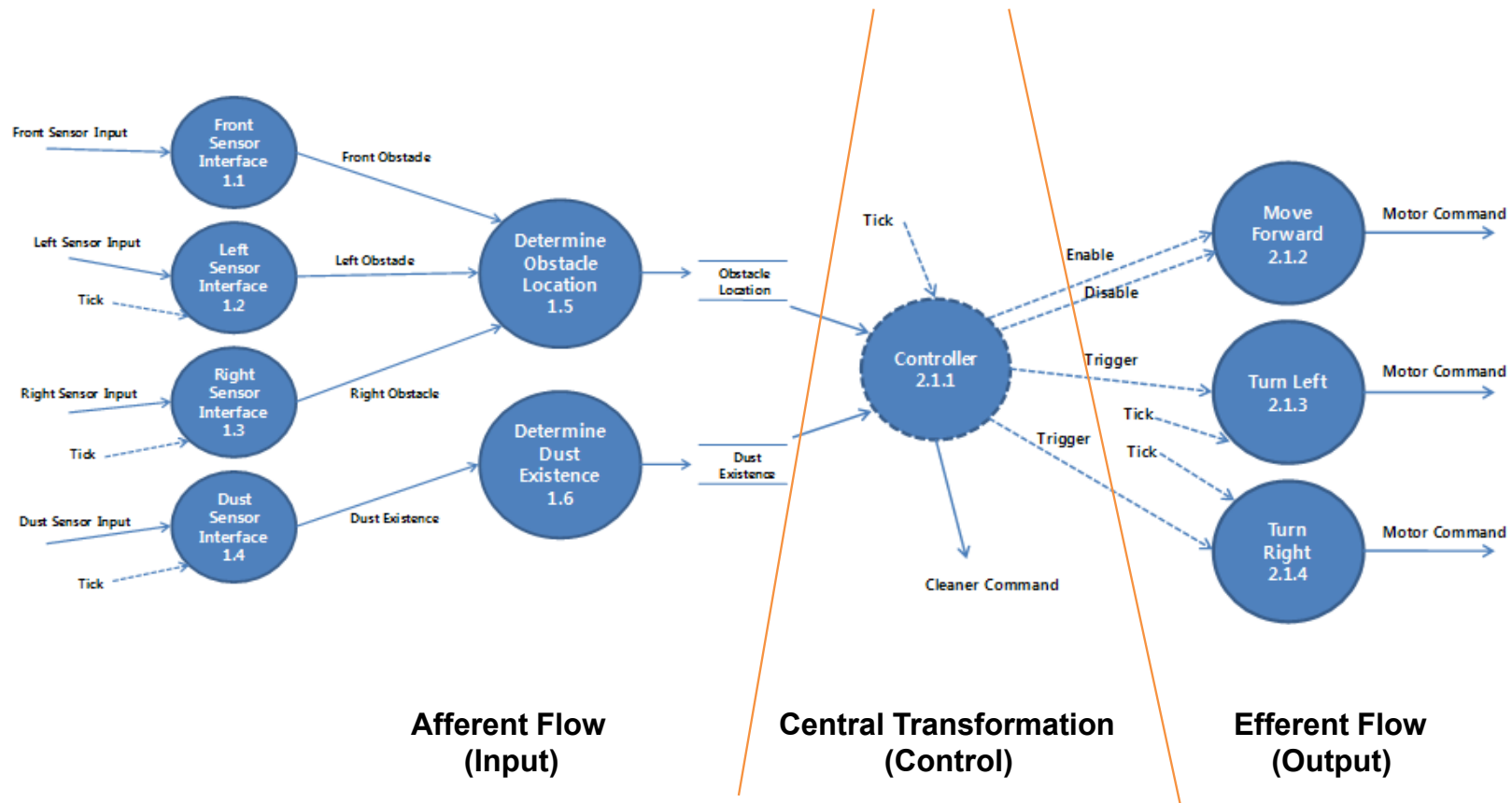
Structured Design

- Needs **transform analysis**
 - No data comes, being processed, and goes out by itself.
 - Somebody should call input/output processes to do something.

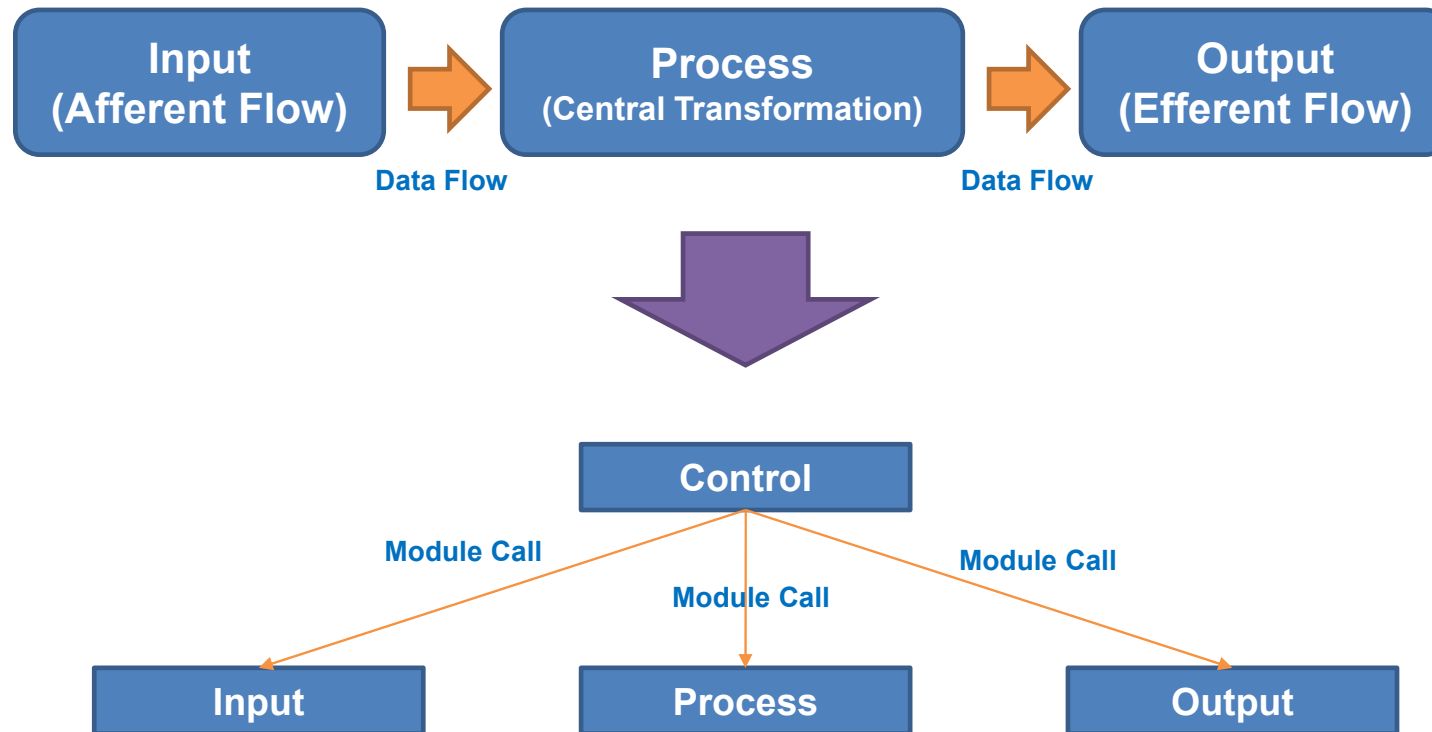
- Needs to design functional decomposition according to SA
 - Information hiding
 - Modularity
 - Low coupling
 - High internal cohesion

- Many models were proposed, but not widely used except
 - **Structured Charts**

Structured Design – Transform Analysis

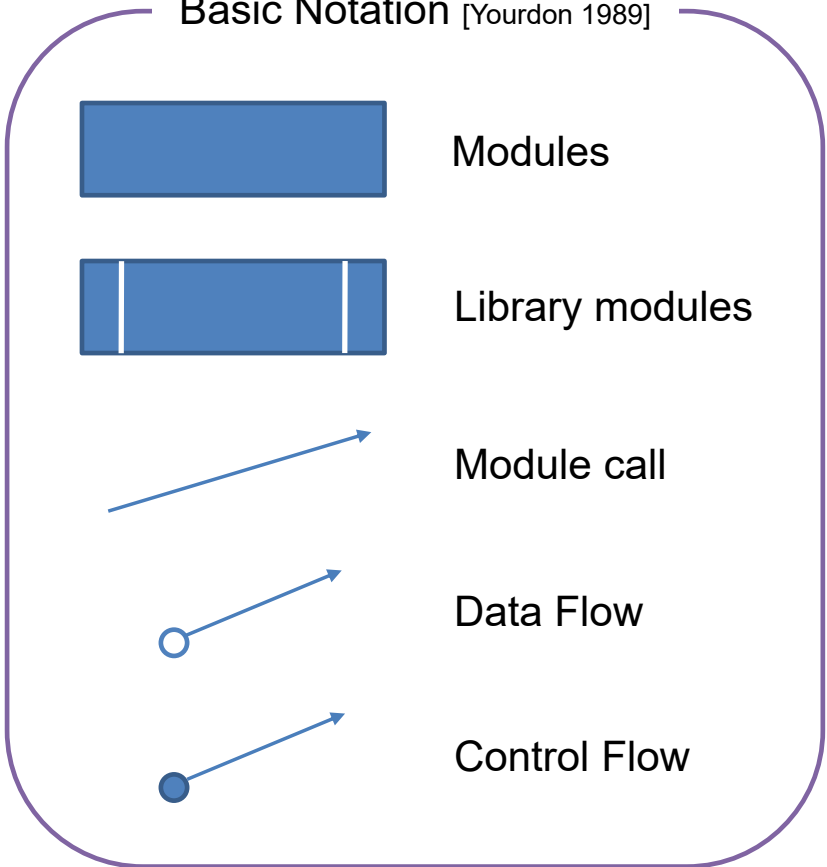


Structured Design – Transform Analysis

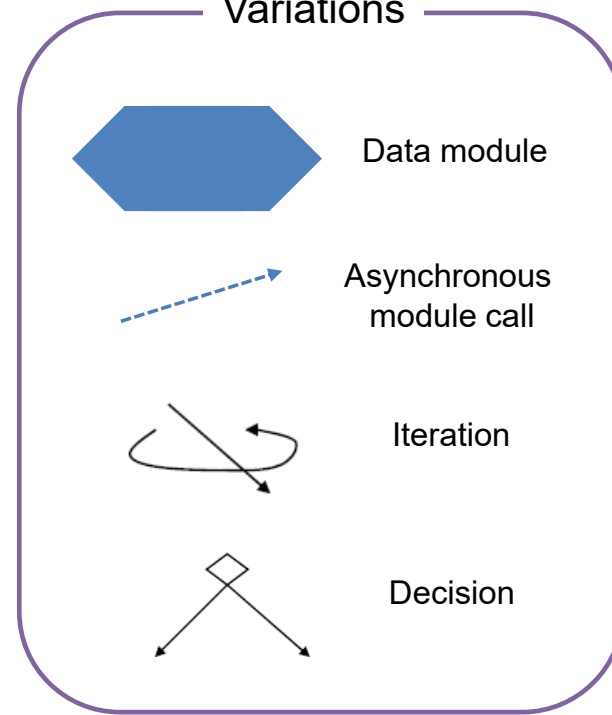


Structured Charts – Notation

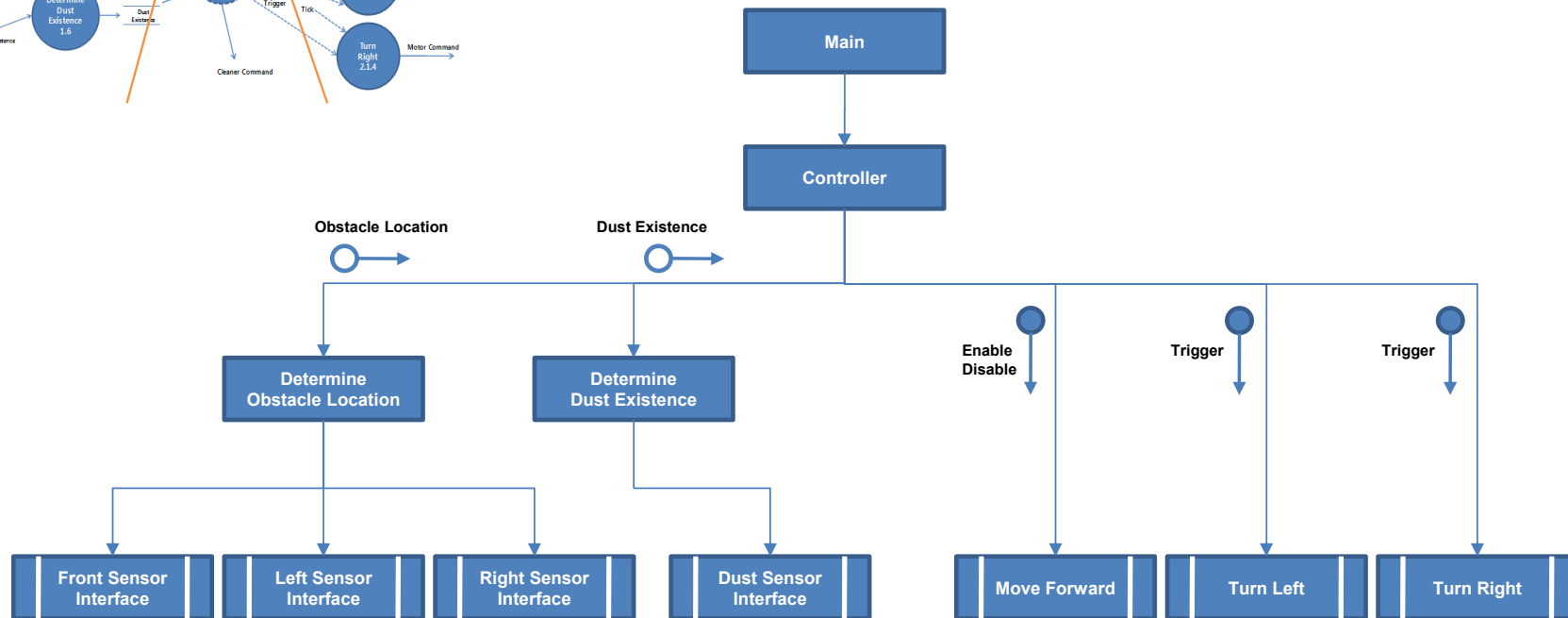
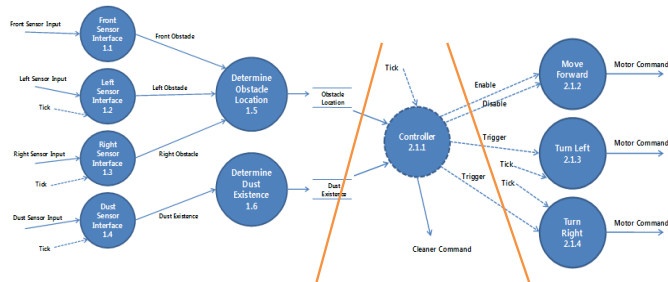
Basic Notation [Yourdon 1989]



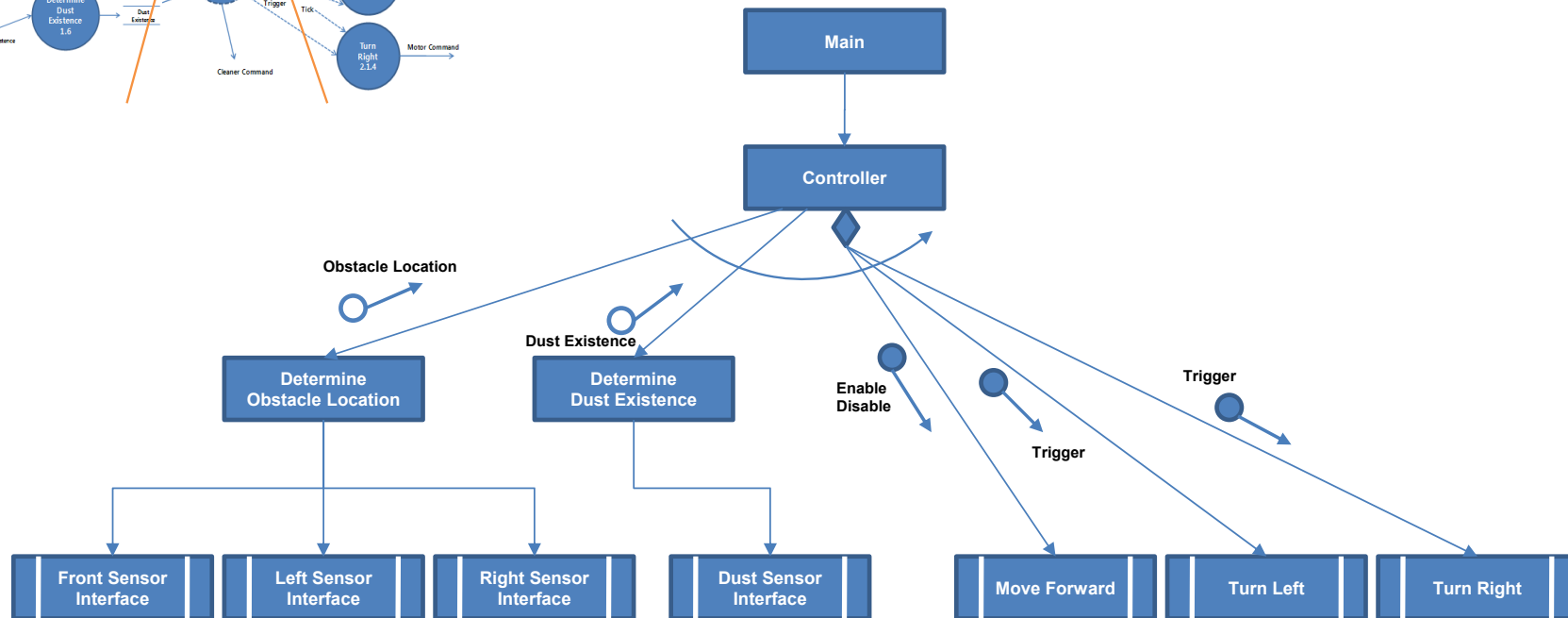
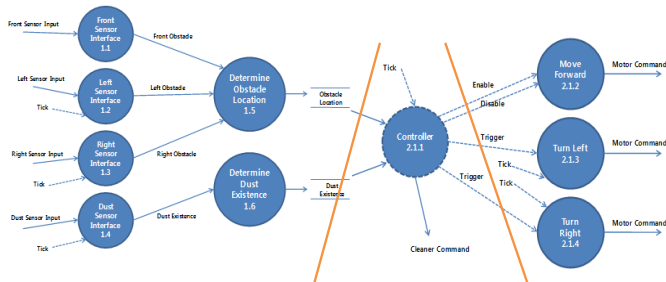
Variations



Structured Charts – RVC Example (Basic)



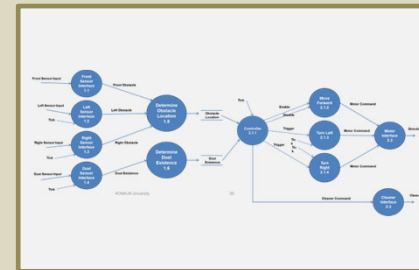
Structured Charts – RVC Example (Advanced)



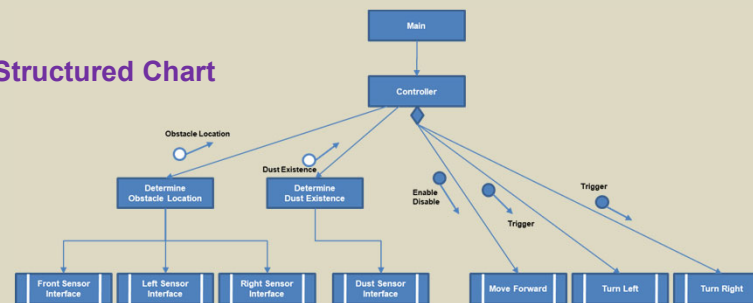
Homework / Activity #7

- **Complete the RVC structured design.**
 - Complete a full version of Structured Charts on your own.
- **Implement a C program as detail as possible.**
 - C program (might be executable with libraries emulated)
 - Based on the DFD and Structured Charts

DFD + Process Specifications



Structured Chart



C Program

```
void main( )
{
  int obstacle_Location;
  bool dust_Existence;

  while(1)
  {
    obstacle_Location = Det_OL( );
    dust_Existence = Det_DE( );

    if( ... )

    wait(200ms);
  }
}
```

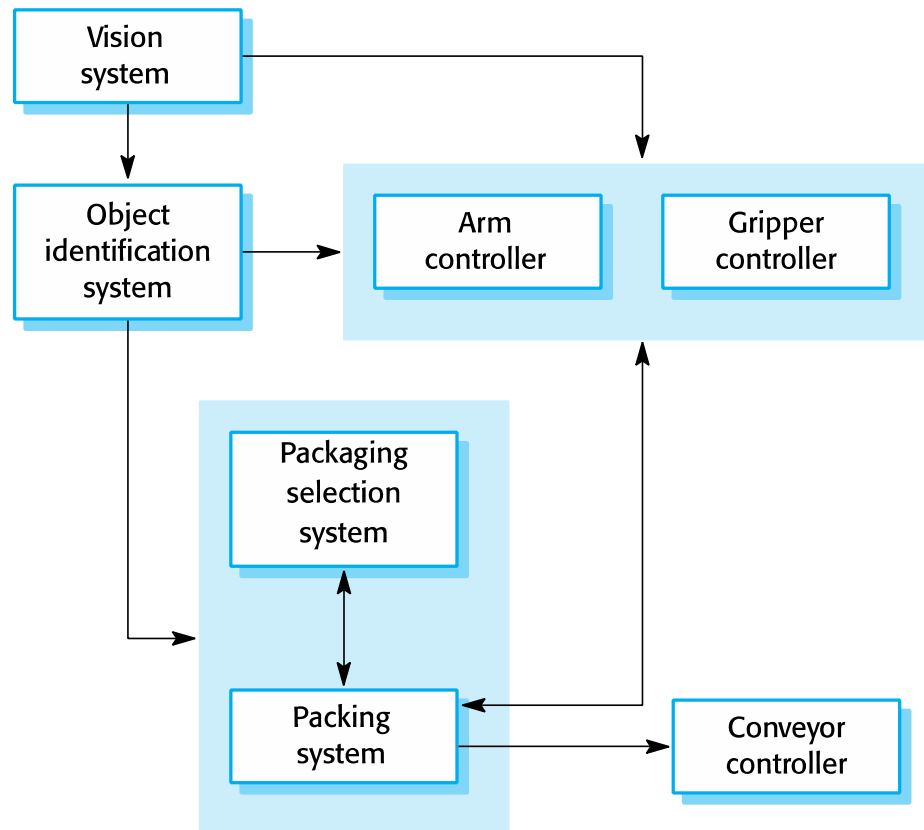

6. Architectural Design

Architectural Design

- **Architectural design** is concerned with understanding how a software system should be organized and designing the overall structure of that system.
 - A critical link between requirements engineering and design
 - Identifies the main structural components in a system and the relationships between them

- **Architecture model** describes how the system is organized as a set of communicating components.

The Architecture of Packing Robot Control System



Architectural Abstraction

- **Architecture in the large**
 - Concerned with the architecture of complex enterprise systems that include other systems, programs and program components
 - Enterprise systems are distributed over different computers, which may be owned and managed by different companies.

- **Architecture in the small**
 - Concerned with the architecture of individual programs
 - Concerned with the way that an individual program is decomposed into components

Advantages of Architectural Design

- Stakeholder communication
 - Architecture may be used as a focus of discussion by system stakeholders.

- System analysis
 - Analysis of whether the system can meet its non-functional requirements is possible.

- Large-scale reuse
 - The architecture may be reusable across a range of systems.
 - Product-line architectures may be developed.

Architectural Models

- **Representation of architecture models**
 - Simple, informal block diagrams
 - Box and Line Diagrams
 - Extensions of UML models

- Use **architecture models**
 - As a way of **facilitating discussion** about the system design
 - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail.
 - Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
 - As a way of **documenting** an architecture that has been designed
 - To produce a complete system model that shows the different components in a system, their interfaces and their connections.

Architectural Representations

- **Simple, informal block diagrams**

- Showing entities and relationships simply
- The most frequently used method for documenting software architectures
- However, lack of semantics do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- The semantics of architectural models depend on how the models are used.

- **Box and Line Diagrams**

- Very abstract - not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

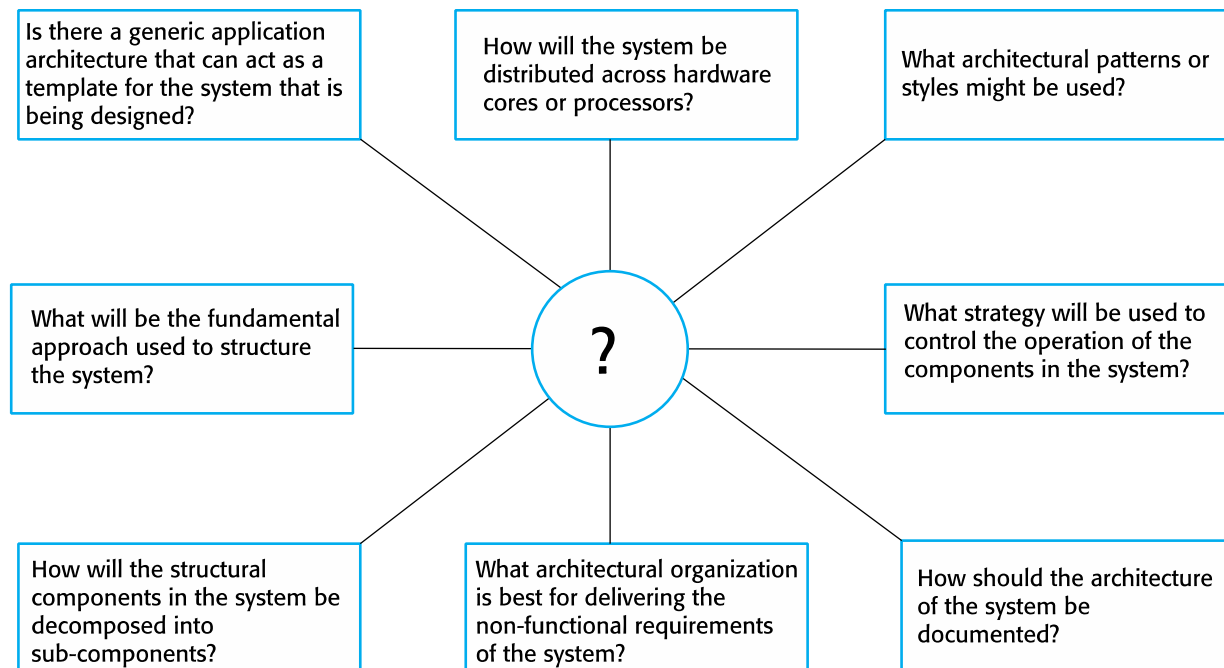
- **Extensions of UML models**

- Extending component diagrams including class diagrams, object diagrams, composite structure diagrams.
- Not widely used yet.

Architectural Design Decisions

Architectural Design Decisions

- **Architectural design** is a creative process.
 - The design process differs depending on the type of system being developed.
- However, several **common design decisions** span all design processes.
 - Affect the non-functional characteristics of the system



Architecture and System Characteristics

- **Performance**
 - Localize critical operations and minimize communications. Use large rather than fine-grain components

- **Security**
 - Use a layered architecture with critical assets in the inner layers

- **Safety**
 - Localize safety-critical features in a small number of sub-systems

- **Availability**
 - Include redundant components and mechanisms for fault tolerance

- **Maintainability**
 - Use fine-grain, replaceable components

Architecture Reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
 - **Application product lines** are built around a core architecture with variants that satisfy particular customer requirements.

- The architecture of a system may be designed around one of more **architecture patterns** or **architecture styles**.
 - Capture the essence of an architecture
 - Can be instantiated in different ways

Architectural Views

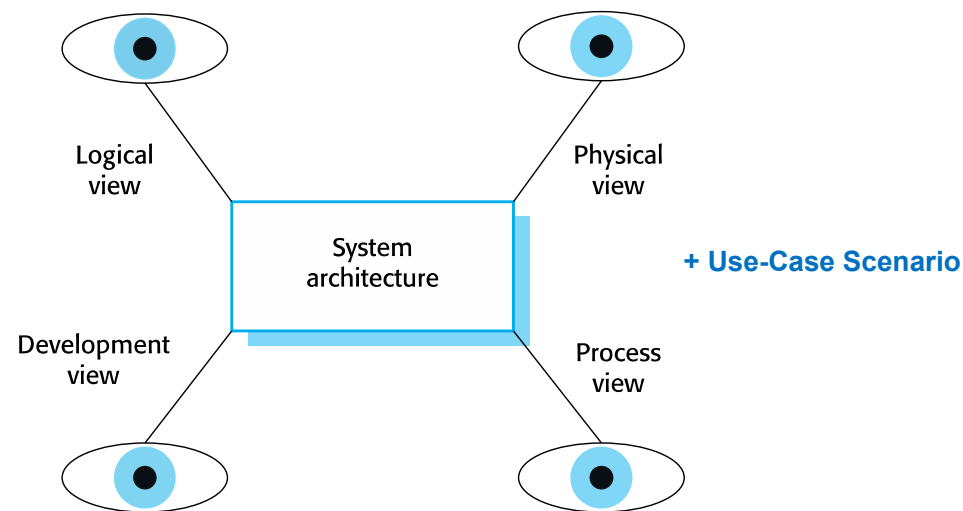
Architectural Views

- Each architectural model only shows **one view** showing
 - How a system is decomposed into modules,
 - How the run-time processes interact, or
 - Which system components are distributed across a network.

- We need **multiple views** of the software architecture for both design and documentation purposes.
 - What views are useful when designing and documenting a system's architecture?
 - What notations should be used for describing architectural models?

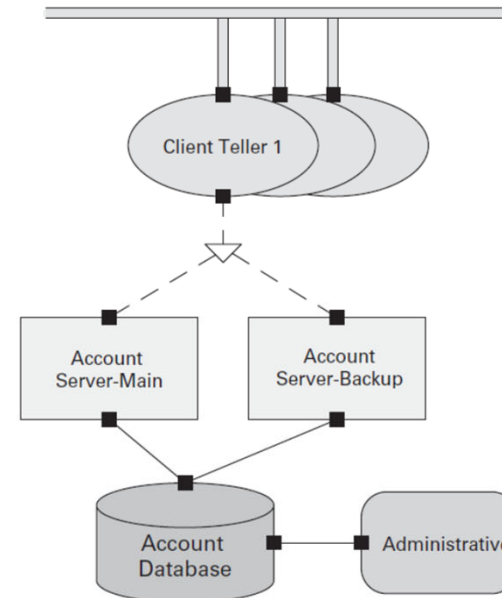
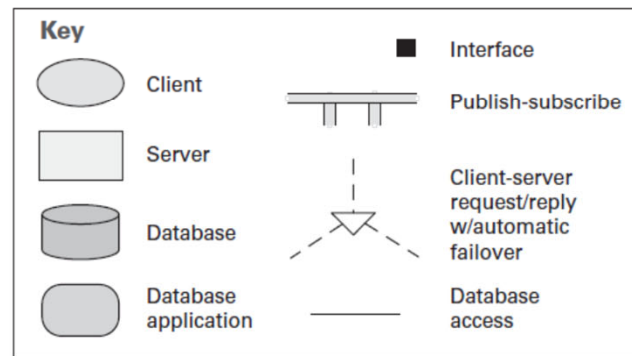
4 + 1 View Model of Software Architecture

- **Logical view** : showing the key abstractions in the system as objects or object classes
- **Process view** : showing how, at run-time, the system is composed of interacting processes
- **Development view** : showing how the software is decomposed for development
- **Physical view** : showing the system hardware and how software components are distributed across the processors in the system
- Related 4 views with **use cases or scenarios** (+1)



Representing Architectural Views

- **Unified Modeling Language (UML)** is a candidate notation for describing and documenting system architectures.
 - Component diagram, Package diagram, Class diagram, etc.
 - However, UML does not include abstractions appropriate for high-level system description.
- **Architectural description languages (ADLs)** have been developed, but not widely used yet.
- **Naive diagrams** have been widely used.
 - Example: C&C View

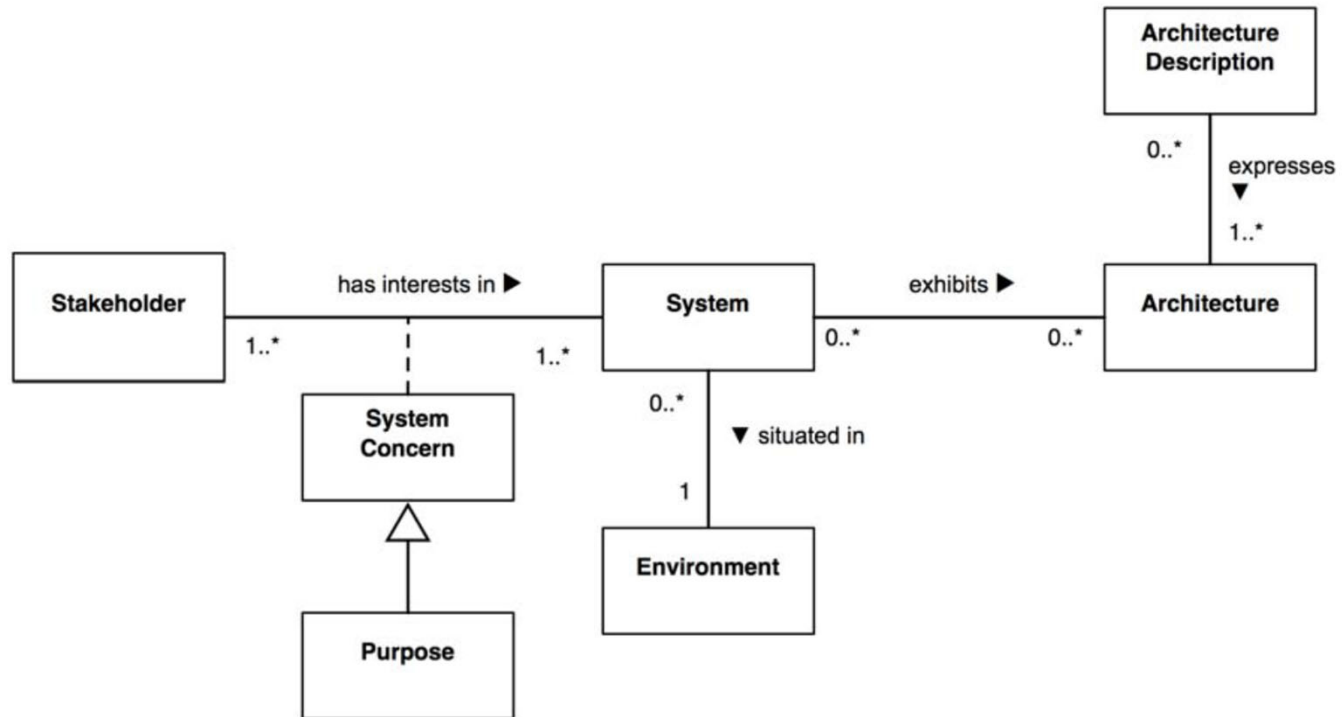


Describing Software Architectures

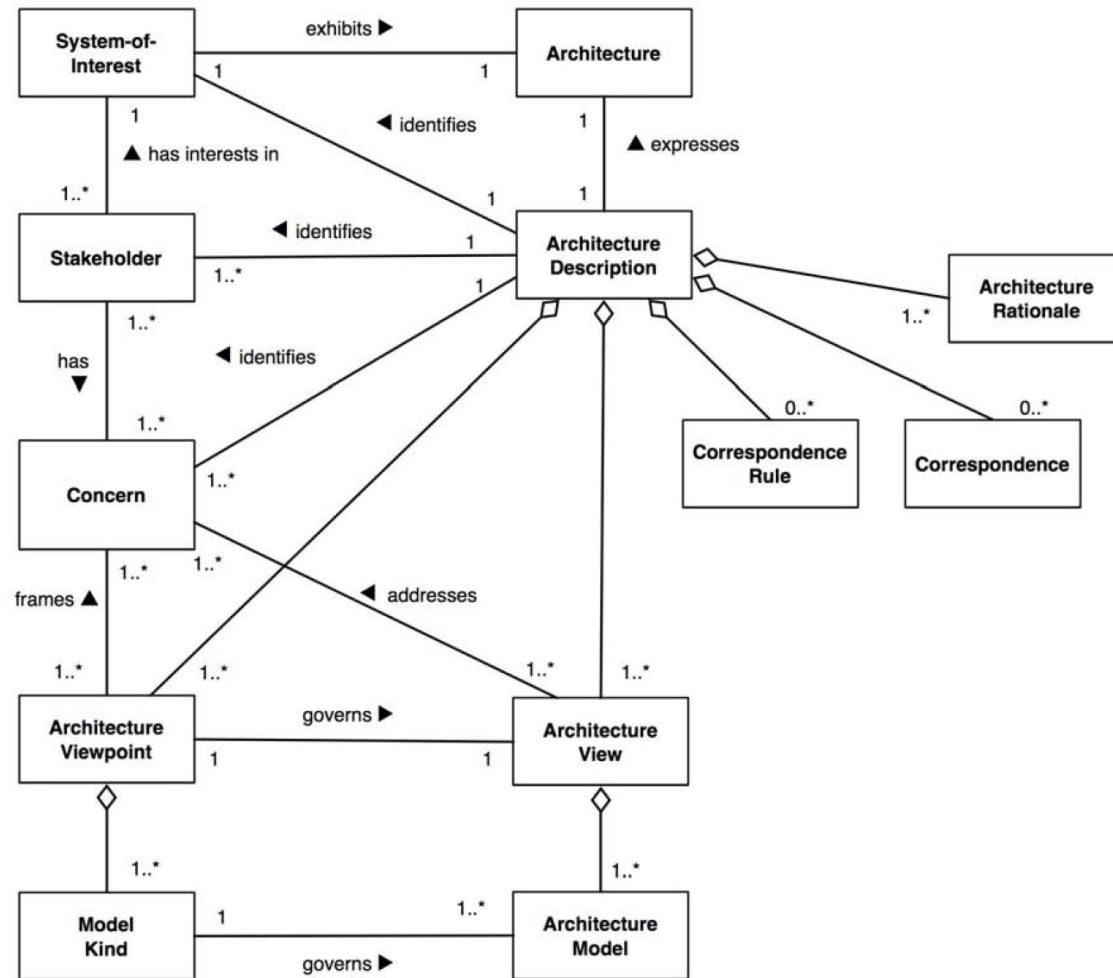
- **ISO/IEC/IEEE 42010:2011 “Systems and Software Engineering - Architecture Description”**
 - Specifies the **requirements** for (to be an) **architecture descriptions (AD)**

- **Key Principles of the Architecture Description Standard**
 - AD should demonstrate how an architecture meets the **needs** of the system’s diverse **stakeholders**.
 - The architectural concerns of the diverse stakeholders can be addressed by an AD constructed with multiple **architecture views** of the system, where each view covers a subset of those **concerns**.
 - The rules for well-formedness, completeness and analyzability of each architecture view should be explicit via an **architecture viewpoint**.

Context of Architecture Description



A Conceptual Model of Architecture Description



Terminologies

Terminology	Definition
Architecture	<ul style="list-style-type: none"> fundamental concepts or properties of a system in its environment, embodied in its elements, relationships, and in the principles of its design and evolution
Architecture Description (AD)	<ul style="list-style-type: none"> work product used to express an architecture
Architecture Framework	<ul style="list-style-type: none"> conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders Examples <ul style="list-style-type: none"> - Generalized Enterprise Reference Architecture and Methodologies (GERAM) [ISO 15704] - Reference Model of Open Distributed Processing (RM-ODP) [ISO/IEC 10746]
Stakeholder	<ul style="list-style-type: none"> individual, team, organization, or classes thereof, having an interest in a system Examples: users, acquirers, developers, maintainers, etc.
Architecture View	<ul style="list-style-type: none"> work product expressing the architecture of a system from the perspective of specific system concerns
Architecture Viewpoint	<ul style="list-style-type: none"> work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns
Concern	<ul style="list-style-type: none"> interest in a system relevant to one or more of its stakeholders a concern pertains to any influence on a system in its environment, including developmental, technological, business, operational, organizational, political, economic, legal, regulatory, ecological and social influences
Model Kind	<ul style="list-style-type: none"> conventions for a type of modelling Examples: data flow diagrams, class diagrams, Petri nets, balance sheets, organization charts and state transition models

Architectural Patterns

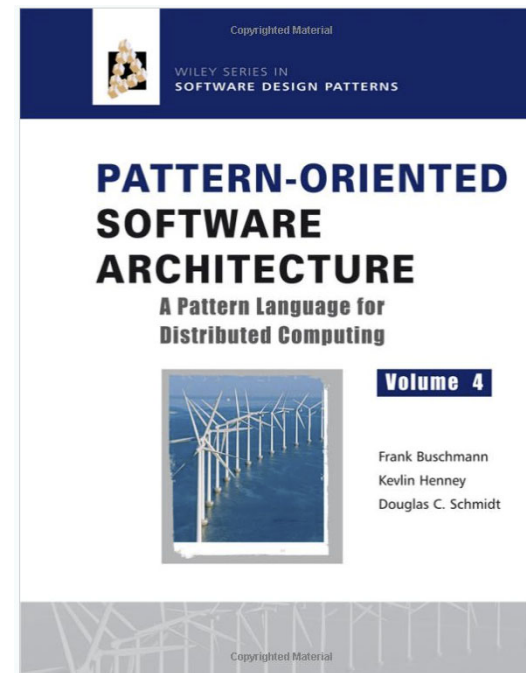
Architectural Pattern/Style

- **Architectural pattern**

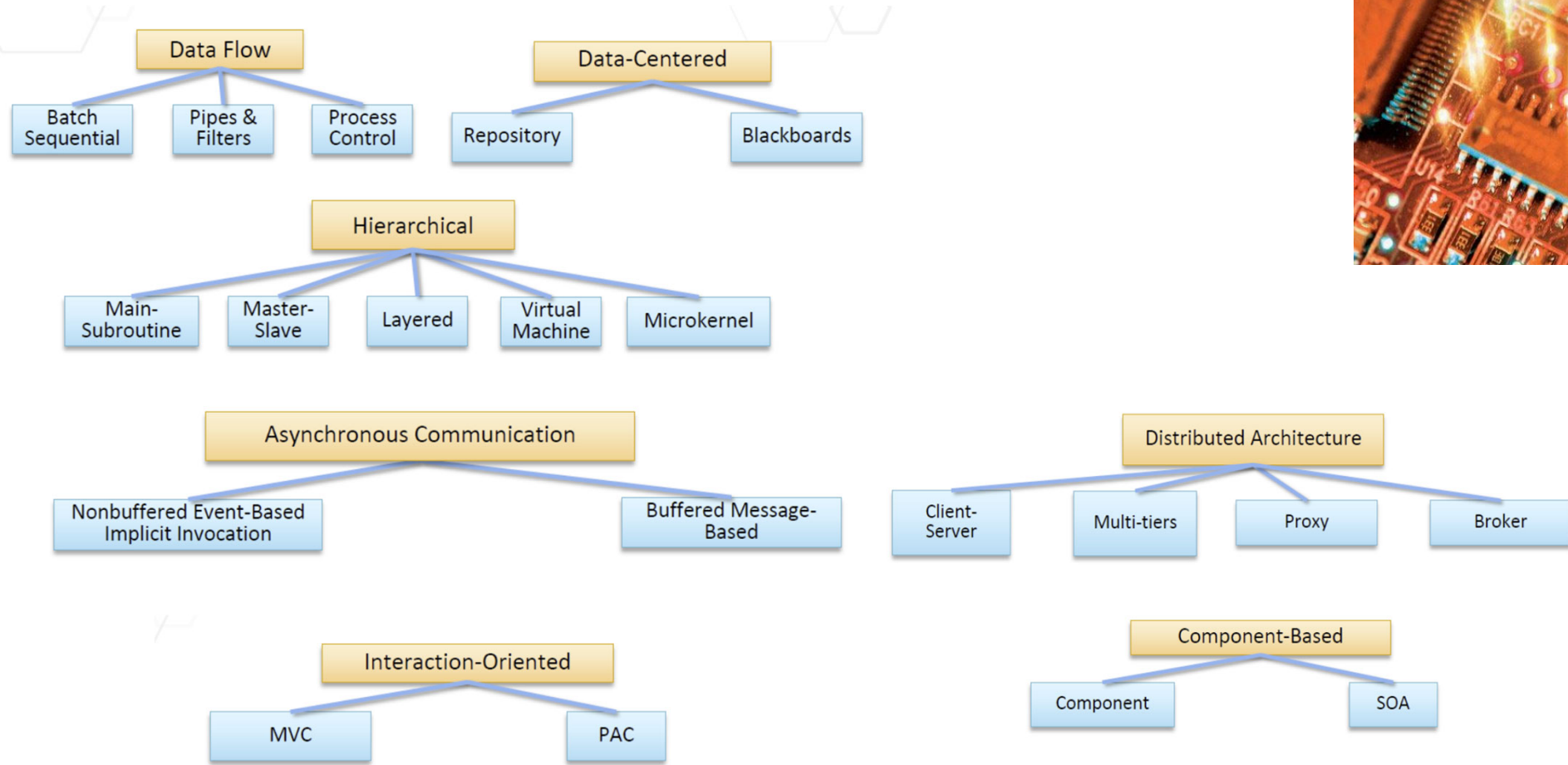
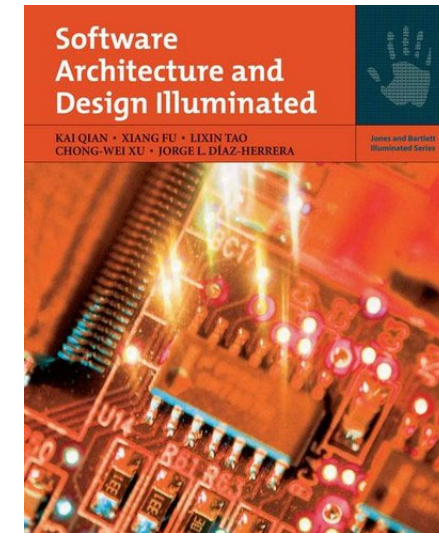
- A stylized description of **good design practice**, which has been tried and tested in different environments
- Include information about when they are and when they are not useful.

- Examples: A series of **POSA**

- **MVC (Model-View-Controller)**
- **Layered**
- **Repository**
- **Client-Server**
- **Pipe & Filter**
- etc.

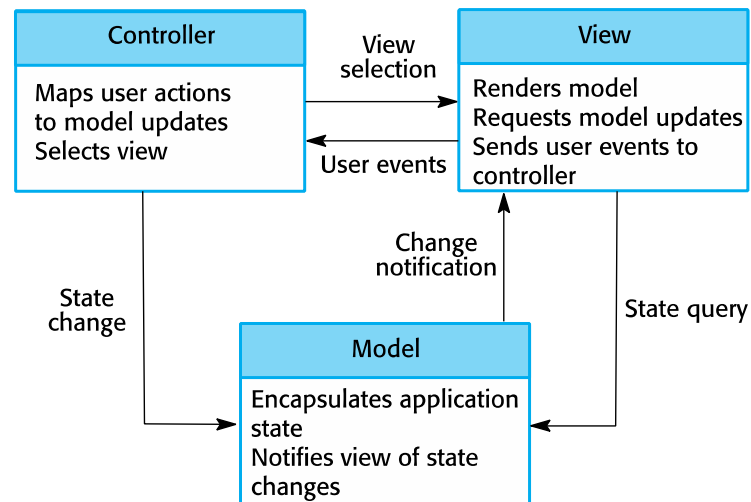


A Taxonomy of Architecture Patterns/Styles

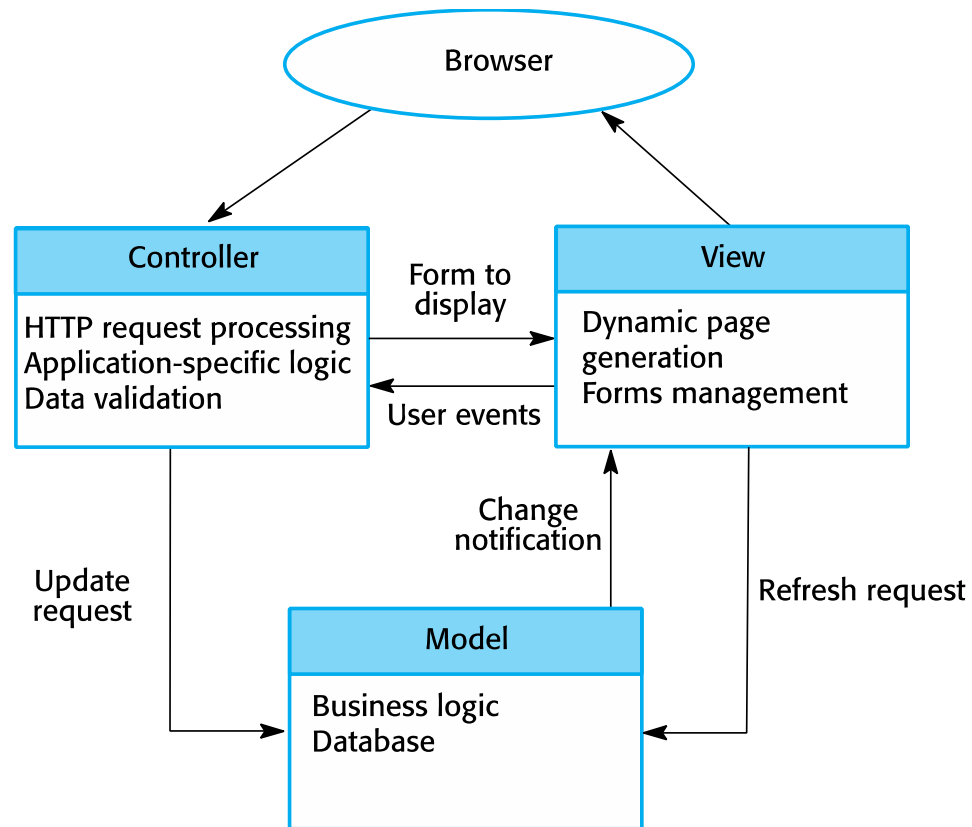


The Model-View-Controller (MVC) Pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

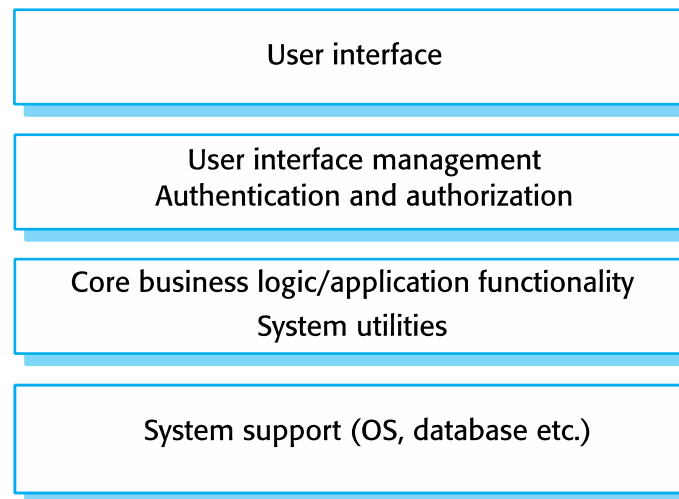


Example : Web Application Architecture



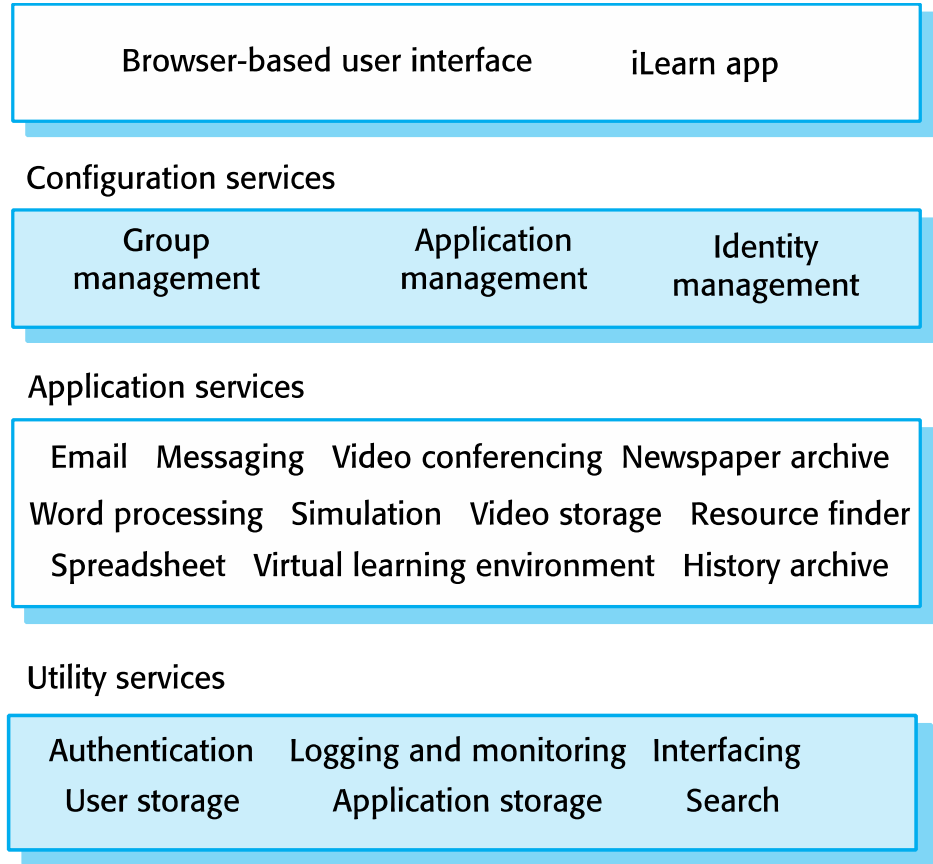
The Layered Architecture Pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
Example	A layered model of a system for sharing copyright documents held in different libraries.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsible for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.



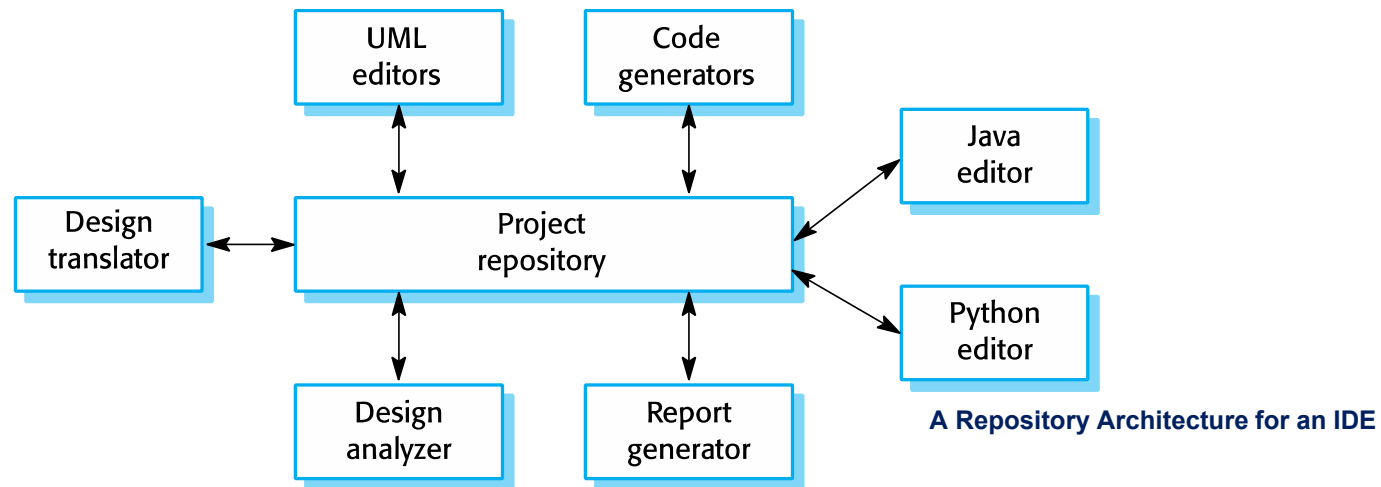
A Generic Layered Architecture

Example : The iLearn System



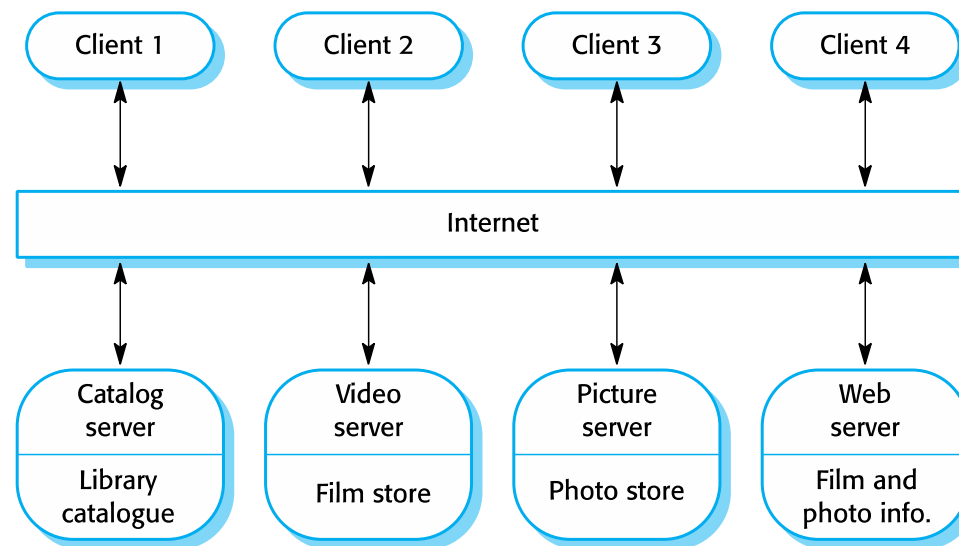
The Repository Pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent. They do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.



The Client-Server Pattern

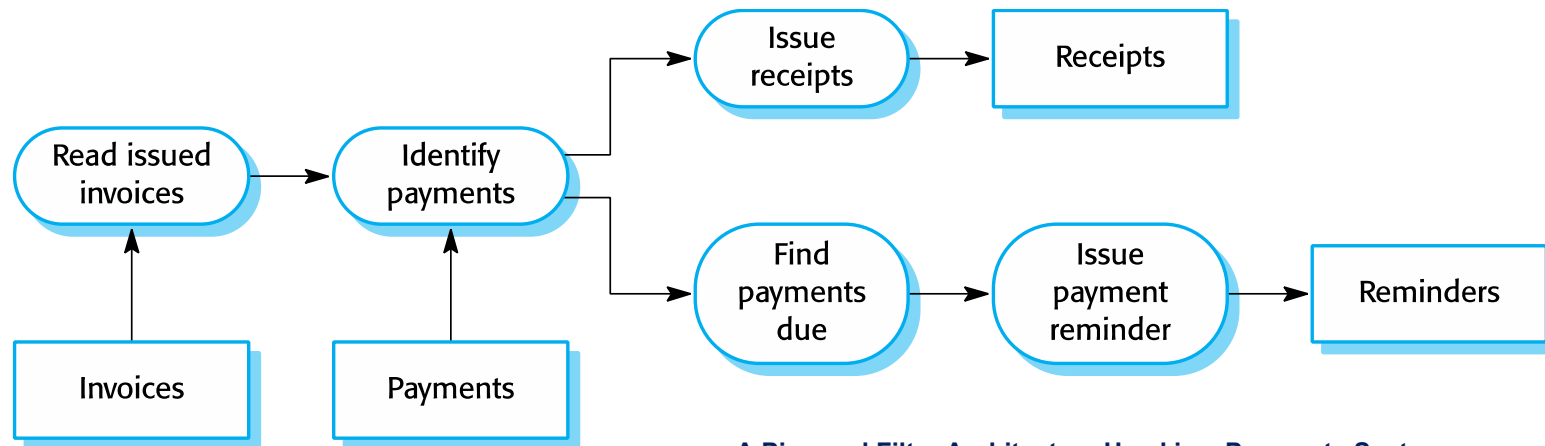
Name	Client-Server
Description	In a client-server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	An example of a film and video/DVD library organized as a client-server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.



A Client-Server Architecture for a Film Library

The Pipe and Filter Pattern

Name	Pipe and Filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	An example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.



A Pipe and Filter Architecture Used in a Payments System

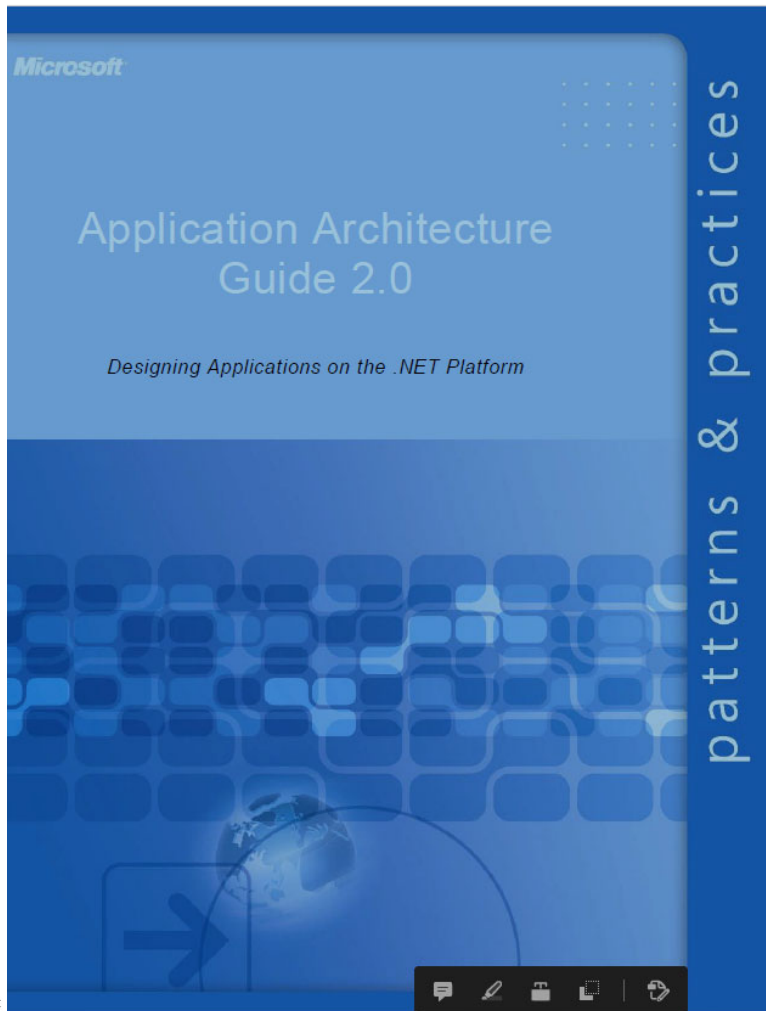
Application Architectures

Application Architectures

- **Application architecture**
 - An architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements
 - As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
 - → Evolving into “**Reference Architecture**”

- Application Types
 - Data processing applications
 - Process data in batches without explicit user intervention during the processing
 - Transaction processing applications
 - Process user requests and update information in a system database
 - Event processing systems
 - Applications where system actions depend on interpreting events from the system’s environment
 - Language processing systems
 - Applications where the users’ intentions are specified in a formal language that is processed and interpreted by the system

Reference Architecture by Microsoft



Application Architecture Guide 2.0

- Chapter 1, "Fundamentals of Application Architecture"
- Chapter 2, ".NET Platform Overview"
- Chapter 3, "Architecture and Design Guidelines"

Part II, "Design"

This part provides an approach to architecture design and discusses key architecture decisions such as deployment, architecture style, quality attributes, and communication options. Part II includes the following chapters:

- Chapter 4, "Designing Your Architecture"
- Chapter 5, "Deployment Patterns"
- Chapter 6, "Architecture Styles"
- Chapter 7, "Quality Attributes"
- Chapter 8, "Communication Guidelines"

Part III, "Layers"

This part provides architectural and design approaches, as well as practices, for each layer, including the presentation, business, service, and data access layers. Part III includes the following chapters:

- Chapter 9, "Layers and Tiers"
- Chapter 10, "Presentation Layer"
- Chapter 11, "Business Layer Guidelines"
- Chapter 12, "Data Access Layer Guidelines"
- Chapter 13, "Service Layer Guidelines"

Part IV, "Archetypes"

This part provides patterns and design frames for each application archetype; including service applications, Web applications, rich client applications, rich Internet applications, and mobile applications. Part IV includes the following chapters:

- Chapter 14, "Application Archetypes"
- Chapter 15, "Web Applications"
- Chapter 16, "Rich Internet Applications (RIA)"
- Chapter 17, "Rich Client Applications"
- Chapter 18, "Services"
- Chapter 19, "Mobile Applications"
- Chapter 20, "Office Business Applications (OBA)"
- Chapter 21, "SharePoint Line-Of-Business (LOB) Applications"

Approach Used in This Guide

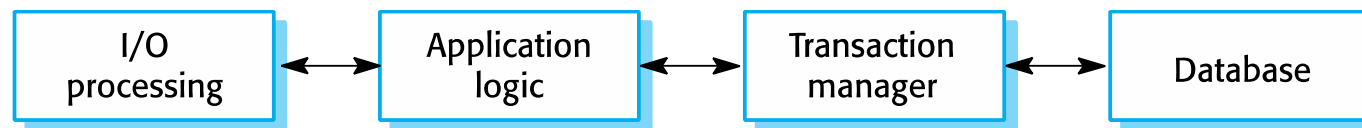
How do you design successful applications on the .NET platform? This guide describes an approach that starts with an understanding of the entire architectural process, and then focuses in on the specific topics, techniques, practices, and application types to help you

Transaction Processing Systems

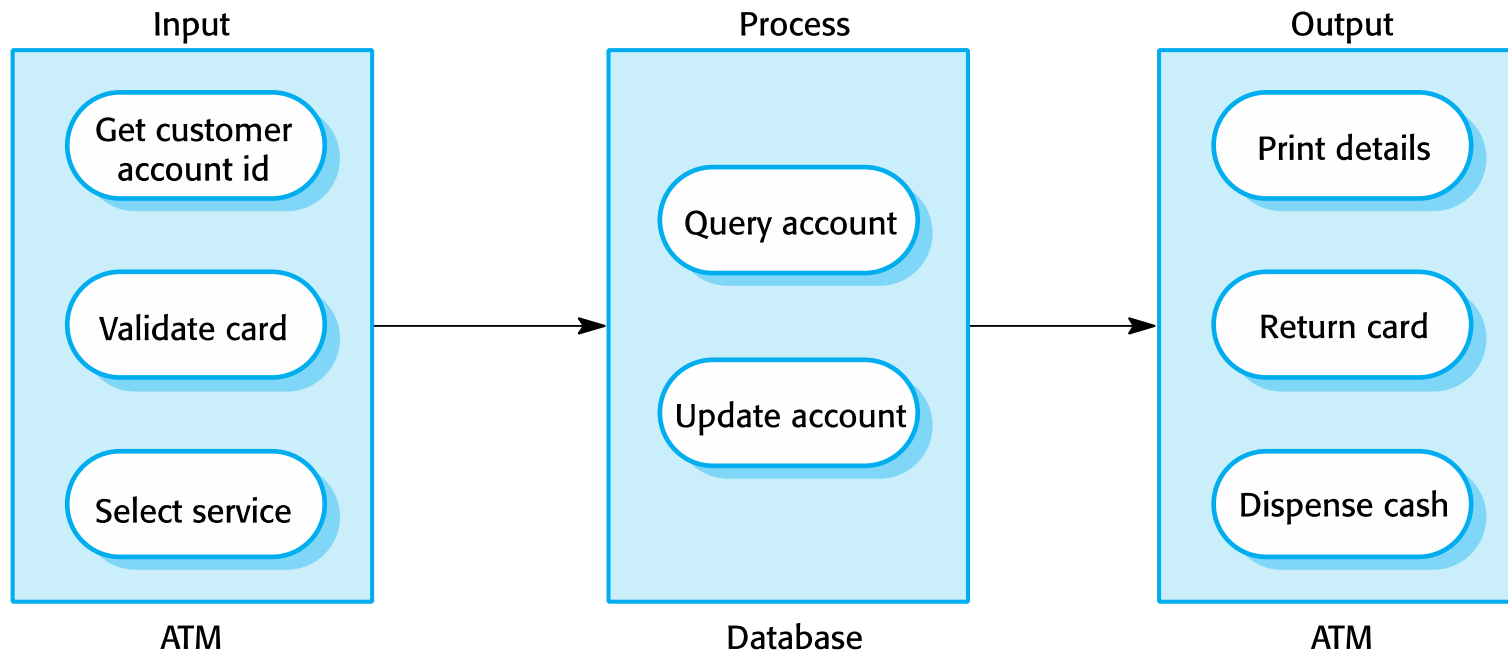
- **Transaction Processing Systems** process user requests for information from a **database**, or process requests to update the **database**.
 - Users make asynchronous requests for service which are then processed by a transaction manager.
 - A transaction is any coherent sequence of operations that satisfies a goal.

 - Example :
 - Find the times of flights from London to Paris

 - A typical structure of the TPS applications :



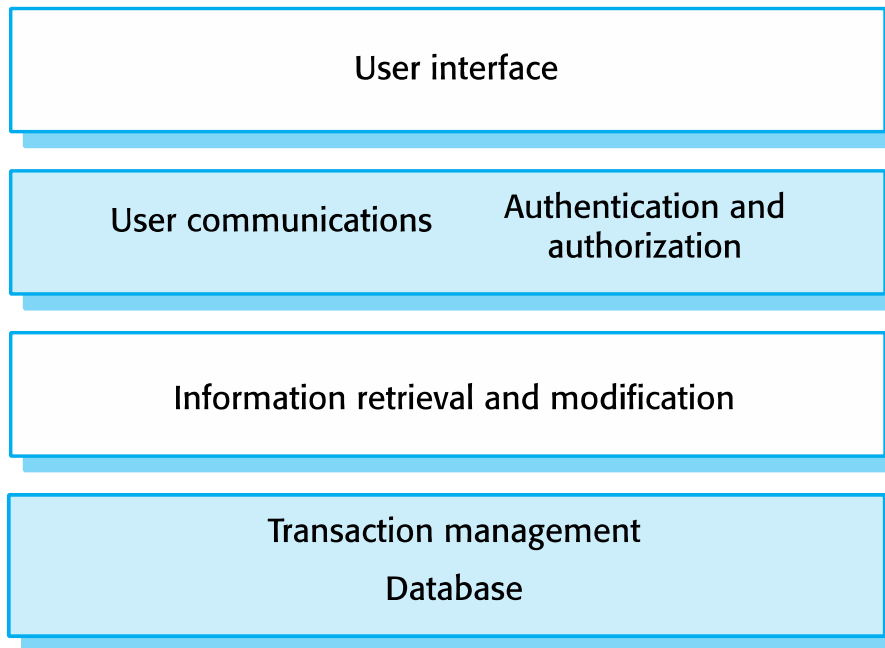
Example : an ATM System



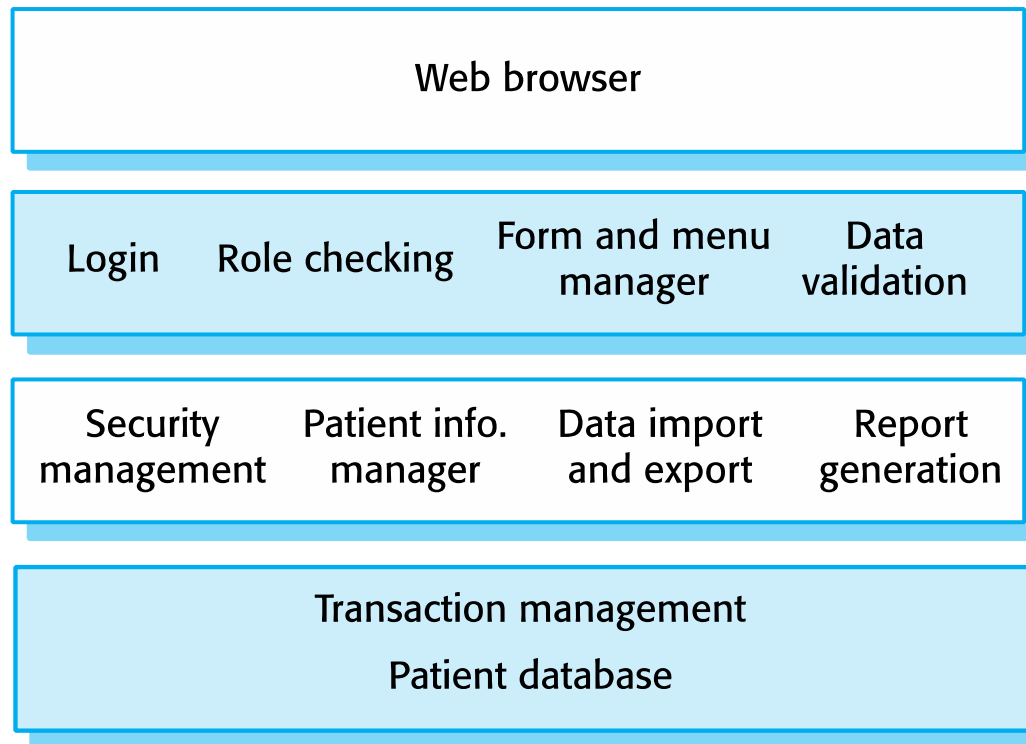
Information Systems Architecture

- Information systems have a generic architecture that can be organized as a **layered architecture**.
 - Also **transaction-based systems** as interaction with these systems generally involves database transactions.

- Layers often include
 - User interface
 - User communications
 - Information retrieval
 - System database



Example : the Mentcare System



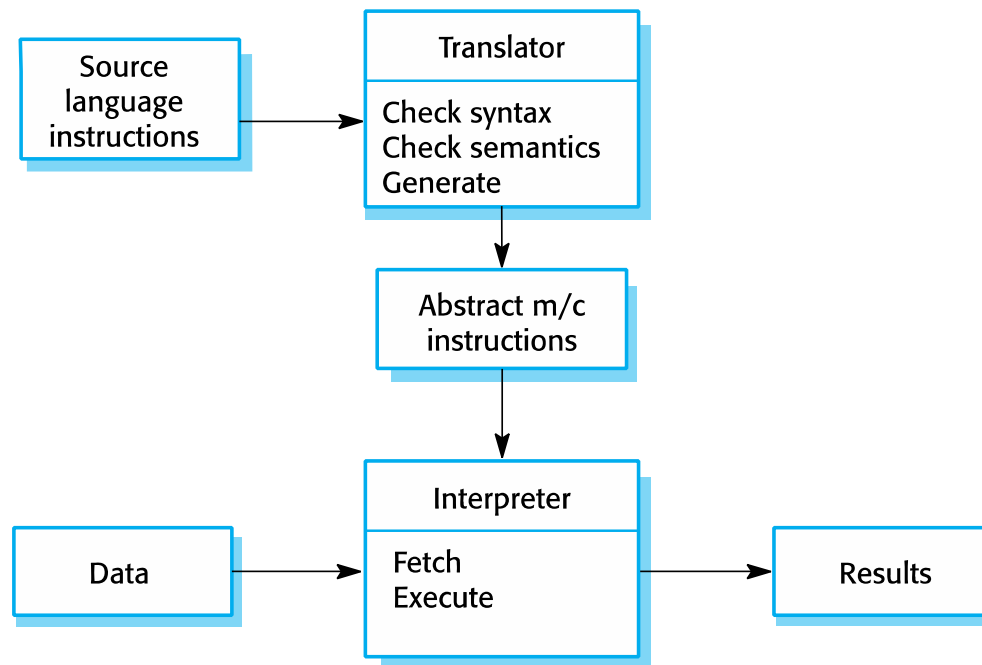
Web-Based Information Systems

- Web-based systems implement user interfaces using a web browser.
 - Example : **e-commerce systems** are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
 - The application-specific layer includes additional functionality supporting a ‘shopping cart’ in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

- Web-based information systems are often implemented as **multi-tier client server/architectures**.
 - Web server: Responsible for all user communications, with the user interface implemented using a web browser
 - Application server: Responsible for implementing application-specific logic as well as information storage and retrieval requests
 - Database server: Moves information to and from the database and handles transaction management

Language Processing Systems

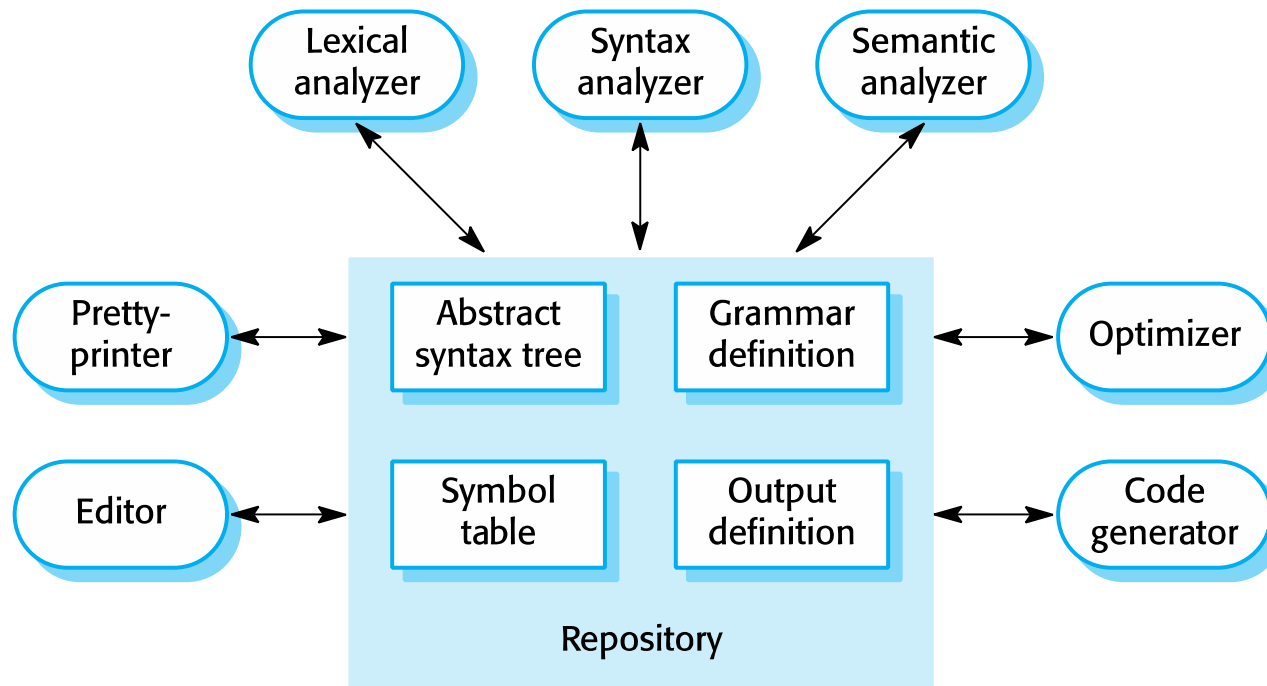
- **Language Processing Systems** accept a natural or artificial **language** as input and generate some other representation of that **language**.
 - May include an interpreter to act on the instructions in the language that is being processed
 - Meta-case tools process tool descriptions, method rules, etc and generate tools.



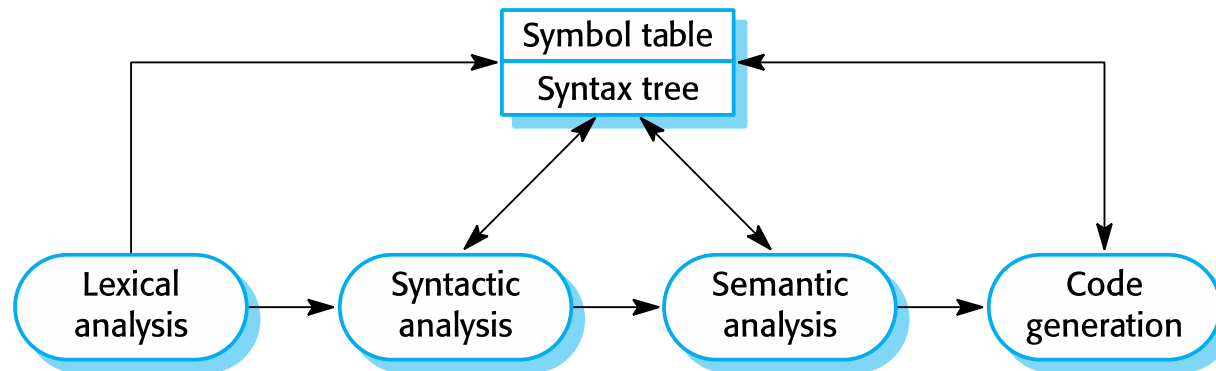
Compiler Components

- Compiler components for language processing systems
 - **Lexical analyzer** : Takes input language tokens and converts them to an internal form
 - **Symbol table** : Holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated
 - **Syntax analyzer** : Checks the syntax of the language being translated
 - **Syntax tree** : An internal structure representing the program being compiled
 - **Semantic analyzer** : Uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text
 - **Code generator** : ‘walks’ the syntax tree and generates abstract machine code

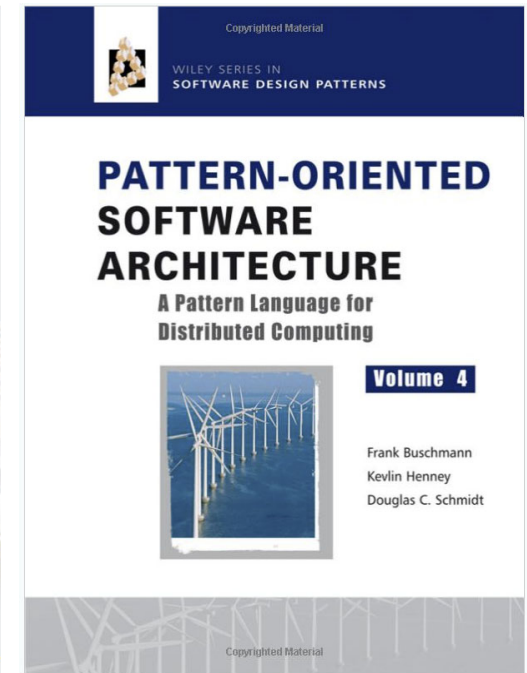
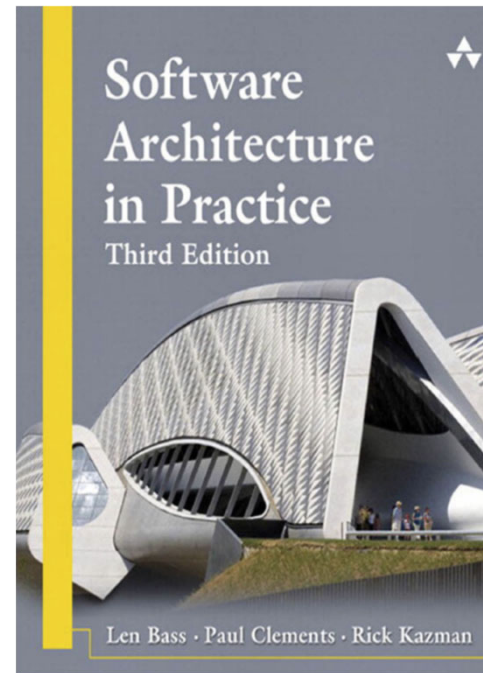
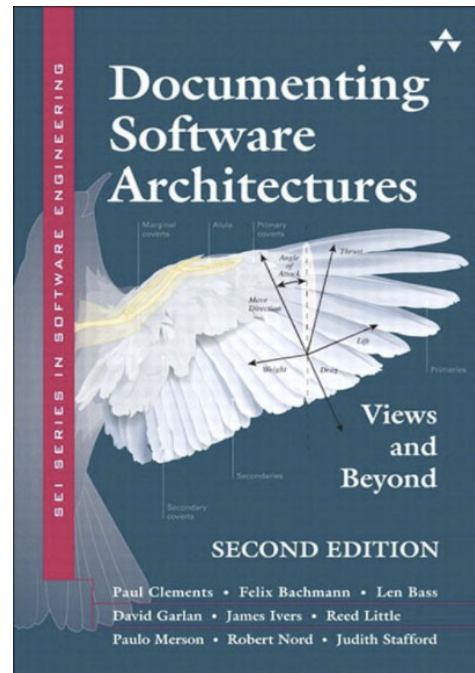
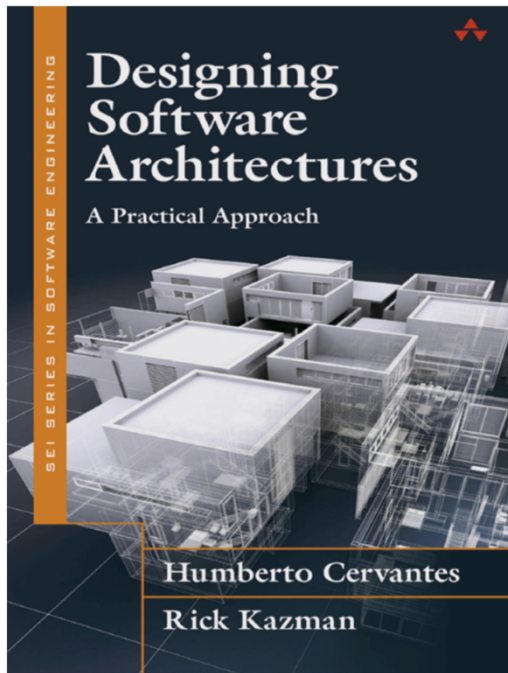
A Repository Architecture for a Language Processing System



A Pipe and Filter Architecture for Compilers



References for Architecture Design



Homework #8

- 기개발한 RVC Controller를 확장하려고 합니다.
 - 핸드폰 App에서 WiFi를 통해 RVC Control SW와 통신하는 기능을 추가하려고 합니다.
- Microsoft Application Architecture Guide 2.0을 공부한 후, 제시된 Application Architecture를 활용해서, 전체 시스템의 Overall Architecture를 하나 제안하세요.
 - 논리적인 설명을 포함하고, 실제 개발에 사용될 수 있을 정도로 자세히 작성하세요. (PPT 4장)



7. Design and Implementation

Design and Implementation

- **Software design and implementation**

- The stage at which an executable software system is developed

- Software design and implementation activities are often inter-leaved.

- **Software design** is a creative activity in which you identify software components and their relationships, based on a customer's requirements.

- **Implementation** is the process of realizing the design as a program.

Build or Buy

- It is possible to buy commercial off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
 - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.

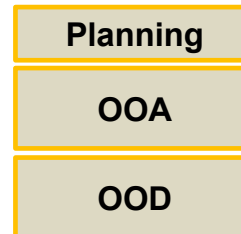
- The design process becomes concerned with **how to use the configuration features of that system** to deliver the system requirements.
 - It requires different ways to develop software.

Object-Oriented Design Using UML

An Object-Oriented Design Process

- **Structured object-oriented design processes** (such as **UP**)
 - Involve developing a number of **different system models**
 - For small systems,
 - Require a lot of effort for development and maintenance of these models, and may not be cost-effective
 - For large systems developed by different groups,
 - Design models are an important communication mechanism

- There are a variety of different object-oriented design processes.
 - **Common activities** in all OO design processes
 1. **Define the context and modes of use of the system**
 2. **Design the system architecture**
 3. **Identify the principal system objects**
 4. **Develop design models**
 5. **Specify object interfaces**



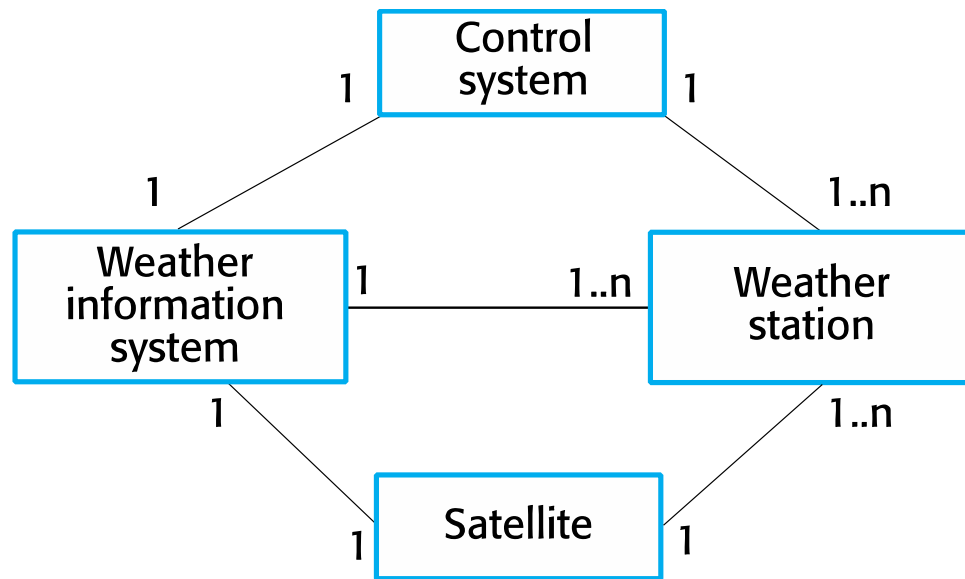
1. System Context and Interactions

- Understanding the **relationships** between the **software** that is being designed and its **external environment** is essential for deciding
 - Essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment
 - Let you establish the boundaries of the system

- **System context model**
 - A structural model that demonstrates the other systems in the environment of the system being developed
 - **System context diagram**

- **Interaction model**
 - A dynamic model that shows how the system interacts with its environment as it is used
 - **Use-case model**

System Context for the Weather Station

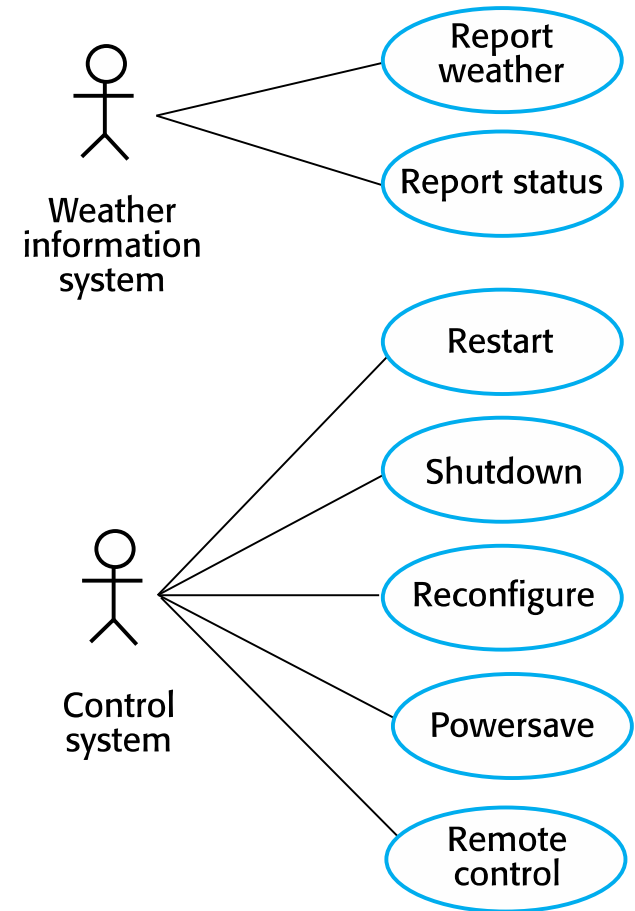


System context diagram in UML class diagram

Use-Case Model for the Weather Station

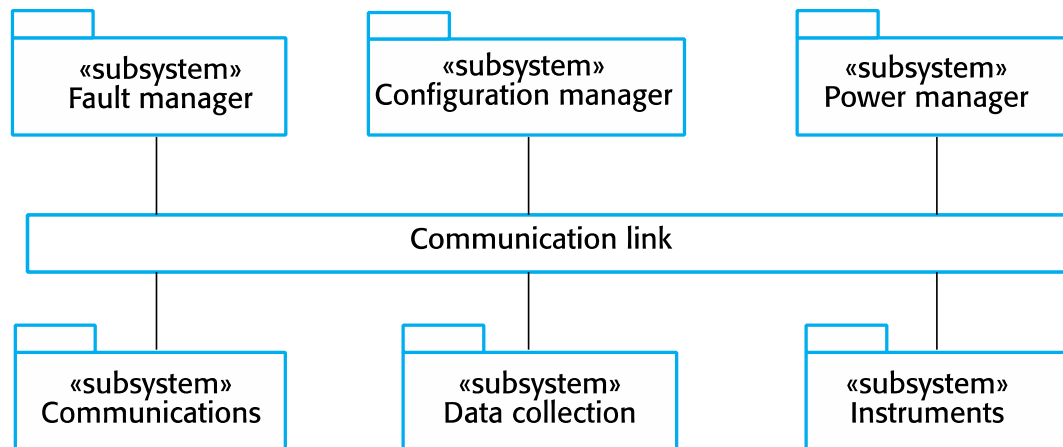
System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

Use-Case Model (Text + Diagram)

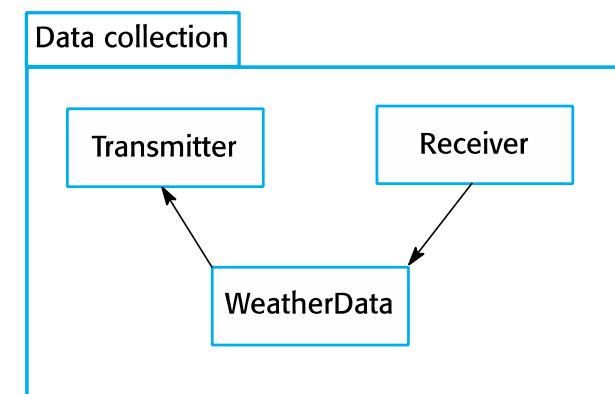


2. Architectural Design

- **Identify the major components** that make up the system and their interactions
 - Organize the components using an **architectural pattern** such as a layered or client-server model, if it needs
 - Example : The weather station is composed of independent subsystems that communicate by **broadcasting messages** on a **common infrastructure**.



High-Level Architecture of the Weather Station



The Architecture of Data Collection System

3. Object Class Identification

- **Identifying object classes** is a difficult part of object-oriented design.
 - There is no 'magic formula' for object identification.
 - It relies on the skill, experience and domain knowledge of system designers.

- **Object identification** is an iterative process.
 - **Domain Model**

- Approaches to object identification
 - Use a **grammatical approach** based on a natural language description of the system.
 - Based on identifying tangible things in the application domain
 - Use a behavioural approach.
 - Identify objects based on what participates in what behaviour
 - Use a **scenario-based analysis**. (Use-case analysis)
 - The objects, attributes, and methods in each scenario are identified

The Weather Station: Object Classes

- Object class identification in the weather station system may be based on the tangible hardware and data in the system.
 - Ground thermometer, Anemometer, Barometer
 - ‘Hardware’ objects related to the instruments in the system
 - Weather station
 - The basic interface of the weather station to its environment
 - It therefore reflects the interactions identified in the use-case model
 - Weather data
 - Encapsulates the summarized data from the instruments

Ground thermometer
gt_Ident temperature
get () test ()

Anemometer
an_Ident windSpeed windDirection
get () test ()

Barometer
bar_Ident pressure height
get () test ()

WeatherStation
identifier
reportWeather () reportStatus () powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

WeatherData
airTemperatures groundTemperatures windSpeeds windDirections pressures rainfall
collect () summarize ()



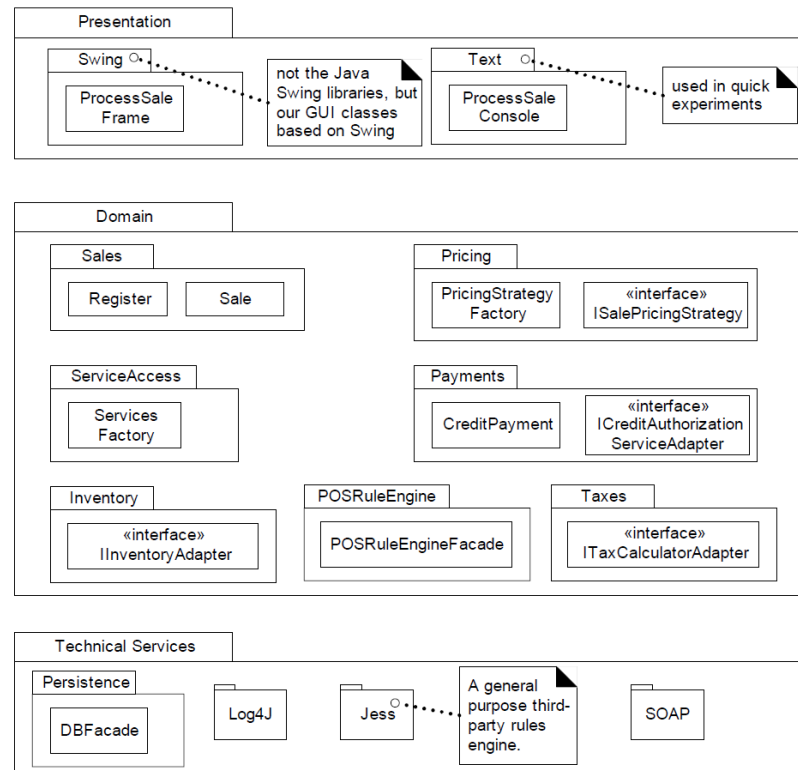
4. Design Models

- **Design models** show the **objects and object classes** and **relationships** between these entities.
- Two types of design models
 - **Structural (Static) model**
 - Describe the static structure of the system in terms of object classes and relationships
 - Class diagram, Object diagram, Package diagram
 - **Dynamic model**
 - Describe the dynamic interactions between objects
 - Sequence diagram, Communication diagram, Statechart diagram

Subsystem Models

- **Subsystem Models** shows how the design is organized into **logically related groups of objects**.
 - Logical model
 - The actual organization of objects in the system may be different.

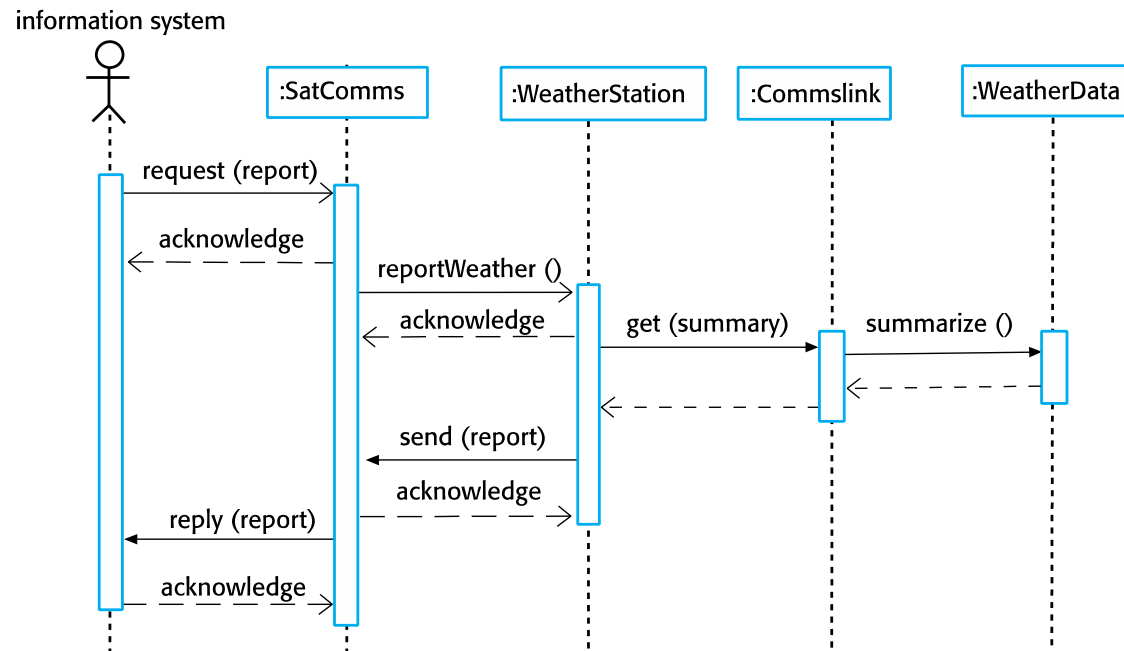
– **The UML package diagram** are often used.



Sequence Models

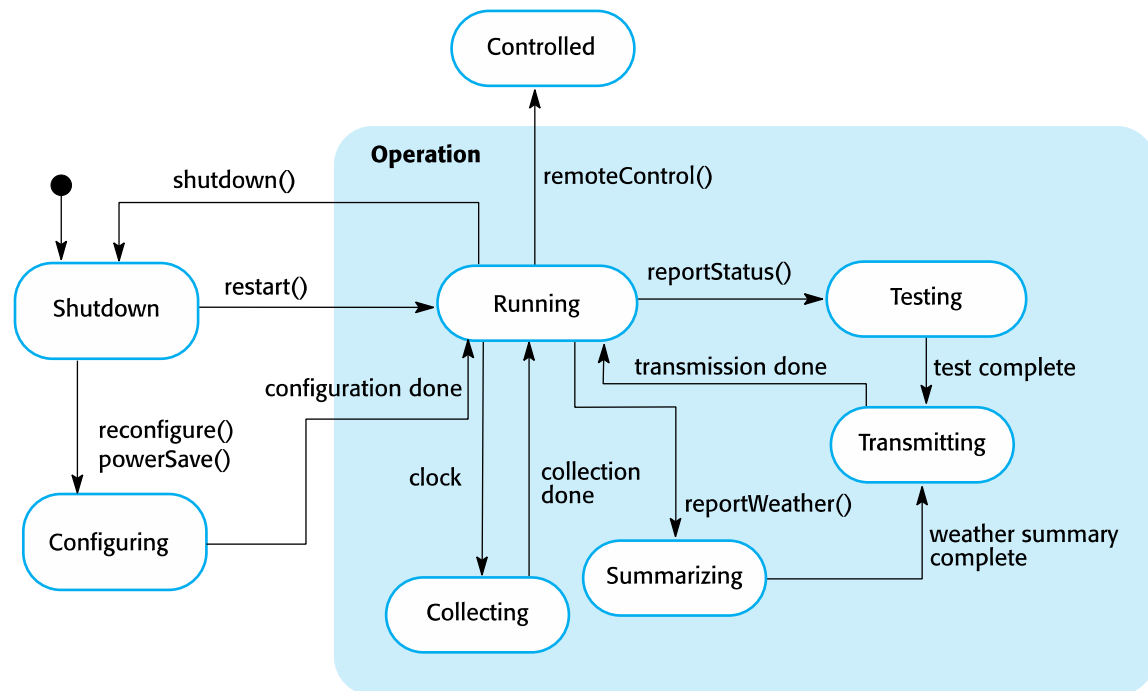
- **Sequence models** show the **sequence of object interactions** that take place.
 - **The UML Sequence diagrams** are used.
 - Objects are arranged horizontally across the top.
 - Time is represented vertically so models are read top to bottom.
 - Interactions are represented by labelled arrows.
 - Different styles of arrow represent different types of interaction.
 - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

- Example:
 - SD for Data Collection



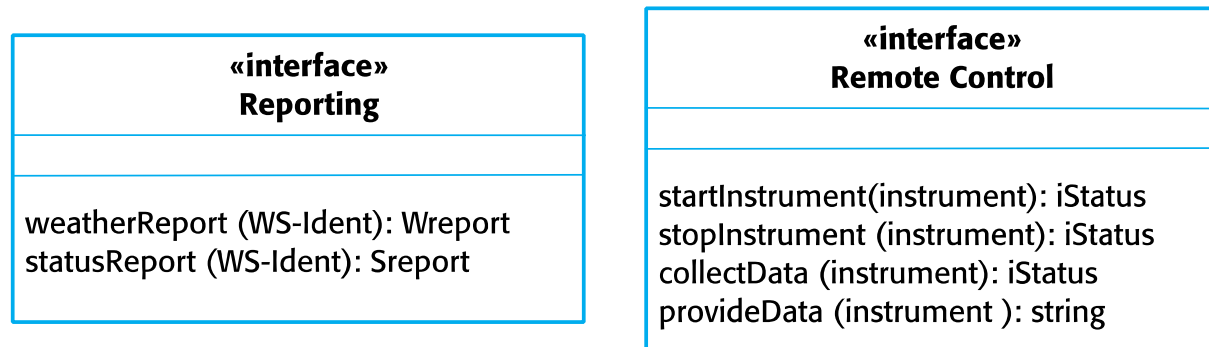
State Machine Models

- **State machine models** are used to show how objects respond to different service requests and the state transitions triggered by these requests.
 - State diagrams are useful high-level models of a **system** or an **object's** run-time behavior.
 - Not usually needed for all objects in the system.
 - **The UML Statecharts diagram** is used.
- Example
 - State diagram for Weather Station



5. Interface Specification

- **Object interfaces** have to be specified so that the objects and other components can be designed in parallel.
 - Objects may have several interfaces which are viewpoints on the methods provided.
 - **The UML Class diagram** is used.
 - Example
 - Interface specification (a part of class diagram) for Weather Station

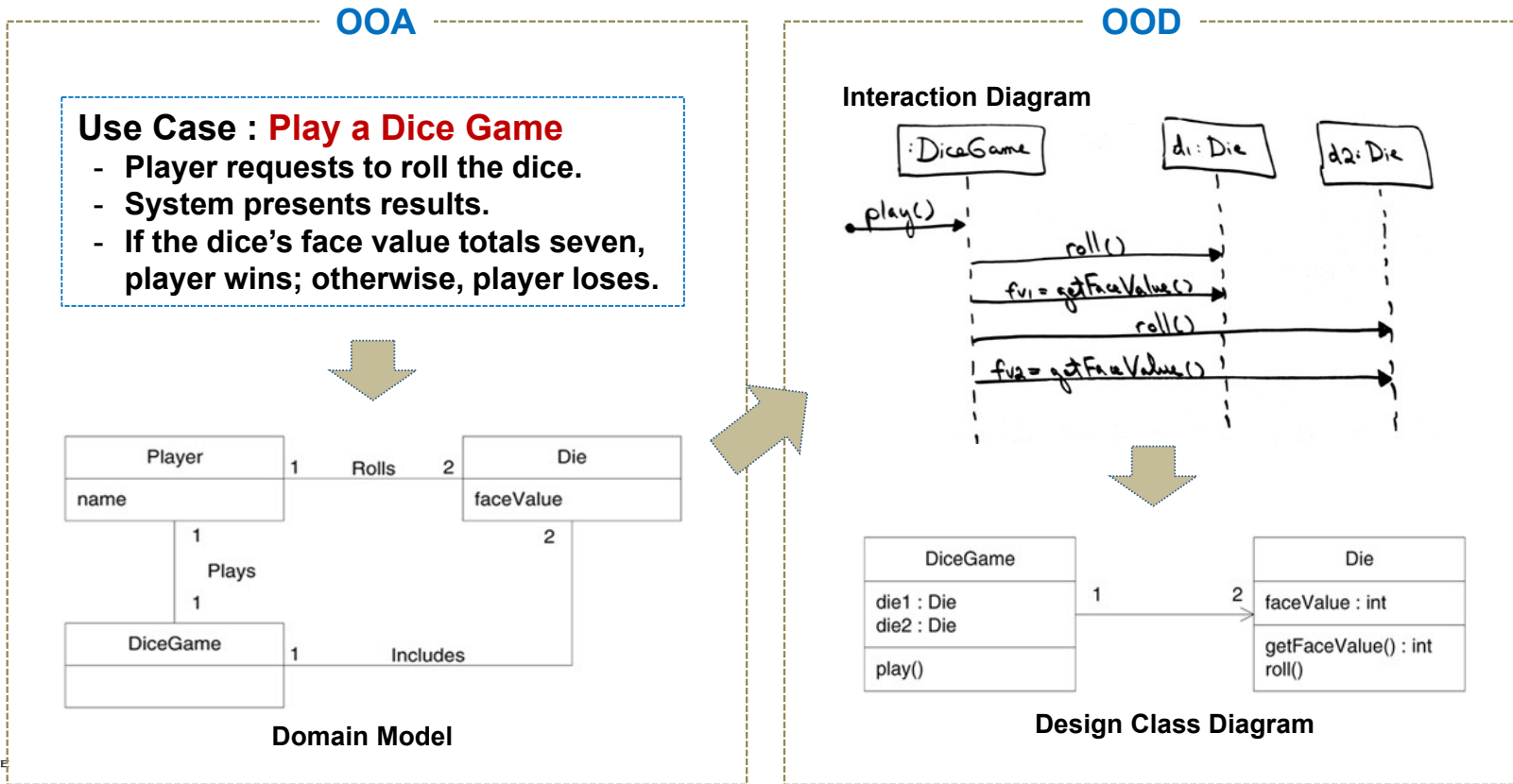
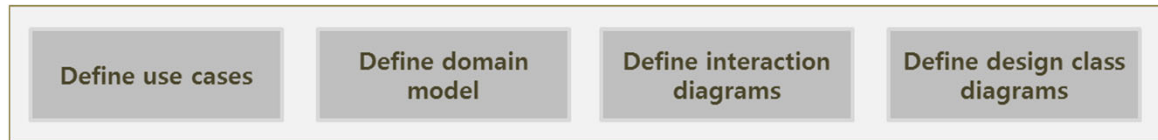


Weather Station Interfaces

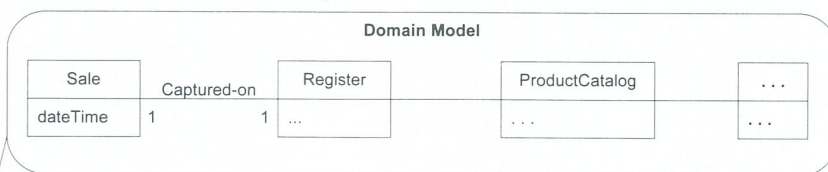
OOAD for Object-Oriented Programming

- **OOAD** (Object-Oriented Analysis and Design, AKA 객체지향개발방법론)
 - *“Identifying your requirements and creating a domain model, and then add methods to the appropriate classes and define the messaging between the objects in order to fulfill the requirements”*
 - **Object-Oriented Analysis (OOA)**
 - Discover the domain concepts/objects (**Domain Model**)
 - Identify requirements (**Use-Case Model**)
 - **Object-Oriented Design (OOD)**
 - Define software objects (Static model → **Class Diagram**)
 - Define how they collaborate to fulfill the requirements (Dynamic model → **Sequence Diagram**)
 - Various development process models are available.
 - **Waterfall**
 - **UP (Iterative)**

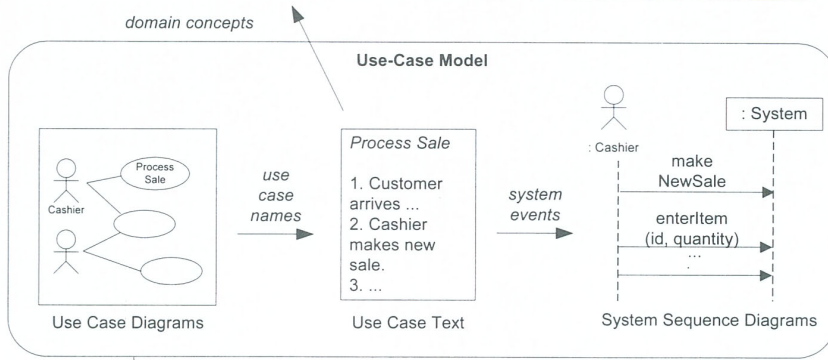
An OOAD Example - Dice Game



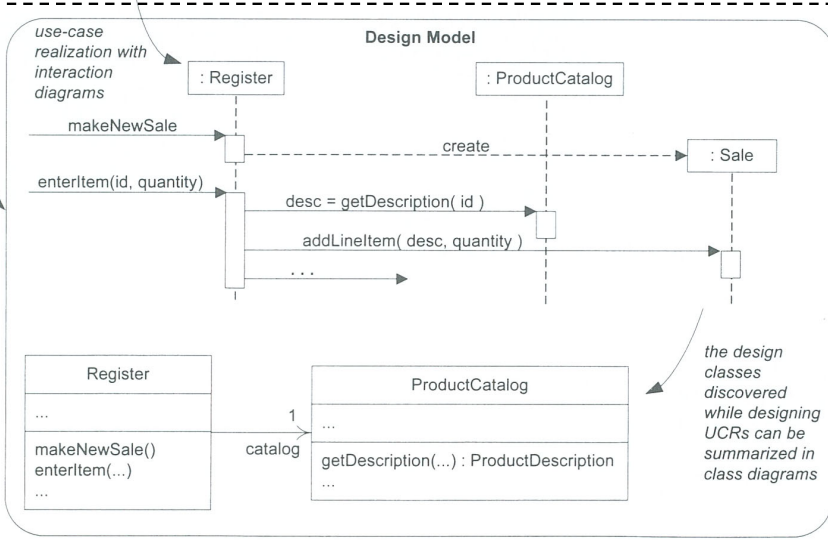
Sample Unified Process Artifact Relationships



Domain Model



Use-Case Model



Sequence Diagrams

Class Diagram

conceptual classes in the domain inspire the names of some software classes in the design

OOA
OOD

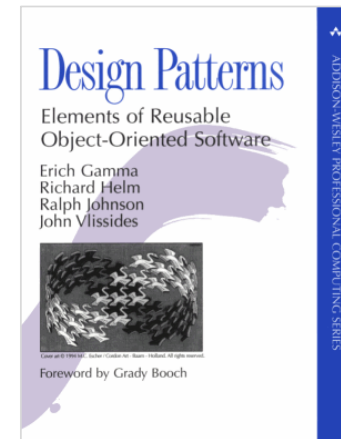
OO Implementation

Design Patterns

Design Patterns

- **Design pattern** is a way to describe best practices, good designs, and capture experience in a way that it is possible for others to **reuse** this experience.
 - Descriptions of the **problem** and the essence of its **solution**
 - Sufficiently abstract to be reused in different settings
 - Pattern descriptions usually make use of **object-oriented characteristics**
 - Inheritance and Polymorphism
 - **23 design patterns of GoF** are widely used.
- Elements of patterns

Element	Description
Name	A meaningful pattern identifier
Problem description	A detailed description on the problem
Solution description	Not a concrete design, but a template for a design solution that can be instantiated in different ways.
Consequences	The results and trade-offs of applying the pattern



23 Design Patterns of GoF

- C Abstract Factory
- S Adapter
- S Bridge
- C Builder
- B Chain of Responsibility
- B Command
- S Composite
- S Decorator
- S Facade
- C Factory Method
- S Flyweight
- B Interpreter
- B Iterator
- B Mediator
- S Memento
- C Prototype
- S Proxy
- B Observer
- C Singleton
- B State
- B Strategy
- B Template Method
- B Visitor

Chain of Responsibility

Type: Behavioral

What it is: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until one object handles it.

```

classDiagram
    class Client
    class Handler {
        <<interface>>
        +handleRequest()
    }
    class ConcreteHandler1 {
        +handleRequest()
    }
    class ConcreteHandler2 {
        +handleRequest()
    }
    Client --> Handler
    Handler <|-- ConcreteHandler1
    Handler <|-- ConcreteHandler2
    ConcreteHandler1 --> Handler : successor
    ConcreteHandler2 --> Handler : successor
    
```

Command

Type: Behavioral

What it is: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

```

classDiagram
    class Client
    class Invoker {
        +execute()
    }
    class ConcreteCommand {
        +execute()
    }
    class Receiver {
        +action()
    }
    class Command {
        +execute()
    }
    Client --> Invoker
    Invoker --> ConcreteCommand
    Invoker --> Command
    ConcreteCommand --> Receiver
    ConcreteCommand --> Command
    
```

Interpreter

Type: Behavioral

What it is: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

```

classDiagram
    class Client
    class Context
    class AbstractExpression {
        <<interface>>
        +interpret()
    }
    class TerminalExpression {
        +interpret()
    }
    class NonterminalExpression {
        +interpret()
    }
    Client --> Context
    Context --> AbstractExpression
    Context --> TerminalExpression
    Context --> NonterminalExpression
    AbstractExpression <|-- TerminalExpression
    AbstractExpression <|-- NonterminalExpression
    
```

Iterator

Type: Behavioral

What it is: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

```

classDiagram
    class Client
    class Aggregate {
        <<interface>>
        +createIterator()
    }
    class Iterator {
        <<interface>>
        +next()
    }
    class ConcreteAggregate {
        +createIterator()
    }
    class ConcreteIterator {
        +next()
    }
    Client --> Aggregate
    Client --> Iterator
    Aggregate <|-- ConcreteAggregate
    Iterator <|-- ConcreteIterator
    
```

Mediator

Type: Behavioral

What it is: Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and lets you vary their interactions independently.

```

classDiagram
    class Mediator {
        <<interface>>
        +Colleague
    }
    class ConcreteMediator
    class Colleague {
        <<interface>>
        +ConcreteColleague
    }
    class ConcreteColleague
    Mediator <|-- ConcreteMediator
    Colleague <|-- ConcreteColleague
    ConcreteMediator --> ConcreteColleague
    ConcreteColleague --> Mediator
    
```

Memento

Type: Behavioral

What it is: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

```

classDiagram
    class Caretaker {
        +state
    }
    class Originator {
        +state
        +setMemento(m: Memento)
        +createMemento()
    }
    class Memento {
        +state
    }
    Caretaker --> Memento
    Originator --> Memento
    
```

Observer

Type: Behavioral

What it is: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

```

classDiagram
    class Subject {
        <<interface>>
        +attach(o: Observer)
        +detach(o: Observer)
        +notify()
    }
    class Observer {
        +update()
    }
    class ConcreteSubject {
        +subjectState
    }
    class ConcreteObserver {
        +observerState
    }
    Subject <|-- ConcreteSubject
    Observer <|-- ConcreteObserver
    ConcreteSubject --> ConcreteObserver : observes
    
```

State

Type: Behavioral

What it is: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

```

classDiagram
    class Context {
        +request()
    }
    class State {
        <<interface>>
        +handle()
    }
    class ConcreteState1 {
        +handle()
    }
    class ConcreteState2 {
        +handle()
    }
    Context --> State
    State <|-- ConcreteState1
    State <|-- ConcreteState2
    
```

Strategy

Type: Behavioral

What it is: Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.

```

classDiagram
    class Context {
        +execute()
    }
    class Strategy {
        <<interface>>
        +execute()
    }
    class ConcreteStrategyA {
        +execute()
    }
    class ConcreteStrategyB {
        +execute()
    }
    Context --> Strategy
    Strategy <|-- ConcreteStrategyA
    Strategy <|-- ConcreteStrategyB
    
```

Template Method

Type: Behavioral

What it is: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

```

classDiagram
    class AbstractClass {
        +TemplateMethod()
        +SubMethod()
    }
    class ConcreteClass {
        +SubMethod()
    }
    AbstractClass <|-- ConcreteClass
    
```

Visitor

Type: Behavioral

What it is: Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

```

classDiagram
    class Visitor {
        <<interface>>
        +visitElementA(e: ConcreteElementA)
        +visitElementB(e: ConcreteElementB)
    }
    class Element {
        <<interface>>
        +accept(v: Visitor)
    }
    class ConcreteVisitor {
        +visitElementA(e: ConcreteElementA)
        +visitElementB(e: ConcreteElementB)
    }
    class ConcreteElementA {
        +accept(v: Visitor)
    }
    class ConcreteElementB {
        +accept(v: Visitor)
    }
    Visitor <|-- ConcreteVisitor
    Element <|-- ConcreteElementA
    Element <|-- ConcreteElementB
    ConcreteVisitor --> ConcreteElementA
    ConcreteVisitor --> ConcreteElementB
    
```

Adapter

Type: Structural

What it is: Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

```

classDiagram
    class Client {
        +operation()
    }
    class Adapter {
        <<interface>>
        +operation()
    }
    class ConcreteAdapter {
        +adaptee
        +adaptedOperation()
    }
    class Adaptee {
        +operation()
    }
    Client --> Adapter
    Adapter <|-- ConcreteAdapter
    ConcreteAdapter --> Adaptee
    
```

Bridge

Type: Structural

What it is: Decouple an abstraction from its implementation so that the two can vary independently.

```

classDiagram
    class Abstraction {
        +operation()
    }
    class Implementor {
        <<interface>>
        +operationImpl()
    }
    class ConcreteImplementorA {
        +operationImpl()
    }
    class ConcreteImplementorB {
        +operationImpl()
    }
    Abstraction <|-- Implementor
    Implementor <|-- ConcreteImplementorA
    Implementor <|-- ConcreteImplementorB
    
```

Composite

Type: Structural

What it is: Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

```

classDiagram
    class Component {
        <<interface>>
        +operation()
        +addIn(c: Composite)
        +remove(c: Composite)
        +getChild(n: int)
    }
    class Composite {
        +operation()
        +addIn(c: Composite)
        +remove(c: Composite)
        +getChild(n: int)
    }
    class Leaf {
        +operation()
    }
    Component <|-- Composite
    Component <|-- Leaf
    Composite --> Component : children
    
```

Decorator

Type: Structural

What it is: Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

```

classDiagram
    class Component {
        <<interface>>
        +operation()
    }
    class ConcreteComponent {
        +operation()
    }
    class Decorator {
        +operation()
    }
    class ConcreteDecorator {
        +addedState
        +operation()
        +addedBehavior()
    }
    Component <|-- ConcreteComponent
    Component <|-- Decorator
    Decorator <|-- ConcreteDecorator
    
```

Facade

Type: Structural

What it is: Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

```

classDiagram
    class Facade
    class ComplexSystem
    Facade --> ComplexSystem
    
```

Flyweight

Type: Structural

What it is: Use sharing to support large numbers of fine grained objects efficiently.

```

classDiagram
    class FlyweightFactory {
        +getFlyweight(key)
    }
    class Flyweight {
        <<interface>>
        +operation(in extrinsicState)
    }
    class ConcreteFlyweight {
        +intrinsicState
        +operation(in extrinsicState)
    }
    class UnsharedConcreteFlyweight {
        +allState
        +operation(in extrinsicState)
    }
    FlyweightFactory --> Flyweight
    Flyweight <|-- ConcreteFlyweight
    Flyweight <|-- UnsharedConcreteFlyweight
    
```

Proxy

Type: Structural

What it is: Provide a surrogate or placeholder for another object to control access to it.

```

classDiagram
    class Client
    class Subject {
        <<interface>>
        +request()
    }
    class RealSubject {
        +request()
    }
    class Proxy {
        +request()
    }
    Client --> Subject
    Subject <|-- RealSubject
    Subject <|-- Proxy
    Proxy --> RealSubject : represents
    
```

Abstract Factory

Type: Creational

What it is: Provides an interface for creating families of related or dependent objects without specifying their concrete class.

```

classDiagram
    class Client
    class AbstractFactory {
        <<interface>>
        +createProductA()
        +createProductB()
    }
    class ConcreteFactory {
        +createProductA()
        +createProductB()
    }
    class AbstractProduct {
        <<interface>>
    }
    class ConcreteProduct {
    }
    Client --> AbstractFactory
    AbstractFactory <|-- ConcreteFactory
    AbstractProduct <|-- ConcreteProduct
    
```

Builder

Type: Creational

What it is: Separate the construction of a complex object from its representing so that the same construction process can create different representations.

```

classDiagram
    class Director {
        +construct()
    }
    class Builder {
        <<interface>>
        +buildPart()
    }
    class ConcreteBuilder {
        +buildPart()
        +getResult()
    }
    Director --> Builder
    Builder <|-- ConcreteBuilder
    
```

Factory Method

Type: Creational

What it is: Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

```

classDiagram
    class Product {
        <<interface>>
    }
    class Creator {
        +factoryMethod()
        +anOperation()
    }
    class ConcreteProduct {
    }
    class ConcreteCreator {
        +factoryMethod()
    }
    Product <|-- ConcreteProduct
    Creator <|-- ConcreteCreator
    ConcreteCreator --> ConcreteProduct
    
```

Prototype

Type: Creational

What it is: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

```

classDiagram
    class Client
    class Prototype {
        <<interface>>
        +clone()
    }
    class ConcretePrototype1 {
        +clone()
    }
    class ConcretePrototype2 {
        +clone()
    }
    Client --> Prototype
    Prototype <|-- ConcretePrototype1
    Prototype <|-- ConcretePrototype2
    
```

Singleton

Type: Creational

What it is: Ensure a class only has one instance and provide a global point of access to it.

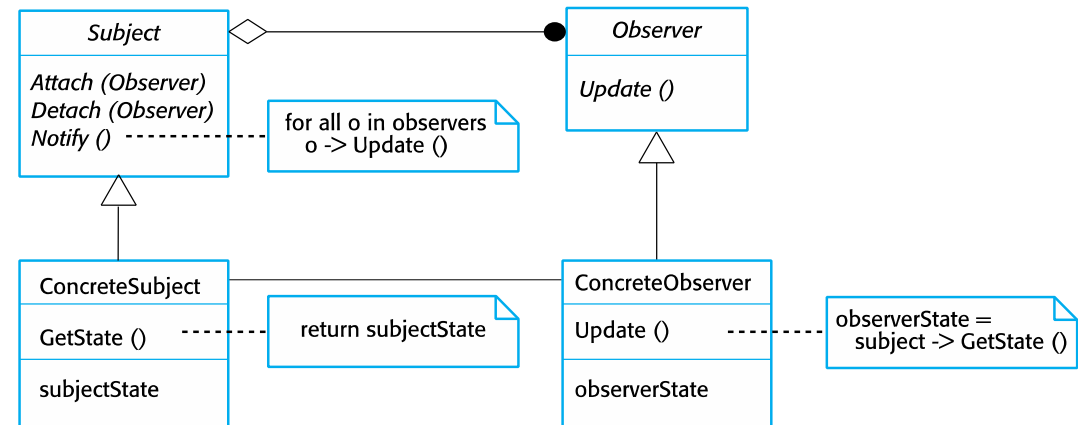
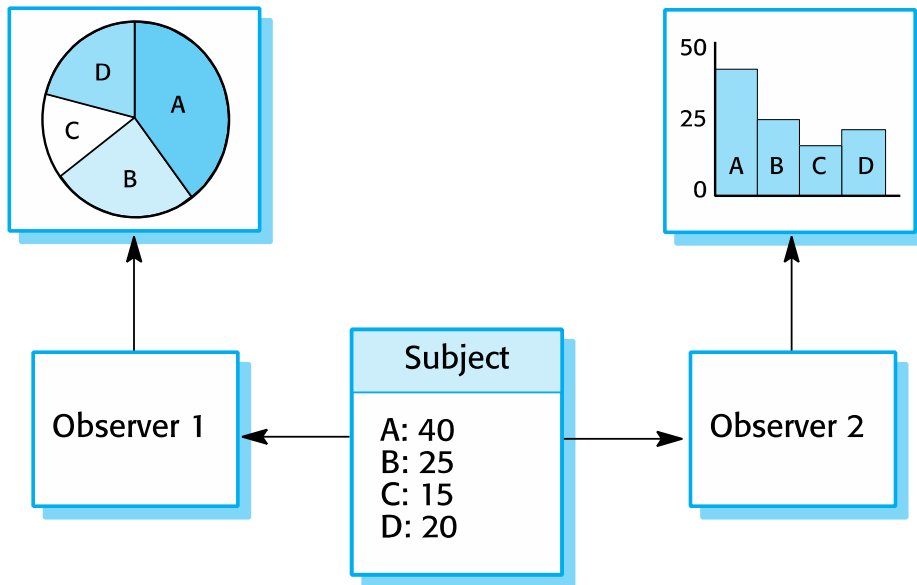
```

classDiagram
    class Singleton {
        +static uniqueInstance
        +singletonData
        +static Instance()
        +SingletonOperation()
    }
    
```


The Observer Pattern

Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore, there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Multiple Displays Using the Observer Pattern



Implementation Issues

Implementation Issues

- **Implementation issues** that are often not covered in programming
 - **Reuse**
 - Most modern software is constructed by reusing existing components or systems.
 - When you are developing software, you should make as much use as possible of existing code.
 - **Configuration management**
 - During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
 - **Host-target development**
 - Production software does not usually execute on the same computer as the software development environment.
 - Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

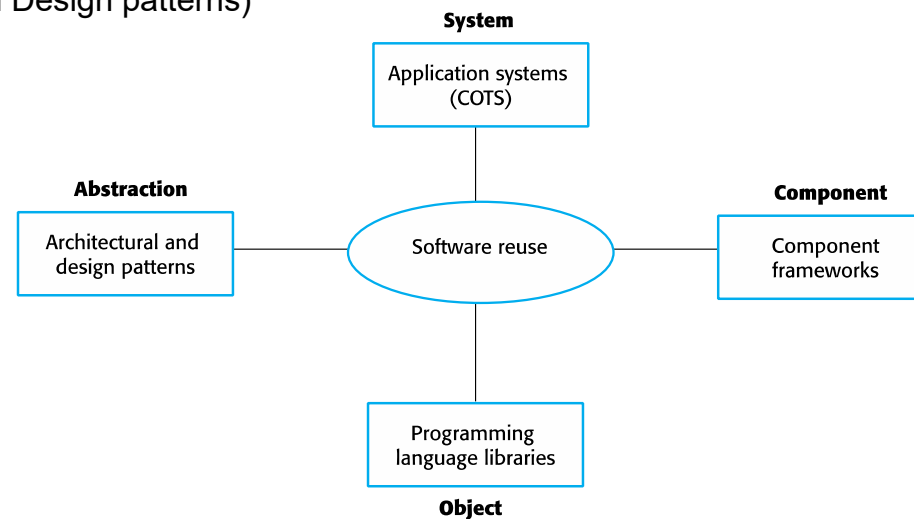
Reuse

- A development approach based on **the reuse of existing software**
 - Until 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
 - Only the reuse of **functions** and **objects** in **programming language libraries**

- **Reuse costs**
 - The costs of **the time spent in looking for** software to reuse and **assessing** whether it meets your needs
 - The costs of **adapting and configuring** the reusable software components or systems to reflect the requirements of the system that you are developing
 - The costs of **integrating reusable software elements** with each other and with the new code that you have developed

Reuse Levels

- Reuse levels
 - **The object level**
 - We directly reuse objects from a library rather than writing the code. (Programming language libraries)
 - **The component level**
 - Components are collections of objects and object classes that we reuse in application systems. (Component frameworks)
 - **The system level**
 - We reuse entire application systems. (COTS)
 - **The abstraction level**
 - We don't reuse software directly but use knowledge of successful abstractions in the design of our software. (like Architecture styles and Design patterns)



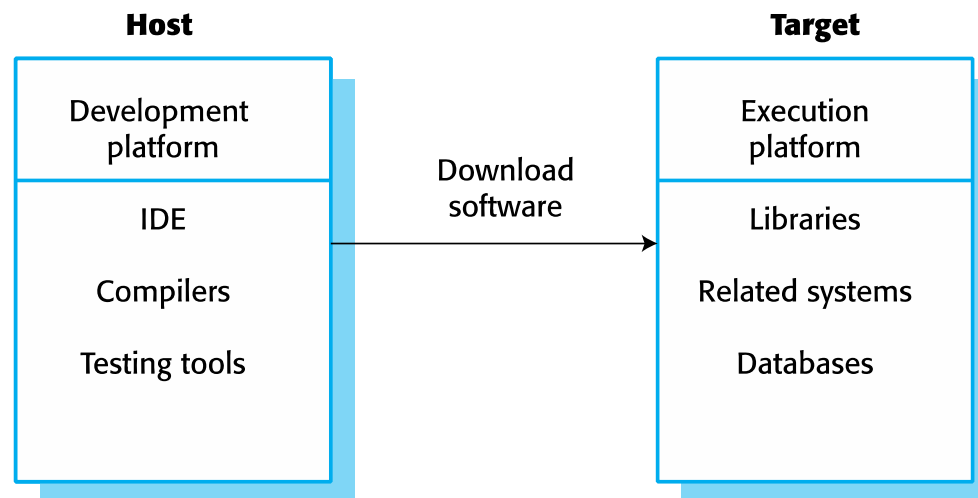
Configuration Management

- **Configuration management** is the general process of managing a changing software system.

- Configuration management activities:
 - **Version management**
 - Keep track of the different versions of software components
 - Include facilities to coordinate development by several programmers
 - **System integration**
 - Help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
 - **Problem tracking**
 - Allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed

Host-Target Development

- Most software is **developed on a computer (the host)** but **runs on a separate machine (the target)**.
 - **Development platform** vs. **Execution platform**
 - A platform is more than just hardware.
 - Includes the installed operating system and other supporting software such as database management systems or, interactive development (environments for development platforms)
 - Development platform usually has different installed software than execution platform.
 - May have different architectures



Tools for Host-Target Development

- **Tools for development platforms**

- Integrated compiler and syntax-directed editing system: create, edit and compile code
- Language debugging system
- Graphical editing tools (UML tools)
- Testing tools (JUnit) that can automatically run a set of tests on a new version of a program.
- Project support tools: organize codes for different development projects

- **IDE (Integrated Development Environments)**

- **A set of software tools** that supports different aspects of software development, within some common framework and user interface
- IDEs are created to support development in a specific programming language such as Java.

Open-Source Development

Open-Source Development

- **Open-source development** is an approach to software development in which
 - the source code of a software system is published, and volunteers are invited to participate in the development process through internet.
 - Rooted on the **Free Software Foundation (www.fsf.org)**
 - Advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish

- Popular examples of open-source systems
 - The Linux operating system
 - Java
 - The Apache web server
 - The mySQL database management system

Open-Source Issues

- **Questions** on open-sources :
 - *“Should the product that is being developed make use of open-source components?”*
 - *“Should we use an open-source approach for the software’s development?”*

- **Business** with opens source
 - More and more product companies are using an open-source approach to development.
 - Business model is not reliant on selling a software product but on selling support for that product.
 - Believe that involving the open-source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

Open-Source Licensing

- Fundamental principle of open-source
 - *“Source code should be freely available.”*

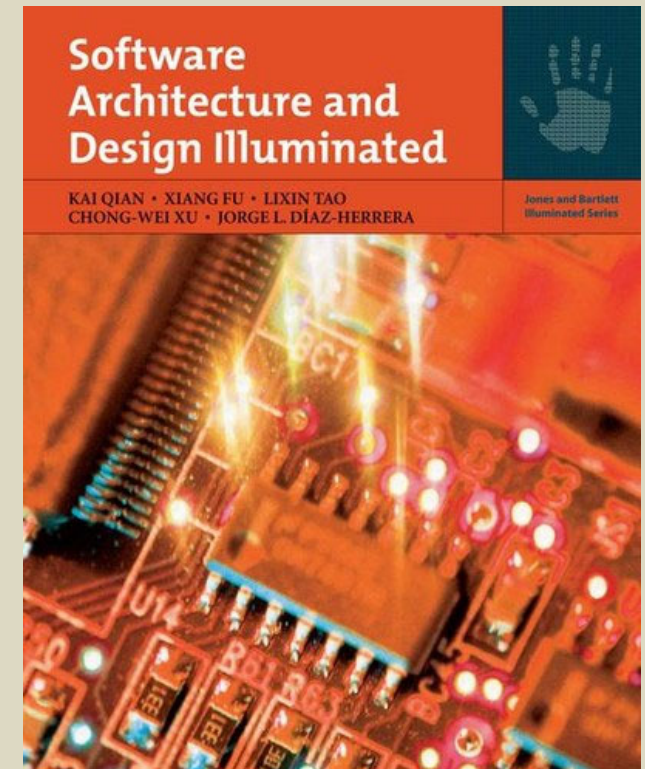
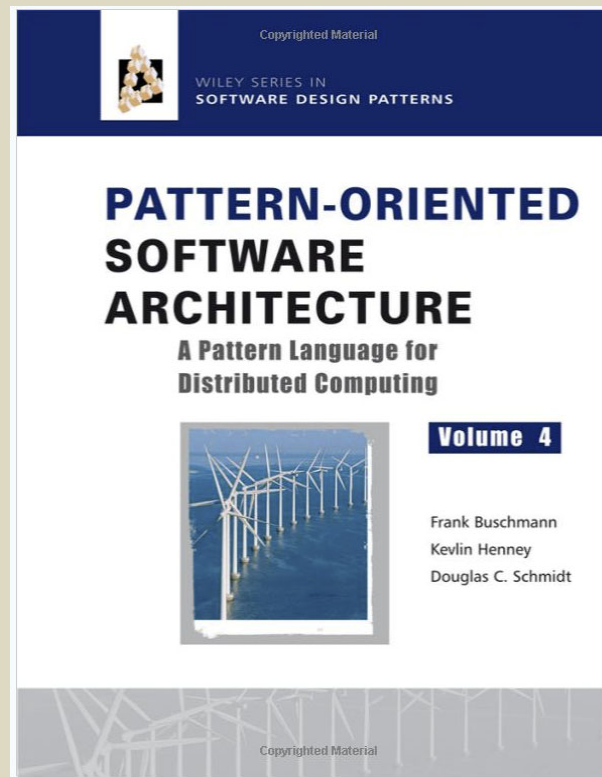
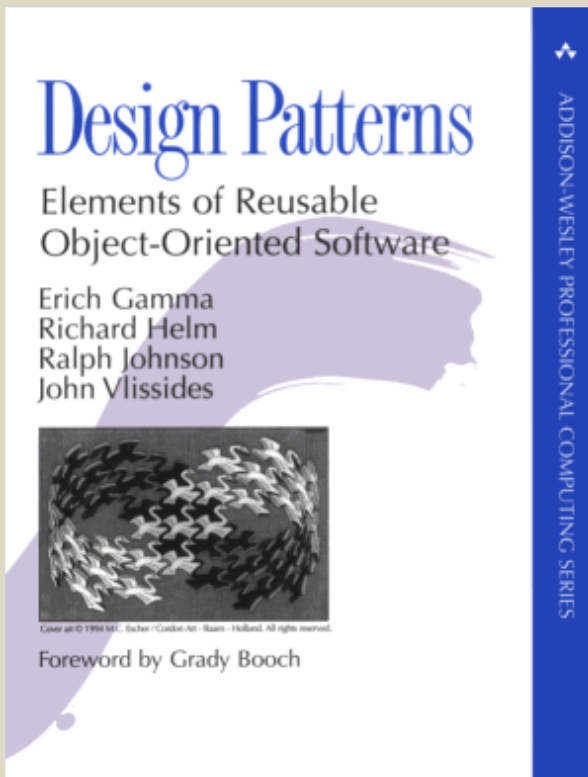
- License Models
 - **The GNU General Public License (GPL)**
 - So-called ‘reciprocal’ license
 - If you use open-source software that is licensed under the GPL license, then you must make that software open source.

 - **The GNU Lesser General Public License (LGPL)**
 - A variant of the GPL license
 - You can write components that link to open-source code without having to publish the source of these components.

 - **The Berkley Standard Distribution (BSD) License**
 - Non-reciprocal license
 - You are not obliged to re-publish any changes or modifications made to open-source code.
 - You can include the code in proprietary systems that are sold

Homework #9

- Design Pattern과 Architecture Style를 조사하고 비교분석 하세요. A4 10장 (글자크기 10 이하)
 - 아래의 기본 교재를 다 읽을 필요는 없습니다.



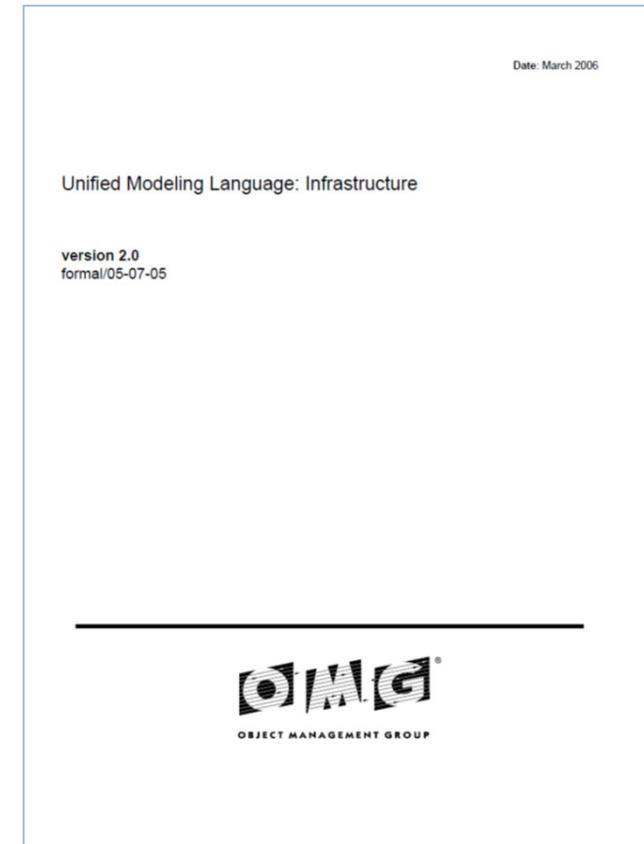
An Introduction to UML

UML



- **Unified Modeling Language** for
 - Visualizing, Specifying, Constructing and Documenting artifacts of software-intensive systems.
- Offer vocabulary and rules for **communication**
 - <http://www.uml.org/>
- Combine the best of the best from
 - Data Modeling (Entity Relationship Diagrams)
 - Business Modeling (workflow)
 - Object Modeling
 - Component Modeling (development and reuse - middleware, COTS)

de facto industry standard



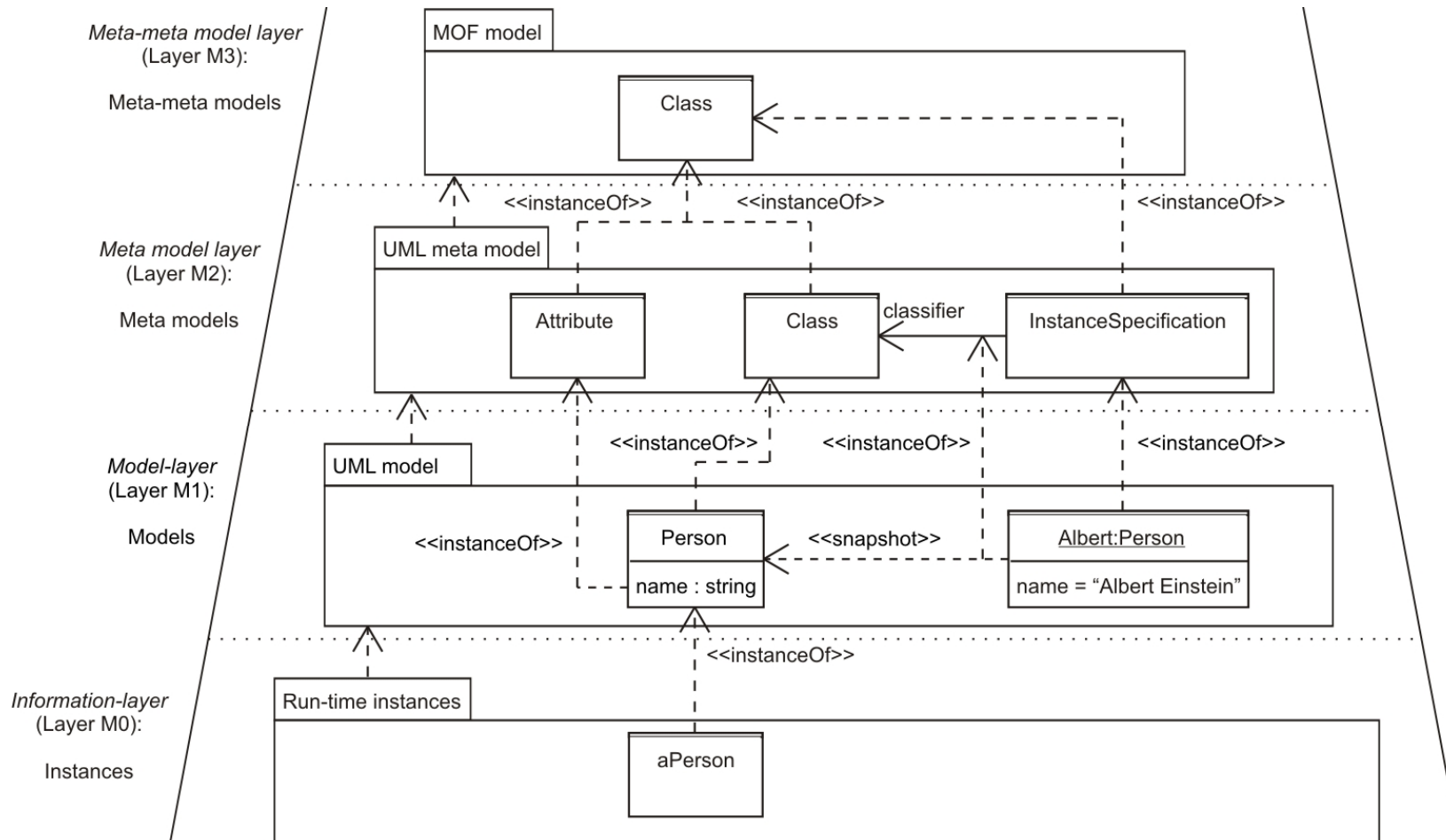
The UML Semantics

- **4-layer metamodel architecture**
 - instance → model → meta model → meta-meta model

- **MOF (Meta Object Facility)** defines a four-layer meta model hierarchy.
 - Layer M3: Meta-meta model layer (**The MOF model**)
 - Layer M2: Meta model layer (**The UML meta model**)
 - Layer M1: Model layer (**The UML model**)
 - Layer M0: Information layer (**the Application**)

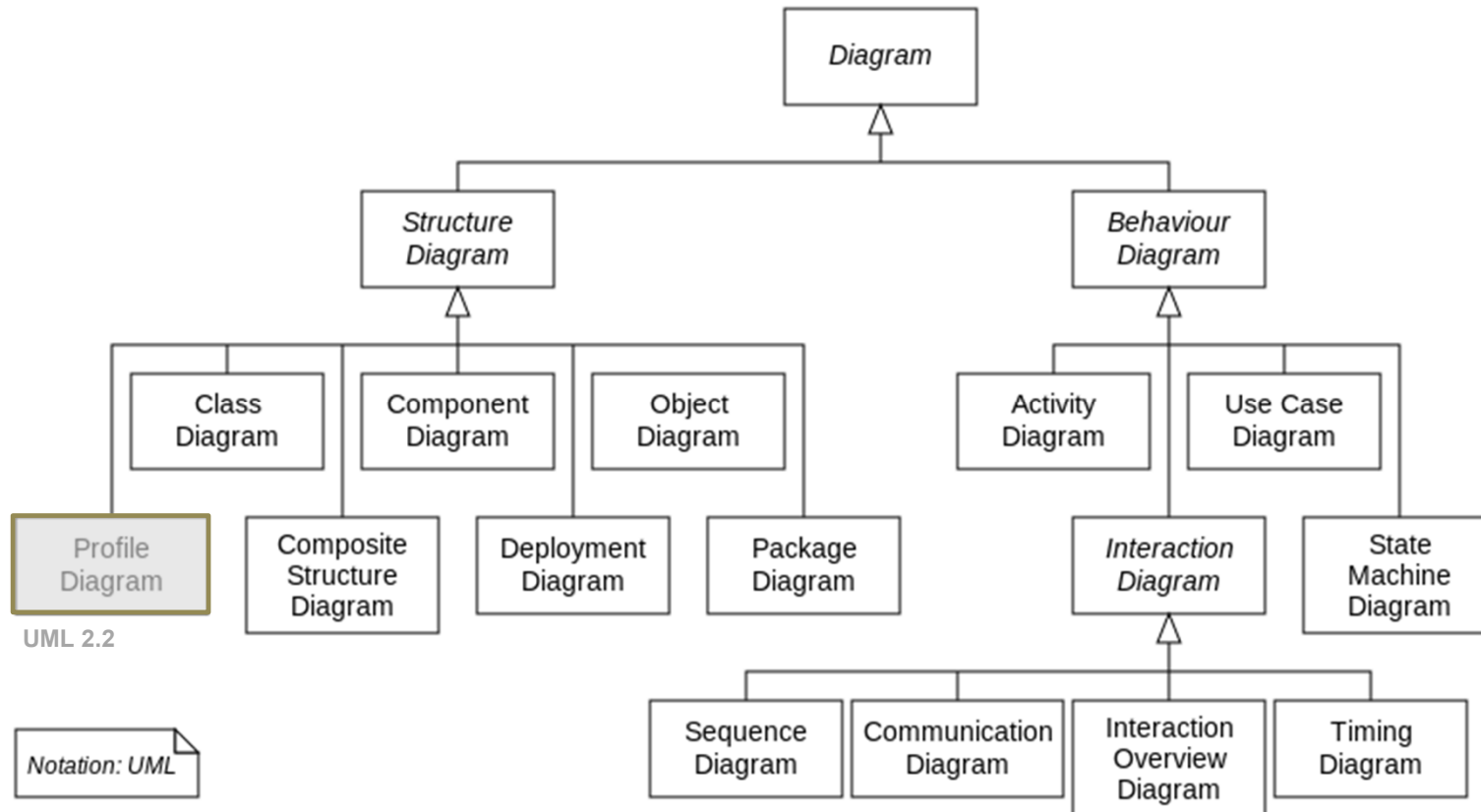
- **MOF and UML are aligned.**
 - The UML infrastructure contains all the concepts needed for the specification of UML and MOF.

The Meta Model Hierarchy of the MOF (for UML)



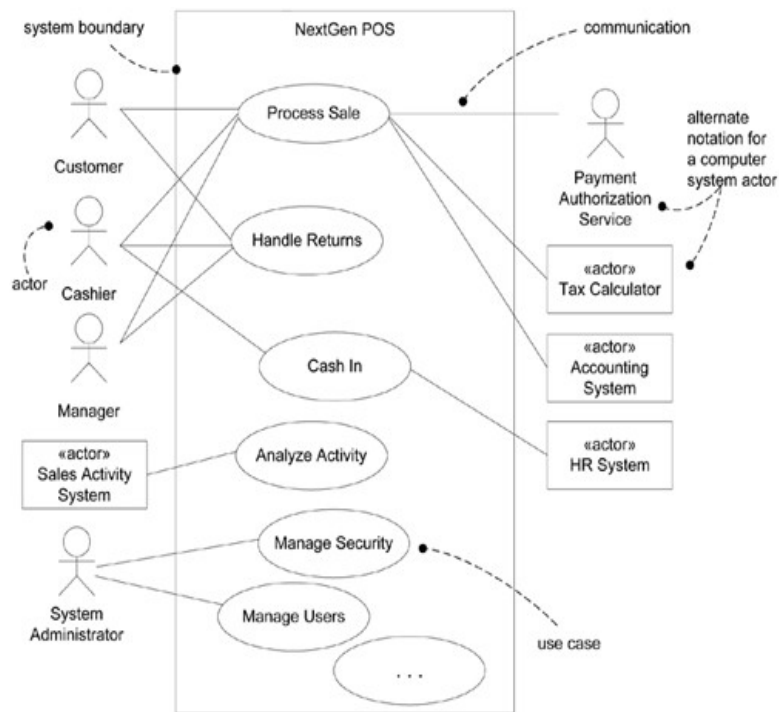
UML 2.0 Diagrams

- 13 UML diagrams



1. Use Case Diagram

- **Use case diagram** illustrates the name of use cases and actors, and the relationships between them.
 - **Use case** : a collection of related success and failure **scenarios in text**, that describe **how an actor uses the system to achieve a goal**
 - **Actor** : something with behavior, such as a person, computer or organization



Use case: **Handle Returns**

Main Success Scenario:

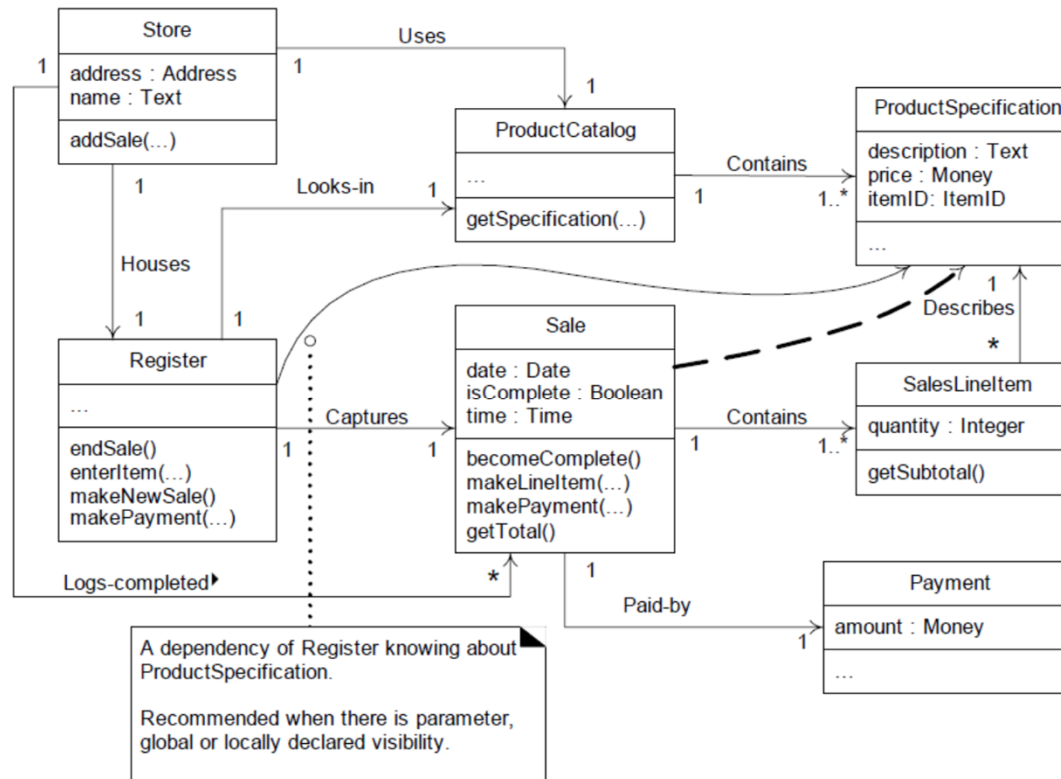
- A customer arrives at a checkout with items to return.
- The cashier uses the POS system to record each returned item ...

Alternate Scenarios:

- If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash ...

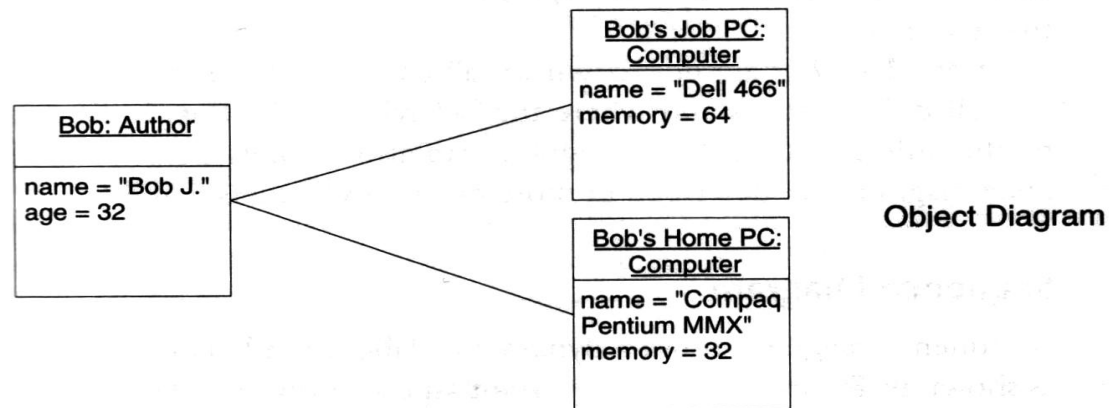
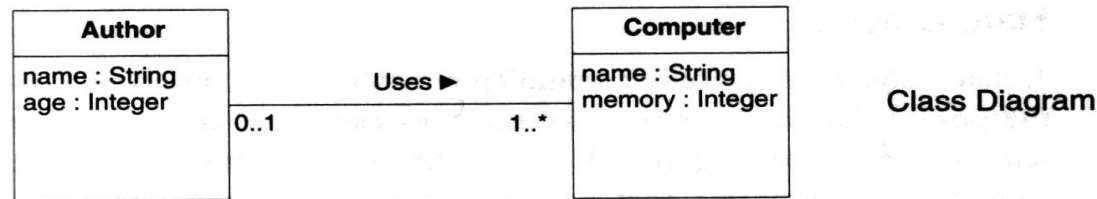
2. Class Diagram

- **Class diagram** shows the classes of the system, their inter-relationships, and the operations and attributes of the classes.
 - Domain model
 - Design class diagram (DCD)



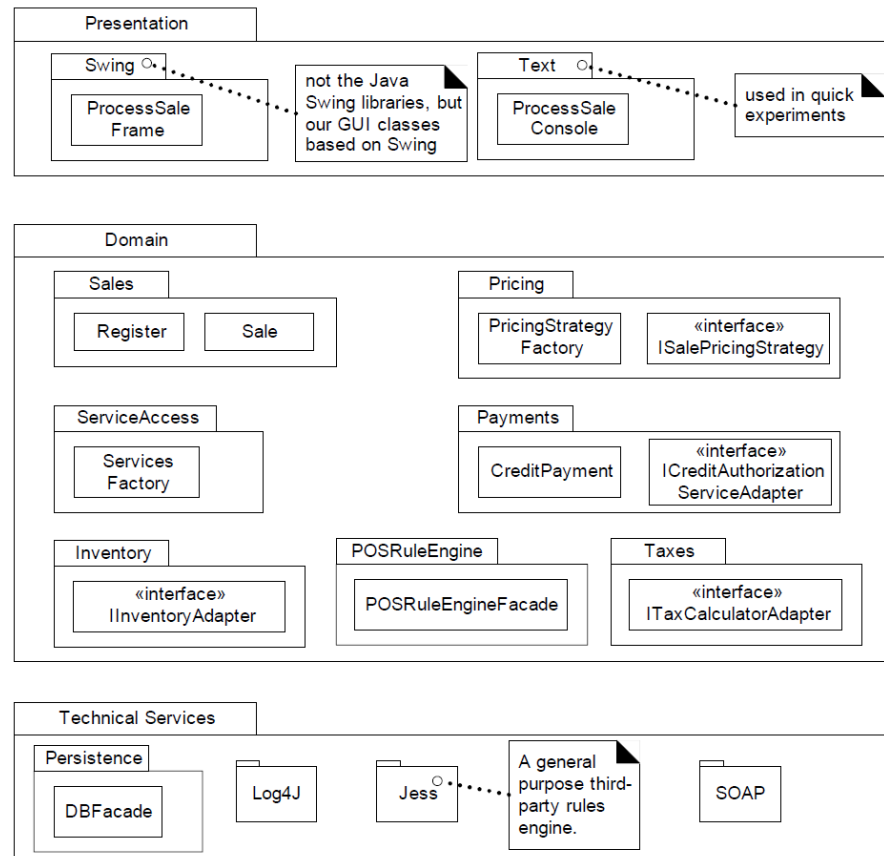
3. Object Diagram

- **Object diagram** is useful for exploring real world examples of objects and the relationships between them.
 - Shows instances of classes at a specific point of time (*i.e.*, snapshot)



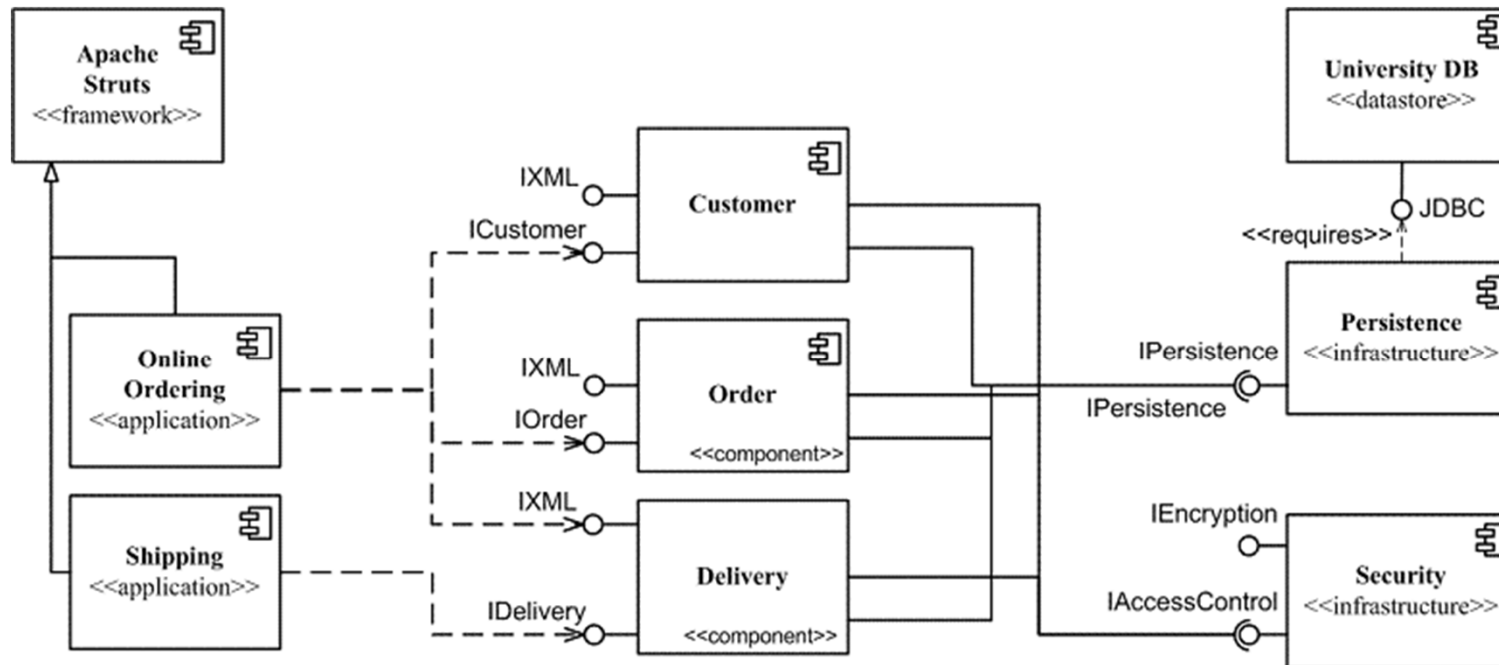
4. Package Diagram

- **Package diagram** groups classes into packages and simplify complex class diagrams.
 - A package is a collection of logically related UML elements.
 - Logical architecture



5. Component Diagram

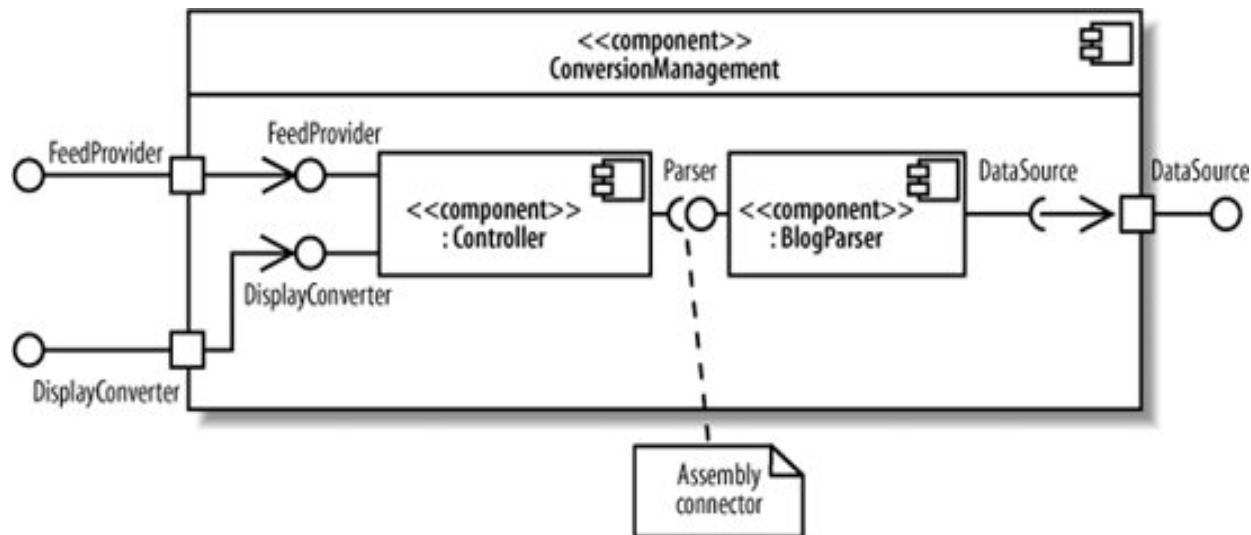
- **Component diagram** depicts how components are wired together to form larger components or software systems.
 - Illustrate the structure and inter-dependency of arbitrarily complex systems



Copyright 2005 Scott W. Ambler

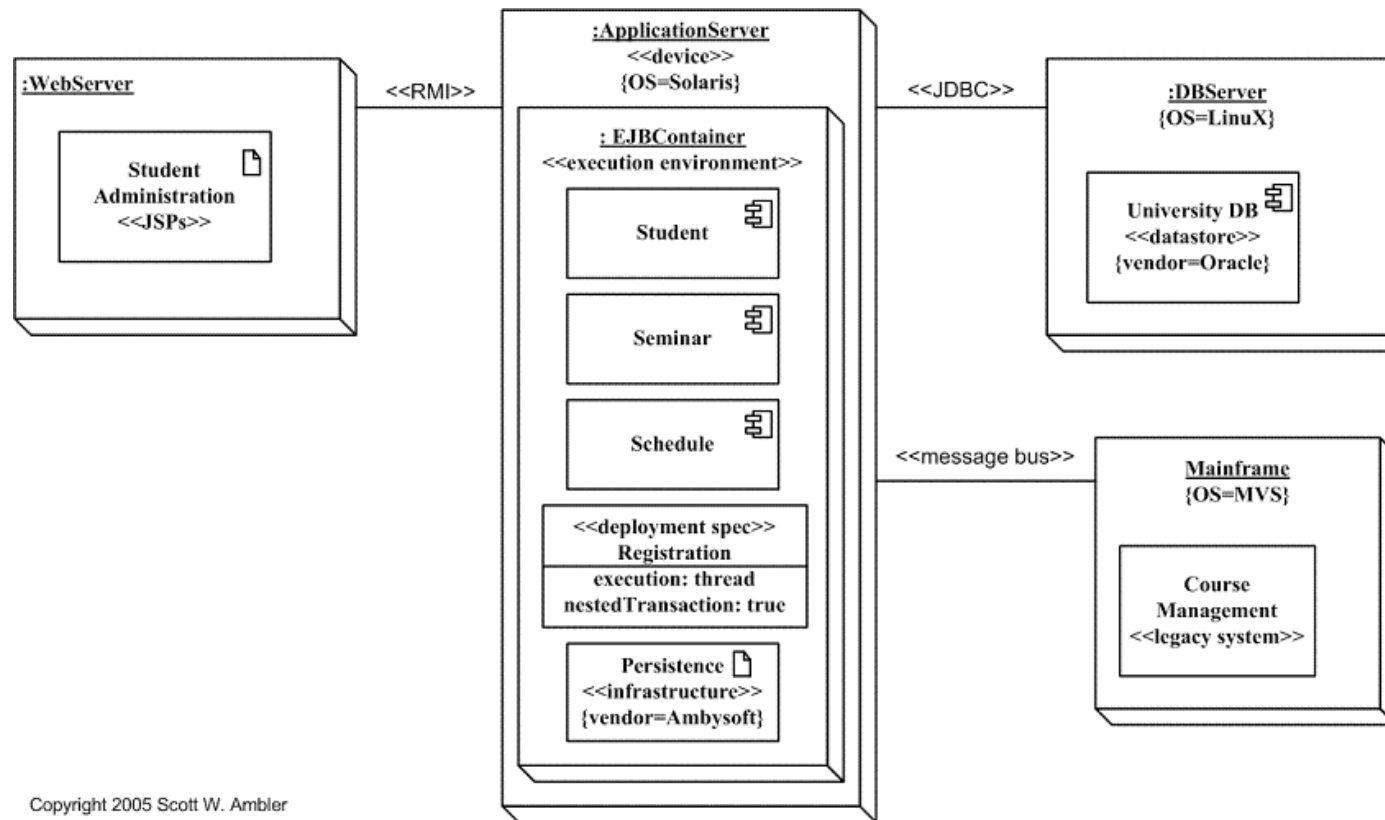
6. Composite Structure Diagram

- **Composite structure diagram** is used to explore run-time instances of interconnected instances collaborating over communications links.
 - Show the internal structure (including parts and connectors) of components.



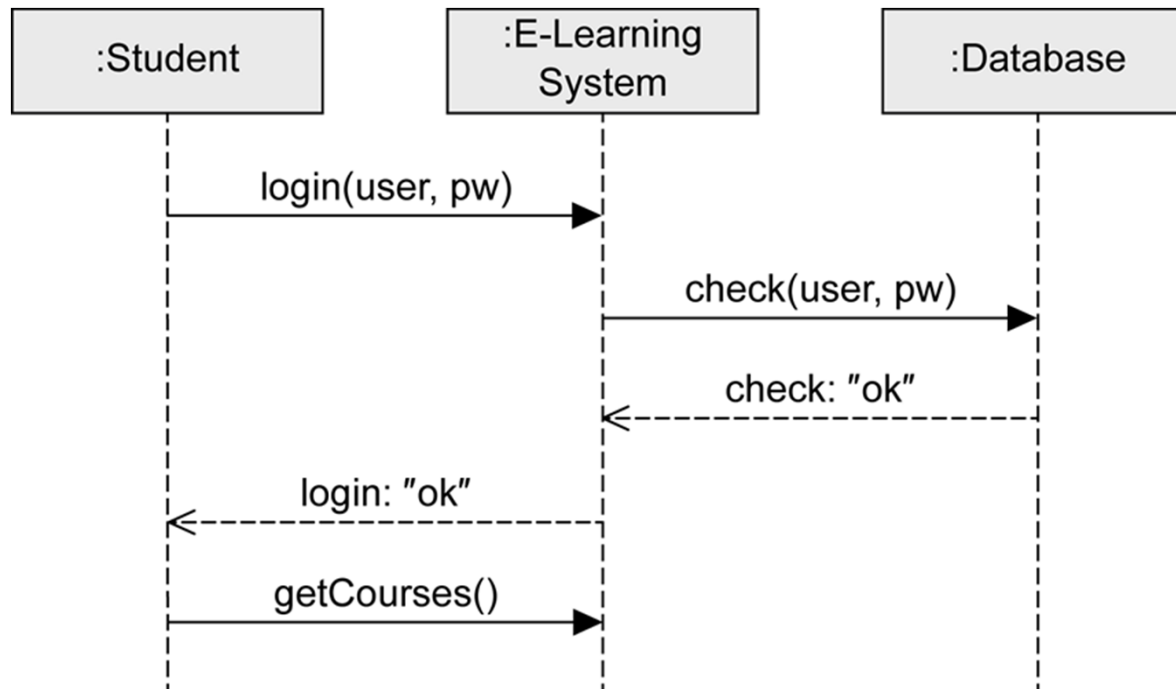
7. Deployment Diagram

- **Deployment diagram** depicts a static view of the run-time configuration of hardware nodes and the software components running on those nodes.



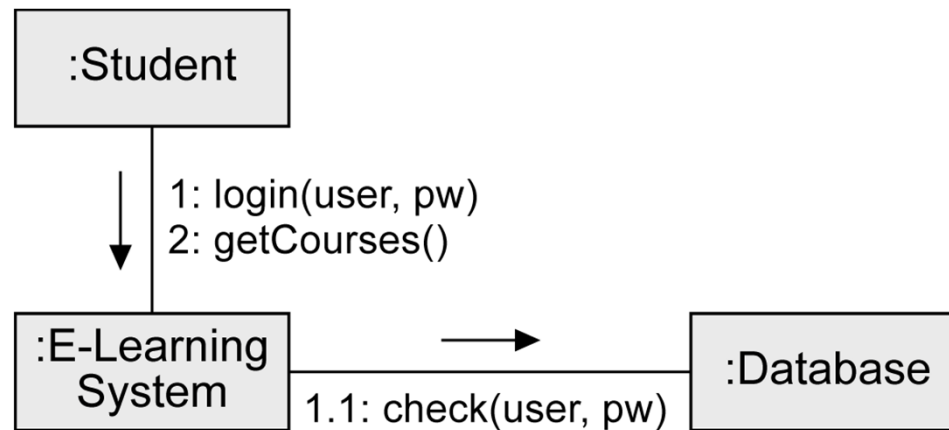
8. Sequence Diagram

- **Sequence diagram** models the collaboration of objects based on a time sequence.
 - Show how the objects interact with others in a particular scenario of a use case



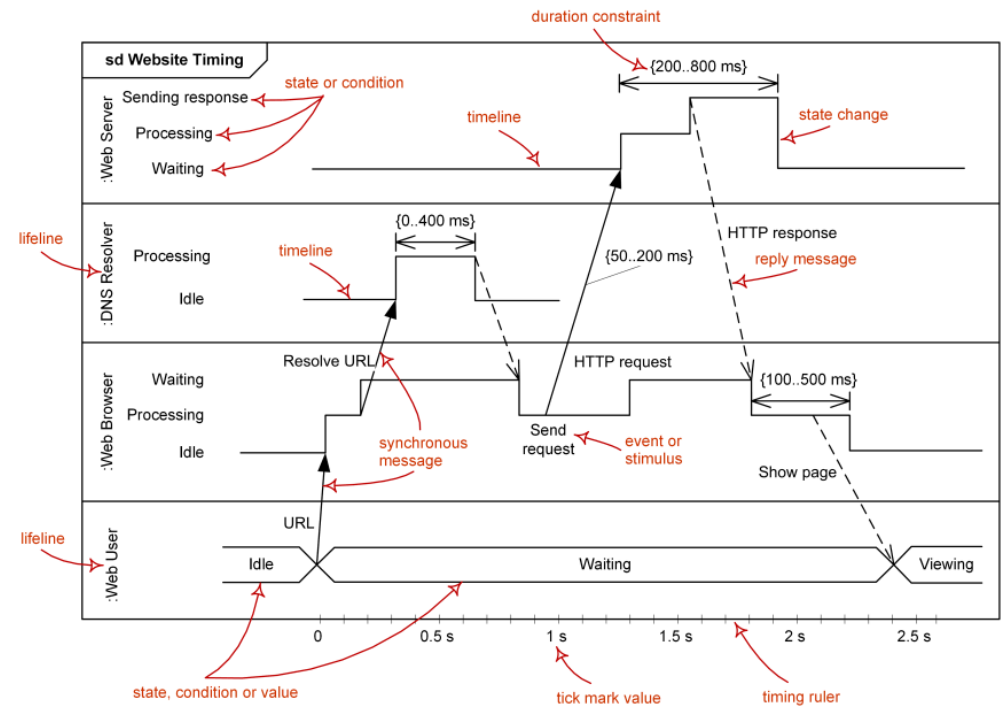
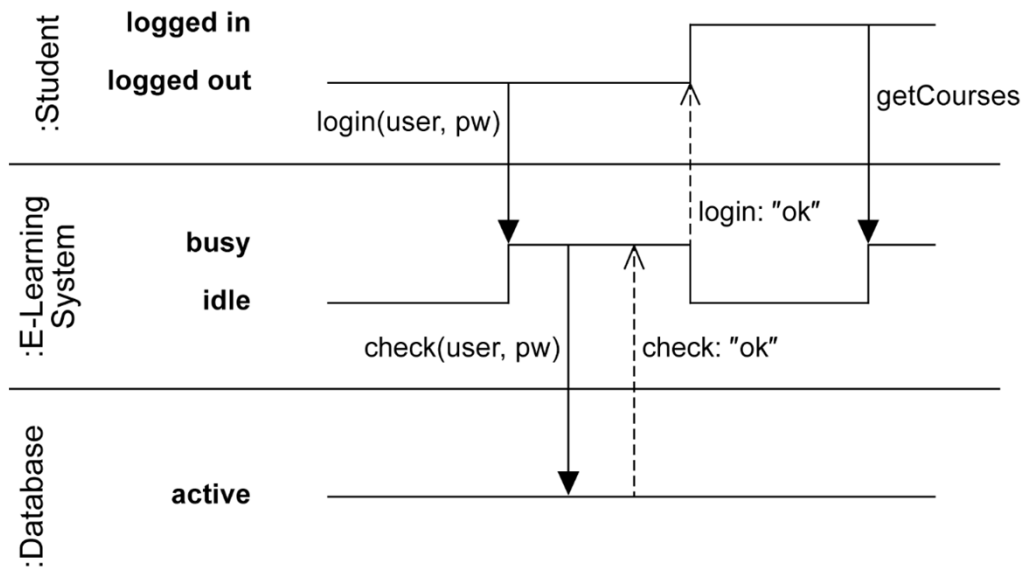
9. Communication Diagram

- **Communication diagram** is used to model the dynamic behavior of the use case. (called collaboration diagram)
 - ≈ Sequence diagram
 - More focused on showing the collaboration of objects rather than the time sequence.



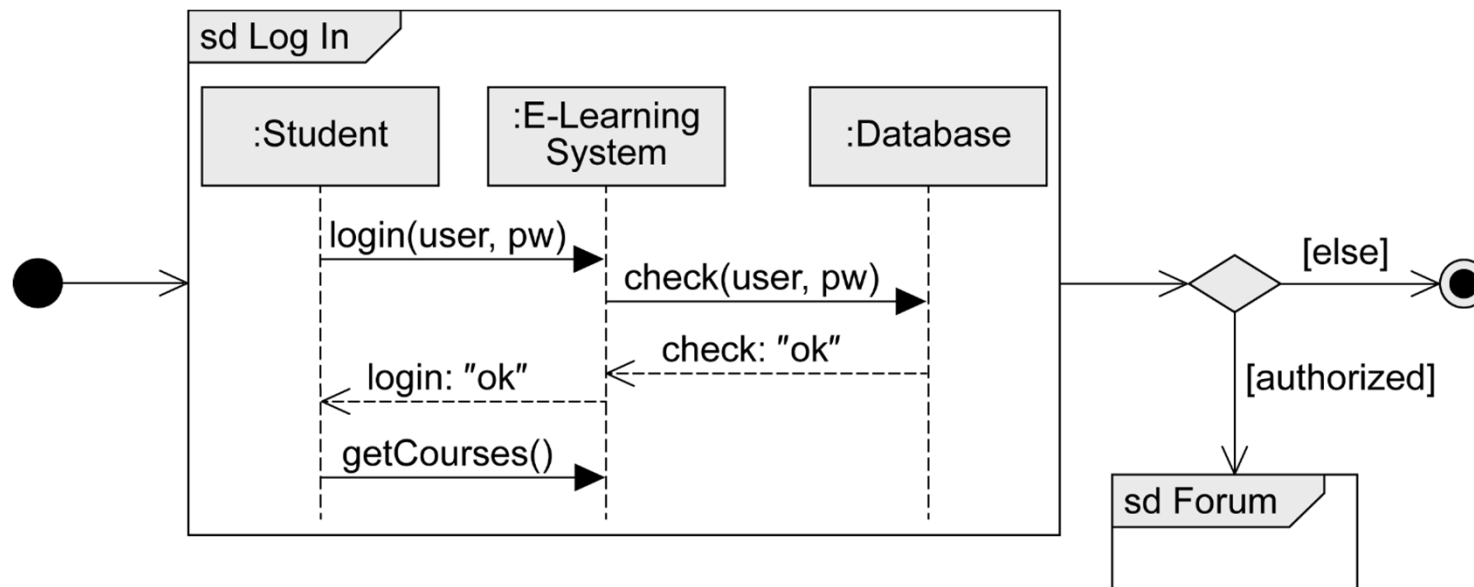
10. Timing Diagram

- **Timing diagram** shows the behavior of the objects in a given period of time.
 - A special form of a sequence diagram
 - The time increases from left to right and the lifelines are shown in separate compartments arranged vertically.



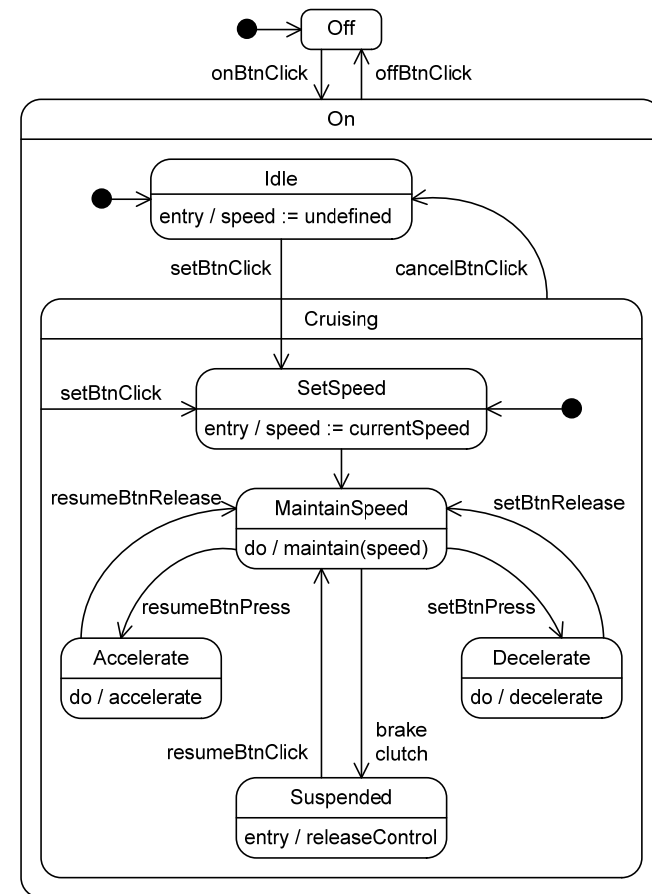
11. Interaction Overview Diagram

- **Interaction overview diagram** focuses on the overview of the flow of control of the interactions.
 - A variant of the Activity Diagram, where the nodes are the interactions or interaction occurrences.



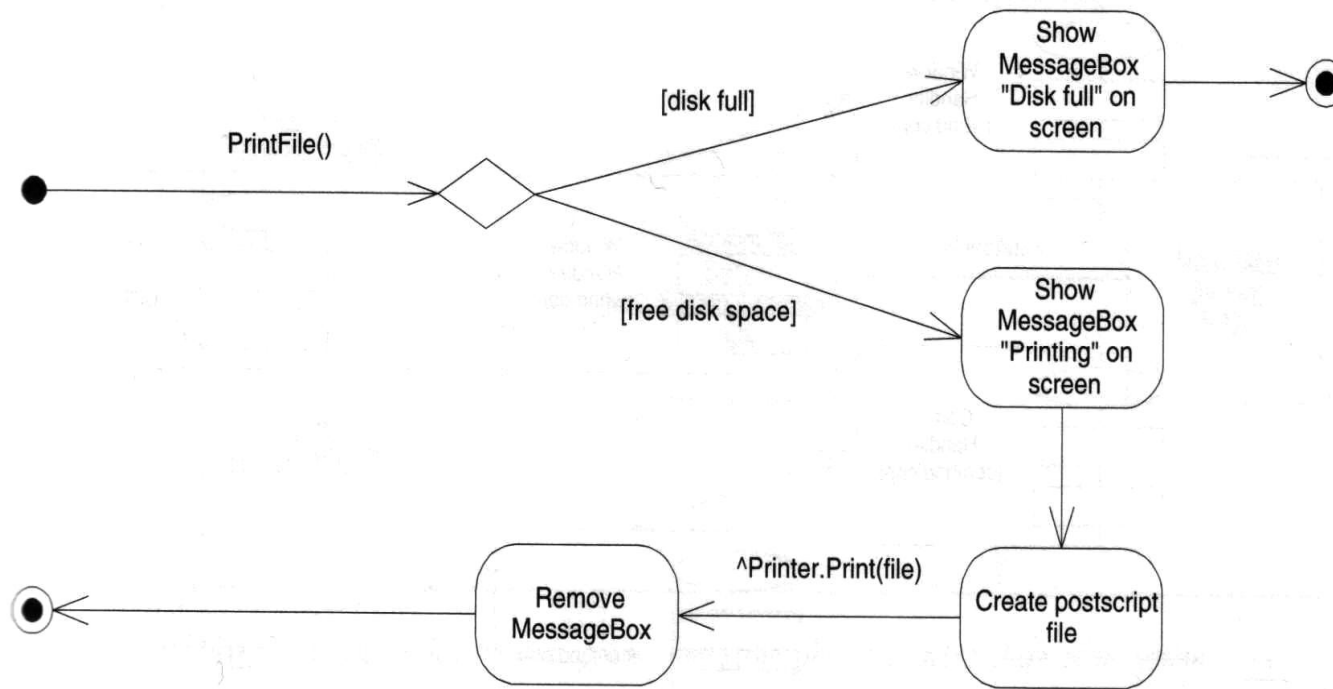
12. State (Statechart) Diagram

- **State diagram** can show different states of an entity and how an entity responds to various events by changing from one state to another.
 - Originated from the **Statecharts** formalism
 - The history of an entity is modeled by a finite state diagram.

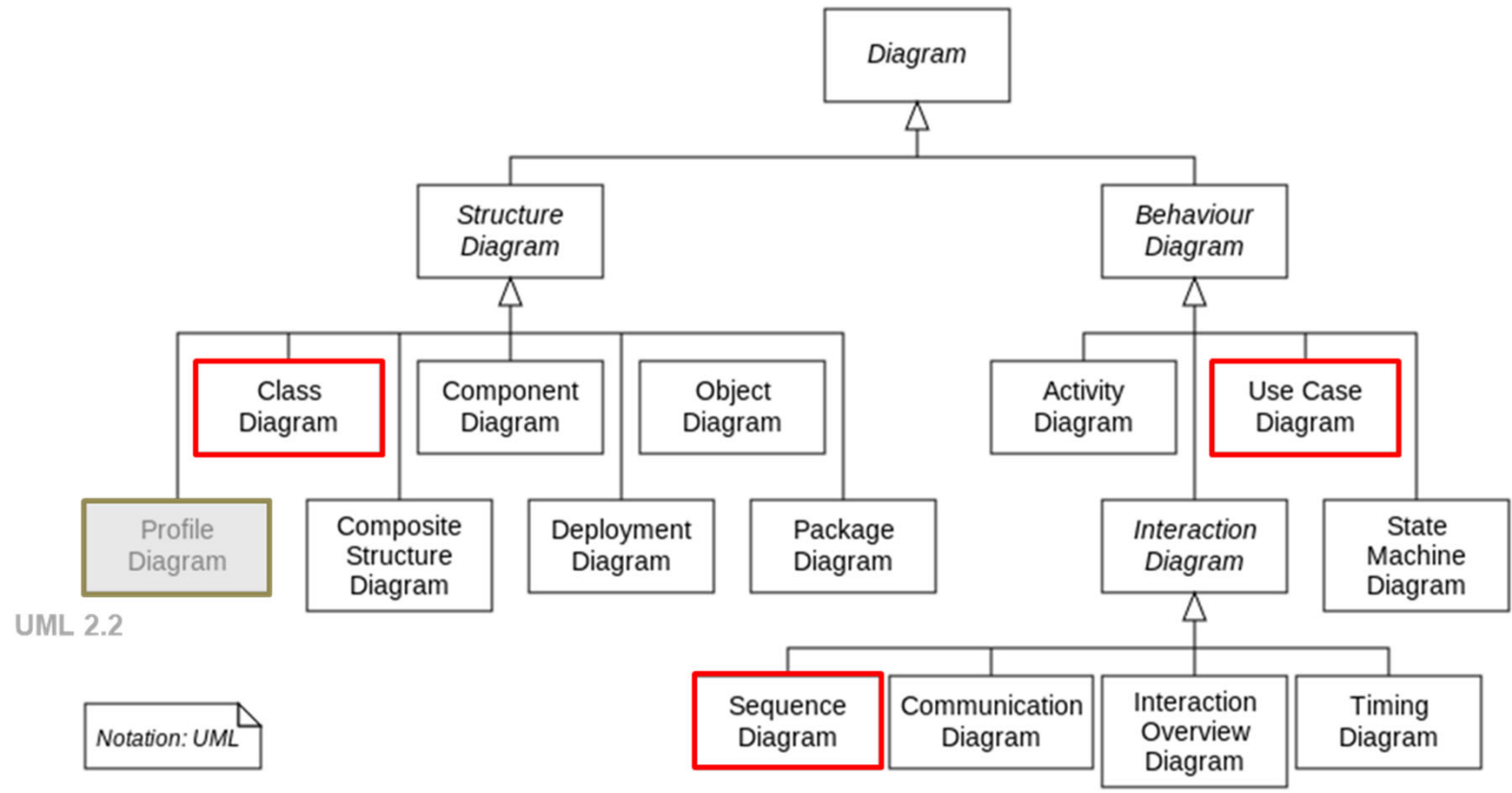


13. Activity Diagram

- **Activity diagram** helps to describe the flow of control of the target system.
 - Exploring complex business rules and operations, describing the use case and the business process
 - It is an object-oriented equivalent of flow-charts and DFDs (data flow diagrams).



13 UML Diagrams



Use Case Diagram

Use Cases

- **Use cases** are **text stories** of some actors using a system to meet goals.
 - A mechanism to capture (identify and 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**.
 - 3 formats (levels) : brief → casual → fully dressed

Use Case Diagram

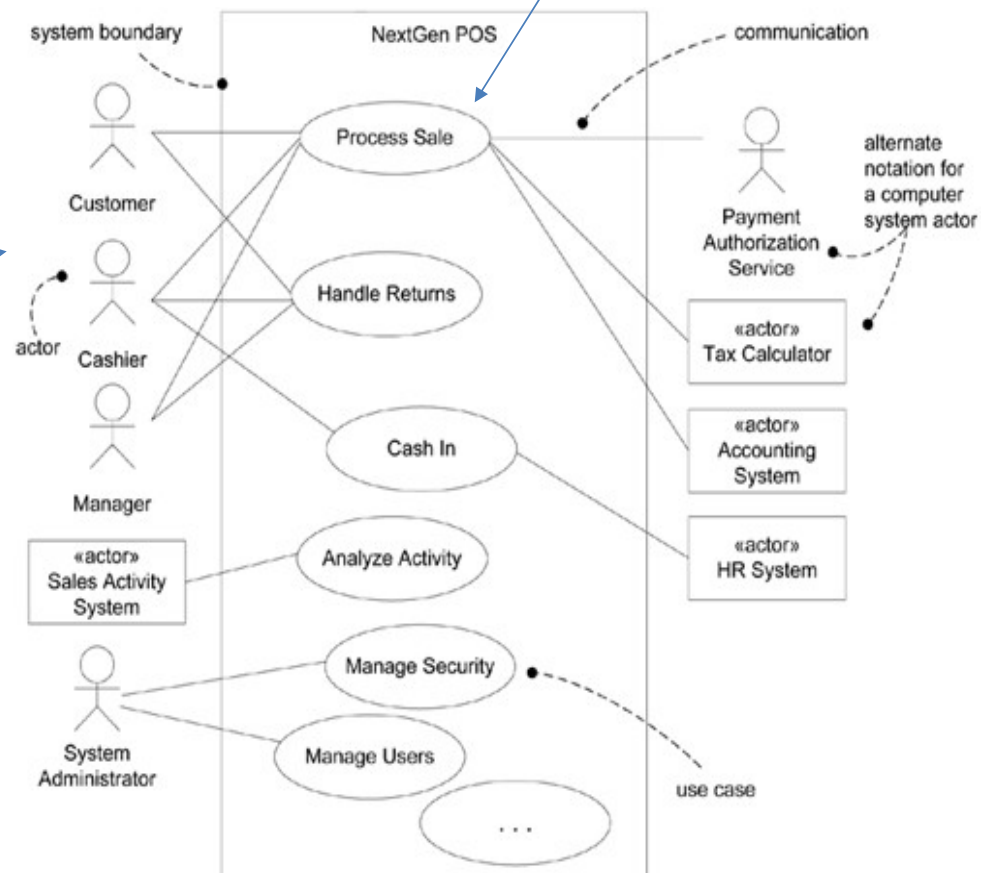
- **Use case diagram** illustrates the name of use cases and actors, and the relationships between them.
 - System context diagram
 - A summary of all use cases

Actor

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

- **Primary Actor** : has user goals fulfilled through using services of the SuD (System Under Discussion), e.g., cashier
- **Supporting Actor** : provides a service to the SuD, e.g., payment authorization service
- **Offstage Actor** : has an interest in the behavior of the use case, but is not primary or supporting, e.g., tax agency

Use case



3 Formats of Use Cases

- **Brief :**
 - Terse one paragraph summary
 - Usually the main success scenario or a happy path

- **Casual :**
 - Informal paragraph format
 - Multiple paragraphs that cover various scenarios

Handle Returns

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...

Alternate Scenarios:

If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

If the system detects failure to communicate with the external accounting system, ...

- **Fully Dressed :**
 - Includes all steps, variations and supporting sections (e.g., preconditions)

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, <i>and</i> worth telling the reader?
Success Guarantee	What must be true on successful completion, <i>and</i> 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.

Case Study: The NextGen POS System

The first case study is the NextGen point-of-sale (POS) system. In this apparently straightforward problem domain, we shall see that there are interesting requirement and design problems to solve. In addition, it's a real problemgroups really do develop POS systems with object technologies.

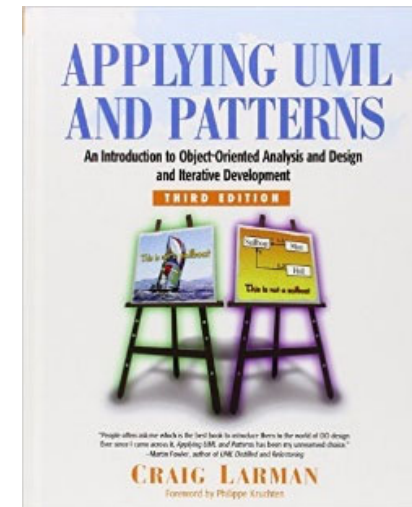
A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).



A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design, and implementation.



Example: Process Sale in Fully Dressed Style

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 Find Product Help 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.

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inventory system is failing.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 3 and 7.
- . . .

Technology and Data Variations List:

- *a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.
- 3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.
- 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- 7a. Credit account information entered by card reader or keyboard.
- 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:

- What are the tax law variations?
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader, or does the cashier have to do it?

Guideline: Write in an Essential Style

- **Essential writing style** is to express user intentions and system responsibilities, rather than concrete actions.
 - UI-free style
 - Concrete use cases are better avoided during early requirements analysis.
 - For example: ***Manage Users*** use case

Essential Style	Concrete Style
<ol style="list-style-type: none"> 1. Administrator identities self. 2. System authenticates identity. 3. ... 	<ol style="list-style-type: none"> 1. Administrator enters ID and PW in dialog box. 2. System authenticates Administrator. 3. System displays the “edit user” window. 4. ...

Guideline: Write Black-Box Use Cases

- **Don't** describe **the internal working** of the system, its components or design.
 - Define what the system does (*analysis*), rather than how it does it (*design*).

Black-box style	Not
The system records the sale.	The system writes the sale to a database. ...or (even worse): The system generates a SQL INSERT statement for the sale...

Are Use Cases Functional Requirements?

- Yes
- Use cases are requirements, primarily **functional requirements**.
 - “F” (functional or behavioral) in terms of **FURPS+** requirements types
 - Can also be used for other types.

System Sequence Diagram

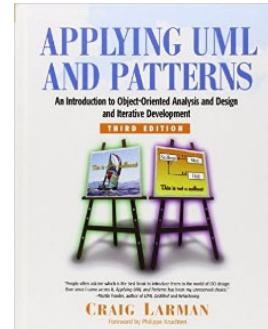
System Sequence Diagram

- **System sequence diagram (SSD)**

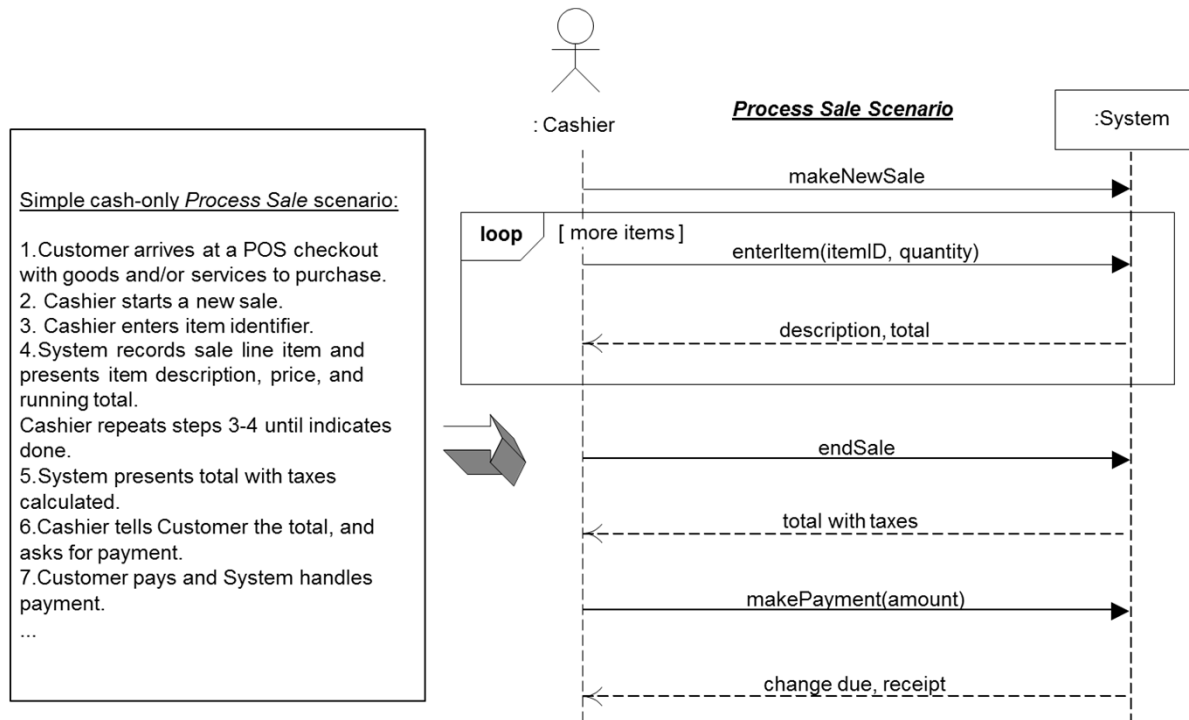
- A picture that shows the events that external actors generate, their order, and inter-system events, **for one particular scenario of a use case**.
 - **the external actors** that interact directly with the system,
 - **the system** (as a black box), and
 - **the system events** that the actors generate
- In the **sequence diagram** notation
- Depict system behavior in terms of **what the system does**, not how it does it
- Used as input to object design → **System operations**

- **Use cases** describe how external actors interact with the software system we are interested in creating.
 - During this interaction, an actor generates system events to a system, usually requesting some system operation to handle the event.

Applying UML Sequence Diagrams



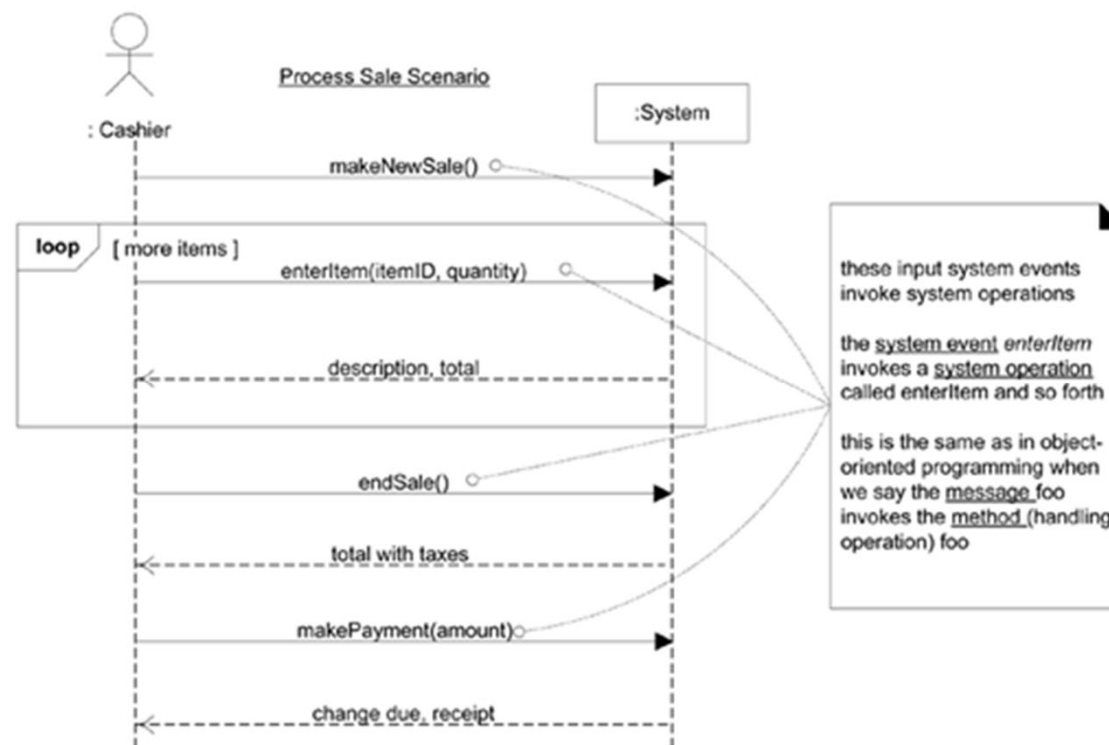
- The UML does not define something called ‘**System Sequence Diagrams**’.
 - We use the general **UML sequence diagram notation**.
 - The term ‘system’ in SSDs is used to emphasize the application of the UML sequence diagram to **systems viewed as black boxes**.
 - An SSD shows system events for one scenario of a use case.



System Operation

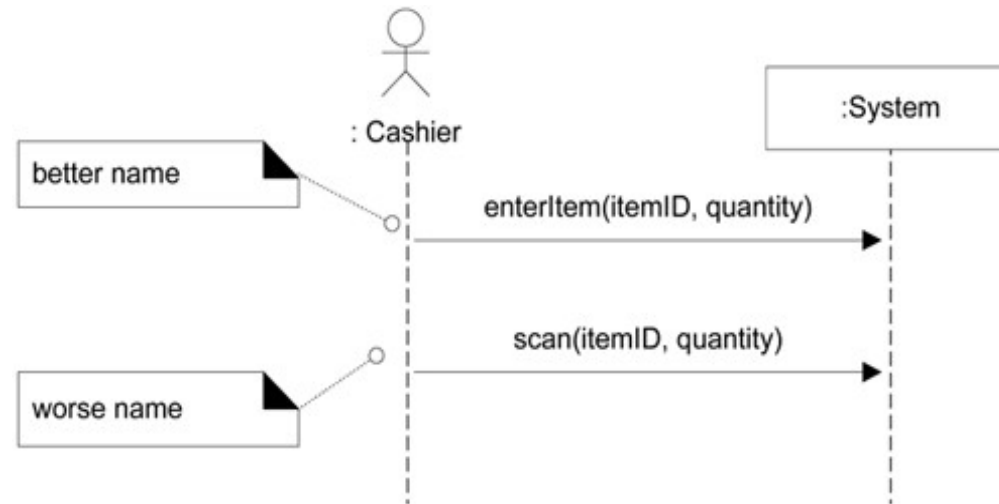
- System operations**

- Operations that the system as a black box component offers in its public interface
- Show **system events**, which the SUD should have system operations to handle the system events.
- **System Interfaces**: the entire set of system operations across all use cases

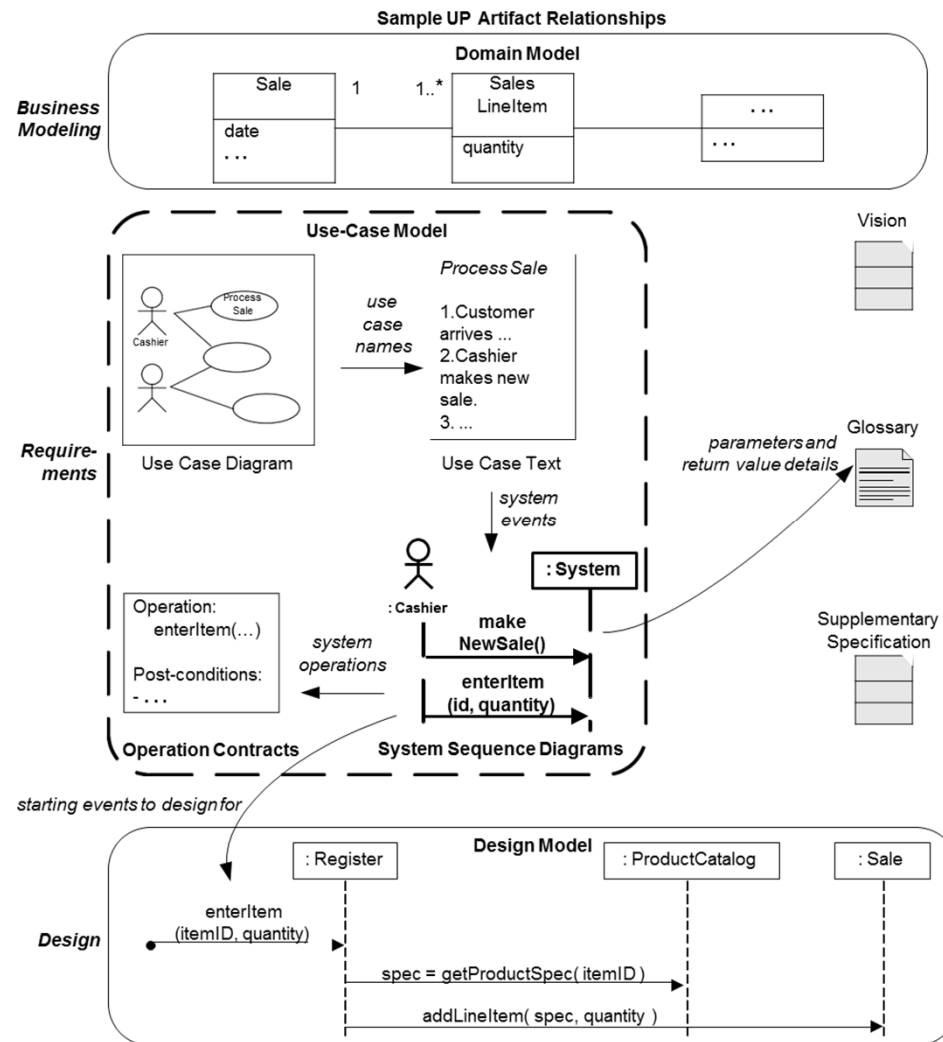


Guideline: How to Name System Events and Operations?

- System events should be expressed at the abstract level of intention rather than in terms of the physical input device.
- Example : *scan(itemID)* vs. *enterItem(itemID)*
 - The *enterItem* name is better, since it communicates intention rather than the input device.



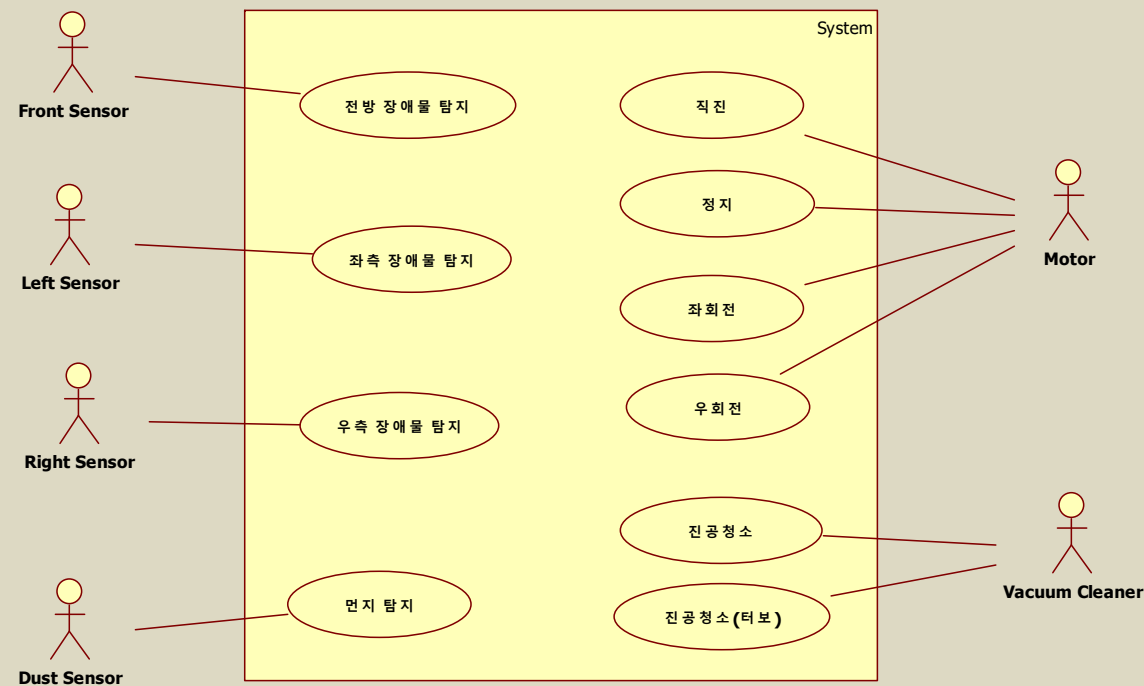
The Relationship to Other Artifacts in UP (OOAD)



Homework / Activity #10



- RVC SW Control을 OOAD 기법으로 분석(OOA)합니다.
 - 모든 Use Cases를 도출하고, SSD를 통해 System Operations을 모두 찾습니다.
 - 추가 UC : “장애물을 회피한다.” “방 전체를 깨끗하게 청소한다.” “장애물을 감지한다.”
 - UML 도구 사용



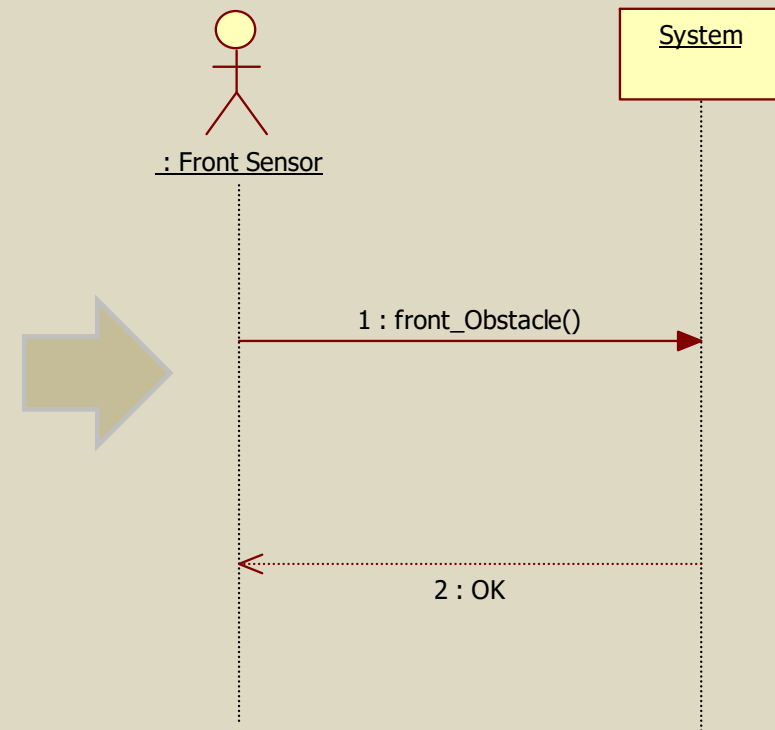
An Example Use-Case Diagram for the RVC Control SW

Samples of Use Cases and SSD



Use Case	Use Case의 이름
Actor	관련된 모든 Actors
Pre-Requisites	이 Use-Case가 실행되기 전에 만족되어야 하는 조건들
Typical Courses of Events	해당 Use-Case가 가장 널리 수행되는 시나리오 - 외부의 Actor와 시스템 간의 Interaction을, 시간 순서대로, 현 시점에서 알 수 있는 범위 내에서, 자세하게 설명합니다. 1. 2. 3. 4. ...
Alternative Courses of Events	가끔 수행되는 시나리오들을 부분적으로 설명합니다. (주 시나리오와 시작과 종료가 같음)
Exceptional Courses of Events	오류/이상상황으로 주시나리오의 종료 상황에 도달할 수 없는 경우를 설명

Use-Case Description



System Sequence Diagram

8. Software Testing

Program Testing

- **Testing** intends to show:
 - “a program does what it is intended to do” and
 - “discover program defects before it is put into use”.

- When you test software, you **execute a program** using artificial data.
 - You check the results of the test run for errors, anomalies or information about the program’s non-functional attributes.
 - Can reveal the presence of errors, but NOT their absence.

- Testing is a part of general **verification and validation (V&V)** process and activities.

Two Types of Program Testing

- **Validation testing**

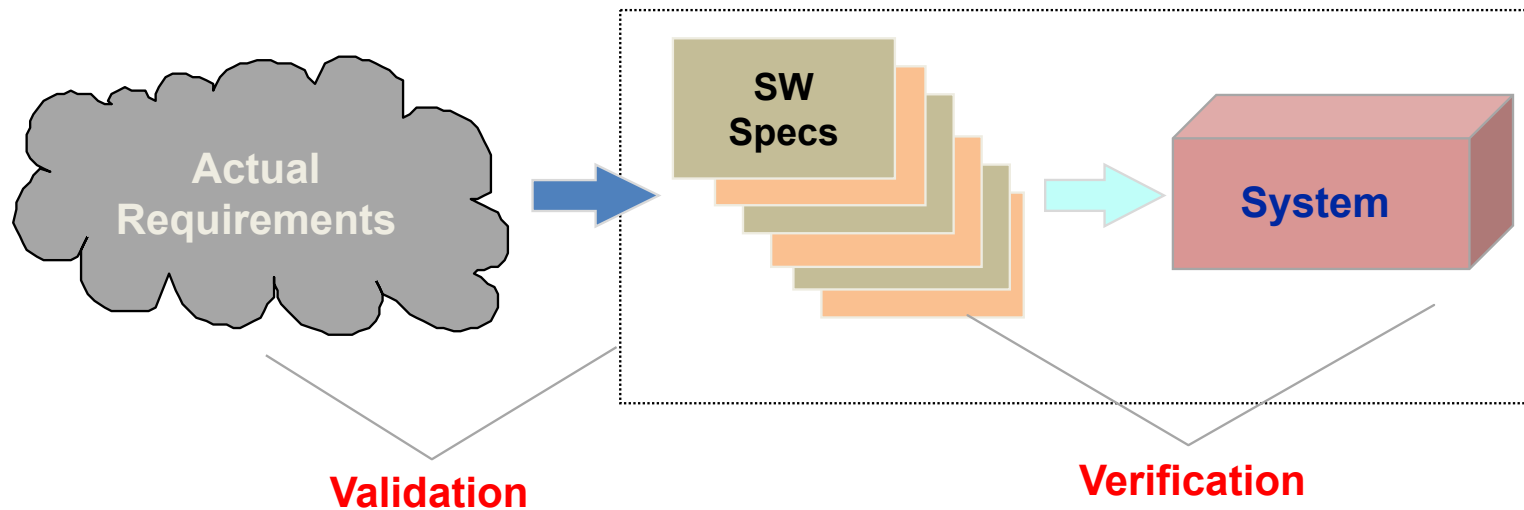
- To demonstrate to the developer and the customer that “*the software meets its (users’) requirements.*”
- A successful test shows that the system operates as intended.
 - You expect the system to perform correctly using a given set of test cases that reflect the system’s expected use.

- **Verification testing**

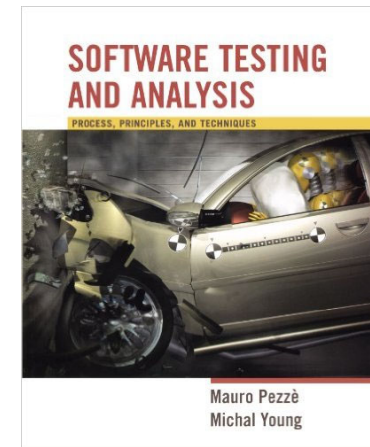
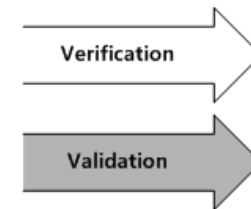
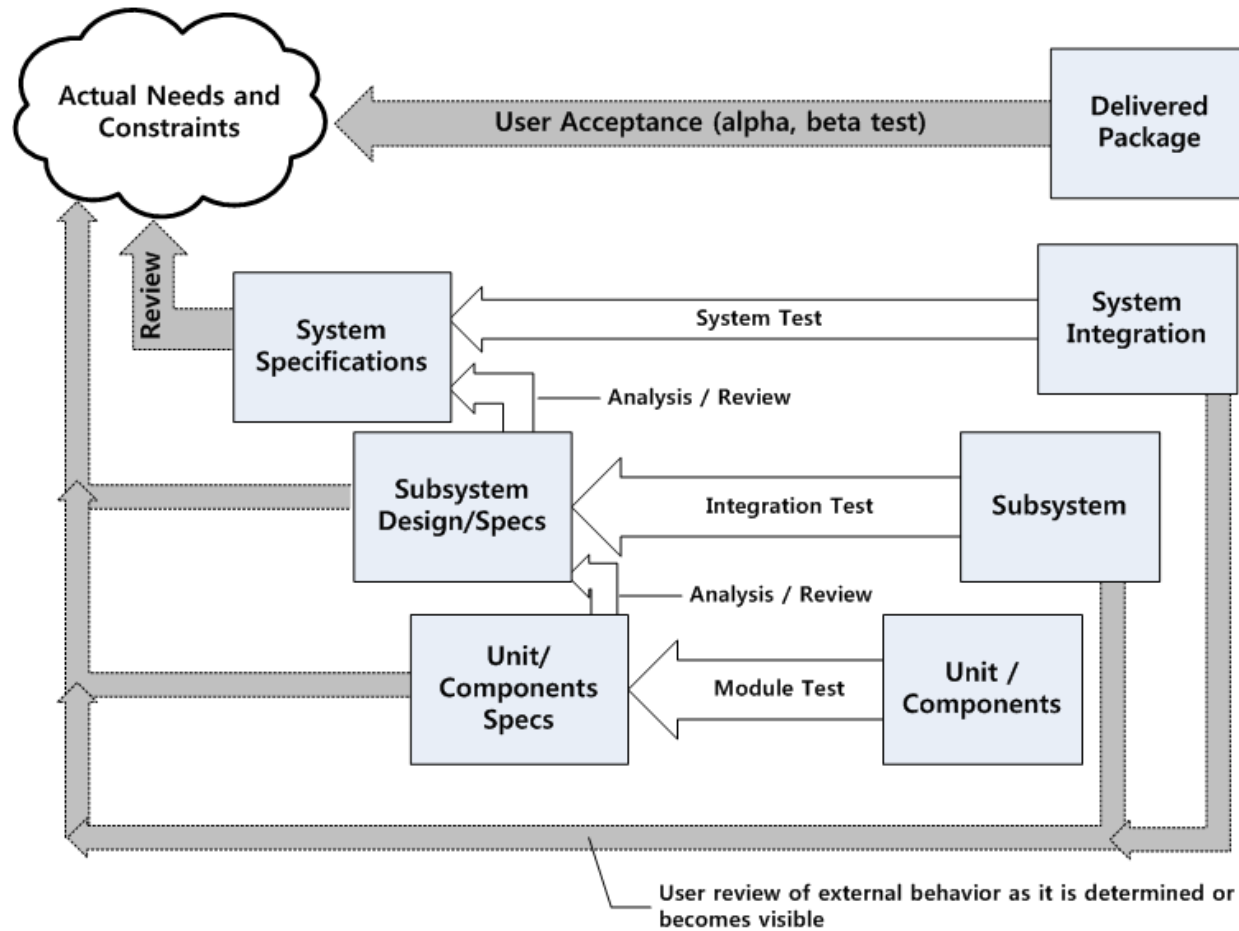
- To discover situations in which “*the behavior of the software is incorrect, undesirable or does not conform to its specification.*”
- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.
 - The test cases are designed to expose defects.
- = Defect testing

Verification and Validation (V&V)

- Validation: Are we building the right software?
 - *“Does the software system meets the user's real needs?”*
- Verification: Are we building the software right? *(with respect to requirements specification)*
 - *“Does the software system meets the requirements specifications?”*



V-Model of V&V Activities

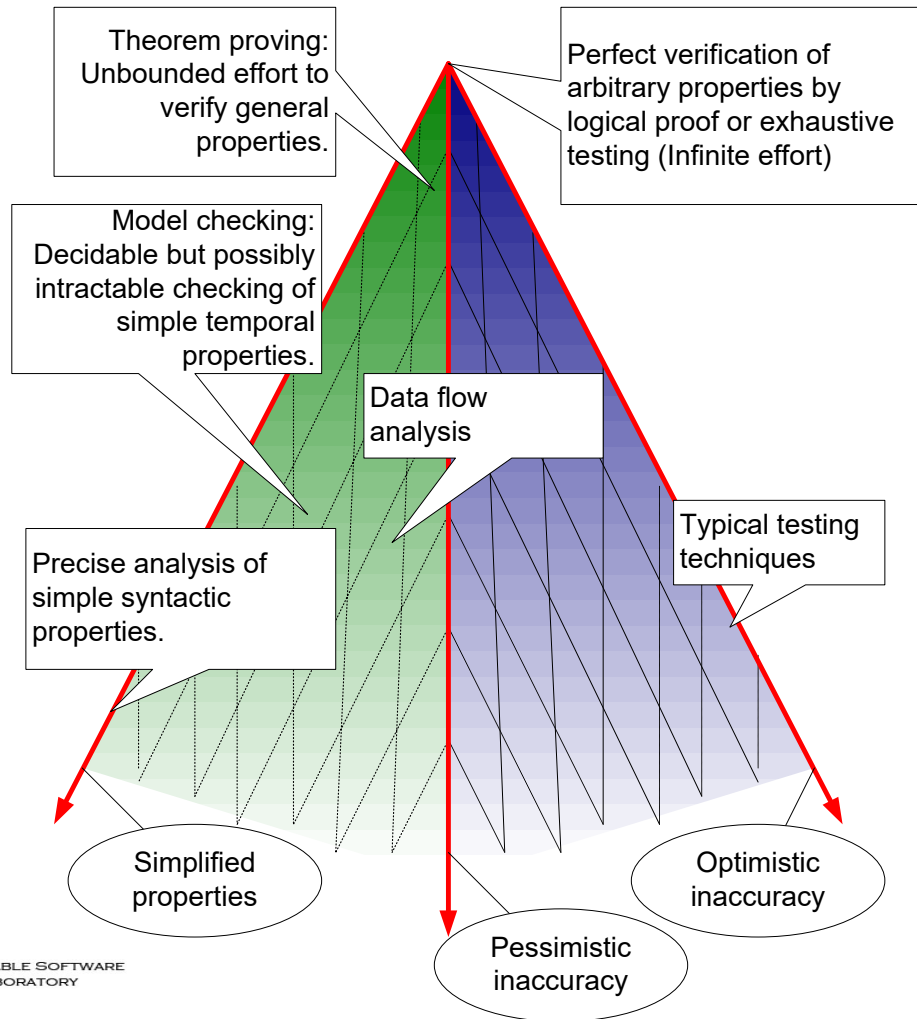


V&V Confidence

- Aim of V&V
 - Establish confidence that the system is ‘fit for purpose’

- **V&V confidence** depends on
 - Software purpose
 - The level of confidence depends on how critical the software is to an organisation.
 - User expectations
 - Users may have low expectations of certain kinds of software.
 - Marketing environment
 - Getting a product to market early may be more important than finding defects in the program.

3 Axes of V&V



- **Optimistic Inaccuracy**

- We may accept some programs that do not possess the property.
- It may not detect all violations.
- Example: Testing

- **Pessimistic Inaccuracy**

- It is not guaranteed to accept a program even if the program does possess the property being analyzed, because of false alarms.
- Example: Automated program analysis

- **Simplified Properties**

- It reduces the degree of freedom by simplifying the property to check.
- Example: Model Checking

Software Testing Stages

- **Software testing stages**

- **Development testing**

- The system is tested during development to discover bugs and defects.

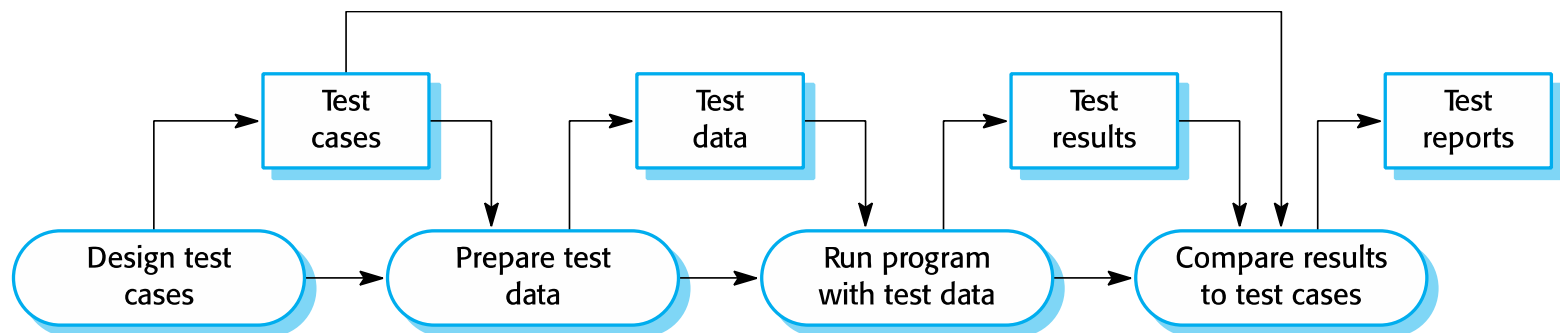
- **Release testing**

- A separate testing team test a complete version of the system, before it is released to users.

- **User testing**

- Users or potential users of a system test the system in their own environment.

- **Software testing process**



Development Testing

Development Testing

- **All testing** activities that are carried out by the team developing the system.
 - **Unit testing**
 - Individual program units or object classes are tested.
 - Unit testing should focus on testing the functionality of objects or methods.
 - **Integrated testing**
 - Several individual units are integrated to create composite components.
 - Integration testing should focus on testing interfaces and interactions among components.
 - **System testing**
 - Some or all components in a system are integrated and the system is tested as a whole.
 - System testing should focus on testing all functionalities as a whole.
 - **Regression testing**
 - Testing a system to check that changes have not 'broken' previously working code
 - In development or maintenance phase

Unit Testing

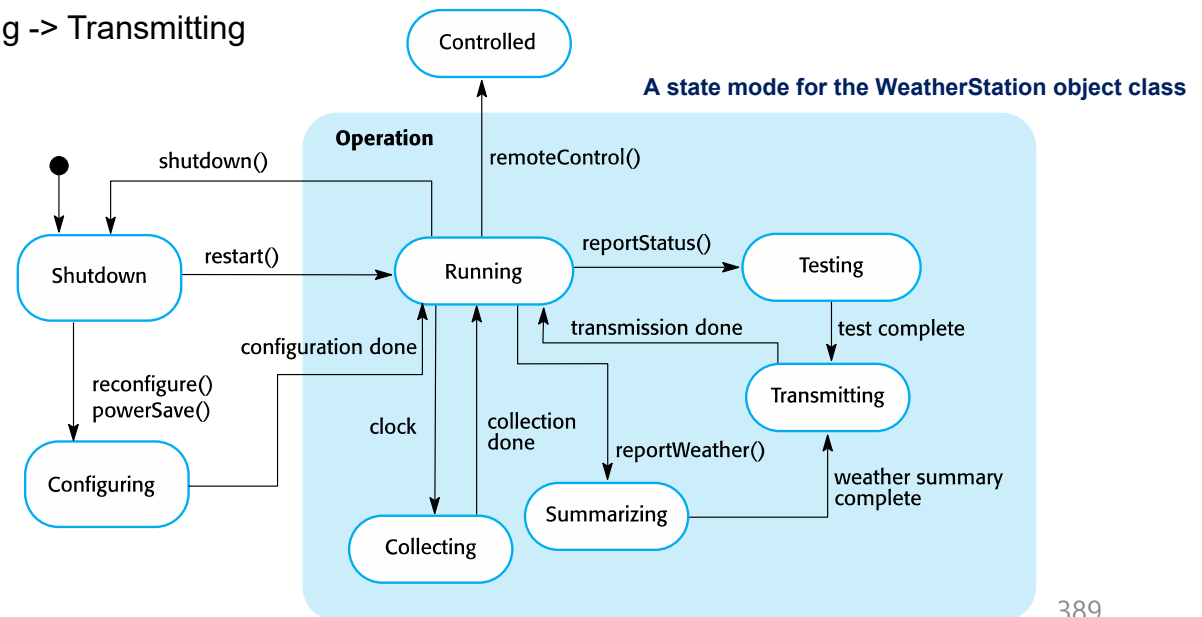
- **Unit testing** is the process of **testing individual components in isolation**.
 - Defect testing

- Units may be:
 - **Individual functions or methods** within an object
 - **Object classes** with attributes and methods
 - Testing all operations associated with an object
 - Setting and interrogating all object attributes
 - Exercising the object in all possible states
 - **Composite components** with defined interfaces used to access their functionality.

The Weather Station: Unit Testing for Objects

- Need to define test cases for all operations in all states of the object.
 - **State model** can identify sequences of state transitions to be tested and the event sequences to cause these transitions.
 - For example:
 - Shutdown -> Running-> Shutdown
 - Configuring-> Running-> Testing -> Transmitting -> Running
 - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running

WeatherStation
identifier
reportWeather () reportStatus () powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)



Automated Testing

- Whenever possible, [unit testing should be automated](#).
 - Tests are run and checked without manual intervention.

- **Unit testing frameworks**
 - Provide generic test classes that you extend to create specific test cases.
 - Can run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.
 - Example: **JUnit**, **xUnit**, etc.

 - Composed of 3 parts
 - **Setup part** : initialize the system with the test case, namely the inputs and expected outputs.
 - **Call part** : call the object or method to be tested.
 - **Assertion part** : compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

Developing Unit Test Cases

- Two types of unit test cases
 - **Positive**
 - Reflect normal operation of a program
 - Should show that the component works as expected
 - **Negative**
 - Based on testing experience of where common problems arise
 - Use abnormal inputs to check that these are properly processed and do not crash the component

Unit Testing Strategies

- **Partition testing**

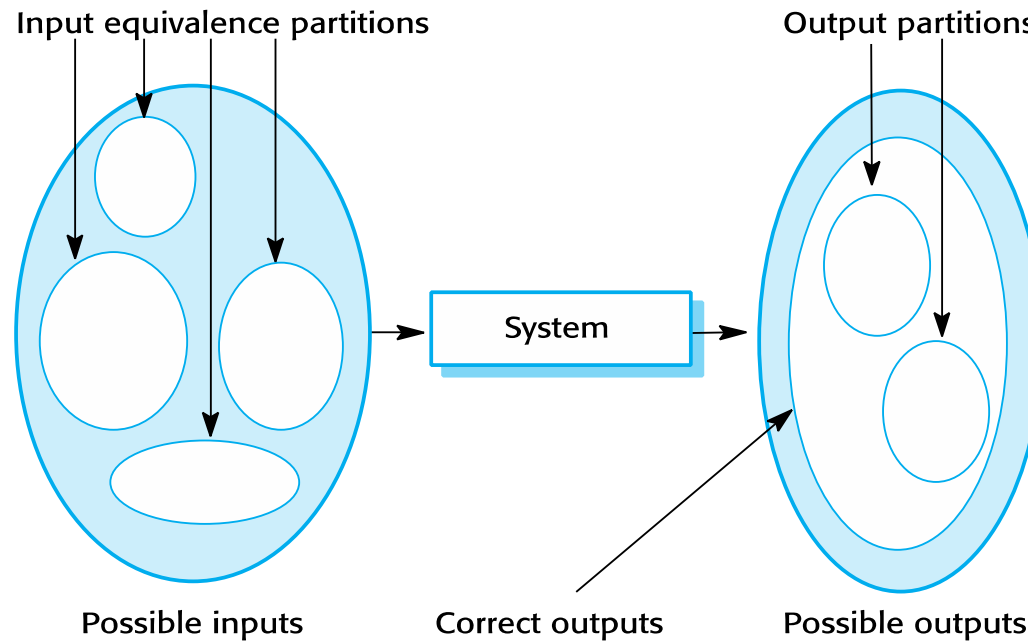
- Identify groups of inputs that have common characteristics and should be processed in the same way.
- Choose tests from within each of these groups.

- **Guideline-based testing**

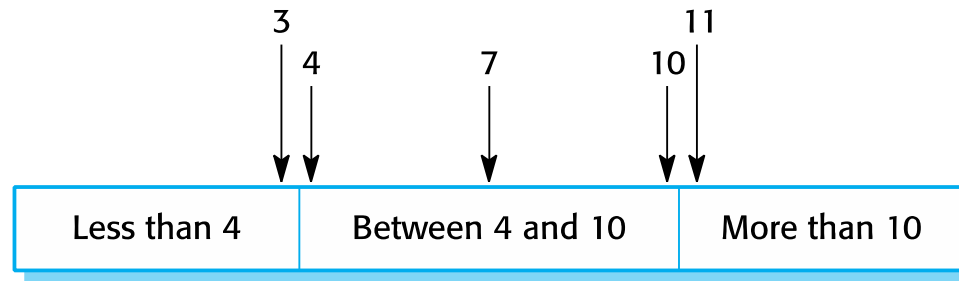
- Use testing guidelines to choose test cases.
- These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.
 - Brute-force testing (AKA 막 테스트)
- Examples:
 - Choose inputs that force the system to generate all error messages.
 - Design inputs that cause input buffers to overflow.
 - Repeat the same input or series of inputs numerous times.
 - Force invalid outputs to be generated.
 - Force computation results to be too large or too small.

Partition Testing

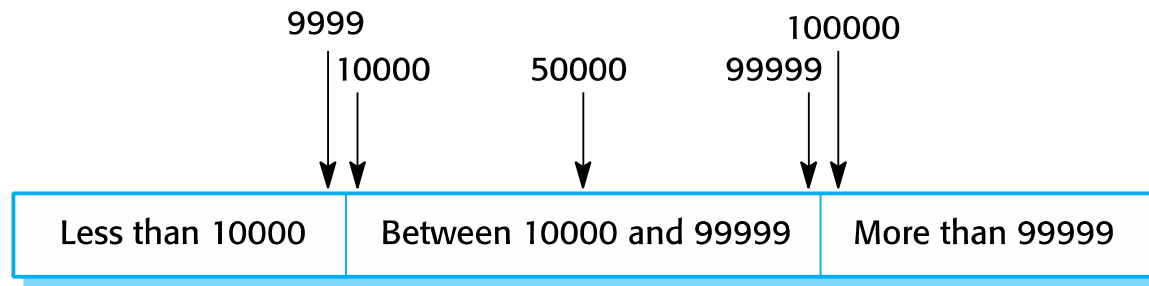
- Input data and output results often fall into different classes where all members of a class are related.
 - Each of these classes is an **equivalence partition** or domain where the program behaves in an equivalent way for each class member.
 - Test cases should be chosen from each partition.



Equivalence Partitions with Boundary Value Analysis

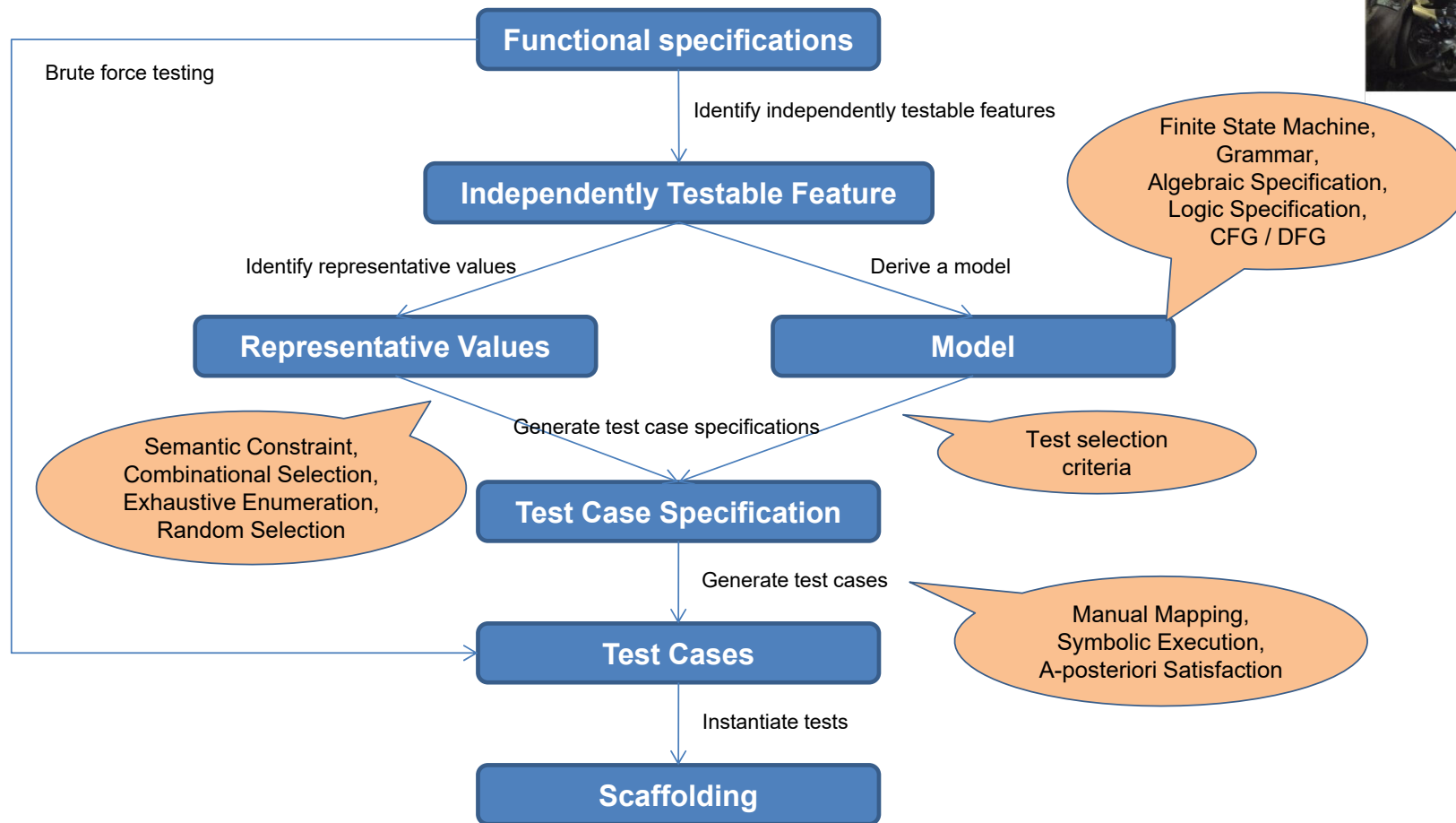
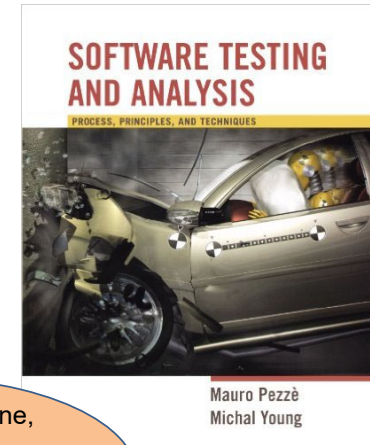


Number of input values



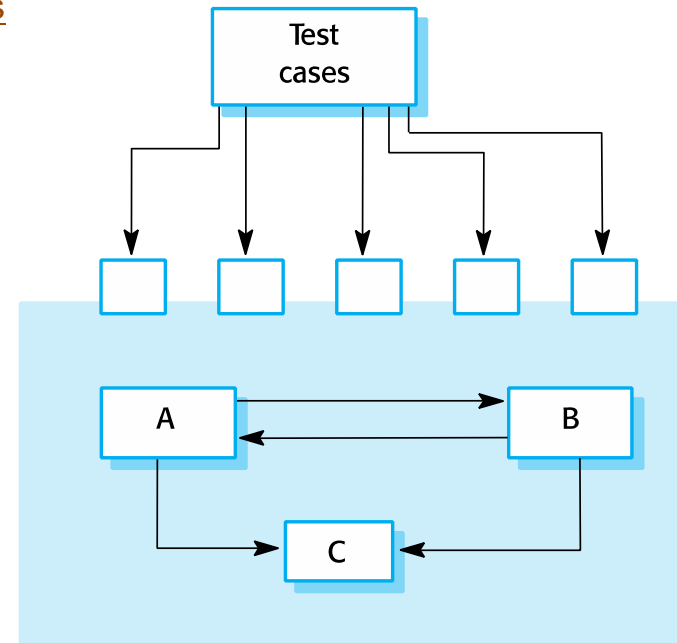
Input values

Functional Testing – Techniques Overview



Integration Testing

- **Software components** are often **composite components** that are made up of **several interacting objects**.
 - Can access the functionality of these objects through the defined **component interface**
- **Integration testing** is the testing of composite components.
 - Focus on showing that the **component interface** behaves according to its specification
 - Focus on testing the **interactions between components**
 - Assume that unit tests on the individual objects within the component have been completed.



Guidelines for Integration Testing

- **Interface Testing Guidelines**

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components are activated

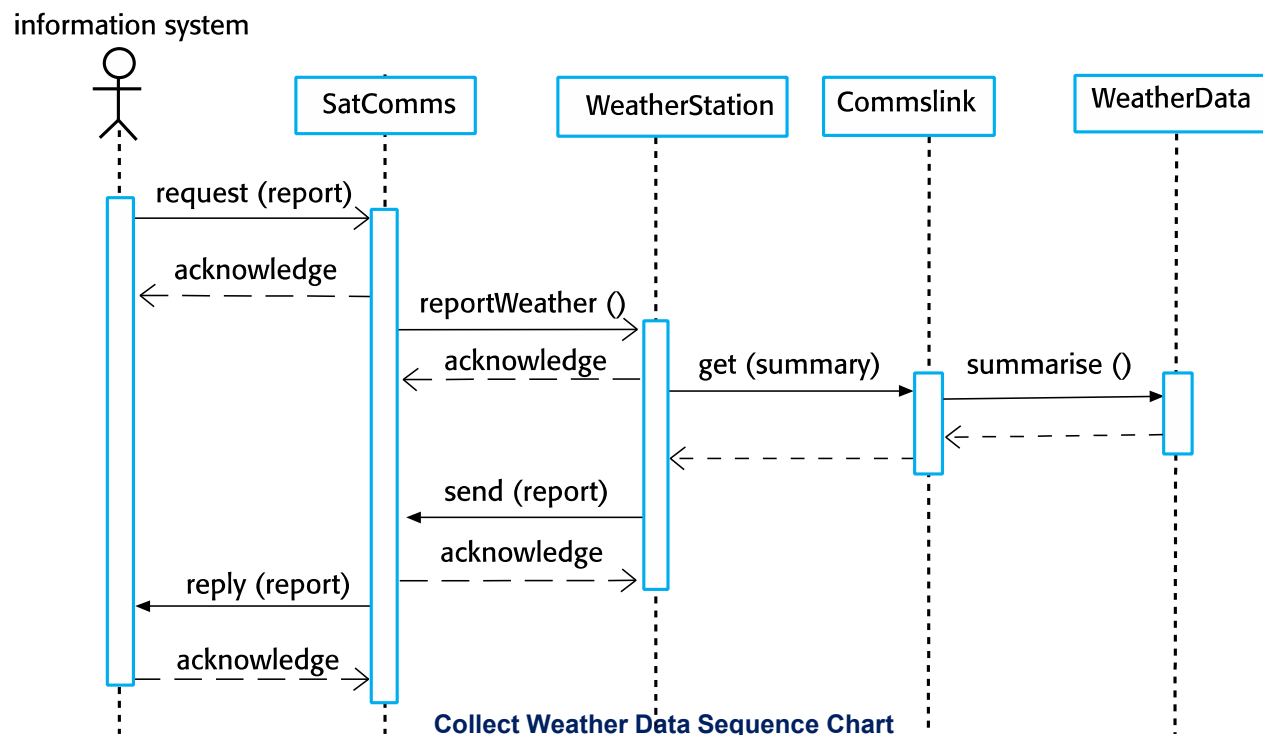
System Testing

- **System testing** during development involves integrating components to create a version of the system and then testing the integrated system.
 - The focus is testing the **interactions between components**. (*Integration testing*)
 - Checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces
 - Tests the **emergent behavior of a system** (*System testing*)

- System testing is a **collective process**.
 - Reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
 - Components developed by different team members or sub-teams may be integrated at this stage.
 - System testing may involve a separate testing team with no involvement from designers and programmers.
 - **Release Testing**

Developing System Test Cases

- **Use-cases** and **Sequence diagrams** can be used as a basis.
 - Each use case usually involves several system components so testing the use case forces these interactions to occur.
 - Sequence diagrams associated with the use case document the components and interactions that are being tested.



Testing Policies

- Exhaustive system testing is a;ways impossible.
 - **Testing policies** define a required system test coverage.

- Examples of testing policies
 - *“All system functions that are accessed through menus should be tested.”*
 - *“Combinations of functions accessed through the same menu must be tested.”*
 - *“Where user input is provided, all functions must be tested with both correct and incorrect input.”*

Regression Testing

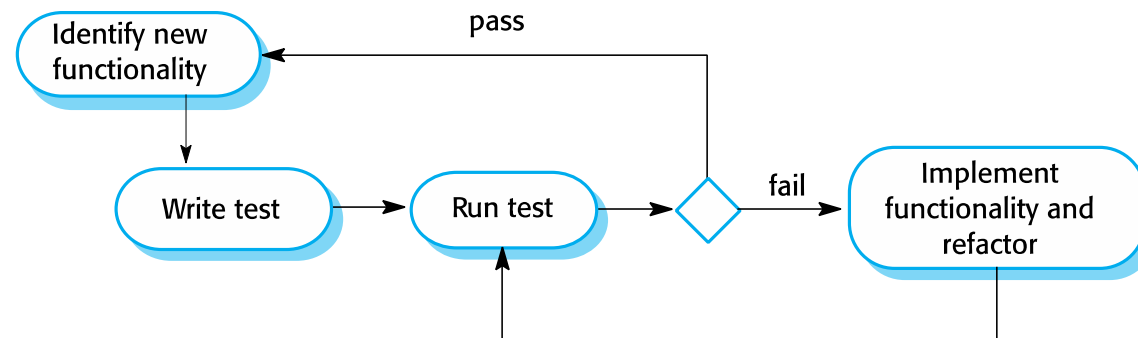
- **Regression testing**
 - Testing a system to check that changes have not '*broken*' previously working code

- In a manual testing process, regression testing is expensive but, with **automated testing**, it is simple and straightforward.
 - All tests are rerun every time a change is made to the program.
 - Tests must run 'successfully' before the change is committed as TFD in XP.

Test-Driven Development

Test-Driven Development

- **Test-driven development (TDD)** is a program development approach inter-leaving testing and code development.
 - Tests are written before code and ‘passing’ the tests is the critical driver of development.
 - Develop code incrementally, along with a test for that increment.
 - Not move on to the next increment, until the code passes its test.
- TDD was introduced as part of agile methods such as **XP**.
 - However, it can also be used in plan-driven development processes.



Benefits of TDD

- **Code coverage**
 - Every code segment that you write has at least one associated test so all code written has at least one test.

- **Regression testing**
 - A regression test suite is developed incrementally as a program is developed.
 - Tests the system to check that changes have not 'broken' previously working code through rerunning the tests every time a change is made to the program.

- **Simplified debugging**
 - When a test fails, it should be obvious where the problem lies.
 - The newly written code needs to be checked and modified.

- **System documentation**
 - The tests themselves are a form of documentation that describe what the code should be doing.

Release Testing

Release Testing

- **Release testing** is the process of testing a particular release of a system that is intended for use outside of the development team.
 - To convince the supplier of the system that it is good enough for use.
 - Should show that the system delivers its specified functionality, performance and dependability
 - Should show the system does not fail during normal use

- Release testing is usually **a black-box testing** process.
 - Tests are only derived from the system specification.

Release Testing vs. System Testing

- **Release testing** is a form of system testing.
- Important differences are
 - A separate team that has not been involved in the system development should be responsible for release testing.
 - System testing by the development team should focus on discovering bugs in the system. (defect/verification testing)
 - Release testing is to check that the system meets its requirements and is good enough for external use. (validation testing)
- **Performance tests**
 - Involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- **Stress testing**
 - A form of performance testing where the system is deliberately overloaded to test its failure behavior.

User Testing

User Testing

- **User or Customer testing** is a stage in which users or customers provide input and advice on system testing.
 - Influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

- Types of user testing
 - **Alpha testing**
 - Users of the software work with the development team to test the software at the developer's site.
 - **Beta testing**
 - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
 - **Acceptance testing**
 - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.
 - Primarily for custom systems

Homework #11

- 다양한 Unit Testing Framework을 찾고, 실제로 적용 가능한 방법론을 하나 선정하세요.
- 선정된 방법론을 Homework #7에서 개발한 C program을 대상으로 적용하세요.
- Unit Test Cases를 20개 이상 개발하고, 실제 테스트를 수행하세요.
- Unit Test Report를 A4 10장 (글자크기 10 이하)으로 작성하세요.

List of C++ Unit Testing Frameworks (Wikipedia)

C++ [\[edit\]](#)

Name	License	xUnit	Fixtures	Group fixtures	Generators	Mocks	Exceptions	Macros	Templates	Grouping	Source	Remarks
Aeryn		No	Yes	Yes	No	No	Yes	Yes	Yes	Yes	^[89]	
API Sanity Checker	GNU LGPL	Yes	Yes (spectypes)	Yes (spectypes)	Yes						^[31]	Unit test generator for C/C++ libraries. Can automatically generate reasonable input data for every API function. LGPL.
ATF	BSD						Yes	Yes	Yes	Yes	^[32]	Originally developed for the NetBSD operating system but works well in most Unix-like platforms. Ability to install tests as part of a release.
Bandit	MIT	No (describe/it)	Yes (describe)	Yes (Nested describe)	No	No	Yes	Yes	No	Yes (Nested describe)	^[90]	Header only. Automatic test registration. Specifically developed for C++11
Boost Test Library	Boost	Yes ^[91]	Yes ^[92]	Yes ^{[93][94]}	Yes	With additional library "Turtle" ^[95]	Yes	User decision	Yes	Suites and labels	^[96]	Part of Boost. Powerful dataset concept for generating test cases. Different levels of fixtures (global, once per test suite, once per each test case in a suite). Powerful floating point comparison.
BugEye	Boost	No	No	No	No	No	Yes	No	No	Yes	^[97]	Header-only. TAP output.
QA Systems Cantata	Proprietary	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	^[34]	Commercial. Automated unit and integration testing tool for C++. Certified testing for host or embedded systems. Code coverage and unique call interface control to simulate and intercept calls.
Casmine	GPL 2.0	No	Yes	Yes	No	No	Yes	Yes	Yes	Yes	^[98]	C++17, modeled after the Jasmine testing framework, type-safe tests, auto-registration, BDD features, focused/disabled/pending tests, flexible configuration (JSON), colored console reporter, extendable, Windows/Linux/macOS
Catch or Catch2	Boost	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	^[99]	Header only, no external dependencies, auto-registration, tdd and bdd features
CATCH-VC6		No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	^[100]	VC6 port of CATCH
cfix		Yes	Yes	No	No	No	Yes	Yes	No		^[37]	Specialized for Windows development—both Win32 and NT kernel mode. Compatible to WinUnit.
Cput		Yes	Yes	Yes	Yes		Yes	Yes	No	Suites	^[101]	Library and MS Visual Studio add-in to create and run unit tests. Open Source.
CPPOCL/test	Apache 2	No	Yes		No		Yes	Yes			^[102]	Released Under Apache 2.0, Compliant with C++ 98 and C++ 11. Works for Linux and Windows 32/64 bit using gcc, Cygwin, VS2005 and VS2015. Header file only library. Provides ability to write performance tests in a similar way to unit tests. Has some support for reporting memory leaks.
CppTest	GNU LGPL		Yes					Yes		Suites	^[103]	Released under LGPL
cpptest-lite	MIT		Yes				Yes	Yes		Suites	^[104]	Released under MIT. Developed for C++11.
CppUnit	GNU LGPL	Yes	Yes	Yes	No	No	Yes	Yes	No	Suites	^{[105][106]}	Released under LGPL

List of C++ Unit Testing Frameworks (Wikipedia)

Name	License	xUnit	Fixtures	Group fixtures	Generators	Mocks	Exceptions	Macros	Templates	Grouping	Source	Remarks
CppUTest		Yes	Yes	Yes	No	Yes	No	Yes	No	Suites	[43]	Limited C++ set by design to keep usage easy and allow it to work on embedded platforms. C++ is buried in macros so the learning curve for C programmers is minimal. Ported to Symbian. Has a mocking support library CppUMock
CppUnitLite		Yes			No	No	No	Yes	No	Suites	[107]	
CPUit		Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	[108]	Released under BSD.
Criterion	MIT	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Suites	[109]	Unit testing framework with automatic test registration. Needs C++11 compiler support for the C++ API. Supports theories and parameterized tests. Each test is run in its own process, so signals and crashes can be reported. Can output to multiple formats, like the TAP format or JUnit XML. Supported on Linux, OS X, FreeBSD, and Windows.
libcester	MIT	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	File	[59]	A robust header only unit testing framework for C and C++ programming language. Support function mocking, memory leak detection, crash report. Works on various platforms including embedded systems and compatible with various compilers. Outputs to multiple format like TAP, JUnitXML, TAPV13 or plain text.
crpcut		No	Yes		No	No		Yes	Yes	Suites within Suites	[110]	BSD 2 clause. Runs each test in its own process, guaranteeing that the test suite continues even in the event of an unexpected crash or infinite loop.
CUTE		Yes	Yes	No	No	Yes	Yes			Suites	[111]	CUTE (C++ Unit Testing Easier) with Eclipse CDT integration. Single line include, without inheritance. Mock support is provided by Mockator.
cutee		No	No	No	Yes						[112]	
CxxTest		Yes	Yes	Yes	No	Yes*	Optional	Yes	No	Suites	[113]	Uses a C++ parser and code generator (requiring Python) for test registration. * Has a framework for generating mocks of global functions, but not for generating mocks of objects.
doctest	MIT ^[114]	No	Yes	Yes	No	No	Yes	Yes	Yes	Yes	[115]	Light, feature rich C++ single header testing framework
Embunit		No				No	Yes				[53]	Commercial. Create unit tests for C/C++ and Embedded C++
Exercisix	BSD	No	No	No	No	No	Yes	Yes	Yes	Executables	[116]	Aimed to make adding tests as fast and easy as possible.
Fakelt	MIT					Yes					[117]	Use the latest C++11 features to create an expressive, yet very simple, API.
FCTX		Yes	Yes	Yes	No	No	No	Yes	No	Yes	[118]	Fast and complete unit testing framework all in one header. Declare and write your functions in one step. No dependencies. Cross platform.
Fructose		No	Yes	No	Yes	No	Yes	Yes	Yes	No	[119]	A simple unit test framework.

List of C++ Unit Testing Frameworks (Wikipedia)

Name	License	xUnit	Fixtures	Group fixtures	Generators	Mocks	Exceptions	Macros	Templates	Grouping	Source	Remarks
Google C++ Mocking Framework						Yes	No	Yes	Yes		[120]	
Google Test	BSD	Yes	Yes			Yes	Yes	Yes	Yes		[121]	Supports automatic test discovery, a rich set of assertions, user-defined assertions, death tests, fatal and non-fatal failures, various options for running the tests, and XML test report generation.
Hestia	MIT	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Suites	[122]	Open source. Can test servers, libraries, and applications, and embedded software. Outputs to stdout, text, html, or xml files. Has several assertions for messaging, warnings, and exceptions, as well as plain conditions.
Hippomocks						Yes	No	Yes	Yes		[123]	
Igloo			Yes (Contexts)	No	No	No	Yes	Yes	Yes	Yes (nested contexts)	[124]	BDD style unit testing in C++
lest		No	Yes	No	No	No	Yes	Yes	Yes	No	[125]	Tiny header-only C++11 test framework
liblittletest		Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	[126]	liblittletest is a portable, one file header-only C++ library for unit testing. Supports a rich set of assertions, automatic test discovering and various options for running the tests.
libunittest		Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	[127]	libunittest is a portable C++ library for unit testing making use of C++11.
mettle	BSD										[128]	
Microsoft Unit Testing Framework for C++	Proprietary	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	[129]	Commercial. Integrated into Microsoft Visual Studio 2012 IDE and later versions.
Mimicc	Proprietary					Yes					[61]	Fully automated mock generation for C and C++. Based on clang, provides the ability to compile header files straight into linkable mock object files and control them with an accompanying API.
Mockator						Yes	No	Yes	Yes		[130]	Header-only mock object library and an Eclipse plug-in to create test doubles in a simple yet powerful way; leverages new C++11 language facilities while still being compatible with C++03; has built-in support for CUTE
mock++/mockcpp		Yes	Yes		No	Yes	Yes	Yes	Yes	Suites	[131]	Simple testing framework for C++ (requires cmake)
mockitopp						Yes					[132]	A C++ mock object framework providing similar syntax to mockito for Java.
mockpp		Yes	Yes		Yes	Yes	Yes	Yes	Yes	Suites	[133]	A C++ mocking framework hosted by Google

...

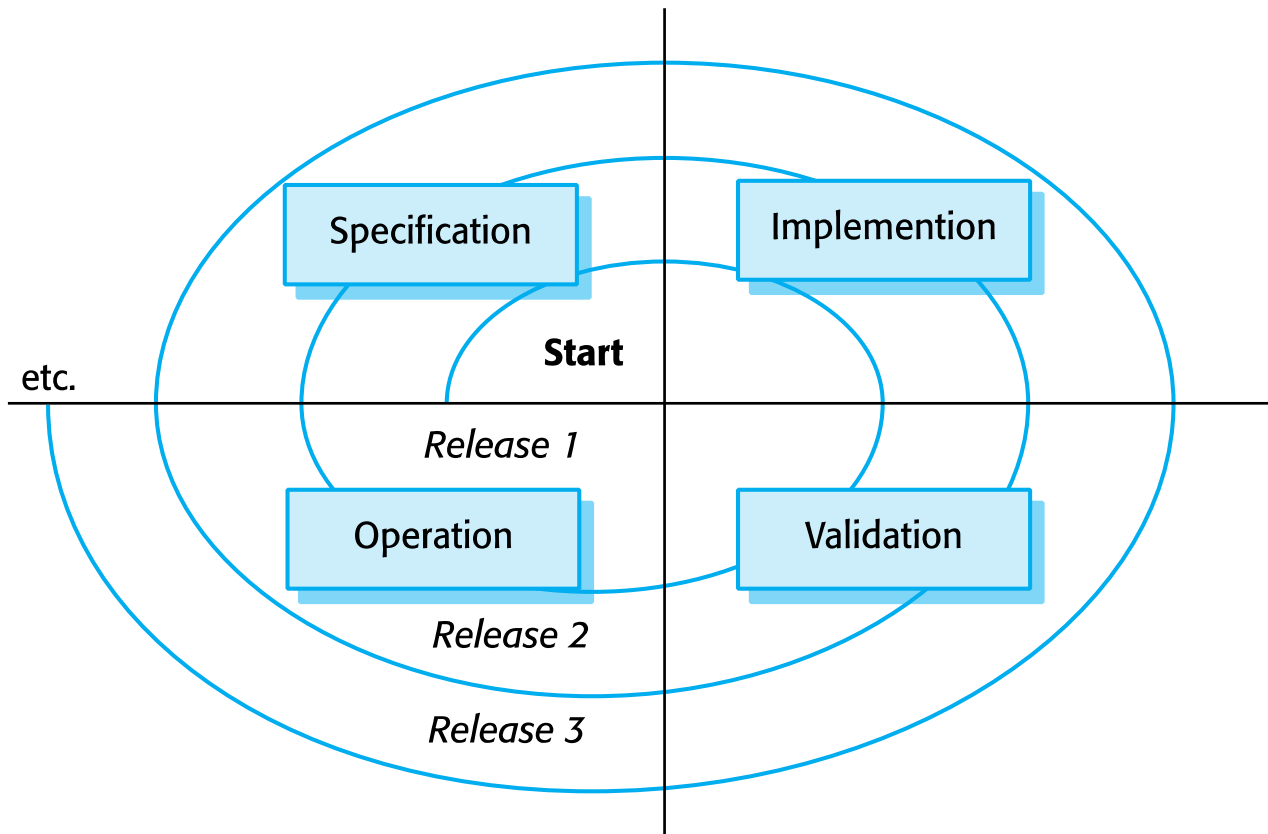
9. Software Evolution

Software Change

- **Software change** is inevitable.
 - New requirements emerge when the software is used.
 - The business environment changes.
 - Errors must be repaired.
 - New computers and equipment is added to the system.
 - The performance or reliability of the system may have to be improved.

- A key problem for all organizations is **implementing and managing change** to their existing software systems.
 - The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

A Spiral Model of Development and Evolution



Evolution and Servicing

- **Evolution**

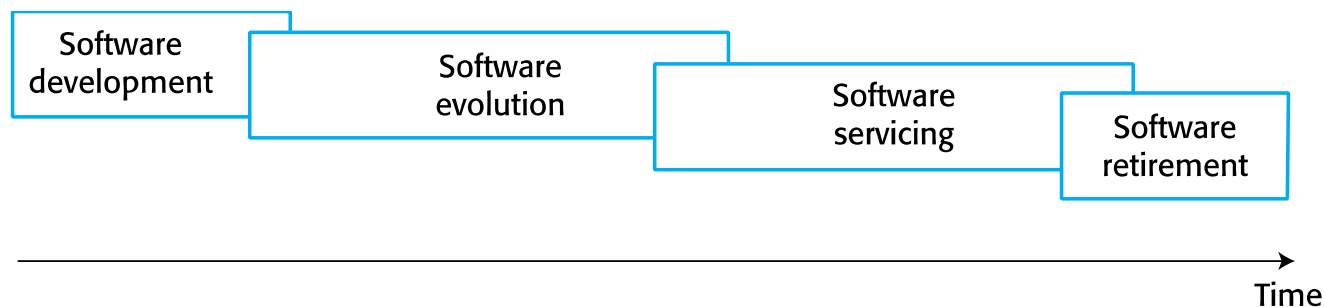
- The stage in a software system’s life cycle, where it is in operational use and is evolving as new requirements are proposed and implemented in the system.

- **Servicing**

- At this stage, the software remains useful, but the only changes made are those required to keep it operational, i.e., bug fixes and changes to reflect changes in the software’s environment.
- No new functionality is added.

- **Phase-out (Retirement)**

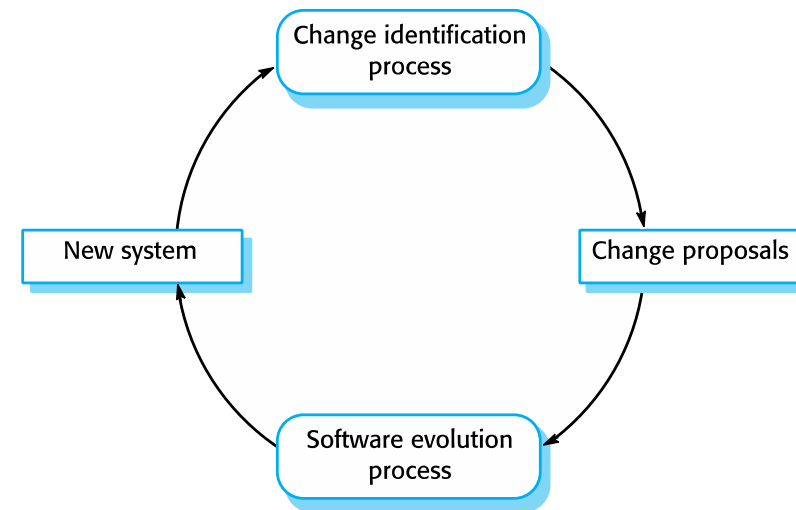
- The software may still be used but no further changes are made to it.



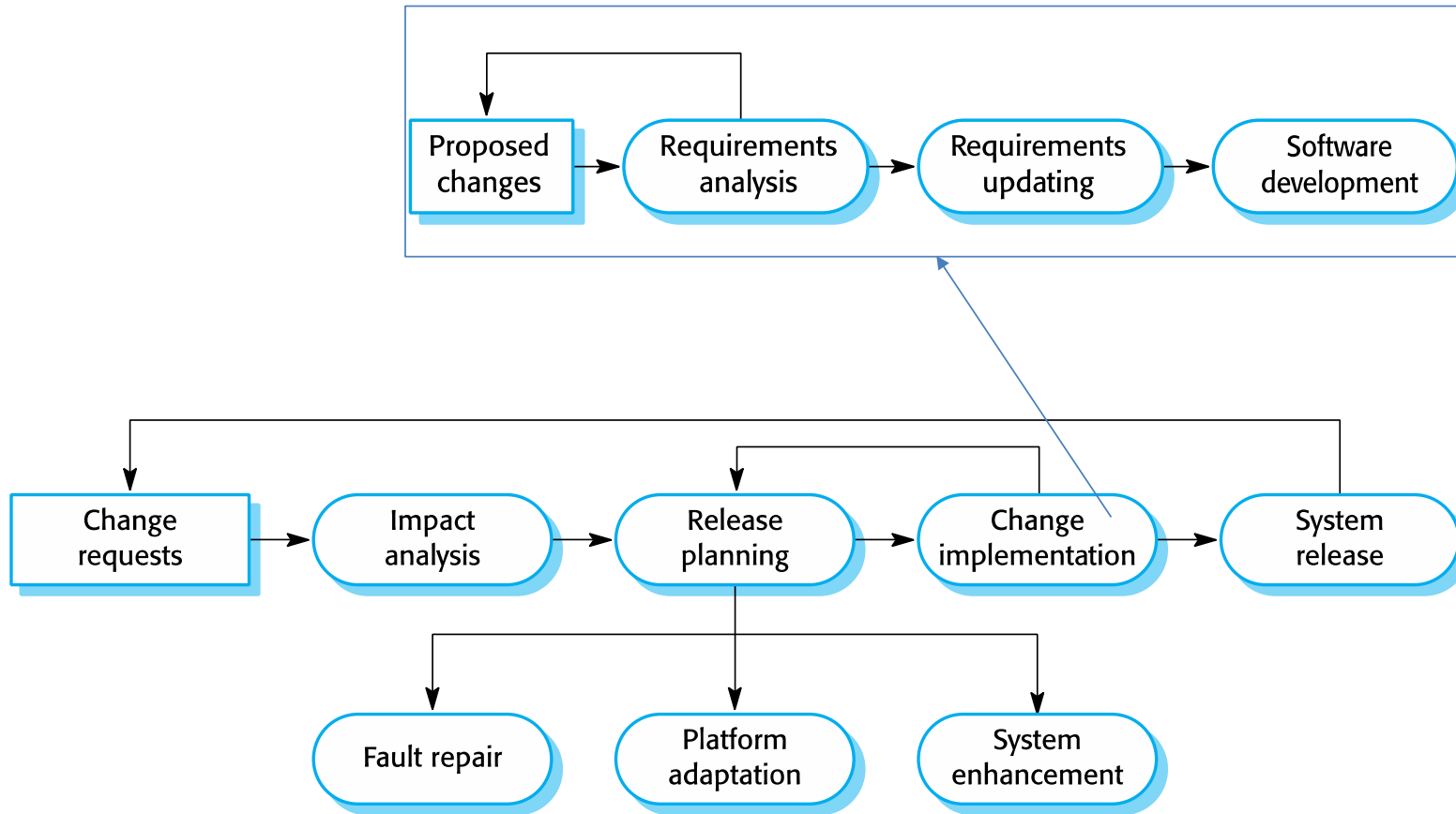
Evolution Processes

Evolution Processes

- **Software evolution** processes depend on
 - The type of software being maintained,
 - The development processes used, and
 - The skills and experience of the people involved.
- Proposals for change are the driver for system evolution.
 - Should be linked with components that are affected by the change
 - Should allow the cost and impact of the change to be estimated
- Change identification and evolution continues throughout the system lifetime.

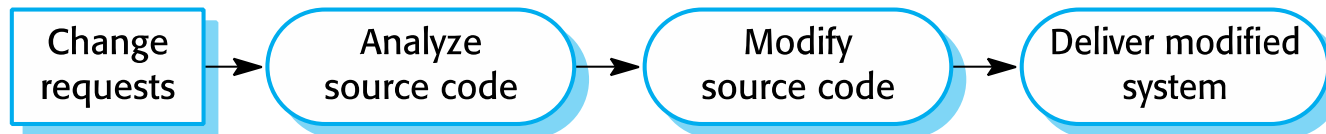


The Software Evolution Process



Urgent Change Requests

- **Urgent changes** may have to be implemented without going through all stages of the software engineering process.
 - If a serious system fault must be repaired to allow normal operation to continue.
 - If changes to the system's environment (e.g., OS upgrade) have unexpected effects.
 - If there are business changes that require a very rapid response (e.g., release of a competing product).



Agile Methods and Evolution

- **Agile methods** are based on incremental development so the transition from development to evolution is a **seamless** one.
 - Evolution is simply a continuation of the development process based on frequent system releases.
 - Automated regression testing is particularly valuable when changes are made to a system.
 - Changes may be expressed as additional user stories.
- Under the assumption that the Agile development teams have been maintained.
 - Should avoid **handover problems**

Handover Problems

- Where the development team have used an agile approach, but the evolution team is unfamiliar with agile methods and prefer a plan-based approach.
 - The evolution team may expect detailed documentation to support evolution, and this is not produced in agile processes.

- Where a plan-based approach has been used for development, but the evolution team prefer to use agile methods.
 - The evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.

Legacy Systems

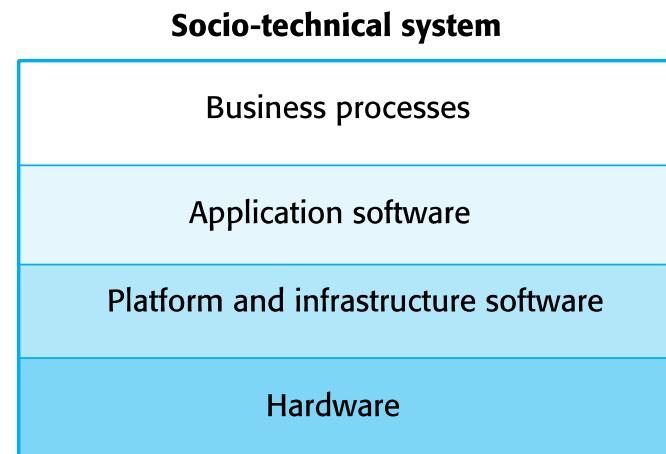
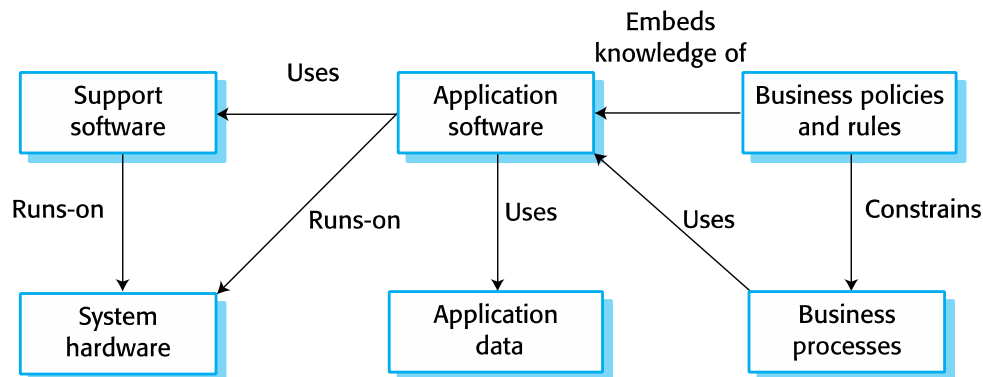
Legacy Systems

- **Legacy systems**

- **Older systems** that rely on languages and technology that are no longer used for new systems development.
 - May be dependent on older hardware such as mainframe computers
 - May have associated legacy processes and procedures

- Legacy systems are often **broader socio-technical systems**.

- Including hardware, software, libraries and other supporting software and business processes
- Elements of legacy systems:



Components of Legacy Systems

Element	Description
System hardware	Legacy systems may have been written for hardware that is no longer available.
Support software	The legacy system may rely on a range of support software, which may be obsolete or unsupported.
Application software	The application system that provides the business services is usually made up of a number of application programs.
Application data	These are data that are processed by the application system. They may be inconsistent, duplicated or held in different databases.
Business processes	These are processes that are used in the business to achieve some business objective. Business processes may be designed around a legacy system and constrained by the functionality that it provides
Business policies and rules	These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

Legacy System Replacement and Change

- **Legacy system replacement** is risky and expensive.
 - Because the system is still in use.
 - Many reasons
 - Lack of complete system specification
 - Tight integration of system and business processes
 - Undocumented business rules embedded in the legacy system
 - New software development may be late and/or over budget.

- **Legacy system change (modification)** is also expensive.
 - Many reasons
 - No consistent programming style
 - Use of obsolete programming languages with few people available with these language skills
 - Inadequate system documentation
 - System structure degradation
 - Program optimizations may make them hard to understand
 - Data errors, duplication and inconsistency

Legacy System Management

- Organizations relying on legacy systems should decide one strategy:
 - **Scrap the system completely** and modify business processes so that it is no longer required, or
 - **Continue maintaining** the system, or
 - **Transform the system by re-engineering** to improve its maintainability, or
 - **Replace the system** with a new system.

- **Legacy system assessment**
 - Assess the system quality and its business value to choose appropriate strategy

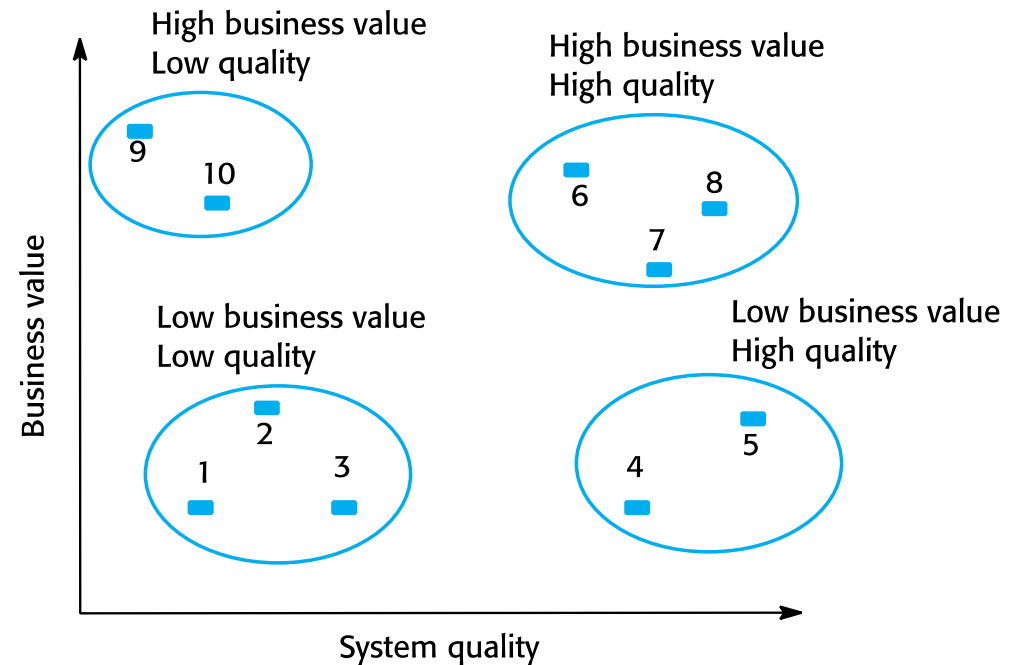
Legacy System Assessment

- **Legacy system assessment**

- Business value assessment
- System quality assessment

- **4 categories**

- **Low quality, low business value**
 - These systems should be scrapped
- **Low-quality, high-business value**
 - These make an important business contribution but are expensive to maintain
 - Should be re-engineered or replaced if a suitable system is available
- **High-quality, low-business value**
 - Replace with COTS, scrap completely or maintain
- **High-quality, high business value**
 - Continue in operation using normal system maintenance



Software Maintenance

Software Maintenance

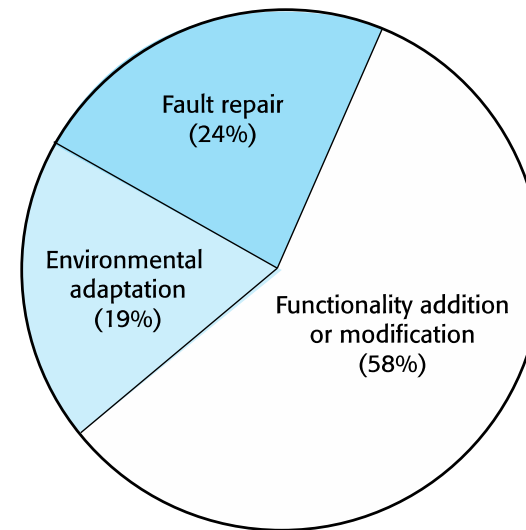
- **Software maintenance**
 - Modifying a program after it has been put into use
 - Mostly used for changing **custom software**
 - **Generic software** products are said to **evolve** to create new versions.
 - Changes are implemented by modifying existing components and adding new components to the system.
 - Not normally involve major changes to the system's architecture

Types of Maintenance

- **Fault repairs**
 - Changing a system to fix bugs/vulnerabilities and correct deficiencies in the way meets its requirements

- **Environmental adaptation**
 - Maintenance to adapt software to a different operating environment
 - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation

- **Functionality addition and modification**
 - Modifying the system to satisfy new requirements



Maintenance Costs

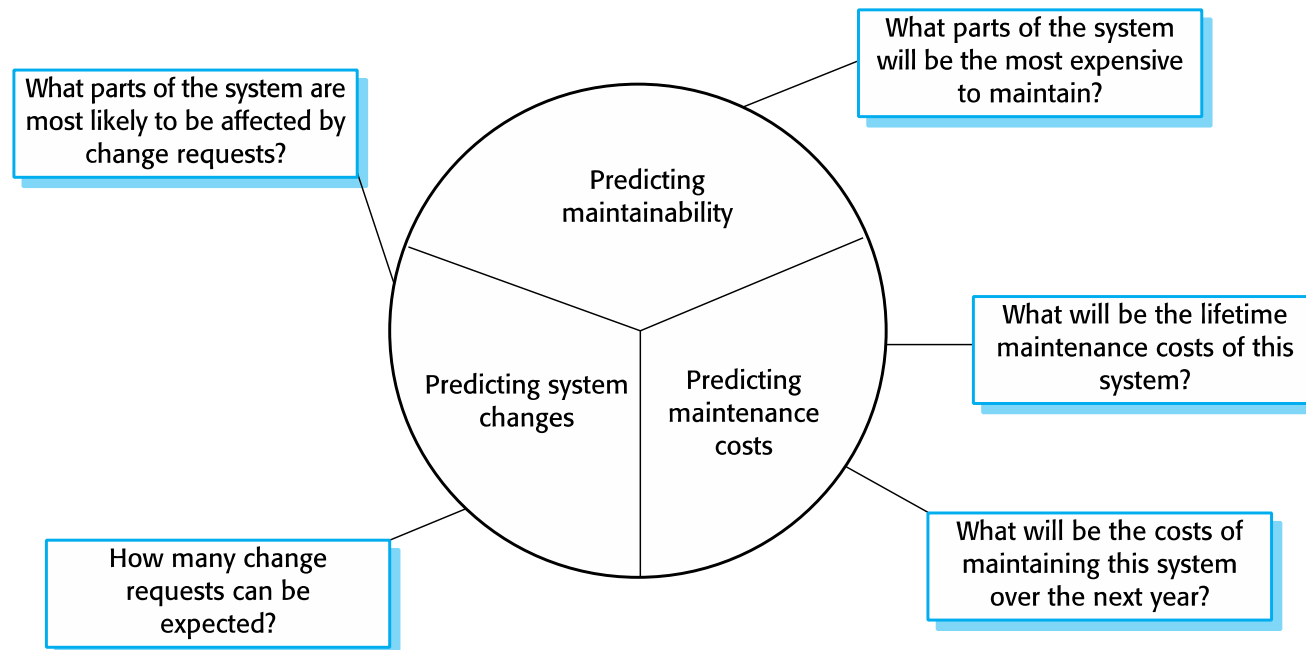
- **Maintenance costs are usually greater than development costs**
 - 2* to 100* depending on the application
 - Affected by both technical and non-technical factors

 - Increases as software is maintained
 - Since maintenance corrupts the software structure so makes further maintenance more difficult.

 - Aging software can have high support costs (e.g. old languages, compilers etc.).

Maintenance Prediction

- **Maintenance prediction** is concerned with assessing which parts of the system may cause problems and have high maintenance costs.
 - Change acceptance depends on the maintainability of the components affected by the change.
 - Implementing changes degrades the system and reduces its maintainability.
 - Maintenance costs depend on the number of changes and costs of change depend on maintainability.



Change Prediction

- **Change prediction**
 - Predicting the number of changes requires
 - Predicting understanding of the relationships between a system and its environment
 - Tightly coupled systems require changes whenever the environment is changed

- Factors influencing this relationship are
 - Number and complexity of system interfaces
 - Number of inherently volatile system requirements
 - The business processes where the system is used

Metrics for Change Prediction

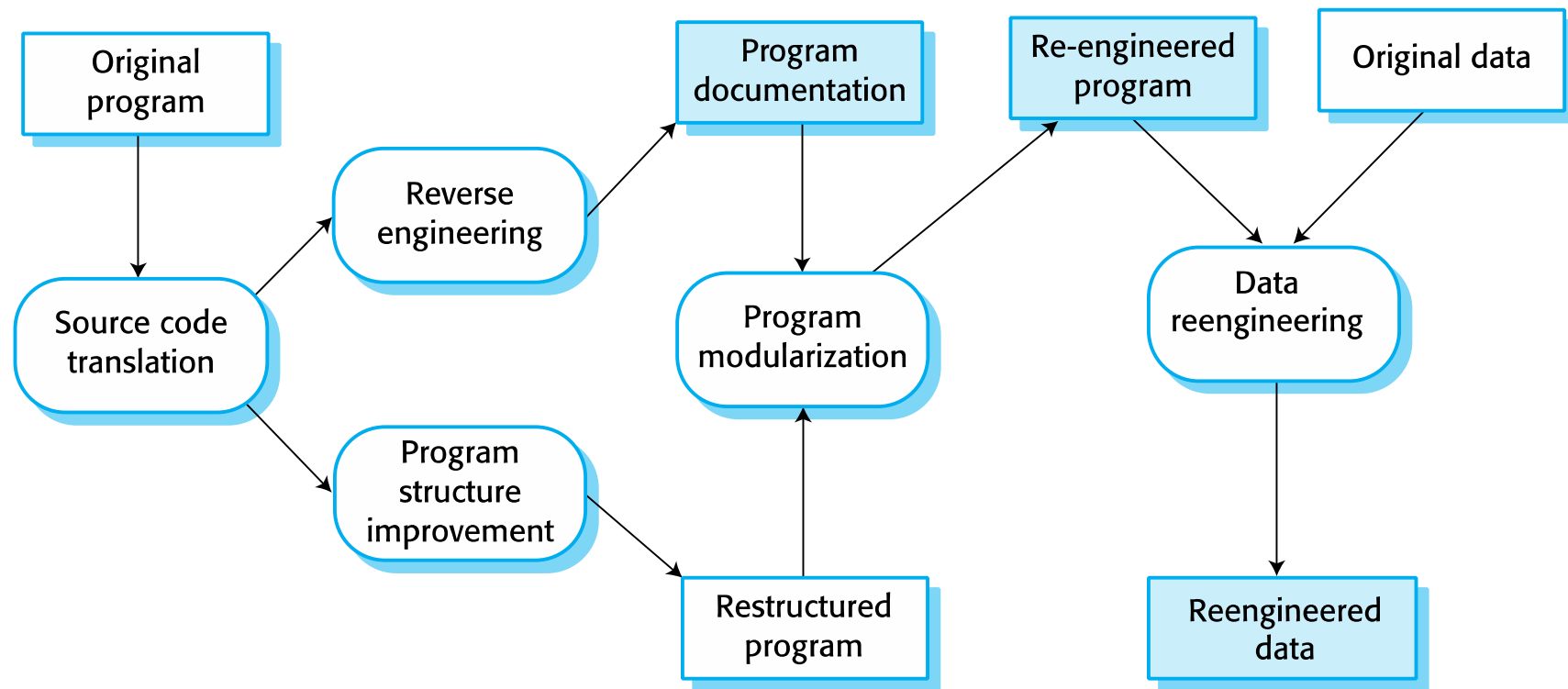
- **Process metrics** may be used to assess maintainability
 - If any or all of these is increasing, this may indicate a decline in maintainability.
 - Number of requests for corrective maintenance
 - Average time required for impact analysis
 - Average time taken to implement a change request
 - Number of outstanding change requests

- **Complexity metrics** of system components may be used to assess maintainability.
 - Studies have shown that most maintenance effort is spent on a relatively small number of system components.
 - Complexity of control structures
 - Complexity of data structures
 - Object, method (procedure) and module size

Software Reengineering

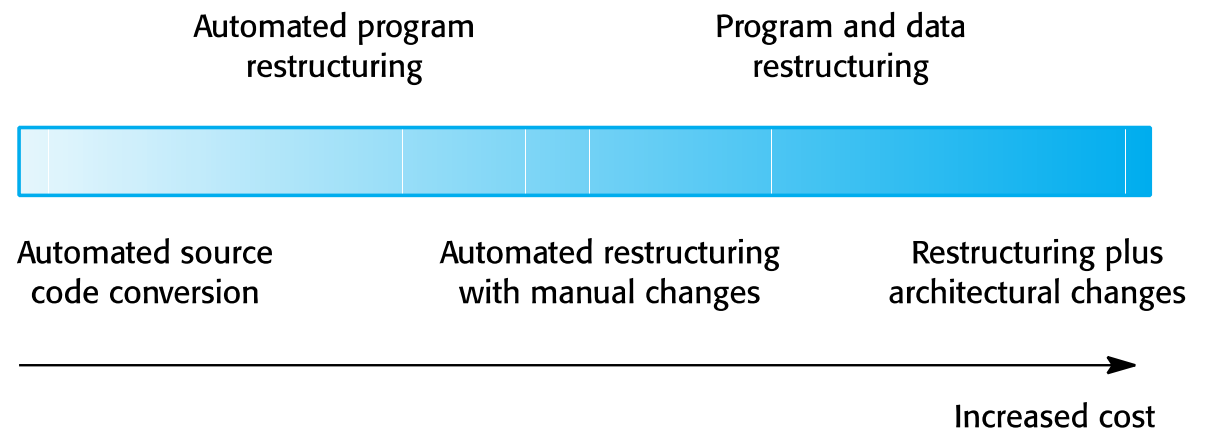
- **Reengineering:** Restructuring or rewriting parts or all of a legacy system without changing its functionality
 - Applicable where some but not all sub-systems of a larger system require frequent maintenance
 - Involves adding effort to make them easier to maintain
 - The system may be re-structured and re-documented.
 - Advantages
 - Reduced risk: There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
 - Reduced cost: The cost of re-engineering is often significantly less than the costs of developing new software.

The Reengineering Process



Reengineering Process Activities

- **Source code translation**
 - Convert code to a new language
- **Reverse engineering**
 - Analyze the program to understand it
- **Program structure improvement**
 - Restructure automatically for understandability
- **Program modularization**
 - Reorganize the program structure
- **Data reengineering**
 - Clean-up and restructure system data



Refactoring

- **Refactoring**: The process of making improvements to a program to slow down degradation through change
 - ‘Preventative maintenance’ that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
 - When you refactor a program, you should not add functionality but rather concentrate on program improvement.

Refactoring and Reengineering

- **Re-engineering** takes place after a system has been maintained for some time and maintenance costs are increasing.
 - Use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable

- **Refactoring** is a continuous process of improvement throughout the development and evolution process.
 - To avoid the structure and code degradation that increases the costs and difficulties of maintaining a system

'Bad smells' in Program Code

- **Duplicate code**
 - The same or very similar code may be included at different places in a program.
 - This can be removed and implemented as a single method or function that is called as required.
- **Long methods**
 - If a method is too long, it should be redesigned as a number of shorter methods.
- **Switch (case) statements**
 - These often involve duplication, where the switch depends on the type of a value.
 - The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.
- **Data clumping**
 - Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program.
 - These can often be replaced with an object that encapsulates all of the data.
- **Speculative generality**
 - This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

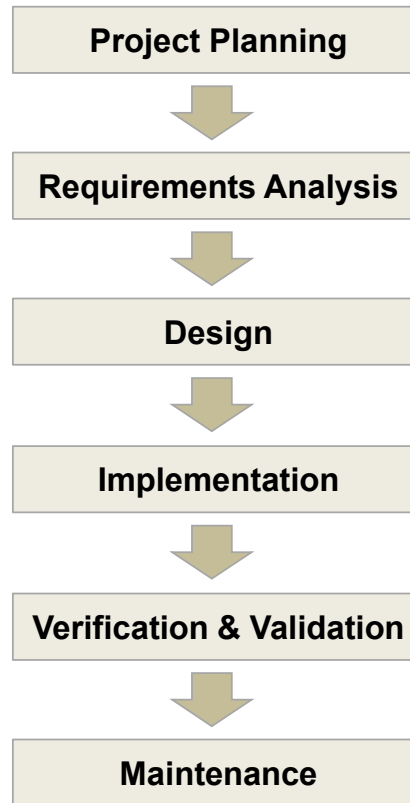


Homework #12

- Clean Code에 대해 조사하고 A4 5장 (글자크기 10 이하)으로 정리하세요.
- C 코드用 Static Code Analysis 도구를 하나 선택한 후, Homework #7에서 개발한 C Program에 적용하세요.
정리하세요.
 - <https://github.com/analysis-tools-dev/static-analysis>
- 분석 결과와 반영 내용 (前/後)을 A4 5장 (글자크기 10 이하)으로 정리하세요.

Summary

Software Engineering



Software Development Life-Cycle Processes
 - Plan-driven (Waterfall) vs. Agile (Iterative)
 - SASD vs. OOAD

Requirements Engineering
 - SRS by IEEE Std 830-1998
 - Spec. Review

Structured Analysis for RVC
Writing an SRS

Architecture Design
 - AD (Architecture Description)
Detailed Design
 - Models with UML Diagrams

Object-Oriented Analysis for RVC
Object-Oriented Design for RVC

Reviews
Levels of Testing
 - Unit / Component / System / Release / User

Reuse
 - COTS and Open-Source SW
 Project / Configuration Management

Software Maintenance
 Legacy System
 Reengineering, Reverse Engineering, Refactoring