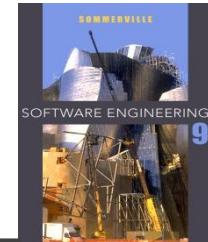


Chapter 1- Introduction to Software Engineering

Frequently Asked Questions about Software Engineering



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

Frequently Asked Questions about Software Engineering

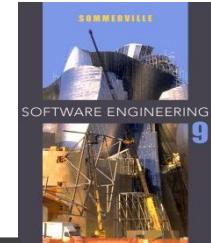


Question	Answer
<i>What are the key challenges facing software engineering?</i>	<i>Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.</i>
<i>What are the costs of software engineering?</i>	<i>Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.</i>
<i>What are the best software engineering techniques and methods?</i>	<i>While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.</i>
<i>What differences has the web made to software engineering?</i>	<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 has led to important advances in programming languages and software reuse.</i>

Essential Attributes of Good Software

Often referred to as "Quality Metrics"

Sometimes called "Non-Functional Requirements"

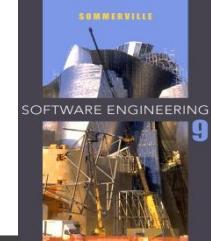


Product characteristic	Description
Maintainability	<i>Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.</i>
Dependability and security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

Essential Attributes of Good Software

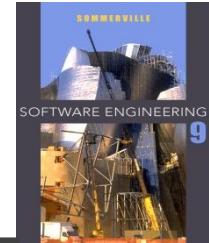
Often referred to as "Quality Metrics"

Sometimes called "Non-Functional Requirements"



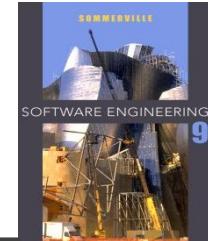
- ⑩ More:
- ⑩ Many other quality metrics such as
 - ⑩ Reliability
 - ⑩ Scalability
 - ⑩ Portability
 - ⑩ Reusability
 - ⑩ Useability

Software Engineering



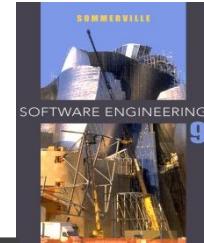
- ⑩ **Software engineering** is an **engineering discipline** that is concerned with **all** aspects of software production from the early stages of system **specification** through to **maintaining** the system after it has gone into use.
- ⑩ **Engineering discipline**
 - ⑩ Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints.
- ⑩ **All aspects of software production**
 - ⑩ Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.

Software Process Activities (High level view)



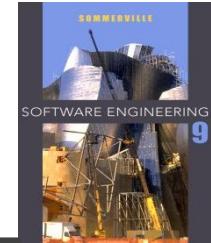
- ⑩ **Software specifications**, where customers and engineers define the software that is to be produced and the constraints on its operation.
- ⑩ **Software development**, where the software is designed and programmed.
- ⑩ **Software validation**, where the software is checked to ensure that it is what the customer requires.
- ⑩ **Software evolution**, where the software is modified to reflect changing customer and market requirements.

Software Engineering Diversity



- ⑩ There are many different types of software system and there is **no universal set of software techniques** that is applicable to all of these.
- ⑩ 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.
- ⑩ Many companies do not subscribe exactly to any specific process. **Rather, they use a hybrid of activities that 'fits' their organization, tools, experience, and frankly what 'works for them.'**

Key examples of diverse systems



Stand alone apps

Interactive transaction-based apps

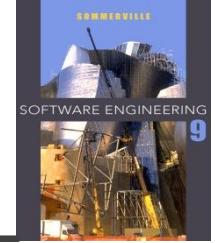
Embedded real time systems

Process control systems

Batch systems

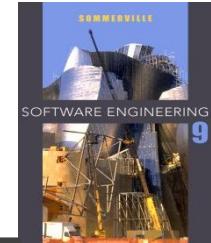
Data collection systems. And more...

Software Engineering Fundamentals



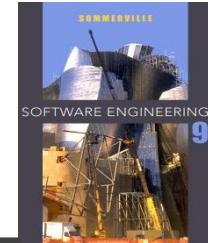
- ⑩ Some fundamental principles apply to all types of software system
 - ⑩ Systems should be developed using a managed and understood development process.
- ⑩ Dependability and performance are important for all systems. But there are many **other quality metrics needed!!**
- ⑩ Understanding and managing the **software specification** and **requirements** (what the software should do) are **vitally** important.
- ⑩ Where appropriate, you should **reuse software** that has already been developed rather than write new software.

Software Engineering and the Web



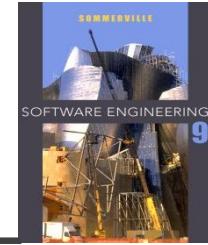
- ⑩ **The Web** is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- ⑩ **Web services** allow application functionality to be accessed **globally**
- ⑩ We must consider our developments global by default.
- ⑩ **Cloud computing** is an approach to the provision of computer services where applications run remotely on the 'cloud'.
 - ⑩ Users do not buy software but pay according to use.

Web Software Engineering



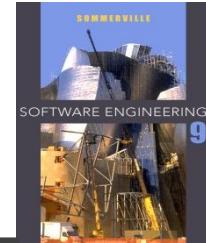
- ⑩ **Software reuse** is the dominant approach for constructing web-based systems.
 - ⑩ When building these systems, you think about how you can assemble them from **pre-existing software components** and systems.
- ⑩ Web-based systems should be developed and delivered **iteratively and incrementally**.
 - ⑩ It is now generally recognized that it is **impractical** to specify all the requirements for such systems in advance.
- ⑩ **User interfaces are constrained** by the capabilities of web browsers.
 - ⑩ Technologies such as AJAX allow rich interfaces to be created within a web browser but are still difficult to use.
 - ⑩ Web forms with local scripting are more commonly used.

Software Engineering Ethics



- ⑩ Software engineering involves **wider responsibilities** than simply the application of technical skills.
- ⑩ This is dramatically different than software development years ago.
- ⑩ Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals.
- ⑩ Ethical behavior is **more** than simply upholding the law but involves following a set of principles that are morally correct.

Issues of Professional Responsibility



⑩ Confidentiality

- ⑩ Engineers should normally **respect the confidentiality of their employers** or clients irrespective of whether or not a formal confidentiality agreement has been signed.
- ⑩ When being fired, often employee is not allowed to return to his/her desk!

⑩ Competence

- ⑩ Engineers should not misrepresent their level of competence. They should not knowingly accept work which is out with their competence.

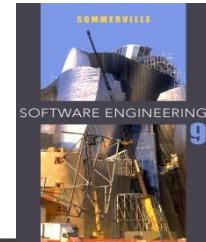
⑩ Intellectual property rights

- ⑩ Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.

⑩ Computer misuse

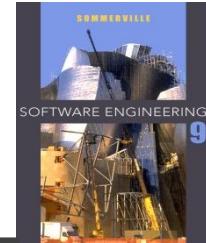
- ⑩ Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

ACM/IEEE Code of Ethics



- ⑩ The professional societies in the US have cooperated to produce a code of ethical practice.
- ⑩ We have these posted along our inner hallways.
- ⑩ Members of these organizations sign up to the code of practice when they join.
- ⑩ Absolutely you need to belong to the ACM, IEEE CS and perhaps a couple of others in your area of specialization.
- ⑩ The Code contains eight Principles related to the behavior of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

The ACM/IEEE Code of Ethics



Software Engineering Code of Ethics and Professional Practice

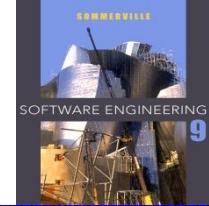
ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

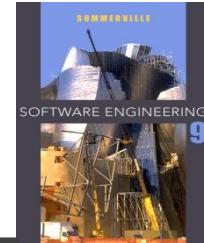
Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

Ethical Principles

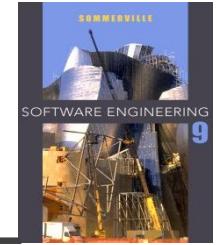


1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Ethical Dilemmas



- ⑩ Disagreement in principle with the policies of senior management. **Discuss**
- ⑩ Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system. **Discuss**
- ⑩ Participation in the development of military weapons systems or nuclear systems. **Discuss**

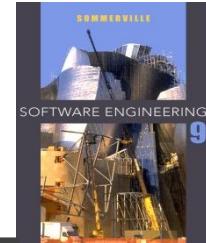


Here in the following slides are some introductory thoughts regarding architecture.

We will be taking through many of these in future lectures.

But for now...

Essential high-level requirements (sometimes called User Stories or Features)



The system shall be available to deliver insulin when required.

The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

The system must therefore be designed and implemented to ensure that the system always meets these requirements.

These (above) came from the book, but clearly they are way to high level to provide any real direction to development. We will discuss mechanisms to capture functional and non-functional requirements in lectures coming up.



Chapter 2 – Software Processes



Topics covered

- ✧ Software process models
- ✧ Process activities
- ✧ Coping with change
- ✧ Process improvement



The software process

- ✧ 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.
- ✧ A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

Software process descriptions



- ✧ When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.
- ✧ Process descriptions may also include:
 - Products, which are the outcomes of a process activity;
 - Roles, which reflect the responsibilities of the people involved in the process;
 - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.



Plan-driven and agile processes

- ✧ Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- ✧ In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- ✧ In practice, most practical processes include elements of both plan-driven and agile approaches.
- ✧ There are no right or wrong software processes.



Software process models



Software process models

✧ The waterfall model

- Plan-driven model. Separate and distinct phases of specification and development.

✧ Incremental development

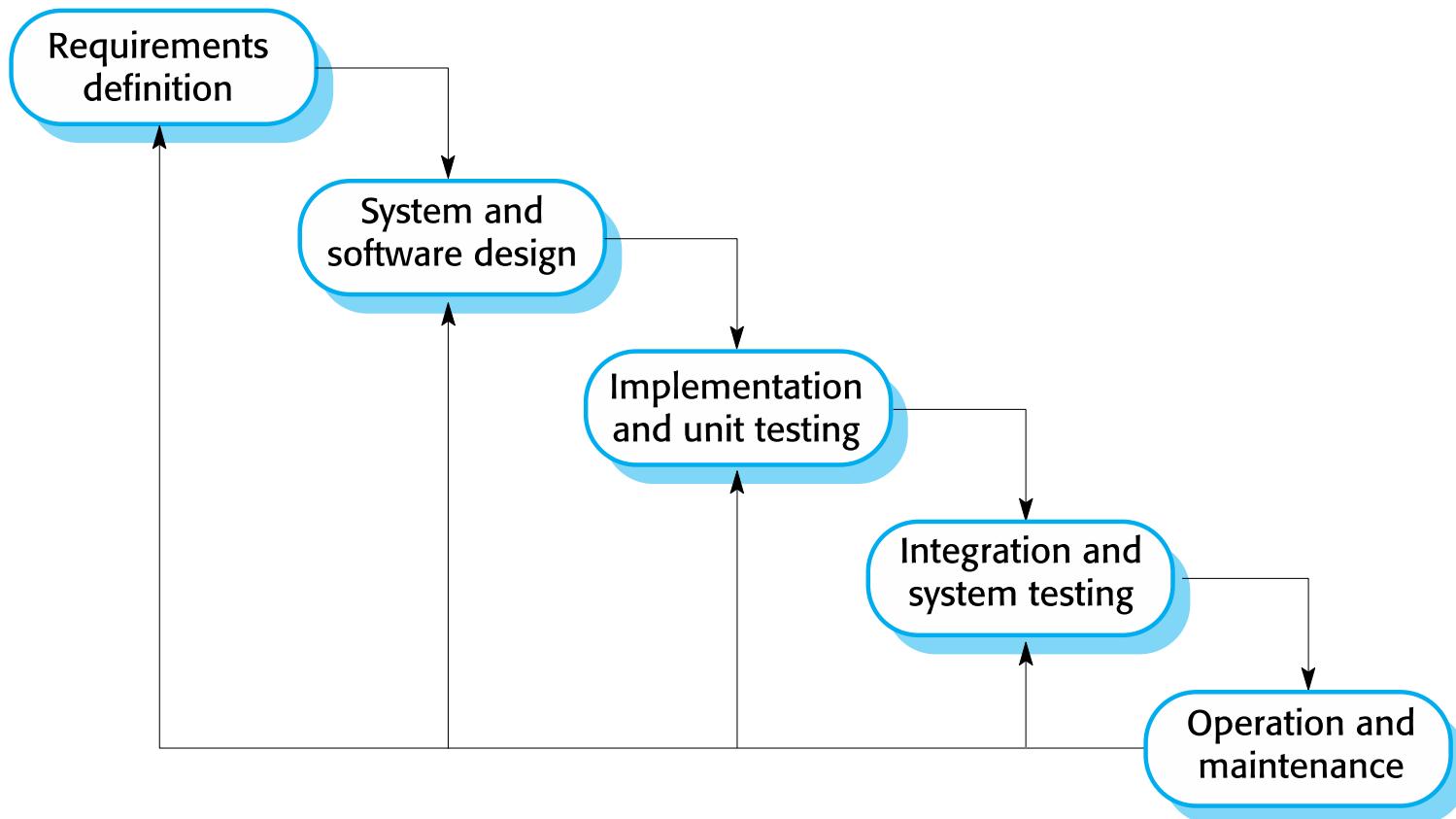
- Specification, development and validation are interleaved. May be plan-driven or agile.

✧ Integration and configuration

- The system is assembled from existing configurable components. May be plan-driven or agile.

✧ In practice, most large systems are developed using a process that incorporates elements from all of these models.

The waterfall model





Waterfall model phases

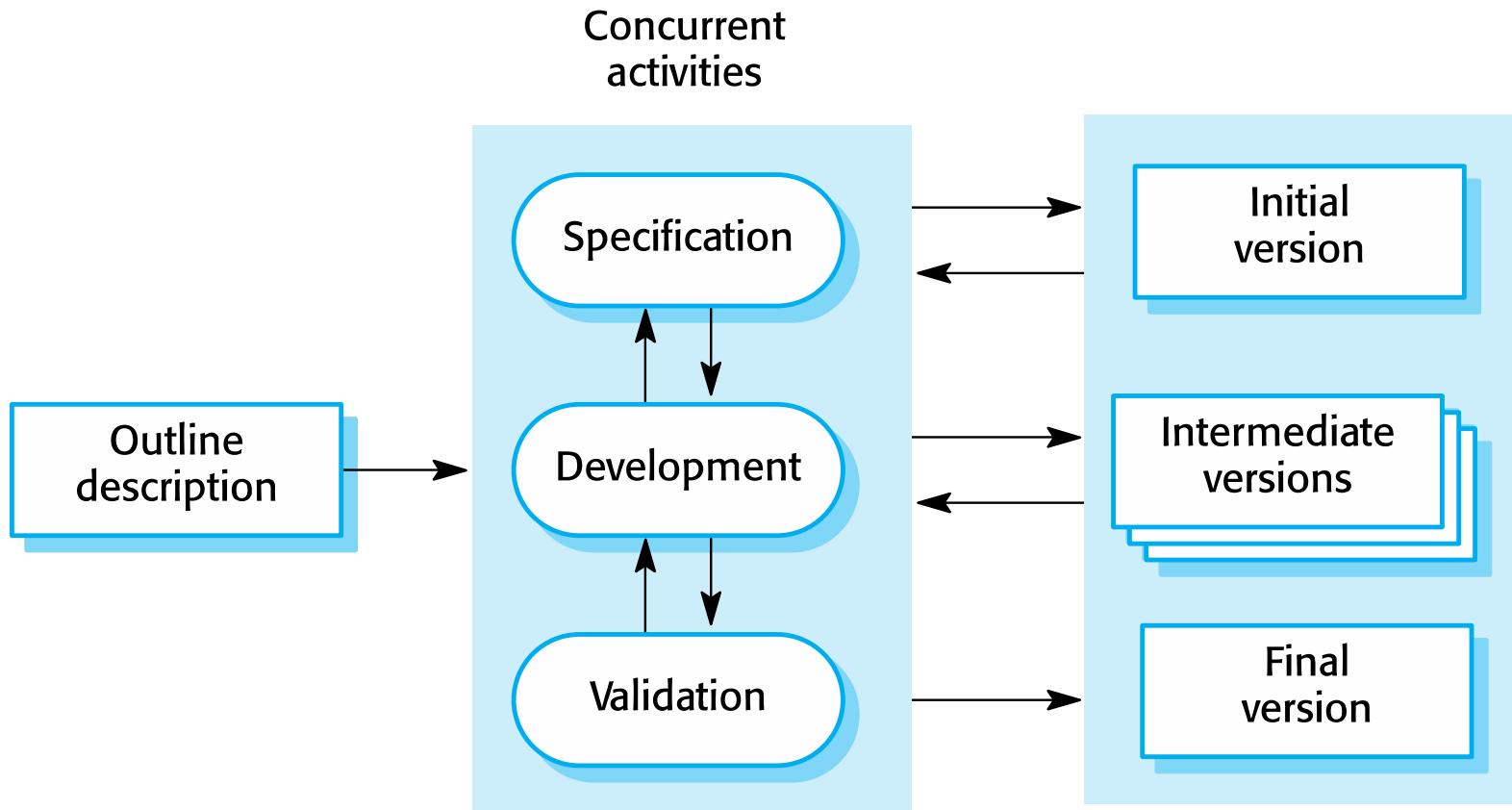
- ✧ There are separate identified phases in the waterfall model:
 - Requirements analysis and definition
 - System and software design
 - Implementation and unit testing
 - Integration and system testing
 - Operation and maintenance
- ✧ The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.



Waterfall model problems

- ✧ Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
 - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
 - Few business systems have stable requirements.
- ✧ The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
 - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

Incremental development





Incremental development benefits

- ✧ The cost of accommodating changing customer requirements is reduced.
 - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- ✧ It is easier to get customer feedback on the development work that has been done.
 - Customers can comment on demonstrations of the software and see how much has been implemented.
- ✧ More rapid delivery and deployment of useful software to the customer is possible.
 - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.



Incremental development problems

- ✧ The process is not visible.
 - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- ✧ System structure tends to degrade as new increments are added.
 - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.



Integration and configuration

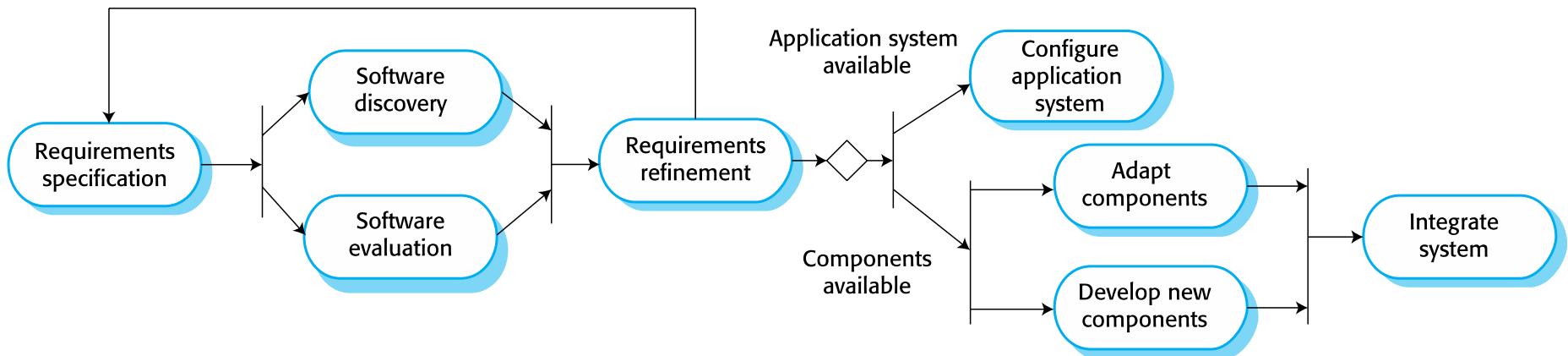
- ✧ Based on software reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- ✧ Reused elements may be configured to adapt their behaviour and functionality to a user's requirements
- ✧ Reuse is now the standard approach for building many types of business system
 - Reuse covered in more depth in Chapter 15.



Types of reusable software

- ✧ Stand-alone application systems (sometimes called COTS) that are configured for use in a particular environment.
- ✧ Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
- ✧ Web services that are developed according to service standards and which are available for remote invocation.

Reuse-oriented software engineering





Key process stages

- ✧ Requirements specification
- ✧ Software discovery and evaluation
- ✧ Requirements refinement
- ✧ Application system configuration
- ✧ Component adaptation and integration



Advantages and disadvantages

- ✧ Reduced costs and risks as less software is developed from scratch
- ✧ Faster delivery and deployment of system
- ✧ But requirements compromises are inevitable so system may not meet real needs of users
- ✧ Loss of control over evolution of reused system elements



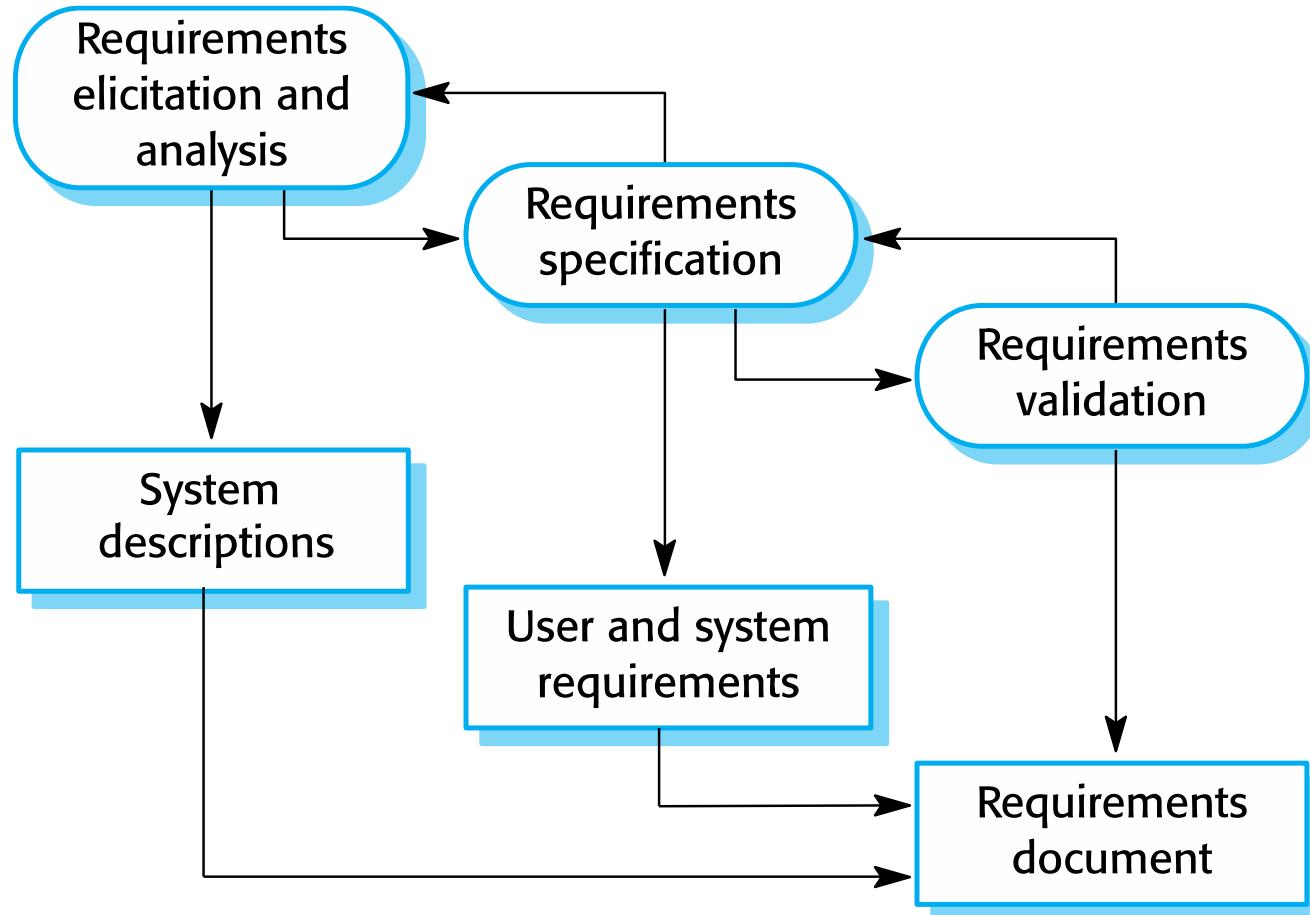
Process activities



Process activities

- ✧ Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- ✧ The four basic process activities of specification, development, validation and evolution are organized differently in different development processes.
- ✧ For example, in the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved.

The requirements engineering process





Software specification

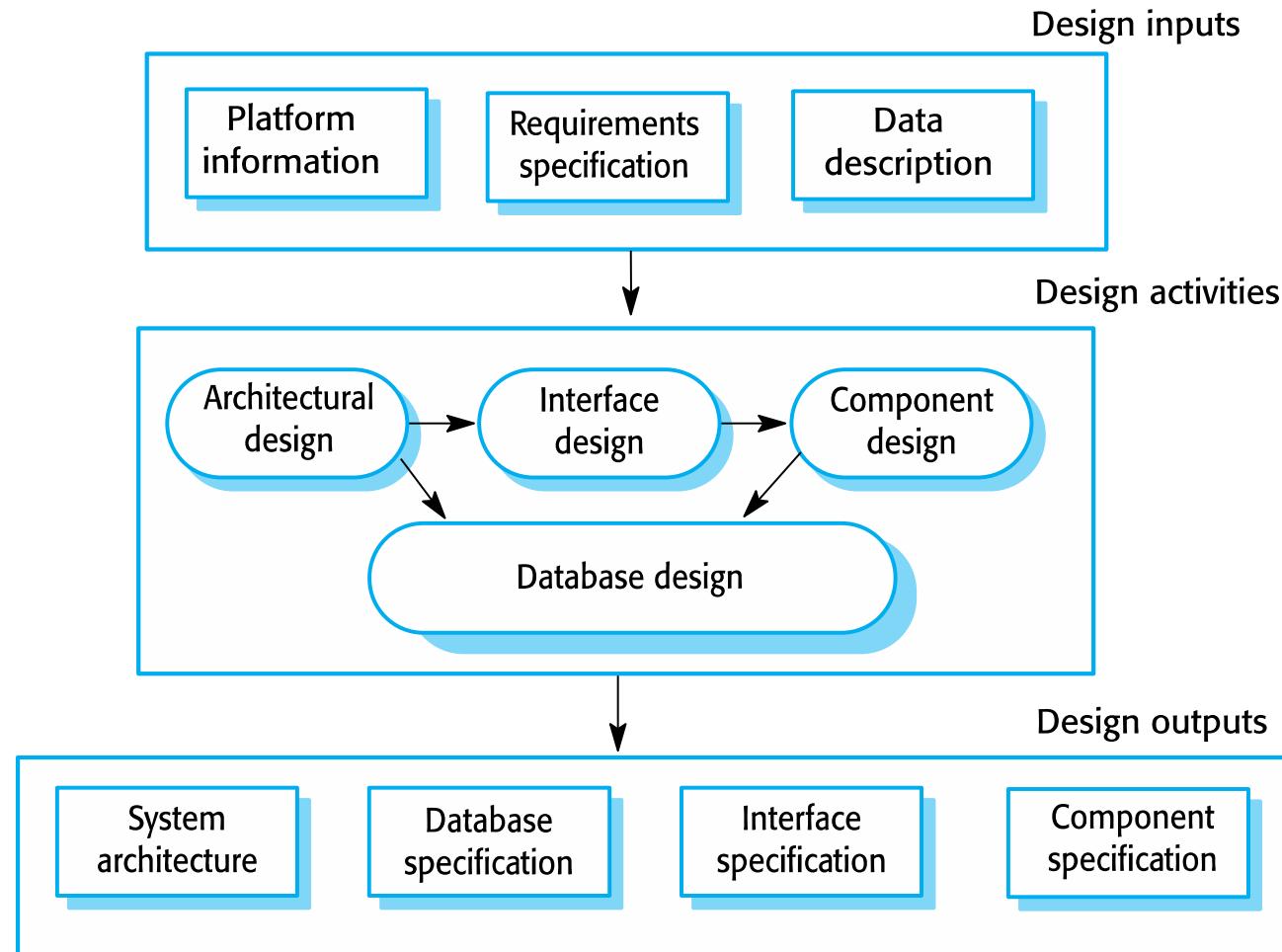
- ✧ The process of establishing what services are required and the constraints on the system's operation and development.
- ✧ Requirements engineering process
 - Requirements elicitation and analysis
 - What do the system stakeholders require or expect from the system?
 - Requirements specification
 - Defining the requirements in detail
 - Requirements validation
 - Checking the validity of the requirements

Software design and implementation



- ✧ The process of converting the system specification into an executable system.
- ✧ Software design
 - Design a software structure that realises the specification;
- ✧ Implementation
 - Translate this structure into an executable program;
- ✧ The activities of design and implementation are closely related and may be inter-leaved.

A general model of the design process





Design activities

- ✧ *Architectural design*, where you identify the overall structure of the system, the principal components (subsystems or modules), their relationships and how they are distributed.
- ✧ *Database design*, where you design the system data structures and how these are to be represented in a database.
- ✧ *Interface design*, where you define the interfaces between system components.
- ✧ *Component selection and design*, where you search for reusable components. If unavailable, you design how it will operate.

System implementation



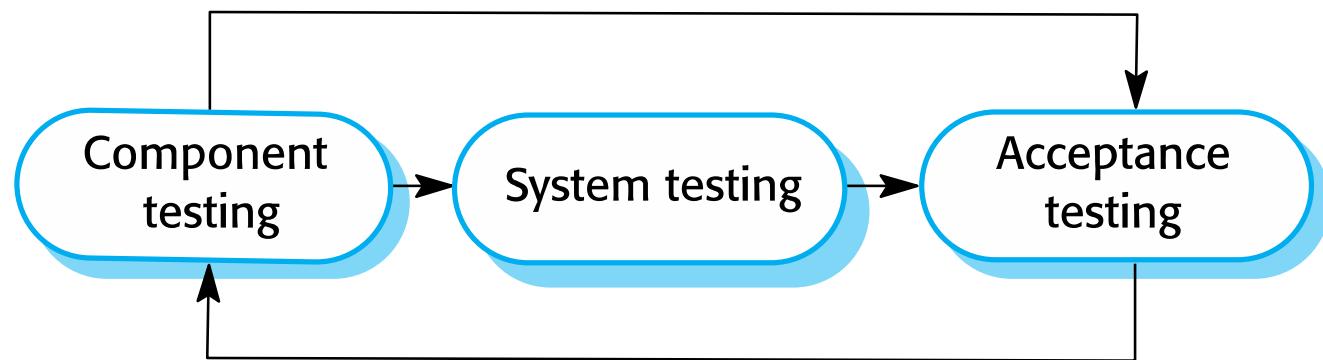
- ✧ The software is implemented either by developing a program or programs or by configuring an application system.
- ✧ Design and implementation are interleaved activities for most types of software system.
- ✧ Programming is an individual activity with no standard process.
- ✧ Debugging is the activity of finding program faults and correcting these faults.



Software validation

- ✧ Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- ✧ Involves checking and review processes and system testing.
- ✧ System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
- ✧ Testing is the most commonly used V & V activity.

Stages of testing





Testing stages

✧ Component testing

- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.

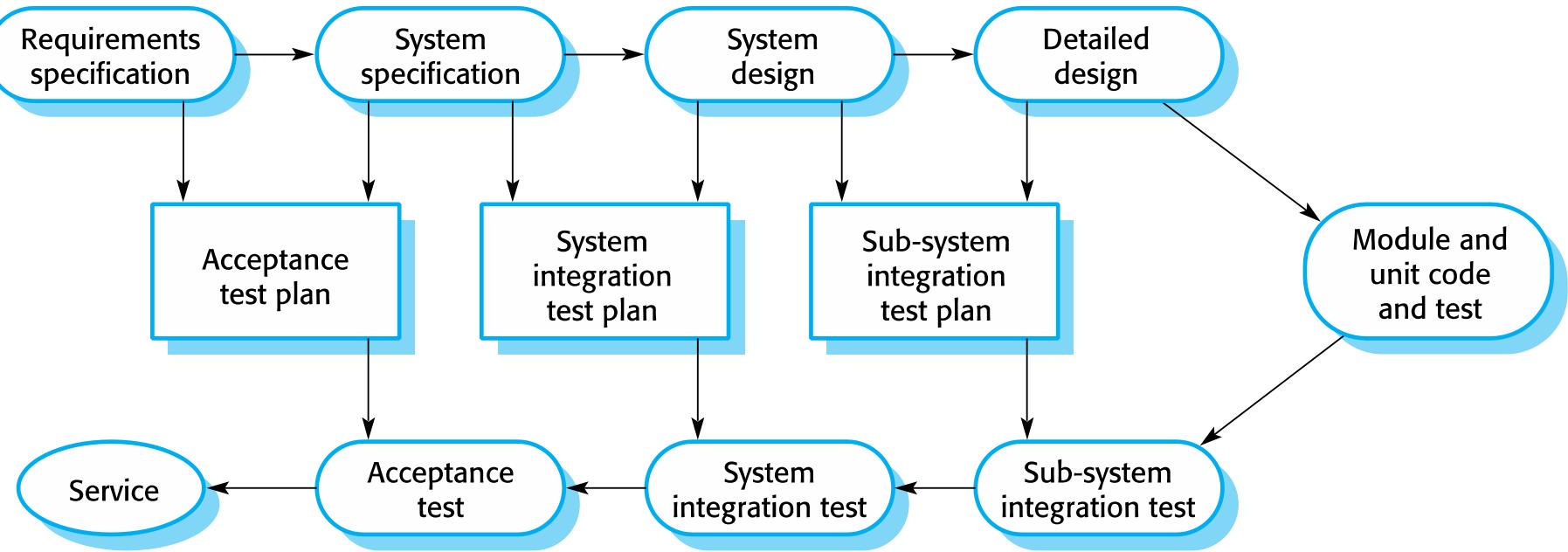
✧ System testing

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

✧ Customer testing

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

Testing phases in a plan-driven software process (V-model)

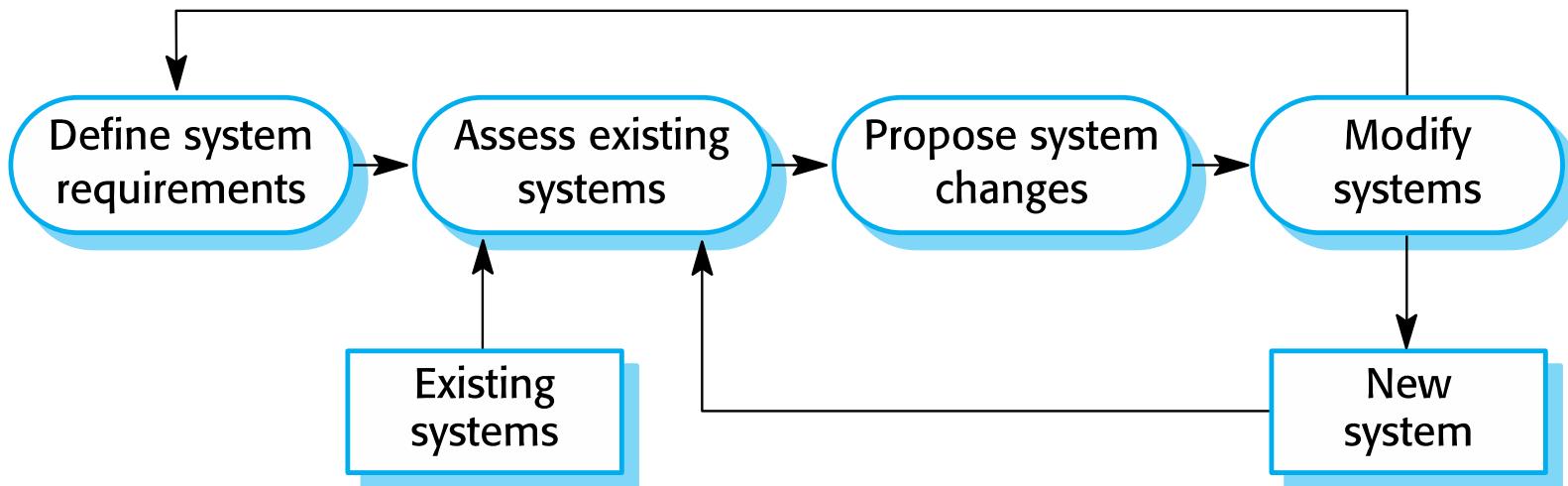




Software evolution

- ✧ Software is inherently flexible and can change.
- ✧ As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- ✧ Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

System evolution





Coping with change



Coping with change

- ✧ Change is inevitable in all large software projects.
 - Business changes lead to new and changed system requirements
 - New technologies open up new possibilities for improving implementations
 - Changing platforms require application changes
- ✧ Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality



Reducing the costs of rework

- ✧ Change anticipation, where the software process includes activities that can anticipate possible changes before significant rework is required.
 - For example, a prototype system may be developed to show some key features of the system to customers.
- ✧ Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost.
 - This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have to be altered to incorporate the change.

Coping with changing requirements



- ✧ System prototyping, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions. This approach supports change anticipation.
- ✧ Incremental delivery, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance.

Software prototyping



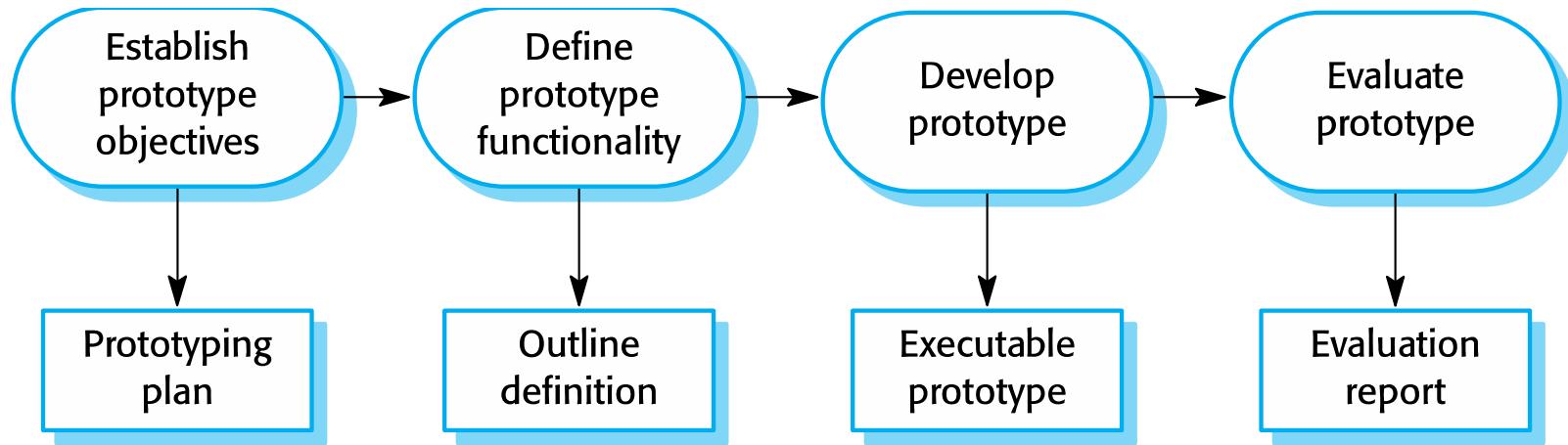
- ✧ A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- ✧ A prototype can be used in:
 - The requirements engineering process to help with requirements elicitation and validation;
 - In design processes to explore options and develop a UI design;
 - In the testing process to run back-to-back tests.



Benefits of prototyping

- ✧ Improved system usability.
- ✧ A closer match to users' real needs.
- ✧ Improved design quality.
- ✧ Improved maintainability.
- ✧ Reduced development effort.

The process of prototype development





Prototype development

- ✧ May be based on rapid prototyping languages or tools
- ✧ May involve leaving out functionality
 - Prototype should focus on areas of the product that are not well-understood;
 - Error checking and recovery may not be included in the prototype;
 - Focus on functional rather than non-functional requirements such as reliability and security



Throw-away prototypes

- ✧ Prototypes should be discarded after development as they are not a good basis for a production system:
 - It may be impossible to tune the system to meet non-functional requirements;
 - Prototypes are normally undocumented;
 - The prototype structure is usually degraded through rapid change;
 - The prototype probably will not meet normal organisational quality standards.



Incremental delivery

- ✧ Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- ✧ User requirements are prioritised and the highest priority requirements are included in early increments.
- ✧ Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.



Incremental development and delivery

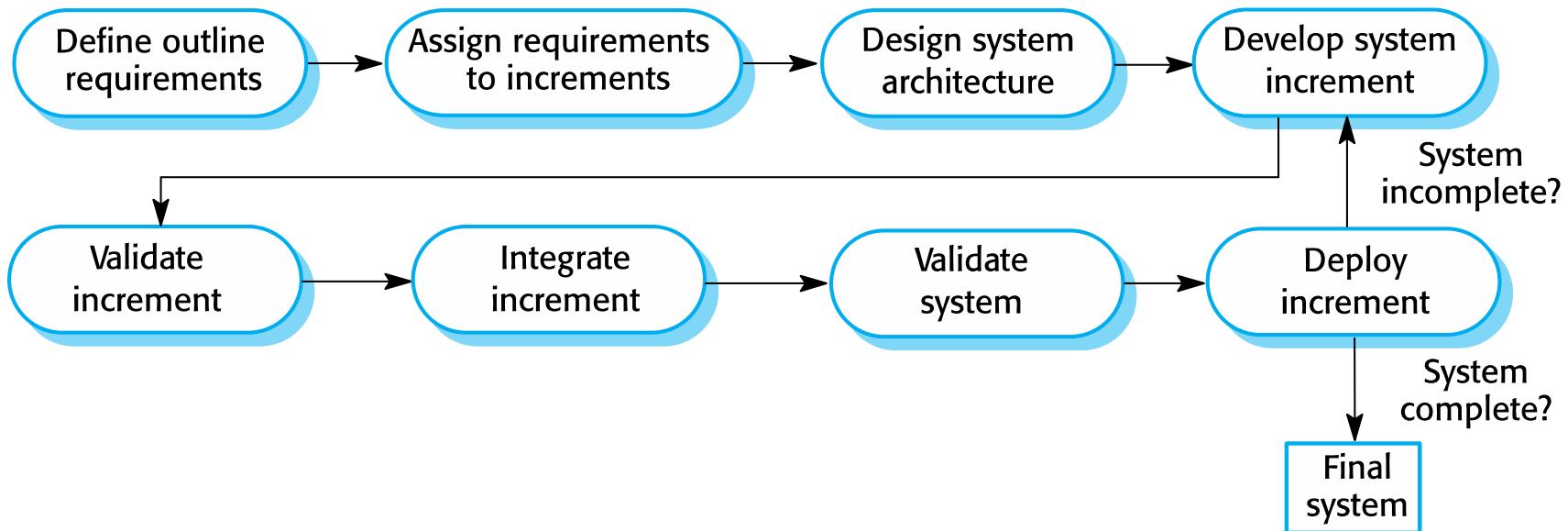
✧ Incremental development

- Develop the system in increments and evaluate each increment before proceeding to the development of the next increment;
- Normal approach used in agile methods;
- Evaluation done by user/customer proxy.

✧ Incremental delivery

- Deploy an increment for use by end-users;
- More realistic evaluation about practical use of software;
- Difficult to implement for replacement systems as increments have less functionality than the system being replaced.

Incremental delivery





Incremental delivery advantages

- ✧ Customer value can be delivered with each increment so system functionality is available earlier.
- ✧ Early increments act as a prototype to help elicit requirements for later increments.
- ✧ Lower risk of overall project failure.
- ✧ The highest priority system services tend to receive the most testing.



Incremental delivery problems

- ✧ Most systems require a set of basic facilities that are used by different parts of the system.
 - As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
- ✧ The essence of iterative processes is that the specification is developed in conjunction with the software.
 - However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.



Process improvement

Process improvement



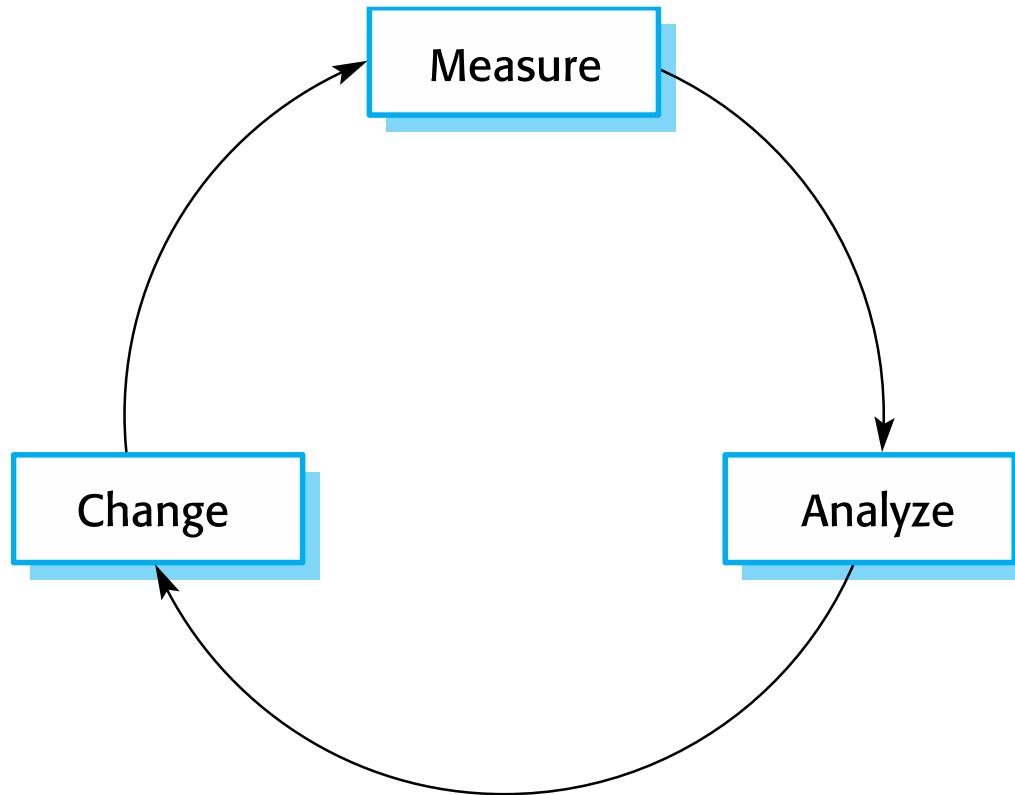
- ✧ Many software companies have turned to software process improvement as a way of enhancing the quality of their software, reducing costs or accelerating their development processes.
- ✧ Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time.



Approaches to improvement

- ✧ The process maturity approach, which focuses on improving process and project management and introducing good software engineering practice.
 - The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes.
- ✧ The agile approach, which focuses on iterative development and the reduction of overheads in the software process.
 - The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements.

The process improvement cycle





Process improvement activities

✧ *Process measurement*

- You 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 analysis*

- The current process is assessed, and process weaknesses and bottlenecks are identified. Process models (sometimes called 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

- ✧ Wherever possible, quantitative process data should be collected
 - However, where organisations do not have clearly defined process standards this is very difficult as you don't know what to measure. A process may have to be defined before any measurement is possible.
- ✧ Process measurements should be used to assess process improvements
 - But this does not mean that measurements should drive the improvements. The improvement driver should be the organizational objectives.

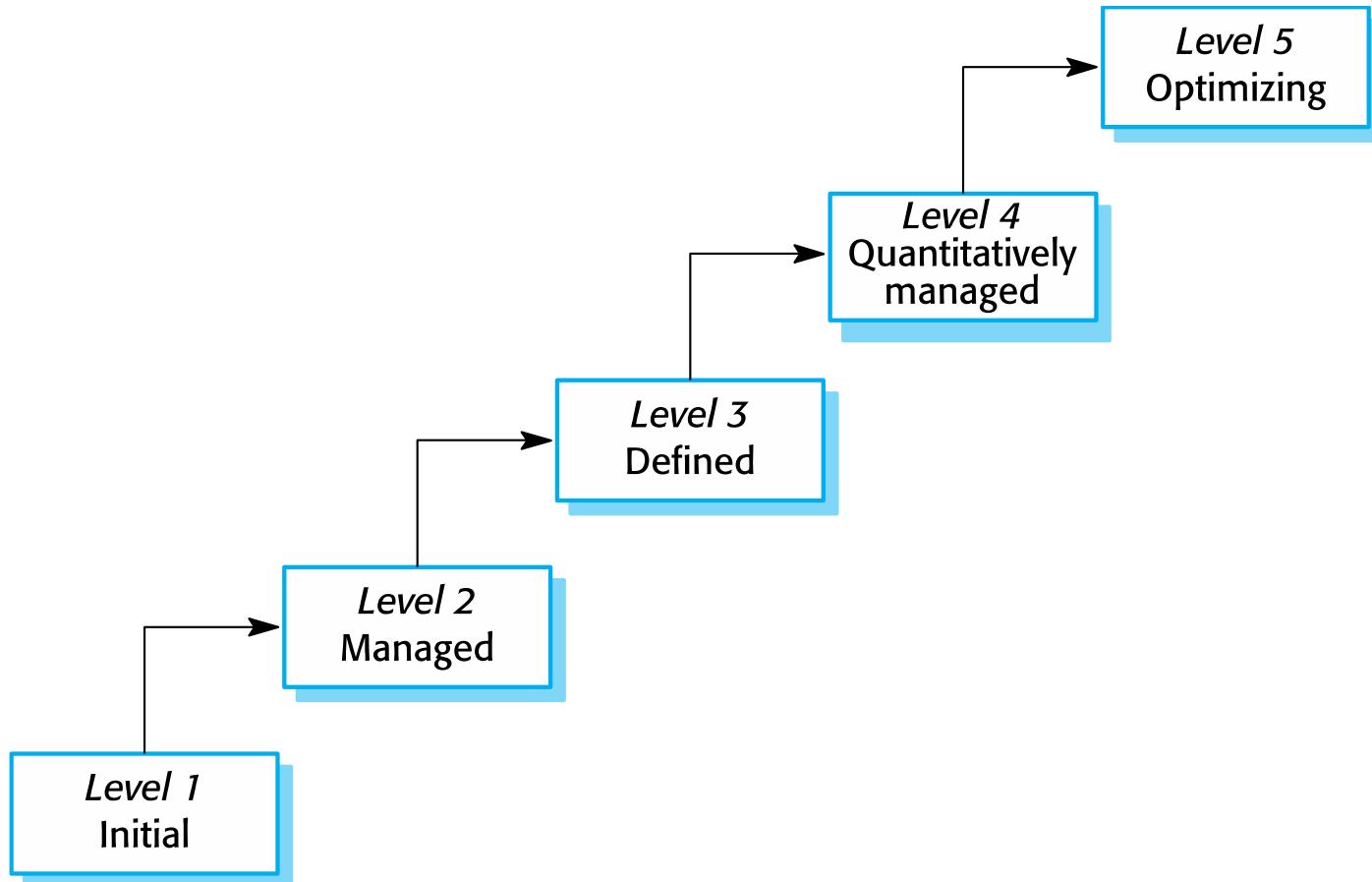


Process metrics

- ✧ Time taken for process activities to be completed
 - E.g. Calendar time or effort to complete an activity or process.
- ✧ Resources required for processes or activities
 - E.g. Total effort in person-days.
- ✧ Number of occurrences of a particular event
 - E.g. Number of defects discovered.



Capability maturity levels





The SEI capability maturity model

✧ Initial

- Essentially uncontrolled

✧ Repeatable

- Product management procedures defined and used

✧ Defined

- Process management procedures and strategies defined and used

✧ Managed

- Quality management strategies defined and used

✧ Optimising

- Process improvement strategies defined and used



Key points

- ✧ Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- ✧ General process models describe the organization of software processes.
 - Examples of these general models include the ‘waterfall’ model, incremental development, and reuse-oriented development.
- ✧ Requirements engineering is the process of developing a software specification.



Key points

- ✧ Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- ✧ Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- ✧ Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.
- ✧ Processes should include activities such as prototyping and incremental delivery to cope with change.



Key points

- ✧ Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- ✧ The principal approaches to process improvement are agile approaches, geared to reducing process overheads, and maturity-based approaches based on better process management and the use of good software engineering practice.
- ✧ The SEI process maturity framework identifies maturity levels that essentially correspond to the use of good software engineering practice.



Chapter 3 – Agile Software Development

Topics covered



- ✧ Agile methods
- ✧ Agile development techniques
- ✧ Agile project management
- ✧ Scaling agile methods

Rapid software development



- ✧ Rapid development and delivery is now often the most important requirement for software systems
 - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
 - Software has to evolve quickly to reflect changing business needs.
- ✧ Plan-driven development is essential for some types of system but does not meet these business needs.
- ✧ Agile development methods emerged in the late 1990s whose aim was to radically reduce the delivery time for working software systems

Agile development

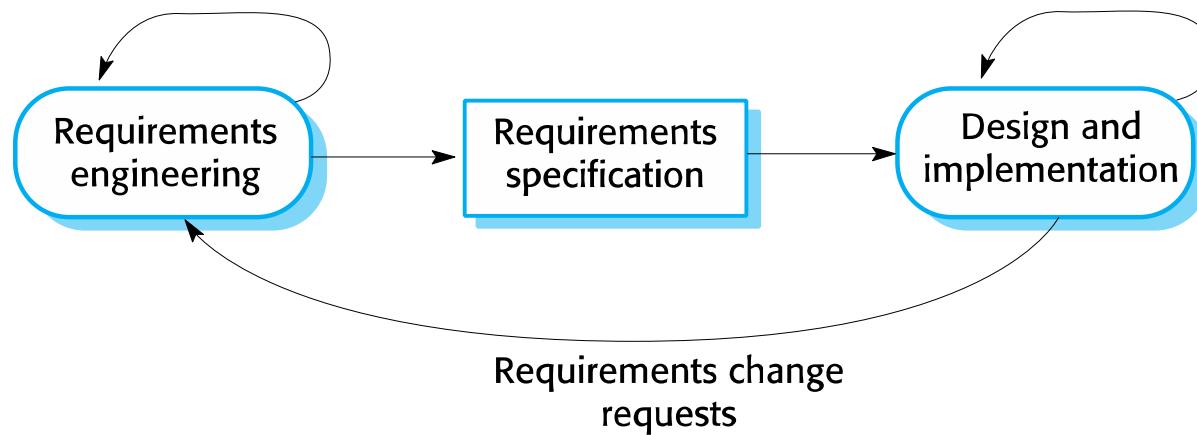


- ✧ Program specification, design and implementation are inter-leaved
- ✧ The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation
- ✧ Frequent delivery of new versions for evaluation
- ✧ Extensive tool support (e.g. automated testing tools) used to support development.
- ✧ Minimal documentation – focus on working code

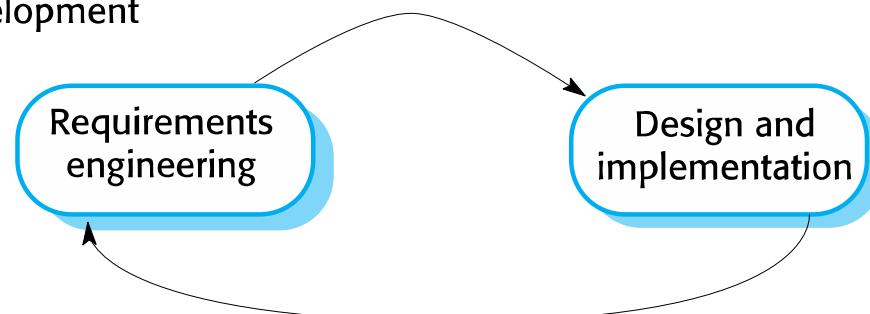
Plan-driven and agile development



Plan-based development



Agile development



Plan-driven and agile development

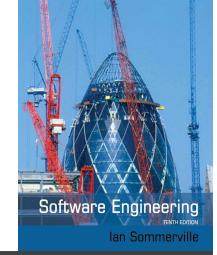


✧ Plan-driven development

- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible
- Iteration occurs within activities.

✧ Agile development

- Specification, design, implementation and testing are interleaved and the outputs from the development process are decided through a process of negotiation during the software development process.



Agile methods

Agile methods



- ✧ Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
 - Focus on the code rather than the design
 - Are based on an iterative approach to software development
 - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- ✧ The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

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:*
 - *Individuals and interactions over processes and tools*
 - Working software over comprehensive documentation*
 - Customer collaboration over contract negotiation*
 - Responding to change over following a plan*
- ✧ *That is, while there is value in the items on the right, we value the items on the left more.*

The principles of agile methods

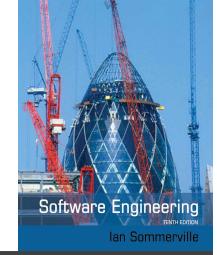


Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Agile method applicability



- ✧ Product development where a software company is developing a small or medium-sized product for sale.
 - Virtually all software products and apps are now developed using an agile approach
- ✧ Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external rules and regulations that affect the software.



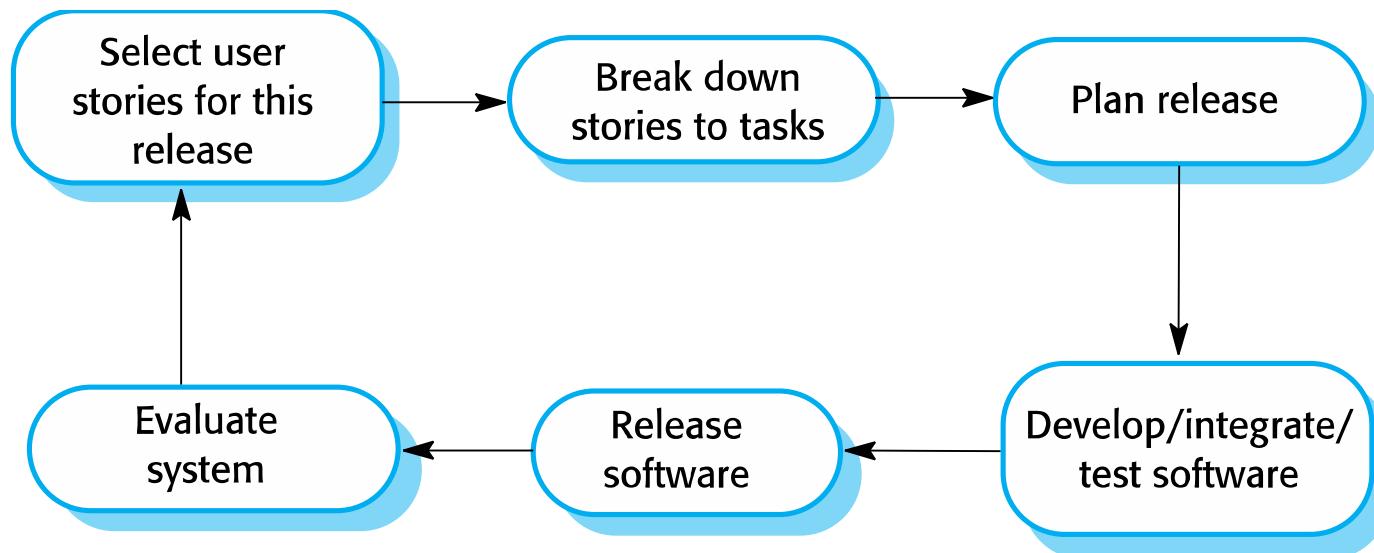
Agile development techniques

Extreme programming



- ✧ A very influential agile method, developed in the late 1990s, that introduced a range of agile development techniques.
- ✧ 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.

The extreme programming release cycle



Extreme programming practices (a)



Principle or practice	Description
Incremental planning	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'. See Figures 3.5 and 3.6.
Small releases	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.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
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>

Extreme programming practices (b)



Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	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.
Continuous integration	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.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	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 and agile principles



- ✧ Incremental development is supported through small, frequent system releases.
- ✧ Customer involvement means full-time customer engagement with the team.
- ✧ People not process through pair programming, collective ownership and a process that avoids long working hours.
- ✧ Change supported through regular system releases.
- ✧ Maintaining simplicity through constant refactoring of code.

Influential XP practices



- ✧ Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.
- ✧ Consequently, while agile development uses practices from XP, the method as originally defined is not widely used.
- ✧ Key practices
 - User stories for specification
 - Refactoring
 - Test-first development
 - Pair programming

User stories for requirements



- ✧ In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- ✧ User requirements are expressed as user stories or scenarios.
- ✧ These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- ✧ The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

A ‘prescribing medication’ story



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.

Examples of task cards for prescribing medication



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.

Refactoring



- ✧ 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, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- ✧ Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.

Refactoring



- ✧ Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- ✧ This improves the understandability of the software and so reduces the need for documentation.
- ✧ Changes are easier to make because the code is well-structured and clear.
- ✧ However, some changes require architecture refactoring and this is much more expensive.

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.

Test-first development



- ✧ Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
- ✧ XP testing features:
 - 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.

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.
 - Usually relies on a testing framework such as Junit.
- ✧ 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 role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- ✧ The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- ✧ However, people adopting the customer role 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 means that tests are written as executable components before the task is implemented
 - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.
- ❖ As testing is automated, there is always a set of tests that can be quickly and easily executed
 - 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.

Problems with test-first development



- ✧ Programmers prefer programming to testing and sometimes they take short cuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- ✧ Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
- ✧ It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.

Pair programming



- ✧ Pair programming involves programmers working in pairs, developing code together.
- ✧ This helps develop common ownership of code and spreads knowledge across the team.
- ✧ It serves as an informal review process as each line of code is looked at by more than 1 person.
- ✧ It encourages refactoring as the whole team can benefit from improving the system code.

Pair programming



- ✧ In pair programming, 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.
- ✧ Pair programming is not necessarily inefficient and there is some evidence that 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 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, which is adapted to incremental development and the practices used in agile methods.

Scrum



- ✧ Scrum is an agile method that focuses on managing iterative development rather than specific agile practices.
- ✧ There are three phases in Scrum.
 - The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
 - This is followed by a series of sprint cycles, where each cycle develops an increment of the system.
 - The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.





Scrum terminology (a)

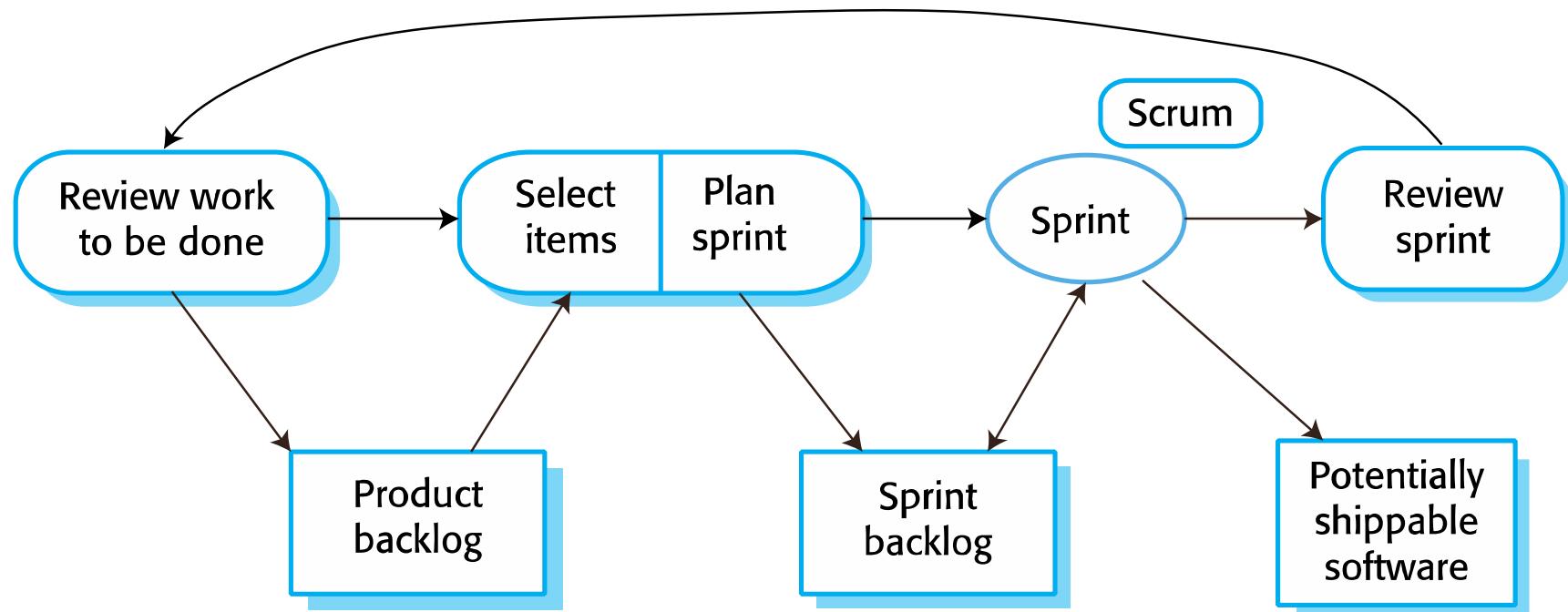
Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a 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 a list of 'to do' items 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 terminology (b)

Scrum term	Definition
Scrum	A daily meeting 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.
ScrumMaster	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	An estimate of how much product backlog effort 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.

Scrum sprint cycle



The Scrum sprint cycle



- ✧ 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 of the project team who work with the customer to select the features and functionality from the product backlog to be developed during the sprint.

The Sprint cycle



- ✧ 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.

Teamwork in Scrum



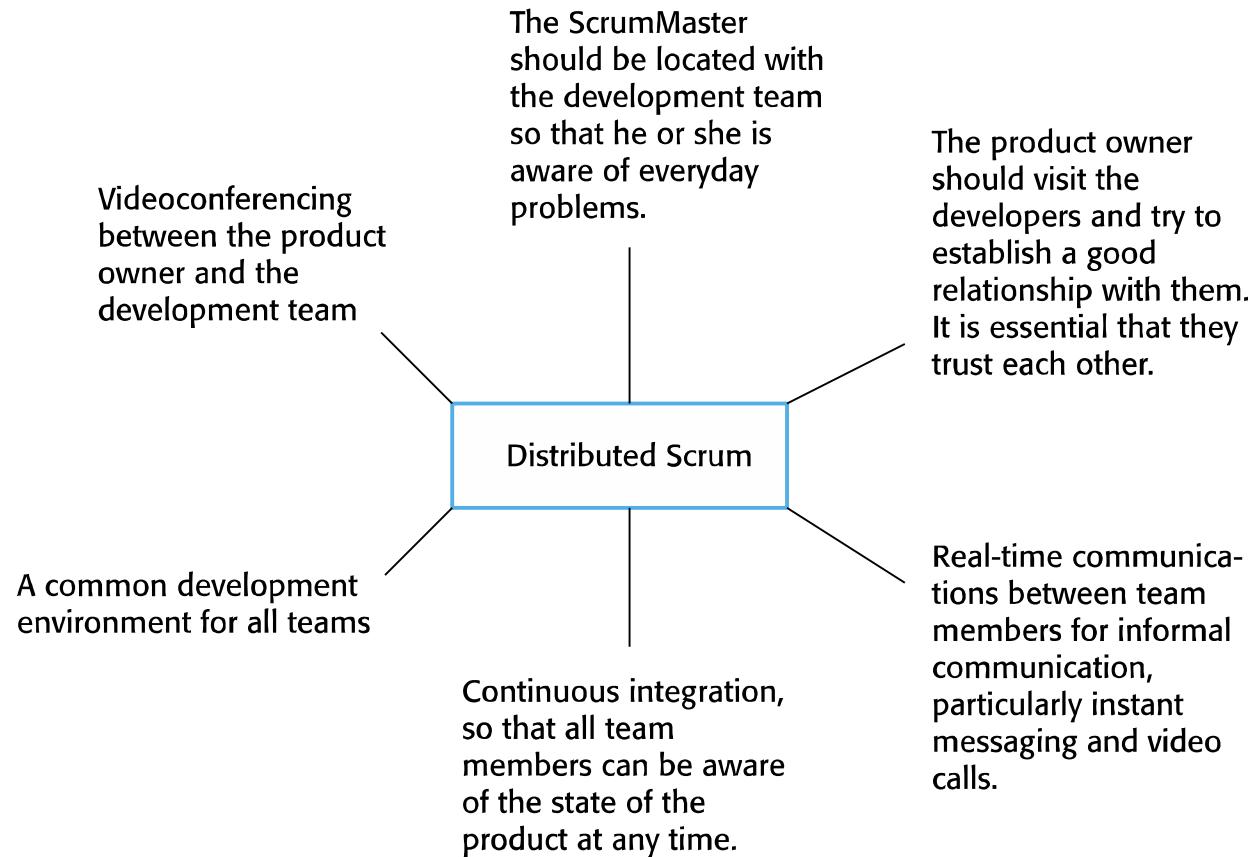
- ✧ The ‘Scrum master’ is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- ✧ The whole team attends short daily meetings (Scrums) where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
 - This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

Scrum benefits



- ✧ The product is broken down into a set of manageable and understandable chunks.
- ✧ Unstable requirements do not hold up progress.
- ✧ The whole team have visibility of everything and consequently team communication is improved.
- ✧ Customers see on-time delivery of increments and gain feedback on how the product works.
- ✧ Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Distributed Scrum





Scaling agile methods

Scaling agile methods



- ✧ Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.
- ✧ It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.
- ✧ Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

Scaling out and scaling up



- ✧ ‘Scaling up’ is concerned with using agile methods for developing large software systems that cannot be developed by a small team.
- ✧ ‘Scaling out’ is concerned with how agile methods can be introduced across a large organization with many years of software development experience.
- ✧ When scaling agile methods it is important to maintain agile fundamentals:
 - Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.

Practical problems with agile methods



- ✧ The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
- ✧ Agile methods are most appropriate for new software development rather than software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
- ✧ Agile methods are designed for small co-located teams yet much software development now involves worldwide distributed teams.

Contractual issues



- ✧ Most software contracts for custom systems are based around a specification, which sets out what has to be implemented by the system developer for the system customer.
- ✧ However, this precludes interleaving specification and development as is the norm in agile development.
- ✧ A contract that pays for developer time rather than functionality is required.
 - However, this is seen as a high risk my many legal departments because what has to be delivered cannot be guaranteed.

Agile methods and software maintenance



- ✧ Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- ✧ Two key issues:
 - Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
 - Can agile methods be used effectively for evolving a system in response to customer change requests?
- ✧ Problems may arise if original development team cannot be maintained.

Agile maintenance



- ✧ Key problems are:
 - Lack of product documentation
 - Keeping customers involved in the development process
 - Maintaining the continuity of the development team
- ✧ Agile development relies on the development team knowing and understanding what has to be done.
- ✧ For long-lifetime systems, this is a real problem as the original developers will not always work on the system.

Agile and plan-driven methods



- ❖ Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:
 - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
 - Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
 - How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

Agile principles and organizational practice



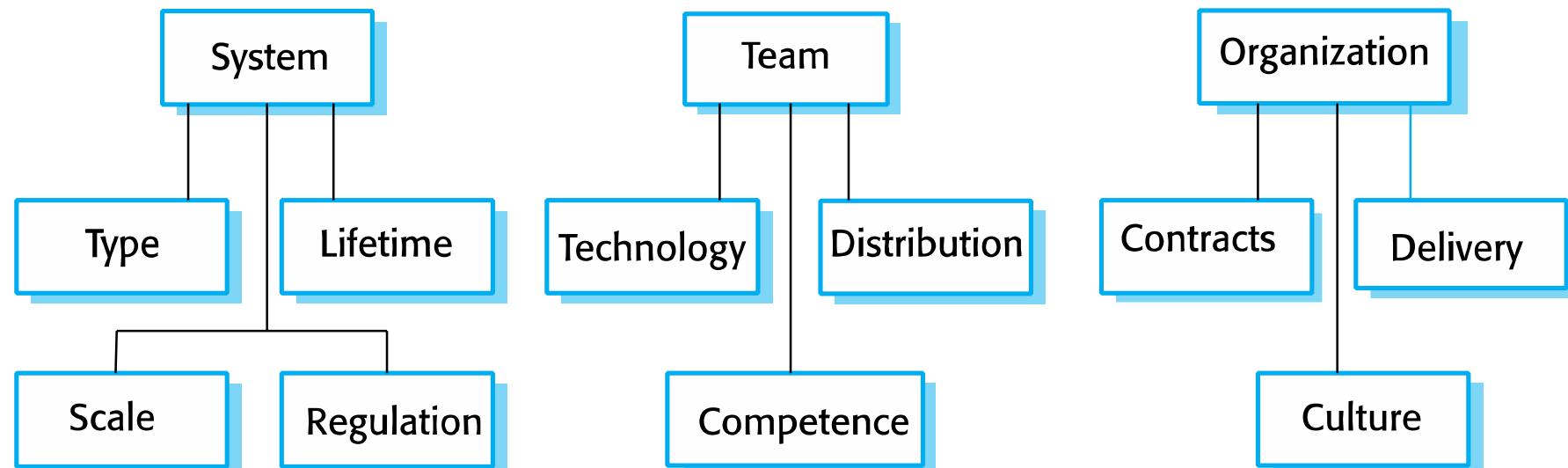
Principle	Practice
Customer involvement	<p>This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development.</p> <p>Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.</p>
Embrace change	Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
Incremental delivery	Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know what product features several months in advance to prepare an effective marketing campaign.

Agile principles and organizational practice



Principle	Practice
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore may not interact well with other team members.

Agile and plan-based factors



System issues



- ✧ How large is the system being developed?
 - Agile methods are most effective a relatively small co-located team who can communicate informally.
- ✧ What type of system is being developed?
 - Systems that require a lot of analysis before implementation need a fairly detailed design to carry out this analysis.
- ✧ What is the expected system lifetime?
 - Long-lifetime systems require documentation to communicate the intentions of the system developers to the support team.
- ✧ Is the system subject to external regulation?
 - If a system is regulated you will probably be required to produce detailed documentation as part of the system safety case.

People and teams



- ✧ How good are the designers and programmers in the development team?
 - It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code.
- ✧ How is the development team organized?
 - Design documents may be required if the team is distributed.
- ✧ What support technologies are available?
 - IDE support for visualisation and program analysis is essential if design documentation is not available.

Organizational issues



- ✧ Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- ✧ Is it standard organizational practice to develop a detailed system specification?
- ✧ Will customer representatives be available to provide feedback of system increments?
- ✧ Can informal agile development fit into the organizational culture of detailed documentation?

Agile methods for large systems



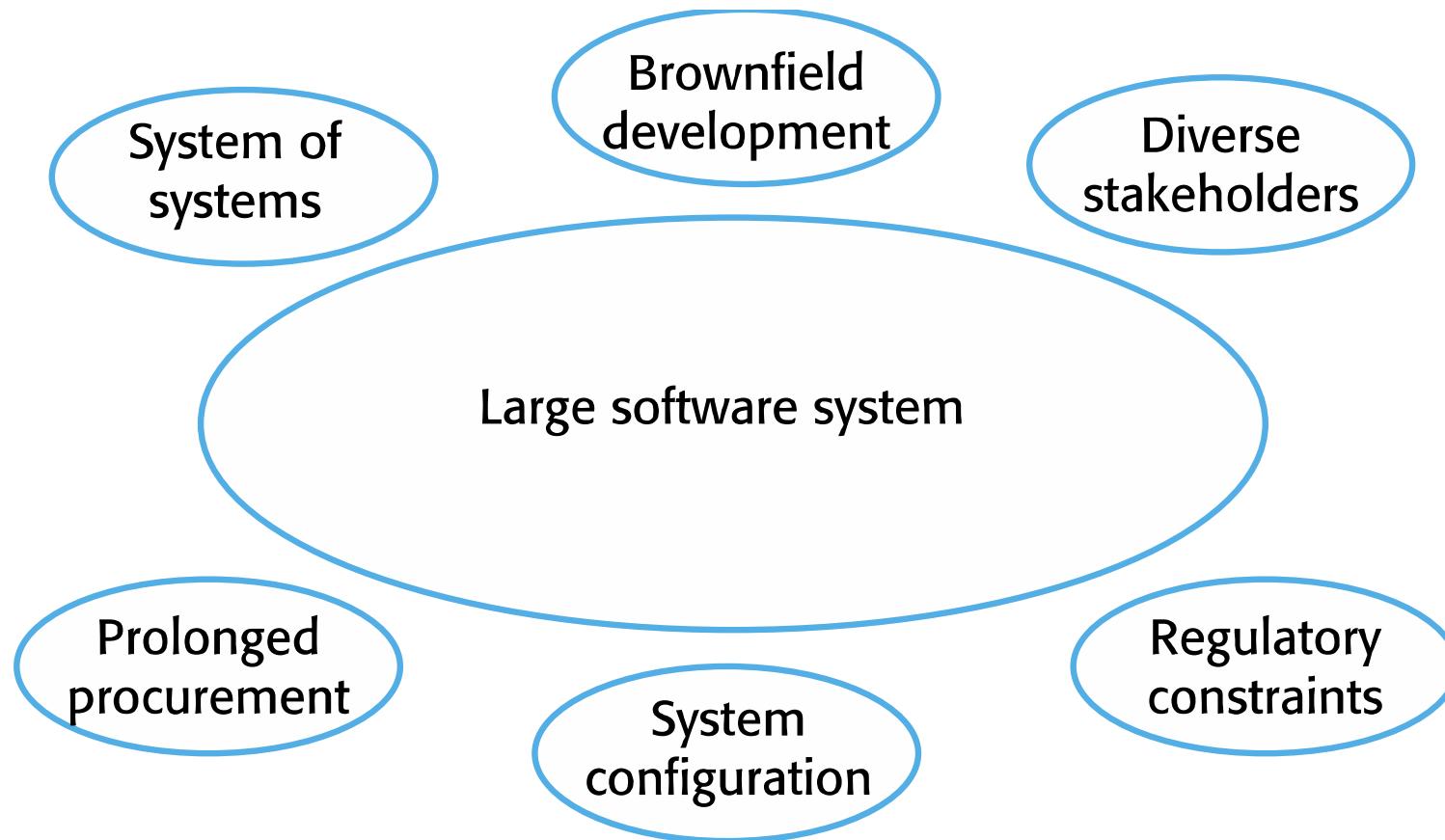
- ✧ Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- ✧ Large systems are ‘brownfield systems’, that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don’t really lend themselves to flexibility and incremental development.
- ✧ Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.

Large system development

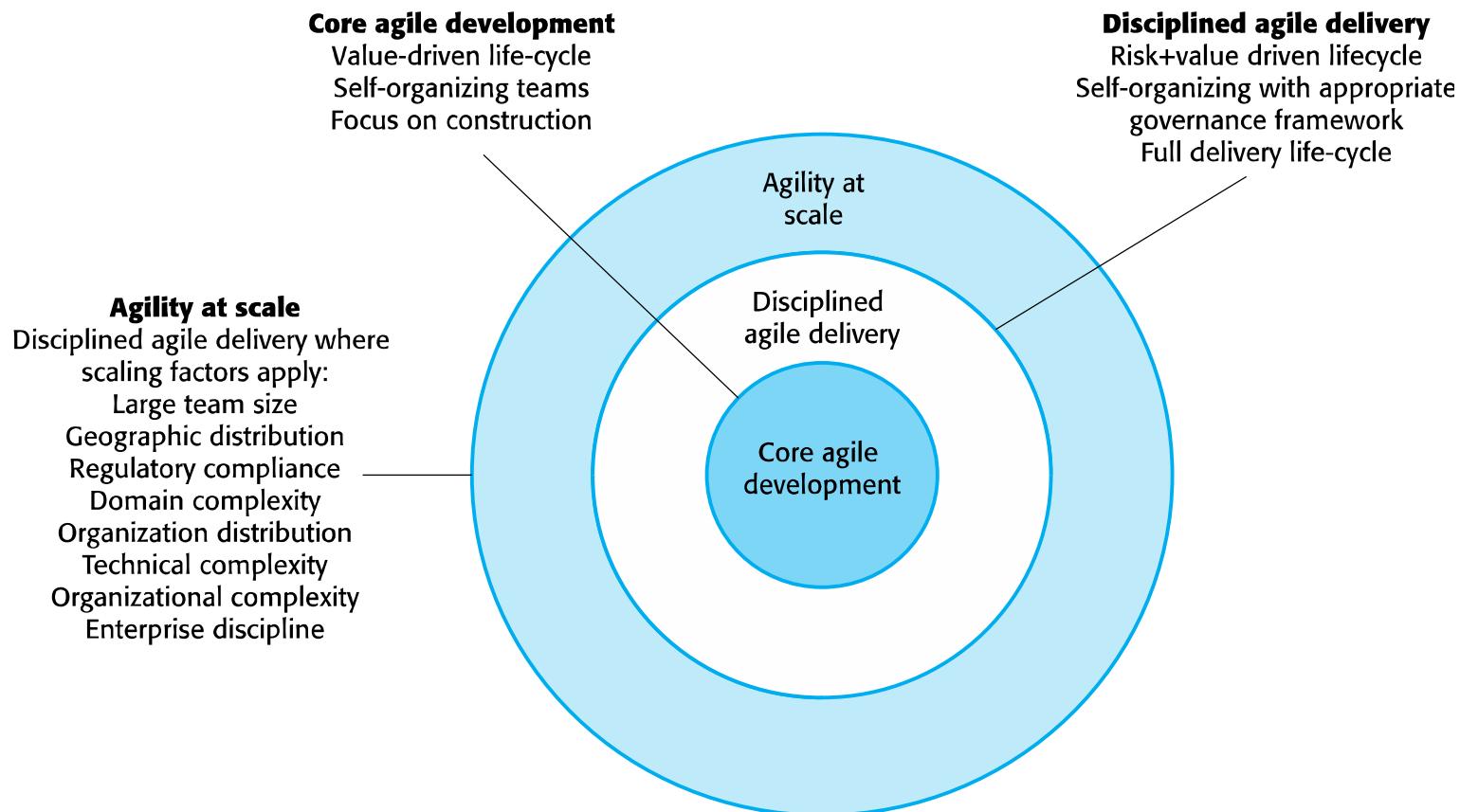


- ✧ Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- ✧ Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
- ✧ Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of these different stakeholders in the development process.

Factors in large systems



IBM's agility at scale model



Scaling up to large systems



- ✧ A completely incremental approach to requirements engineering is impossible.
- ✧ There cannot be a single product owner or customer representative.
- ✧ For large systems development, it is not possible to focus only on the code of the system.
- ✧ Cross-team communication mechanisms have to be designed and used.
- ✧ Continuous integration is practically impossible. However, it is essential to maintain frequent system builds and regular releases of the system.

Multi-team Scrum



✧ *Role replication*

- Each team has a Product Owner for their work component and ScrumMaster.

✧ *Product architects*

- Each team chooses a product architect and these architects collaborate to design and evolve the overall system architecture.

✧ *Release alignment*

- The dates of product releases from each team are aligned so that a demonstrable and complete system is produced.

✧ *Scrum of Scrums*

- There is a daily Scrum of Scrums where representatives from each team meet to discuss progress and plan work to be done.

Agile methods across organizations



- ✧ Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
- ✧ Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.
- ✧ Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities.
- ✧ There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Key points



- ✧ Agile methods are incremental development methods that focus on rapid software development, frequent releases of the software, reducing process overheads by minimizing documentation and producing high-quality code.
- ✧ Agile development practices include
 - User stories for system specification
 - Frequent releases of the software,
 - Continuous software improvement
 - Test-first development
 - Customer participation in the development team.

Key points



- ✧ Scrum is an agile method that provides a project management framework.
 - It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- ✧ Many practical development methods are a mixture of plan-based and agile development.
- ✧ Scaling agile methods for large systems is difficult.
 - Large systems need up-front design and some documentation and organizational practice may conflict with the informality of agile approaches.



Chapter 4 – Requirements Engineering



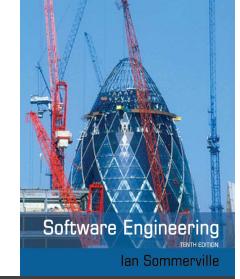
Topics covered

- ✧ Functional and non-functional requirements
- ✧ Requirements engineering processes
- ✧ Requirements elicitation
- ✧ Requirements specification
- ✧ Requirements validation
- ✧ Requirements change



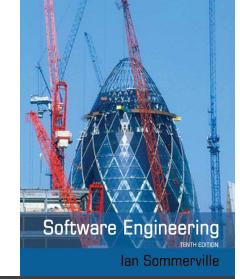
Requirements engineering

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



What is a requirement?

- ✧ It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- ✧ This is inevitable as requirements may serve a dual function
 - May be the basis for a bid for a contract - therefore must be open to interpretation;
 - May be the basis for the contract itself - therefore must be defined in detail;
 - Both these statements may be called requirements.



Requirements abstraction (Davis)

“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.”



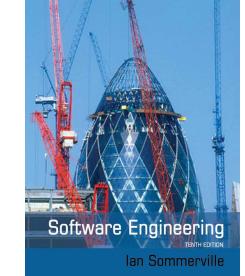
Types of requirement

✧ User requirements

- Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written 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 so may be part of a contract between client and contractor.**



User and system requirements (ment care example)

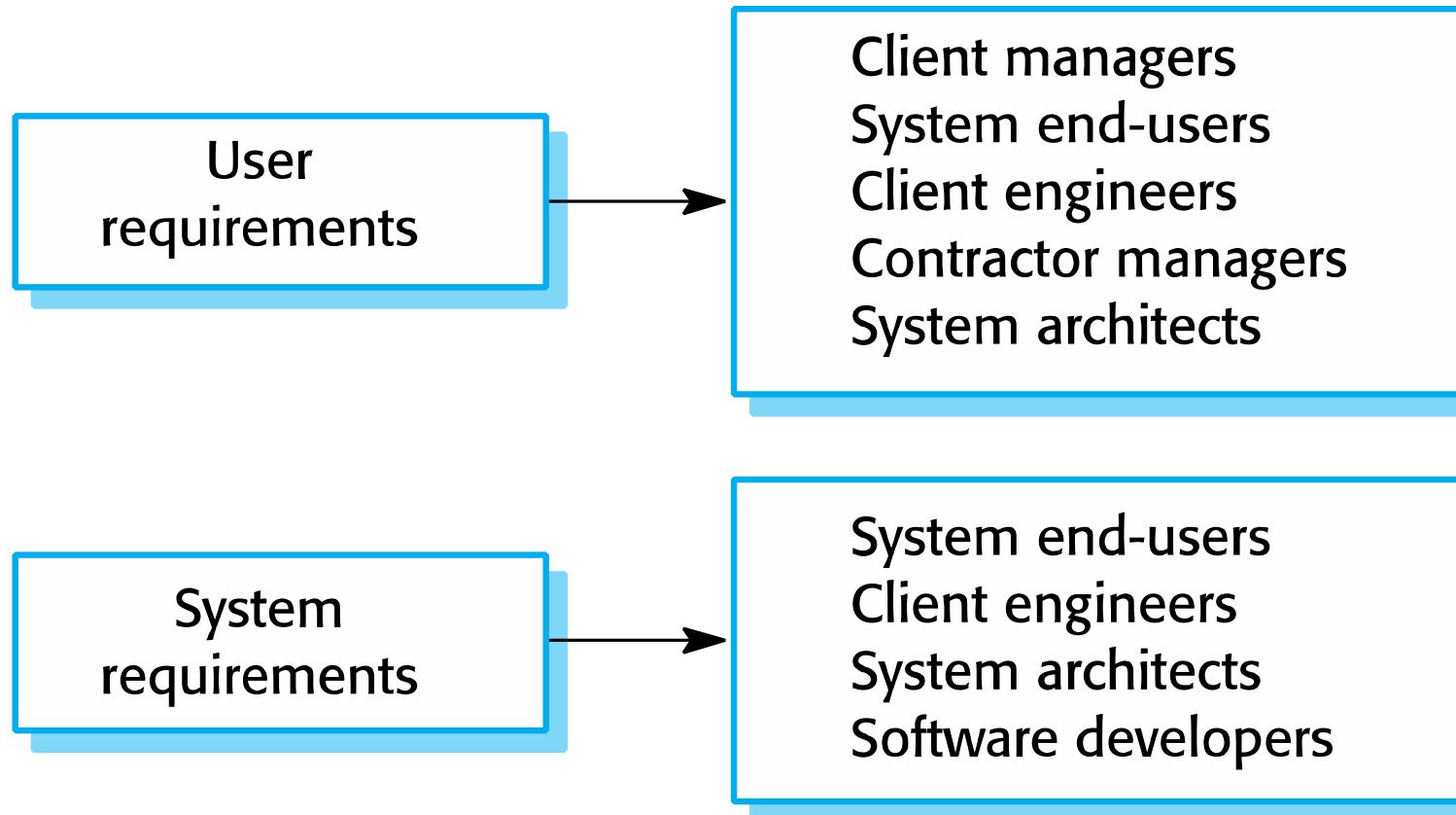
User requirements definition

- 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.
Dikumpulkan tgl. 17 maret 2020
Dokumentasi harus dapat terbaca (anda boleh mencetak di kertas A3)
- 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.

Readers of different types of requirements specification





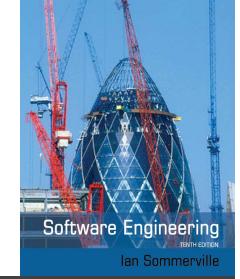
System stakeholders

- ✧ Any person or organization who is affected by the system in some way and so who has a legitimate interest
- ✧ Stakeholder types
 - End users
 - System managers
 - System owners
 - External stakeholders



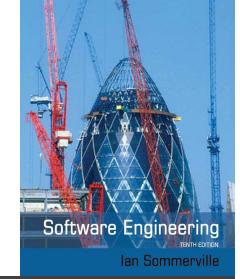
Example : Stakeholders in the Mentcare system

- ✧ Patients whose information is recorded in the system.
- ✧ Doctors who are responsible for assessing and treating patients.
- ✧ Nurses who coordinate the consultations with doctors and administer some treatments.
- ✧ Medical receptionists who manage patients' appointments.
- ✧ IT staff who are responsible for installing and maintaining the system.



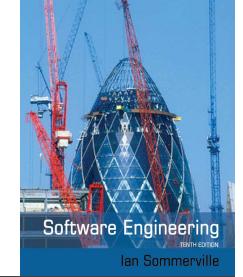
Stakeholders in the Mentcare system

- ✧ A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- ✧ Health care managers who obtain management information from the system.
- ✧ Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

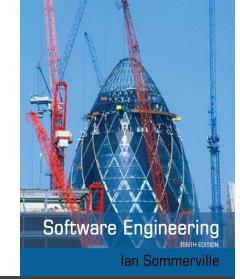


Agile methods and requirements

- ✧ 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’** (discussed in Chapter 3).
- ✧ This is practical for business systems but 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 and 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.

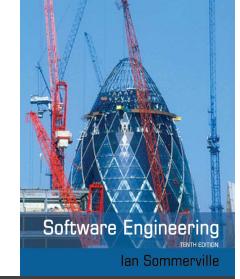
❖ Domain requirements

- Constraints on the system from the domain of operation



Functional requirements

- ✧ Describe functionality or system services.
- ✧ Depend 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 in detail.



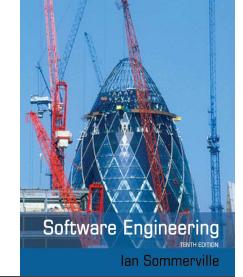
Example : Mentcare system: functional requirements

- ✧ A user shall be able to search the appointments lists for all clinics.
- ✧ The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- ✧ Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.



Requirements imprecision

- ✧ Problems arise when functional requirements are not precisely stated.
- ✧ Ambiguous requirements may be interpreted in different ways by developers and users.
- ✧ Consider the term 'search' in requirement 1
 - 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 clinic then search.



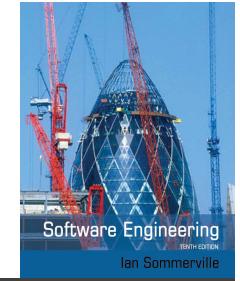
Requirements completeness and consistency

- ✧ In principle, requirements should be both complete and consistent.
- ✧ 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, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.



Non-functional requirements

- ✧ These define system properties and constraints e.g. **reliability, response time and storage requirements.** **Constraints are I/O device capability, system representations, etc.**
- ✧ Process requirements may also be specified mandating a particular IDE, programming language or development method.
- ✧ Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

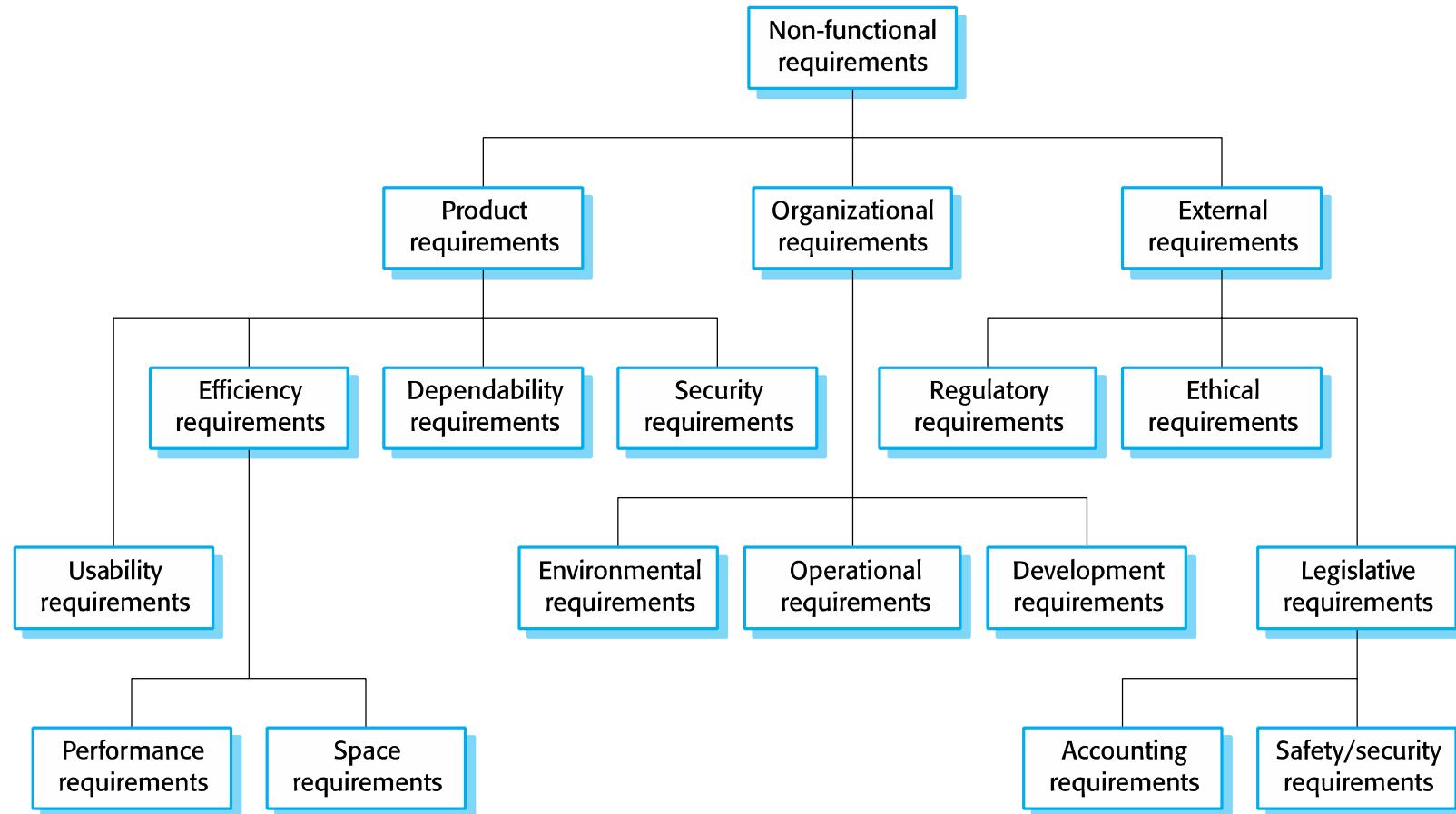


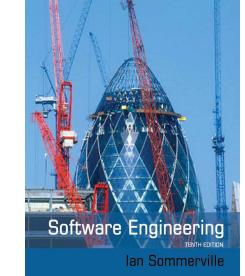
Software Engineering

9TH EDITION

Ian Sommerville

Types of nonfunctional requirement





Non-functional requirements implementation

- ✧ Non-functional requirements may affect the overall architecture of a system rather than the individual components.
 - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- ✧ A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.
 - It may also generate requirements that restrict existing 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.

✧ Organisational requirements

- **Requirements which are a consequence of organisational policies and procedures** e.g. process standards used, implementation requirements, etc.

✧ External requirements

- **Requirements which arise from factors which are external to the system and its development process** e.g. interoperability requirements, legislative requirements, etc.



Examples of nonfunctional requirements in the Mentcare system

Product requirement

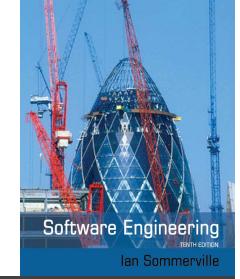
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement

Users of the Mentcare system shall authenticate themselves using their health authority identity card.

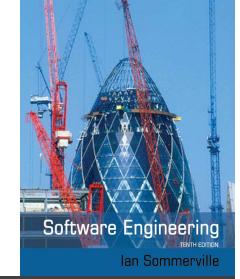
External requirement

The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.



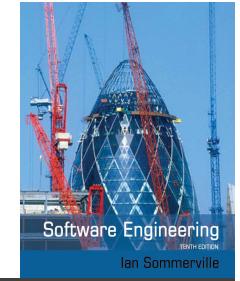
Goals and requirements

- ✧ Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- ✧ Goal
 - A general intention of the user such as ease of use.
- ✧ Verifiable non-functional requirement
 - A statement using some measure that can be objectively tested.
- ✧ Goals are helpful to developers as they convey the intentions of the system users.



Usability requirements

- ✧ The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal)
- ✧ 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. (Testable non-functional requirement)

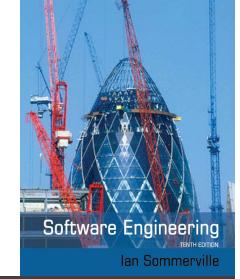


Metrics for specifying nonfunctional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems



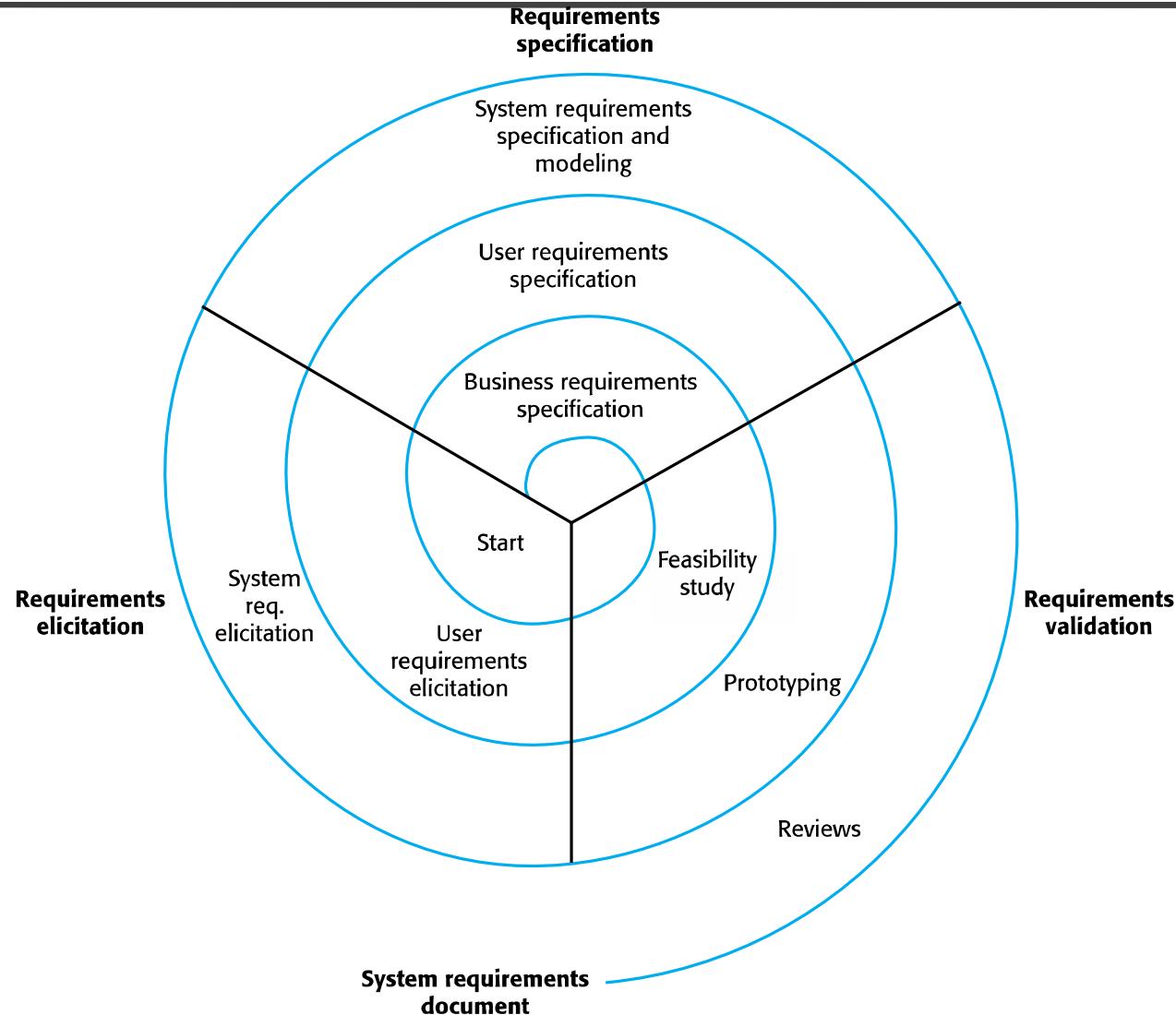
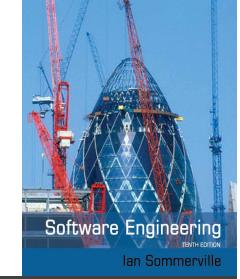
Requirements engineering processes

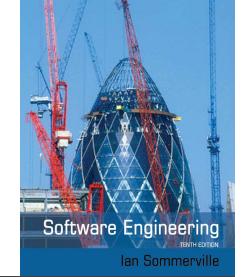


Requirements engineering processes

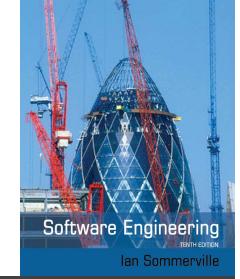
- ✧ The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- ✧ However, there are a number of generic activities common to all processes
 - Requirements elicitation;
 - Requirements analysis;
 - Requirements validation;
 - Requirements management.
- ✧ In practice, RE is an iterative activity in which these processes are interleaved.

A spiral view of the requirements engineering process





Requirements elicitation



Requirements elicitation and analysis

- ✧ Sometimes called requirements elicitation or requirements discovery.
- ✧ Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- ✧ May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.



Requirements elicitation



Requirements elicitation

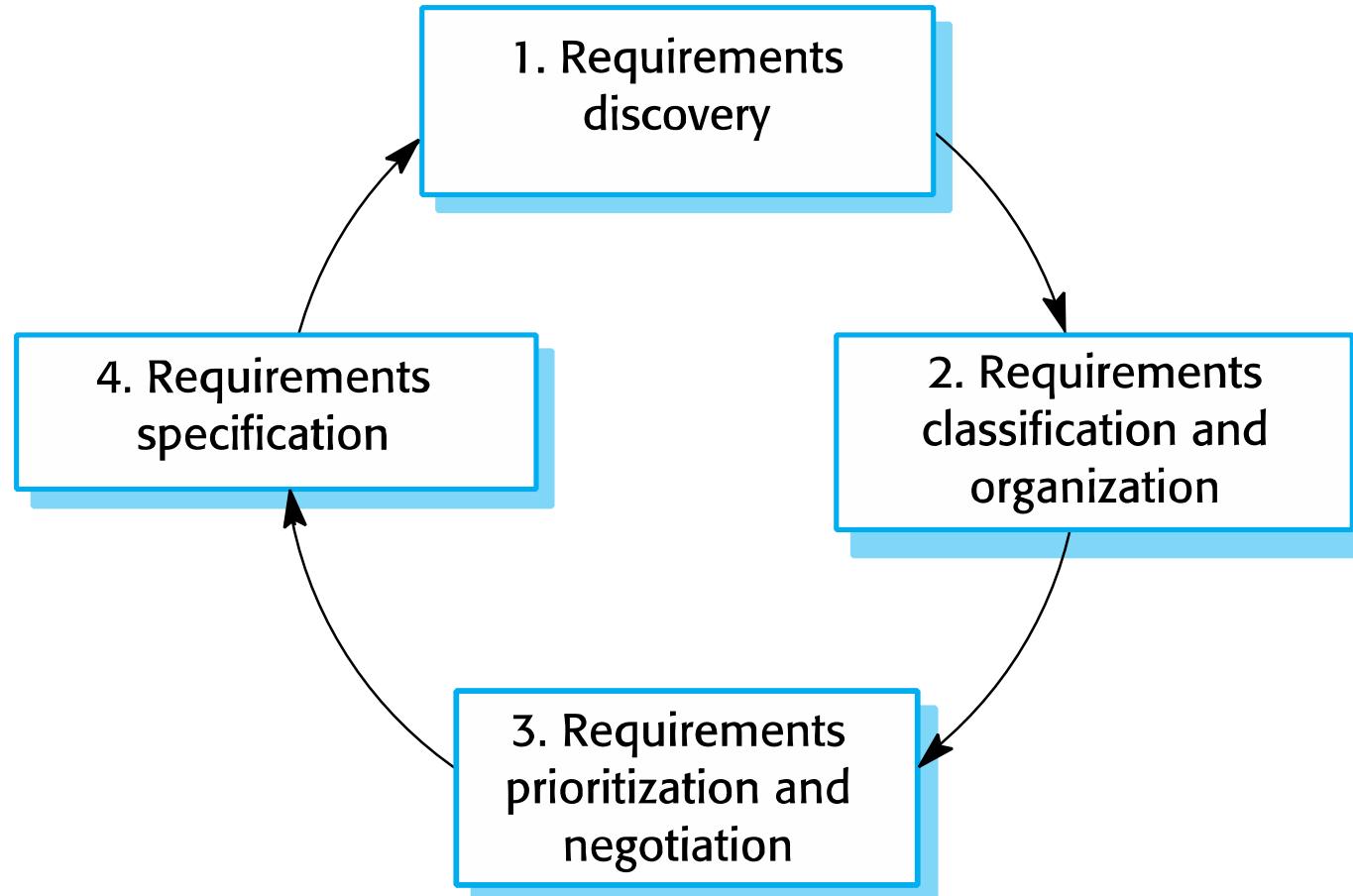
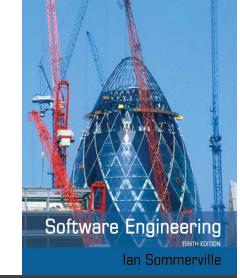
- ✧ Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.
- ✧ Stages include:
 - Requirements discovery,
 - Requirements classification and organization,
 - Requirements prioritization and negotiation,
 - Requirements specification.

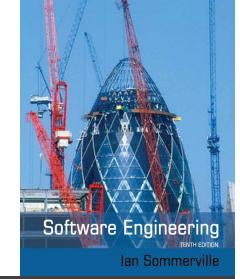
Problems of requirements elicitation



- ✧ Stakeholders don't know what they really want.
- ✧ Stakeholders express requirements in their own terms.
- ✧ Different stakeholders may have conflicting requirements.
- ✧ Organisational 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.

The requirements elicitation and analysis process





Process activities

✧ Requirements discovery

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

✧ Requirements classification and organisation

- Groups related requirements and organises them into coherent clusters.

✧ Prioritisation and negotiation

- Prioritising requirements and resolving requirements conflicts.

✧ Requirements specification

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



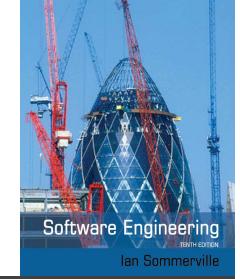
Requirements discovery

- ✧ The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.
- ✧ Interaction is with system stakeholders from managers to external regulators.
- ✧ Systems normally have a range of stakeholders.



Interviewing

- ✧ Formal or informal interviews with stakeholders are part of most RE processes.
- ✧ Types of interview
 - Closed interviews based on pre-determined list of questions
 - Open interviews where various issues are explored with stakeholders.
- ✧ Effective interviewing
 - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
 - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.



Interviews in practice

- ✧ Normally a mix of closed and open-ended interviewing.
- ✧ Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- ✧ Interviewers need to be open-minded without pre-conceived ideas of what the system should do
- ✧ You need to prompt the user to talk about the system by suggesting requirements rather than simply asking them what they want.



Problems with interviews

- ✧ Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
- ✧ Interviews are not good for understanding domain requirements
 - Requirements engineers cannot understand specific domain terminology;
 - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.



Ethnography

- ✧ A social scientist spends a considerable time observing and analysing how people actually work.
- ✧ People do not have to explain or articulate their work.
- ✧ Social and organisational factors of importance may be observed.
- ✧ Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.



Scope of ethnography

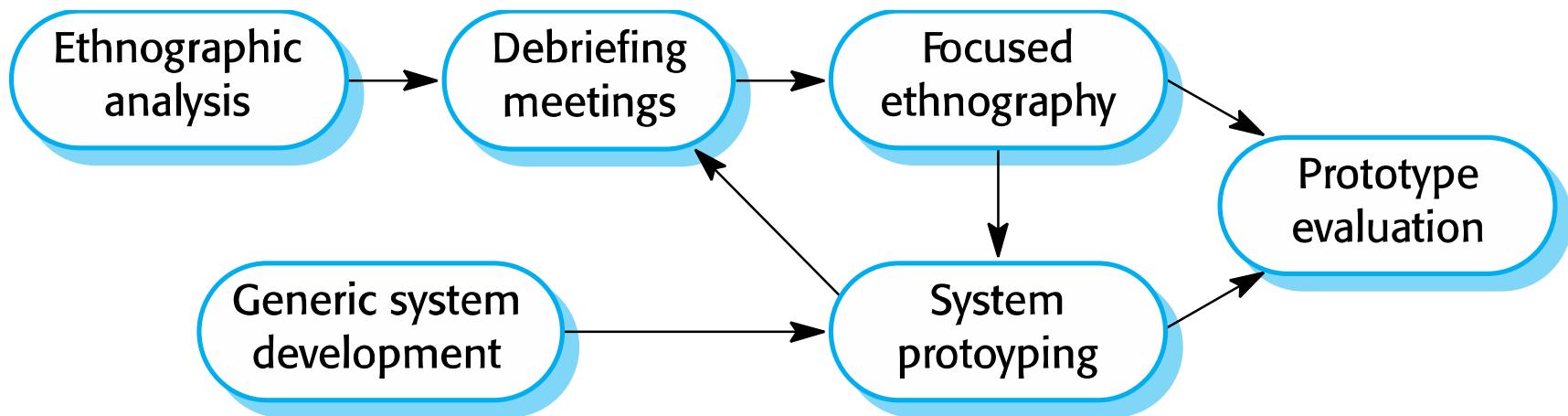
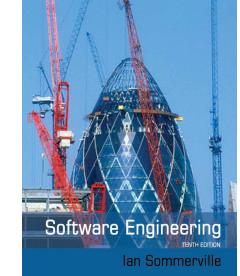
- ✧ Requirements that are derived from the way that people actually work rather than the way in which process definitions suggest that they ought to work.
- ✧ Requirements that are derived from cooperation and awareness of other people's activities.
 - Awareness of what other people are doing leads to changes in the ways in which we do things.
- ✧ Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system.



Focused ethnography

- ✧ Developed in a project studying the air traffic control process
- ✧ Combines ethnography with prototyping
- ✧ Prototype development results in unanswered questions which focus the ethnographic analysis.
- ✧ The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.

Ethnography and prototyping for requirements analysis





Stories and scenarios

- ✧ Scenarios and user stories are real-life examples of how a system can be used.
- ✧ Stories and scenarios are a description of how a system may be used for a particular task.
- ✧ Because they are based on a practical situation, stakeholders can relate to them and can comment on their situation with respect to the story.

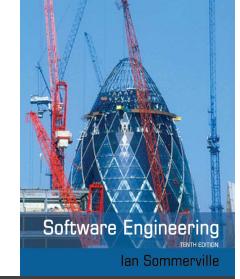
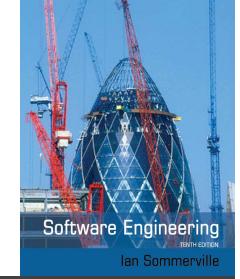


Photo sharing in the classroom (iLearn)

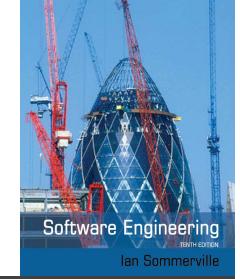
- ✧ Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development and economic impact of fishing. As part of this, pupils are asked to gather and share reminiscences from relatives, use newspaper archives and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo sharing site as he wants pupils to take and comment on each others' photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers group, which he is a member of to see if anyone can recommend an appropriate system. Two teachers reply and both suggest that he uses KidsTakePics, a photo sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.



Scenarios

- ✧ A structured form of user story
- ✧ Scenarios should include
 - A description of the starting situation;
 - A description of the normal flow of events;
 - A description of what can go wrong;
 - Information about other concurrent activities;
 - A description of the state when the scenario finishes.



Uploading photos iLearn)

- ✧ **Initial assumption:** A user or a group of users have one or more digital photographs to be uploaded to the picture sharing site. These are saved on either a tablet or laptop computer. They have successfully logged on to KidsTakePics.
- ✧ **Normal:** The user chooses upload photos and they are prompted to select the photos to be uploaded on their computer and to select the project name under which the photos will be stored. They should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.
- ✧ On completion of the upload, the system automatically sends an email to the project moderator asking them to check new content and generates an on-screen message to the user that this has been done.



Uploading photos

- ✧ **What can go wrong:**
- ✧ No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed that there could be a delay in making their photos visible.
- ✧ Photos with the same name have already been uploaded by the same user. The user should be asked if they wish to re-upload the photos with the same name, rename the photos or cancel the upload. If they chose to re-upload the photos, the originals are overwritten. If they chose to rename the photos, a new name is automatically generated by adding a number to the existing file name.
- ✧ **Other activities:** The moderator may be logged on to the system and may approve photos as they are uploaded.
- ✧ **System state on completion:** User is logged on. The selected photos have been uploaded and assigned a status 'awaiting moderation'. Photos are visible to the moderator and to the user who uploaded them.

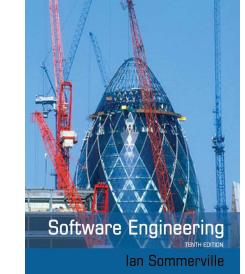


Requirements specification



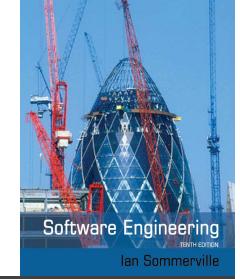
Requirements specification

- ✧ 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
 - It is therefore important that these are as complete as possible.



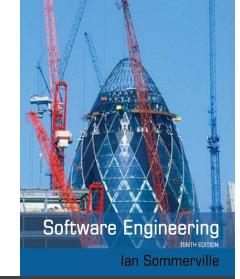
Ways of writing a system requirements specification

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. They cannot check that it represents what they want and are reluctant to accept it as a system contract



Requirements and design

- ✧ In principle, requirements should state what the system should do and the design should describe how it does this.
- ✧ In practice, requirements and design are inseparable
 - A system architecture may be designed to structure the requirements;
 - The system may inter-operate with other systems that generate design requirements;
 - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
 - This may be the consequence of a regulatory requirement.



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. This means that the requirements can be understood by users and customers.

Guidelines for writing requirements



- ✧ 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.



Problems with natural language

✧ 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.

Example requirements for the insulin pump software system



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.*)



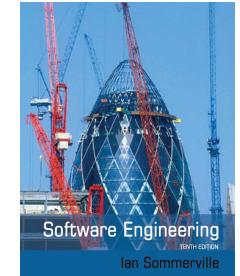
Structured specifications

- ✧ An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
- ✧ This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.



Form-based specifications

- ✧ Definition of the function or entity.
- ✧ Description of inputs and where they come from.
- ✧ Description of outputs and where they go to.
- ✧ Information about the information needed for the computation and other entities used.
- ✧ Description of the action to be taken.
- ✧ Pre and post conditions (if appropriate).
- ✧ The side effects (if any) of the function.



A structured specification of a requirement for an insulin pump

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 (r_2); the previous two readings (r_0 and r_1).

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

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.



A structured specification of a requirement for an insulin pump

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

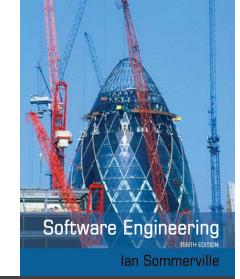
Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition

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

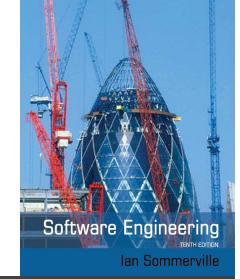
Post-condition r0 is replaced by r1 then r1 is replaced by r2.

Side effects None.



Tabular specification

- ✧ Used to supplement natural language.
- ✧ Particularly useful when you have to define a number of possible alternative courses of action.
- ✧ For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.



Tabular specification of computation for an insulin pump

Condition	Action
Sugar level falling ($r_2 < r_1$)	$\text{CompDose} = 0$
Sugar level stable ($r_2 = r_1$)	$\text{CompDose} = 0$
Sugar level increasing and rate of increase decreasing $((r_2 - r_1) < (r_1 - r_0))$	$\text{CompDose} = 0$
Sugar level increasing and rate of increase stable or increasing $((r_2 - r_1) \geq (r_1 - r_0))$	$\text{CompDose} = \text{round}((r_2 - r_1)/4)$ If rounded result = 0 then $\text{CompDose} = \text{MinimumDose}$

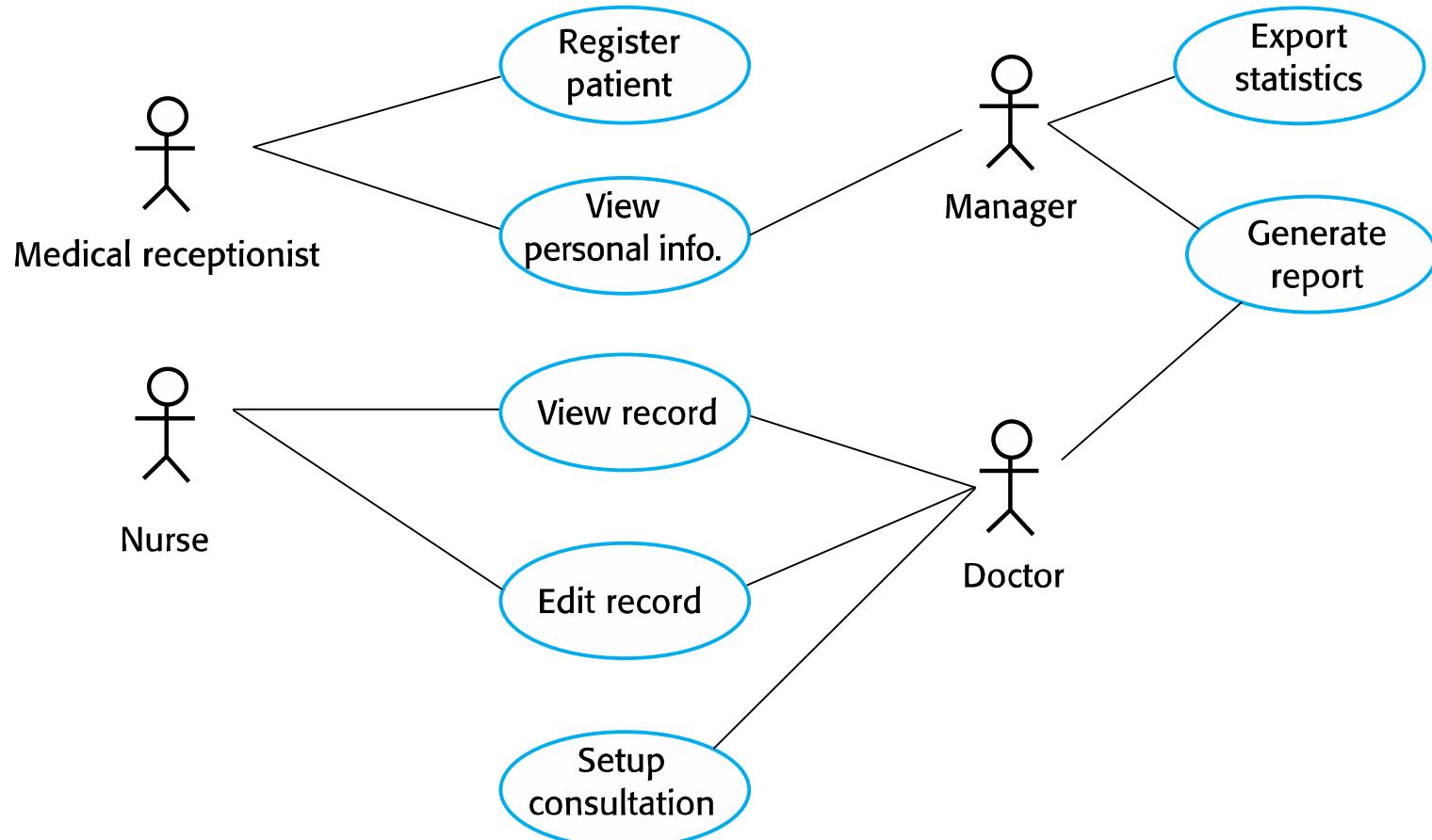


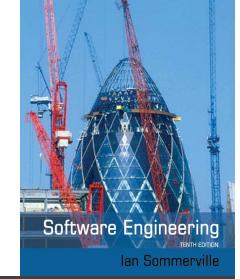
Use cases

- ✧ Use-cases are a kind of scenario that are included in the UML.
- ✧ Use cases 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 supplemented by more detailed tabular description (see Chapter 5).
- ✧ UML sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.



Use cases for the Mentcare system



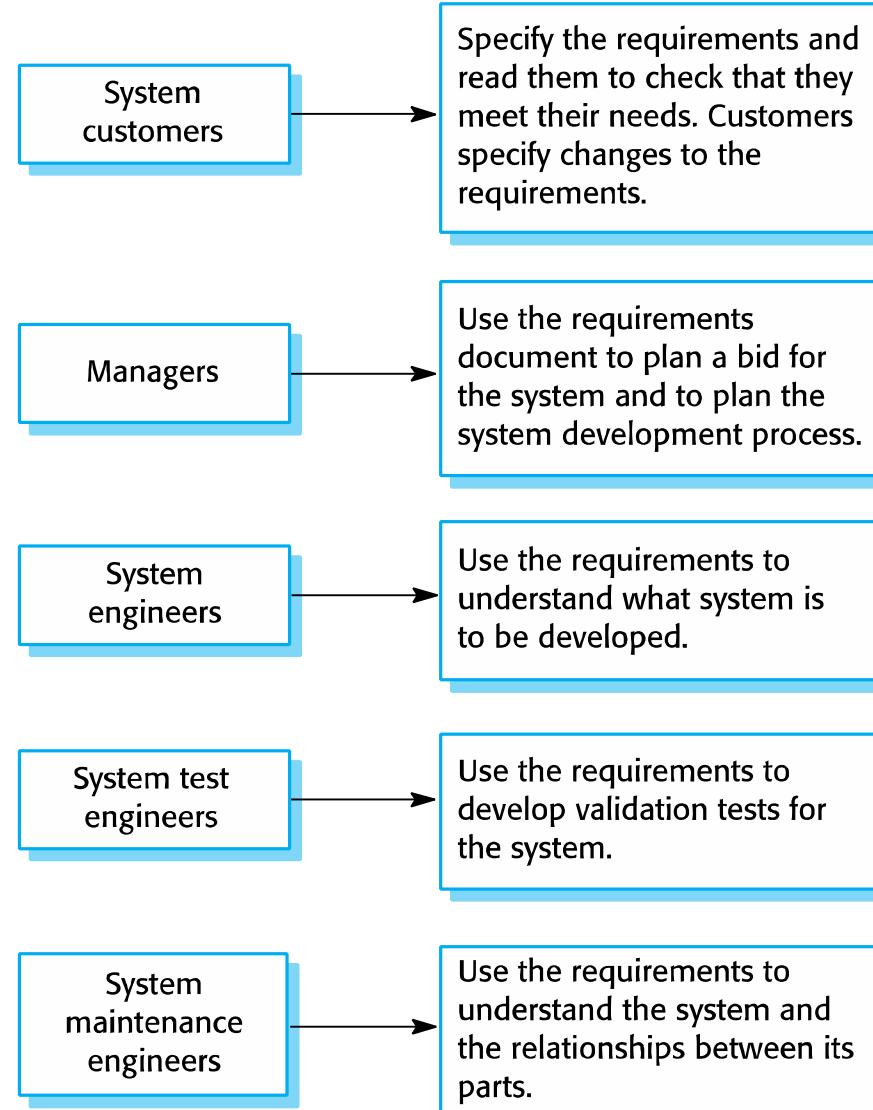


The software requirements document

- ✧ The software requirements document is the 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 a requirements document





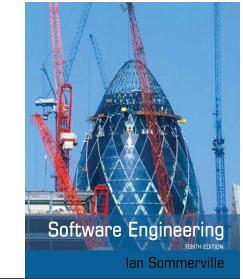
Requirements document variability

- ✧ Information in requirements document depends on type of system and the approach to development used.
- ✧ Systems developed incrementally will, typically, have less detail in the requirements document.
- ✧ Requirements documents standards have been designed e.g. IEEE standard. These are mostly applicable to the requirements for large systems engineering projects.

The structure of a requirements document



Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.

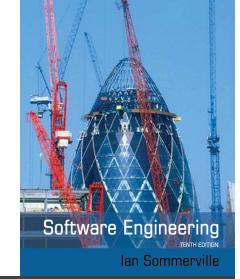


The structure of a requirements document

Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.



Requirements validation



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

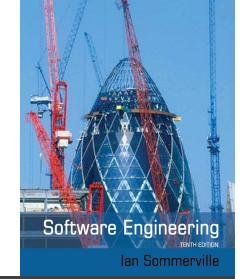


✧ Requirements reviews

- Systematic manual analysis of the requirements.

✧ Prototyping

- Using an executable model of the system to check requirements.



Review checks

✧ Verifiability

- Is the requirement realistically testable?

✧ Comprehensibility

- Is the requirement properly understood?

✧ Traceability

- Is the origin of the requirement clearly stated?

✧ Adaptability

- Can the requirement be changed without a large impact on other requirements?



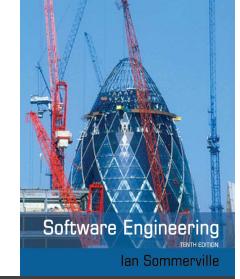
Key points

- ✧ Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- ✧ Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- ✧ Non-functional requirements often constrain the system being developed and the development process being used.
- ✧ They often relate to the emergent properties of the system and therefore apply to the system as a whole.



Key points

- ✧ The requirements engineering process is an iterative process that includes requirements elicitation, specification and validation.
- ✧ Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.
- ✧ You can use a range of techniques for requirements elicitation including interviews and ethnography. User stories and scenarios may be used to facilitate discussions.



Key points

- ✧ Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- ✧ The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.



Key points

- ✧ Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.
- ✧ Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.



Chapter 5 – System Modeling

Topics covered



- ✧ Context models
- ✧ Interaction models
- ✧ Structural models
- ✧ Behavioral models
- ✧ Model-driven engineering

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.
- ✧ System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
- ✧ **System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.**

Existing and planned system models



- ✧ **Models of the existing system are used during requirements engineering.** They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
- ✧ Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.
- ✧ In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.



System perspectives

- ✧ An external perspective, where you model the context or environment of the system.
- ✧ An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
- ✧ A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- ✧ A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.



UML diagram types

- ✧ Activity diagrams, which show the activities involved in a process or in data processing .
- ✧ Use case diagrams, which show the interactions between a system and its environment.
- ✧ Sequence diagrams, which show interactions between actors and the system and between system components.
- ✧ Class diagrams, which show the object classes in the system and the associations between these classes.
- ✧ State diagrams, which show how the system reacts to internal and external events.



Use of graphical models

- ✧ As a means of facilitating discussion about an existing or proposed system
 - Incomplete and incorrect models are OK as their role is to support discussion.
- ✧ As a way of documenting an existing system
 - Models should be an accurate representation of the system but need not be complete.
- ✧ As a detailed system description that can be used to generate a system implementation
 - Models have to be both correct and complete.



Context models

Context models



- ✧ Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- ✧ Social and organisational concerns may affect the decision on where to position system boundaries.
- ✧ Architectural models show the system and its relationship with other systems.

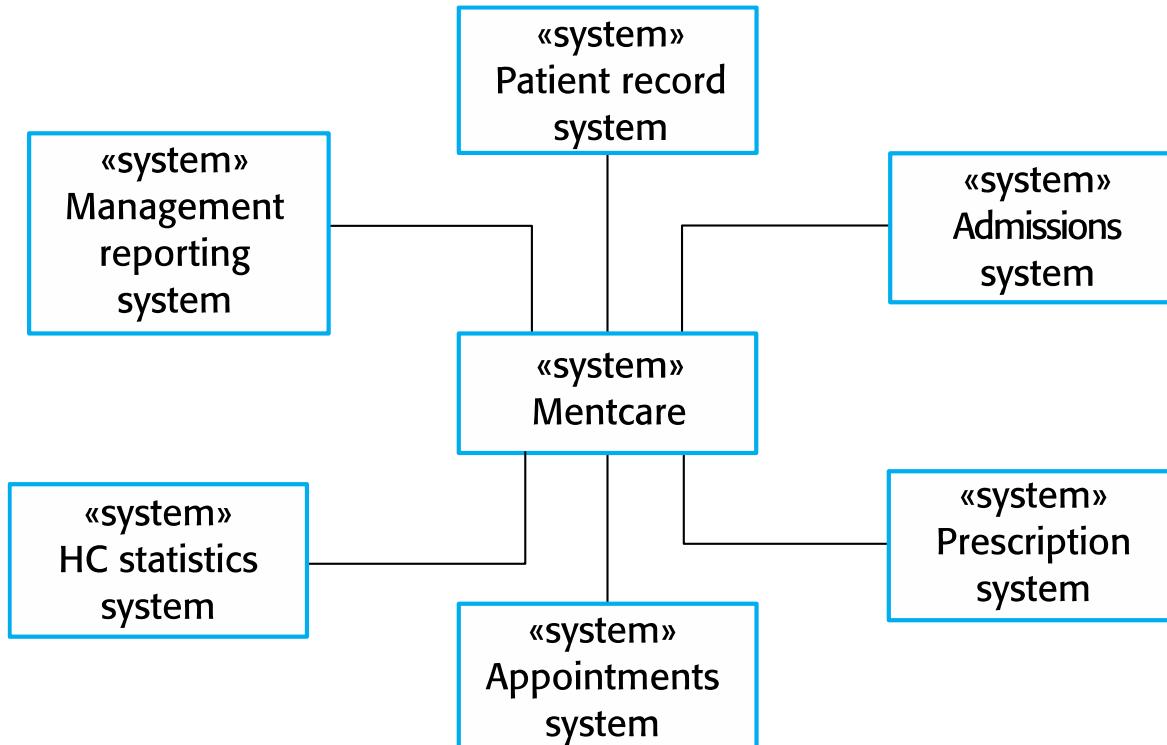


System boundaries

- ✧ System boundaries are established to define what is inside and what is outside the system.
 - They show other systems that are used or depend on the system being developed.
- ✧ The position of the system boundary has a profound effect on the system requirements.
- ✧ Defining a system boundary is a political judgment
 - There may be pressures to develop system boundaries that increase / decrease the influence or workload of different parts of an organization.



The context of the Mentcare system

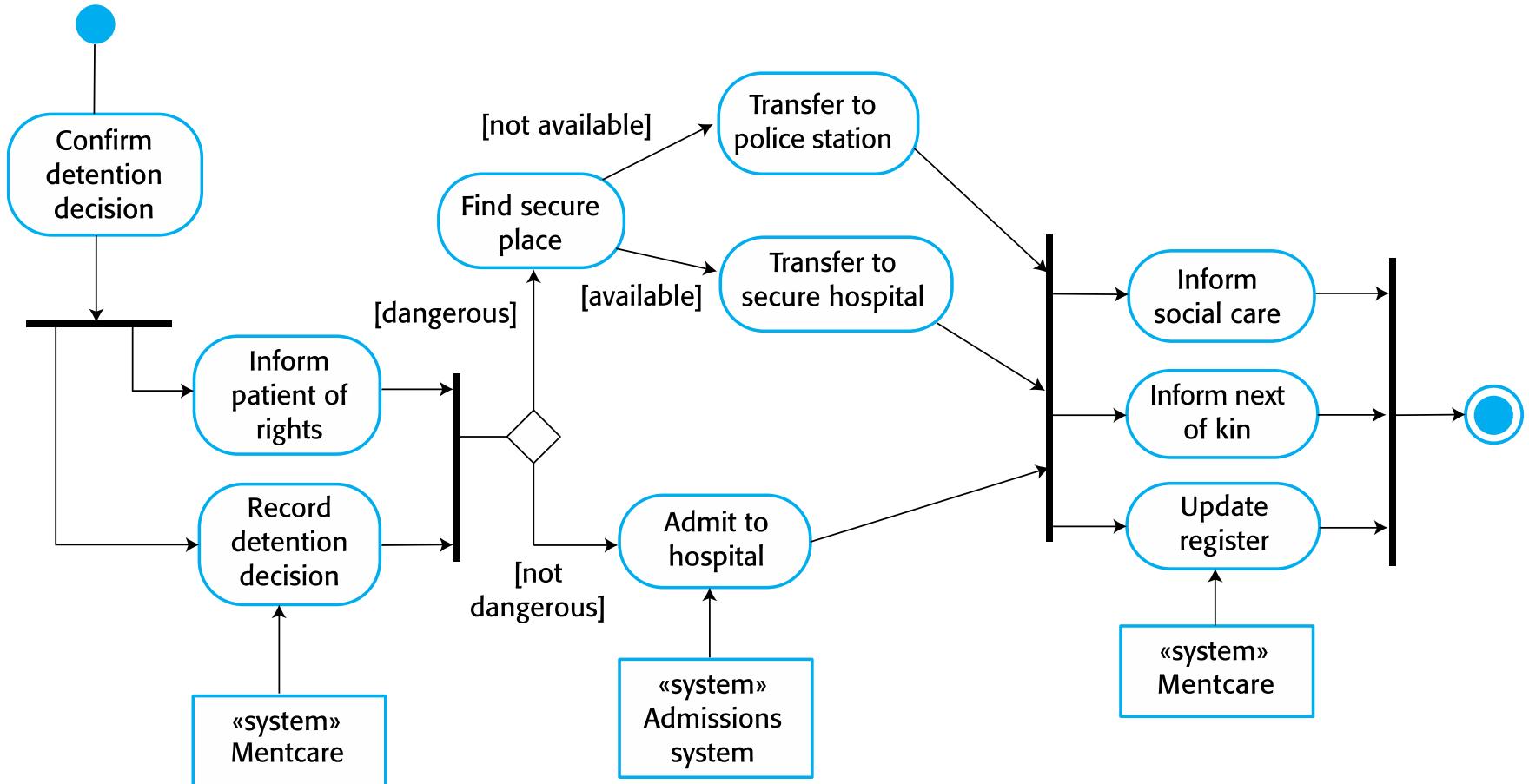


Process perspective



- ✧ Context models simply show the other systems in the environment, not how the system being developed is used in that environment.
- ✧ Process models reveal how the system being developed is used in broader business processes.
- ✧ UML activity diagrams may be used to define business process models.

Process model of involuntary detention





Interaction models

Interaction models



- ✧ Modeling user interaction is important as it 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.
- ✧ Use case diagrams and sequence diagrams may be used for interaction modeling.

Use case modeling



- ✧ Use cases were developed originally to support requirements elicitation and now incorporated into the UML.
- ✧ Each use case represents a discrete task that involves external interaction with a system.
- ✧ Actors in a use case may be people or other systems.
- ✧ Represented diagrammatically to provide an overview of the use case and in a more detailed textual form.

Transfer-data use case



✧ A use case in the Mentcare system



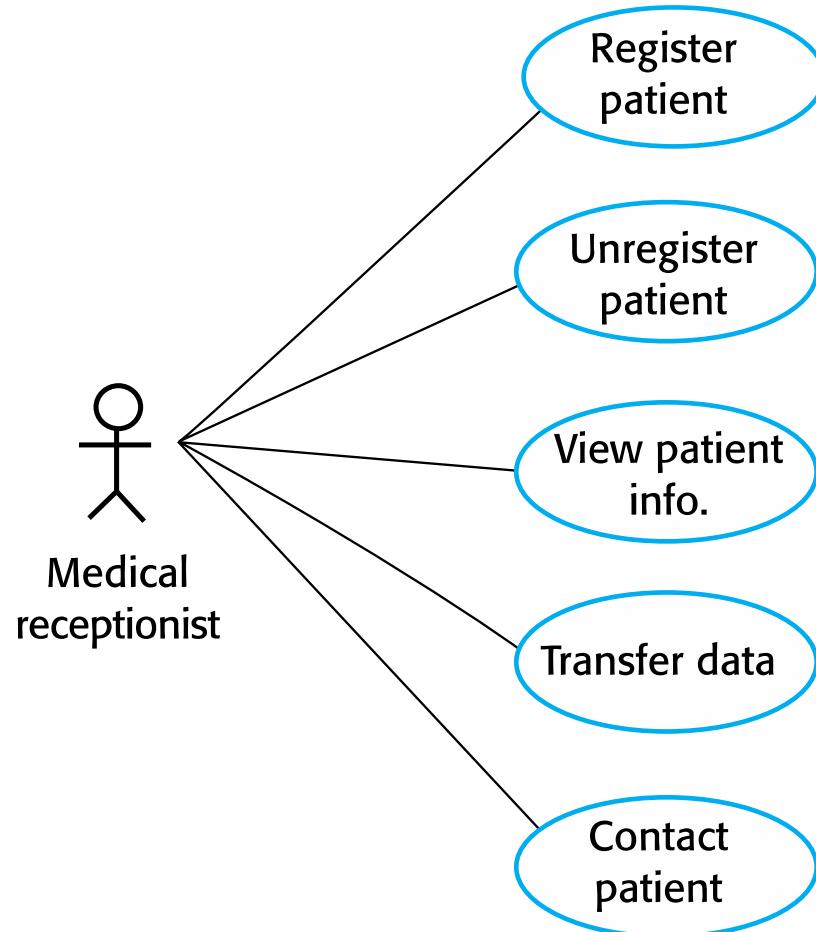
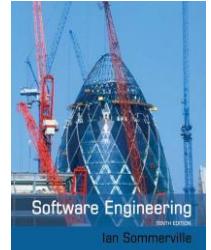
Tabular description of the ‘Transfer data’ use-case



MHC-PMS: Transfer data

Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the Mentcase 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.

Use cases in the Mentcare system involving the role 'Medical Receptionist'



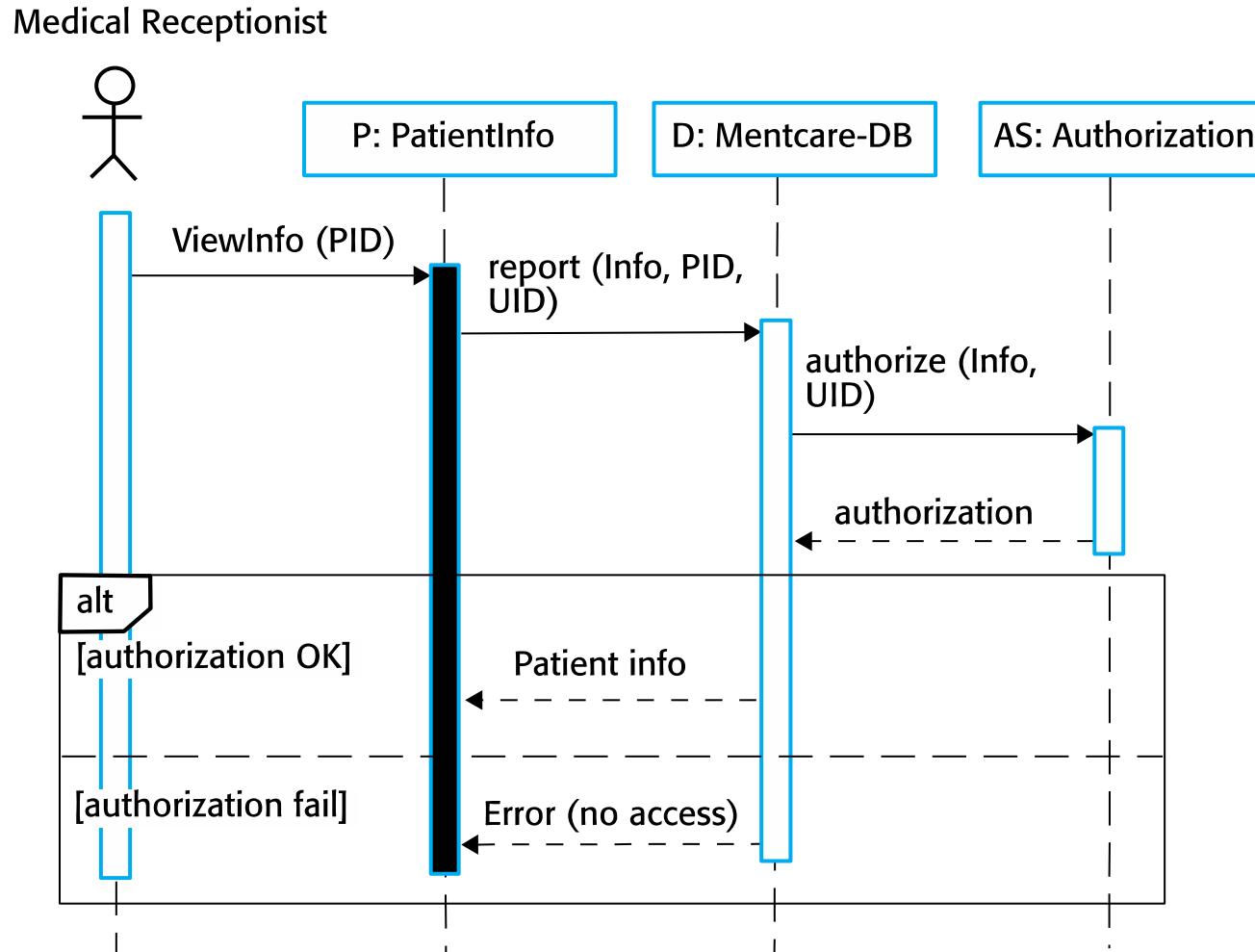
Sequence diagrams

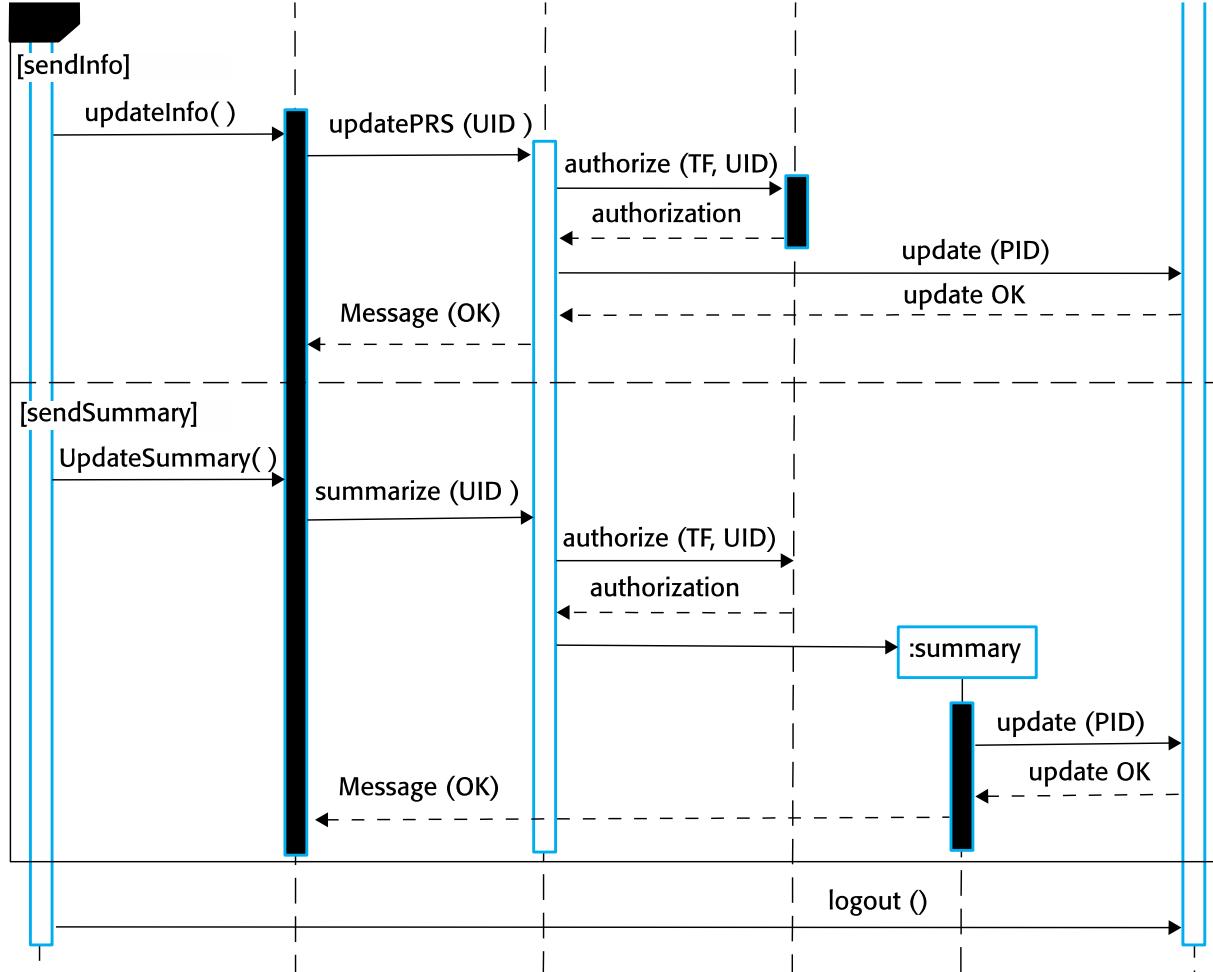
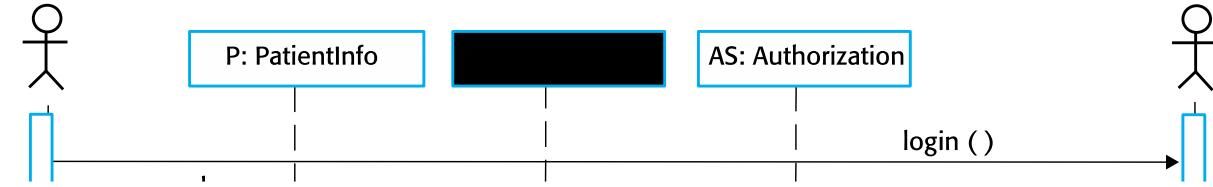


- ✧ Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
- ✧ A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- ✧ The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- ✧ Interactions between objects are indicated by annotated arrows.



Sequence diagram for View patient information





Sequence diagram for Transfer Data



Structural models

Structural models



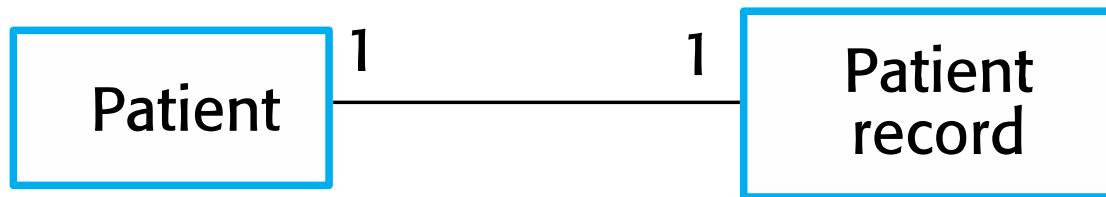
- ✧ Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- ✧ Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- ✧ You create structural models of a system when you are discussing and designing the system architecture.

Class diagrams

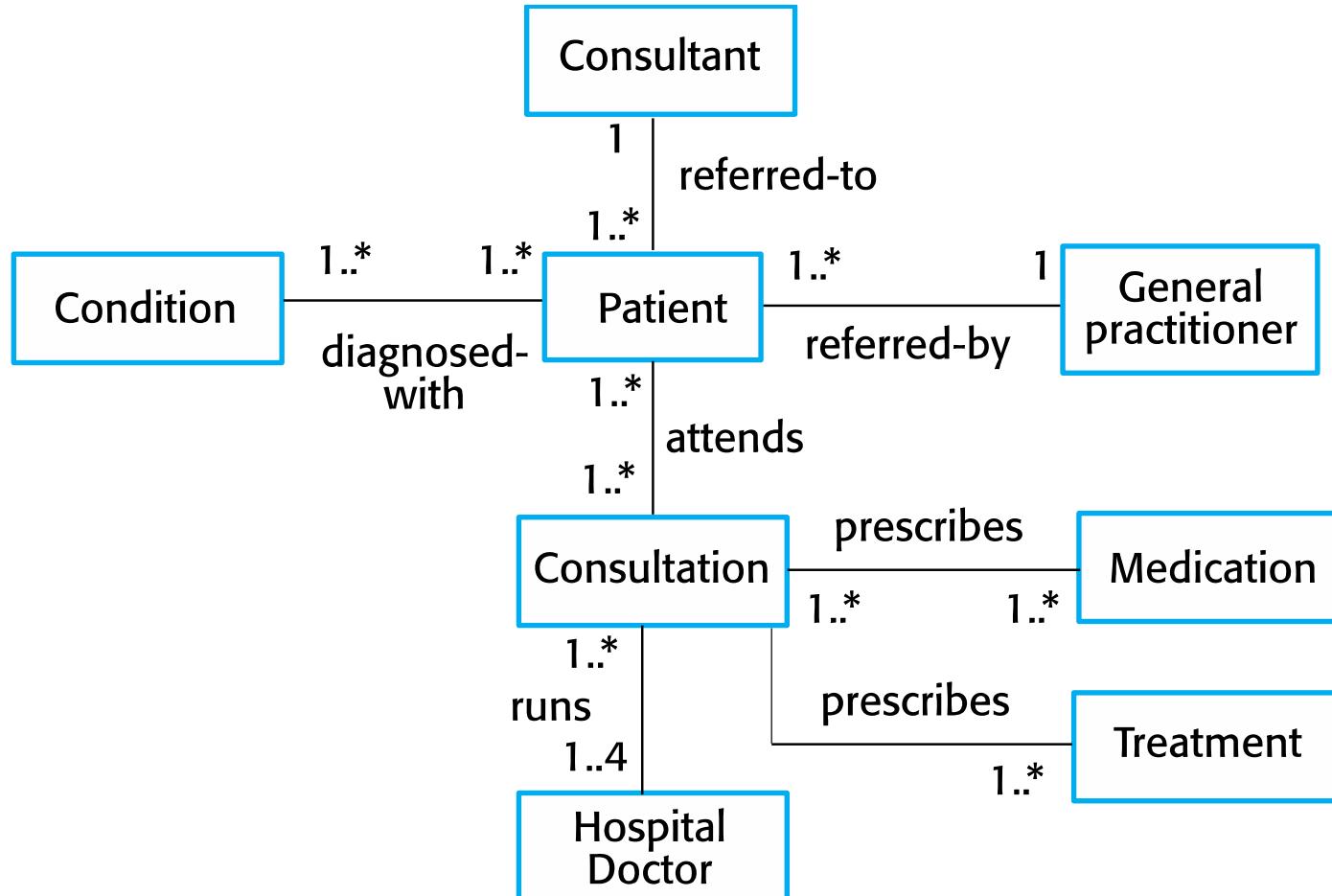


- ✧ Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- ✧ An object class can be thought of as a general definition of one kind of system object.
- ✧ An association is a link between classes that indicates that there is some relationship between these classes.
- ✧ When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

UML classes and association



Classes and associations in the MHC-PMS



The Consultation class



Consultation

Doctors
Date
Time
Clinic
Reason
Medication prescribed
Treatment prescribed
Voice notes
Transcript
...

New ()
Prescribe ()
RecordNotes ()
Transcribe ()
...

Generalization



- ✧ Generalization is an everyday technique that we use to manage complexity.
- ✧ Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- ✧ This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.

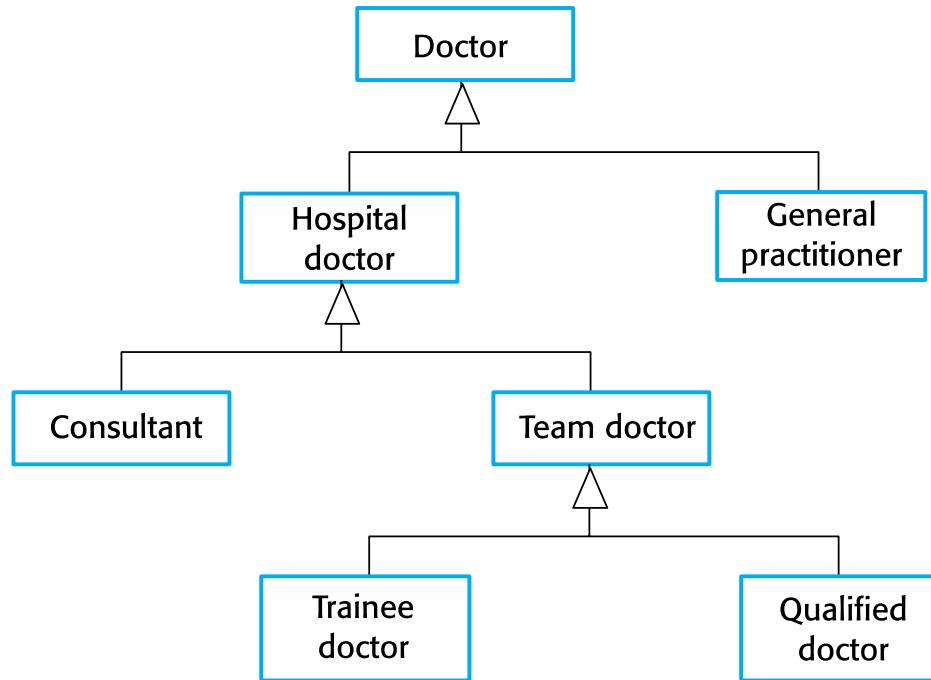
Generalization



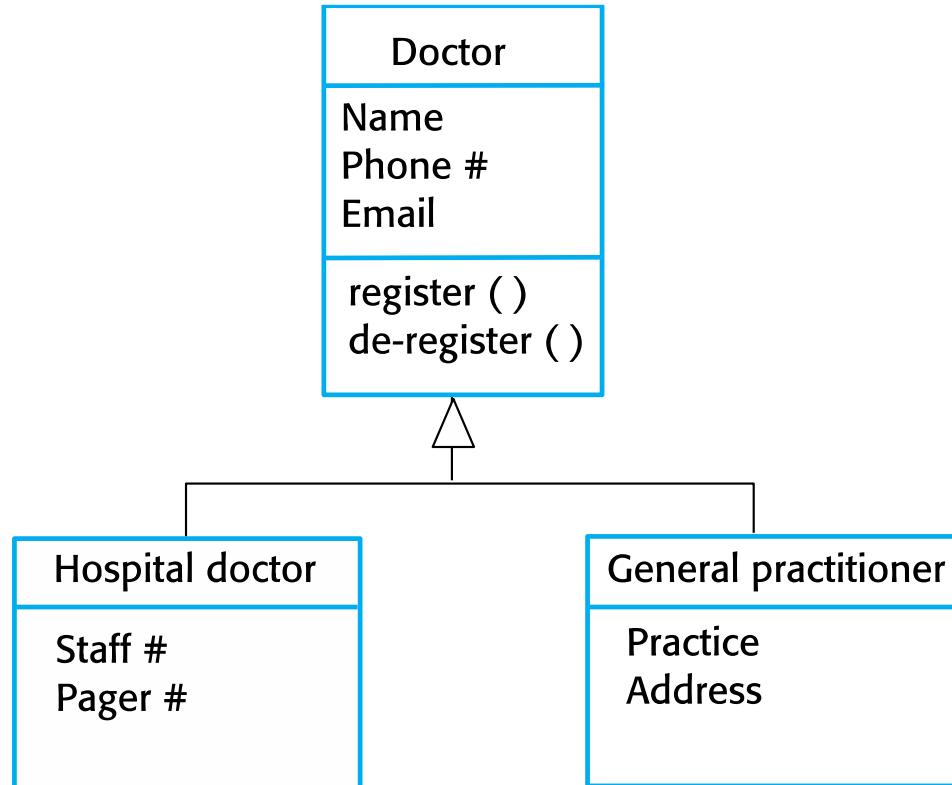
- ✧ In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- ✧ In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
- ✧ In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- ✧ The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.



A generalization hierarchy



A generalization hierarchy with added detail

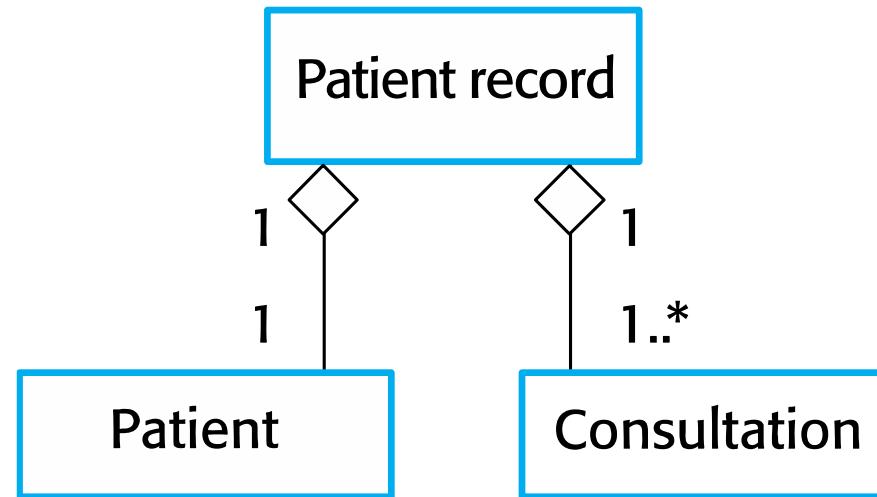


Object class aggregation models



- ✧ An aggregation model shows how classes that are collections are composed of other classes.
- ✧ Aggregation models are similar to the part-of relationship in semantic data models.

The aggregation association





Behavioral models



Behavioral models

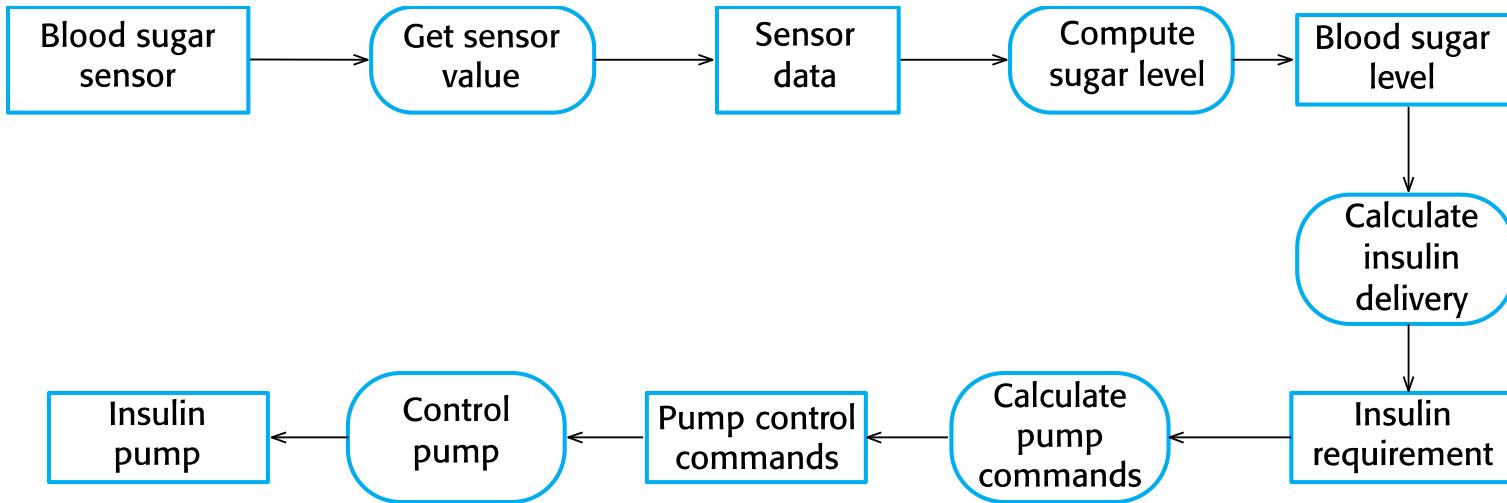
- ✧ Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- ✧ You can think of these stimuli as being of two types:
 - **Data** Some data arrives that has to be processed by the system.
 - **Events** Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

Data-driven modeling

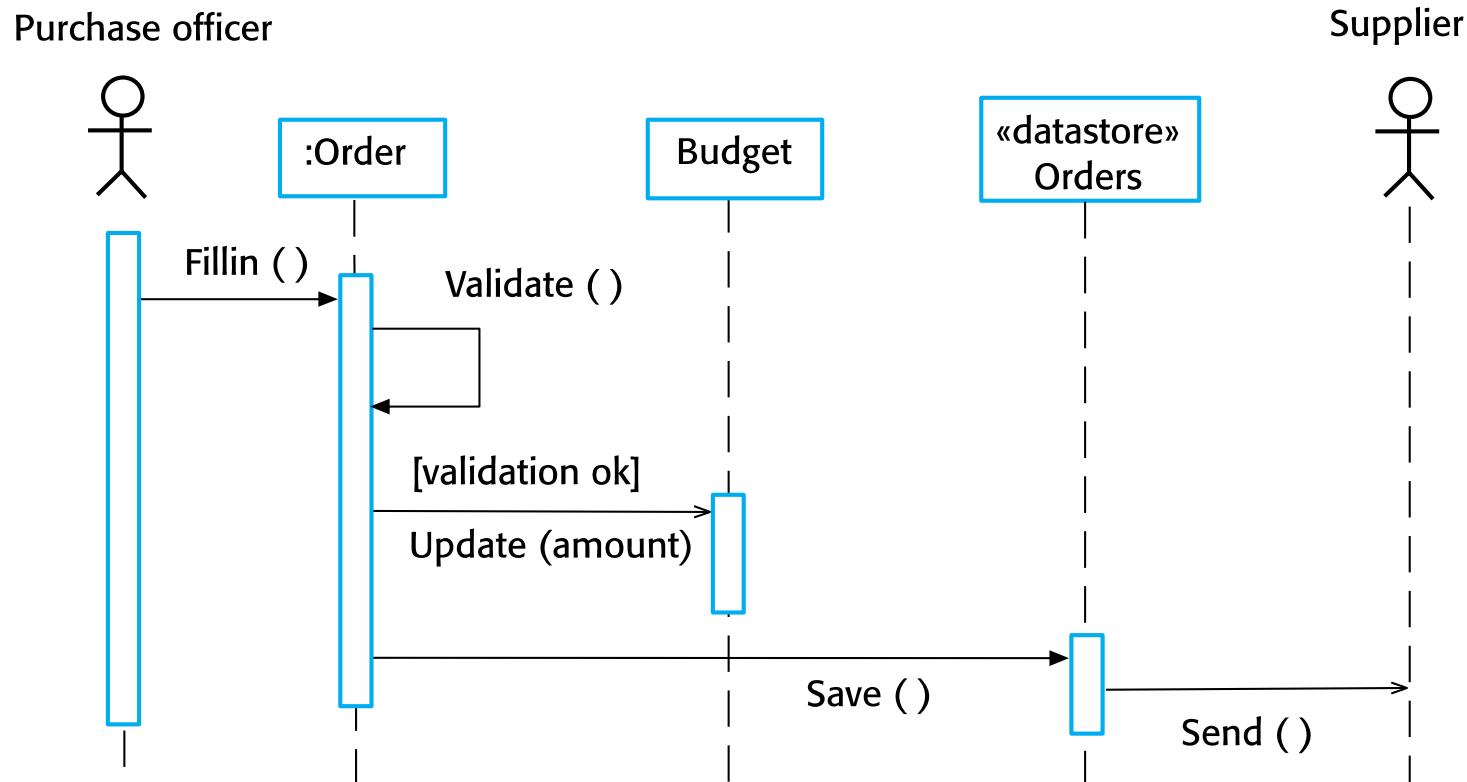


- ✧ Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- ✧ Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
- ✧ They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

An activity model of the insulin pump's operation



Order processing



Event-driven modeling



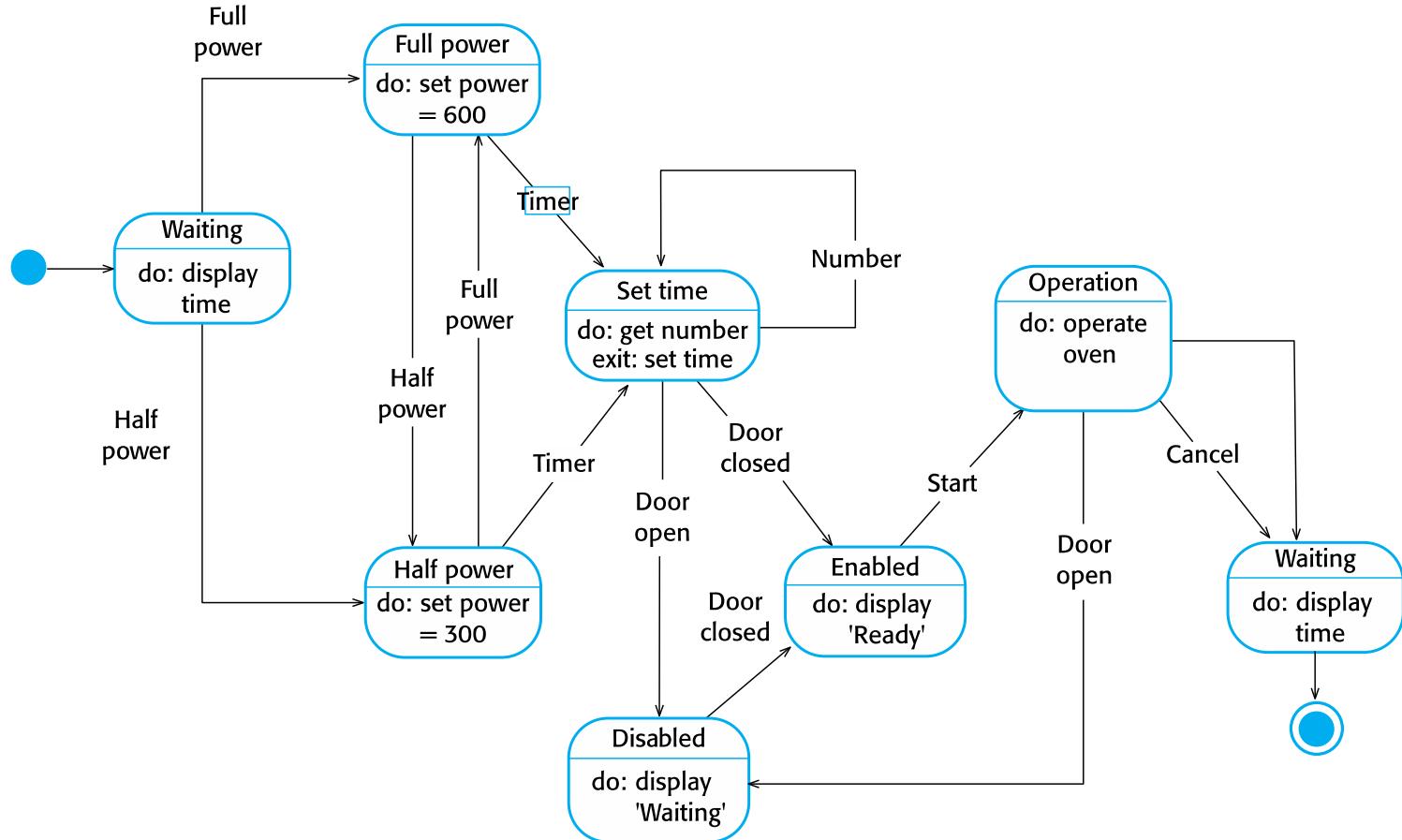
- ✧ Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone.
- ✧ Event-driven modeling shows how a system responds to external and internal events.
- ✧ It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

State machine models



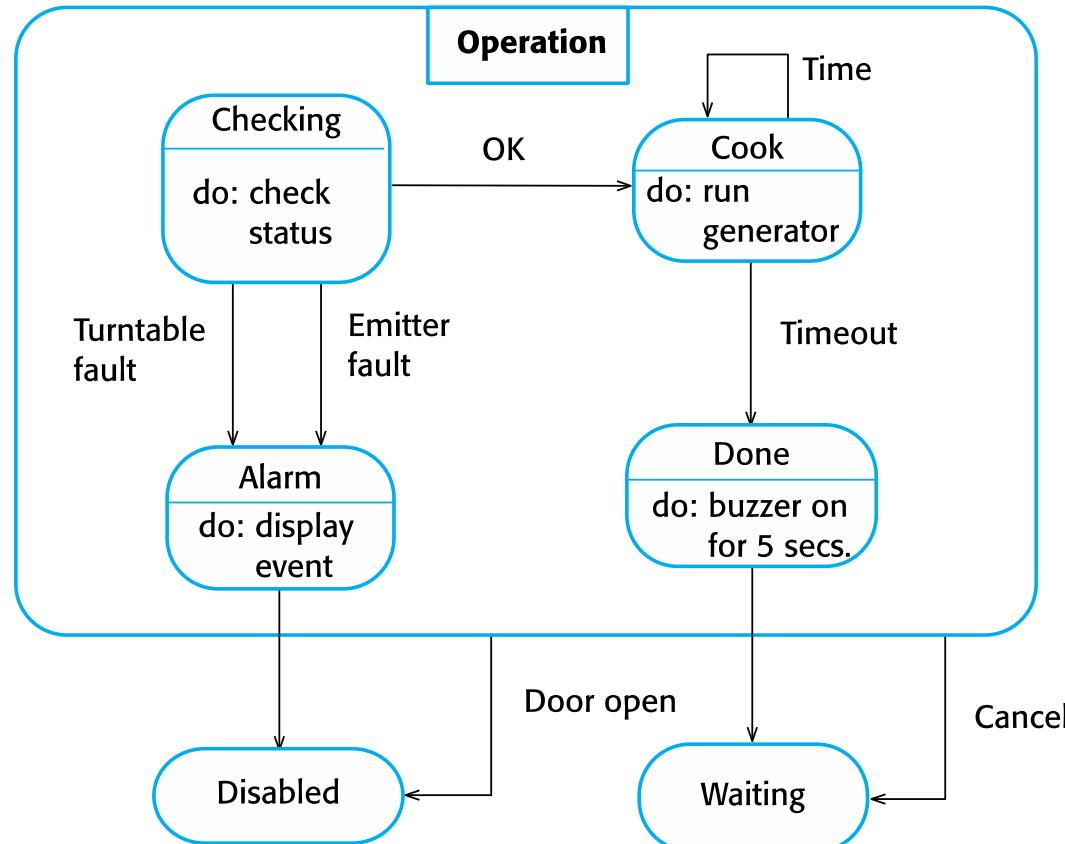
- ✧ These model the behaviour of the system in response to external and internal events.
- ✧ They show the system's responses to stimuli so are often used for modelling real-time systems.
- ✧ State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- ✧ Statecharts are an integral part of the UML and are used to represent state machine models.

State diagram of a microwave oven





Microwave oven operation



States and stimuli for the microwave oven (a)



State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

States and stimuli for the microwave oven (b)



Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.



Model-driven engineering

Model-driven engineering



- ✧ Model-driven engineering (MDE) is an **approach to software development where models rather than programs are the principal outputs of the development process.**
- ✧ The programs that execute on a hardware/software platform are then generated automatically from the models.

Usage of model-driven engineering



✧ Pros

- Allows systems to be considered at higher levels of abstraction
- Generating code automatically means that it is cheaper to adapt systems to new platforms.

✧ Cons

- Models for abstraction and not necessarily right for implementation.
- Savings from generating code may be outweighed by the costs of developing translators for new platforms.

Model driven architecture



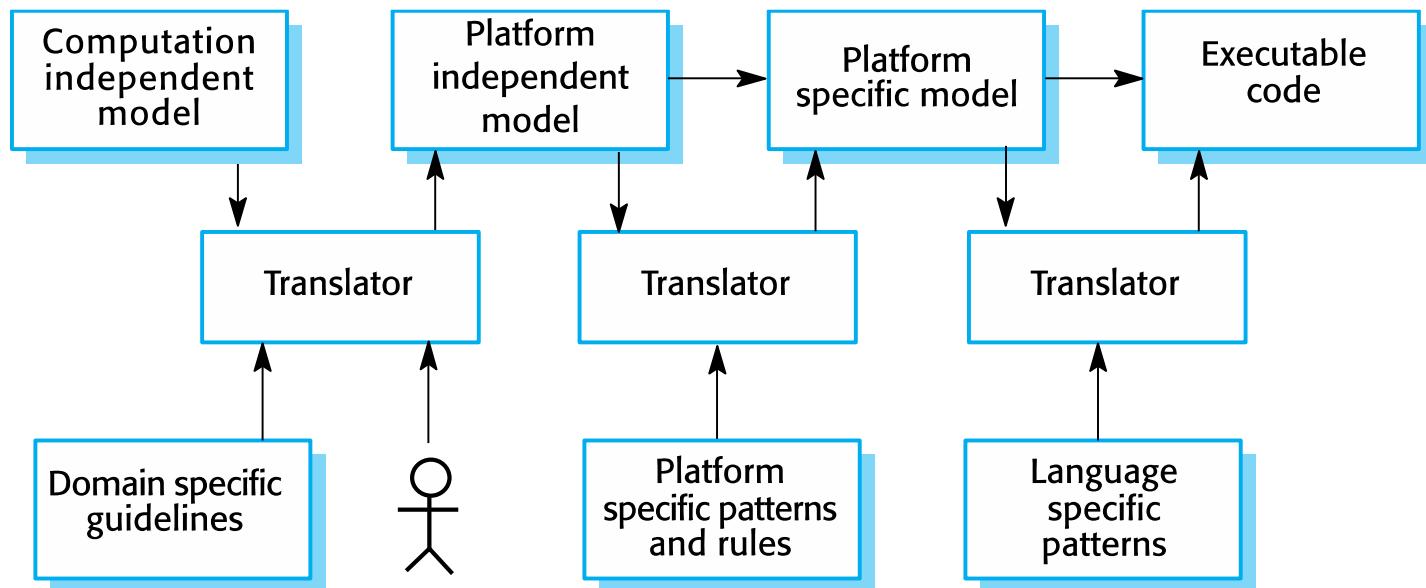
- ✧ Model-driven architecture (MDA) was the precursor of more general model-driven engineering
- ✧ MDA is a model-focused approach **to software design and implementation that uses a subset of UML models to describe a system.**
- ✧ Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.



Types of model

- ✧ A computation independent model (CIM)
 - These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
- ✧ A platform independent model (PIM)
 - These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
- ✧ Platform specific models (PSM)
 - These are transformations of the platform-independent model with 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





Key points

- ✧ A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behavior.
- ✧ Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
- ✧ Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- ✧ Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.



Key points

- ✧ Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system, or by the events that stimulate responses from a system.
- ✧ Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- ✧ State diagrams are used to model a system's behavior in response to internal or external events.
- ✧ Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

DATA FLOW DIAGRAM

1. KONSEP PERANCANGAN TERSTRUKTUR

Pendekatan perancangan terstruktur dimulai dari awal 1970. Pendekatan terstruktur dilengkapi dengan alat-alat (*tools*) dan teknik-teknik (*techniques*) yang dibutuhkan dalam pengembangan sistem, sehingga hasil akhir dari sistem yang dikembangkan akan diperoleh sistem yang strukturnya didefinisikan dengan baik dan jelas.

Melalui pendekatan terstruktur, permasalahan yang kompleks di organisasi dapat dipecahkan dan hasil dari sistem akan mudah untuk dipelihara, fleksibel, lebih memuaskan pemakainya, mempunyai dokumentasi yang baik, tepat waktu, sesuai dengan anggaran biaya pengembangan, dapat meningkatkan produktivitas dan kualitasnya akan lebih baik (bebas kesalahan)

2. DATA FLOW DIAGRAM (DFD)

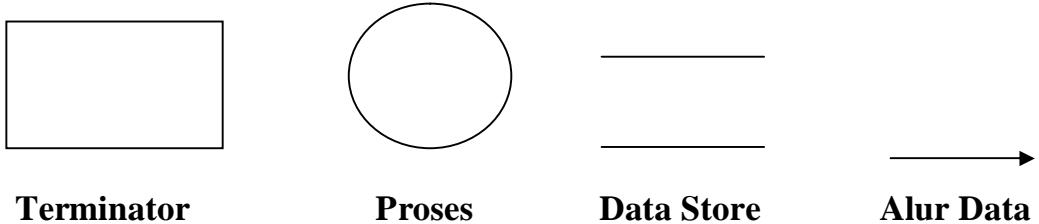
Data Flow Diagram (DFD) adalah alat pembuatan model yang memungkinkan profesional sistem untuk menggambarkan sistem sebagai suatu jaringan proses fungsional yang dihubungkan satu sama lain dengan alur data, baik secara manual maupun komputerisasi. DFD ini sering disebut juga dengan nama Bubble chart, Bubble diagram, model proses, diagram alur kerja, atau model fungsi.

DFD ini adalah salah satu alat pembuatan model yang sering digunakan, khususnya bila fungsi-fungsi sistem merupakan bagian yang lebih penting dan kompleks dari pada data yang dimanipulasi oleh sistem. Dengan kata lain, DFD adalah alat pembuatan model yang memberikan penekanan hanya pada fungsi sistem.

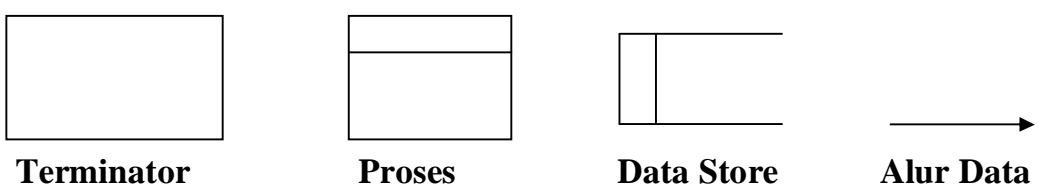
DFD ini merupakan alat perancangan sistem yang berorientasi pada alur data dengan konsep dekomposisi dapat digunakan untuk penggambaran analisa maupun rancangan sistem yang mudah dikomunikasikan oleh profesional sistem kepada pemakai maupun pembuat program.

3. KOMPONEN DATA FLOW DIAGRAM

Menurut Yourdan dan DeMarco



Menurut Gene dan Serson

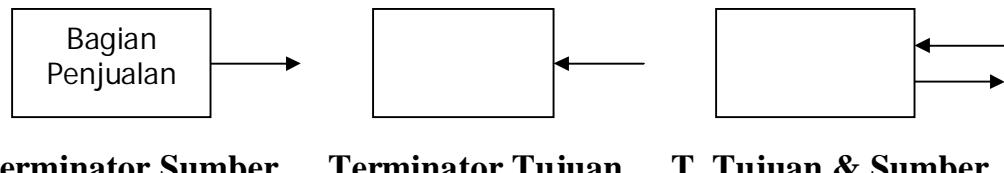


3.1. Komponen Terminator / Entitas Luar

Terminator mewakili entitas eksternal yang berkomunikasi dengan sistem yang sedang dikembangkan. Biasanya terminator dikenal dengan nama entitas luar (*external entity*).

Terdapat dua jenis terminator :

1. Terminator Sumber (*source*) : merupakan terminator yang menjadi sumber.
2. Terminator Tujuan (*sink*) : merupakan terminator yang menjadi tujuan data / informasi sistem.



Terminator dapat berupa orang, sekelompok orang, organisasi, departemen di dalam organisasi, atau perusahaan yang sama tetapi di luar kendali sistem yang sedang dibuat modelnya.

Terminator dapat juga berupa departemen, divisi atau sistem di luar sistem yang berkomunikasi dengan sistem yang sedang dikembangkan.

Komponen terminator ini perlu **diberi nama** sesuai dengan dunia luar yang berkomunikasi dengan sistem yang sedang dibuat modelnya, dan biasanya menggunakan **kata benda**, misalnya **Bagian Penjualan, Dosen, Mahasiswa**.

Ada tiga hal penting yang harus diingat tentang terminator :

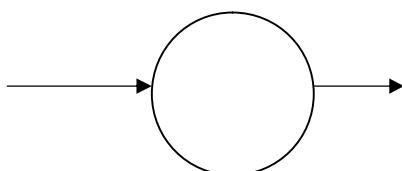
1. Terminator merupakan bagian/lingkungan luar sistem. Alur data yang menghubungkan terminator dengan berbagai proses sistem, menunjukkan hubungan sistem dengan dunia luar.
2. Profesional sistem tidak dapat mengubah isi atau cara kerja organisasi, atau prosedur yang berkaitan dengan terminator.
3. Hubungan yang ada antar terminator yang satu dengan yang lain tidak digambarkan pada DFD.

3.2. Komponen Proses

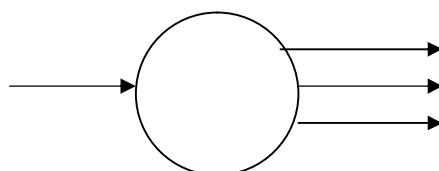
Komponen proses menggambarkan bagian dari sistem yang mentransformasikan input menjadi output.

Proses diberi nama untuk menjelaskan proses/kegiatan apa yang sedang/akan dilaksanakan. Pemberian nama proses dilakukan dengan menggunakan **kata kerja transitif** (kata kerja yang membutuhkan obyek), seperti **Menghitung Gaji, Mencetak KRS, Menghitung Jumlah SKS**.

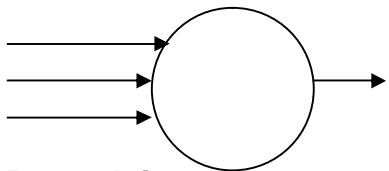
Ada empat kemungkinan yang dapat terjadi dalam proses sehubungan dengan input dan output :



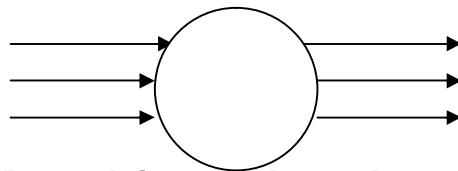
1 input & 1 output



1 input & banyak output



Banyak input & 1 output

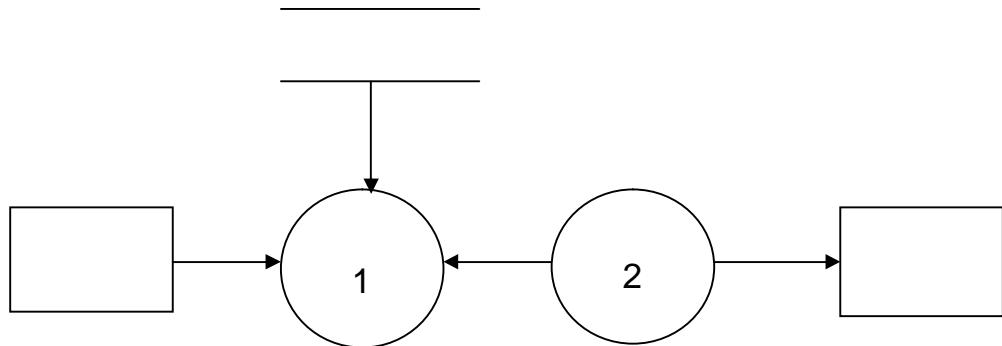


Banyak input & banyak output

Ada beberapa hal yang perlu diperhatikan tentang proses :

- Ø Proses harus memiliki input dan output.
- Ø Proses dapat dihubungkan dengan komponen terminator, data store atau proses melalui alur data.
- Ø Sistem/bagian/divisi/departemen yang sedang dianalisis oleh profesional sistem digambarkan dengan komponen proses.

Berikut ini merupakan suatu contoh proses yang salah :



Gambar 1. Contoh proses

Umumnya kesalahan proses di DFD adalah :

1. Proses mempunyai input tetapi tidak menghasilkan output. Kesalahan ini disebut dengan **black hole** (lubang hitam), karena data masuk ke dalam proses dan lenyap tidak berbekas seperti dimasukkan ke dalam lubang hitam (*lihat proses 1*).
2. Proses menghasilkan output tetapi tidak pernah menerima input. Kesalahan ini disebut dengan **miracle** (ajaib), karena ajaib dihasilkan output tanpa pernah menerima input (*lihat proses 2*).

3.3. Komponen Data Store

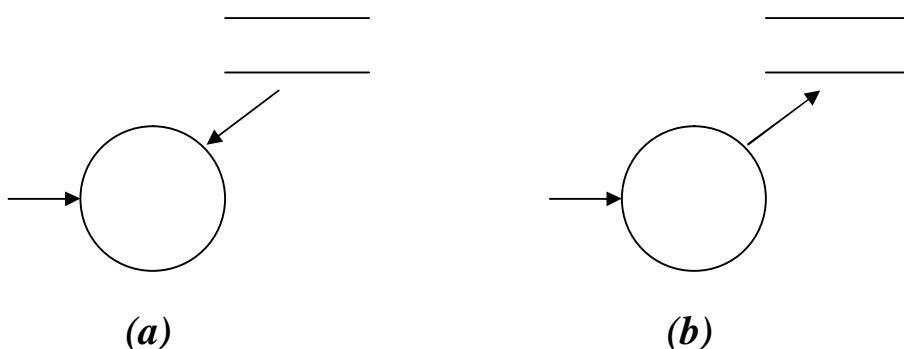
Komponen ini digunakan untuk membuat model sekumpulan paket data dan **diberi nama** dengan **kata benda jamak**, misalnya **Mahasiswa**.

Data store ini biasanya berkaitan dengan penyimpanan-penyimpanan, seperti file atau database yang berkaitan dengan penyimpanan secara komputerisasi, misalnya file disket, file harddisk, file pita magnetik. Data store juga berkaitan dengan penyimpanan secara manual seperti buku alamat, file folder, dan agenda.

Suatu data store dihubungkan dengan alur data **hanya pada komponen proses**, tidak dengan komponen DFD lainnya. Alur data yang menghubungkan data store dengan suatu proses mempunyai pengertian sebagai berikut :

- **Alur data dari data store** yang berarti sebagai pembacaan atau pengaksesan satu paket tunggal data, lebih dari satu paket data, sebagian dari satu paket tunggal data, atau sebagian dari lebih dari satu paket data untuk suatu proses (*lihat gambar 2 (a)*).
- **Alur data ke data store** yang berarti sebagai pengupdatean data, seperti menambah satu paket data baru atau lebih, menghapus satu paket atau lebih, atau mengubah/memodifikasi satu paket data atau lebih (*lihat gambar 2 (b)*).

Pada pengertian pertama jelaslah bahwa data store tidak berubah, jika suatu paket data/informasi berpindah dari data store ke suatu proses. Sebaliknya pada pengertian kedua data store berubah sebagai hasil alur yang memasuki data store. Dengan kata lain, proses alur data bertanggung jawab terhadap perubahan yang terjadi pada data store.



Gambar 2. Implementasi data store

3.4. Komponen Data Flow / Alur Data

Suatu data flow / alur data digambarkan dengan anak panah, yang menunjukkan arah menuju ke dan keluar dari suatu proses. Alur data ini digunakan untuk menerangkan perpindahan data atau paket data/informasi dari satu bagian sistem ke bagian lainnya.

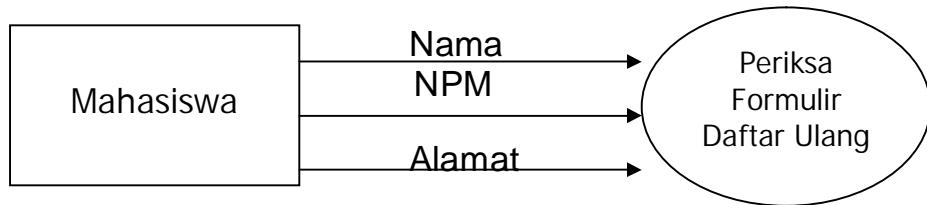
Selain menunjukkan arah, alur data pada model yang dibuat oleh profesional sistem dapat merepresentasikan bit, karakter, pesan, formulir, bilangan real, dan macam-macam informasi yang berkaitan dengan komputer. Alur data juga dapat merepresentasikan data/informasi yang tidak berkaitan dengan komputer.

Alur data perlu **diberi nama** sesuai dengan data/informasi yang dimaksud, biasanya pemberian nama pada alur data dilakukan dengan menggunakan **kata benda**, contohnya **Laporan Penjualan**.

Ada empat konsep yang perlu diperhatikan dalam penggambaran alur data, yaitu :

1. Konsep Paket Data (*Packets of Data*)

Apabila dua data atau lebih mengalir dari *sumber yang sama* menuju ke *tujuan yang sama* dan mempunyai hubungan, dan harus dianggap sebagai satu alur data tunggal, karena data itu mengalir bersama-sama sebagai satu paket.



(a) Konsep paket data yang salah

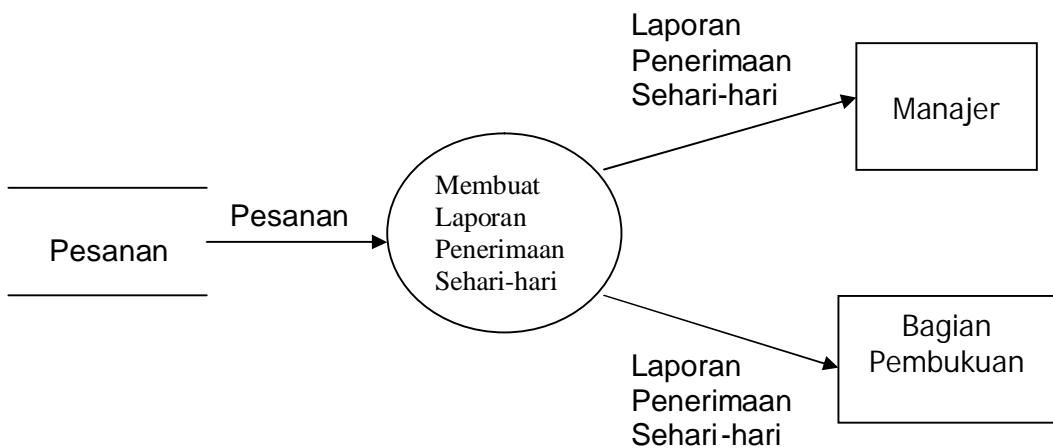


(b) Konsep paket data yang benar

Gambar 3. Konsep paket data

2. Konsep Alur Data Menyebar (*Diverging Data Flow*)

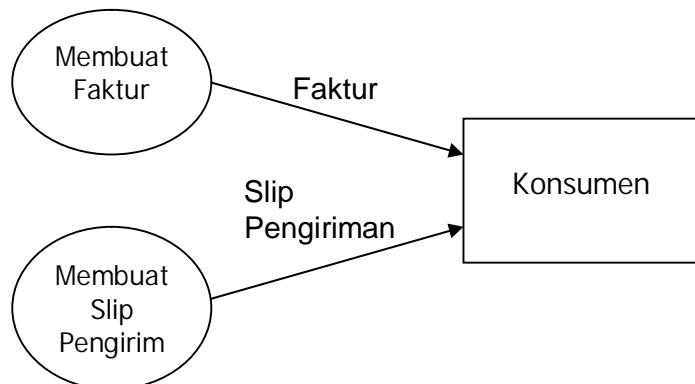
Alur data menyebar menunjukkan sejumlah tembusan paket data yang berasal dari *sumber yang sama* menuju ke *tujuan yang berbeda*, atau paket data yang kompleks dibagi menjadi beberapa elemen data yang dikirim ke tujuan yang berbeda, atau alur data ini membawa paket data yang memiliki nilai yang berbeda yang akan dikirim ke tujuan yang berbeda.



Gambar 4. Konsep alur data menyebar

3. Konsep Alur Data Mengumpul (*Converging Data Flow*)

Beberapa alur data yang *berbeda sumber* bergabung bersama-sama menuju ke *tujuan yang sama*.



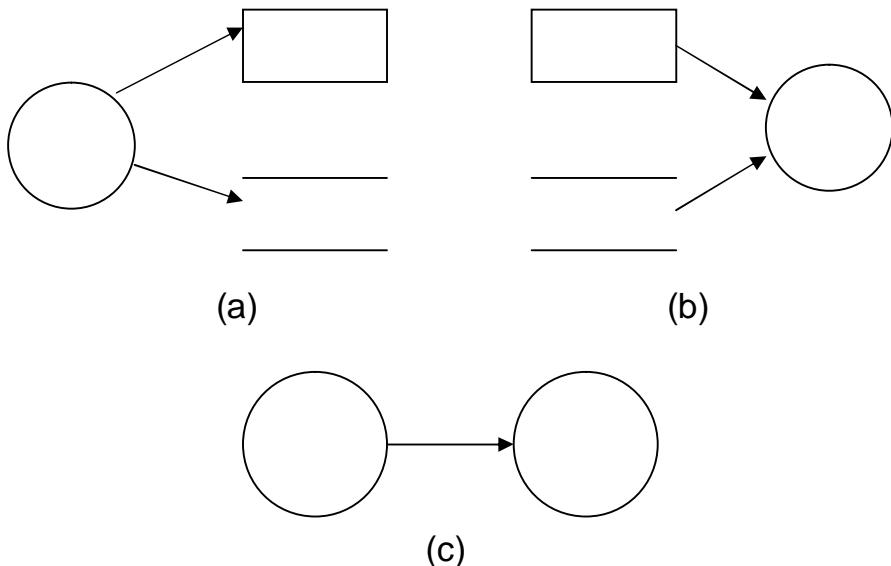
Gambar 5. Konsep alur data mengumpul

4. Konsep Sumber atau Tujuan Alur Data

Semua alur data harus *minimal mengandung satu proses*.

Maksud kalimat ini adalah :

- q Suatu alur data dihasilkan dari suatu proses dan menuju ke suatu *data store* dan/atau *terminator* (*lihat gambar 6 (a)*).
- q Suatu alur data dihasilkan dari suatu *data store* dan/atau *terminator* dan menuju ke suatu proses (*lihat gambar 6 (b)*).
- q Suatu alur data dihasilkan dari suatu proses dan menuju ke suatu proses (*lihat gambar 6 (c)*).



Gambar 6. Konsep sumber atau tujuan alur data

4. BENTUK DATA FLOW DIAGRAM

Terdapat dua bentuk DFD, yaitu **Diagram Alur Data Fisik**, dan **Diagram Alur data Logika**. Diagram alur data fisik lebih menekankan pada bagaimana proses dari sistem diterapkan, sedangkan diagram alur data logika lebih menekankan proses-proses apa yang terdapat di sistem.

4.1. Diagram Alur Data Fisik (DADF)

DADF lebih tepat digunakan untuk menggambarkan sistem yang ada (sistem yang lama). Penekanan dari DADF adalah bagaimana proses-proses dari sistem diterapkan (dengan cara apa, oleh siapa dan dimana), termasuk proses-proses manual.

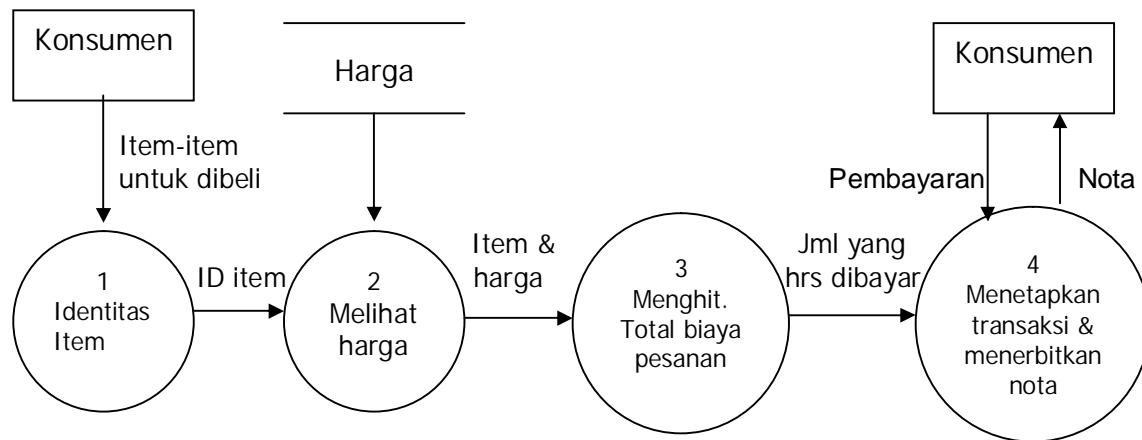
Untuk memperoleh gambaran bagaimana sistem yang ada diterapkan, DADF harus memuat :

1. Proses-proses manual juga digambarkan.
2. Nama dari alur data harus memuat keterangan yang cukup terinci untuk menunjukkan bagaimana pemakai sistem memahami kerja sistem.
3. Simpanan data dapat menunjukkan simpanan non komputer.
4. Nama dari simpanan data harus menunjukkan tipe penerapannya apakah secara manual atau komputerisasi. Secara manual misalnya dapat menunjukkan buku catatan, meja pekerja. Sedang cara komputerisasi misalnya menunjukkan file urut, file database.

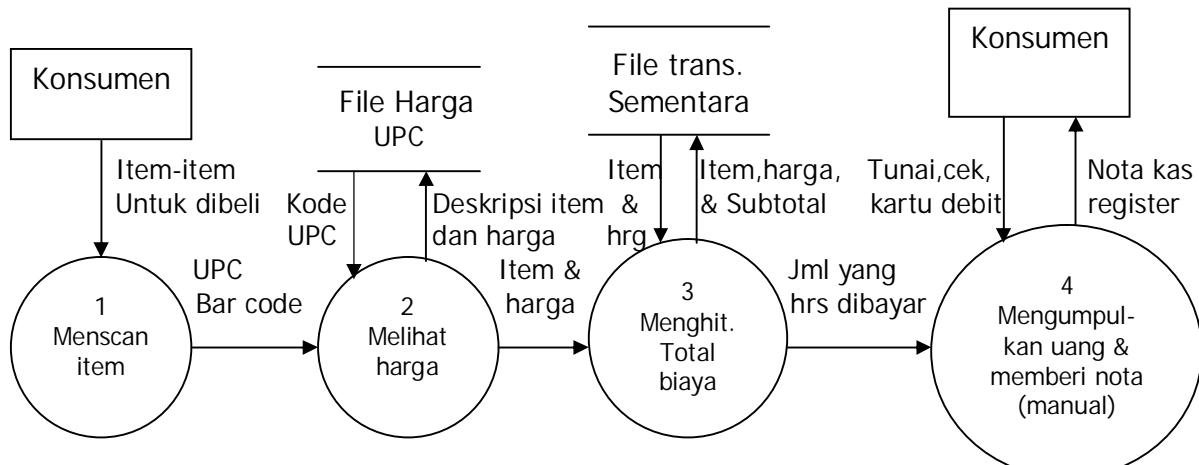
5. Proses harus menunjukkan nama dari pemroses, yaitu orang, departemen, sistem komputer, atau nama program komputer yang mengakses proses tersebut.

4.2. Diagram Alur Data Logika (DADL)

DADL lebih tepat digunakan untuk menggambarkan sistem yang akan diusulkan (sistem yang baru). Untuk sistem komputerisasi, penggambaran DADL hanya menunjukkan kebutuhan proses dari sistem yang diusulkan secara logika, biasanya proses-proses yang digambarkan hanya merupakan proses-proses secara komputer saja.



(a) Diagram Alur Data Logika



(b) Diagram Alur Data Fisik

Gambar 7. DADF dan DADL

5. SYARAT-SYARAT PEMBUATAN DATA FLOW DIAGRAM

Syarat pembuatan DFD ini akan menolong profesional sistem untuk menghindari pembentukan DFD yang salah atau DFD yang tidak lengkap atau tidak konsisten secara logika. Beberapa syarat pembuatan DFD dapat menolong profesional sistem untuk membentuk DFD yang benar, menyenangkan untuk dilihat dan mudah dibaca oleh pemakai.

Syarat-syarat pembuatan DFD ini adalah :

1. Pemberian nama untuk tiap komponen DFD
2. Pemberian nomor pada komponen proses
3. Penggambaran DFD sesering mungkin agar enak dilihat
4. Penghindaran penggambaran DFD yang rumit
5. Pemastian DFD yang dibentuk itu konsisten secara logika

5.1. Pemberian Nama untuk Tiap komponen DFD

Seperti yang telah dijelaskan sebelumnya, komponen terminator mewakili lingkungan luar dari sistem, tetapi mempunyai pengaruh terhadap sistem yang sedang dikembangkan ini. Maka agar pemakai mengetahui dengan lingkungan mana saja sistem mereka berhubungan, komponen terminator ini harus diberi nama sesuai dengan lingkungan luar yang mempengaruhi sistem ini. Biasanya komponen terminator diberi nama dengan kata benda.

Selanjutnya adalah komponen proses. Komponen proses ini mewakili fungsi sistem yang akan dilaksanakan atau menunjukkan bagaimana fungsi sistem dilaksanakan oleh seseorang, sekelompok orang atau mesin. Maka sangatlah jelas bahwa komponen ini perlu diberi nama yang tepat, agar siapa yang membaca DFD khususnya pemakai akan merasa yakin bahwa DFD yang dibentuk ini adalah model yang akurat.

Pemberian nama pada komponen proses lebih baik menunjukkan aturan-aturan yang akan dilaksanakan oleh seseorang dibandingkan dengan memberikan nama atau identitas orang yang akan melaksanakannya. Ada dua alasan mengapa bukan nama atau identitas orang (yang melaksanakan fungsi sistem) yang digunakan sebagai nama proses, yaitu :

1. Orang tersebut mungkin diganti oleh orang lain saat mendatang, sehingga bila tiap kali ada pergantian orang yang melaksanakan fungsi tersebut, maka sistem yang dibentuk harus diubah lagi.

2. Orang tersebut mungkin tidak melaksanakan satu fungsi sistem saja, melainkan beberapa fungsi sistem yang berbeda. Daripada menggambarkan beberapa proses dengan nama yang sama tetapi artinya berbeda, lebih baik tunjukkan dengan tugas/fungsi sistem yang sebenarnya akan dilaksanakan.

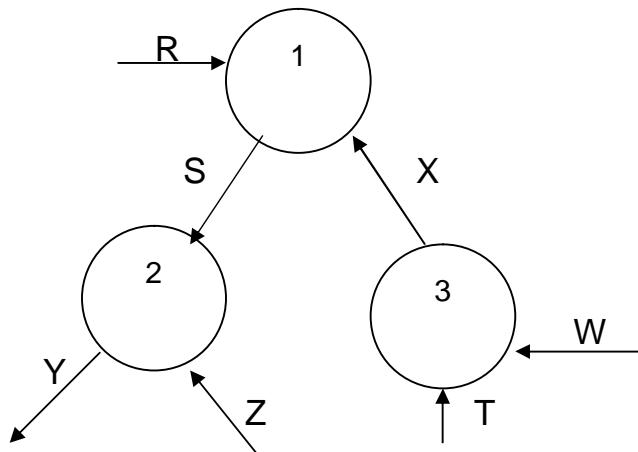
Karena nama untuk komponen proses lebih baik menunjukkan tugas/fungsi sistem yang akan dilaksanakan, maka lebih baik pemberian nama ini menggunakan kata kerja transitif.

Pemberian nama untuk komponen data store menggunakan kata benda, karena data store menunjukkan data apa yang disimpan untuk kebutuhan sistem dalam melaksanakan tugasnya. Jika sistem sewaktu-waktu membutuhkan data tersebut untuk melaksanakan tugasnya, maka data tersebut tetap ada, karena sistem menyimpannya.

Begitu pula untuk komponen alur data, namanya lebih baik diberikan dengan menggunakan kata benda. Karena alur data ini menunjukkan data dan informasi yang dibutuhkan dan yang dikeluarkan oleh sistem dalam pelaksanaan tugasnya.

5.2. Pemberian Nomor pada Komponen Proses

Biasanya profesional sistem memberikan nomor dengan bilangan terurut pada komponen proses sebagai referensi. Tidak jadi masalah bagaimana nomor-nomor proses ini diberikan. Nomor proses dapat diberikan dari kiri ke kanan, atau dari atas ke bawah, atau dapat pula dilakukan dengan pola-pola tertentu selama pemberian nomor ini tetap konsisten pada nomor yang dipergunakan.



Gambar 8. Contoh Pemberian nomor pada proses

Nomor-nomor proses yang diberikan terhadap komponen proses ini tidak dimaksudkan bahwa proses tersebut dilaksanakan secara berurutan. Pemberian nomor ini dimaksudkan agar pembacaan suatu proses dalam suatu diskusi akan lebih mudah dengan hanya menyebutkan prosesnya saja jika dibandingkan dengan menyebutkan nama prosesnya, khususnya jika nama prosesnya panjang dan sulit.

Maksud pemberian nomor pada proses yang lebih penting lagi adalah untuk menunjukkan referensi terhadap skema penomoran secara hirarki pada levelisasi DFD. Dengan kata lain, nomor proses ini merupakan dasar pemberian nomor pada levelisasi DFD (*lihat gambar 11*).

5.3. Penggambaran DFD sesering mungkin

Penggambaran DFD dapat dilakukan berkali-kali sampai secara teknik DFD itu benar, dapat diterima oleh pemakai, dan sudah cukup rapih sehingga profesional sistem tidak merasa malu untuk menunjukkan DFD itu kepada atasannya dan pemakai.

Dengan kata lain, penggambaran DFD ini dilakukan sampai terbentuk DFD yang enak dilihat, dan mudah dibaca oleh pemakai dan profesional sistem lainnya. Keindahan penggambaran DFD tergantung pada standar-standar yang diminta oleh organisasi tempat profesional sistem itu bekerja dan perangkat lunak yang dipakai oleh profesional sistem dalam membuat DFD.

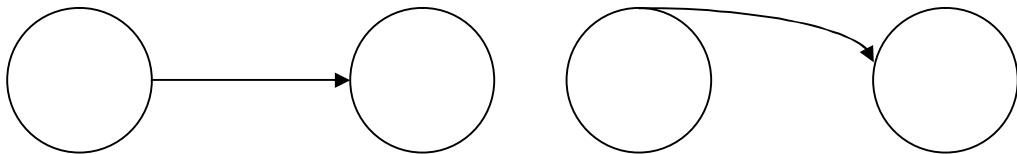
Penggambaran yang enak untuk dilihat dapat dilakukan dengan memperhatikan hal-hal berikut ini :

Ø ***Ukuran dan bentuk proses.***

Beberapa pemakai kadang-kadang merasa bingung bila ukuran proses satu berbeda dengan proses yang lain. Mereka akan mengira bahwa proses dengan ukuran yang lebih besar akan diduga lebih penting dari proses yang lebih kecil. Hal ini sebenarnya hanya karena nama proses itu lebih panjang dibandingkan dengan proses yang lain. Jadi, sebaiknya proses yang digambarkan memiliki ukuran dan bentuk yang sama.

Ø ***Alur data melingkar dan alur data lurus.***

Alur data dapat digambarkan dengan melingkar atau hanya garis lurus. Mana yang lebih enak dipandang tergantung siapa yang akan melihat DFD tersebut.



- (a). Alur data dengan garis lurus (b). Alur data dengan melingkar
Gambar 9

Ø ***DFD dengan gambar tangan dan gambar menggunakan mesin.***

DFD dapat digambarkan secara manual atau dengan menggunakan bantuan mesin, tergantung pilihan pemakai atau profesional sistem.

5.4. Penghindaran Penggambaran DFD yang rumit

Tujuan DFD adalah untuk membuat model fungsi yang harus dilaksanakan oleh suatu sistem dan interaksi antar fungsi. Tujuan lainnya adalah agar model yang dibuat itu mudah dibaca dan dimengerti tidak hanya oleh profesional sistem yang membuat DFD, tetapi juga oleh pemakai yang berpengalaman dengan subyek yang terjadi. Hal ini berarti DFD harus mudah dimengerti, dibaca, dan menyenangkan untuk dilihat.

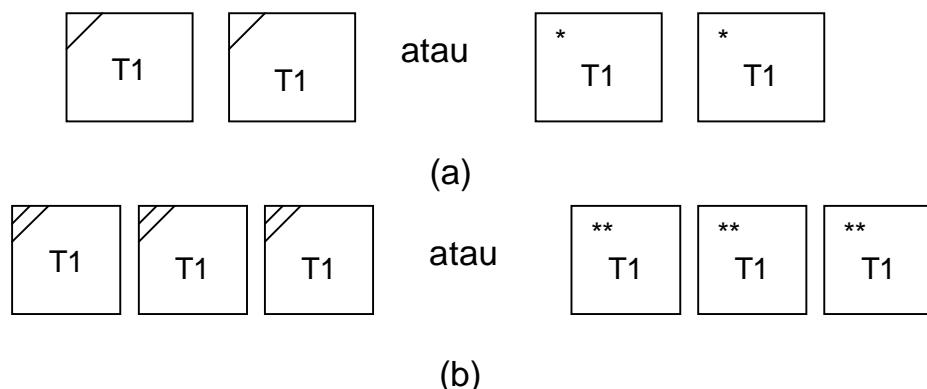
Pada banyak masalah, DFD yang dibuat tidak memiliki terlalu banyak proses (maksimal enam proses) dengan data store, alur data, dan terminator yang berkaitan dengan proses tersebut dalam satu diagram.

Bila terlalu banyak proses, terminator, data store, dan alur data digambarkan dalam satu DFD, maka ada kemungkinan terjadi banyak persilangan alur data dalam DFD tersebut. Persilangan alur data ini menyebabkan pemakai akan sulit membaca dan mengerti DFD yang terbentu. Jadi semakin sedikit adanya persilangan data pada DFD, maka makin baik DFD yang dibentuk oleh profesional sistem.

Persilangan alur data ini dapat dihindari dengan menggambarkan DFD secara bertingkat-tingkat (levelisasi DFD), atau dengan menggunakan pemakaian duplikat terhadap komponen DFD.

Komponen DFD yang dapat menggunakan duplikat hanya **komponen store** dan **terminator**. Pemberian duplikat ini juga tidak dapat diberikan sesuka profesional sistem yang membuat DFD, tetapi makin sedikit pemakaian duplikat, makin baik DFD yang terbentuk.

Pemberian duplikat terhadap data store dilakukan dengan memberikan simbol garis lurus (ξ) atau asterik (*), sedangkan untuk terminator menggunakan simbol garis miring (/) atau asterik (*). Banyaknya pemberian simbol duplikat pada duplikat yang digunakan tergantung banyaknya duplikat yang digunakan.



Gambar 10. Contoh pemakaian simbol duplikat pada komponen terminator

- (a) Satu duplikat yang digunakan
- (b) Dua duplikat yang digunakan

5.5. Penggambaran DFD yang Konsisten

Penggambaran DFD harus konsisten terhadap kelompok DFD lainnya. Profesional sistem menggambarkan DFD berdasarkan tingkatan DFD dengan tujuan agar DFD yang dibuatnya itu mudah dibaca dan dimengerti oleh pemakai sistem. Hal ini sesuai dengan salah satu tujuan atau syarat membuat DFD.

6. PENGGAMBARAN DFD

Tidak ada aturan baku untuk menggambarkan DFD. Tapi dari berbagai referensi yang ada, secara garis besar langkah untuk membuat DFD adalah :

1. Identifikasi terlebih dahulu semua entitas luar yang terlibat di sistem.

2. Identifikasi semua input dan output yang terlibat dengan entitas luar.

3. Buat Diagram Konteks (*diagram context*)

Diagram ini adalah diagram level tertinggi dari DFD yang menggambarkan hubungan sistem dengan lingkungan luarnya.

Caranya :

- q Tentukan nama sistemnya.
- q Tentukan batasan sistemnya.
- q Tentukan terminator apa saja yang ada dalam sistem.
- q Tentukan apa yang diterima/diberikan terminator dari/ke sistem.
- q Gambarkan diagram konteks.

4. Buat Diagram Level Zero

Diagram ini adalah dekomposisi dari diagram konteks.

Caranya :

- q Tentukan proses utama yang ada pada sistem.
- q Tentukan apa yang diberikan/diterima masing-masing proses ke/dari sistem sambil memperhatikan konsep keseimbangan (alur data yang keluar/masuk dari suatu level harus sama dengan alur data yang masuk/keluar pada level berikutnya).
- q Apabila diperlukan, munculkan data store (master) sebagai sumber maupun tujuan alur data.
- q Gambarkan diagram level zero.
 - Hindari perpotongan arus data
 - Beri nomor pada proses utama (nomor tidak menunjukkan urutan proses).

5. Buat Diagram Level Satu

Diagram ini merupakan dekomposisi dari diagram level zero.

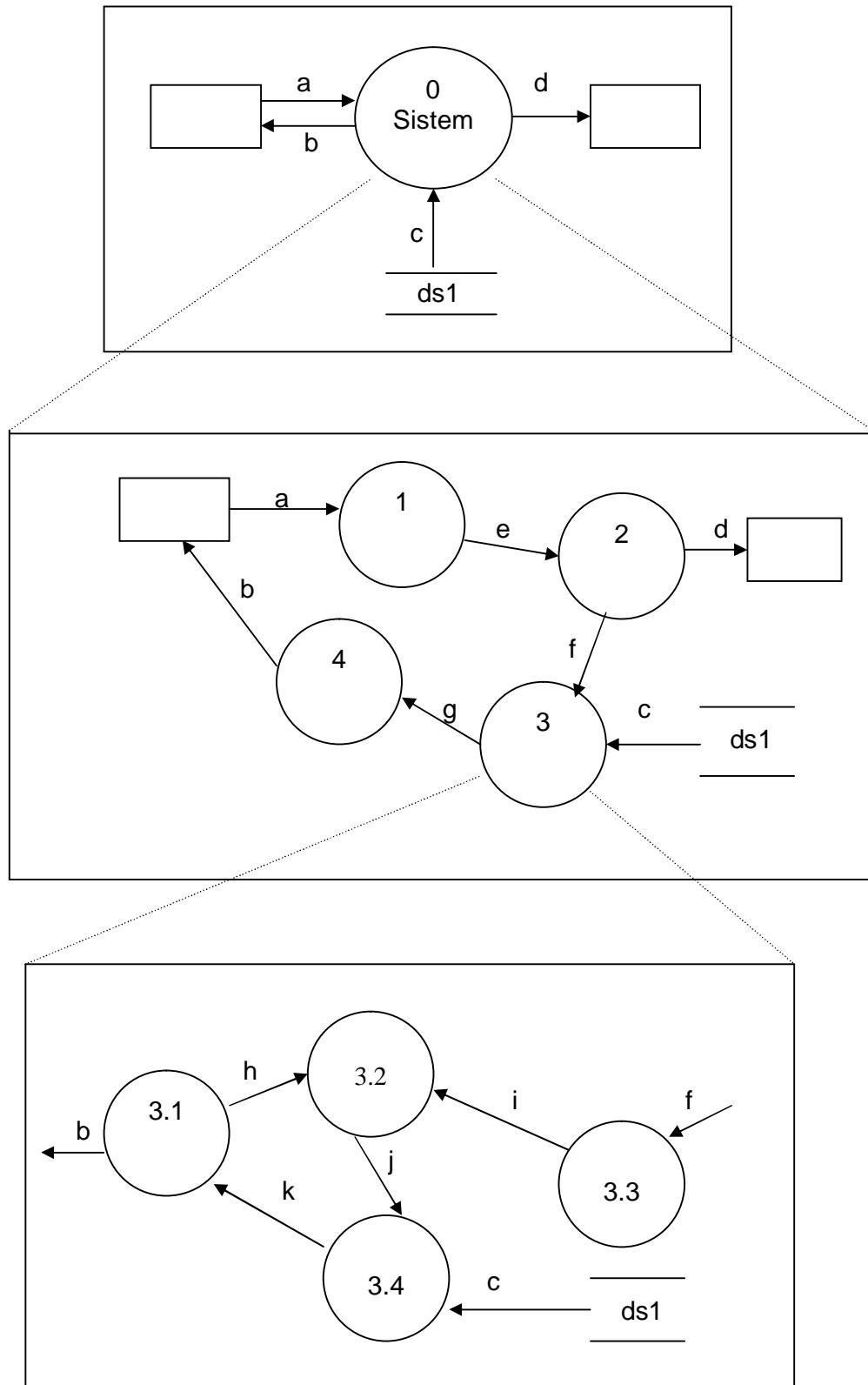
Caranya :

- q Tentukan proses yang lebih kecil (sub-proses) dari proses utama yang ada di level zero.
- q Tentukan apa yang diberikan/diterima masing-masing sub-proses ke/dari sistem dan perhatikan konsep keseimbangan.
- q Apabila diperlukan, munculkan data store (transaksi) sebagai sumber maupun tujuan alur data.
- q Gambarkan DFD level Satu
 - Hindari perpotongan arus data.
 - Beri nomor pada masing-masing sub-proses yang menunjukkan dekomposisi dari proses sebelumnya.

Contoh : 1.1, 1.2, 2.1

6. DFD Level Dua, Tiga, ...

Diagram ini merupakan dekomposisi dari level sebelumnya. Proses dekomposisi dilakukan sampai dengan proses siap dituangkan ke dalam program. Aturan yang digunakan sama dengan level satu.



Gambar 11. Levelisasi DFD

Metode Berorientasi Objek

Panji Wisnu Wirawan

Review

Jika anda menggunakan metode terstruktur /klasik/konvensional (baik analisis maupun desain), apa yang anda modelkan ke dalam diagram ?

(petunjuk : diagram apa yang anda buat ?)

Agenda

- Objek
- Metode Berorientasi Objek
- Metode Terstruktur vs Berorientasi Objek
- Elemen Model Objek
- Analisis Berorientasi Objek
- Desain Berorientasi Objek

Objek

*An **object** is an entity that has state, behavior, and identity. The structure and behavior of similar objects are defined in their common **class**. The terms instance and object are interchangeable.*
(Booch, dkk., 2007)

Objek

Objek	Class
  	MOBIL

State

The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.

(Booch, 2007)



- no rangka : 123 (static)
- no mesin : 456 (static)
- warna : hitam (static)
- merk : koyoto (static)
- kecepatan : 40km/h (dynamic)
- rpm : 2000 rpm (dynamic)

Behavior

Behavior is how an object acts and reacts, in terms of its state changes and message passing.

(Booch, 2007)



- berjalan
- berhenti
- bertambah kecepatan
- berkurang kecepatan

Identity

Property of an object which distinguishes it from all other object

(Khosafian & Copeland dikutip dalam Booch, 2007)



-no rangka : 123
-no mesin : 456



-no rangka : 312
-no mesin : 645

Objek

Objek	Class
	MAHASISWA

Metode Berorientasi Objek

- Memandang masalah yang akan diselesaikan sebagai objek-objek.
- Objek-objek saling berhubungan dan berinteraksi membentuk tugas-tugas tertentu (*use case*).

Metode Terstruktur vs Berorientasi Objek

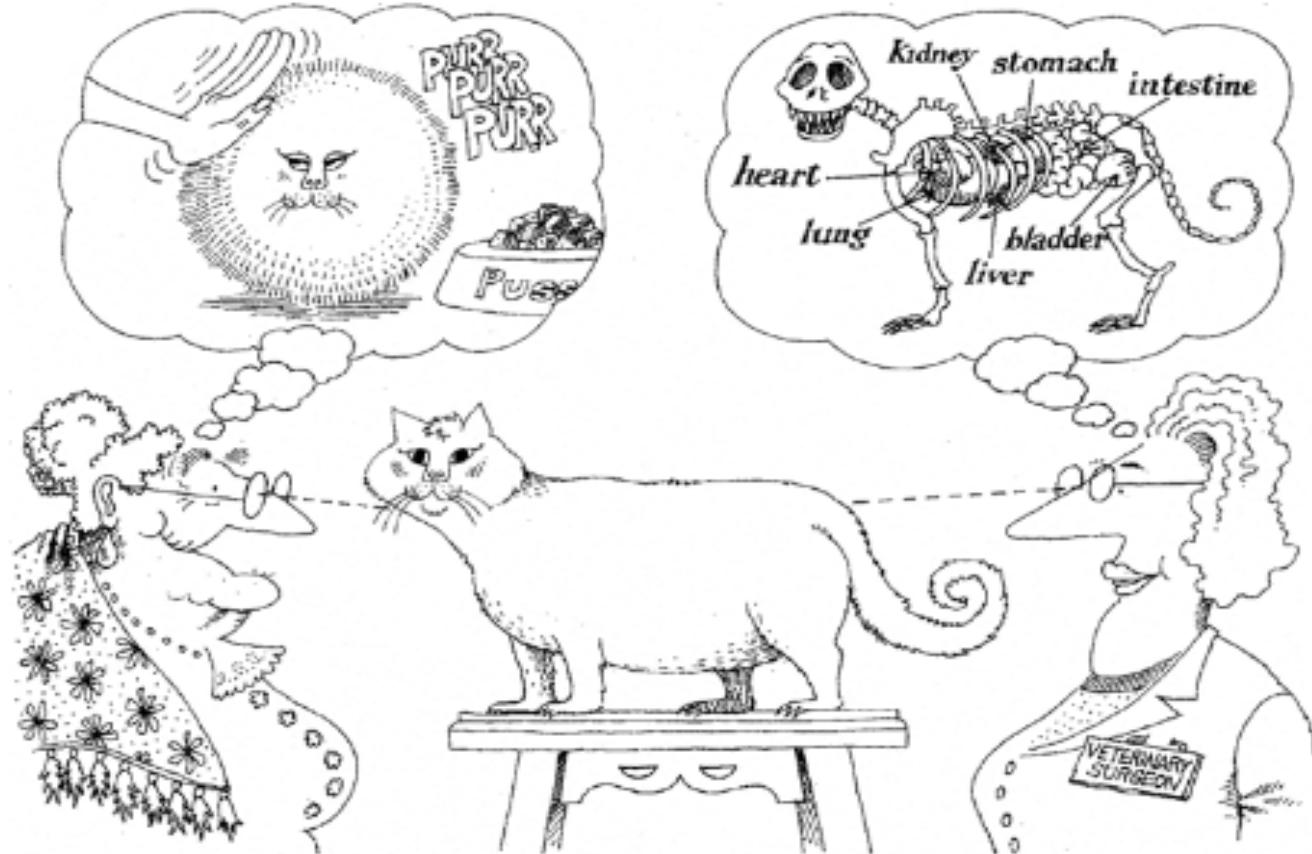
Pembeda	Metode Terstruktur	Metode Berorientasi Objek
Fokus	Proses yang berinteraksi	Objek yang berhubungan dan berinteraksi
Pemodelan	DFD, ERD, STD	UML
Reusing	sulit, sebab me-reuse sebuah proses cenderung melibatkan proses yang lain. (dependensi tinggi)	Mudah, sebab proyek diorganisasi dalam objek-objek dimana abstraksi lebih diutamakan. OO mengarah pada <i>high cohesion</i> dan <i>low coupling</i> .
Metodologi	SDLC (waterfall)	Iteratif & Inkremental
Dekomposisi	Algoritmik	<i>Object oriented</i>

Elemen Model Objek

(Booch,dkk. 2007)

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Abstraction



An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

Abstraction

- Fokus pada tingkah laku objek yang dapat diamati.
- Digunakan untuk membantu implementasi.

Abstraction

Contoh :

Dalam sebuah *greenhouse* yang relatif besar, diperlukan sistem untuk memonitor suhu *greenhouse*. Secara periodik, sistem akan mengukur suhu *greenhouse* dan melaporkan kepada pemilik *greenhouse*. Hal ini diperlukan karena suhu pada *greenhouse* harus optimal untuk pertumbuhan tanaman di dalamnya.



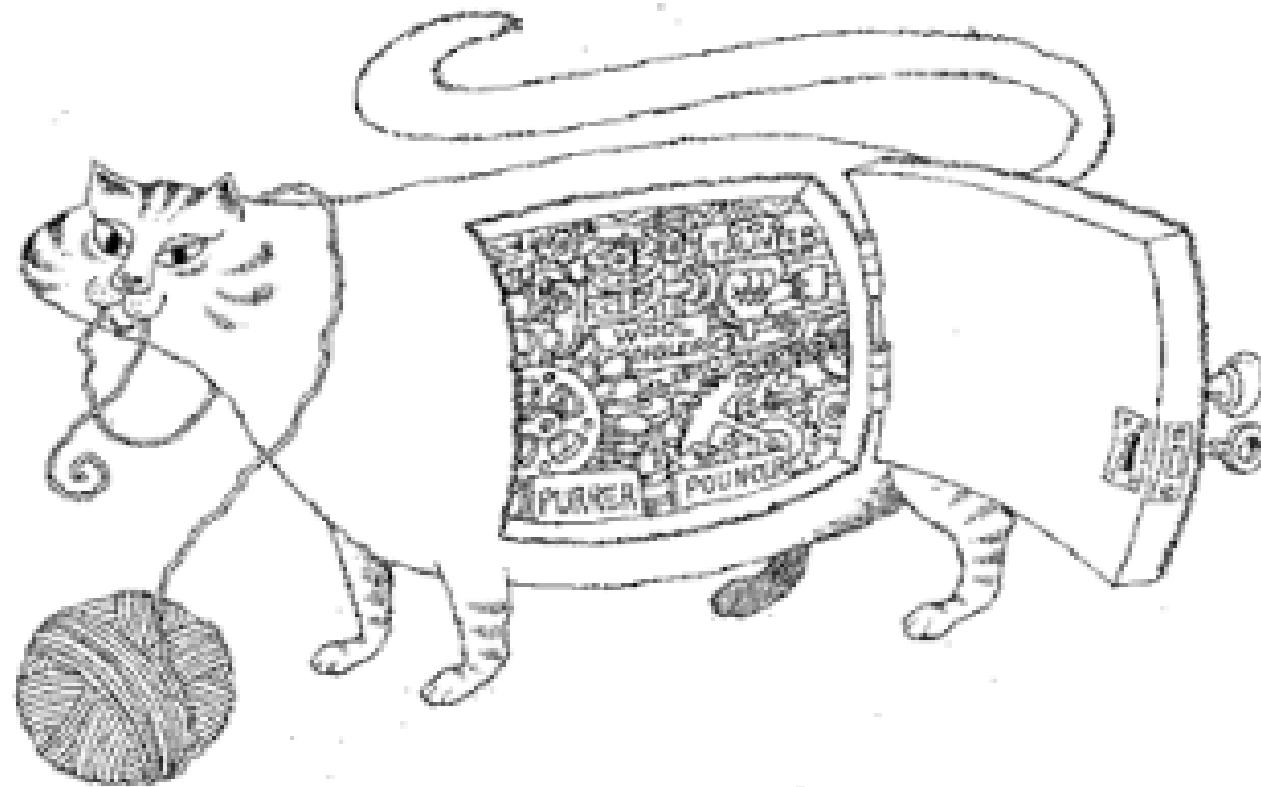
Abstraction

- *Abstraction* :
 - apa yang digunakan untuk mengukur suhu ?
 - karakteristik yang dimiliki pengukur suhu dalam *greenhouse* ?
 - apa yang dapat dilakukan oleh pengukur suhu ?

Abstraction

- Abstraction : sensor suhu
- Karakteristik : memiliki data temperatur
- Yang dapat dilakukan : mengukur temperatur, melakukan kalibrasi temperatur

Encapsulation



Encapsulation hides the details of the implementation of an object.

Encapsulation

- Berkaitan erat dengan abstraction.
- Fokus : implementasi yang dapat menimbulkan tingkah laku tertentu.
- Menyembunyikan struktur dan implementasi setiap tingkah laku.

Encapsulation

Contoh :

Untuk mendapatkan suhu *greenhouse*, cukup dikirimkan pesan ke sensor bahwa “**saya ingin mengetahui suhu *greenhouse***” dan sensor akan mengembalikan suhu saat itu juga. Detail bagaimana sensor mengukur suhu tidak perlu diketahui pengguna.

Modularity



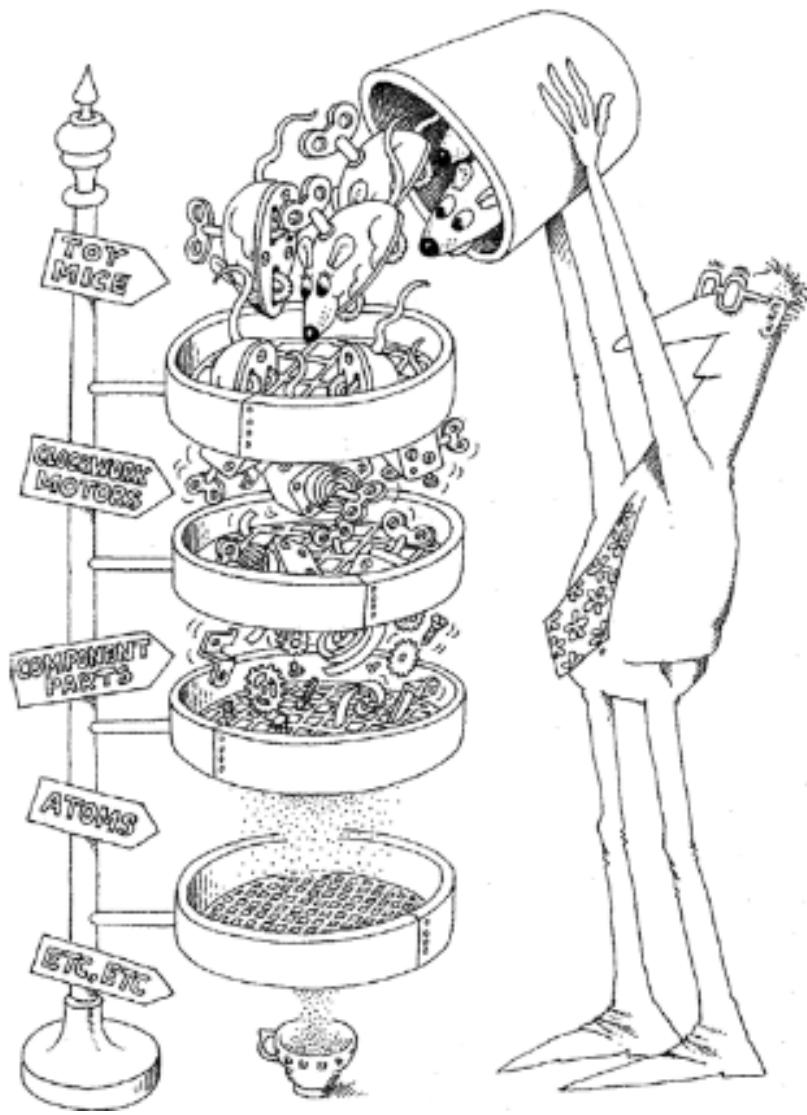
Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Modularity

- Modularity fokus pada layanan yang akan diberikan oleh sebuah modul perangkat lunak.
 - Modul dapat berupa sekelompok class / kelas yang membentuk fungsi tertentu / memiliki karakteristik yang sejenis.

Modularity

Contoh : dalam perangkat lunak untuk pengukuran suhu untuk *Greenhouse*, diperlukan modul untuk pengukuran suhu dan modul untuk display suhu hasil pengukuran.



“ranking or ordering of abstraction”

Hierarchy

Hierarchy

- Contoh hierarchy adalah *inheritance* / pewarisan.
- Pewarisan merupakan hubungan “is-a” antar kelas.
- Pewarisan membentuk hierarchy generalisasi / spesialisasi.
- Contoh :
 - sensor suhu “is-a” sensor
 - mahasiswa s1 “is-a” mahasiswa

Hierarchy

- Dalam pewarisan, objek/kelas dapat mewarisi atribut dan metode dari objek/kelas yang lain.

Elemen Model Objek

- Pengetahuan mengenai elemen-elemen dalam model objek (model berorientasi objek) diperlukan dalam analisis dan desain perangkat lunak berorientasi objek.

Analisis Berorientasi Objek

- Fokus : membuat model objek dari domain aplikasi.
- Menggambarkan apa yang dapat dilakukan oleh sistem.
- Belum memiliki kecenderungan ke arah implementasi.

Desain Berorientasi Objek

- Fokus : membuat model objek dari perangkat lunak yang siap untuk diimplementasikan.
- Menggambarkan bagaimana sistem melakukan sesuatu.
- Lebih ke arah teknis implementasi.

Kesimpulan

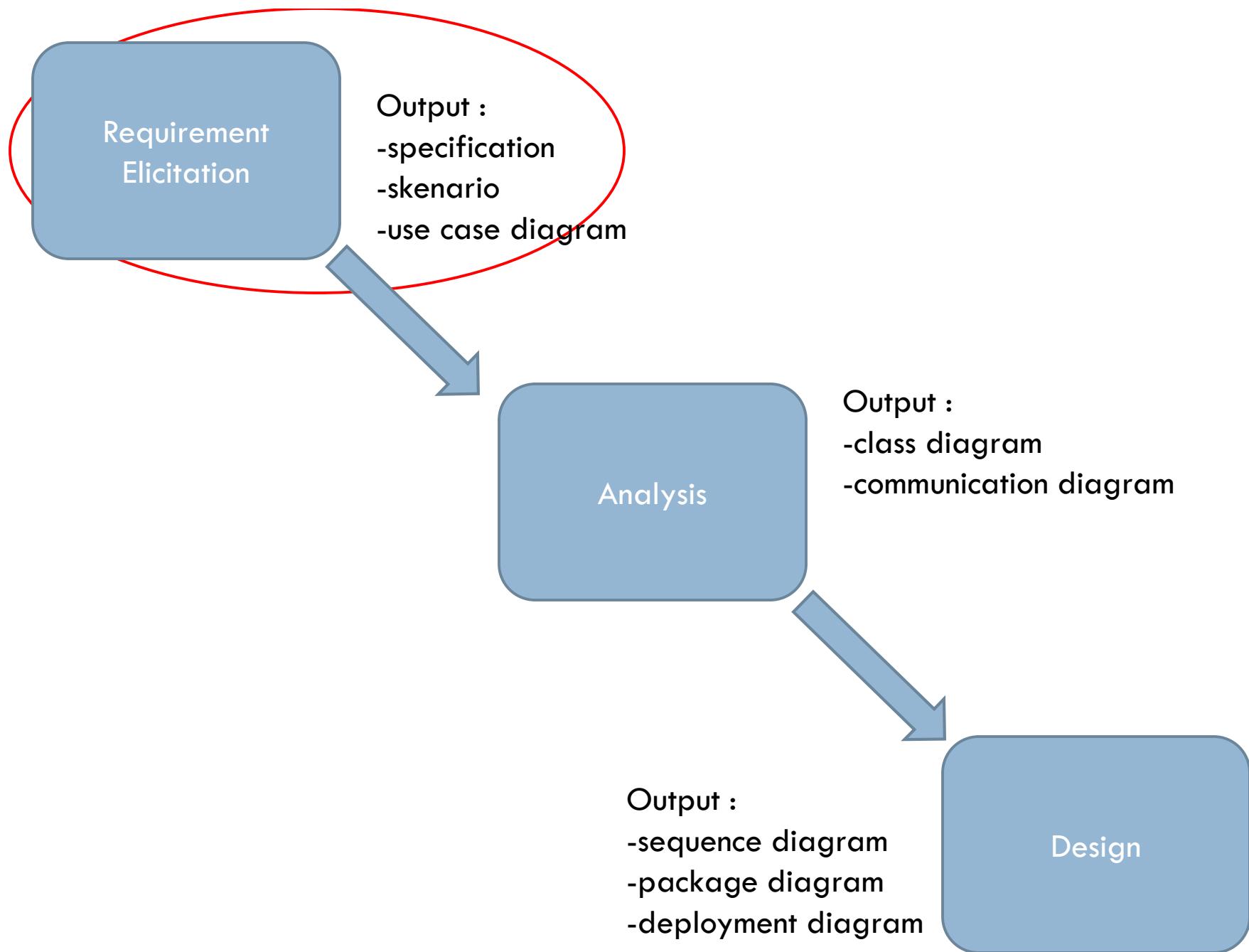
- Metode berorientasi objek berfokus pada objek-objek yang saling berinteraksi beserta hubungannya.
- Elemen yang harus diperhatikan dalam model berorientasi objek adalah *abstraction*, *modularity*, *encapsulation*, dan *hierarchy*.
- Pengetahuan mengenai elemen model objek diperlukan untuk analisis dan desain perangkat lunak berorientasi objek.

Referensi

- Booch, G., et.al, 2007 , *Object Oriented Analysis And Design With Application*, Addison-Wesley.
- Sommerville, I, 2011, *Software Engineering*, Pearson Education, 8th edition.

OO REQUIREMENT ELICITATION

Panji Wisnu Wirawan



Agenda

- 
- Requirement Elicitation
 - Aktivitas
 - Use Case Diagram

TextBook



Brugge, B & Dutoit, A.H ; 2004 ; *Object Oriented Software Engineering Using UML, Patterns And Java* ; Prentice Hall

Requirement Elicitation

- Sering disebut sebagai *requirement specification*.
- Fokus : apa tujuan membuat sistem ?
- Yang terlibat : client, developer, end-user.
- Hasil : *requirement specification* (berupa bahasa alami / natural language) diwujudkan dalam *functional* dan *non-functional requirement*.

Requirement Elicitation

- **Functional Requirement** mendefinisikan interaksi antara sistem dengan ‘lingkungannya’ lepas dari bagaimana sistem tsb diimplementasikan
- **Non-Functional Requirement** mendefinisikan aspek-aspek yang mendukung kerja sistem di luar functional requirement.

Requirement Elicitation

- Beberapa kategori untuk non functional requirement:
 - ❑ Usability
 - ❑ Reliability
 - ❑ Performance
 - ❑ Supportability

Requirement Elicitation

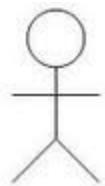
- Contoh *functional requirement* :
eCatalog adalah sebuah online catalog yang memungkinkan seorang pengguna internet untuk mencari buku-buku di seluruh perpustakaan di Indonesia.
- Contoh *non functional requirement* :
Dalam sekali pencarian, eCatalog memerlukan waktu maksimal 10 detik. (performance)
eCatalog harus dapat melayani 10 ribu pengguna dalam satu detik. (reliability)

Aktivitas

- Aktivitas dalam Requirement Elicitation :
 - Identifikasi aktor
 - Identifikasi skenario
 - Identifikasi use case
 - Refine use case
 - Identifikasi hubungan antar use case
 - Identifikasi kebutuhan non-fungsional

Identifikasi Aktor

- Aktor : representasi entitas luar yang berinteraksi dengan sistem (sekelompok pengguna / sistem luar yang akan ikut berinteraksi).
- Aktor merupakan abstraksi peranan, tidak menunjuk ke orang tertentu.
- Simbol aktor dalam UML use case diagram : *stick figure*.



Identifikasi Aktor

- Pertanyaan pembantu untuk identifikasi aktor :
 - Siapa yang akan didukung kerjanya oleh sistem ?
 - Siapa yang akan menjalankan kerja utama sistem ?
 - Siapa yang akan melakukan tugas sekunder sistem seperti administrasi dan pemeliharaan ?
 - Dengan sistem perangkat lunak / perangkat keras yang mana sistem akan berinteraksi ?

Identifikasi Skenario

- Skenario : deskripsi (informal & naratif) dari sebuah kemampuan yang dapat dilakukan oleh sistem dari sudut pandang seorang aktor.
- Pertanyaan-pertanyaan untuk identifikasi skenario :
 - ▣ Kerja seperti apa yang diinginkan aktor terhadap sistem ?
 - ▣ Informasi apa yang akan diperoleh aktor ?
 - ▣ Perubahan luar seperti apa yang diinginkan informasinya oleh aktor ?

Identifikasi Skenario

Aktor	
Nama Skenario	
Flow Of Events / Urutan Kejadian	1. 2. 3. 4.
Alternate Scenario	1. 2.
Exceptional Scenario	1. 2.

Identifikasi Skenario (contoh)

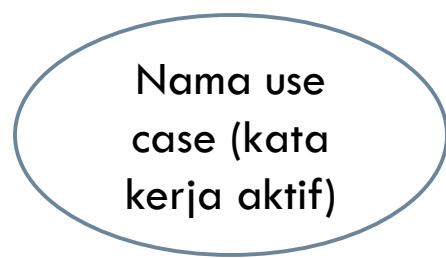
Aktor	Pengunjung perpustakaan
Nama Skenario	Mencari buku
Flow Of Events / Urutan Kejadian	1. Pengunjung memasukkan kata kunci pada katalog elektronik. 2. Sistem mencari buku berdasarkan kata kunci yang dimasukkan. 3. Sistem menemukan hasil pencarian 4. Sistem menampilkan buku yang dimaksud.
Alternate Scenario	-
Exceptional Scenario	3.1. Sistem tidak menemukan buku yang dimaksud 3.2. Sistem menampilkan pesan bahwa buku tidak ditemukan.

Identifikasi Use Case

- Use Case menggambarkan semua skenario yang mungkin.
- Skenario merupakan perwujudan nyata (*instance*) dari use case.
- Beberapa skenario dapat menjadi sebuah use case.

Identifikasi Use Case

- Simbol Use Case dalam UML use case diagram :



Refine Use Case

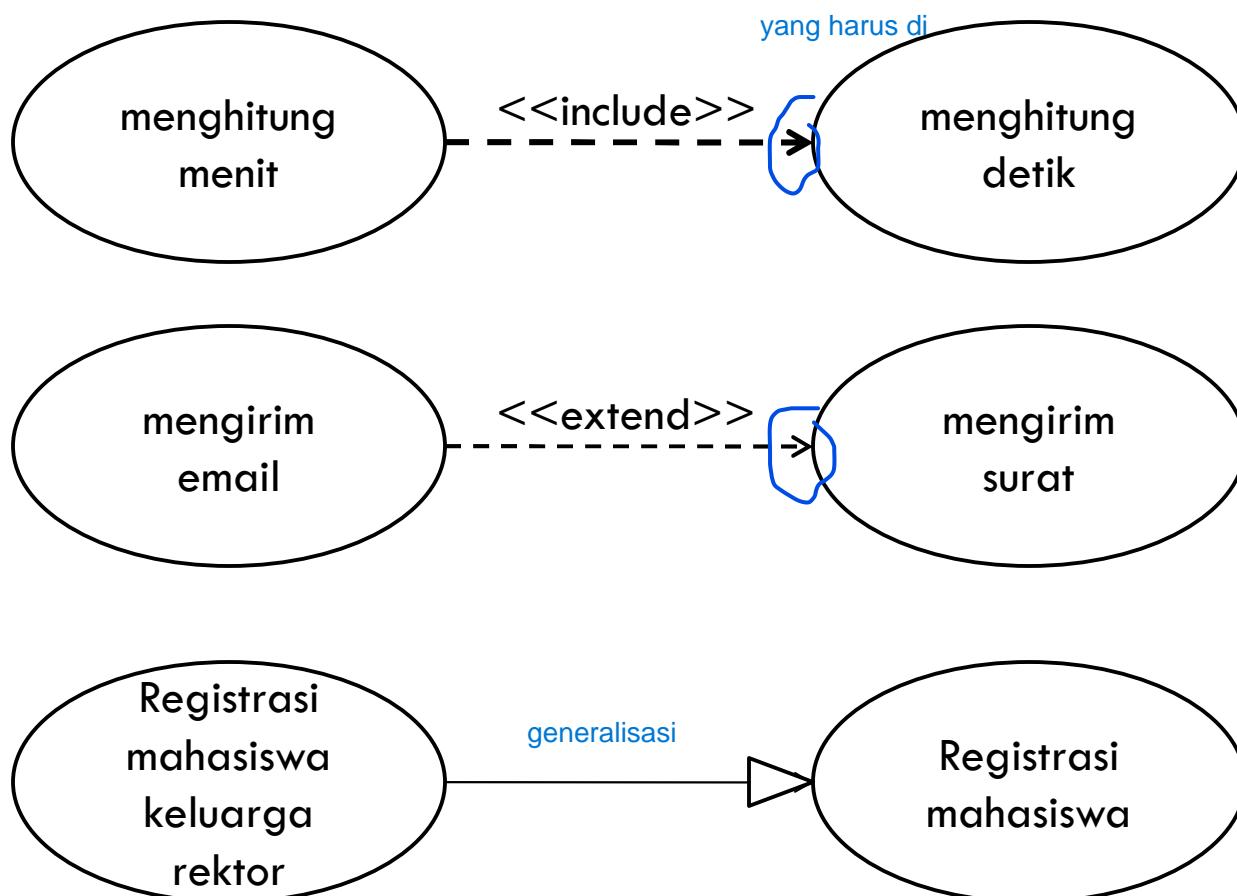
- Proses ‘refine’ dilakukan untuk mengevaluasi use case yang telah dibuat sebelumnya.
- Evaluasi bisa dilakukan dengan mengkomunikasikan use case yang dibentuk dengan customer.
- Hasil evaluasi tersebut bisa tetap, bisa berubah maupun bertambah.
- Refining → Iteration.

Identifikasi Hubungan Antar Use case

- Hubungan antar use case :

- Include
- Extend
- Generalisasi

Identifikasi Hubungan Antar Use Case



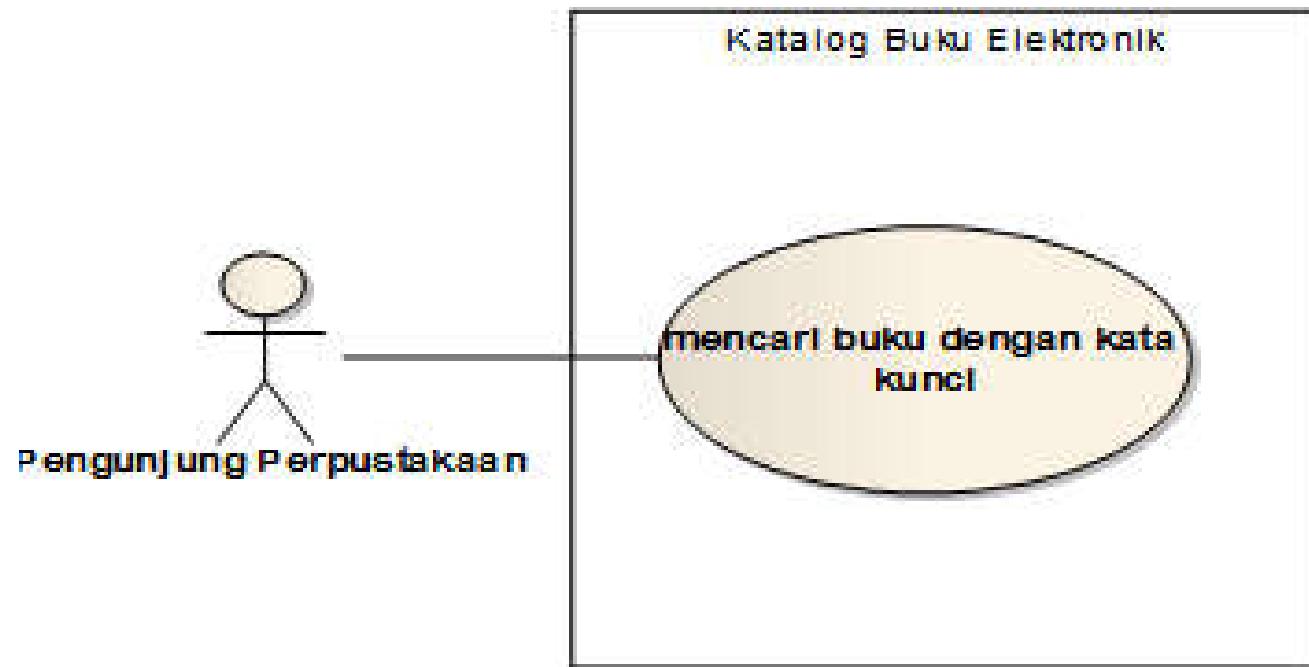
Identifikasi Hubungan Antar Use Case

PERHATIAN :

Ketiga hubungan antar use case tersebut tidak harus selalu ada dalam diagram use case.

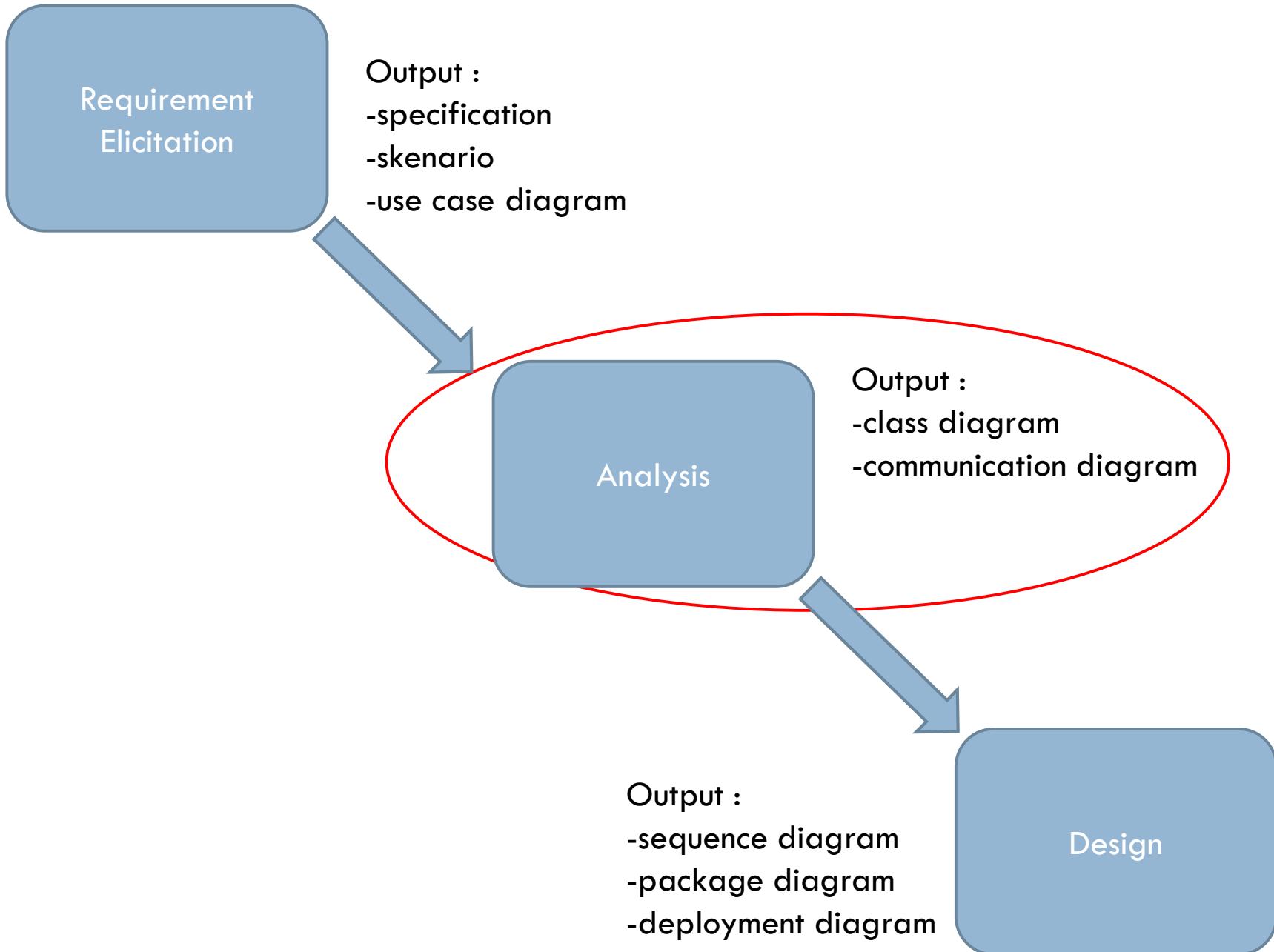
Use Case Diagram

- Actor + use case + use case relationship + System boundary.



OBJECT ORIENTED ANALYSIS

Panji Wisnu Wirawan



Agenda



- Object Oriented Analysis (OOA)
- Aktivitas OOA

Object Oriented Analysis (OOA)

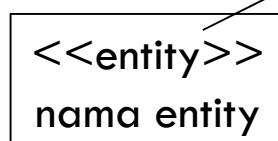
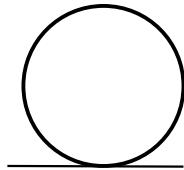
- 
- Fokus : membentuk model analisis.
 - Model analisis memodelkan sistem yg. akan dibuat dari sudut pandang pengguna.
 - Model analisis :
 - Model fungsional
 - Model objek
 - Model dinamik

Aktivitas OOA

- 
- Identifikasi objek entity, boundary, dan control.
 - Mapping use case dengan sequence/communication diagram.
 - Memodelkan hubungan antar objek dengan CRC.
 - Identifikasi hubungan antar objek.
 - Identifikasi atribut kelas
 - Membentuk class diagram.

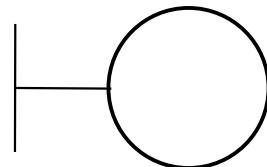
Entity

- Objek Entity : objek-objek yang digunakan sistem.
- Identifikasi entity :
 - Identifikasi kata benda dalam use case
 - Real world entities & activities
 - Data sources
- Simbol dalam UML :



Boundary

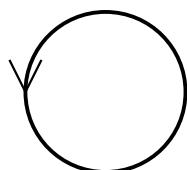
- Objek Boundary : objek yang mewakili interaksi aktor dengan sistem.
- Identifikasi objek boundary :
 - Identifikasi user interface untuk inisiasi use case
 - Identifikasi form untuk memasukkan data ke sistem
 - Identifikasi notifikasi / pesan dari sistem
- Simbol dalam UML :



```
<<boundary>>  
nama boundary
```

Control

- Objek Control : objek yang merealisasikan use case.
- Identifikasi objek control
 - Identifikasi 1 control object per use case
 - Identifikasi 1 control object per actor dalam use case.
- Simbol dalam UML :



<<control>>
nama control

Contoh

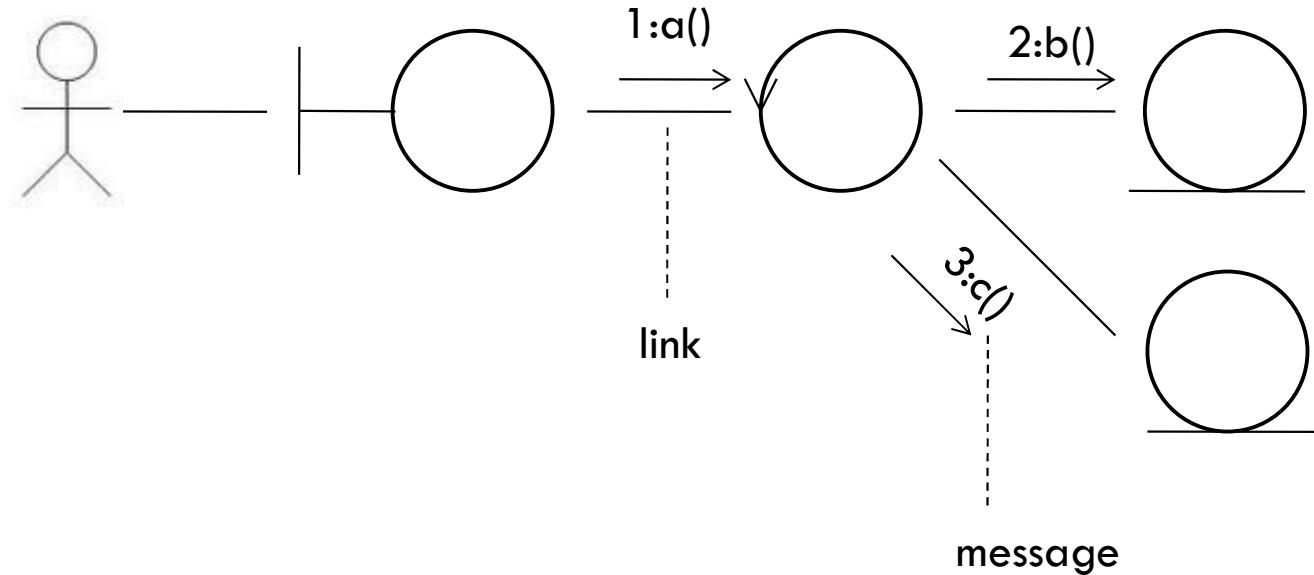
Dalam eCatalog :

- Entity : Buku
- Boundary : Form Pencarian Buku,Display Hasil Pencarian
- Control : Pencarian Buku

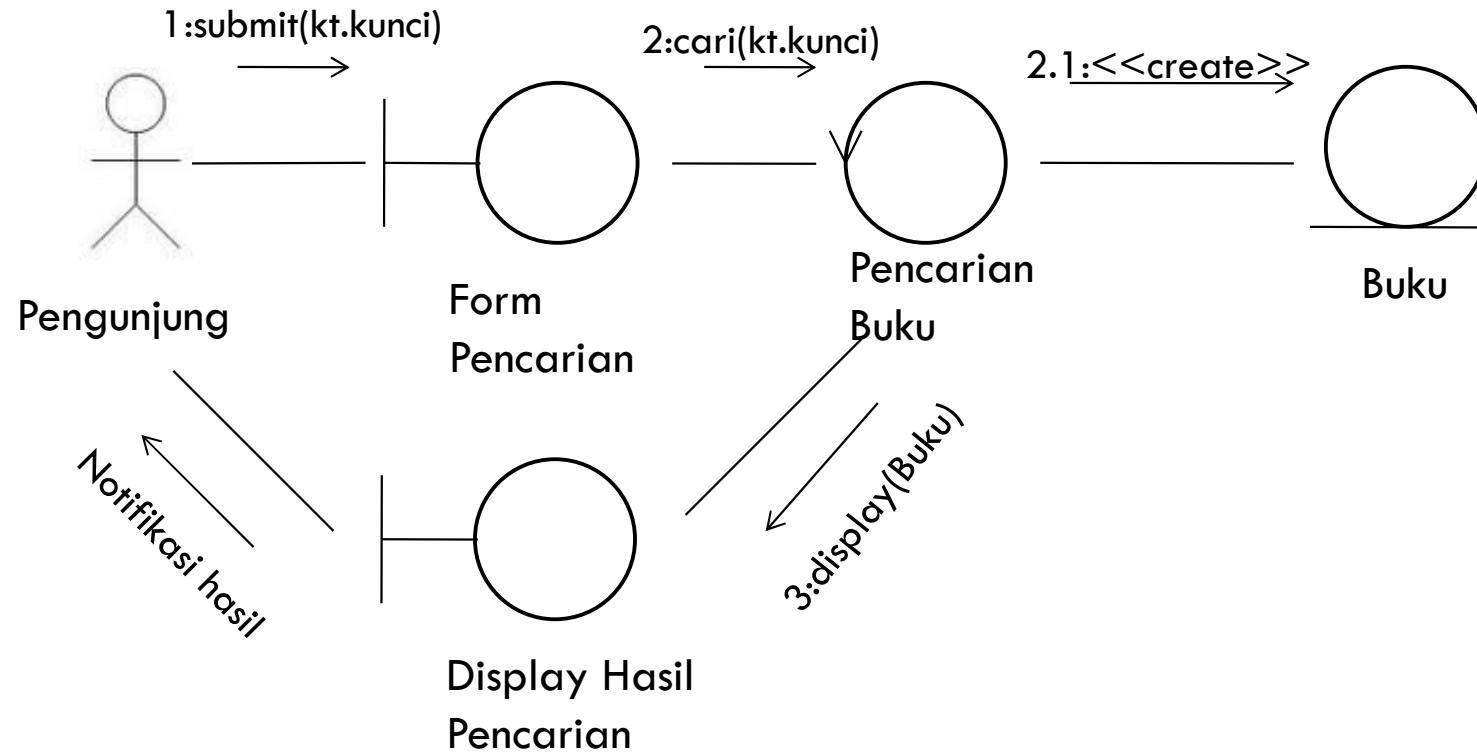
Communication Diagram

- Communication diagram fokus pada link objek yang terlibat dan message yang terlibat di dalamnya.
- Communication diagram termasuk model dinamis.
- Communication diagram dibuat berdasarkan use case.
- 1 use case = 1 communication diagram.

Communication Diagram



Communication Diagram

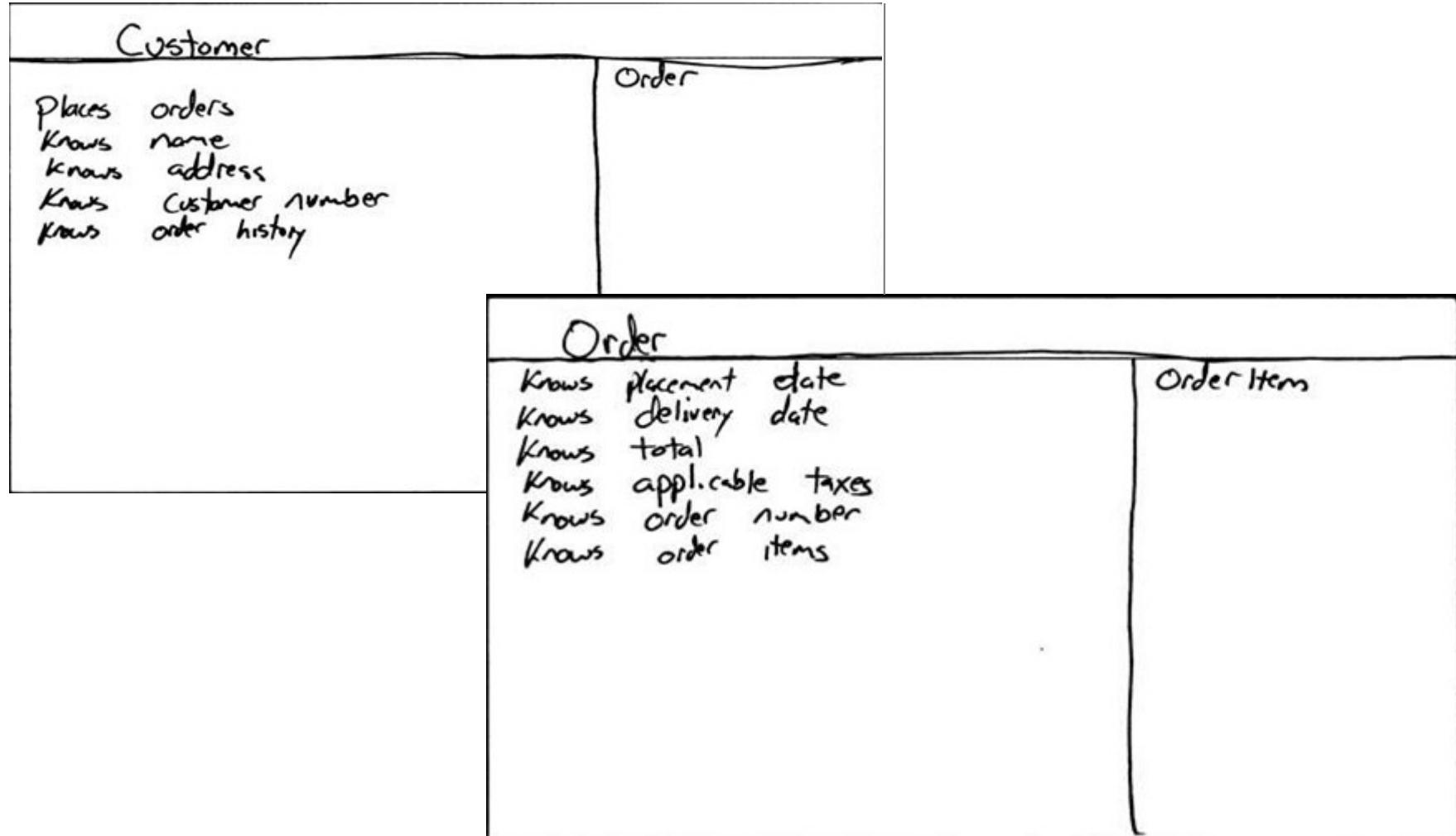


Class Responsibility Collaboration (CRC)

- Digunakan untuk memodelkan interaksi antar objek.
- Setiap kelas direpresentasikan dengan sebuah kartu.
- CRC nantinya digunakan sebagai acuan untuk menggambarkan hubungan antar objek.

Class Name	
Responsibilities	Collaborations
(what the class does)	(related objects)

Class Responsibility Collaboration (CRC)



Identifikasi Hubungan Antar Objek

- Hubungan Antar objek (dalam UML) :
 - Dependency
 - Asosiasi
 - Agregasi
 - Komposisi
 - Pewarisan / Inheritance / Generalisasi / Spesialisasi

Identifikasi Hubungan Antar Objek

- 
- Hubungan antar objek diidentifikasi berdasarkan karakteristik masing-masing hubungan.
 - Memanfaatkan CRC dan Communication diagram.

Identifikasi Atribut Kelas

- Atribut merupakan apa yang menjadi ciri/milik objek.
- Atribut merupakan sesuatu yang bisa dijelaskan.
- Hal-hal yang bisa membantu :
 - ▣ Carilah frase-frase yang termasuk posesif.
 - ▣ Atribut merepresentasikan apa yang akan disimpan pada kelas entity.
 - ▣ Atribut yang berupa objek, diwujudkan dalam bentuk asosiasi.

Membuat Kelas Diagram

- Kelas diagram merupakan salah satu luaran dari analisis.
- Dibuat dengan menghubungkan kelas yang sudah teridentifikasi.

Referensi

- Brugge, B & Dutoit, A.H ; 2004 ; *Object Oriented Software Engineering Using UML, Patterns And Java* ; Prentice Hall
- Hamilton,K, & Mikes, R ; *Learning UML2.0* ; O'Reilly

CLASS DIAGRAM

Panji Wisnu Wirawan

Agenda

- 
- Pengertian
 - Notasi
 - Dependency
 - Asosiasi
 - Generalisasi
 - Multiplicity
 - Contoh Kasus

Pengertian

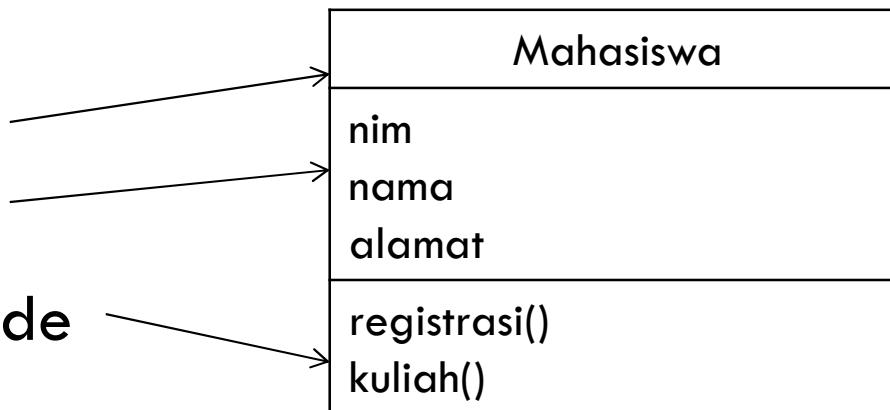
- Model statik UML.
- Memodelkan kelas dan hubungannya dengan kelas yang lain.

- Elemen :

- Nama kelas

- State/Atribut

- Behavior/Metode



Notasi

- Class Diagram boleh tidak digambarkan secara lengkap elemennya.
- Alternatif :
 - Nama kelas
 - Nama kelas + atribut
 - Nama kelas + metode

Mahasiswa

Mahasiswa

Mahasiswa

nim
nama
alamat

registrasi()
kuliah()

Notasi

- Tipe data atribut dan parameter metode dapat ditambahkan pada class diagram

Mahasiswa
nim : char
nama : string
alamat : string
registrasi(nim : char, nama : string)
kuliah(kuliah : MataKuliah)

Notasi

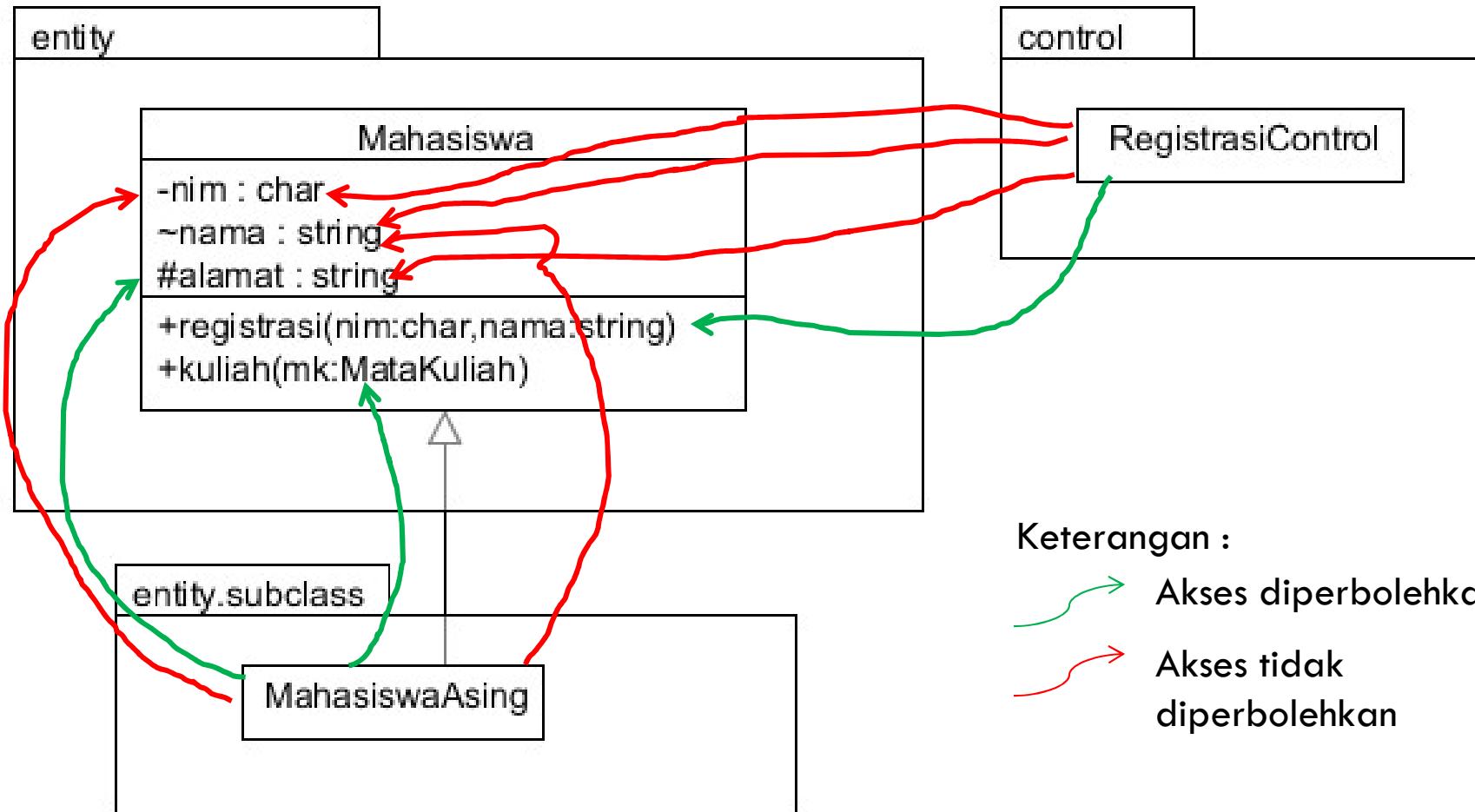
- ‘Visibility’ setiap atribut dan metode dapat ditambahkan pada class diagram.
- Visibility adalah salah satu cara untuk *showing/hiding* data.
- Visibility :
 - Private (-)
 - Package (~)
 - Protected (#)
 - Public (+)

Mahasiswa
- nim : char
~nama : string
#Alamat : string
+registrasi(nim : char, nama : string)
+kuliah(kuliah : MataKuliah)

Notasi

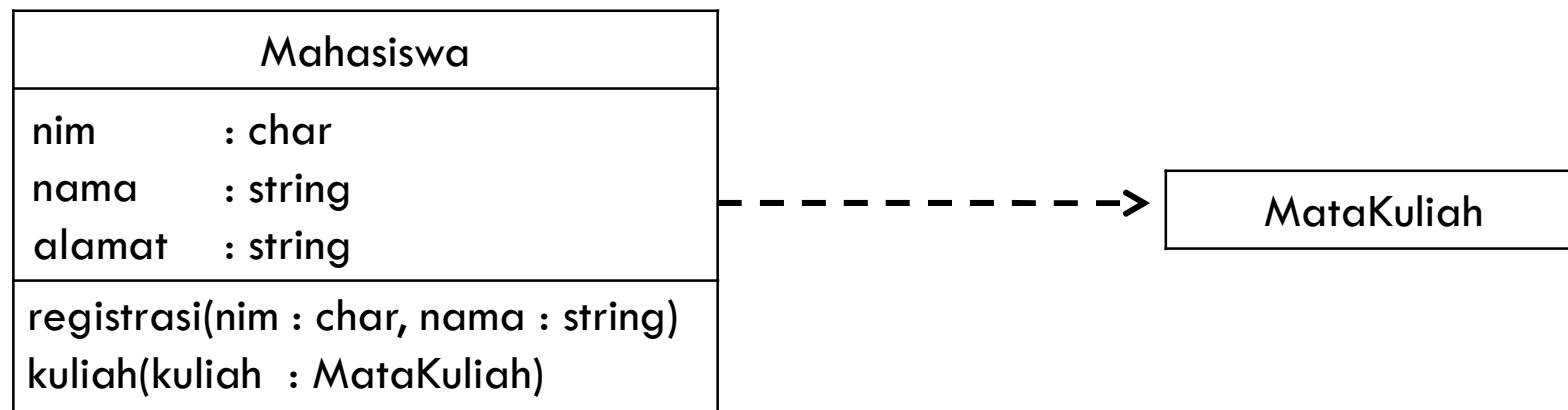
- Private : atribut/metode hanya dapat digunakan oleh kelas yang bersangkutan saja. Kelas yang lain tidak dapat menggunakan secara langsung.
- Package : atribut/metode hanya dapat digunakan oleh kelas-kelas dalam 1 “package”. (package : mekanisme pengelompokan kelas).
- Protected : atribut/metode hanya dapat digunakan oleh kelas-kelas dalam 1 “package”, dan kelas lain yang merupakan spesialisasi dari kelas tsb, walaupun berbeda “package”.
- Public : atribut/metode dapat digunakan oleh semua kelas tanpa terkecuali.

Notasi



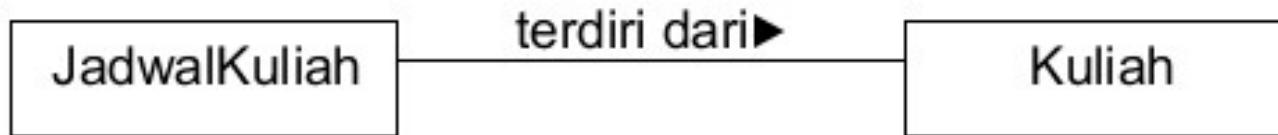
Dependency

- Hubungan antar kelas yang menunjukkan bahwa sebuah kelas perlu tahu kelas yang lain untuk menggunakan objek dari kelas yang lain tersebut.
- Objek yang dihubungkan dengan Dependency digunakan secara singkat.



Asosiasi

- Hubungan antar kelas yang menunjukkan bahwa satu kelas menunjuk kepada kelas yang lain melalui atribut.
- Dengan asosiasi, kelas akan bekerja dengan kelas yang lain, termasuk memberi nilai, mengolah nilai yang ada pada objek yang dibentuk kelas lain tersebut.



Multiplicity

- Class diagram dapat memiliki multiplicity, yang menunjukkan seberapa banyak objek yang bekerja dengan objek yang lain.
- Multiplicity dapat dikenakan pada asosiasi, agregasi, dan komposisi.

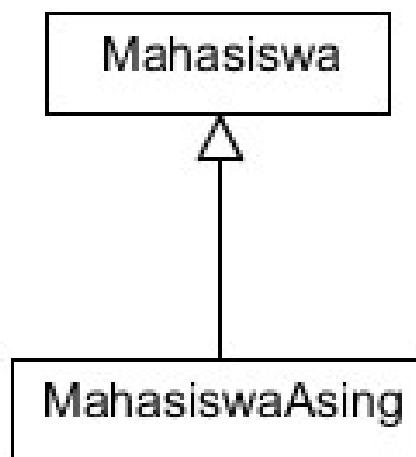
Satu objek yang terlibat	1
Satu objek atau tak terbatas	1..*
Objek yang terlibat tidak terbatas	*
Minimal 2 objek yang terlibat maksimal 4 (range)	2..4

Multiplicity



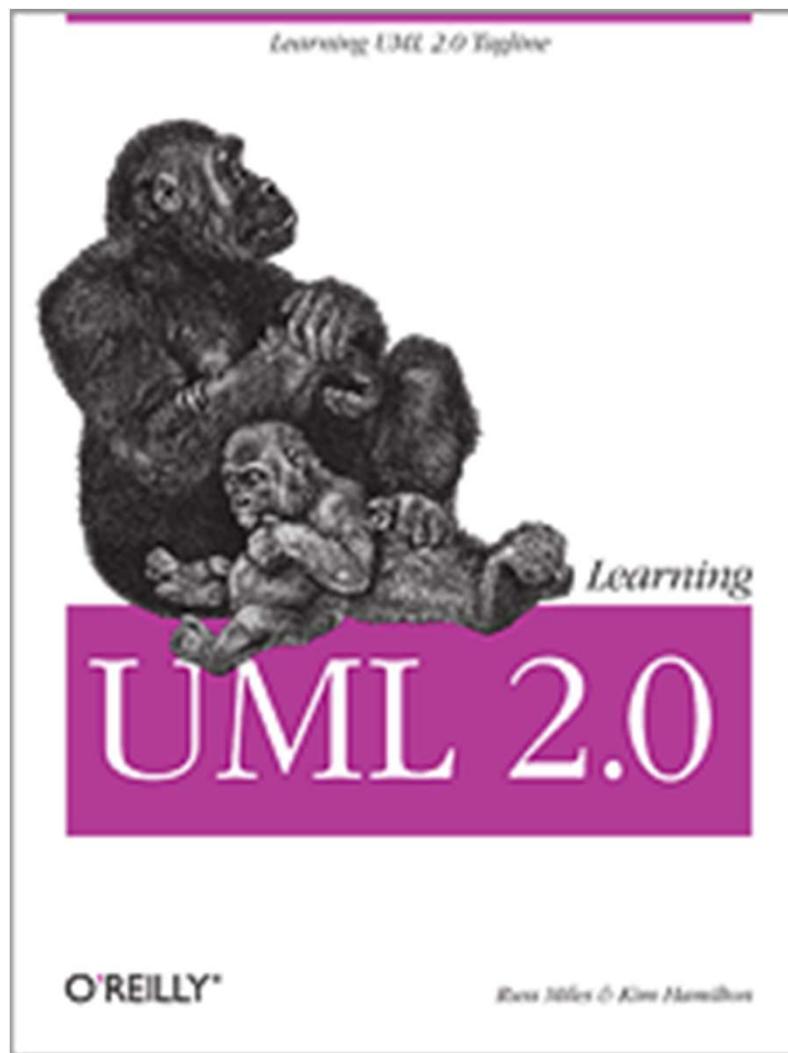
Generalisasi

- Menunjukkan bahwa sebuah kelas merupakan tipe dari kelas yang lain.
- Dikenal dengan *inheritance/pewarisan / IS-A relationship*.



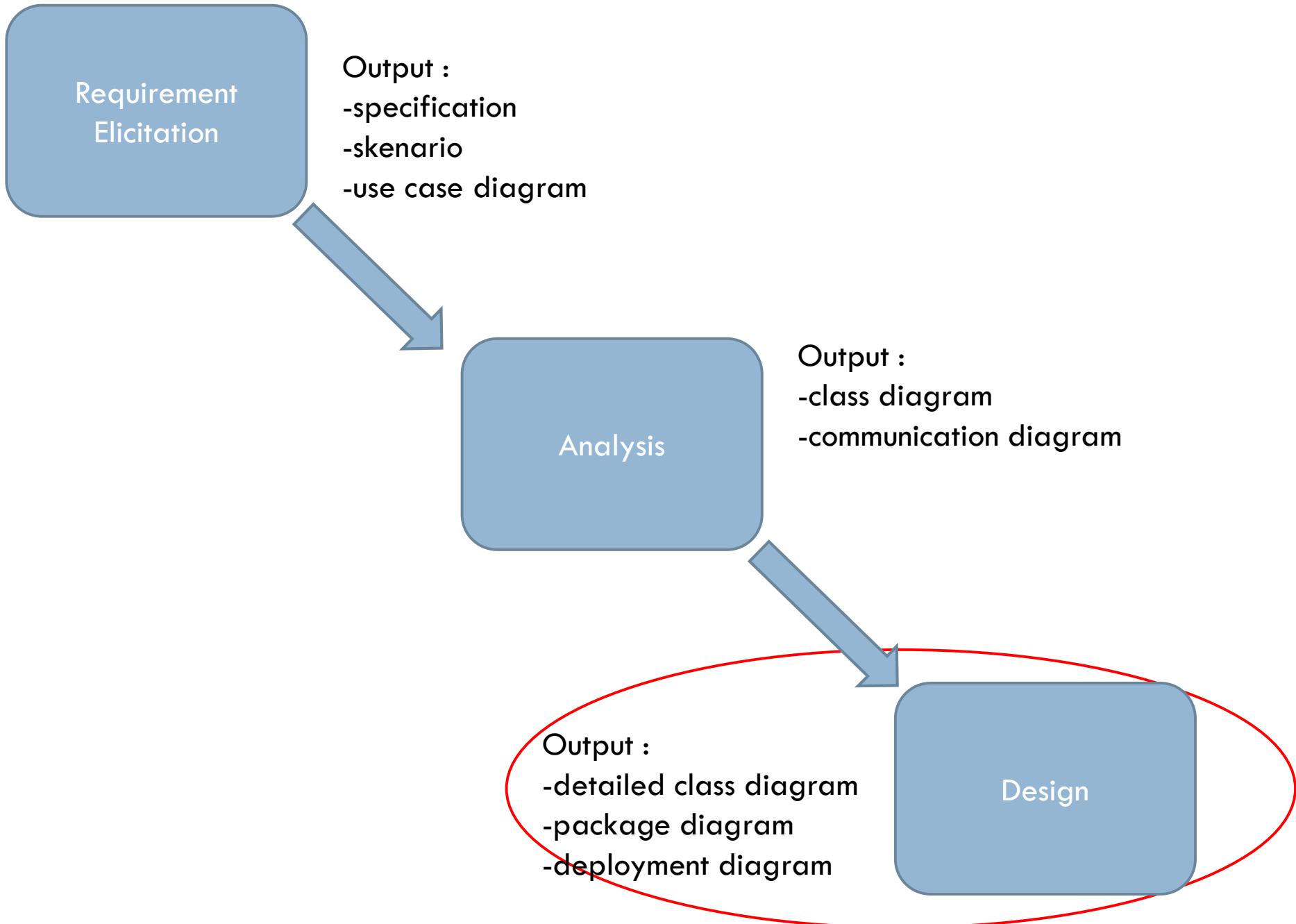
- Mahasiswa Asing merupakan tipe dari mahasiswa
- Mahasiswa Asing IS-A Mahasiswa
- Mahasiswa Asing mewarisi atribut dan metode dari Mahasiswa
- atribut yang dapat diwariskan : public, protected.
- Mahasiswa disebut “super class”
- Mahasiswa Asing disebut “sub class”

Referensi



OBJECT ORIENTED DESIGN

Panji Wisnu Wirawan



Agenda



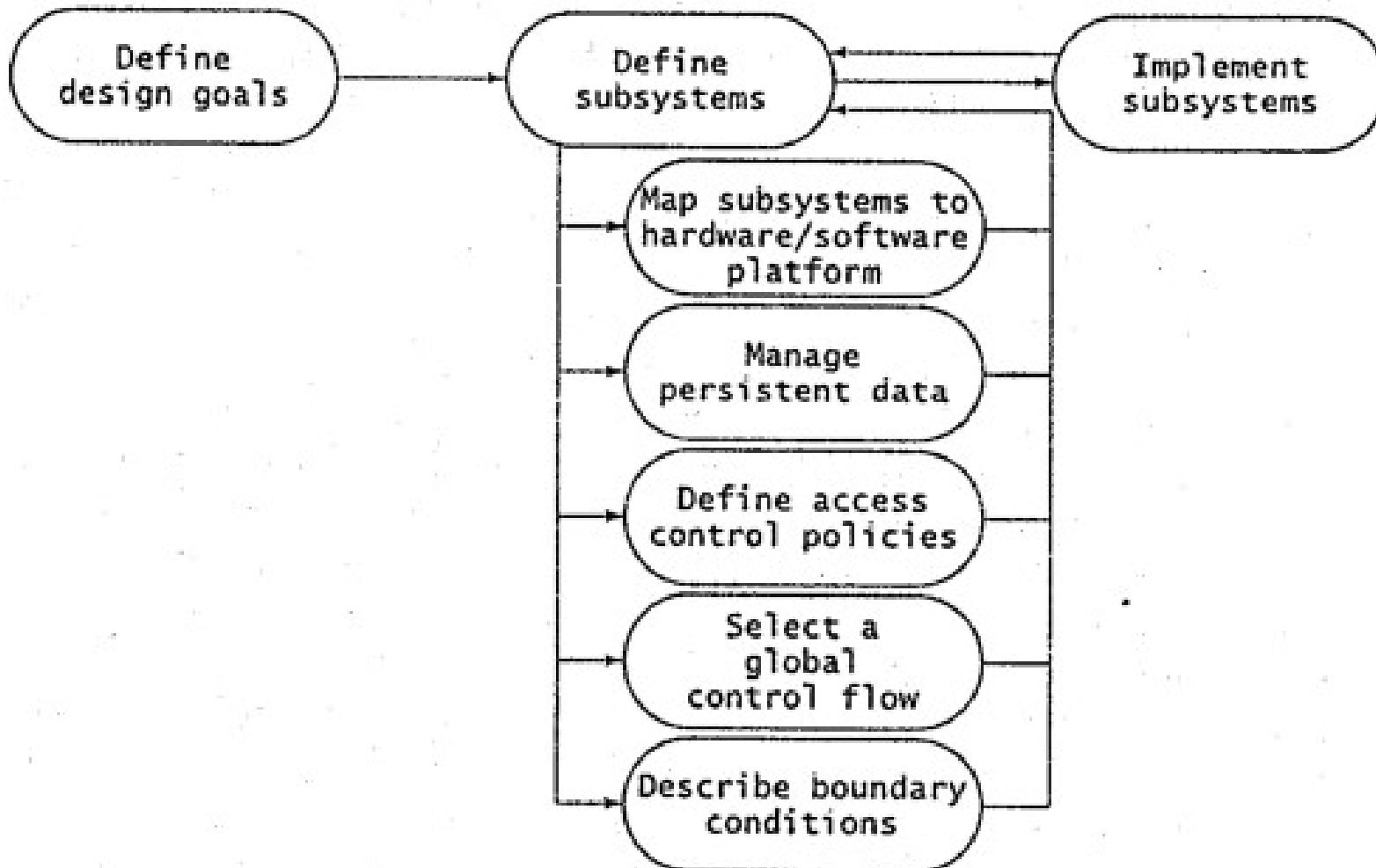
- Prinsip Design
- Aktivitas Design
- Dekomposisi Sistem
- Menuju Tujuan Desain
- *Object Design*
- Dokumentasi Fase Desain

Prinsip Design



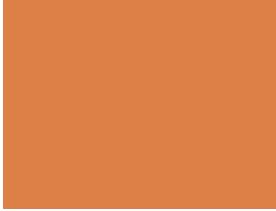
- Fokus : bagaimana sistem melakukan fungsinya.
- Mengarah ke hal teknis implementasi.
- Mapping model analisis → model desain.
- Dekomposisi sistem.

Aktivitas Desain



Aktivitas Design

- Identifikasi tujuan design.
- Melakukan design awal : dekomposisi sistem.
 - Membuat sub-sistem
 - Mapping sub-sistem ke hardware/software platform.
 - Manage persistent data
 - Define access control policies
 - Select global control flow
 - Describe boundary condition
- Revisi dekomposisi sistem untuk mencapai tujuan design.

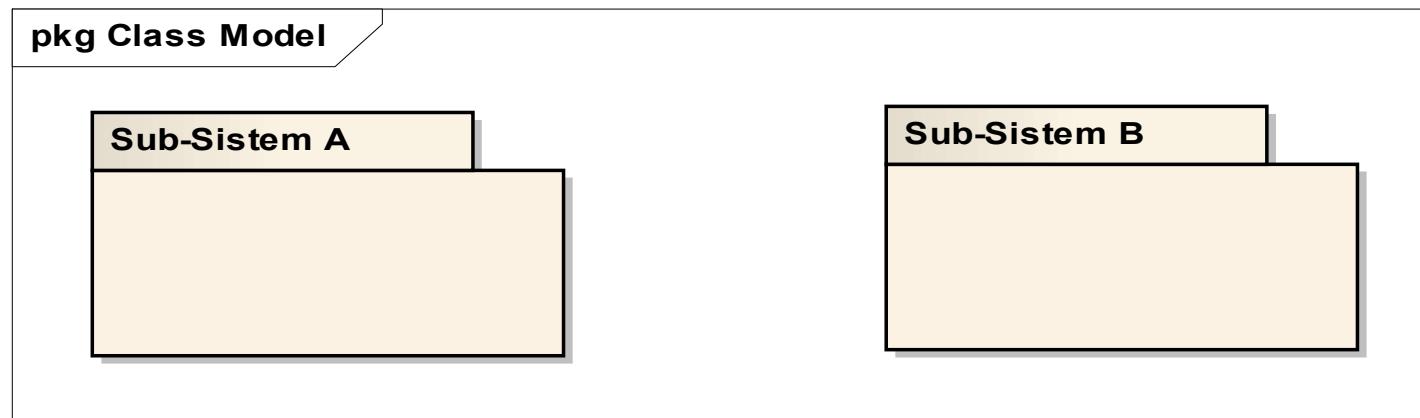


Dekomposisi Sistem

@see oose ch.6

Dekomposisi Sistem

- Membuat sistem menjadi sub-sub sistem.
- Pemodelan UML : package diagram.



- Karakter Sub-sistem : memberikan layanan pada sub-sistem yang lain.

Dekomposisi Sistem

- Dekomposisi Sistem harus memperhatikan *coupling* & *cohesion*.
- *Coupling* : dependensi antar sub-sistem.
 - Loosely coupled → relatif independen
 - Strongly coupled → ketergantungan tinggi
- *Cohesion* : dependensi didalam sub-sistem
 - Sub-sistem dengan banyak objek dengan ketergantungan tinggi : *high cohesion*.
 - Sub-sistem dengan banyak objek yang masing-masing objek independen : *low cohesion*.

Dekomposisi Sistem

- Cara dekomposisi :
 - Objek-objek dalam sebuah use case, dijadikan sebuah sub-sistem.
 - Membuat subsistem yang terdiri dari objek-objek yang digunakan untuk perpindahan data antar sub-sistem.
 - Meminimalkan jumlah hubungan asosiasi antar sub-sistem.
 - Semua objek dalam sub-sistem yang sama, secara fungsional harus berkaitan.



Menuju Tujuan Desain

@see oose ch. 7

Topik



- Mapping sub system to hardware/software platform.
- Manage persistent data.
- Define access control policies.
- Select global control flow.
- Describe boundary condition.

Mapping sub system to ...

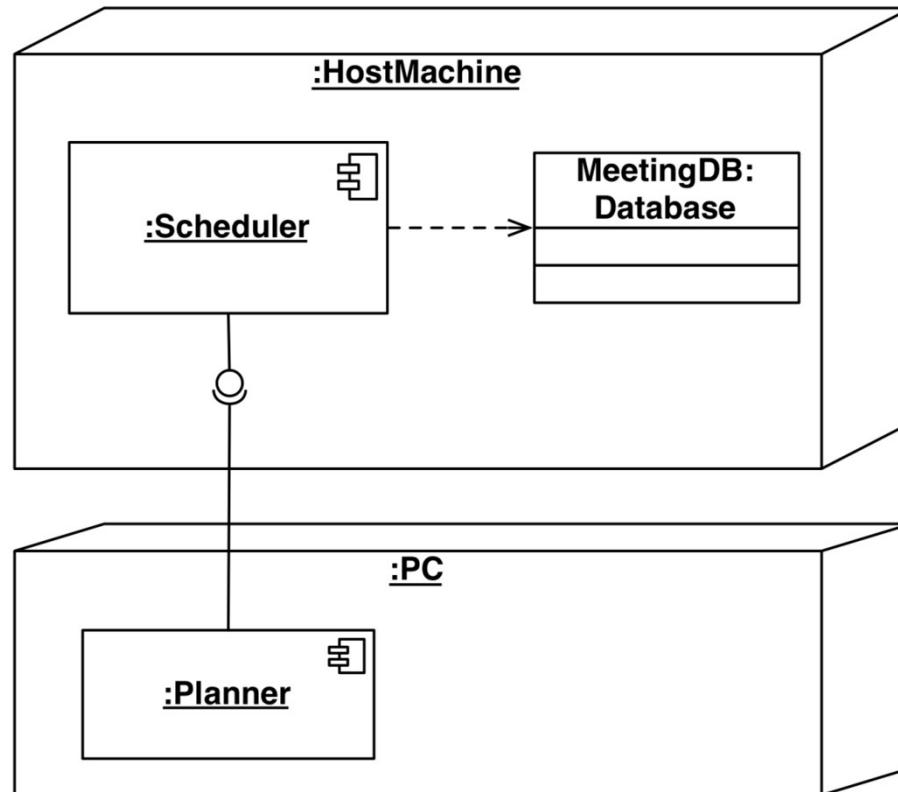
- Digunakan untuk memetakan subsistem yang mana yang berada di hardware/platform yang mana.
- Diperlukan karena :
 - Control Object memerlukan prosesor
 - Entity Object memerlukan memory
 - Boundary Object memerlukan devices

Mapping sub system to ...

- Pertanyaan yg dapat membantu :
 - Bagaimana subsistem direalisasikan ? Dengan hardware atau software ?
 - Bagaimana memetakan model ke dalam hardware yang dipilih?
- Setiap subsistem yang direalisasikan dengan hardware yang berbeda, dapat dihubungkan (asosiasi).

Mapping sub system to ...

Contoh (deployment diagram) :

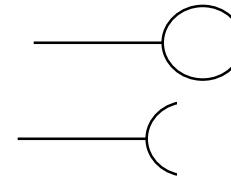


Keterangan :

- 1.Terdapat 2 platform (PC & HostMachine)
- 2.Host Machine : terdapat komponen Scheduler & Object untuk menangani database.
3. PC:terdapat komponen Planner

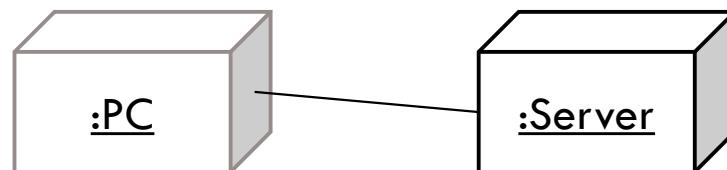
Mapping sub system dalam UML

- UML dapat digunakan untuk menggambarkan sub sistem.
- Sub sistem dapat digambarkan dengan package atau component diagram.
- Menunjukkan bagaimana software dirangkai menjadi satu.
- Setiap komponen memberikan atau memerlukan interface.
 - Memberikan interface
 - Memerlukan interface



Mapping sub system dalam UML

- Hardware/platform dapat dimodelkan menggunakan deployment diagram.
- Setiap hardware/platform dimodelkan menggunakan node (3D box) dan setiap node dapat berasosiasi.



- Dapat menggambarkan bagaimana komponen disusun, hubungan antar komponen dan hubungan antar hardware/platform.

Manage Persistent Data

- Sebagian objek dalam sistem perlu untuk menetap (persist) dalam sistem.
- Mekanisme :
 - File system (multi readers, single writer)
 - Database (concurrent readers & writers)
- Pertanyaan yang dapat membantu :
 - Seberapa sering akses data ?
 - Perlukah arsip data ?
 - Apakah data akan didistribusikan ?
 - Perlukah interface tunggal untuk akses data ?

Manage Persistent Data



- UML dapat digunakan untuk mapping model objek ke model relasional.
- Ketentuan dasar :
 - Setiap kelas dipetakan ke dalam satu tabel.
 - Setiap atribut dipetakan ke dalam satu kolom.
 - Instance dari kelas (objek) direpresentasikan dalam tuple.
 - Method tidak dipetakan.

Define Access Control Policies

- 
- Mendefinisikan mengenai siapa berhak mengakses fungsi dan data apa.
 - Digunakan untuk sistem dengan multi-user.
 - Ditentukan berdasarkan use case & aktor (analisis) dan berdasarkan objek yang dibagi-pakai oleh aktor (desain).
 - Dapat dimodelkan dengan access matrix.

Define Access Control Policies

The diagram illustrates access control policies for four classes: Arena, League, Tournament, and Match. These classes are represented by columns in a table. The rows represent different actors: Operator, LeagueOwner, Player, and Spectator. Each cell in the table contains a list of access rights. The table is annotated with three callout bubbles: 'Classes' points to the column headers, 'Access Rights' points to the list of methods in the first cell, and 'Actors' points to the row headers.

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
LeagueOwner	view ()	edit ()	<<create>> archive() schedule() view()	<<create>> end()
Player	view() applyForOwner()	view() subscribe()	applyFor() view()	play() forfeit()
Spectator	view() applyForPlayer()	view() subscribe()	view()	view() replay()

Select Global Control Flow

- Digunakan untuk memodelkan urut-urutan operasi sistem → menentukan bagaimana control object diimplementasikan.
- Mekanisme
 - Procedure driven : operasi yang menunggu input dari aktor.
 - Event driven : main loop yang menunggu external event.
 - Thread : concurrent procedure driven.
- Pertanyaan yg dapat membantu memilih control flow :
 - Bagaimana urutan operasi sistem ?
 - Apakah sistem tsb. event driven / procedure driven ?

Describe Boundary Condition



- Penentuan bagaimana :
 - Sistem di-inisialisasi.
 - Sistem shut down.
 - Kasus-kasus exceptional ditangani.
- Acuan : scenario, use case dan actor yang menjalankannya.
- Dapat digunakan untuk menentukan exception class yang menangani kasus-kasus dalam exceptional scenario.



Object Design

@OOSE Ch.9

Object Design



- Kegiatan untuk lebih mendetailkan artefak yang sudah dedefinisikan sebelumnya, supaya siap diimplementasikan.
- Fokus : *interface specification*
- Kegiatan :
 - Identifikasi atribut dan perilaku yang belum tercantum.
 - Menentukan *signature & visibility*.
 - Menentukan kontrak.

Identifikasi atribut & perilaku yang belum tercantum

- Menggunakan acuan diagram kelas analisis.
- Diagram kelas analisis berfokus pada fungsionalitas, tidak detail.
- Diagram kelas desain mendetailkan atribut dan perilaku yang diperlukan untuk merealisasikan layanan (service) pada sub sistem (hasil dekomposisi).

Mendefinisikan signature & visibility

- *Signature : method parameter & return value.*
- *Visibility : kemampuan atribut/operasi untuk digunakan pada kelas atau sub sistem lain.*
- Menggunakan hasil diagram kelas analisis.
- Tujuan :
 - Mengurangi coupling antar subsistem.
 - Memberikan interface yang sederhana yang mudah dipahami.

Menentukan Kontrak

- Kontrak : batasan (*constraints*) pada kelas yang bisa membuat asumsi yang sama mengenai kelas tersebut.
- Jenis batasan :
 - Invariant : predikat yang nilainya selalu benar.
 - Precondition : batasan sebelum sebuah operasi dijalankan.
 - Postcondition : batasan setelah sebuah operasi dijalankan.

Dokumentasi Fase Desain

- Hal-hal yang dikerjakan pada fase desain, didokumentasikan pada dokumen desain.
- Isi dokumen desain :
 - System Design Document**
 - 1. Introduction
 - 1.1 Purpose of the system
 - 1.2 Design goals
 - 1.3 Definitions, acronyms, and abbreviations
 - 1.4 References
 - 1.5 Overview
 - 2. Current software architecture
 - 3. Proposed software architecture
 - 3.1 Overview
 - 3.2 Subsystem decomposition
 - 3.3 Hardware/software mapping
 - 3.4 Persistent data management
 - 3.5 Access control and security
 - 3.6 Global software control
 - 3.7 Boundary conditions
 - 4. Subsystem services
 - Glossary

Kesimpulan



- Fase desain diperlukan untuk mempersiapkan model siap diimplementasikan.
- Kegiatan desain yang dilakukan meliputi : dekomposisi sistem, memenuhi tujuan desain, dan object design.
- Hasil kegiatan di fase desain dituliskan pada dokumen desain perangkat lunak.

Deployment and Maintenance

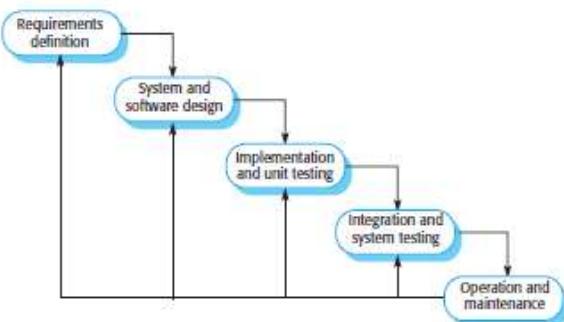
MK - Rekayasa Perangkat Lunak

Ilmu Komputer/Informatika

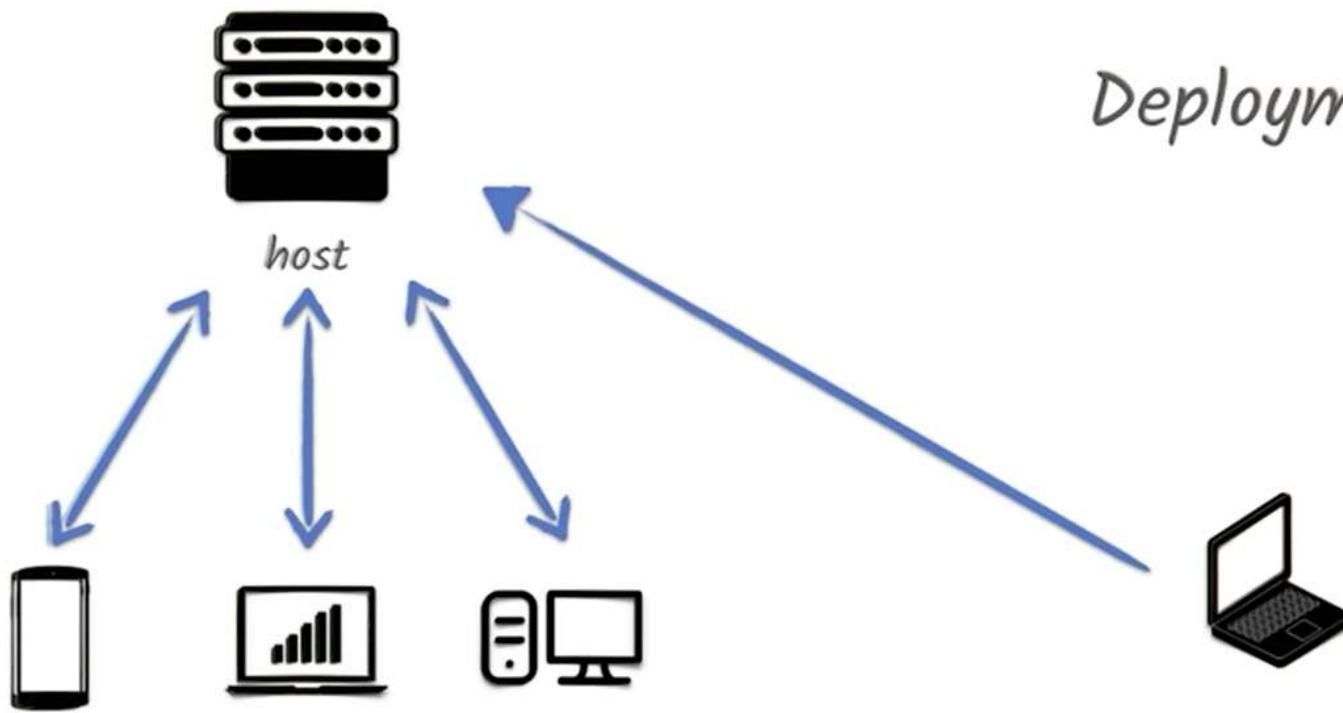
Universitas Diponegoro

Deployment

- is the process that makes software available for use
- includes all of the steps, processes, and activities that are required to make a software system or update available to its intended users.
- Development to Production

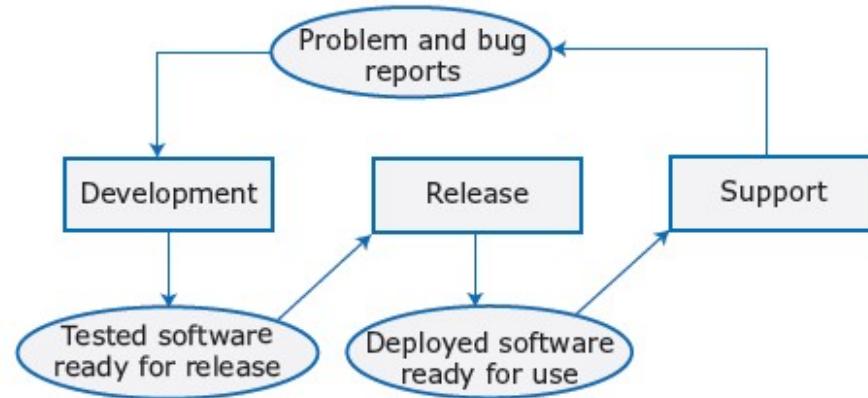


Deployment



Traditional Model

- separate teams were responsible for software development, software release, and software support

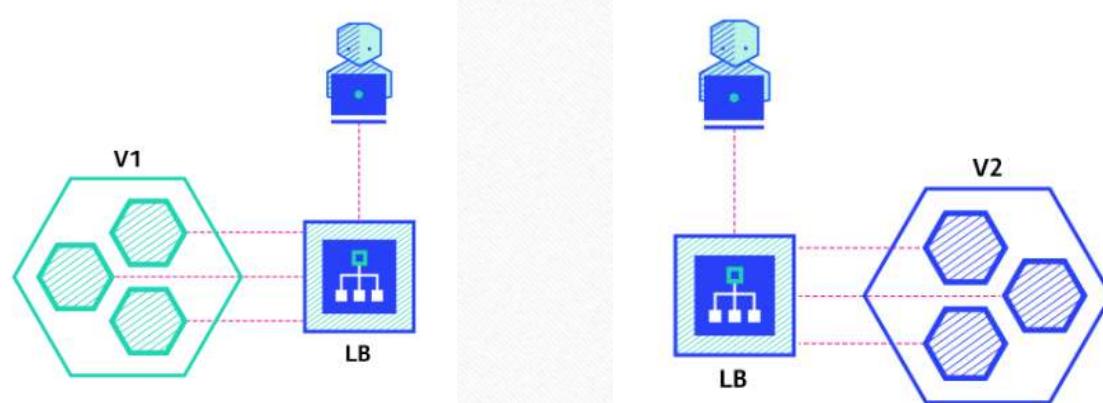


Deployment Strategies

- Recreate
- Rolling Updates
- Blue/Green
- Canary
- A/B Testing

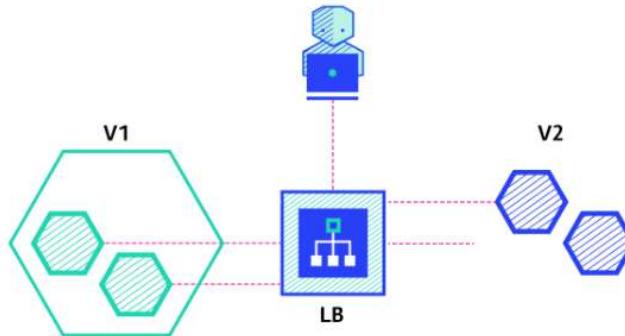
Recreate

- Shutting down version A then deploying version B after version A is turned off



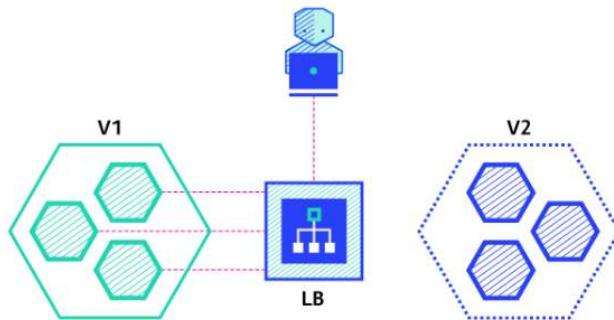
Rolling Updates

- slowly rolling out a version of an application by replacing instances one after the other until all the instances are rolled out



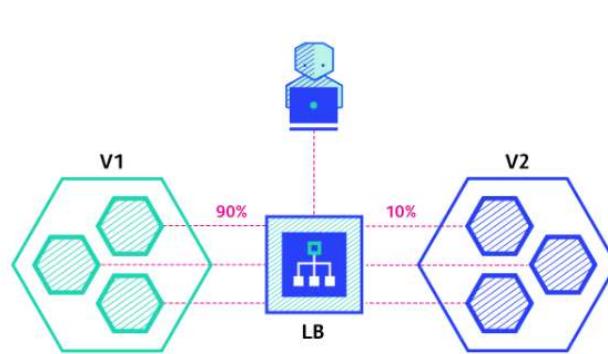
Blue/Green

- version B (green) is deployed alongside version A (blue) with exactly the same amount of instances.



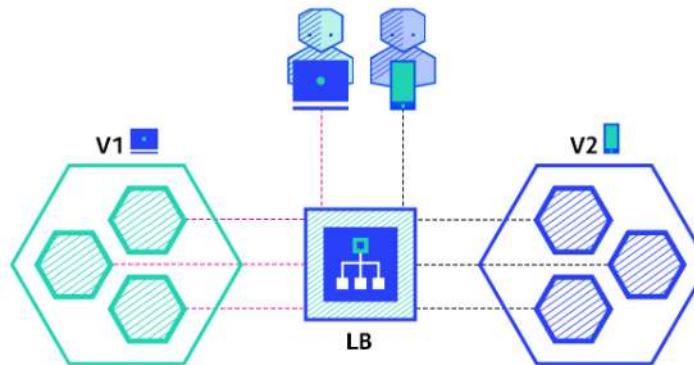
Canary

- gradually shifting production traffic from version A to version B. Usually the traffic is split based on weight.



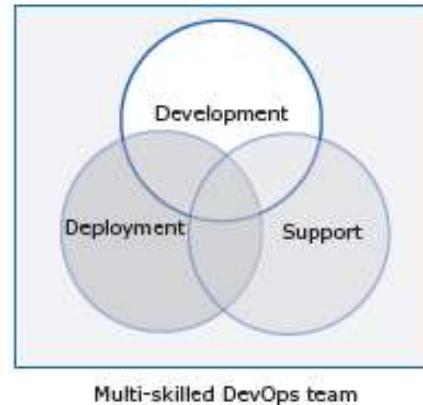
A/B Testing

- routing a subset of users to a new functionality under specific conditions.



DevOps

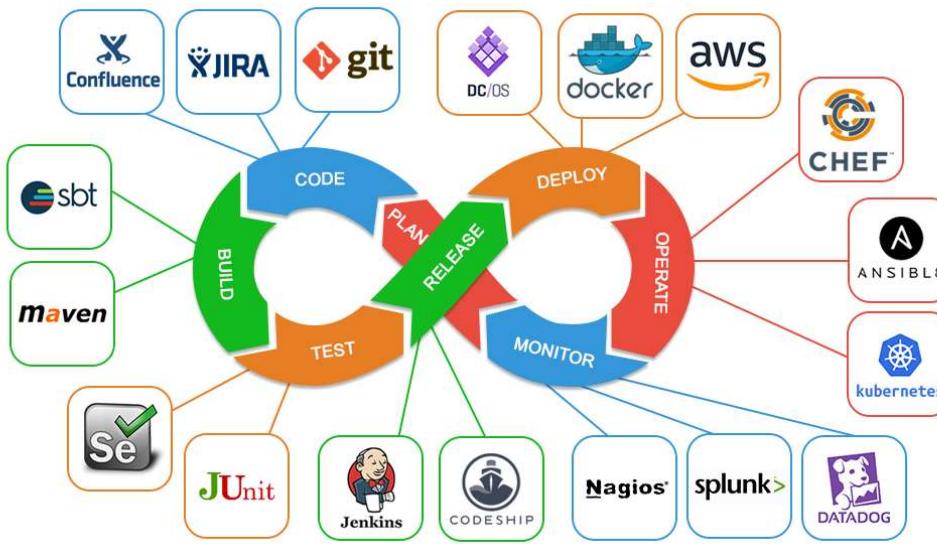
- Integrates development, deployment, and support



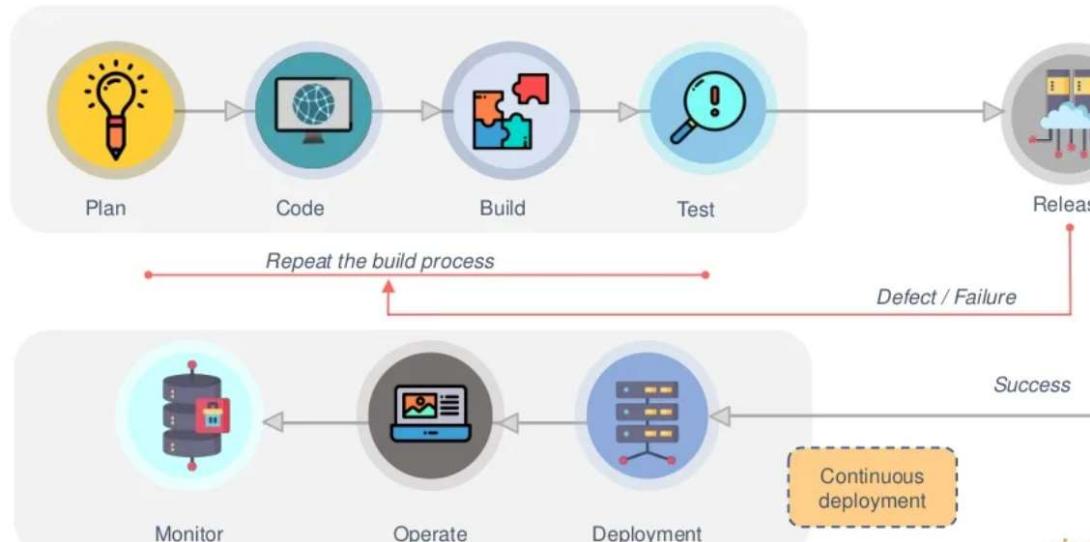
DevOps Principle

Principle	Explanation
Everyone is responsible for everything.	All team members have joint responsibility for developing, delivering, and supporting the software.
Everything that can be automated should be automated.	All activities involved in testing, deployment, and support should be automated if it is possible to do so. There should be minimal manual involvement in deploying software.
Measure first, change later.	DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

Tools



CI/CD



©Simplilearn. All rights reserved.

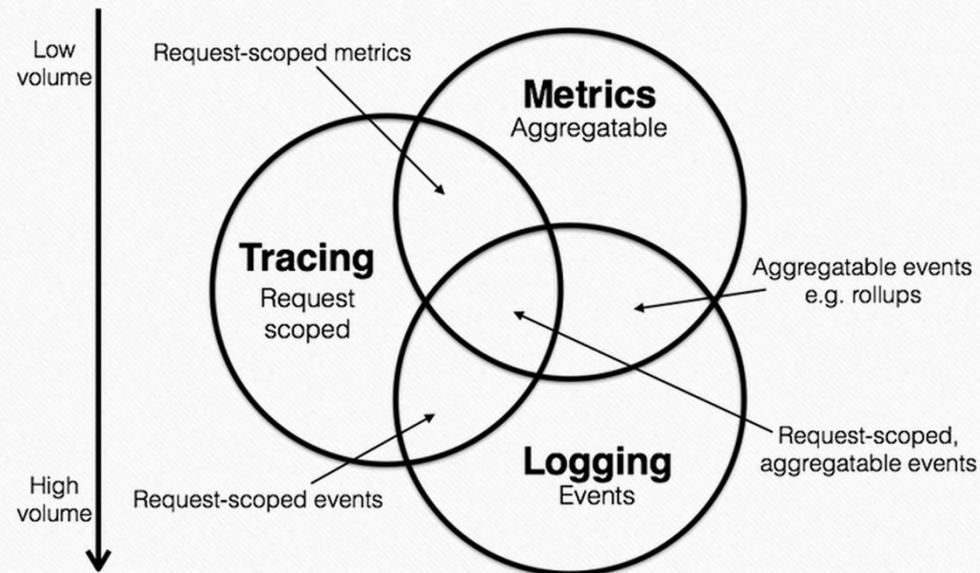
Monitoring

- is the process of collecting log data in order to help developers track availability, bugs, resource use, and changes to performance in applications that affect the end-user experience

Main Function

- To observe app components - Components may include servers, databases, and message queues or catches.
- To provide app dashboards and alerts - Dashboards give an overview, alerts drive attention to specific problems.
- Anomaly detection - Can vary from simple threshold detection to advanced machine learning pattern recognition.
- Distributed tracing - Tracking how one event connects across multiple nodes to detect the origins of errors.
- Dependency & flow mapping - A visual representation of how requests travel between services.

Observability



SOFTWARE TESTING

MK - Rekayasa Perangkat Lunak

Ilmu Komputer/Informatika

Universitas Diponegoro

What is Software Testing

- IEEE/ANSI
 - The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system/component.
 - The process of analyzing a SW item to detect the difference between existing and required conditions (that is, bugs) and to evaluate the features of SW items.
- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.

Things To-Do

- Validation Testing
 - expect the system to perform correctly using a set of test cases that reflect the system's expected use
- Defect Testing
 - the test cases are designed to expose defects

“Testing can only show the presence of errors, not their absence” (Dijkstra, 1972)

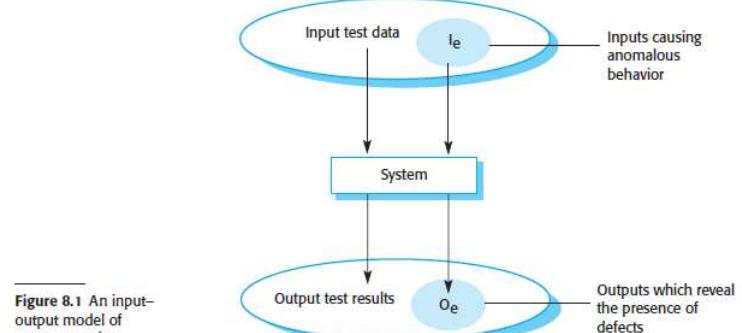


Figure 8.1 An input-output model of program testing

What is Defect

- Defect is a variance from a desired product specification and customer/user expectation
- Categories of defects
 - Wrong – The specification have been implemented incorrectly
 - Missing – A specified requirement is not build into product
 - Extra – A requirement incorporated into the product was not specified
- Defect - Error - Failure
 - Defect is incorporated in software system (within the SW, manuals or documentation) and a flaw
 - Defect that caused an error in operation is called a failure

Validation & Verification

- Concerned with checking that SW being developed meets its specification and delivers the functionality expected by the people paying for the SW.
 - **Validation:** Are we building the right product?
 - **Verification:** Are we building the product right?
- (Boehm, 1979)
- The goal is to establish confidence that the software system is “fit for purpose.”

Level of Confidence

- Depends on :
 - **Software purpose** - the more critical the software, the more important it is that it is reliable.
 - **User expectations** - if the previous experiences with buggy, unreliable software, users sometimes have low expectations of software quality.
 - **Marketing environment** - when a software company brings a system to market, it must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system.

Traditional Testing Process

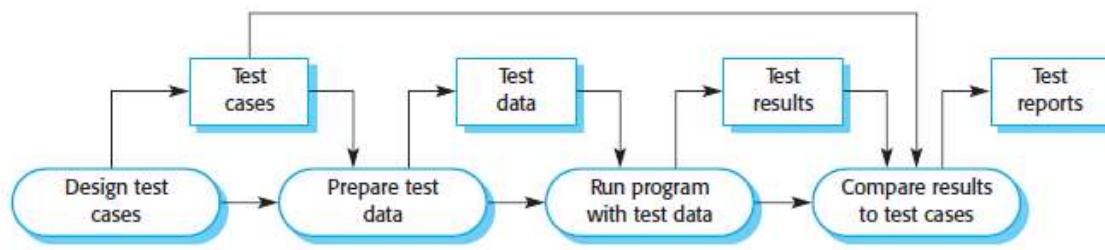


Figure 8.3 A model

- Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested.
- Test data are the inputs that have been devised to test a system.
- Test execution can be automated

Manual vs Automated Testing

- The testing process usually involves a mixture of manual and automated testing.
 - In manual testing, a tester runs the program with some test data and compares the results to their expectations. They note and report discrepancies to the program developers.
 - In automated testing, the tests are encoded in a program that is run each time the system under development is to be tested. – Faster than manual, especially when it involves regression testing
- Unfortunately, testing can never be completely automated as automated tests can only check that a program does what it is supposed to do.

Software Testability

- SW can be tested when:
 - **Operability** – it operates cleanly
 - **Observability** – the result of each test case are readily observed
 - **Controllability** – the degree to which testing can be automated and optimized
 - **Decomposability** – testing can be targeted
 - **Simplicity** – reduce complex architecture and logic to simplify tests
 - **Stability** – few changes are requested during testing
 - **Understandability** – of the design

The Actors

- Tester can be:
 - Developer – Understand the system but will test “gently” and is driven by “delivery”
 - Independent – Must learn about the system, but will attempt to break it and is driven by “quality”
- Testing will involve:
 - SW Customer – that contracts for SW to be developed
 - SW User – that will use the SW
 - SW Developer – that develops the SW
 - SW Tester – that performs the check function on the SW

Test Planning

- Test planning is concerned with scheduling and resourcing all of the activities in the testing process. It involves defining the testing process, taking into account the people and the time available.
- Usually, a test plan will be created that defines what is to be tested, the predicted testing schedule, and how tests will be recorded.
- For critical systems, the test plan may also include details of the tests to be run on the software.

Stages of Testing

- **Development testing**, where the system is tested during development to discover bugs and defects
- **Release testing**, where a separate testing team tests a complete version of the system before it is released to users.
- **User testing**, where users or potential users of a system test the system in their own environment.

Development Testing

Overview

- Development testing includes all testing activities that are carried out by the team developing the system.
- The tester:
 - Programmer it self
 - Associated individual tester
 - Testing group within development team

Stages

- **Unit testing**, where individual program units or object classes are tested.
- **Component testing**, where several individual units are integrated to create composite components.
- **System testing**, where some or all of the components in a system are integrated and the system is tested as a whole.

Unit Testing

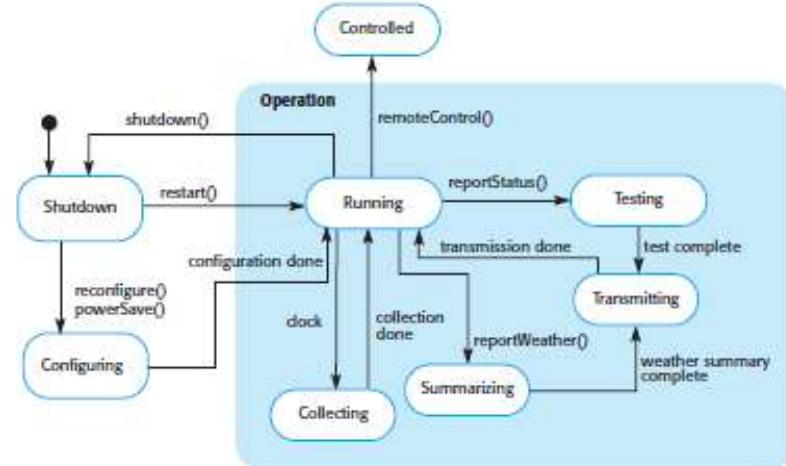
- Unit testing should focus on testing the functionality of objects or methods.
 - Your tests should be calls to these routines with different input parameters.
 - You should design your tests to provide coverage of all of the features of the object.
 - You should test all operations associated with the object; set and check the value of all attributes associated with the object; and put the object into all possible states.
 - You should simulate all events that cause a state change.

Figure 8.4 The weather station object interface

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

Example

- Test a method individually (isolated)
- Test methods in sequence
 - Shutdown → Running → Shutdown
 - Configuring → Running → Testing → Transmitting → Running
 - Running → Collecting → Running → Summarizing → Transmitting → Running



Tools for Automated Unit Test

- JUnit (Tahchiev et al. 2010) - A test automation framework
- Unit testing frameworks provide generic test classes that you extend to create specific test cases.
- They can then run all of the tests that you have implemented and report, often through some graphical unit interface (GUI), on the success or otherwise of the tests.
- An entire test suite can often be run in a few seconds, so it is possible to execute all tests every time you make a change to the program.

Automated Unit Test

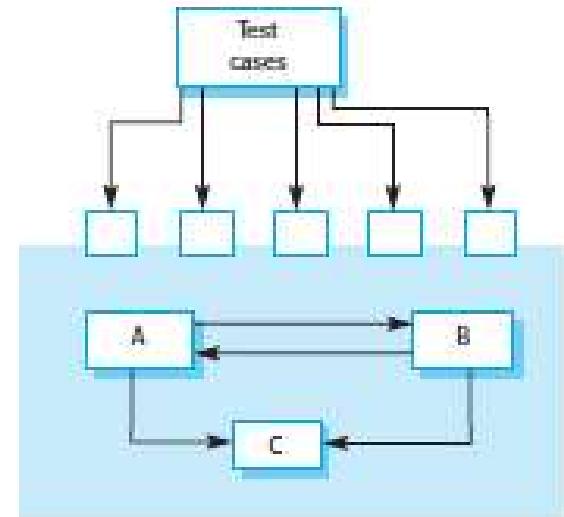
- Parts :
 - A setup part, where you initialize the system with the test case, namely, the inputs and expected outputs.
 - A call part, where you call the object or method to be tested.
 - An assertion part, where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.
- Sometimes, to faster the process we need a mock objects - objects with the same interface as the external objects being used that simulate its functionality

Effective Unit Test Cases

- Effectiveness means:
 - The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
 - If there are defects in the component, these should be revealed by test cases.
- Strategies:
 - Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way. You should choose tests from within each of these groups
 - Guideline-based testing, where you use testing guidelines to choose test cases. These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components

Component Testing

- Software components are often made up of several interacting objects.
- Testing composite components should therefore focus on showing that the component interface or interfaces behave according to its specification.
- You can assume that unit tests on the individual objects within the component have been completed.



Type of Interface

- Parameter interfaces.
 - These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.
- Shared memory interfaces.
 - These are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem and retrieved from there by other subsystems.
- Procedural interfaces.
 - These are interfaces in which one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.
- Message passing interfaces.
 - These are interfaces in which one component requests a service from another component by passing a message to it.

System Testing

- System testing checks that components are compatible, interact correctly, and transfer the right data at the right time across their interfaces.
- Difference with component testing
 - During system testing, 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 is a collective rather than an individual process. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

Release Testing

Overview

- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- Release testing is a process of validation checking to ensure that a system meets its requirements and is good enough for use by system customers.
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.

Types of Release Testing

- Requirement-based testing
- Scenario testing
- Performance testing

User Testing

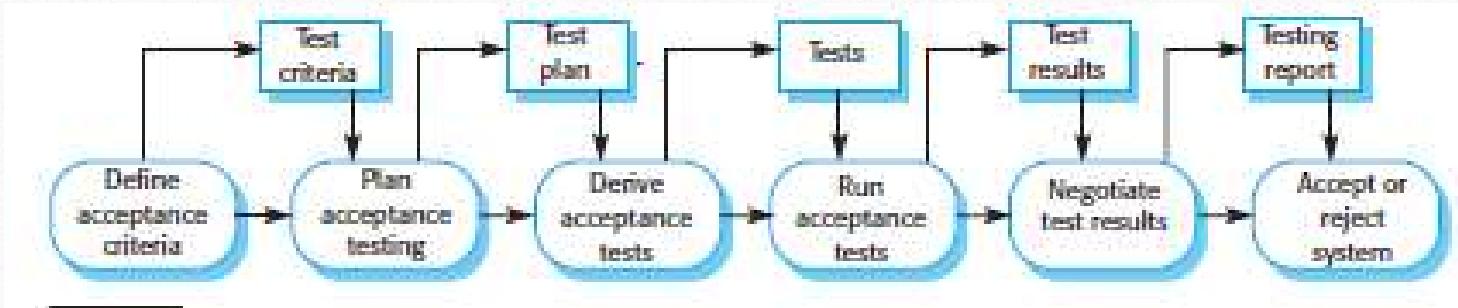
Overview

- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- This may involve formally testing a system that has been commissioned from an external supplier.

Types of User testing

- Alpha testing, where a selected group of software users work closely with the development team to test early releases of the software.
- Beta testing, where a release of the software is made available to a larger group of users to allow them to experiment and to raise problems that they discover with the system developers.
- Acceptance testing, where 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

User Acceptance Test



Perbandingan Metode Perangkat Lunak

Oleh : Prof. Dr. Wiranto Herry Utomo

Software Engineering
A Practitioner's Approach

Seventh Edition

Roger S. Pressman

J.E.D.I.



**Software
Engineering**

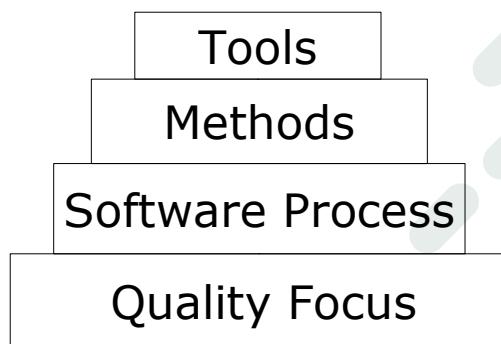
Author
Ma. Rowena C. Solamo

Team
Jaqueline Antonio
Naveen Asrani
Doris Chen
Oliver de Guzman
Rommel Feria
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson
Daniel Villafuerte

Version 1.2
July 3, 2006

Pengertian Metode

A Layered View



Methods

- It provides the technical *how-to's* for building software.
- It is a systematic, established, or orderly procedure or way of doing anything.
- It includes a wide range of tasks such as requirements analysis, design, program construction, testing and maintenance.

5

Examples of Methodology

- Structured Methodologies
 - Information Engineering
 - Systems Development Life Cycle/Project Life Cycle
 - Rapid Application Development Methodology
 - Joint Application Development Methodology
 - CASE*Method

6

Examples of Methodology

- Object-oriented Methodology
 - Booch Method
 - Coad and Yourdon Method
 - Jacobson Method
 - Rumbaugh Method
 - Wirfs-Brock Method

7

Tools

- It provides automated or semi-automated support for the process and methods.
- Most tools are used to develop models.

8

Model

- a simplification of reality

9

Examples of Modeling Tools

- Structured Approach (Digunakan dalam buku Pressman)
 - Data Flow Diagrams
 - Entity-relationship Diagrams
 - Structured English/Pseudocodes
 - Flow Charts
- Object-oriented Approach
 - Unified Modeling Language (UML)

10



Perkembangan Perangkat Lunak dan Pergeseran Paradigma

Perkembangan perangkat lunak

- Sebelum tahun 1970 :
 - Pengembangan Berorientasi Proses
 - Tool pemodelan : DFD
- Tahun 1980 s/d 1995 :
 - Pengembangan Berorientasi Data
 - Pelopor : Peter Chen (buku : Entity–Relationship Approach to Systems Analysis and Design. North-Holland, 1980)
 - Tool pemodelan : ERD

12

Perkembangan perangkat lunak

- Tahun 1995 sd 2010 :
 - Pengembangan Berorientasi Objek
 - Tool pemodelan : UML
- Tahun 2010 sd skrg :
 1. Pengembangan Berorientasi Service
 2. Pengembangan secara Top-Down
 3. Berbasis web service, dengan Metode SOA (Service Oriented Architecture), MDA (Model Driven Architecture)
 4. Tool Pemodelan : MBPM, Activity Diagram

13

Metode Bottom UP

- Asumsi :
- Berbasis kebutuhan pengguna (requirement engineering)
- Membangun software dari NOL, seolah belum ada software sebelumnya

14

Metode Top Down : Metode SOA

- Ketidakselarasan antara sistem bisnis dan sistem informasi biasa terjadi di hampir tiap perusahaan.
- Perbaikan ketidakselarasan ini biasanya kurang membawa hasil karena :
 - 1) kompleksitas arsitektur TI yang berasal dari aplikasi yang heterogen, yang dibangun dari arsitektur dan bahasa pemrograman yang berbeda, serta pada platform yang berbeda
 - 2) aplikasi yang sudah ada ini harus tetap berjalan pada saat diperbaiki .

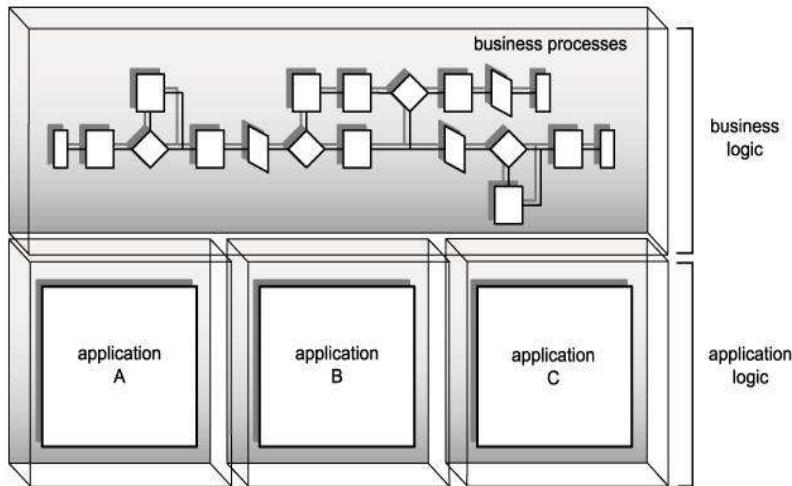
15

Metode Top Down : Metode SOA

- Dalam mengelola masalah keselarasan sistem bisnis dan sistem informasi, dapat menggunakan integrasi sistem dengan metode SOA

16

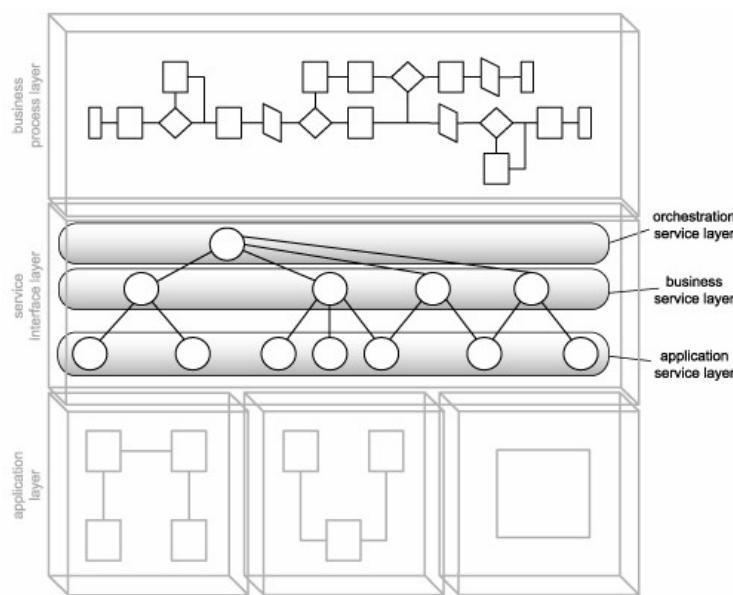
Metode SOA Thomas Erl



Lapisan lojik
bisnis dan lojik
aplikasi dalam
SOA (Erl, 2005)

17

Metode SOA Thomas Erl



Interface Service
Layer pada SOA
(Erl, 2005)

18



Pergeseran Paradigma

Pergeseran Paradigma

- Pergeseran paradigma adalah perubahan besar cara berpikir tentang benda.
- Brooks (1976), mengatakan bahwa pergeseran paradigma yang sesungguhnya sangat jarang terjadi. Peristiwa pergeseran paradigma yang pernah terjadi adalah:
 - Dalam Ilmu Fisika: dari paradigma Newton ke paradigma Einstein.
 - Dalam Ilmu Kimia: dari paradigma Alchemy ke paradigma Atom.
 - Dalam Astronomi: dari “Bible” ke Galileo.
 - Dalam Ilmu Biologi: dari teori Penciptaan Bible ke teori evolusi Darwin.

Pergeseran Paradigma

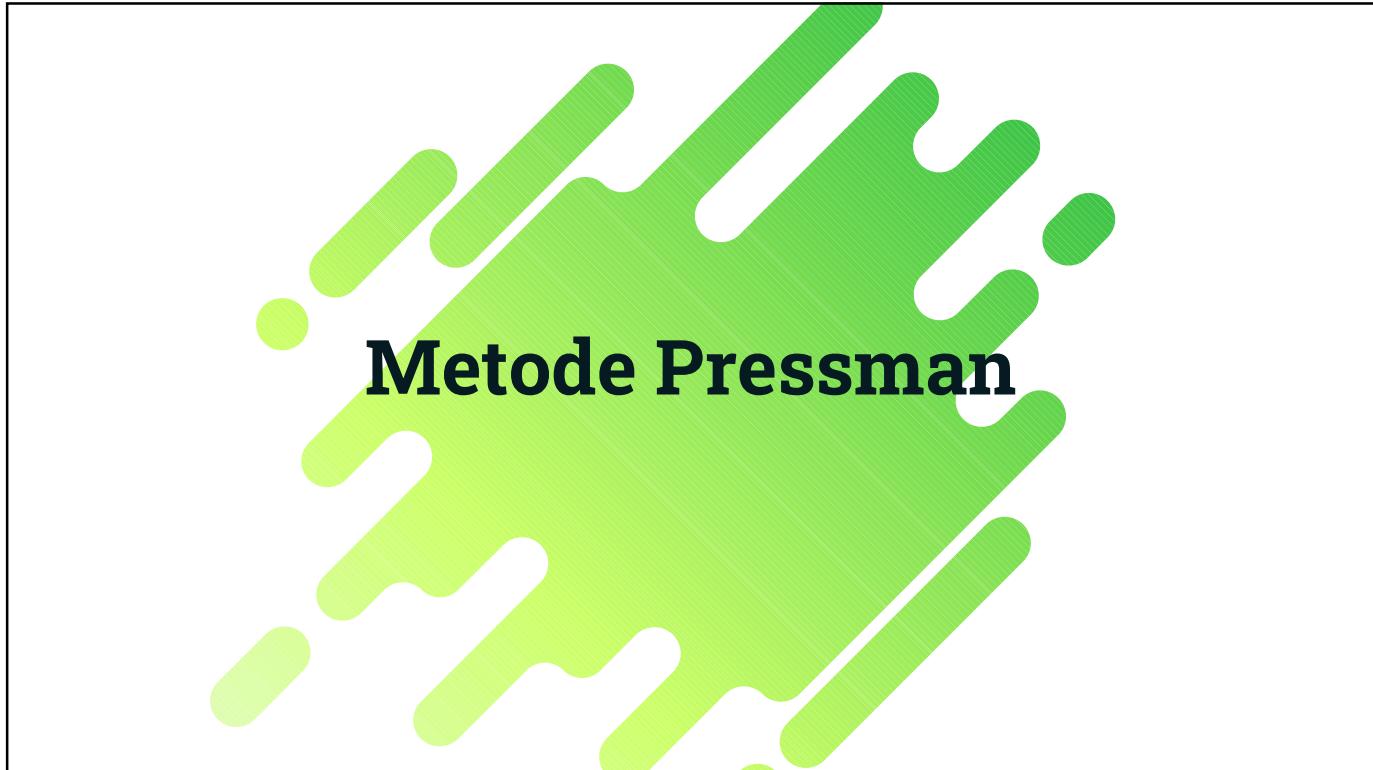
- Setiap peristiwa tersebut di atas telah mengubah cara pandang manusia tentang dunianya.
- Pergeseran paradigma terjadi ketika manusia mengubah cara pandangnya secara radikal.
- Pergeseran paradigma dalam teknologi informasi adalah:
 - Dari Berorientasi Proses (*Process-oriented*) bergeser ke Berorientasi Data (*Data-oriented*)
 - Dari Berorientasi Data (*Data-oriented*) bergeser ke Berorientasi Objek (*Object-oriented*)
 - Cloud Computing

21

Pergeseran Paradigma

- Pergeseran paradigma di bidang TI jarang disadari oleh para praktisi di bidang TI.
- Pergeseran tersebut terjadi tanpa hiruk-pikuk, seperti halnya pergeseran paradigma di bidang astronomi yang memakan korban Galileo.

22



Metode Pressman

The book cover for "Software Engineering: A Practitioner's Approach" by Roger S. Pressman, Seventh Edition, is shown. The cover features a blue and orange design with abstract shapes and the author's name at the bottom.

Understanding Requirement
Requirement Modeling
Design Concept
Construction
Mantainance

```
graph TD; RE[Requirements Engineering] --> DE[Design Engineering]; DE --> C[Coding]; C --> T[Testing]; T --> OMO[Operation and Maintenance]
```

A flowchart illustrates the software engineering process. It starts with "Requirements Engineering", followed by "Design Engineering", then "Coding", "Testing", and finally "Operation and Maintenance". Arrows indicate a sequential flow between these stages. The word "Mantainance" appears to be a typo for "Maintenance".

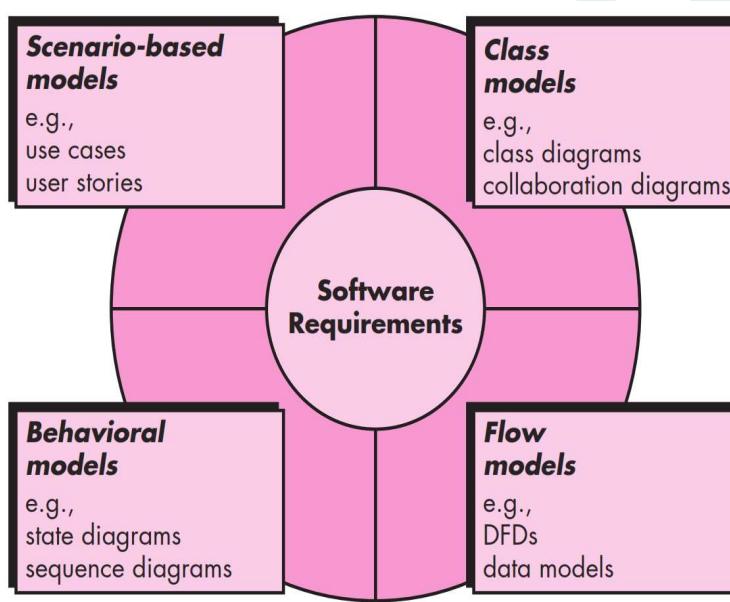
24

Understanding Requirement

- inception,
- elicitation,
- elaboration,
- negotiation,
- specification,
- validation and
- management

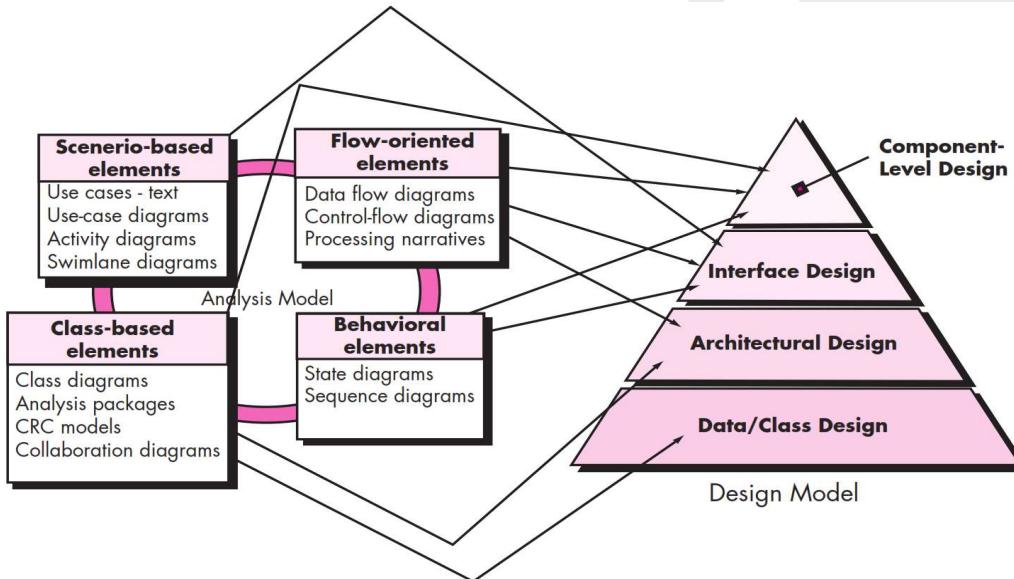
25

Requirement Modeling



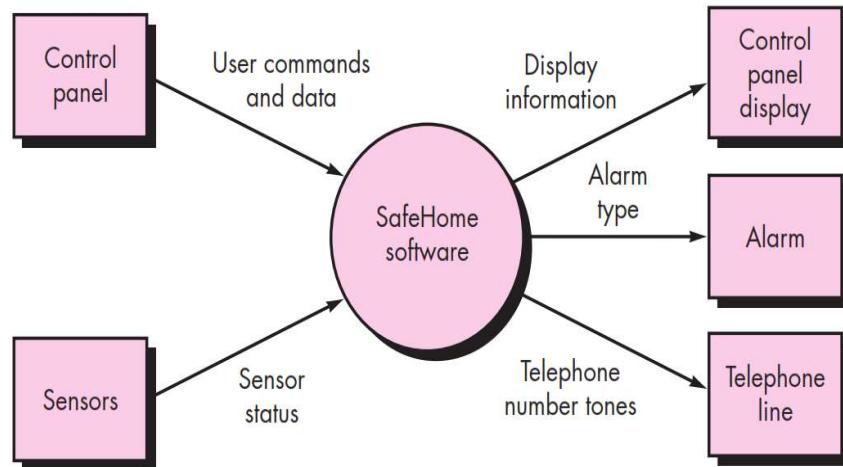
26

Design Concept



27

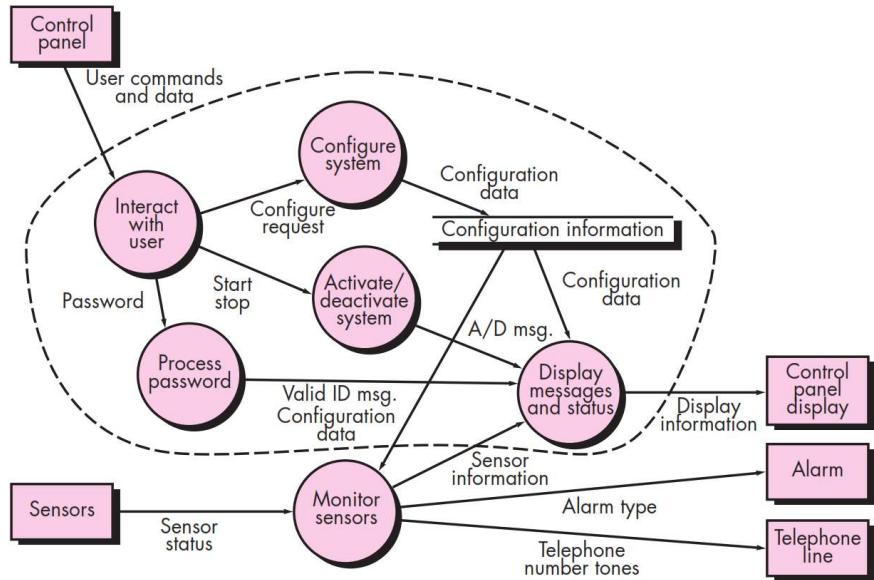
Architectural Design



**Context-level DFD
for the SafeHome
security function**

28

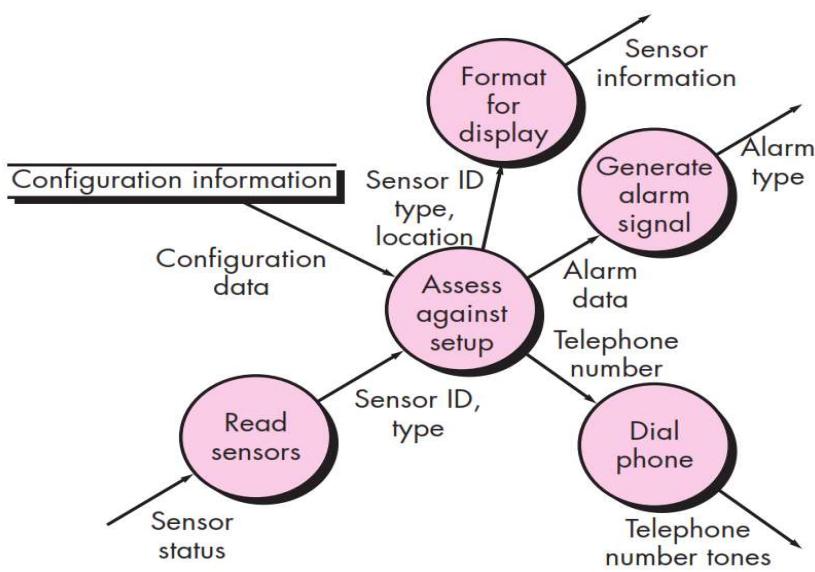
Architectural Design



Level 1 DFD for the SafeHome security function

29

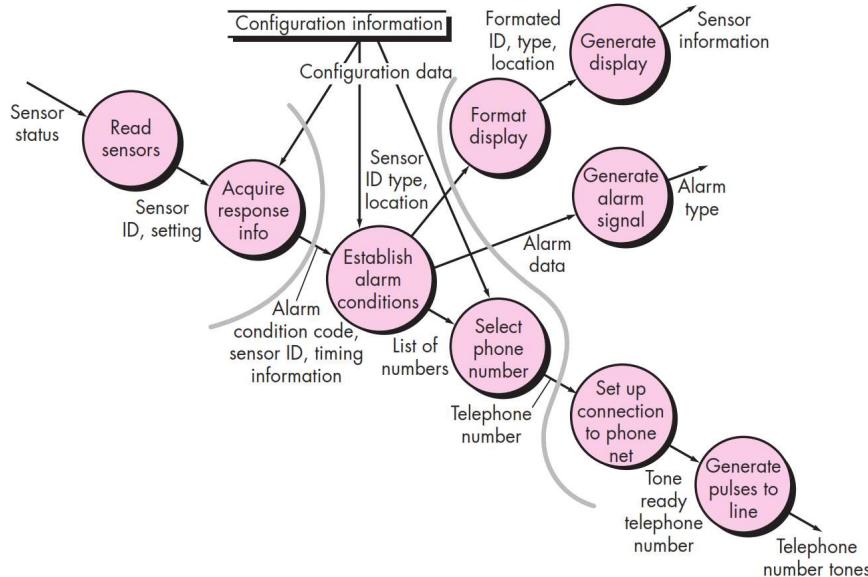
Architectural Design



Level 2 DFD that refines the monitor sensors transform

30

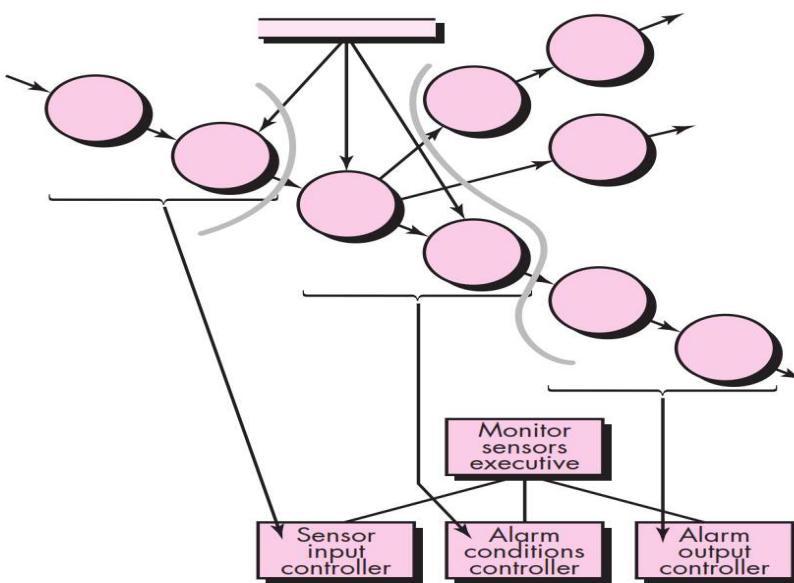
Architectural Design



Level 3 DFD for monitor sensors with flow boundaries

31

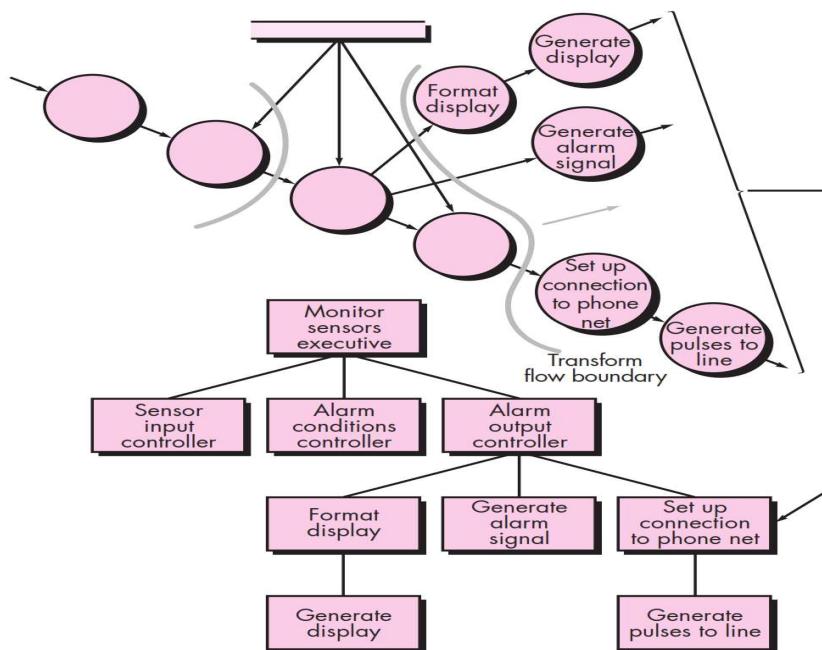
Architectural Design



First-level factoring for monitor sensors

32

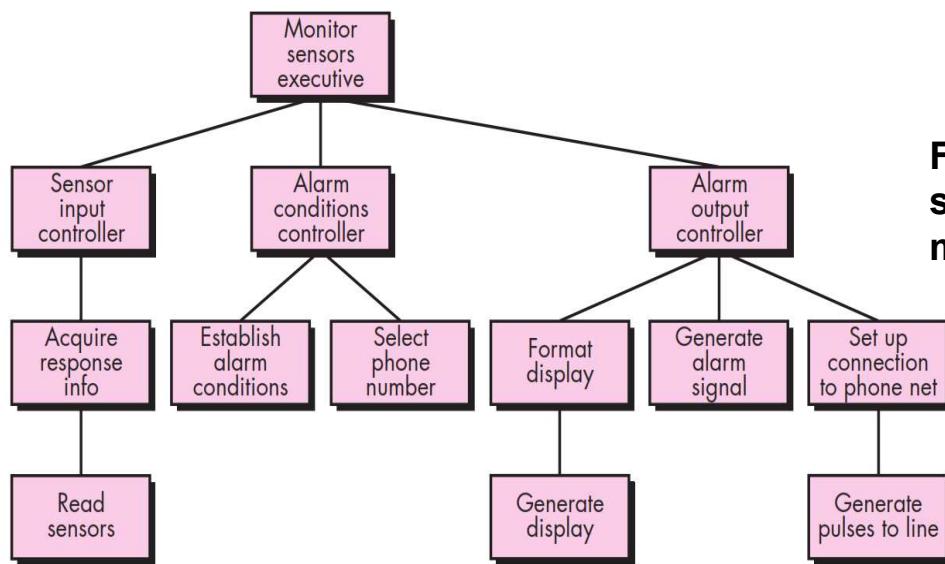
Architectural Design



Second-level
factoring for
monitor sensors

33

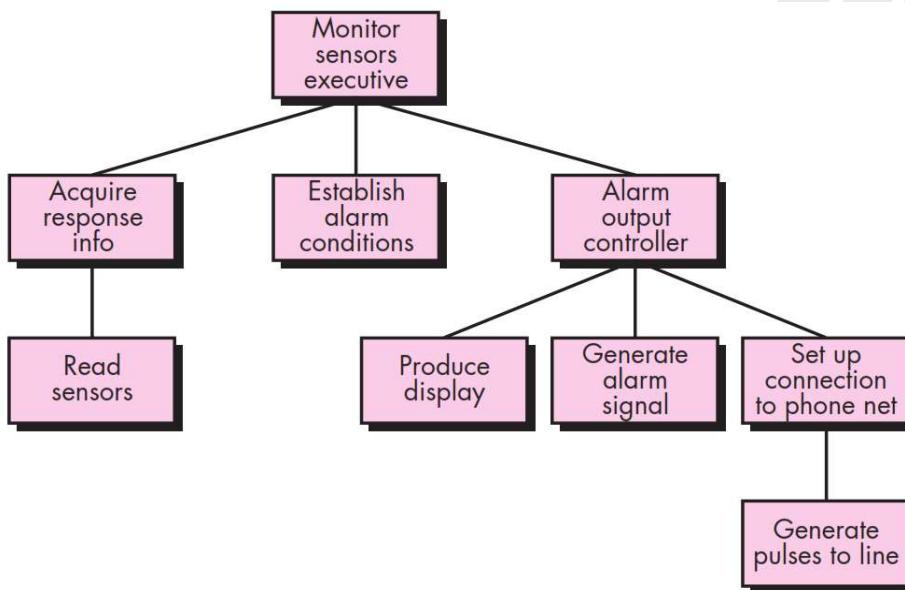
Architectural Design



First-iteration
structure for
monitor sensors

34

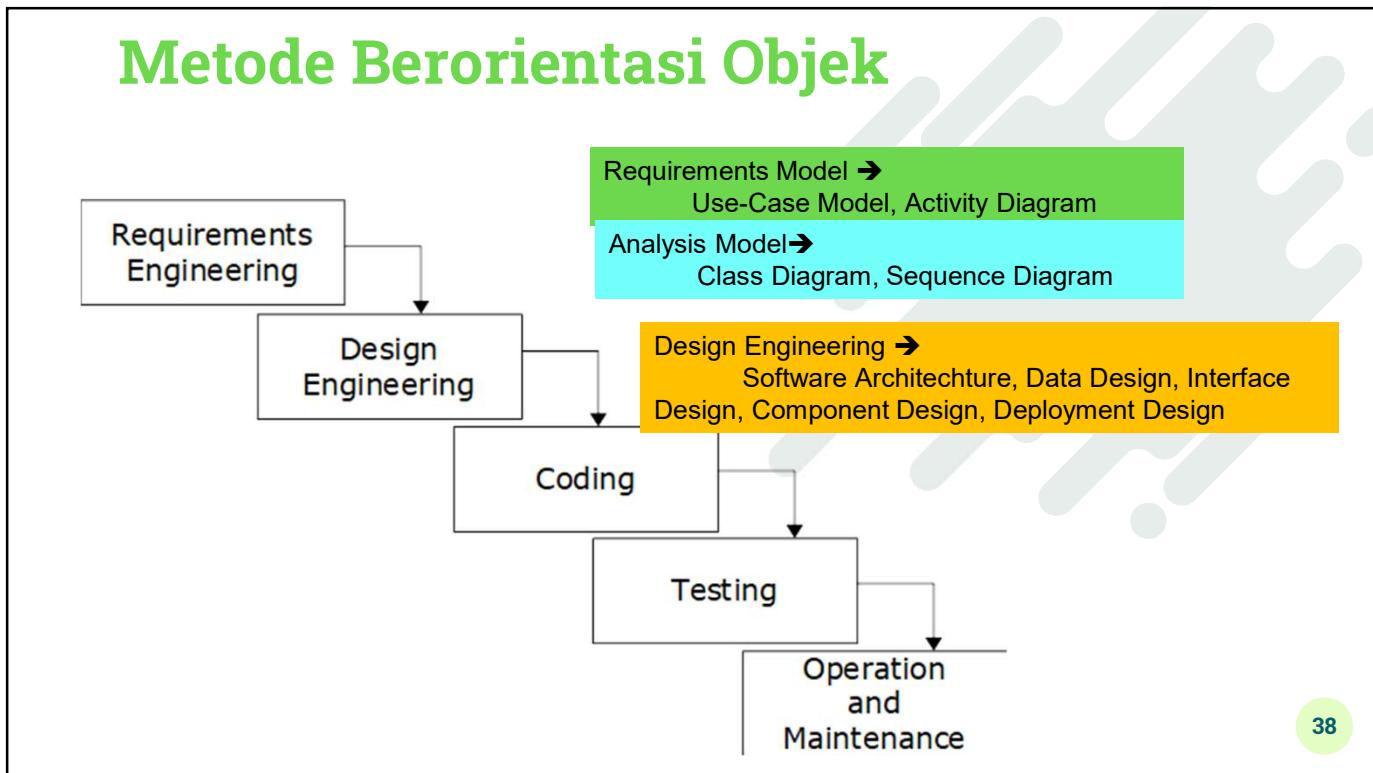
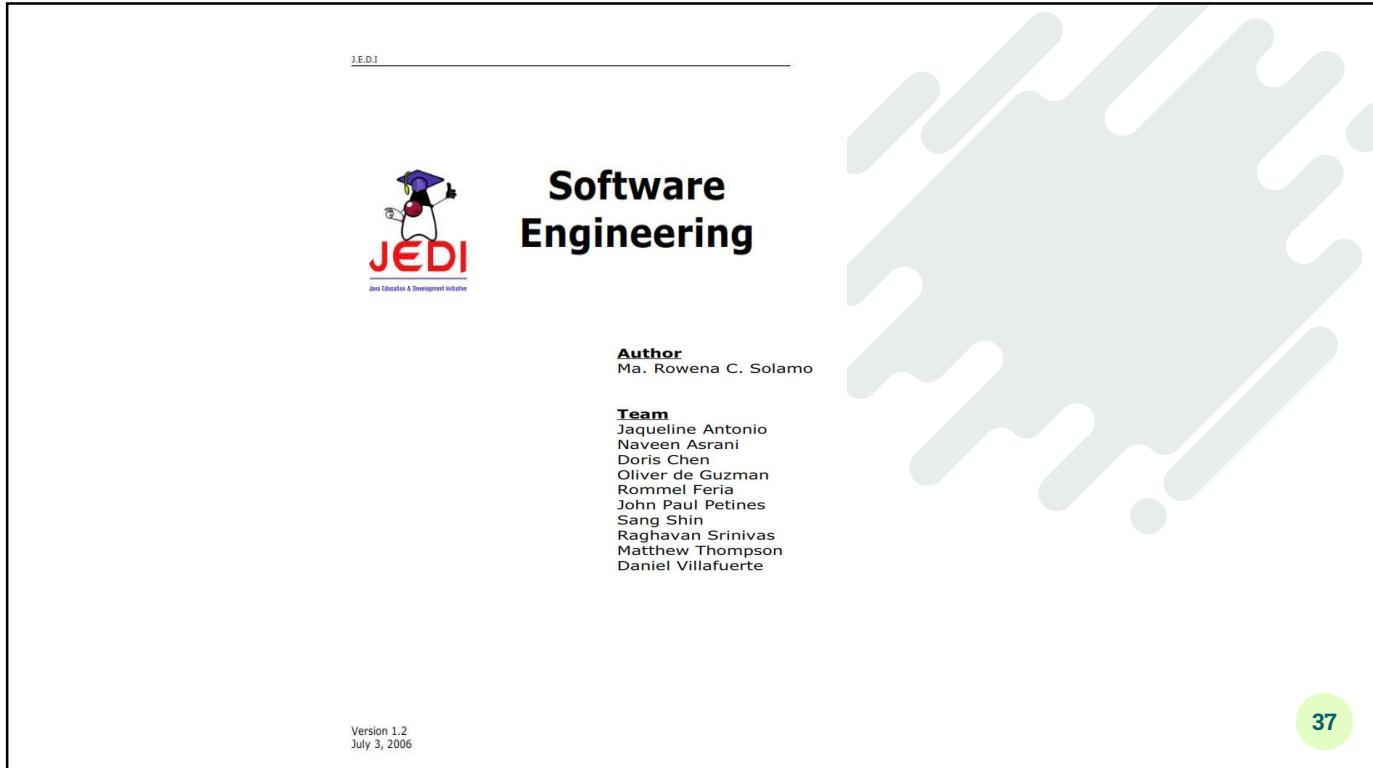
Architectural Design



Refined program
structure for
monitor sensors

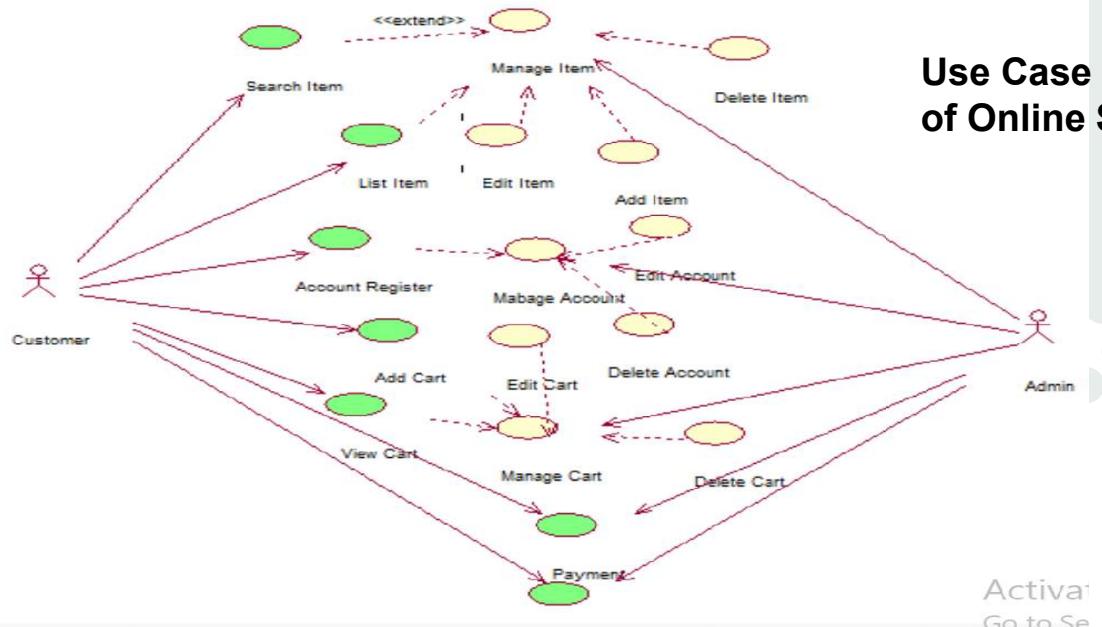
35

Metode Berorientasi Objek



Requirements Model

**Use Case Diagram
of Online Store**

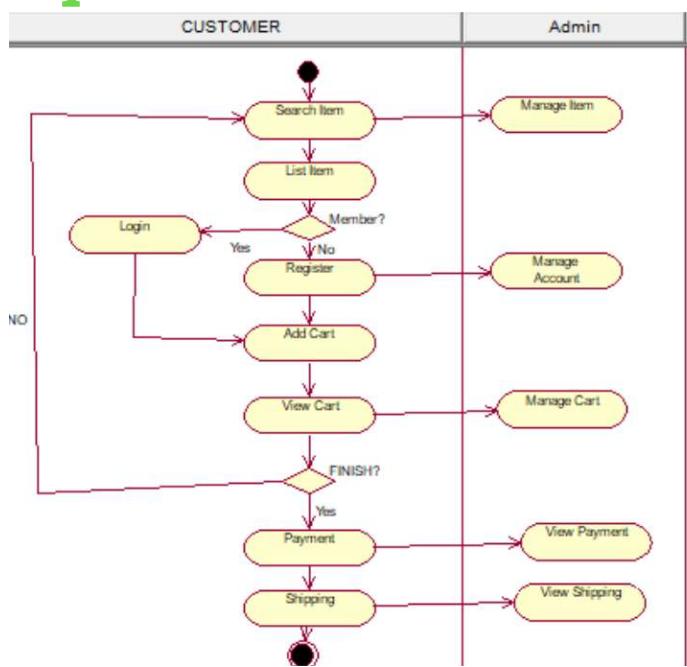


39

Activat
Go to Se

Requirements Model

**Activity Diagram
of Online Store**



40

Analysis Model

Analysis Classes

Three Perspectives

The boundary between the system and its actors.

The information the system uses.

The control logic of the system.

41

Analysis Model

Boundary Classes

It intermediates between the interface and something outside the system.

Types:

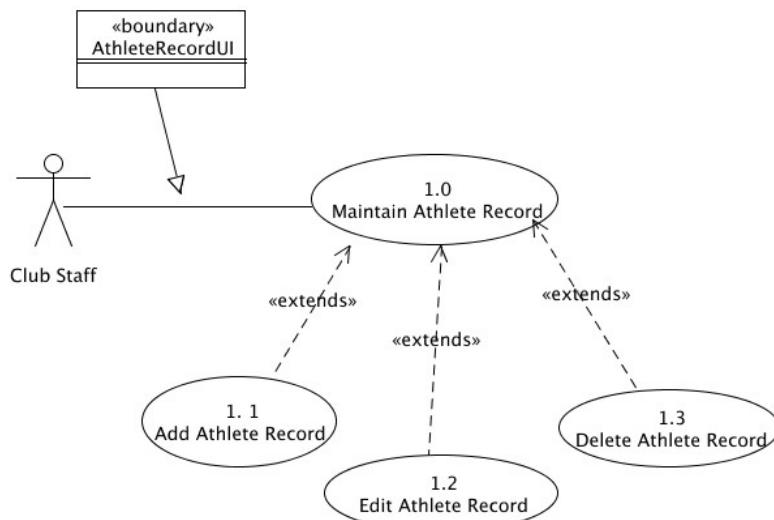
- User-interface
- System-interface
- Device-interface
- 1 boundary class per actor/use-case pair.

<<boundary>>
Class Name

42

Analysis Model

Boundary Classes

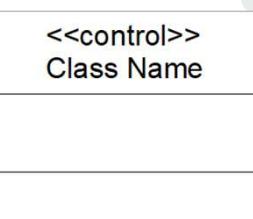


43

Analysis Model

Control Classes

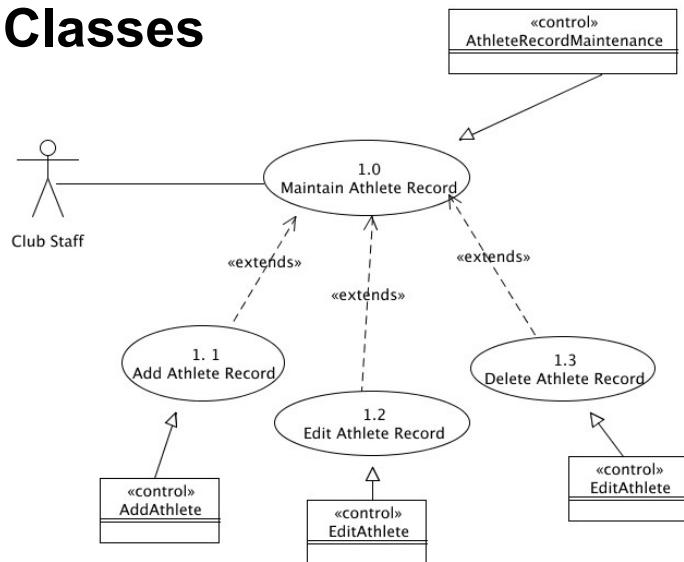
- It provides coordinating behavior of the system.
- It decouples boundary and entity classes from one another.
- 1 control class per use-case



44

Analysis Model

Control Classes

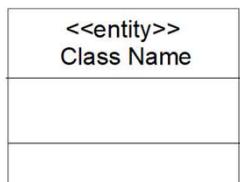


45

Analysis Model

Entity Classes

It represents the key concepts of the system being developed.
 It shows the logical data structure.
 It helps in understanding what the system is supposed to offer.



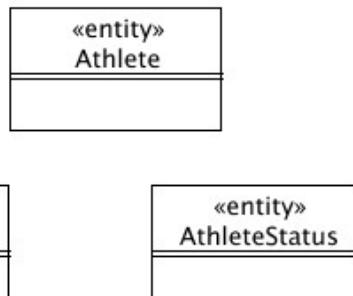
46

Analysis Model

Entity Classes

It can be found by:

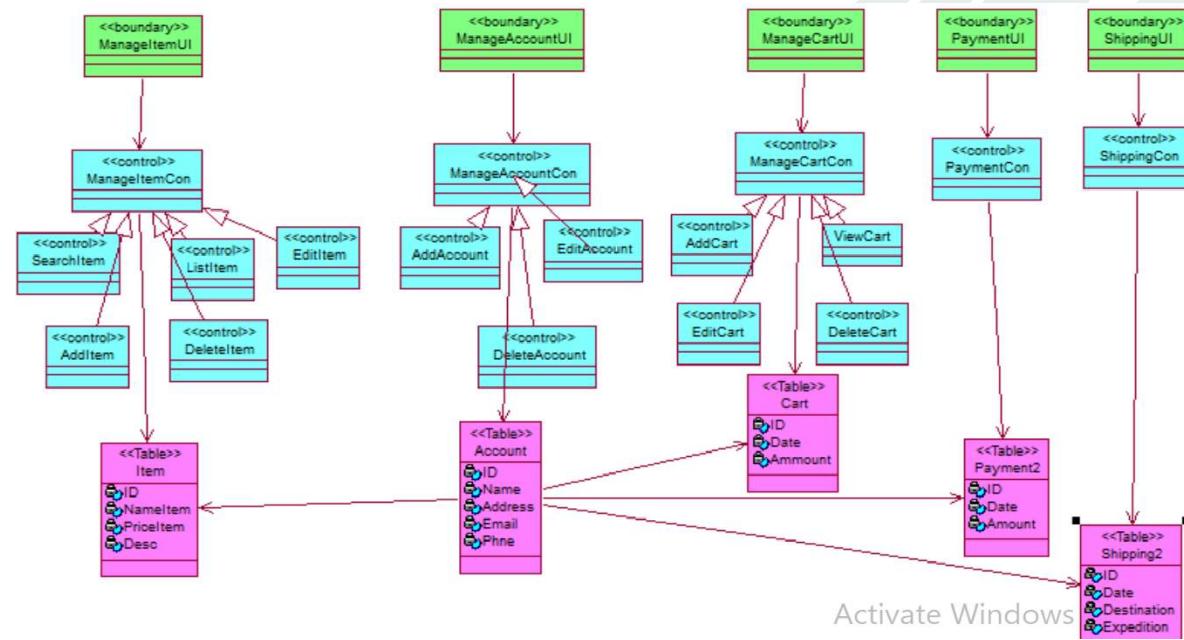
- Using use-case flow of events as input.
- Getting key abstractions from use-cases.
- Filtering nouns



47

Analysis Model

Class Diagram of Online Store

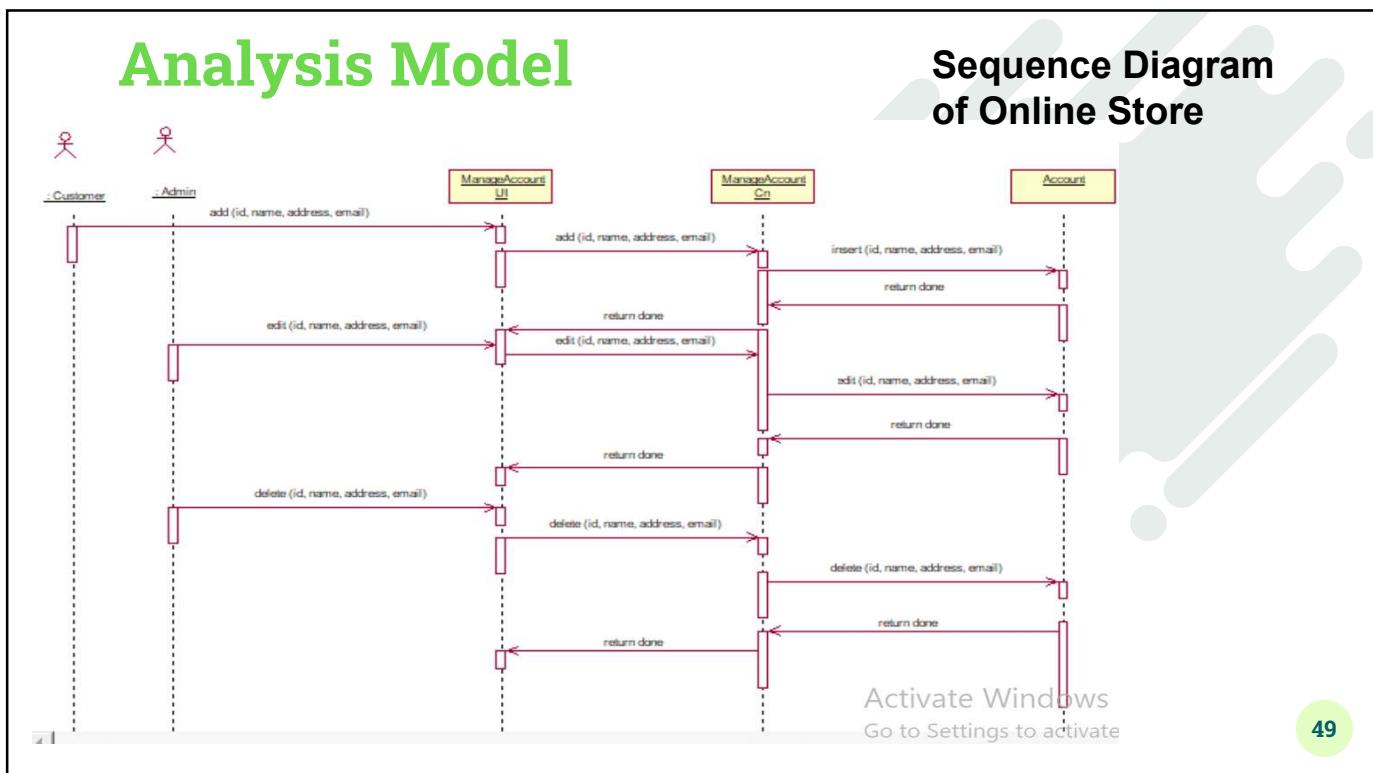


Activate Windows

48

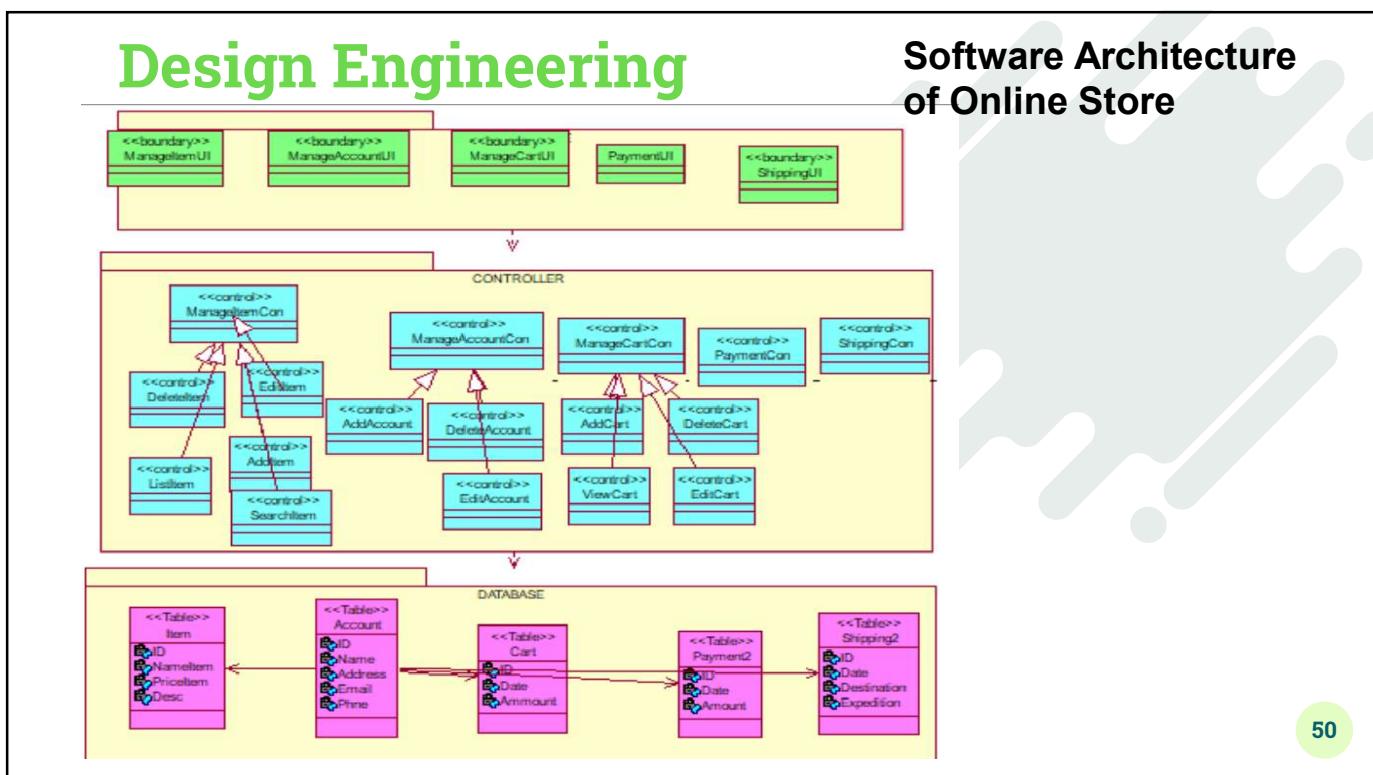
Analysis Model

Sequence Diagram of Online Store

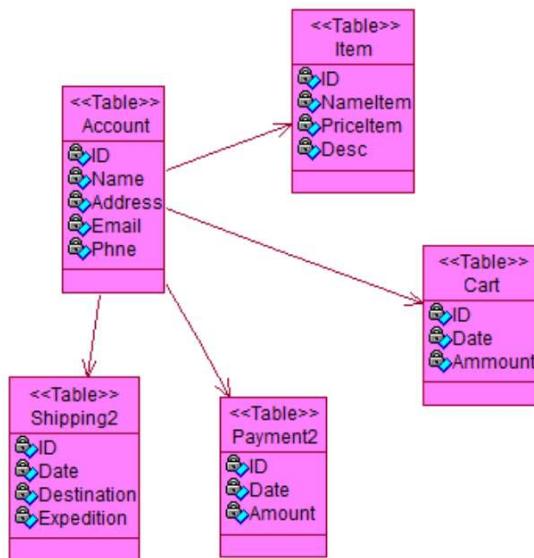


Design Engineering

Software Architecture of Online Store



Design Engineering



Data Design of
Online Store

51

Design Engineering

Athlete

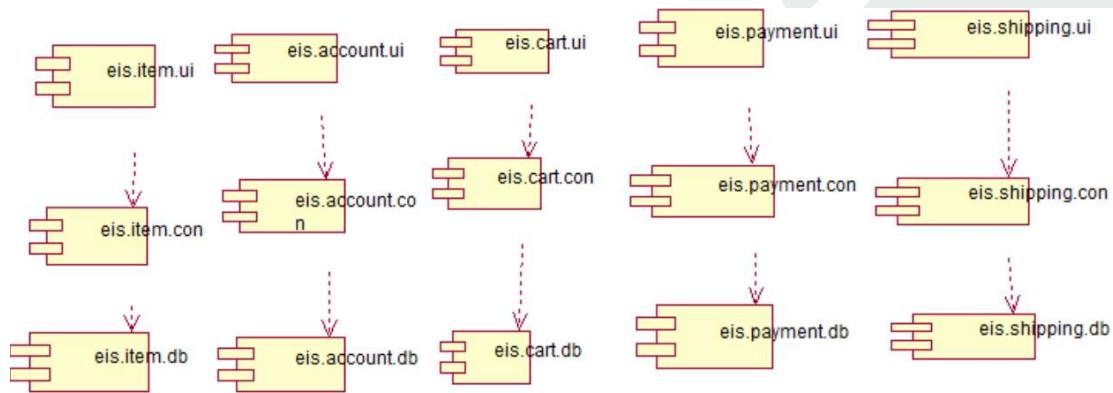
Membership No:	3456	Status:	Active	<input type="button" value="Find..."/>
Last Name:	<input type="text" value="De la Cruz"/>			<input type="button" value="Save"/>
First Name:	<input type="text" value="Johnny"/>			<input type="button" value="Delete"/>
Middle Initial:	<input type="text" value="A."/>			<input type="button" value="Cancel"/>
Address:	<input type="text" value="Lot 24 Block 18, St. Andrewsfield Quezon City"/>		Postal Code:	<input type="text" value="2619"/>
Date of Birth:	<input type="text" value="24/9/1998"/>			
Gender:	<input checked="" type="radio"/> Male	<input type="radio"/> Female		
Guardian: Last Name: <input type="text" value="De la Cruz"/> First Name: <input type="text" value="Johnny"/> Middle Initial: <input type="text" value="A."/> Address: <input type="text" value="Lot 24 Block 18, St. Andrewsfield Quezon City"/> Postal Code: <input type="text" value="2619"/> Postal Code: <input type="text" value="555-9895"/>				

Interface Design of
Online Store

52

Design Engineering

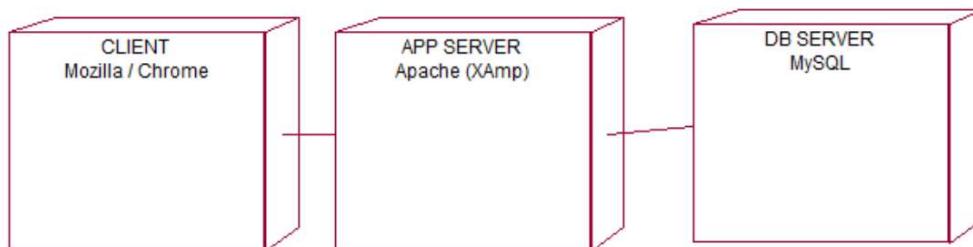
**Component Diagram
of Online Store**



53

Design Engineering

**Deployment Diagram
of Online Store**



54



Terimakasih

55

Dr. Andi W.R. Emanuel, BSEE, MSSE

PENJAMINAN MUTU PERANGKAT LUNAK



The image shows a flyer for a visiting lecturer event. At the top left is the logo of Universitas Diponegoro (UNDIP) and its faculty. To the right, there is text for 'Departemen Ilmu Komputer / Informatika', 'Fakultas Sains dan Matematika', and 'Universitas Diponegoro'. On the far right are logos for 'Kampus Mero INDONESIA JAYA' and 'G2O DEPARTMENT OF COMPUTER SCIENCE'. The main title 'VISITING LECTURER DALAM NEGERI' is centered at the top. Below it is the subtitle 'Software Process and Quality'. Two photographs of the visiting lecturers are shown in teal-bordered boxes: Prof. Dr. Ir. Wiranto Herry Utomo, M.Kom (left) and Dr. Andi Wahju Rahardjo, BSEE., MSSE (right). Their names and titles are listed below their respective photos. A teal box at the bottom contains the date 'Kamis, 12 Mei 2022', the time 'Pukul 08.00 - 12.30 WIB', and a Zoom link: 'Link <https://bit.ly/VLSPQIF22>'. It also lists the Meeting ID (234 573 4739) and Passcode (IF2022). At the bottom of the flyer are links for the department's website (if.fsm.undip.ac.id), email (if@live.undip.ac.id), YouTube channel (Informatika Undip), and Instagram account (@if.undip).

Departemen Ilmu Komputer / Informatika
Fakultas Sains dan Matematika
Universitas Diponegoro

Kampus Mero INDONESIA JAYA

G2O DEPARTMENT OF COMPUTER SCIENCE

VISITING LECTURER DALAM NEGERI

Software Process and Quality



Prof. Dr. Ir. Wiranto Herry Utomo, M.Kom
Dosen President University



Dr. Andi Wahju Rahardjo, BSEE., MSSE
Wakil Dekan III Fak. Teknologi Industri Univ. Atma Jaya Yogyakarta

 Kamis,
12 Mei 2022
Pukul 08.00 - 12.30 WIB

 zoom
- Link <https://bit.ly/VLSPQIF22>
- Meeting ID : 234 573 4739
- Passcode : IF2022

if.fsm.undip.ac.id | if@live.undip.ac.id | [Informatika Undip](#) | [if.undip](#)



Apa itu Penjaminan Mutu Perangkat Lunak?

Apa Itu Mutu / Kualitas?

Mutu / Kualitas – produk yang dikembangkan memenuhi spesifikasinya

Permasalahan:

- Organisasi pengembang memiliki kebutuhan melebihi spesifikasi konsumen (penambahan biaya dari pengembangan produk)
- Beberapa karakteristik tidak dapat dispesifikasikan dalam terminologi yang tidak membingungkan (contoh: kemampuan pemeliharaan / *maintainability*)
- Meskipun sebuah produk memenuhi spesifikasinya, pengguna mungkin tidak menganggapnya sebagai produk berkualitas (karena pengguna mungkin tidak terlibat dalam pengembangan kebutuhan)

Padanan dari Bermutu / Berkualitas

- Luar Biasa
- Superior
- Berkelas
- Berharga
- Bernilai
- Bersinar

Lawan Kata: Inferior

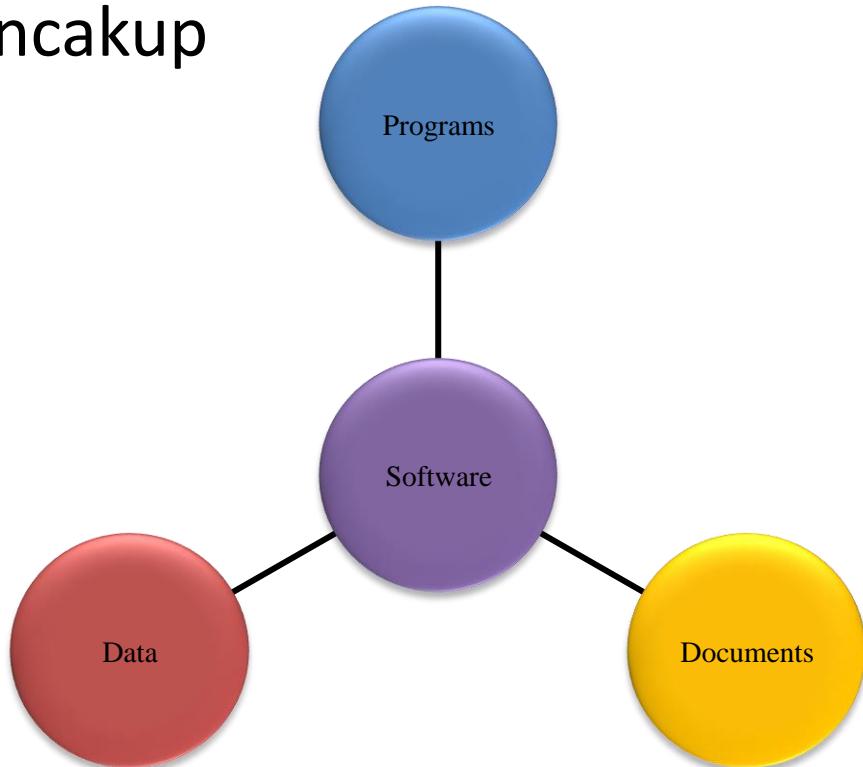


Apa itu Perangkat Lunak dan Bagaimana Membangunnya?

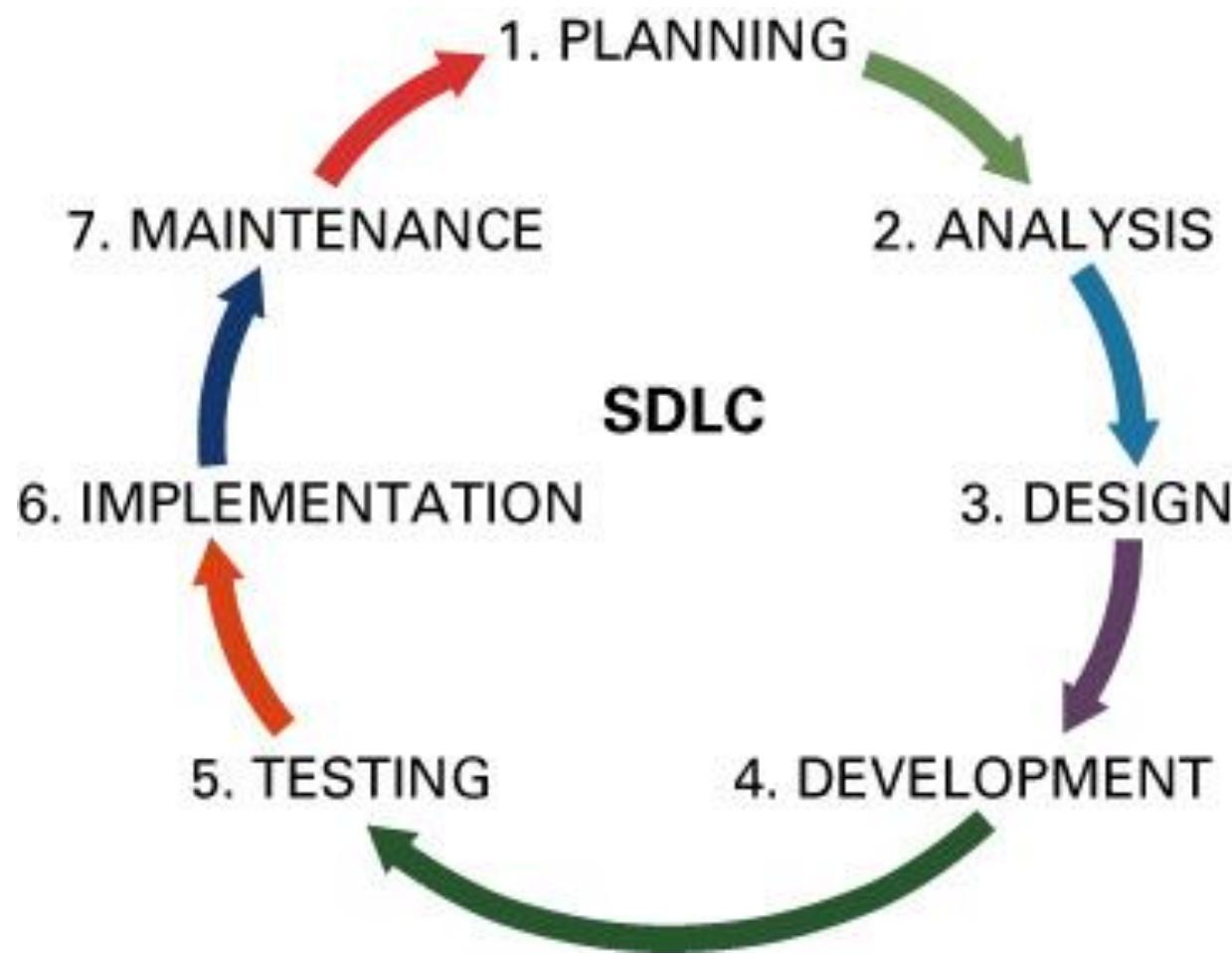


Apa itu Perangkat Lunak / Software?

- **Perangkat Lunak / Software** - produk yang dirancang dan dibangun oleh *software engineer*.
- Komponennya mencakup
 - Program
 - Data
 - Dokumen

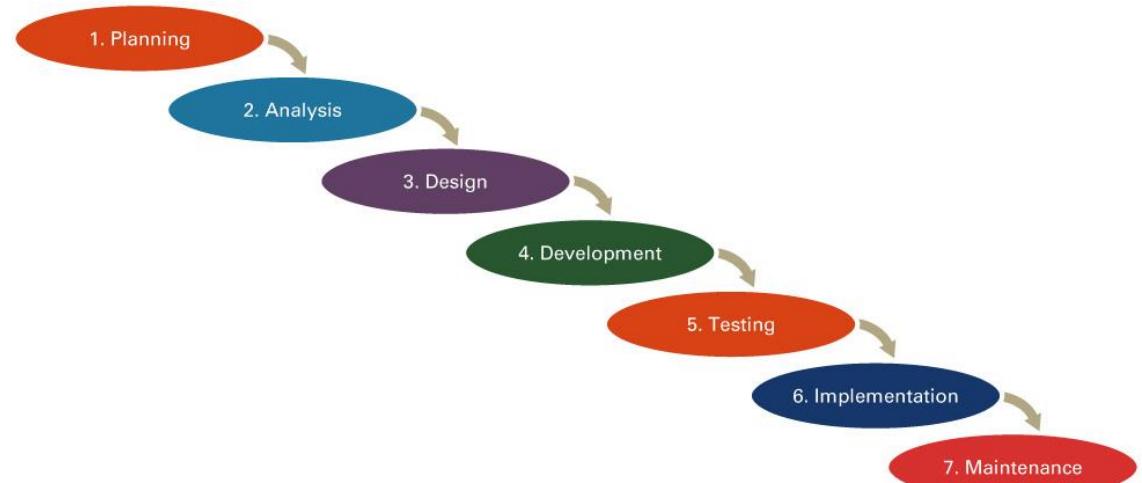


Siklus Hidup Pengembangan Sistem (SDLC)



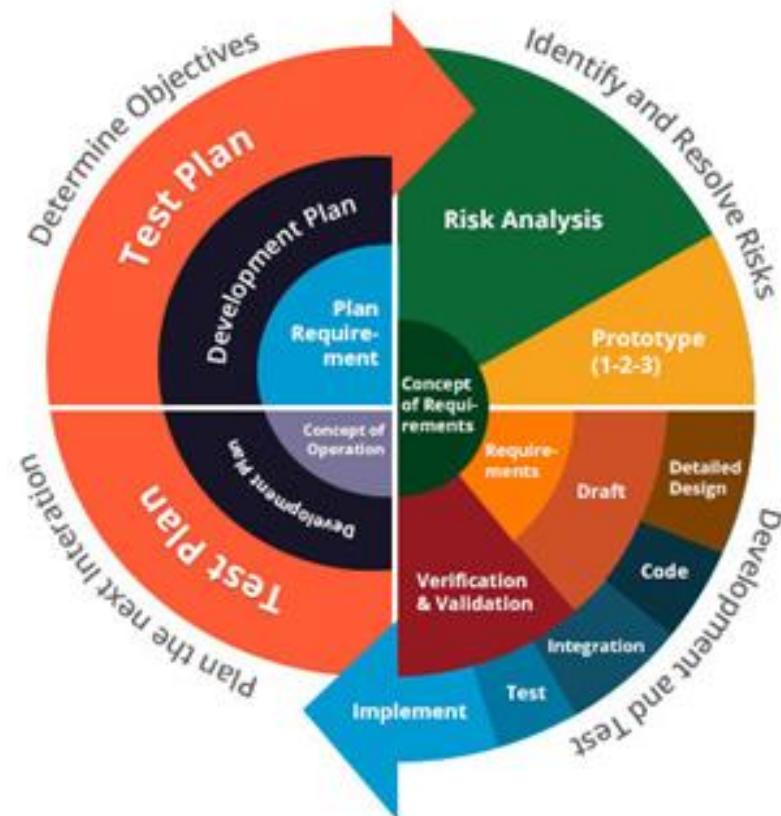
Waterfall

- **Waterfall** – proses sekuensial berbasis aktivitas mulai dari perencanaan sampai implementasi



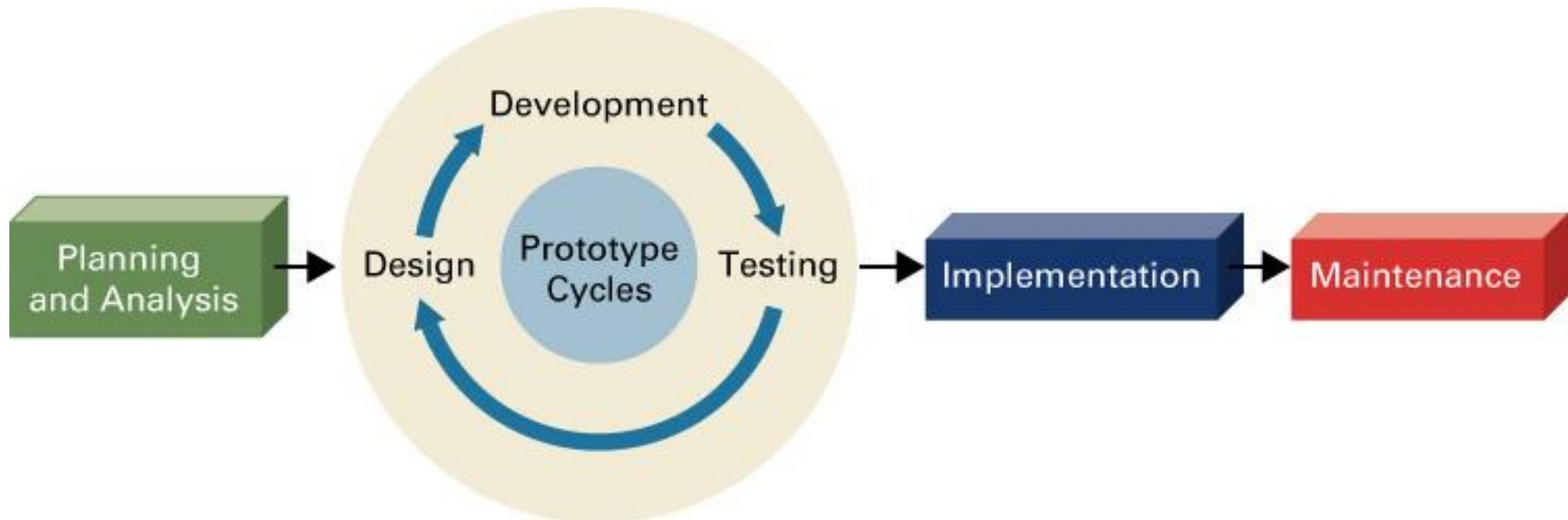
Spiral

- **Spiral** - evolusi yang menggunakan metode iterasi natural yang dimiliki oleh model prototyping dan digabungkan dengan aspek sistematis yang dikembangkan dengan model waterfall



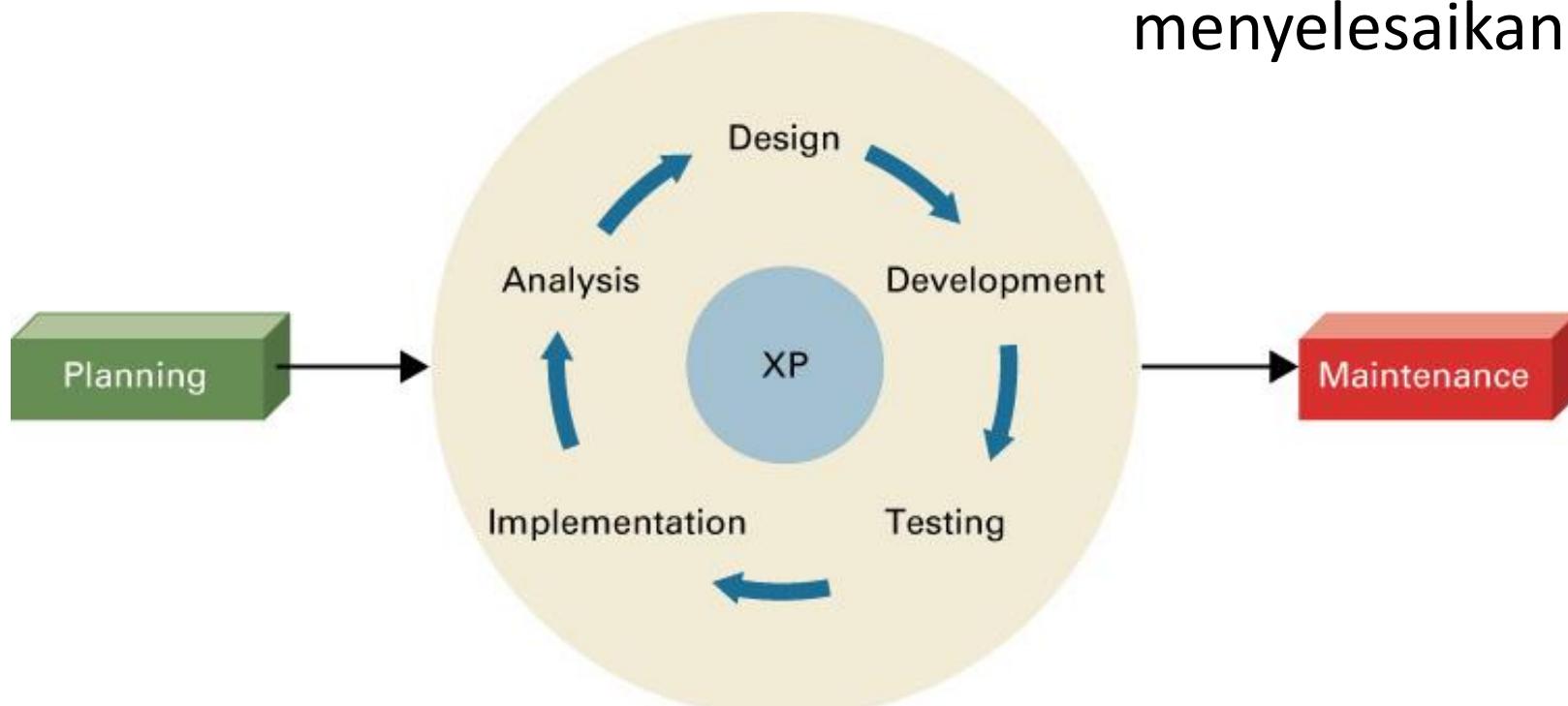
Rapid Application Development (RAD)

Rapid Application Development / Rapid Prototyping – menekankan peran pengguna secara ekstensif dalam konstruksi yang cepat dan evolusioner dari prototype yang bekerja



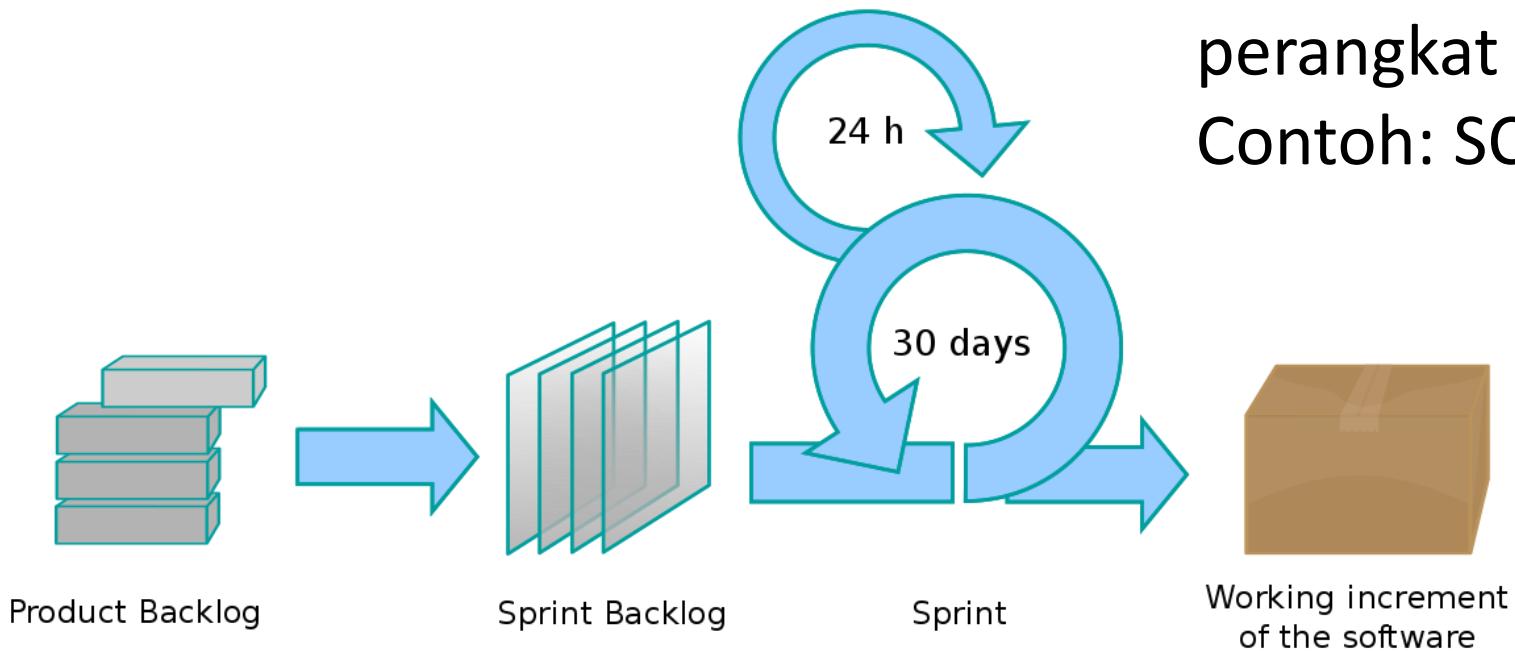
Extreme Programming (XP)

- *Extreme Programming (XP)* – memecah proyek ke dalam fase – fase kecil dan pengembangan tidak dapat maju sebelum menyelesaikan fase tsb

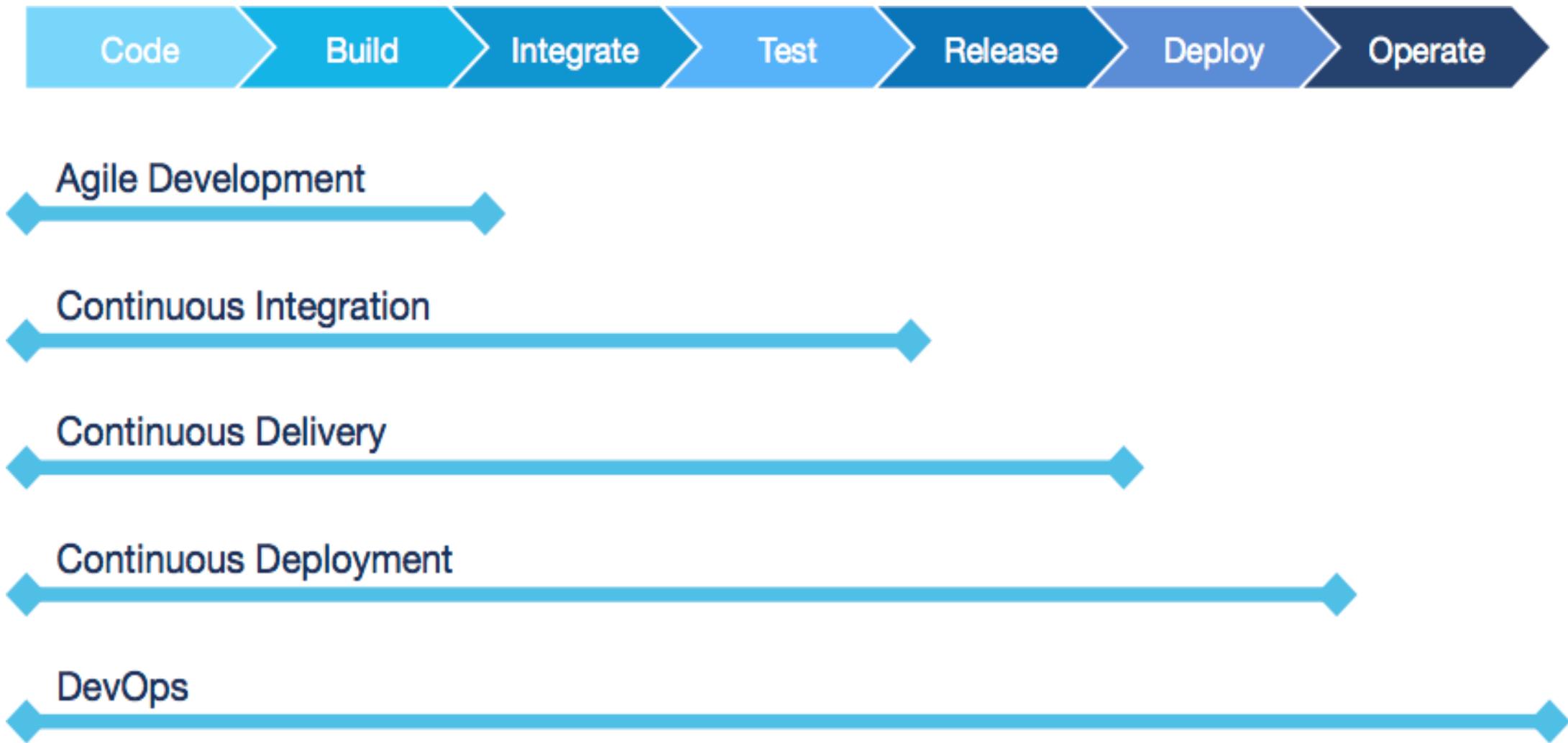


Agile Methodology

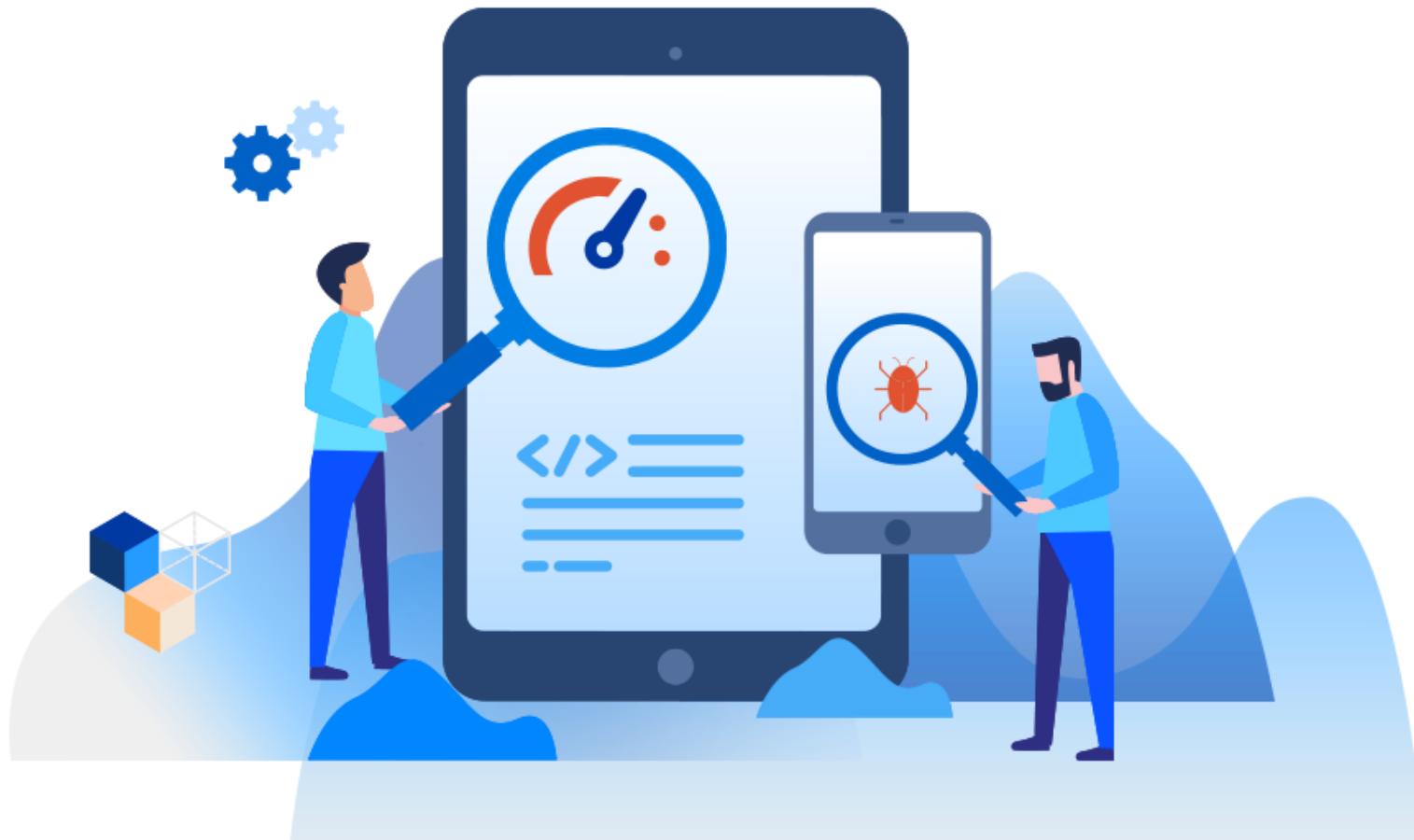
- ***Agile methodology*** – salah satu bentuk XP, dengan tujuan untuk kepuasan konsumen dengan delivery yang awal dan berkelanjutan dari komponen perangkat lunak yang berguna. Contoh: SCRUM



CI/CD/CD/DevOps



Perangkat Lunak, Mutu, dan Cacat



6 Faktor Kunci Perangkat Lunak Bermutu?

Kecacatan/*defect*
yang rendah
(mendekati nol)

Kehandalan tinggi

Kepuasan pemakai
yang tinggi

Memiliki struktur
yang meminimalisir
“perbaikan yang
buruk”

Dukungan
konsumen yang
efektif

Perbaikan yang
sangat cepat
apabila ditemukan
cacat

Kategori Cacat Perangkat Lunak

Kesalahan karena komisi

- Fitur yang diminta dan yang dibuat tidak sama

Kesalahan karena kelalaian

- Ada fitur yang lupa dibuat

Kesalahan karena kejelasan dan ambiguitas

- Kebutuhan tidak jelas, penafsiran salah/berbeda

Kesalahan karena kecepatan dan kapasitas

- Fitur sudah benar namun lambat/kapasitas rendah

Berbagai Sumber Cacat Perangkat Lunak

Kesalahan dalam
Kebutuhan /
Requirement

Kesalahan dalam
Desain

Kesalahan dalam
Kode Sumber

Kesalahan dalam
Dokumentasi
Pemakai

Kesalahan karena
“Perbaikan yang
Buruk”

Kesalahan dalam
Data dan Tabel

Kesalahan dalam
Kasus Pengujian



**Membuat Perangkat Lunak yang
Bermutu Membutuhkan
Manajemen Mutu yang Baik**

Apa itu Manajemen Mutu?

Manajemen Mutu – memastikan bahwa tingkatan yang dibutuhkan dari kualitas produk dapat dicapai

- Definisi prosedur dan standar
- Aplikasi prosedur dan standar ke produk dan proses
- Cek ketaatan pada prosedur
- Koleksi dan analisa berbagai data mutu

Permasalahan:

- Aspek – aspek tidak terlihat (*intangible*) dari kualitas perangkat lunak tidak dapat distandardisasi, contohnya keanggunan (*elegance*) dan kemudahan dibaca (*readibility*)

Elemen dari Manajemen Mutu PL?



Penjaminan Mutu Perangkat Lunak (PMPL/SQA) – pembangunan jaringan dari berbagai prosedur dan standar organisasi menuju perangkat lunak berkualitas tinggi



Rencana Mutu Perangkat Lunak (RMPL/SQP) – pemilihan dari berbagai prosedur dan standar yang cocok dari kerangka kerja ini dan adaptasinya untuk proyek perangkat lunak spesifik tertentu



Kontrol Mutu Perangkat Lunak (KMPL/SQC) – definisi dan pelaksanaan dari berbagai proses yang memastikan bahwa prosedur dan standar mutu proyek ditaati oleh tim pengembangan perangkat lunak



Metriks Mutu Perangkat Lunak (MMPL/SQM) – mengumpulkan dan menganalisa data mutu untuk memprediksi dan mengontrol mutu dari produk perangkat lunak yang dikembangkan.



Standar Pengembangan Perangkat Lunak

Mengapa Standar Penting?

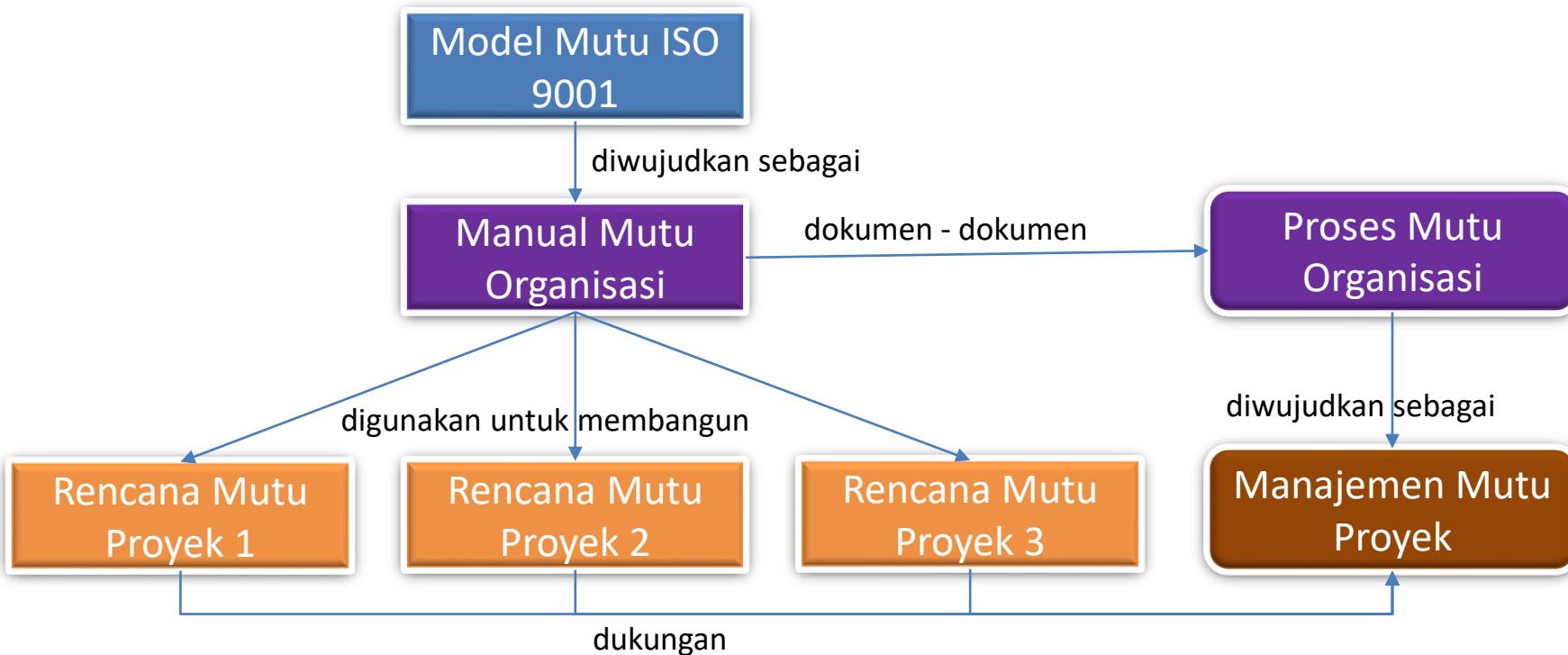
Standar menyediakan enkapsulasi dari praktek yang terbaik, atau paling tidak praktek yang paling sesuai

Standar menyediakan sebuah kerangka kerja untuk kemungkinan implementasi dari penjaminan mutu

Standar membantu kelanjutan dari pekerjaan ketika dilaksanakan oleh orang yang berbeda sepanjang siklus hidup produk perangkat lunak

Standar seharusnya tidak dihindari. Jika terasa terlalu berat untuk tugas yang dijalankan, maka standar tersebut bisa disesuaikan/dimodifikasi

Pendekatan Sederhana dari Standar Pengembangan PL

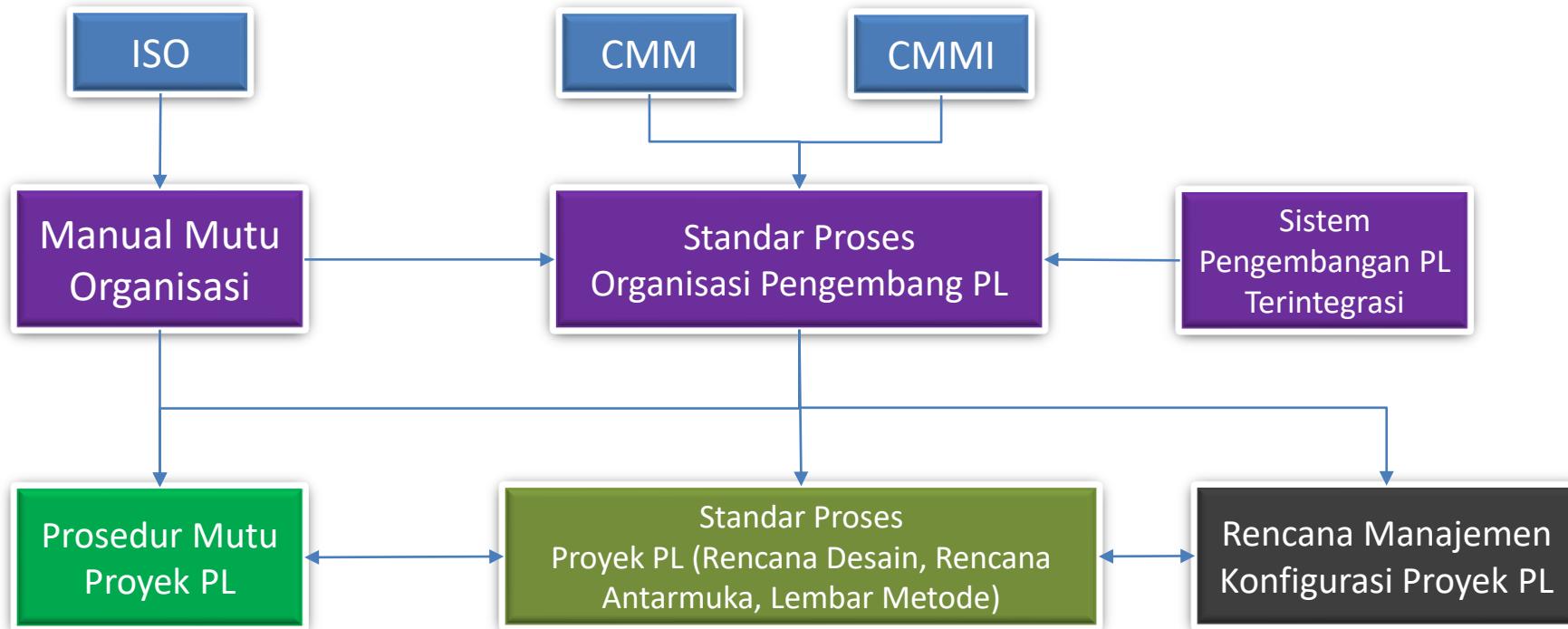


Di sebagian besar dari organisasi yang sudah matang:

- ISO bukan satu – satunya sumber dari Standar Pengembangan Perangkat Lunak
- Standar proses dan standar produk dikembangkan sendiri – sendiri
- Standar produk tidak dikembangkan oleh bagian Penjaminan Mutu Perangkat Lunak

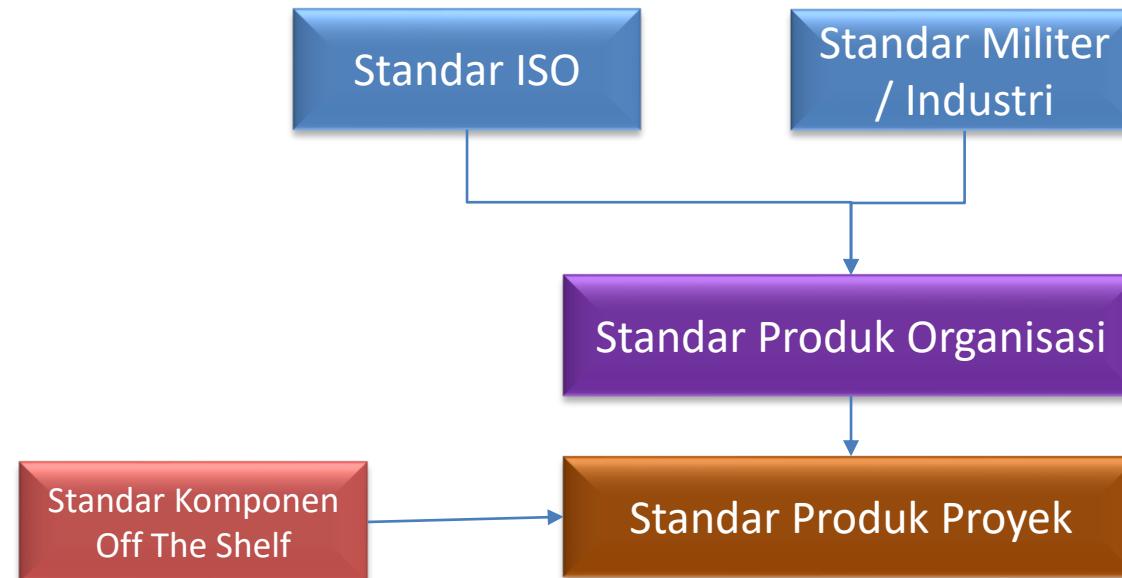
Standar Proses

Standar Proses – berbagai standar yang mendefinisikan proses yang harus diikuti selama pengembangan perangkat lunak



Standar Produk

Standar Produk – berbagai standar yang berlaku untuk produk yang sedang dikembangkan



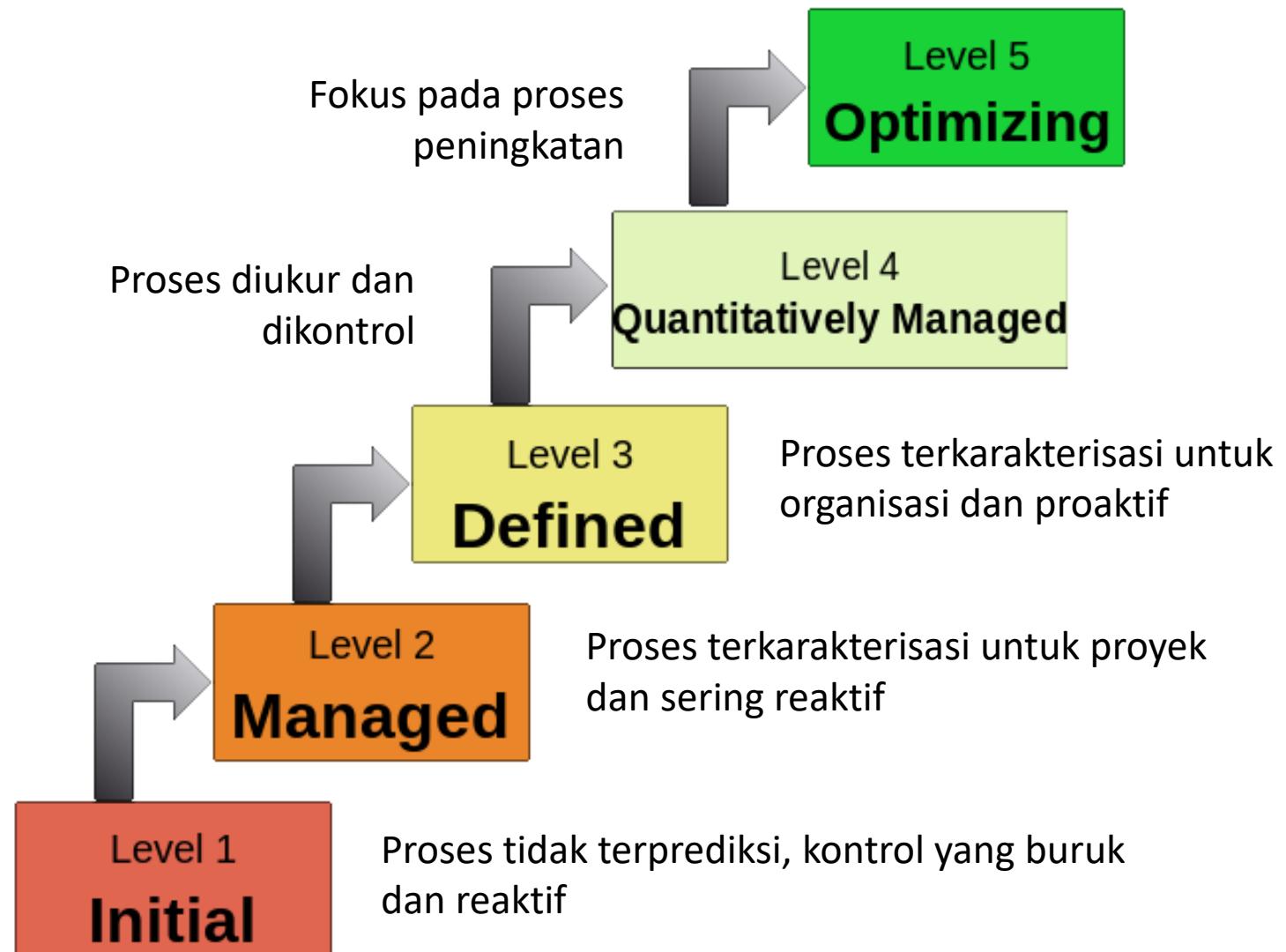


Model – Model Mutu

ISO 9001:2015



Karakteristik dan Model Kematangan Kemampuan Terintegrasi (CMMI)





Peningkatan Mutu – Six Sigma

qualitymag.com

Tim Penjaminan Mutu



Kontinuitas dan Independensi dari Tim PMPL

Tim Penjaminan Mutu Perangkat Lunak harus independen agar dapat mengambil pandangan obyektif dari proses dan melaporkan permasalahan secara langsung ke manajemen senior

Jika proses yang ditetapkan tidak tepat untuk jenis perangkat lunak yang sedang dikembangkan, maka dapat dimodifikasi

Standar harus diterapkan sekecil apapun tugasnya. Pembuatan purwarupa bukan berarti tanpa standar. Ini artinya standar yang dimodifikasi

Mutu itu **GRATIS**, jika merupakan **tanggung jawab semua orang!**

Rencana Mutu Perangkat Lunak



Rencana Mutu Perangkat Lunak (RMPL)

RMPL (*SQP*) tidak ditulis untuk pengembang perangkat lunak. Ini ditulis untuk Perekayasa Mutu Perangkat Lunak (*SQE*) sebagai panduan untuk Kontrol Mutu PL (KMPL) dan bagi konsumen untuk memonitor aktivitas pengembangan Faktor – faktor Mutu seharusnya tidak dikorbankan untuk mencapai efisiensi. Jangan mengambil pekerjaan jika proses mutunya tidak dapat dipertahankan

Aktivitas:

- **Memodifikasi** – memilih standar – standar organisasi yang sesuai untuk produk tertentu
- **Standardisasi** – menggunakan (atau memanggil) standar proses dan produk organisasi yang telah disetujui, jika standar baru diperlukan maka peningkatan mutu harus diinisialisasi
- **Elemen** – penerapan berbagai elemen RMPL (*SQP*) berdasarkan elemen model di ISO-9001

Kontrol Mutu Perangkat Lunak (KMPL)



Metode untuk Kontrol Mutu Perangkat Lunak

KMPL melibatkan pemantauan proses pengembangan perangkat lunak untuk memastikan bahwa berbagai prosedur dan standar telah ditaati

Aktivitas berikut ini yang melambangkan KPML:

- Review
- Inspeksi
- Pengujian
- Audit Mutu

Apa itu Review?



Review - sebuah proses atau pertemuan dimana sebuah produk kerja, atau sebuah set produk kerja, dipresentasikan kepada personil proyek, manajer, pengguna, atau pihak berkepentingan lainnya untuk mendapatkan komentar atau persetujuan



Tipe Review:

- Review kode
- Review desain
- Review kualifikasi formal
- Review kebutuhan
- Review kesiapan pengujian



Sumber: IEEE Std. 610.12-1990

Peran – Peran dalam Review

- Fasilitator
- Pembuat
- Perekam
- Reviewer
- Observer



Frekuensi dari Review

Pada awal / akhir dari fase kebutuhan

Pada awal / akhir dari fase desain

Pada awal / akhir dari fase koding

Pada awal / akhir dari fase pengujian

Persetujuan dari rencana pengujian

Inspeksi

- Inspeksi adalah sebuah review tim secara sangat lengkap dari sebuah produk kerja oleh rekan sejawat dari pembuat produk kerja tersebut
- Ukuran dari tim akan bervariasi tergantung dari karakteristik dari produk kerja yang diinspeksi; contohnya ukuran, tipe
- Penemu: Michael Fagan sejak 1971
- Sering disebut: Fagan Inspection



Produk Kerja

Spesifikasi
kebutuhan

Spesifikasi
desain

Kode Sumber

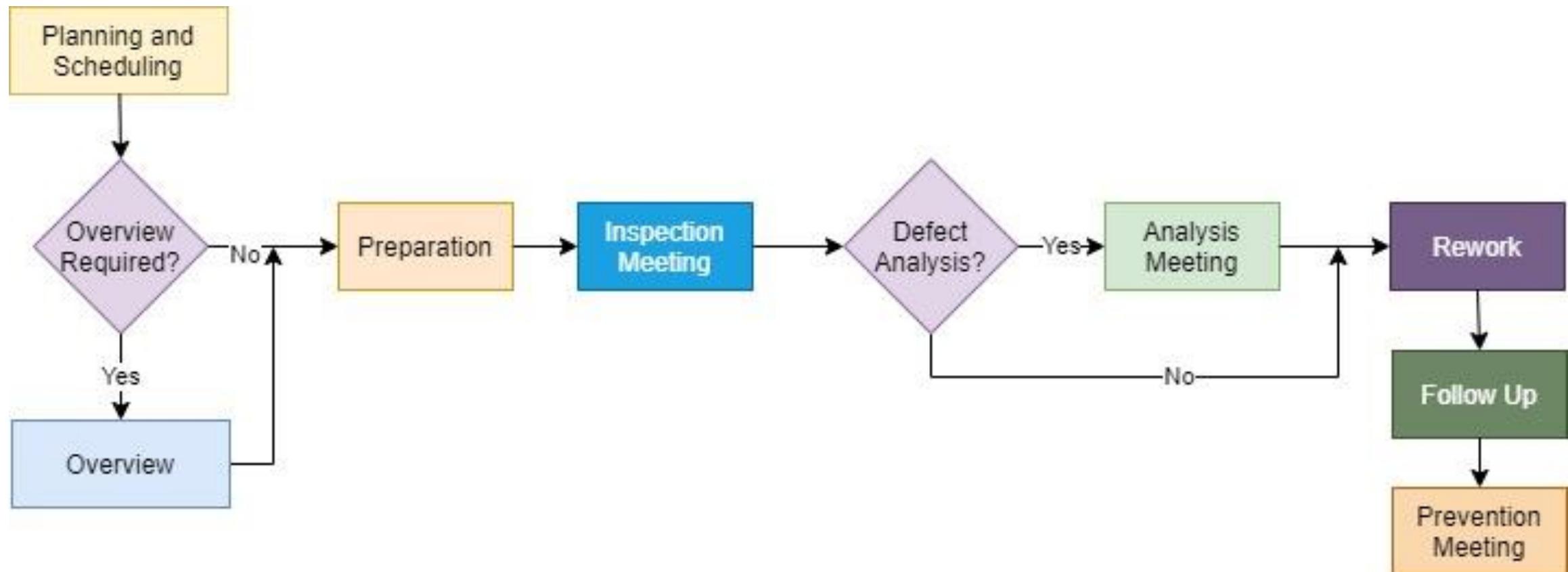
Dokumentasi
pengguna

Rencana

Kasus
pengujian

Semua
dokumen
lainnya

Aliran Proses Inspeksi

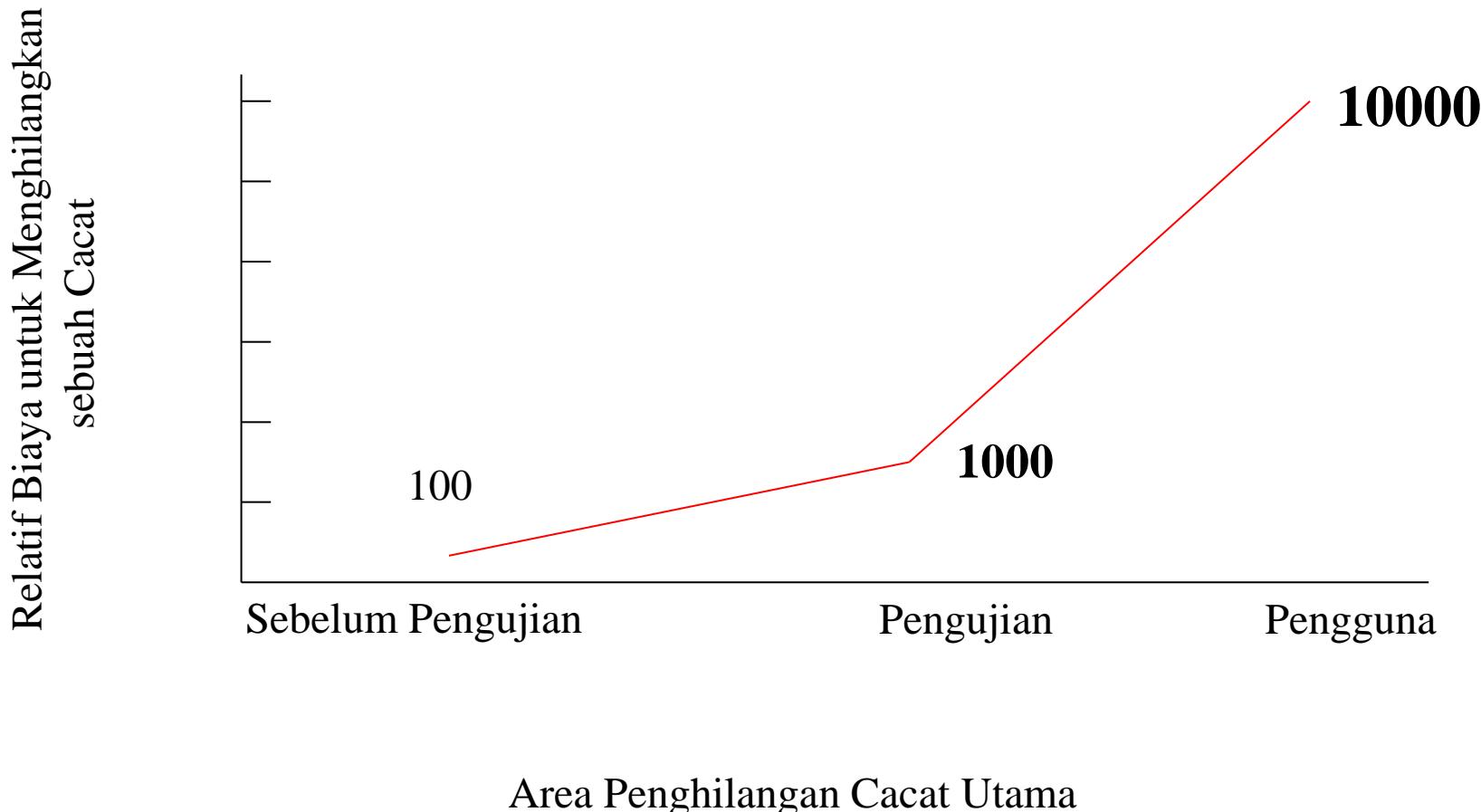


Peran dalam Inspeksi

- Moderator
- Pembuat
- Pembaca
- Perekam
- Inspektur

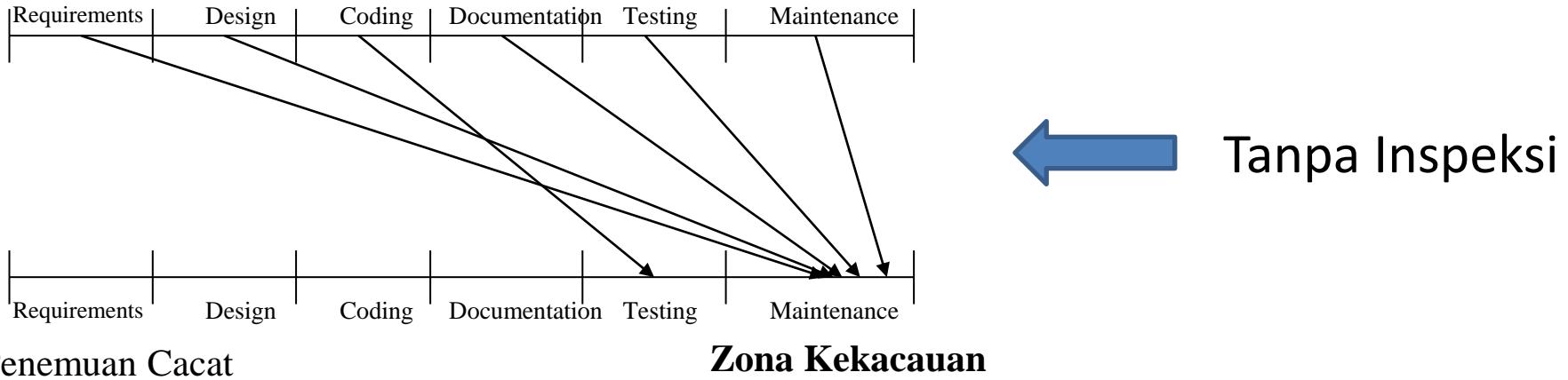


Manfaat Review dan Inspeksi: Hubungan Biaya dan Cacat



Manfaat Review dan Inspeksi: Menghindari Zona Kekacauan (*Chaos Zone*)

Asal Cacat

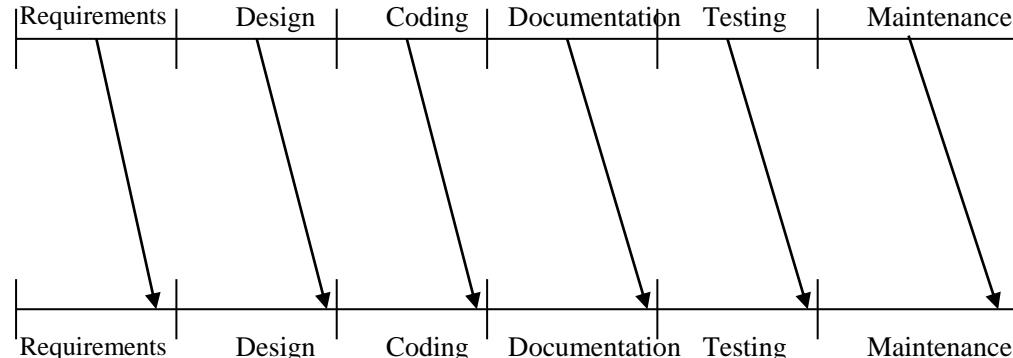


Zona Kekacauan

Asal Cacat

Dengan Inspeksi →

Penemuan Cacat



Pengujian

Engineering Dry-run – pengujian yang dijalankan oleh pembuat tanpa kehadiran Perekayasa Mutu PL/SQE. Pengujian ini meliputi Pengujian Unit dan Engineering Dry-run dari pengujian formal.

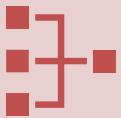
SQE Dry-run – pengujian oleh Perekayasa Mutu Perangkat Lunak/SQE.

TFR – pengujian yang dijalankan sebagai “RFR – run-for-record” dengan Perekayasa Mutu PL dan konsumen. Pengujian ini meliputi pengujian penerimaan (FAT dan SAT). Pengujian ini dijalankan untuk penyerahan produk akhir kepada konsumen.

Audit Mutu

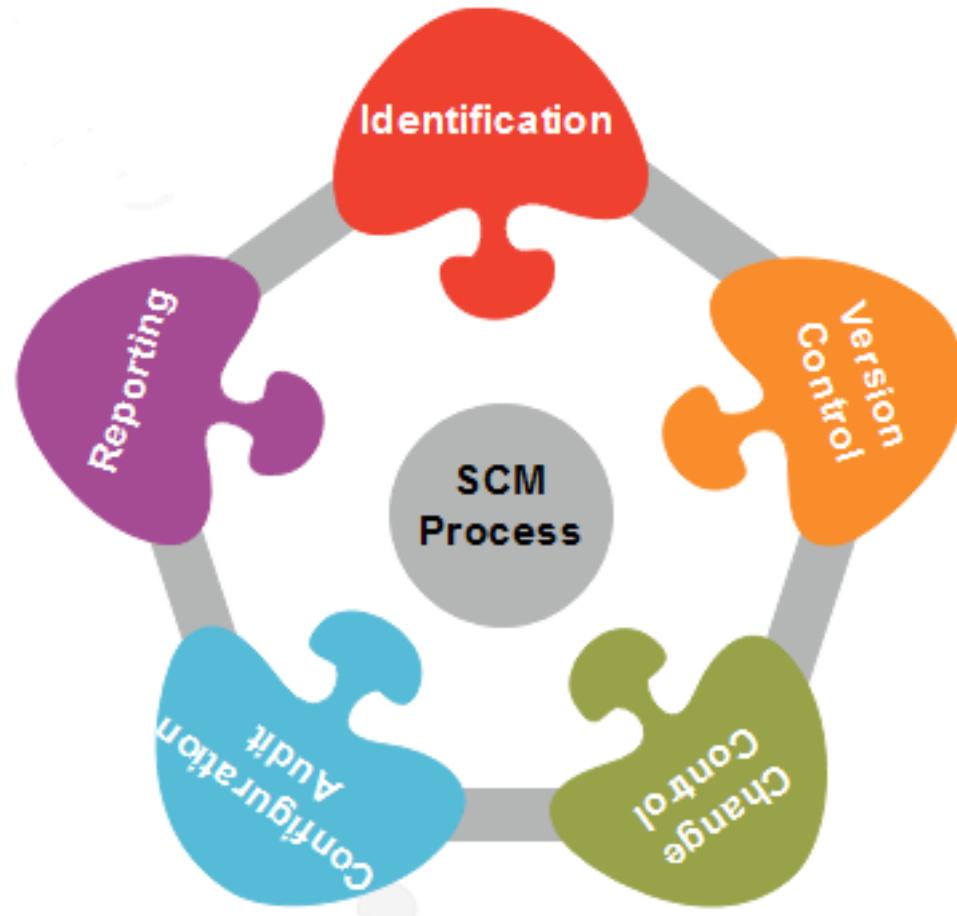


Audit oleh Perekayasa Mutu PL – audit yang dilaksanakan oleh Perekayasa Mutu PL untuk memverifikasi bahwa standar proses telah diikuti. Contoh audit ini adalah Kesesuaian SPPT (IPDS), Kontrol Konfigurasi, dan Manajemen Rekayasa Perangkat Lunak. Semua temuan dalam audit ini didokumentasikan dalam formulir QER. Hasil – hasil dari audit ini didistribusikan ke tingkatan selanjutnya dari manajemen (lebih tinggi dari tingkatan proyek).



Audit Independen – audit yang dilaksanakan oleh seorang generalis ISO atau entitas independen lainnya untuk memverifikasi bahwa standar proses telah diikuti. Audit ini biasanya dilaksanakan di tingkat sebuah divisi / fasilitas. Hasil – hasil dari audit ini didistribusikan ke manajemen tingkat lebih tinggi.

Manajemen Konfigurasi Perangkat Lunak



Manajemen Konfigurasi Perangkat Lunak (MKPL/SCM)

MKPL / SCM – berbagai aktivitas yang memastikan bahwa produk – produk perangkat lunak telah diidentifikasi seperlunya dan transisinya terlacak. Dalam banyak organisasi yang sudah matang MKPL bukan tanggung jawab dari bagian PMPL.

Aktivitas:

- **Identifikasi Baseline** – identifikasi kondisi awal dari produk
- **Identifikasi Perubahan** – identifikasi perubahan – perubahan yang dibuat terhadap baseline
- **Kontrol Perubahan** – dokumentasi dari perubahan – perubahan lewat history revisi, ringkasan perubahan atau menggunakan perangkat pengembangan otomatis (*ClearCase* atau *Apex*)
- **Akunting Status** – melaporkan perubahan – perubahan ke pihak lain dan memonitor kelengkapan dari arsip proyek
- **Preservasi** – menjaga dari produk – produk perangkat lunak

Metriks Mutu Perangkat Lunak



Koleksi Metriks

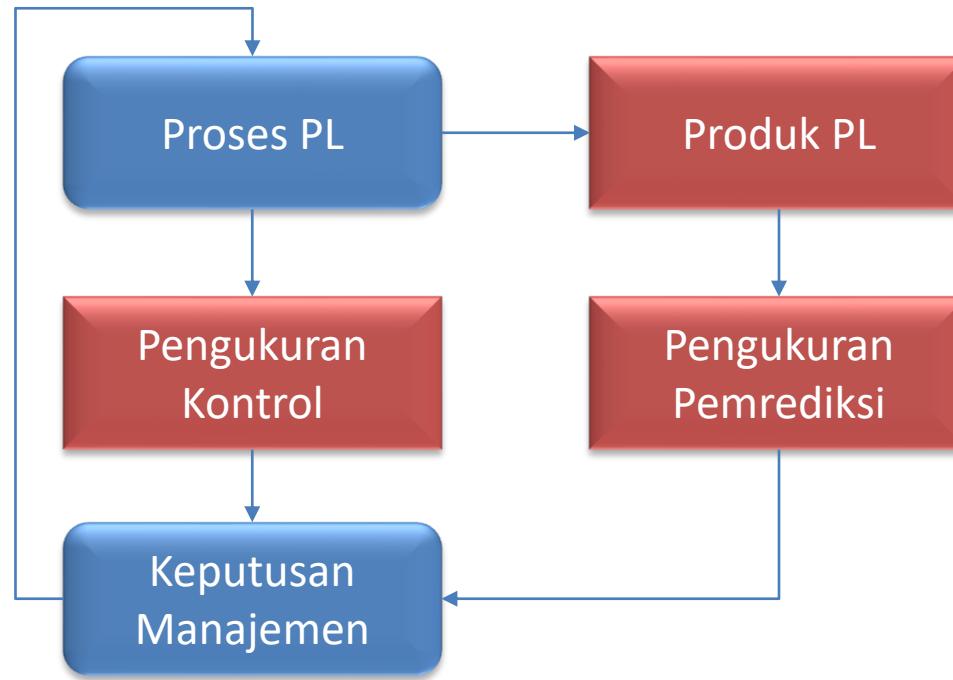
Pengukuran Perangkat Lunak – proses untuk mendapatkan nilai numerik, untuk beberapa atribut dari produk perangkat lunak atau proses perangkat lunak. Perbandingan dari nilai - nilai ini satu sama lain dan terhadap standar mengijinkan untuk mengambil kesimpulan tentang mutu dari produk dan proses perangkat lunak.

- Fokus untuk mengumpulkan metriks tentang cacat program dan proses V&V (verifikasi dan validasi)
- Metriks dapat berupa **Metriks Kontrol** atau **Metriks Prediktor**
- Kebanyakan '*illities*' tidak dapat diukur secara langsung jika tidak terdapat data historis.
- Kondisi – kondisi batas untuk semua pengukuran seharusnya dibangun di awal dan kemudian direvisi ketika bank data yang besar dari data historis telah terbangun

Pengukuran Proses dan Produk



Metriks Kontrol dan Prediktor



Contoh – contoh dari Analisis Prediktor:

- Code Reuse: SLOC = ELOC = Ported Code
- Nesting Depth: ND > 5 = Keterbacaan Rendah
- Analisis Resiko: # STR P1 > 0 pada SAT = Kehandalan Produk Rendah

Contoh – contoh Analisis Kontrol:

- STR aging: Old STRs = Produktivitas Rendah
- Volatilitas Kebutuhan: Volatilitas Tinggi = Cacat Kebutuhan

Kategori Metriks Produk Perangkat Lunak

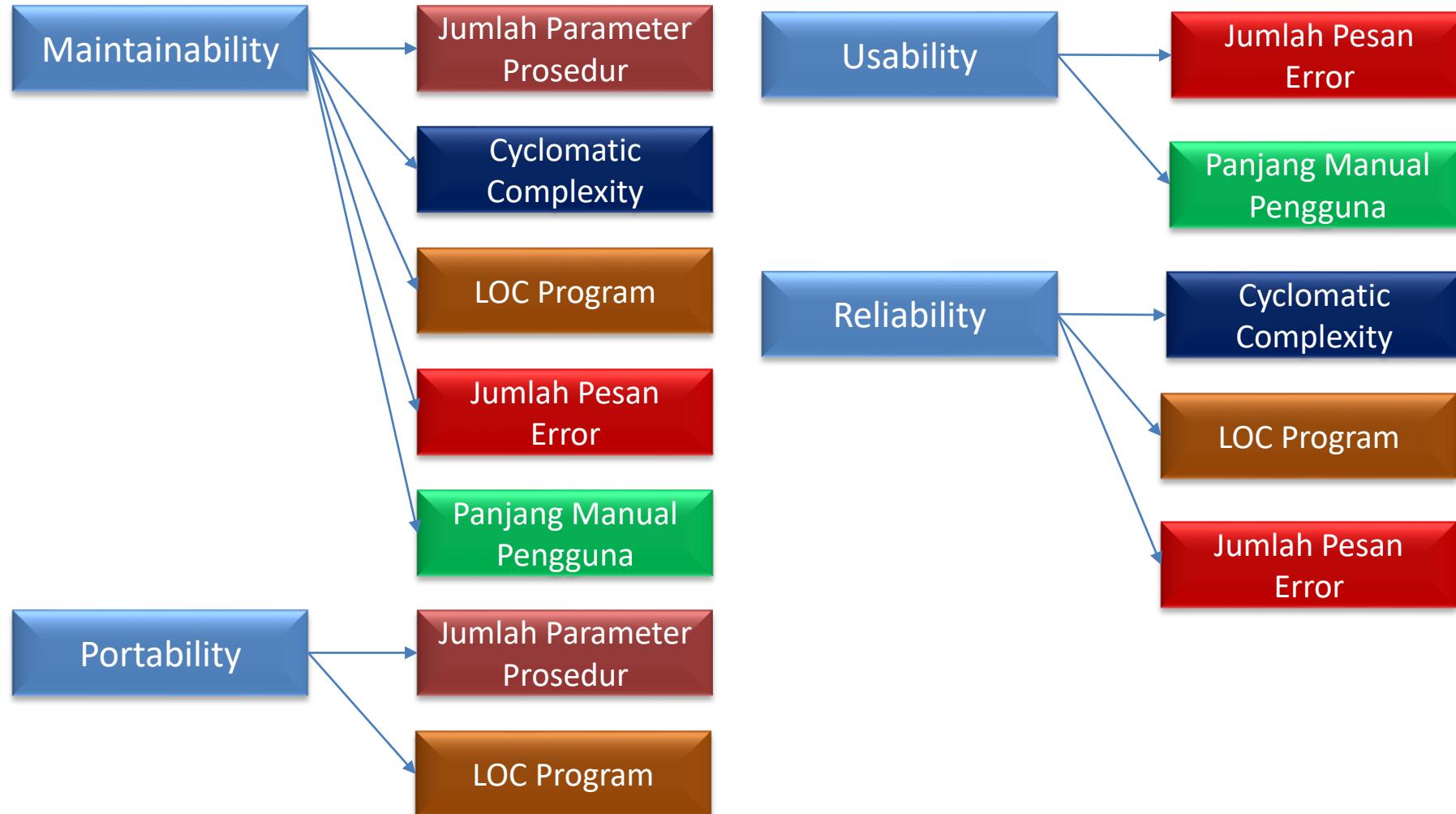


Metriks dinamis – metriks ini dikumpulkan dengan mengukur elemen – elemen selama eksekusi program. Metriks ini membantu mengkaji efisiensi dan kehandalan dari sebuah produk perangkat lunak. Parameter yang dikumpulkan dapat dengan mudah diukur (contohnya: waktu eksekusi, MTBF)



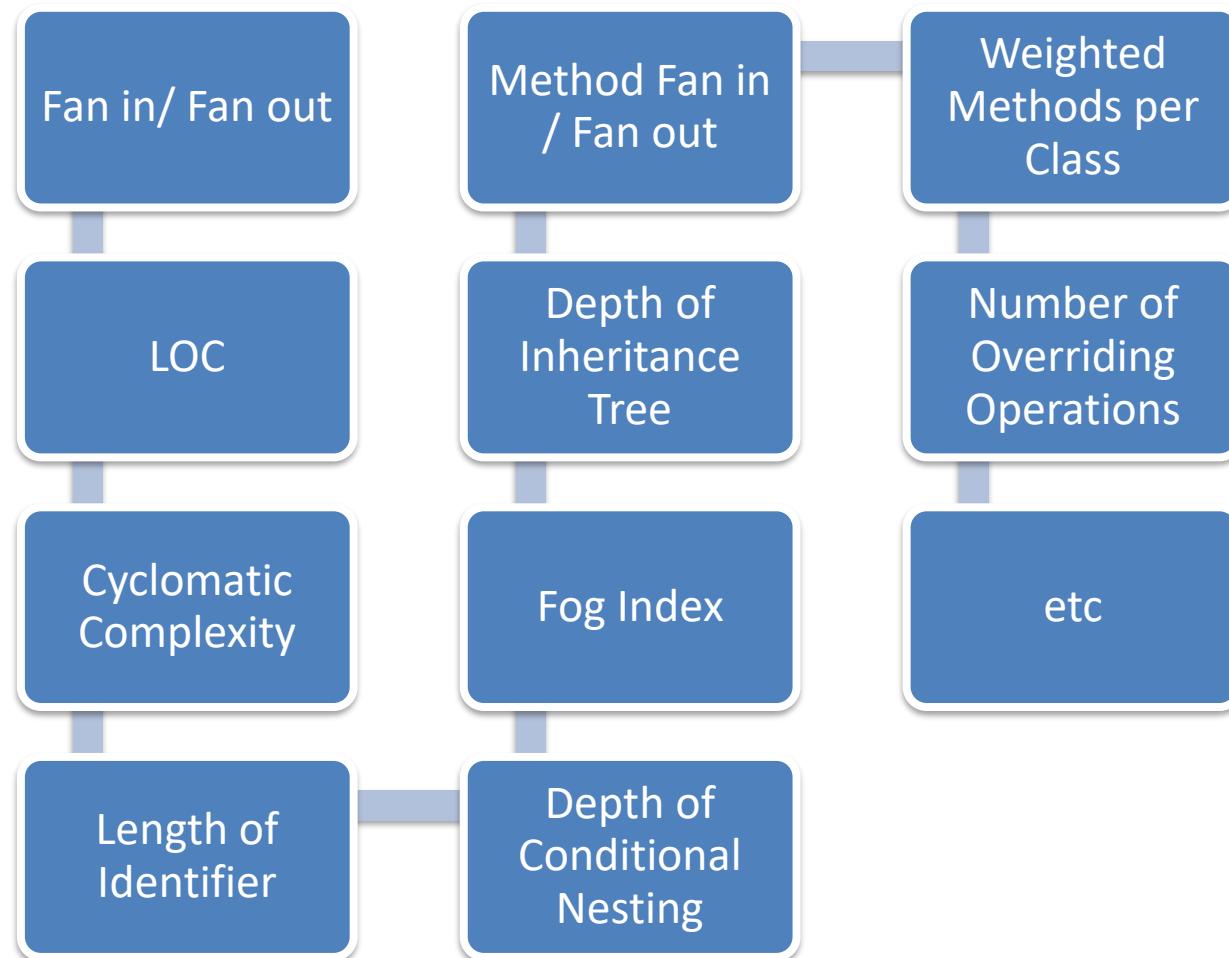
Metriks Statis – metriks ini dikumpulkan dengan mengukur berbagai parameter dari produk akhir dari pengembangan perangkat lunak. Metriks ini membantu mengkaji kompleksitas, tingkat pemahaman, dan tingkat pemeliharaan dari sebuah produk perangkat lunak. Ukuran SLOC dan ND adalah prediktor yang paling diandalkan untuk tingkat pemahaman, kompleksitas, dan pemeliharaan

“ilities” Terindikasi dari Metriks



Metriks spesifik yang relevan tergantung pada proyek, tujuan dari PMPL, dan tipe dari perangkat lunak yang sedang dikembangkan

Contoh Metriks Perangkat Lunak



Pencegahan Kecacatan



Pencegahan Kecacatan

Pencegahan Kecacatan – membangun praktek – praktek yang menurunkan ketergantungan dari teknik – teknik deteksi cacat untuk menemukan mayoritas dari bug.

- **Pelajaran yang didapatkan** – belajar dari pengalaman orang lain atau berbagi pengalaman sendiri ke orang lain di proyek
- **Mengelola dengan Metriks** – mengumpulkan metriks, memahaminya, dan membuat perubahan dari produk atau prosed berdasarkan berdasarkan hasil analisa. Metriks harus distandardisasi agar efektif.
- **Analisis Resiko** – mengidentifikasi resiko dan peluang potensial di awal program dan melacaknya untuk merealisasikannya.
- ***Build Freeze*** – tidak ada perubahan yang dibuat selama pengujian formal
- **Panduan Pengujian Tingkat Unit** – rencana pengujian dan prosedur untuk setiap pengujian unit
- **Kriteria Penerimaan Baseline** – pembangunan dari kriteria penutupan di awal (contohnya P1 STR pada FAT TRR)

Dokumentasi



Dokumen Hasil SDLC

Produk Akhir

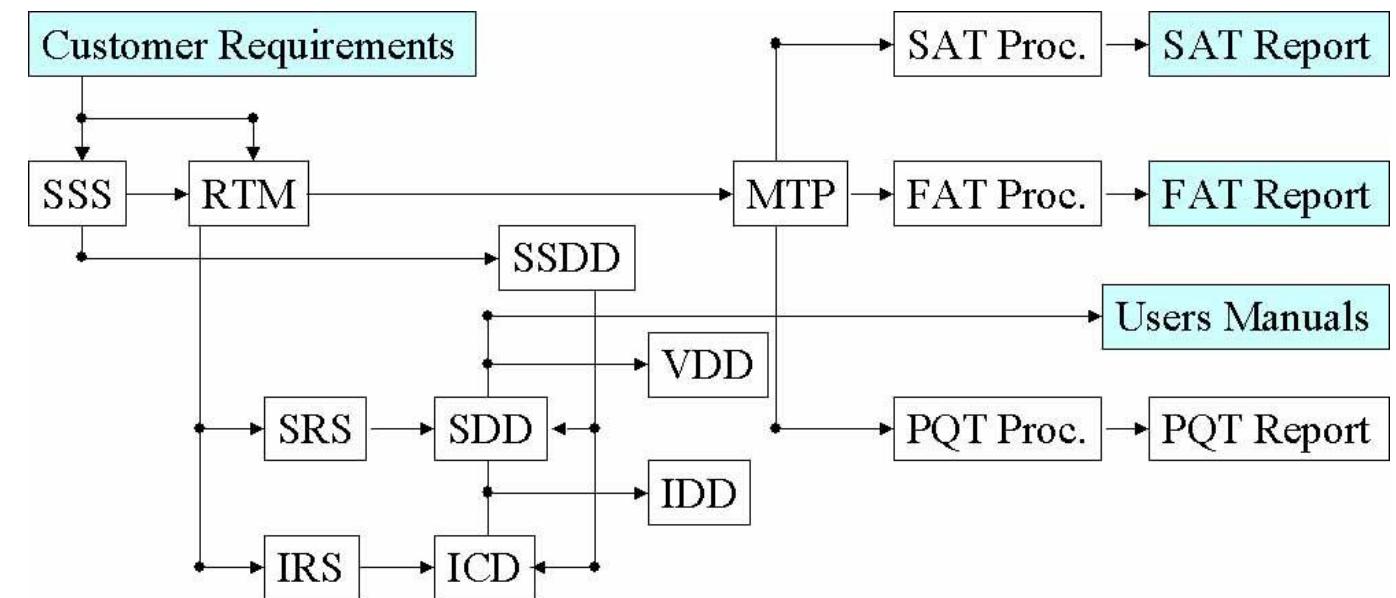
- SKPL(SRS), IRS
- PDPL(SDD), Rencana & Prosedur Uji Program / Fungsi / Sistem (PQT/FAT/SAT) & Desain Antarmuka (ICD, IDD)
- PDL, Manual Pengguna
- Kode, Rencana & Prosedur Pengujian Unit
- Hasil Pengujian Unit
- Dokumen Verifikasi Desain
- Laporan PQT
- Laporan Uji Fungsi & SAT (FAT & SAT)
- ECP menuju ke updates

Fase Pengembangan PL

- Kebutuhan Perangkat Lunak
- Desain Awal
- Desain Detil
- Kode
- Pengujian Unit
- Integrasi Perangkat Lunak
- Pengujian Komponen PL
- Pengujian Sistem PL
- Pemeliharaan dan Dukungan

Hirarki Dokumentasi

- Dokumen bukan satu – satunya cara yang nyata untuk merepresentasikan produk perangkat lunak. **Sistem Perangkat Lunak yang bekerja** adalah cara yang paling nyata untuk merepresentasikan produk perangkat lunak
- Dokumen adalah cara paling baik untuk memastikan tingkat pemahaman dari produk perangkat lunak



Andi W. R. Emanuel
andi.emmanuel@uajy.ac.id
081573011030



Terima kasih
Matur nuwun
Thank you

Dependensi vs Asosiasi vs Agregasi vs Komposisi vs Generalisasi

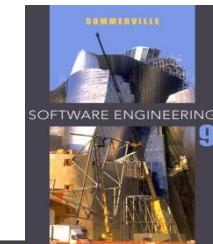
Pola Hubungan	Dependensi	Asosiasi	Agregasi	Komposisi	Generalisasi
Penjelasan	Objek yang dihubungkan hanya digunakan secara singkat (<i>brief</i>) misalnya dalam sebuah method, diinstankan, digunakan, selesai. Ketika method tersebut selesai dipanggil, objek yang dihubungkan tersebut sudah tidak ada lagi.	Objek yang dihubungkan akan bekerja dengan objek yang menghubungkan, dalam sebuah kelas sepenuhnya.	Objek yang dihubungkan akan bekerja dengan objek yang menghubungkan dengan share reference. Objek yang dihubungkan merupakan bagian dari objek yang menghubungkan dan dapat menghilangkan reference kepadanya.	Objek yang dihubungkan merupakan bagian internal dari sebuah kelas, dimana hidupnya tergantung pada kelas yang menghubungkannya.	Objek yang dihubungkan merupakan tipe dari objek yang menghubungkan.
Simbol					
Contoh	<pre> classDiagram class SMS { -text -nmr_tujuan } class PengirimSMS { +kirim() } SMS "3" --> "1" PengirimSMS </pre>	<pre> classDiagram class FormPengirimSMS class PengirimSMS FormPengirimSMS "3" --> "1" PengirimSMS </pre>	<pre> classDiagram class FormPengirimSMS class TextField class Button FormPengirimSMS "3" *--> "1" TextField FormPengirimSMS "3" *--> "1" Button </pre>	<pre> classDiagram class SMSConversation class SMS SMSConversation "3" *--> "1..*" SMS </pre>	<pre> classDiagram class MessageService class SMS class MMS MessageService < -- SMS MessageService < -- MMS </pre>
Implementasi (Java)	<pre> class SMS{ private String text; private String nmr_tujuan; } class PengirimSMS{ public void kirim(){ SMS sms = new SMS(); //objek sms hanya ada dalam kirim() } } </pre>	<pre> class FormPengirimSMS{ PengirimSMS pengirimSMS; } class PengirimSMS{ FormPengirimSMS formSMS; } </pre>	<pre> class FormPengirimSMS{ TextField sms_input; Button btn_kirim; } </pre> <p>Catatan : Anggap TextField dan Button merupakan bagian dari API pemrograman yang digunakan.</p>	<pre> class SMSConversation{ ArrayList<SMS> sms; } </pre>	<pre> class MessageService{} class SMS extends MessageService{} class MMS extends MessageService{} </pre>

Referensi

Learning UML 2.0. Rus Miles & Kim Hamilton

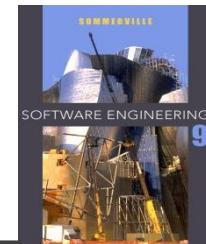
Object Oriented Software Engineering. Third edition. Bernd Bruegge & Allen H. Dutoit

Disajikan dalam mata kuliah Rekayasa Perangkat Lunak, Departemen Ilmu Komputer/Informatika, Universitas Diponegoro.



Bab 1- Pengantar Rekayasa Perangkat Lunak

Pertanyaan yang Sering Diajukan tentang Rekayasa Perangkat Lunak



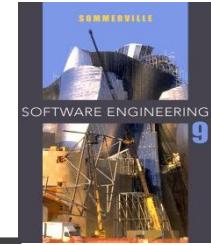
Pertanyaan	Menjawab
Apa itu perangkat lunak?	Program komputer dan dokumentasi terkait. Produk perangkat lunak dapat dikembangkan untuk pelanggan tertentu atau dapat dikembangkan untuk pasar umum.
Apa atribut perangkat lunak yang baik?	Perangkat lunak yang baik harus memberikan fungsionalitas dan kinerja yang diperlukan kepada pengguna dan harus dapat dipelihara, dapat diandalkan, dan dapat digunakan.
Apa itu rekayasa perangkat lunak?	Rekayasa perangkat lunak adalah disiplin rekayasa yang berkaitan dengan semua aspek produksi perangkat lunak.
Apa itu mendasar kegiatan rekayasa?	perangkat lunakSpesifikasi perangkat lunak, pengembangan perangkat lunak, perangkat lunak validasi dan evolusi perangkat lunak.
Apa perbedaan antara rekayasa perangkat lunak dan ilmu komputer?	Ilmu komputer berfokus pada teori dan dasar-dasar; rekayasa perangkat lunak berkaitan dengan kepraktisan mengembangkan dan memberikan perangkat lunak yang berguna.
Apa perbedaan antara rekayasa perangkat lunak dan rekayasa sistem?	Sistem yang berkaitan dengan semua aspek rekayasa pengembangan sistem berbasis komputer termasuk perangkat keras, perangkat lunak dan rekayasa proses. Rekayasa perangkat lunak adalah bagian dari proses yang lebih umum ini.

Pertanyaan yang Sering Diajukan tentang Rekayasa Perangkat Lunak



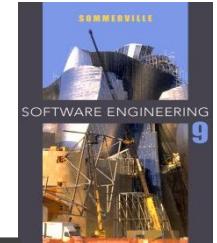
Pertanyaan	Menjawab
Apa tantangan utama yang dihadapi rekayasa perangkat lunak?	<i>Mengatasi keragaman yang meningkat, tuntutan untuk mengurangi waktu pengiriman dan mengembangkan perangkat lunak yang dapat dipercaya.</i>
Apa itu biaya dari perangkat lunak rekayasa?	<i>Sekitar 60% dari biaya perangkat lunak adalah biaya pengembangan, 40% adalah biaya pengujian. Untuk perangkat lunak kustom, biaya evolusi sering kali melebihi biaya pengembangan.</i>
Apa teknik dan metode rekayasa perangkat lunak terbaik?	<i>Sementara semua proyek perangkat lunak harus dikelola dan dikembangkan secara profesional, teknik yang berbeda sesuai untuk berbagai jenis sistem. Misalnya, game harus selalu dikembangkan menggunakan serangkaian prototipe sedangkan sistem kontrol kritis keselamatan memerlukan spesifikasi yang lengkap dan dapat dianalisis untuk dikembangkan. Oleh karena itu, Anda tidak dapat mengatakan bahwa satu metode lebih baik daripada yang lain.</i>
Apa perbedaan yang dibuat web dengan rekayasa perangkat lunak?	<i>Web telah menyebabkan ketersediaan layanan perangkat lunak dan kemungkinan mengembangkan sistem berbasis layanan yang sangat terdistribusi. Pengembangan sistem berbasis web telah menyebabkan perubahan besar dalam bahasa pemrograman dan penggunaan kembali perangkat lunak.</i>

Atribut Esensial dari Perangkat Lunak yang Baik Sering disebut sebagai "Metrik Kualitas" Terkadang disebut "Persyaratan Non-Fungsional"



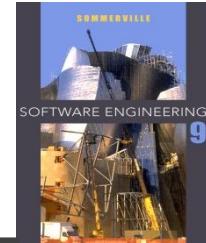
karakteristik produk	Keterangan
Pemeliharaan	<i>Perangkat lunak harus ditulis sedemikian rupa sehingga dapat berkembang untuk memenuhi perubahan kebutuhan pelanggan. Ini adalah atribut penting karena perubahan perangkat lunak merupakan persyaratan tak terelakkan dari lingkungan bisnis yang berubah.</i>
Keandalan dan keamanan	Keandalan perangkat lunak mencakup berbagai karakteristik termasuk keandalan, keamanan, dan keselamatan. Perangkat lunak yang dapat diandalkan tidak boleh menyebabkan kerusakan fisik atau ekonomi jika terjadi kegagalan sistem. Pengguna jahat tidak boleh mengakses atau merusak sistem.
Efisiensi	<i>Perangkat lunak tidak boleh memboroskan sumber daya sistem seperti siklus memori dan prosesor. Oleh karena itu, efisiensi mencakup daya tanggap, waktu pemrosesan, pemanfaatan memori, dll.</i>
Keberterimaan	Perangkat lunak harus dapat diterima oleh tipe pengguna yang dirancang. Ini berarti bahwa itu harus dapat dimengerti, dapat digunakan dan kompatibel dengan sistem lain yang mereka gunakan.

Atribut Esensial dari Perangkat Lunak yang Baik Sering disebut sebagai "Metrik Kualitas" Terkadang disebut "Persyaratan Non-Fungsional"



- Lagi:
- Banyak metrik kualitas lainnya seperti
- Keandalan
- Skalabilitas
- Portabilitas
- Dapat digunakan kembali
- Kegunaan

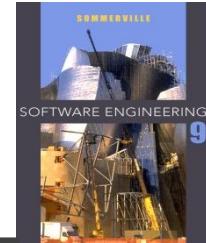
Rekayasa Perangkat Lunak



- **Rekayasa Perangkat Lunak** adalah **disiplin teknik** yang bersangkutan dengan **semua aspek** produksi perangkat lunak dari tahap awal sistem **spesifikasi** melalui **memelihara** sistem setelah mulai digunakan.
- **insinyur Gdisiplin**
 - Menggunakan teori dan metode yang tepat untuk memecahkan masalah mengingat kendala organisasi dan keuangan.
- **Semua aspek produksi perangkat lunak**
 - Bukan hanya proses teknis pembangunan. Juga manajemen proyek dan pengembangan alat, metode, dll. untuk mendukung produksi perangkat lunak.

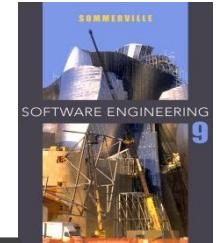
Aktivitas Proses Perangkat Lunak

(Tampilan tingkat tinggi)



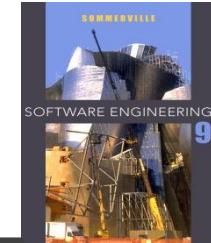
- **Spesifikasi perangkat lunak**, di mana pelanggan dan insinyur menentukan perangkat lunak yang akan diproduksi dan batasan operasinya.
- **Pengembangan perangkat lunak**, di mana perangkat lunak dirancang dan diprogram.
- **Validasi perangkat lunak**, di mana perangkat lunak diperiksa untuk memastikan bahwa itu yang dibutuhkan pelanggan.
- **Evolusi perangkat lunak**, di mana perangkat lunak dimodifikasi untuk mencerminkan perubahan kebutuhan pelanggan dan pasar.

Keanekaragaman Rekayasa Perangkat Lunak



- Ada banyak jenis sistem perangkat lunak dan adatidak ada seperangkat teknik perangkat lunak yang universal yang berlaku untuk semua ini.
- Metode dan alat rekayasa perangkat lunak yang digunakan bergantung pada:**jenis aplikasi yang sedang dikembangkan**, persyaratan pelanggan dan latar belakang tim pengembangan.
- Banyak perusahaan tidak berlangganan persis ke proses tertentu. **Sebaliknya, mereka menggunakan gabungan aktivitas yang 'sesuai' dengan organisasi, alat, pengalaman, dan terus terang apa yang 'berhasil untuk mereka'.**

Contoh kunci dari beragam sistem



Aplikasi yang berdiri sendiri

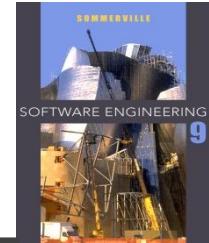
Aplikasi berbasis transaksi interaktif

**Sistem waktu nyata tertanam Sistem
kontrol proses**

Sistem batch

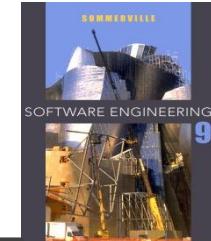
Sistem pengumpulan data. Dan banyak lagi...

Dasar-dasar Rekayasa Perangkat Lunak



- Beberapa prinsip dasar berlaku untuk semua jenis sistem perangkat lunak
- Sistem harus dikembangkan menggunakan **Sebuah dikelola dan memahami perkembangan proses**.
- Ketergantungan dan kinerja penting untuk semua sistem. Tapi ada banyak **metrik kualitas lain yang diperlukan!!**
- Memahami dan mengelola **spesifikasi perangkat lunak** dan **persyaratan** (apa yang harus dilakukan perangkat lunak) adalah **sangat penting**.
- Jika perlu, Anda harus **menggunakan kembali perangkat lunak** yang telah dikembangkan daripada menulis perangkat lunak baru.

Rekayasa Perangkat Lunak dan Web



- **Web** sekarang menjadi platform untuk menjalankan aplikasi dan organisasi semakin mengembangkan sistem berbasis web daripada sistem lokal.

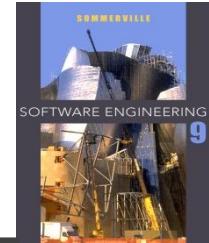
- **Layanan web** memungkinkan fungsionalitas aplikasi untuk diakses **secara global**

- Kita harus mempertimbangkan perkembangan kita secara global secara default.

- **Komputasi awan** adalah pendekatan untuk penyediaan layanan komputer di mana aplikasi berjalan dari jarak jauh di 'cloud'.

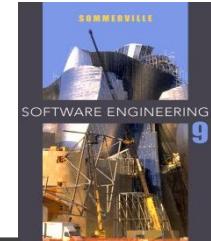
- Pengguna tidak membeli software beli bayar sesuai pemakaian.

Rekayasa Perangkat Lunak Web



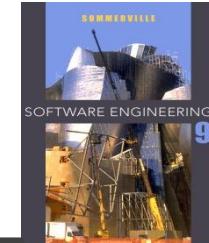
- **Perangkat lunak penggunaan kembali** adalah pendekatan dominan untuk membangun sistem berbasis web.
 - Saat membangun sistem ini, Anda berpikir tentang bagaimana Anda dapat merakitnya dari komponen perangkat lunak yang sudah ada sebelumnya dan sistem.
- Sistem berbasis web harus dikembangkan dan disampaikan secara iteratif dan bertahap.
 - Sekarang secara umum diakui bahwa itu adalah **tidak praktis** untuk menentukan semua persyaratan untuk sistem tersebut terlebih dahulu.
- **Antarmuka pengguna** adalah **dibatasi** oleh kemampuan web browser.
 - Teknologi seperti AJAX memungkinkan antarmuka yang kaya dibuat dalam browser web tetapi masih sulit digunakan.
 - Formulir web dengan skrip lokal lebih umum digunakan.

Etika Rekayasa Perangkat Lunak



- Rekayasa perangkat lunak melibatkan **tanggung jawab yang lebih luas** dari sekedar penerapan keterampilan teknis.
 - Ini secara dramatis berbeda dari pengembangan perangkat lunak. bertahun-tahun lalu.
-
- Insinyur perangkat lunak harus berperilaku jujur dan bertanggung jawab secara etis jika mereka ingin dihormati sebagai profesional.
 - Perilaku etis adalah **lagidari** sekadar menegakkan hukum tetapi melibatkan mengikuti serangkaian prinsip yang benar secara moral.

Masalah Tanggung Jawab Profesional



-Kerahasiaan

- Insinyur biasanya harus **menghormati kerahasiaan majikan mereka** atau klien terlepas dari apakah perjanjian kerahasiaan formal telah ditandatangani atau tidak.
- Saat dipecat, seringkali karyawan tidak diperbolehkan kembali ke mejanya!

-Kompetensi

- Insinyur tidak boleh salah menggambarkan tingkat kompetensi mereka. Mereka seharusnya tidak secara sadar menerima pekerjaan yang berada di luar kompetensi mereka.

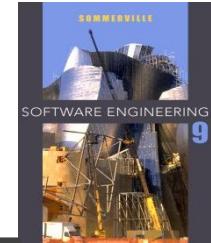
-Hak kekayaan intelektual

- Insinyur harus mengetahui undang-undang setempat yang mengatur penggunaan kekayaan intelektual seperti paten, hak cipta, dll. Mereka harus berhati-hati untuk memastikan bahwa kekayaan intelektual pemberi kerja dan klien dilindungi.

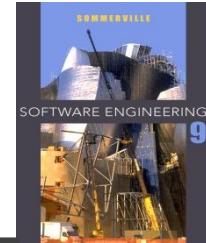
-Penyalahgunaan komputer

- Insinyur perangkat lunak tidak boleh menggunakan keterampilan teknis mereka untuk menyalahgunakan komputer orang lain. Penyalahgunaan komputer berkisar dari yang relatif sepele (misalnya bermain game di mesin perusahaan) hingga yang sangat serius (penyebaran virus).

Kode Etik ACM/IEEE



- Masyarakat profesional di AS telah bekerja sama untuk menghasilkan kode etik praktik.
 - Kami memiliki ini diposting di sepanjang lorong batin kita.
- Anggota organisasi ini mendaftar ke kode praktik ketika mereka bergabung.
 - Tentu saja Anda harus menjadi anggota ACM, IEEE CS dan mungkin beberapa orang lain di bidang spesialisasi Anda.
- Kode ini berisi delapan Prinsip yang terkait dengan perilaku dan keputusan yang dibuat oleh insinyur perangkat lunak profesional, termasuk praktisi, pendidik, manajer, penyelia dan pembuat kebijakan, serta peserta pelatihan dan mahasiswa dari profesi tersebut.



Kode Etik ACM/IEEE

Kode Etik dan Praktik Profesional Rekayasa Perangkat Lunak

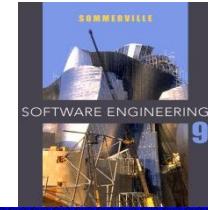
Gugus Tugas Gabungan ACM/IEEE-CS tentang Etika Rekayasa Perangkat Lunak dan Praktik Profesional

PEMBUKAAN

Versi singkat dari kode ini merangkum aspirasi pada tingkat abstraksi yang tinggi; klausa yang disertakan dalam versi lengkap memberikan contoh dan detail tentang bagaimana aspirasi ini mengubah cara kita bertindak sebagai profesional rekayasa perangkat lunak. Tanpa aspirasi, detail bisa menjadi legalistik dan membosankan; tanpa detail, aspirasi bisa terdengar tinggi tetapi kosong; bersama-sama, aspirasi dan detail membentuk kode yang kohesif.

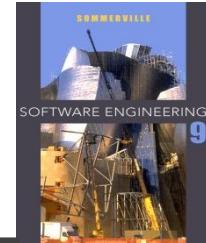
Insinyur perangkat lunak harus berkomitmen untuk membuat analisis, spesifikasi, desain, pengembangan, pengujian, dan pemeliharaan perangkat lunak sebagai profesi yang bermanfaat dan dihormati. Sesuai dengan komitmen mereka terhadap kesehatan, keselamatan, dan kesejahteraan publik, insinyur perangkat lunak harus mematuhi Delapan Prinsip berikut:

Prinsip Etika

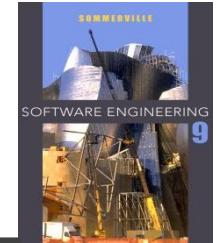


1. PUBLIK - Insinyur perangkat lunak harus bertindak secara konsisten dengan kepentingan publik.
2. KLIEN DAN MAJIKAN - Insinyur perangkat lunak harus bertindak dengan cara yang sesuai dengan kepentingan terbaik klien dan pemberi kerja mereka sesuai dengan kepentingan umum.
3. PRODUK - Insinyur perangkat lunak harus memastikan bahwa produk mereka dan modifikasi terkait memenuhi standar profesional setinggi mungkin.
4. PERTIMBANGAN - Insinyur perangkat lunak harus menjaga integritas dan independensi dalam penilaian profesional mereka.
5. MANAJEMEN - Manajer dan pemimpin rekayasa perangkat lunak harus berlangganan dan mempromosikan pendekatan etis untuk pengelolaan pengembangan dan pemeliharaan perangkat lunak.
6. PROFESI - Insinyur perangkat lunak harus memajukan integritas dan reputasi profesi yang konsisten dengan kepentingan publik.
7. KOLEGA - Insinyur perangkat lunak harus adil dan mendukung rekan-rekan mereka.
8. DIRI - Insinyur perangkat lunak harus berpartisipasi dalam pembelajaran seumur hidup mengenai praktik profesi mereka dan harus mempromosikan pendekatan etis terhadap praktik profesi.

Dilema Etis



- Ketidaksepakatan pada prinsipnya dengan kebijakan manajemen senior.**Membahas**
- Majikan Anda bertindak dengan cara yang tidak etis dan merilis sistem yang kritis terhadap keselamatan tanpa menyelesaikan pengujian sistem.**Membahas**
- Partisipasi dalam pengembangan sistem senjata militer atau sistem nuklir.**Membahas**

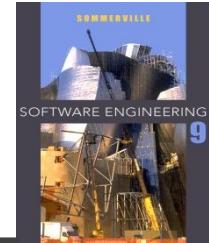


Di sini, di slide berikut adalah beberapa pemikiran pengantar tentang arsitektur.

Kami akan membahas banyak dari ini di kuliah mendatang.

Tapi untuk saat ini...

Persyaratan tingkat tinggi yang penting (terkadang disebut Cerita Pengguna atau Fitur)



Sistem harus tersedia untuk memberikan insulin bila diperlukan.

Sistem akan bekerja dengan andal dan mengirimkan jumlah insulin yang benar untuk melawan tingkat gula darah saat ini.

Oleh karena itu, sistem harus dirancang dan diimplementasikan untuk memastikan bahwa sistem selalu memenuhi persyaratan ini.

Ini (di atas) berasal dari buku, tetapi jelas mereka adalah cara untuk tingkat tinggi untuk memberikan arah nyata untuk pembangunan. Kita akan membahas mekanisme untuk menangkap fungsional dan non-kebutuhan fungsional dalam perkuliahan yang akan datang.



Bab 2 – Proses Perangkat Lunak



Topik yang dibahas

Model proses perangkat lunak

Proses kegiatan

Mengatasi perubahan

Peningkatan proses



Proses perangkat lunak

Serangkaian aktivitas terstruktur yang diperlukan untuk mengembangkan sistem perangkat lunak.

Banyak proses perangkat lunak yang berbeda tetapi semuanya melibatkan:

- ÿ Spesifikasi – mendefinisikan apa yang harus dilakukan sistem;
- ÿ Desain dan implementasi – mendefinisikan organisasi sistem dan mengimplementasikan sistem;
- ÿ Validasi – memeriksa apakah itu melakukan apa yang diinginkan pelanggan;
- ÿ Evolusi – mengubah sistem dalam menanggapi perubahan kebutuhan pelanggan.

Model proses perangkat lunak adalah representasi abstrak dari suatu proses. Ini menyajikan deskripsi proses dari beberapa perspektif tertentu.



Deskripsi proses perangkat lunak

Ketika kita menjelaskan dan mendiskusikan proses, kita biasanya berbicara tentang aktivitas dalam proses ini seperti menentukan model data, merancang antarmuka pengguna, dll. dan mengurutkan aktivitas ini.

Deskripsi proses juga dapat mencakup:

- ÿ Produk, yang merupakan hasil dari aktivitas proses;
- ÿ Peran, yang mencerminkan tanggung jawab orang-orang yang terlibat dalam proses;
- ÿ Pre-and post-conditions, yaitu pernyataan-pernyataan yang benar sebelum dan sesudah suatu kegiatan proses dilakukan atau suatu produk dihasilkan.



Proses yang digerakkan oleh rencana dan tangkas

Proses yang digerakkan oleh rencana adalah proses di mana semua kegiatan proses direncanakan sebelumnya dan kemajuan diukur terhadap rencana ini.

Dalam proses tangkas, perencanaan bersifat inkremental dan lebih mudah untuk mengubah proses untuk mencerminkan perubahan kebutuhan pelanggan.

Dalam praktiknya, sebagian besar proses praktis mencakup elemen pendekatan yang digerakkan oleh rencana dan tangkas.

Tidak ada proses perangkat lunak yang benar atau salah.



Model proses perangkat lunak



Model proses perangkat lunak

Model air terjun

ÿ Model yang digerakkan oleh rencana. Fase spesifikasi dan pengembangan yang terpisah dan berbeda.

Perkembangan bertahap

ÿ Spesifikasi, pengembangan dan validasi disisipkan. Mungkin menjadi rencana-driven atau gesit.

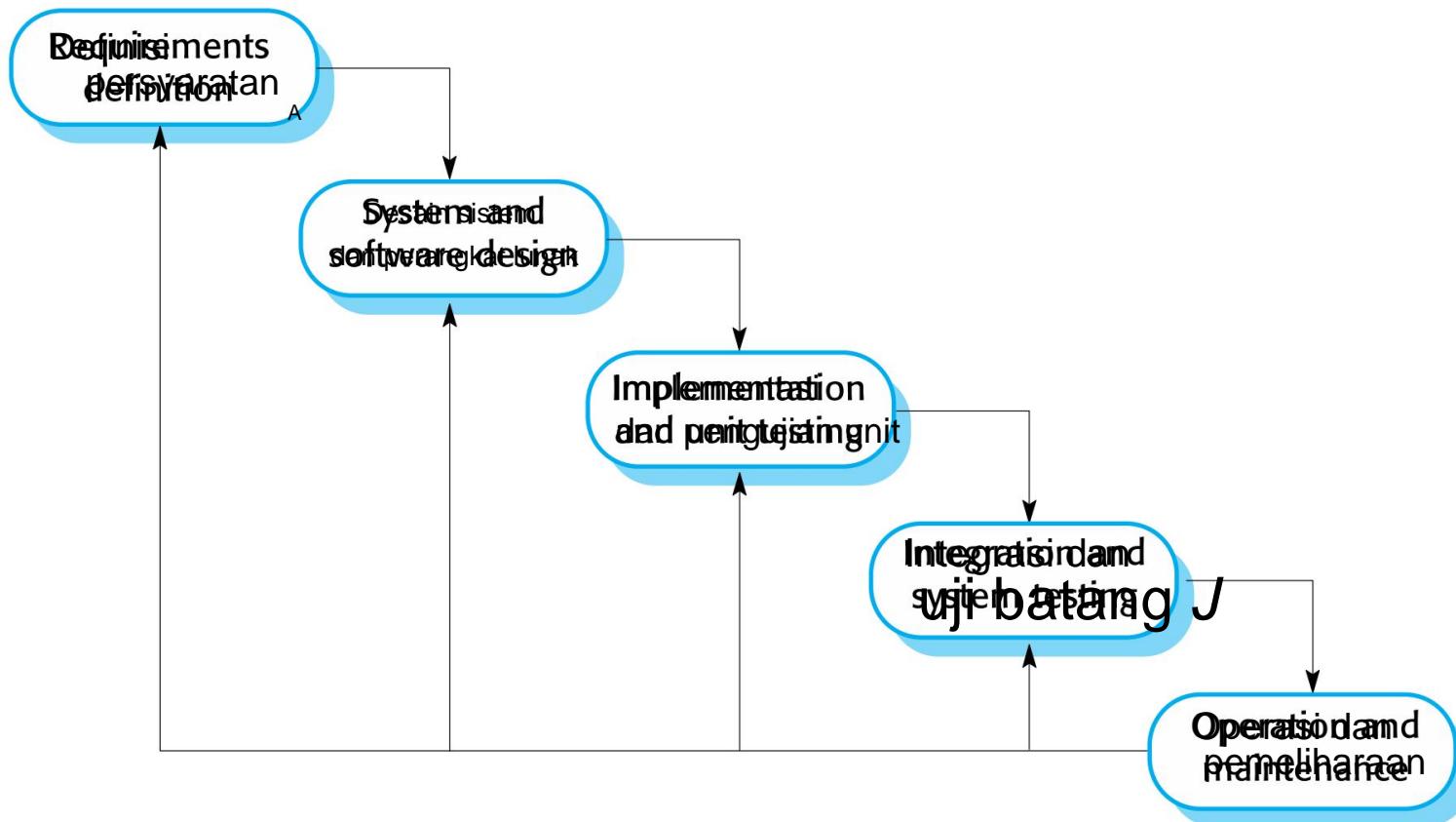
Integrasi dan konfigurasi

ÿ Sistem dirakit dari komponen yang dapat dikonfigurasi yang ada. Mungkin digerakkan oleh rencana atau gesit.

Dalam praktiknya, sebagian besar sistem besar dikembangkan menggunakan proses yang menggabungkan elemen-elemen dari semua model ini.



Model air terjun





Fase model air terjun

Ada fase yang diidentifikasi terpisah di air terjun model:

- ÿ Analisis dan definisi kebutuhan
- ÿ Desain sistem dan perangkat lunak
- ÿ Implementasi dan pengujian unit
- ÿ Integrasi dan pengujian sistem
- ÿ Pengoperasian dan Pemeliharaan

Kelemahan utama model waterfall adalah sulitnya mengakomodasi perubahan setelah proses berlangsung. Pada prinsipnya, suatu fase harus diselesaikan sebelum pindah ke fase berikutnya.



Masalah model air terjun

Pemisahan proyek yang tidak fleksibel ke dalam tahapan-tahapan yang berbeda membuat sulit untuk menanggapi perubahan kebutuhan pelanggan.

ÿ Oleh karena itu, model ini hanya sesuai ketika persyaratan dipahami dengan baik dan perubahan akan cukup terbatas selama proses desain.

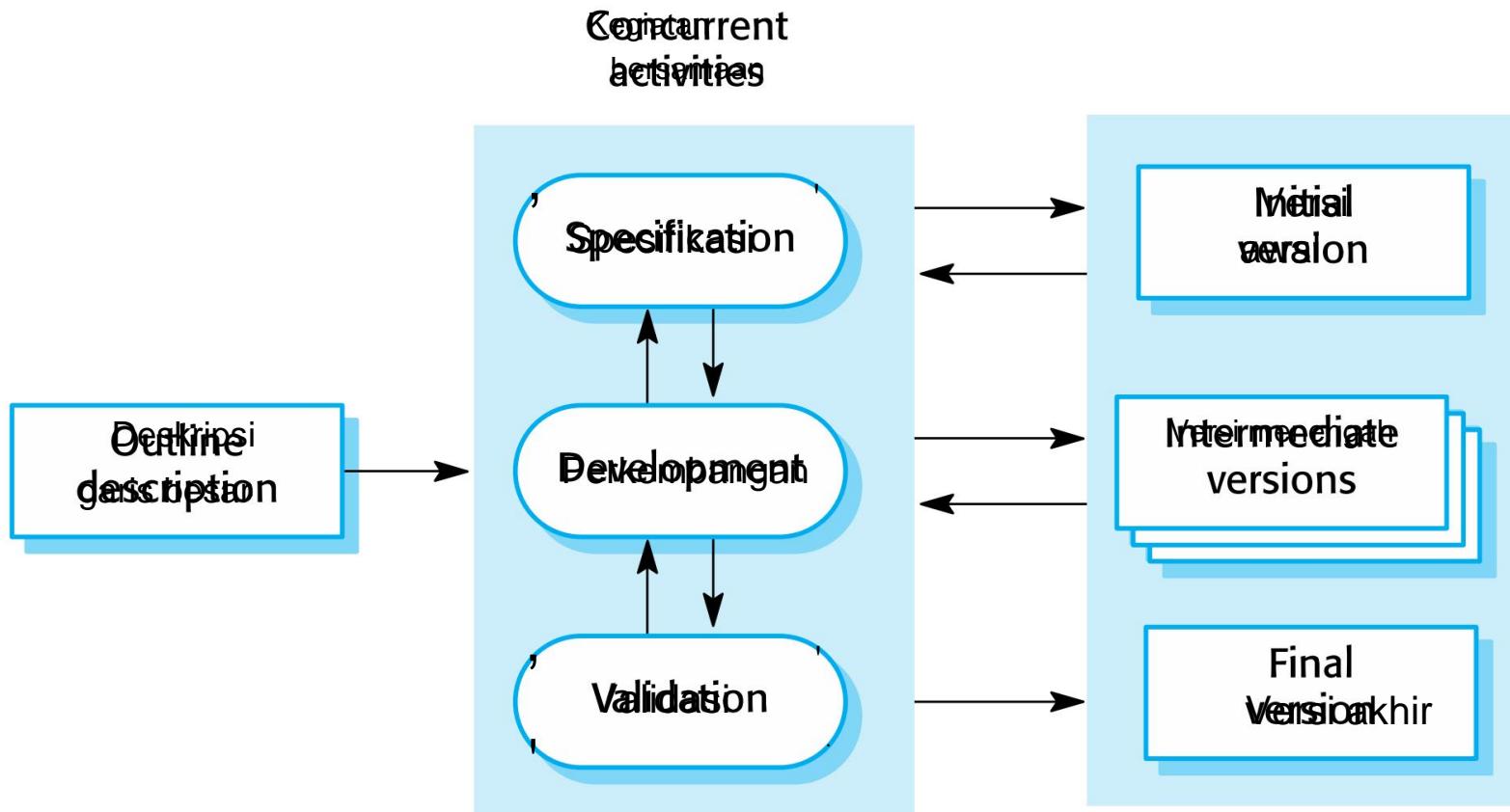
ÿ Beberapa sistem bisnis memiliki persyaratan yang stabil.

Model air terjun sebagian besar digunakan untuk proyek rekayasa sistem besar di mana sistem dikembangkan di beberapa lokasi.

ÿ Dalam keadaan seperti itu, sifat air terjun yang digerakkan oleh rencana model membantu mengkoordinasikan pekerjaan.



Perkembangan bertahap





Manfaat pengembangan tambahan

Biaya untuk mengakomodasi perubahan pelanggan
persyaratan berkurang.

- ÿ Jumlah analisis dan dokumentasi yang harus redone jauh lebih sedikit daripada yang dibutuhkan dengan model air terjun.

Lebih mudah untuk mendapatkan umpan balik pelanggan atas pekerjaan pengembangan yang telah dilakukan.

- ÿ Pelanggan dapat mengomentari demonstrasi perangkat lunak dan melihat berapa banyak yang telah dilaksanakan.

Pengiriman dan penyebaran perangkat lunak yang bermanfaat kepada pelanggan dapat dilakukan dengan lebih cepat.

- ÿ Pelanggan dapat menggunakan dan memperoleh nilai dari perangkat lunak lebih awal dari yang dimungkinkan dengan proses air terjun.



Masalah perkembangan tambahan

Prosesnya tidak terlihat.

- ÿ Manajer membutuhkan kiriman reguler untuk mengukur kemajuan. Jika sistem dikembangkan dengan cepat, tidak hemat biaya untuk menghasilkan dokumen yang mencerminkan setiap versi sistem.

Struktur sistem cenderung menurun seiring peningkatan baru

sudah ditambahkan.

- ÿ Kecuali waktu dan uang dihabiskan untuk refactoring untuk meningkatkan perangkat lunak, perubahan reguler cenderung merusak strukturnya. Memasukkan perubahan perangkat lunak lebih lanjut menjadi semakin sulit dan mahal.



Integrasi dan konfigurasi

Berdasarkan penggunaan kembali perangkat lunak di mana sistem terintegrasi dari komponen yang ada atau sistem COTS (Commercial-off-the shelf).

Elemen yang digunakan kembali dapat dikonfigurasi untuk menyesuaikan perilaku dan fungsinya dengan kebutuhan pengguna

Penggunaan kembali sekarang menjadi pendekatan standar untuk membangun banyak jenis sistem bisnis

↳ Penggunaan kembali dibahas lebih mendalam di Bab 15.

Jenis perangkat lunak yang dapat digunakan kembali



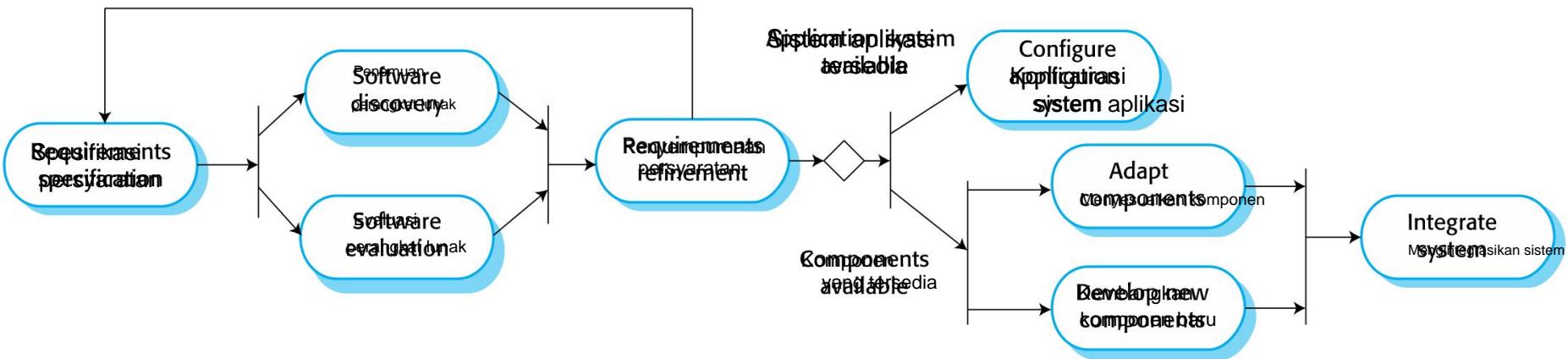
Sistem aplikasi yang berdiri sendiri (kadang disebut COTS) yang dikonfigurasi untuk digunakan dalam lingkungan tertentu.

Kumpulan objek yang dikembangkan sebagai paket untuk diintegrasikan dengan kerangka komponen seperti .NET atau J2EE.

Layanan web yang dikembangkan sesuai dengan layanan standar dan yang tersedia untuk pemanggilan jarak jauh.



Rekayasa perangkat lunak berorientasi penggunaan kembali





Tahapan proses utama

Spesifikasi persyaratan

Penemuan dan evaluasi perangkat lunak

Penyempurnaan persyaratan

Konfigurasi sistem aplikasi

Adaptasi dan integrasi komponen



Keuntungan dan kerugian

Mengurangi biaya dan risiko karena lebih sedikit perangkat lunak yang dikembangkan dari awal

Pengiriman dan penerapan sistem yang lebih cepat

Tetapi kompromi persyaratan tidak dapat dihindari sehingga sistem mungkin tidak memenuhi kebutuhan nyata pengguna

Hilangnya kendali atas evolusi elemen sistem yang digunakan kembali



Proses kegiatan



Proses kegiatan

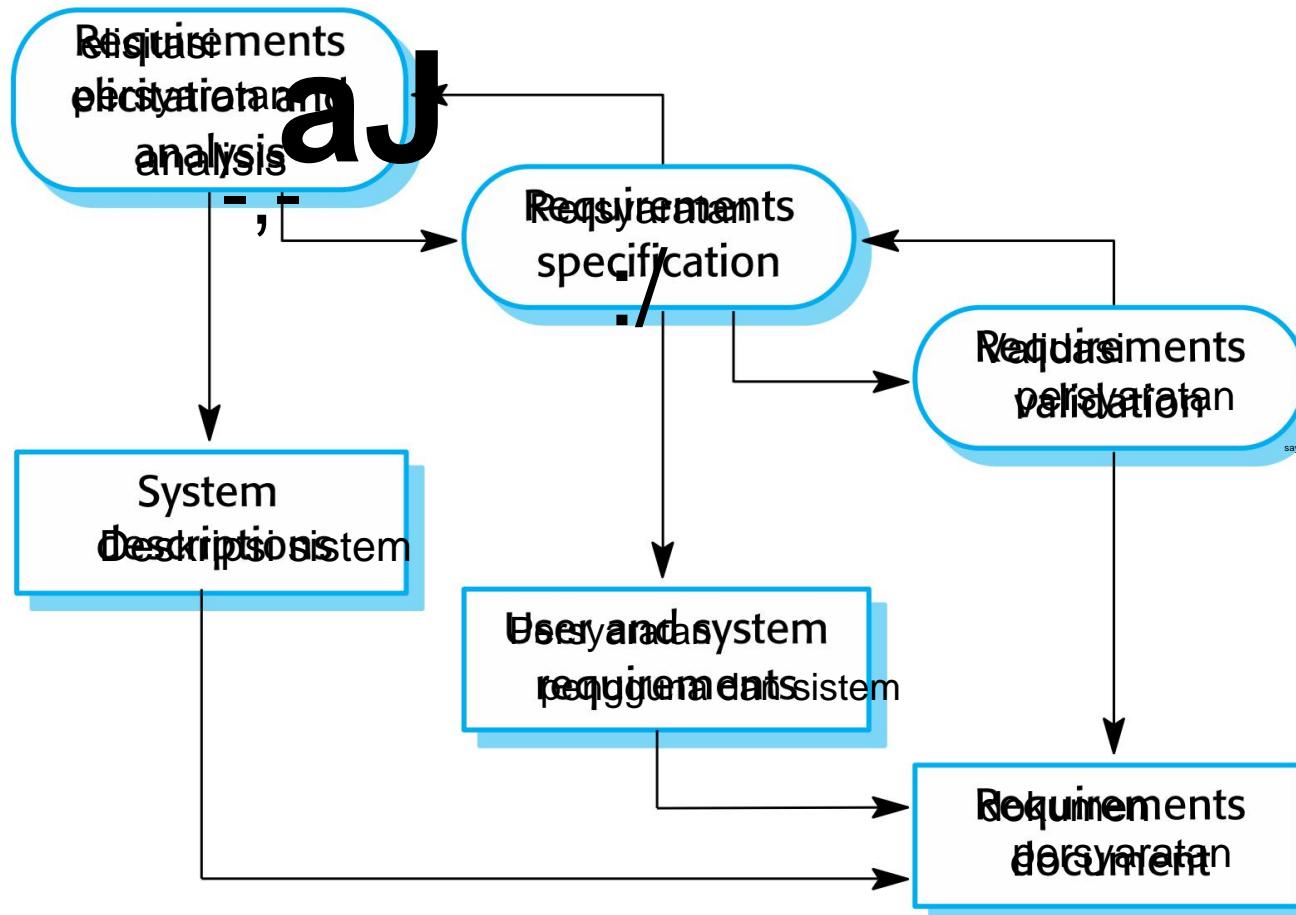
Proses perangkat lunak nyata adalah rangkaian aktivitas teknis, kolaboratif, dan manajerial antar-daun dengan tujuan keseluruhan untuk menentukan, merancang, mengimplementasikan, dan menguji sistem perangkat lunak.

Empat aktivitas proses dasar spesifikasi, pengembangan, validasi, dan evolusi diatur secara berbeda dalam proses pengembangan yang berbeda.

Misalnya, dalam model air terjun, mereka diatur secara berurutan, sedangkan dalam pengembangan inkremental mereka disisipkan.



Proses rekayasa persyaratan





Spesifikasi perangkat lunak

Proses menetapkan layanan apa yang dibutuhkan dan kendala pada operasi dan pengembangan sistem.

Proses rekayasa persyaratan

ÿ Persyaratan elisitasi dan analisis

- Apa yang dibutuhkan atau diharapkan oleh pemangku kepentingan sistem dari sistem?

ÿ Spesifikasi kebutuhan

- Mendefinisikan persyaratan secara rinci

ÿ Validasi persyaratan

- Memeriksa validitas persyaratan



Desain dan implementasi perangkat lunak

Proses mengubah spesifikasi sistem menjadi sistem yang dapat dieksekusi.

Desain perangkat lunak

ÿ Merancang struktur perangkat lunak yang mewujudkan spesifikasi;

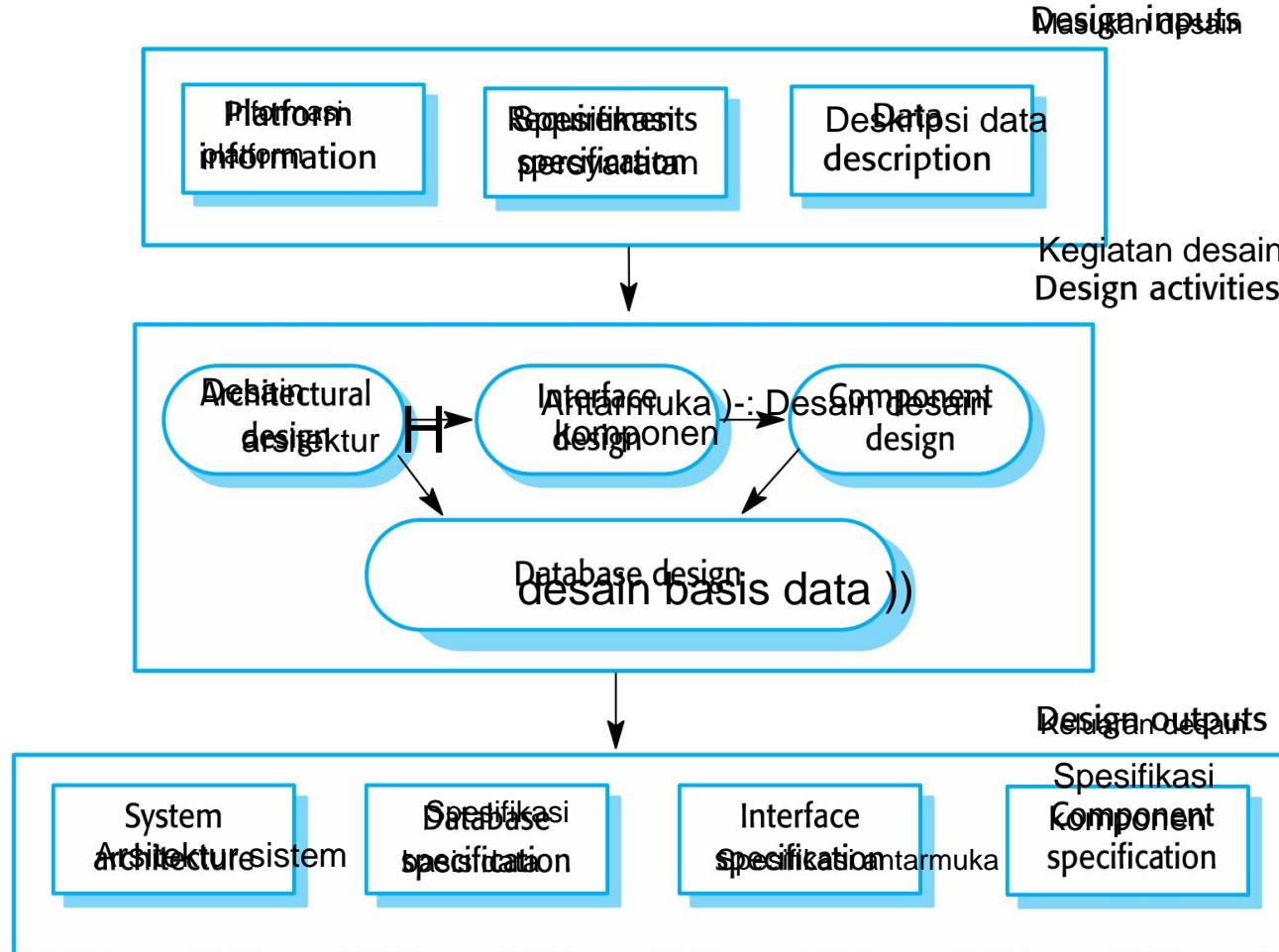
Implementasi

ÿ Terjemahkan struktur ini ke dalam program yang dapat dieksekusi;

Kegiatan desain dan implementasi erat terkait dan mungkin antar-daun.



Model umum dari proses desain





Kegiatan desain

Desain *arsitektur*, di mana Anda mengidentifikasi struktur keseluruhan sistem, komponen utama (subsistem atau modul), hubungan mereka dan bagaimana mereka didistribusikan.

Desain *basis data*, di mana Anda mendesain struktur data sistem dan bagaimana ini direpresentasikan dalam basis data.

Desain *antarmuka*, di mana Anda menentukan antarmuka antara komponen sistem.

Pemilihan *dan desain komponen*, tempat Anda mencari komponen yang dapat digunakan kembali. Jika tidak tersedia, Anda mendesain cara pengoperasiannya.



Implementasi sistem

Perangkat lunak diimplementasikan baik dengan mengembangkan program atau program atau dengan mengkonfigurasi sistem aplikasi.

Desain dan implementasi adalah aktivitas yang disisipkan untuk sebagian besar jenis sistem perangkat lunak.

Pemrograman adalah aktivitas individu tanpa standar proses.

Debugging adalah aktivitas menemukan kesalahan program dan memperbaiki kesalahan tersebut.



Validasi perangkat lunak

Verifikasi dan validasi (V & V) dimaksudkan untuk menunjukkan bahwa suatu sistem sesuai dengan spesifikasinya dan memenuhi persyaratan pelanggan sistem.

Melibatkan pemeriksaan dan peninjauan proses dan sistem pengujian.

Pengujian sistem melibatkan pelaksanaan sistem dengan kasus uji yang diturunkan dari spesifikasi data nyata yang akan diproses oleh sistem.

Pengujian adalah aktivitas V & V yang paling umum digunakan.



Tahapan pengujian





Tahap pengujian

Pengujian komponen

- ÿ Komponen individu diuji secara independen; ÿ
- Komponen dapat berupa fungsi atau objek atau pengelompokan yang koheren dari entitas ini.

Pengujian sistem

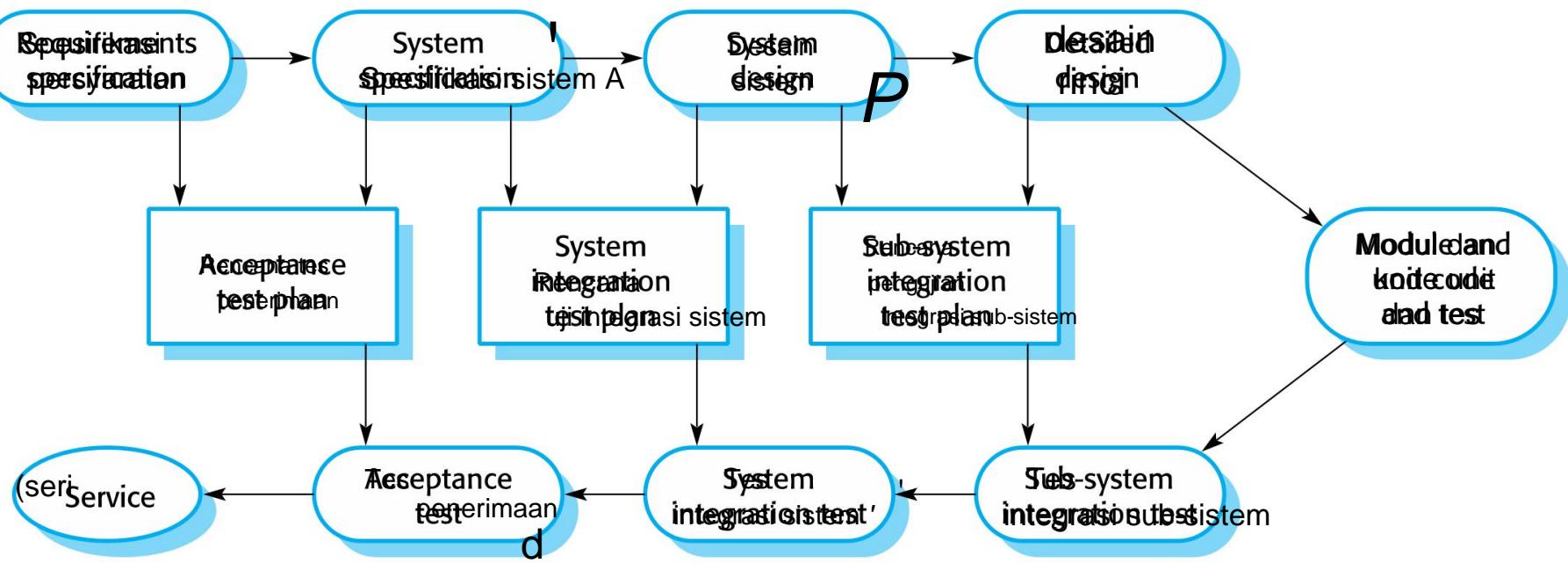
- ÿ Pengujian sistem secara keseluruhan. Pengujian properti yang muncul sangat penting.

Pengujian pelanggan

- ÿ Pengujian dengan data pelanggan untuk memeriksa bahwa sistem memenuhi kebutuhan pelanggan.



Fase pengujian dalam proses perangkat lunak yang digerakkan oleh rencana (model-V)





Evolusi perangkat lunak

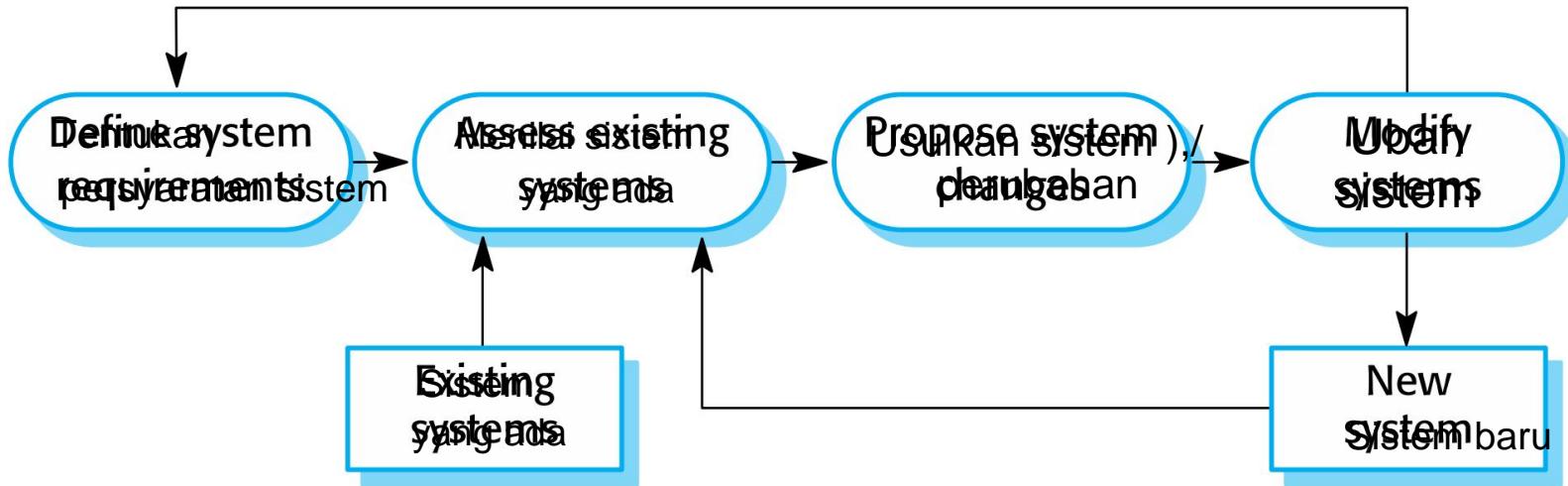
Perangkat lunak pada dasarnya fleksibel dan dapat berubah.

Saat persyaratan berubah melalui perubahan bisnis keadaan, perangkat lunak yang mendukung bisnis juga harus berkembang dan berubah.

Meskipun telah ada demarkasi antara pengembangan dan evolusi (pemeliharaan), hal ini semakin tidak relevan karena semakin sedikit sistem yang benar-benar baru.



Evolusi sistem





Mengatasi perubahan



Mengatasi perubahan

Perubahan tidak dapat dihindari di semua proyek perangkat lunak besar.

- ÿ Perubahan bisnis menyebabkan persyaratan sistem yang baru dan berubah
- ÿ Teknologi baru membuka kemungkinan baru untuk meningkatkan implementasi
- ÿ Mengubah platform memerlukan perubahan aplikasi

Perubahan mengarah pada pengrajan ulang sehingga biaya perubahan mencakup pengrajan ulang (misalnya persyaratan analisis ulang) serta biaya penerapan fungsionalitas baru



Mengurangi biaya pengrajan ulang

Perubahan antisipasi, di mana proses perangkat lunak mencakup aktivitas yang dapat mengantisipasi kemungkinan perubahan sebelum pengrajan ulang yang signifikan diperlukan.

ÿ Misalnya, sistem prototipe dapat dikembangkan untuk menunjukkan beberapa fitur kunci dari sistem kepada pelanggan.

Toleransi perubahan, dimana proses dirancang agar perubahan dapat diakomodasi dengan biaya yang relatif rendah.

ÿ Ini biasanya melibatkan beberapa bentuk pengembangan tambahan. Perubahan yang diusulkan dapat diimplementasikan secara bertahap yang belum dikembangkan. Jika ini tidak mungkin, maka hanya satu kenaikan (sebagian kecil dari sistem) yang dapat diubah untuk memasukkan perubahan tersebut.



Mengatasi persyaratan yang berubah

Sistem prototipe, di mana versi sistem atau bagian dari sistem dikembangkan dengan cepat untuk memeriksa kebutuhan pelanggan dan kelayakan keputusan desain. Pendekatan ini mendukung antisipasi perubahan.

Pengiriman tambahan, di mana peningkatan sistem dikirimkan ke pelanggan untuk komentar dan eksperimen. Ini mendukung penghindaran perubahan dan toleransi perubahan.



Prototipe perangkat lunak

Prototipe adalah versi awal dari sistem yang digunakan untuk mendemonstrasikan konsep dan mencoba opsi desain.

Sebuah prototipe dapat digunakan dalam:

- ÿ Proses rekayasa persyaratan untuk membantu dengan persyaratan elisitasi dan validasi;
- ÿ Dalam proses desain untuk mengeksplorasi pilihan dan mengembangkan desain UI;
- ÿ Dalam proses pengujian untuk menjalankan tes back-to-back.



Manfaat membuat prototipe

Peningkatan kegunaan sistem.

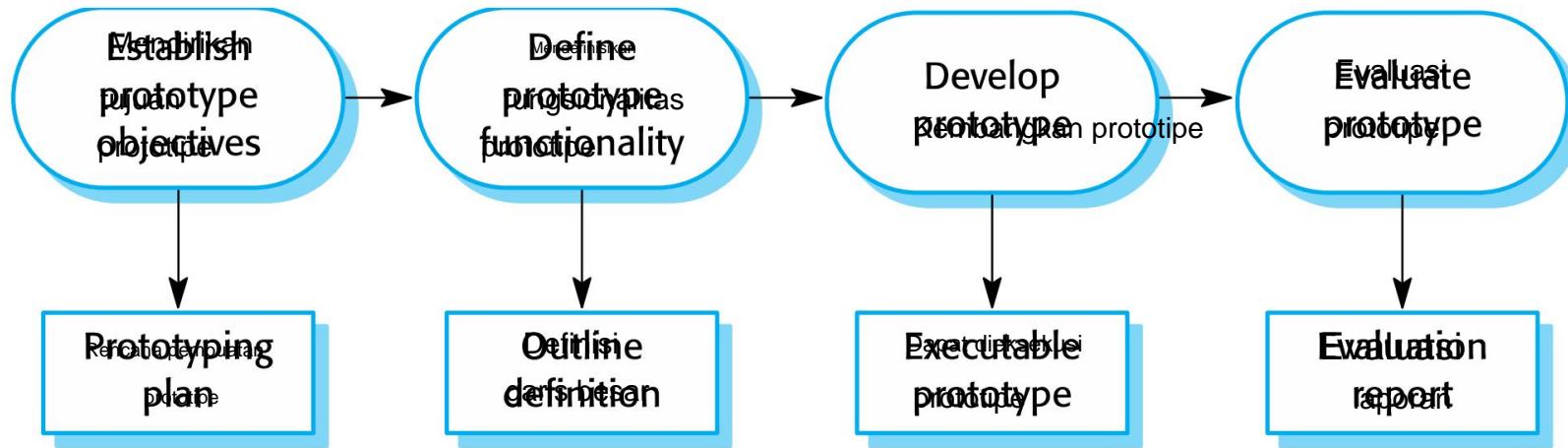
Kecocokan yang lebih dekat dengan kebutuhan nyata pengguna.

Peningkatan kualitas desain.

Peningkatan pemeliharaan.

Mengurangi upaya pengembangan.

Proses pengembangan prototipe

Software Engineering
Ian Sommerville 10e



Pengembangan prototipe

Mungkin didasarkan pada bahasa atau alat prototipe cepat

Mungkin melibatkan meninggalkan fungsionalitas

- ÿ Prototipe harus fokus pada area produk yang tidak baik dipahami;
- ÿ Pemeriksaan kesalahan dan pemulihan mungkin tidak disertakan dalam prototipe;
- ÿ Fokus pada persyaratan fungsional daripada non-fungsional seperti keandalan dan keamanan



Prototipe yang dibuang

Prototipe harus dibuang setelah pengembangan karena bukan dasar yang baik untuk sistem produksi:

- ÿ Mungkin tidak mungkin untuk menyetel sistem agar memenuhi persyaratan non-fungsional;
- ÿ Prototipe biasanya tidak didokumentasikan;
- ÿ Struktur prototipe biasanya terdegradasi melalui rapid mengubah;
- ÿ Prototipe mungkin tidak akan memenuhi standar kualitas organisasi yang normal.



Pengiriman tambahan

Daripada mengirimkan sistem sebagai pengiriman tunggal, pengembangan dan pengiriman dipecah menjadi peningkatan dengan setiap peningkatan memberikan bagian dari fungsionalitas yang diperlukan.

Persyaratan pengguna diprioritaskan dan persyaratan prioritas tertinggi disertakan dalam peningkatan awal.

Setelah pengembangan suatu kenaikan dimulai, persyaratan dibekukan meskipun persyaratan untuk kenaikan selanjutnya dapat terus berkembang.



Pengembangan dan pengiriman tambahan

Perkembangan bertahap

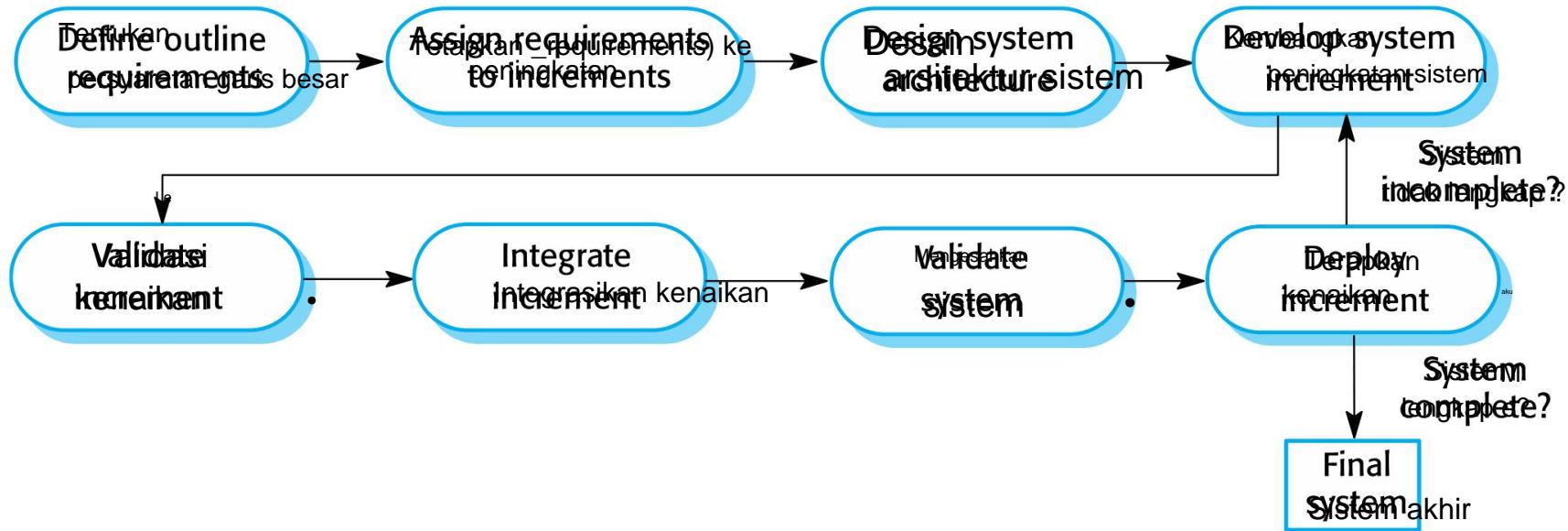
- ÿ Mengembangkan sistem secara inkremental dan mengevaluasi setiap inkremen sebelum melanjutkan ke pengembangan inkremental berikutnya;
- ÿ Pendekatan normal yang digunakan dalam metode tangkas;
- ÿ Evaluasi dilakukan oleh proxy pengguna/pelanggan.

Pengiriman bertahap

- ÿ Menyebarkan kenaikan untuk digunakan oleh pengguna akhir;
- ÿ Evaluasi yang lebih realistik tentang penggunaan praktis perangkat lunak;
- ÿ Sulit diimplementasikan untuk sistem pengganti karena peningkatan memiliki fungsionalitas yang lebih sedikit daripada sistem yang diganti.



Pengiriman tambahan





Keuntungan pengiriman tambahan

Nilai pelanggan dapat dikirimkan dengan setiap kenaikan jadi fungsionalitas sistem tersedia sebelumnya.

Peningkatan awal bertindak sebagai prototipe untuk membantu memperoleh persyaratan untuk kenaikan selanjutnya.

Risiko kegagalan proyek secara keseluruhan lebih rendah.

Layanan sistem dengan prioritas tertinggi cenderung menerima pengujian paling banyak.



Masalah pengiriman tambahan

Sebagian besar sistem memerlukan seperangkat fasilitas dasar yang digunakan oleh bagian yang berbeda dari sistem.

ÿ Karena persyaratan tidak didefinisikan secara rinci sampai kenaikan diimplementasikan, akan sulit untuk mengidentifikasi fasilitas umum yang dibutuhkan oleh semua kenaikan.

Inti dari proses berulang adalah bahwa spesifikasi dikembangkan dalam hubungannya dengan perangkat lunak.

ÿ Namun, ini bertentangan dengan model pengadaan banyak organisasi, di mana spesifikasi sistem yang lengkap adalah bagian dari kontrak pengembangan sistem.



Peningkatan proses



Peningkatan proses

Banyak perusahaan perangkat lunak telah beralih ke perangkat lunak perbaikan proses sebagai cara untuk meningkatkan kualitas perangkat lunak mereka, mengurangi biaya atau mempercepat proses pengembangan mereka.

Perbaikan proses berarti memahami proses yang ada dan mengubah proses ini untuk meningkatkan kualitas produk dan/atau mengurangi biaya dan waktu pengembangan.



Pendekatan untuk perbaikan

Pendekatan kematangan proses, yang berfokus pada peningkatan proses dan manajemen proyek serta memperkenalkan praktik rekayasa perangkat lunak yang baik.

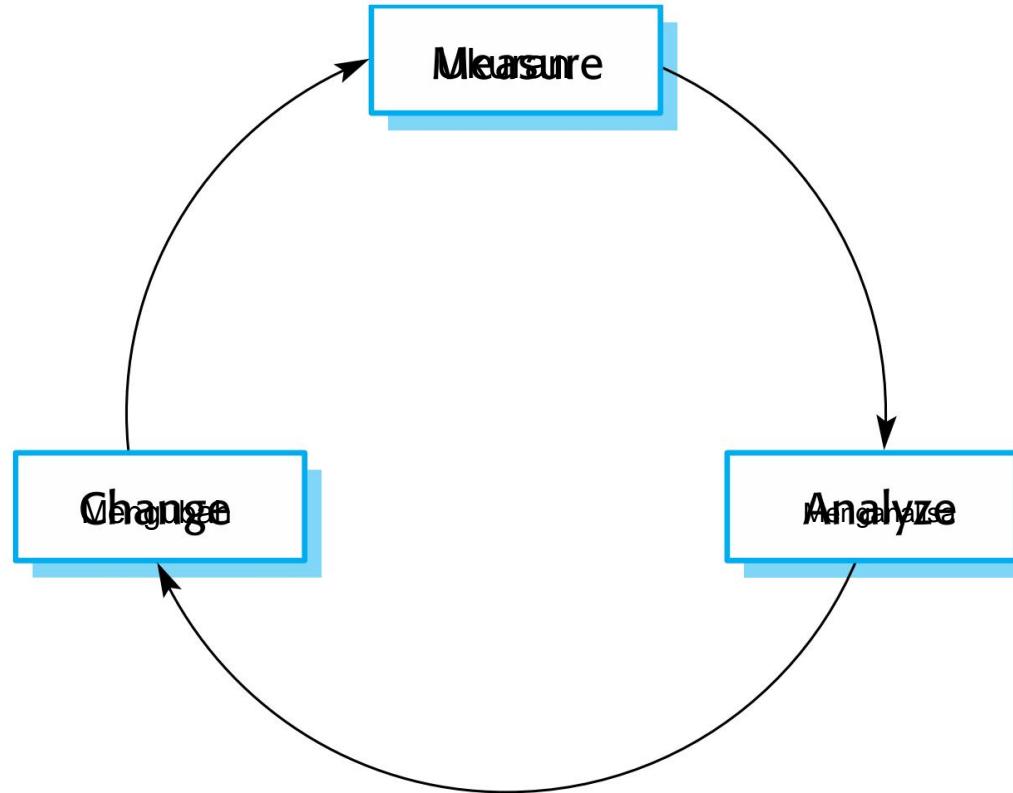
ÿ Tingkat kematangan proses mencerminkan sejauh mana praktik teknis dan manajemen yang baik telah diadopsi dalam proses pengembangan perangkat lunak organisasi.

Pendekatan tangkas, yang berfokus pada iteratif pengembangan dan pengurangan overhead dalam proses perangkat lunak.

ÿ Karakteristik utama dari metode tangkas adalah pengiriman fungsionalitas yang cepat dan responsif terhadap perubahan kebutuhan pelanggan.



Siklus perbaikan proses





Kegiatan perbaikan proses

Pengukuran proses

ŷ Anda mengukur satu atau lebih atribut dari proses perangkat lunak atau produk. Pengukuran ini membentuk dasar yang membantu Anda memutuskan apakah perbaikan proses telah efektif.

Analisis proses

ŷ Proses saat ini dinilai, dan kelemahan proses serta hambatan diidentifikasi. Model proses (kadang-kadang disebut peta proses) yang menggambarkan proses dapat dikembangkan.

Perubahan proses

ŷ Perubahan proses diusulkan untuk mengatasi beberapa kelemahan proses yang teridentifikasi. Ini diperkenalkan dan siklus dilanjutkan untuk mengumpulkan data tentang efektivitas perubahan.



Pengukuran proses

Jika memungkinkan, data proses kuantitatif harus dikumpulkan

ÿ Namun, di mana organisasi tidak memiliki definisi yang jelas standar proses ini sangat sulit karena Anda tidak tahu apa yang harus diukur. Suatu proses mungkin harus didefinisikan sebelum pengukuran apa pun dimungkinkan.

Pengukuran proses harus digunakan untuk menilai perbaikan proses

ÿ Tetapi ini tidak berarti bahwa pengukuran harus mendorong perbaikan. Penggerak perbaikan harus menjadi tujuan organisasi.



Metrik proses

Waktu yang dibutuhkan untuk menyelesaikan aktivitas proses

ÿ Misalnya Kalender waktu atau upaya untuk menyelesaikan suatu kegiatan atau proses.

Sumber daya yang dibutuhkan untuk proses atau aktivitas

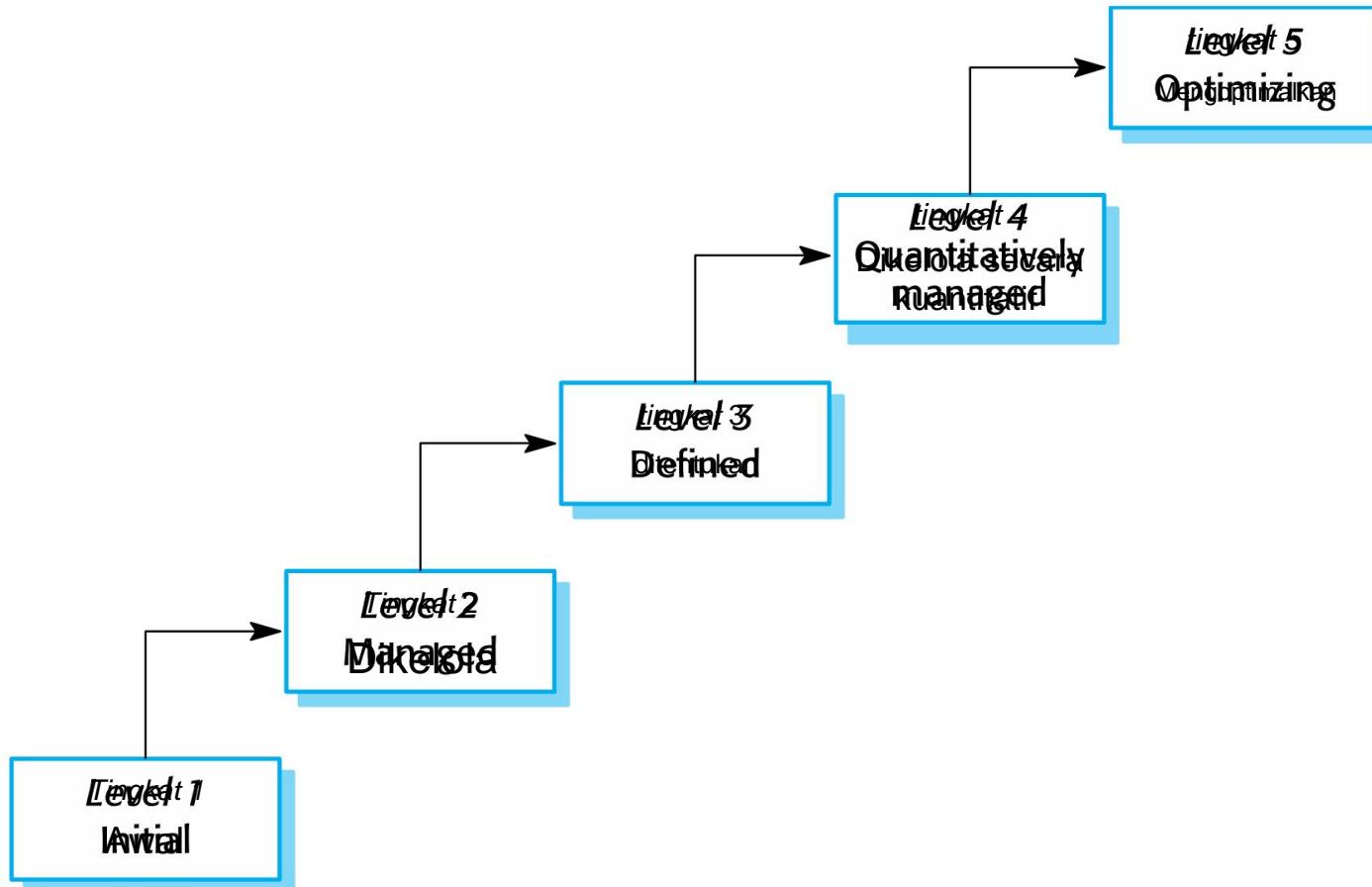
ÿ Misal Total usaha dalam person-days.

Jumlah kemunculan peristiwa tertentu

ÿ Misalnya Jumlah cacat yang ditemukan.



Tingkat kematangan kemampuan





Model kematangan kemampuan SEI

Inisial

ÿ Pada dasarnya tidak terkontrol Dapat diulang

Ditetapkan

ÿ Prosedur dan strategi manajemen proses didefinisikan dan digunakan

Dikelola

ÿ Strategi manajemen mutu ditetapkan dan digunakan

Pengoptimalan

ÿ Strategi peningkatan proses ditetapkan dan digunakan



Poin-poin penting

Proses perangkat lunak adalah aktivitas yang terlibat dalam menghasilkan sistem perangkat lunak. Model proses perangkat lunak adalah representasi abstrak dari proses ini.

Model proses umum menggambarkan organisasi proses perangkat lunak. Contoh model umum ini termasuk model 'air terjun', pengembangan inkremental, dan pengembangan berorientasi penggunaan kembali.

Rekayasa kebutuhan adalah proses mengembangkan spesifikasi perangkat lunak.



Poin-poin penting

Proses desain dan implementasi berkaitan dengan mengubah spesifikasi persyaratan menjadi sistem perangkat lunak yang dapat dieksekusi.

Validasi perangkat lunak adalah proses pengecekan bahwa sistem sesuai dengan spesifikasinya dan memenuhi kebutuhan nyata pengguna sistem.

Evolusi perangkat lunak terjadi saat Anda berubah sistem perangkat lunak yang ada untuk memenuhi persyaratan baru. Perangkat lunak harus berevolusi agar tetap berguna.

Proses harus mencakup aktivitas seperti pembuatan prototipe dan pengiriman tambahan untuk mengatasi perubahan.



Poin-poin penting

Proses dapat terstruktur untuk pengembangan dan pengiriman berulang sehingga perubahan dapat dilakukan tanpa mengganggu sistem secara keseluruhan.

Pendekatan utama untuk perbaikan proses adalah pendekatan tangkas, diarahkan untuk mengurangi biaya proses, dan pendekatan berbasis kedewasaan berdasarkan manajemen proses yang lebih baik dan penggunaan praktik rekayasa perangkat lunak yang baik.

Kerangka maturitas proses SEI mengidentifikasi tingkat maturitas yang pada dasarnya sesuai dengan penggunaan praktik rekayasa perangkat lunak yang baik.



Bab 3 – Pengembangan Perangkat Lunak Tangkas

Topik yang dibahas



- Metode tangkas
- Teknik pengembangan tangkas
- Manajemen proyek yang gesit
- Menskalakan metode tangkas

Pengembangan perangkat lunak yang cepat



-Perkembangan dan pengiriman yang cepat sekarang sering menjadi persyaratan terpenting untuk sistem perangkat lunak

- Bisnis beroperasi dalam persyaratan yang berubah dengan cepat dan secara praktis tidak mungkin menghasilkan serangkaian persyaratan perangkat lunak yang stabil
- Perangkat lunak harus berkembang dengan cepat untuk mencerminkan perubahan kebutuhan bisnis.

-Pengembangan berbasis rencana sangat penting untuk beberapa jenis sistem tetapi tidak memenuhi kebutuhan bisnis ini.

-Metode pengembangan tangkas muncul pada akhir 1990-an yang bertujuan untuk secara radikal mengurangi waktu pengiriman untuk sistem perangkat lunak yang berfungsi

Pengembangan tangkas

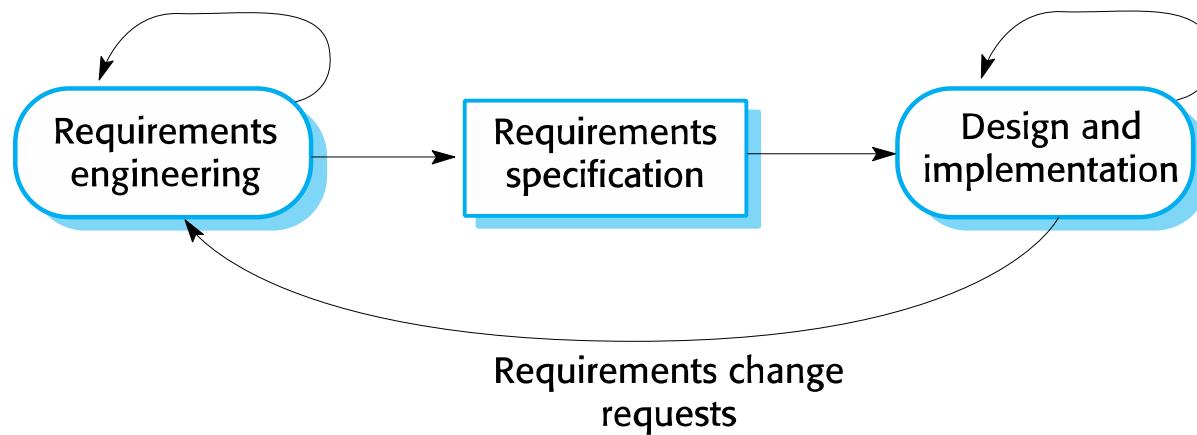


- Spesifikasi, desain, dan implementasi program saling terkait
- Sistem dikembangkan sebagai serangkaian versi atau peningkatan dengan pemangku kepentingan yang terlibat dalam spesifikasi dan evaluasi versi
- Pengiriman versi baru yang sering untuk evaluasi
- Dukungan alat yang luas (misalnya alat pengujian otomatis) yang digunakan untuk mendukung pengembangan.
- Dokumentasi minimal – fokus pada kode kerja

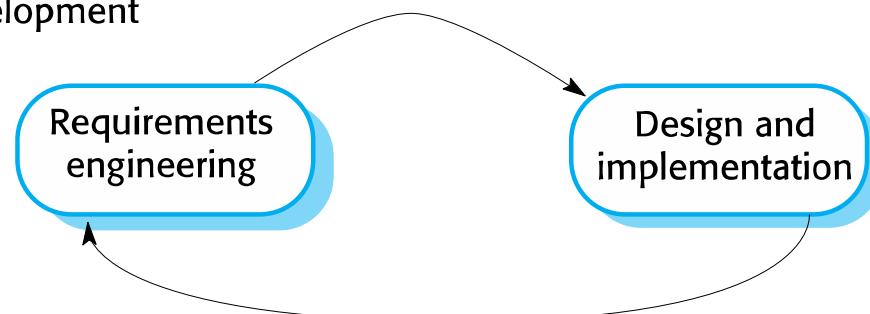
Pengembangan yang didorong oleh rencana dan gesit



Plan-based development



Agile development



Pengembangan yang didorong oleh rencana dan gesit



-Pengembangan yang didorong oleh rencana

- Pendekatan berbasis rencana untuk rekayasa perangkat lunak didasarkan pada tahap pengembangan yang terpisah dengan output yang akan dihasilkan pada setiap tahap yang direncanakan sebelumnya.
- Belum tentu model air terjun – pengembangan bertahap yang didorong oleh rencana
- Iterasi terjadi dalam aktivitas.

-Pengembangan tangkas

- Spesifikasi, desain, implementasi dan pengujian disisipkan dan keluaran dari proses pengembangan diputuskan melalui proses negosiasi selama proses pengembangan perangkat lunak.



Metode tangkas

Metode tangkas



-Ketidakpuasan dengan overhead yang terlibat dalam metode desain perangkat lunak tahun 1980-an dan 1990-an menyebabkan penciptaan metode tangkas. Metode ini:

- Fokus pada kode daripada desain
- Didasarkan pada pendekatan berulang untuk pengembangan perangkat lunak
- Dimaksudkan untuk memberikan perangkat lunak yang berfungsi dengan cepat dan berkembang dengan cepat untuk memenuhi persyaratan yang berubah.

-Tujuan dari metode tangkas adalah untuk mengurangi overhead dalam proses perangkat lunak (misalnya dengan membatasi dokumentasi) dan untuk dapat merespon dengan cepat terhadap perubahan persyaratan tanpa penggerjaan ulang yang berlebihan. .

Manifesto tangkas



-Kami menemukan cara yang lebih baik untuk mengembangkan perangkat lunak dengan melakukannya dan membantu orang lain melakukannya. Melalui pekerjaan ini kami telah mencapai nilai:

- *Individu dan interaksi atas proses dan alat Perangkat lunak yang bekerja melalui dokumentasi yang komprehensif
Kolaborasi pelanggan melalui negosiasi kontrak
Menanggapi perubahan atas mengikuti rencana*

-Artinya, sementara ada nilai pada barang-barang di sebelah kanan, kami lebih menghargai barang-barang di sebelah kiri.

Prinsip-prinsip metode tangkas



Prinsip	Keterangan
Keterlibatan pelanggan	Pelanggan harus terlibat erat selama proses pengembangan. Peran mereka adalah menyediakan dan memprioritaskan kebutuhan sistem baru dan untuk mengevaluasi iterasi sistem.
Pengiriman tambahan	Perangkat lunak ini dikembangkan secara bertahap dengan pelanggan menentukan persyaratan yang akan disertakan dalam setiap kenaikan.
Orang tidak memproses	Keterampilan tim pengembangan harus diakui dan dimanfaatkan. Anggota tim harus dibiarkan mengembangkan cara kerja mereka sendiri tanpa proses preskriptif.
Rangkullah perubahan	Harapkan persyaratan sistem berubah dan rancang sistem untuk mengakomodasi perubahan ini.
Pertahankan kesederhanaan	Fokus pada kesederhanaan dalam perangkat lunak yang sedang dikembangkan dan dalam proses pengembangan. Jika memungkinkan, bekerjalah secara aktif untuk menghilangkan kerumitan dari sistem.

Penerapan metode tangkas



- Pengembangan produk di mana perusahaan perangkat lunak sedang mengembangkan produk kecil atau menengah untuk dijual.
 - Hampir semua produk perangkat lunak dan aplikasi sekarang dikembangkan menggunakan pendekatan tangkas
- Pengembangan sistem kustom dalam suatu organisasi, di mana ada komitmen yang jelas dari pelanggan untuk terlibat dalam proses pengembangan dan di mana ada beberapa aturan dan regulasi eksternal yang mempengaruhi perangkat lunak.



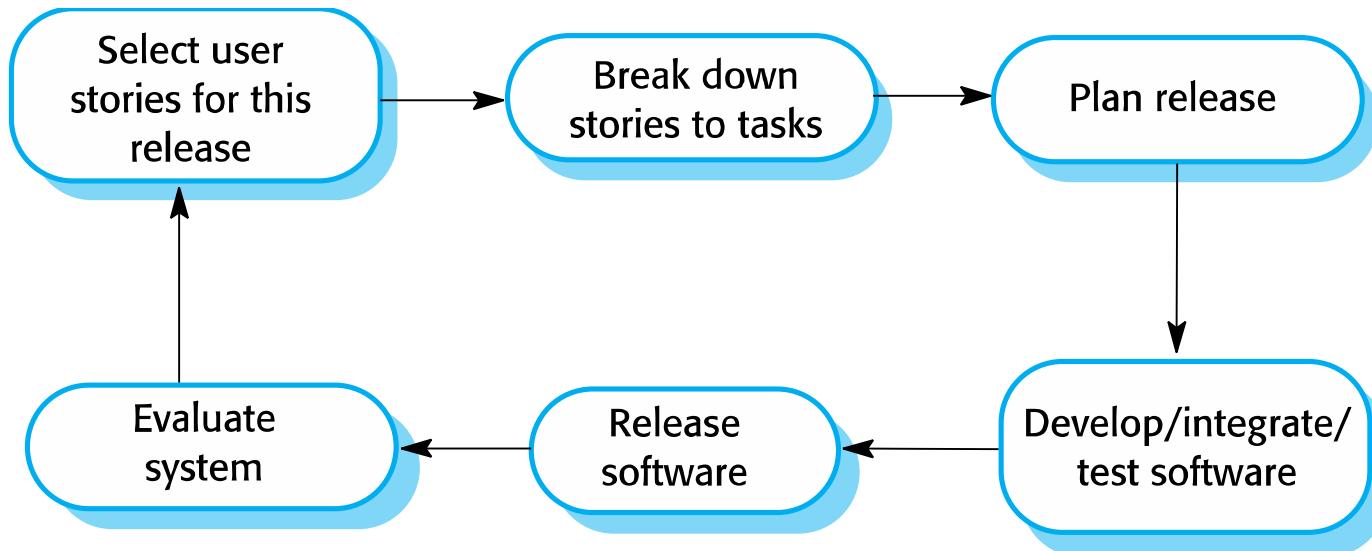
Teknik pengembangan tangkas

Pemrograman ekstrim



- Metode tangkas yang sangat berpengaruh, dikembangkan pada akhir 1990-an, yang memperkenalkan berbagai teknik pengembangan tangkas.
- Extreme Programming (XP) mengambil pendekatan 'ekstrim' untuk pengembangan berulang.
 - Versi baru dapat dibuat beberapa kali per hari;
 - Kenaikan dikirimkan ke pelanggan setiap 2 minggu;
 - Semua pengujian harus dijalankan untuk setiap build dan build hanya diterima jika pengujian berhasil dijalankan.

Siklus rilis pemrograman ekstrim



Praktik pemrograman ekstrem (a)



Prinsip atau praktik	Keterangan
Perencanaan tambahan	Persyaratan dicatat pada kartu cerita dan cerita yang akan dimasukkan dalam rilis ditentukan oleh waktu yang tersedia dan prioritas relatifnya. Pengembang memecah cerita ini menjadi 'Tugas' pengembangan. Lihat Gambar 3.5 dan 3.6.
Rilisan kecil	Serangkaian fungsionalitas minimal yang berguna yang memberikan nilai bisnis dikembangkan terlebih dahulu. Rilis sistem sering terjadi dan secara bertahap menambahkan fungsionalitas ke rilis pertama.
Desain sederhana	Cukup desain dilakukan untuk memenuhi persyaratan saat ini dan tidak lebih.
Pengembangan uji-pertama	Kerangka kerja pengujian unit otomatis digunakan untuk menulis pengujian untuk fungsionalitas baru sebelum fungsionalitas itu sendiri diimplementasikan.
Pemfaktoran ulang	<u>Semua pengembang diharapkan untuk memfaktorkan ulang kode secara terus menerus secepat mungkin ditemukan perbaikan kode. Ini membuat kode tetap sederhana dan dapat dipelihara .</u>

Praktik pemrograman ekstrem (b)



Pemrograman pasangan	Pengembang bekerja berpasangan, memeriksa pekerjaan satu sama lain dan memberikan dukungan untuk selalu melakukan pekerjaan dengan baik.
Kepemilikan kolektif	Pasangan pengembang bekerja di semua area sistem, sehingga tidak ada pulau keahlian yang berkembang dan semua pengembang bertanggung jawab atas semua kode. Siapapun bisa mengubah apapun.
Integrasi berkelanjutan	Segera setelah pekerjaan pada tugas selesai, itu diintegrasikan ke dalam keseluruhan sistem. Setelah integrasi tersebut, semua unit test dalam sistem harus lulus.
Kecepatan berkelanjutan	Lembur dalam jumlah besar tidak dianggap dapat diterima karena efek bersihnya sering kali mengurangi kualitas kode dan produktivitas jangka menengah
Pelanggan di tempat	Perwakilan dari pengguna akhir sistem (pelanggan) harus tersedia penuh waktu untuk penggunaan tim XP. Dalam proses pemrograman ekstrem, pelanggan adalah anggota tim pengembangan dan bertanggung jawab untuk membawa persyaratan sistem ke tim untuk implementasi.

XP dan prinsip tangkas



- Pengembangan bertahap didukung melalui rilis sistem yang kecil dan sering.
- Keterlibatan pelanggan berarti keterlibatan pelanggan penuh waktu dengan tim.
- Orang tidak memproses melalui pemrograman berpasangan, kepemilikan kolektif, dan proses yang menghindari jam kerja yang panjang.
- Perubahan didukung melalui rilis sistem reguler.
- Mempertahankan kesederhanaan melalui refactoring kode yang konstan.

Praktik XP yang berpengaruh



- Pemrograman ekstrem memiliki fokus teknis dan tidak mudah diintegrasikan dengan praktik manajemen di sebagian besar organisasi.
- Akibatnya, sementara pengembangan tangkas menggunakan praktik dari XP, metode seperti yang didefinisikan pada awalnya tidak banyak digunakan.

-Praktik utama

- Cerita pengguna untuk spesifikasi
- Pemfaktoran ulang
- Pengembangan uji-pertama
- Pemrograman pasangan

Cerita pengguna untuk persyaratan



- Di XP, pelanggan atau pengguna adalah bagian dari tim XP dan bertanggung jawab untuk membuat keputusan tentang persyaratan.
- Persyaratan pengguna dinyatakan sebagai cerita atau skenario pengguna.
- Ini ditulis di kartu dan tim pengembangan memecahnya menjadi tugas implementasi. Tugas-tugas ini adalah dasar dari perkiraan jadwal dan biaya.
- Pelanggan memilih cerita untuk dimasukkan dalam rilis berikutnya berdasarkan prioritas mereka dan perkiraan jadwal.

Kisah 'meresepkan obat'



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.

Contoh kartu tugas untuk meresepkan obat



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.

Pemfaktoran ulang



- Kebijaksanaan konvensional dalam rekayasa perangkat lunak adalah merancang untuk perubahan. Perlu menghabiskan waktu dan upaya untuk mengantisipasi perubahan karena hal ini akan mengurangi biaya di kemudian hari dalam siklus hidup.
- XP, bagaimanapun, menyatakan bahwa ini tidak bermanfaat karena perubahan tidak dapat diantisipasi dengan andal.
- Sebaliknya, ini mengusulkan peningkatan kode konstan (refactoring) untuk membuat perubahan lebih mudah ketika harus diimplementasikan.

Pemfaktoran ulang



- Tim pemrograman mencari kemungkinan peningkatan perangkat lunak dan melakukan perbaikan ini bahkan ketika tidak ada kebutuhan mendesak untuk itu.
- Ini meningkatkan pemahaman perangkat lunak dan dengan demikian mengurangi kebutuhan akan dokumentasi.
- Perubahan lebih mudah dilakukan karena kodennya terstruktur dengan baik dan jelas.
- Namun, beberapa perubahan memerlukan refactoring arsitektur dan ini jauh lebih mahal.

Contoh refactoring



- Pengorganisasian ulang hierarki kelas untuk menghapus kode duplikat.
- Merapikan dan mengganti nama atribut dan metode agar lebih mudah dipahami.
- Penggantian kode sebaris dengan panggilan ke metode yang telah disertakan dalam pustaka program.

Pengembangan uji-pertama



-Pengujian adalah inti dari XP dan XP telah mengembangkan pendekatan di mana program diuji setelah setiap perubahan dilakukan.

-Fitur pengujian XP:

- Pengembangan uji-pertama.
- Pengembangan tes tambahan dari skenario.
- Keterlibatan pengguna dalam pengembangan dan validasi pengujian.
- Rangkaian uji otomatis digunakan untuk menjalankan semua pengujian komponen setiap kali rilis baru dibuat.

Pengembangan yang didorong oleh tes



- Menulis tes sebelum kode menjelaskan persyaratan yang harus diterapkan.
- Pengujian ditulis sebagai program dan bukan data sehingga dapat dieksekusi secara otomatis. Tes ini mencakup pemeriksaan bahwa itu telah dijalankan dengan benar.
 - Biasanya mengandalkan kerangka pengujian seperti Junit.
- Semua pengujian sebelumnya dan baru dijalankan secara otomatis ketika fungsionalitas baru ditambahkan, sehingga memeriksa bahwa fungsionalitas baru tidak menimbulkan kesalahan.

Keterlibatan pelanggan



- Peran pelanggan dalam proses pengujian adalah untuk membantu mengembangkan tes penerimaan untuk cerita yang akan diimplementasikan dalam rilis sistem berikutnya.
- Pelanggan yang merupakan bagian dari tim menulis tes saat pengembangan berlangsung. Oleh karena itu, semua kode baru divalidasi untuk memastikan bahwa itulah yang dibutuhkan pelanggan.
- Namun, orang yang mengadopsi peran pelanggan memiliki waktu yang terbatas sehingga tidak dapat bekerja penuh waktu dengan tim pengembangan. Mereka mungkin merasa bahwa memberikan persyaratan sudah cukup berkontribusi dan mungkin enggan untuk terlibat dalam proses pengujian.

Deskripsi kasus uji untuk pemeriksaan dosis



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.

Otomatisasi pengujian



-Otomatisasi pengujian berarti bahwa pengujian ditulis sebagai komponen yang dapat dieksekusi sebelum tugas diimplementasikan

- Komponen pengujian ini harus berdiri sendiri, harus mensimulasikan pengajuan input yang akan diuji dan harus memeriksa apakah hasilnya memenuhi spesifikasi output. Kerangka kerja pengujian otomatis (misalnya Junit) adalah sistem yang memudahkan penulisan pengujian yang dapat dijalankan dan mengirimkan serangkaian pengujian untuk dieksekusi.

-Karena pengujian dilakukan secara otomatis, selalu ada serangkaian pengujian yang dapat dilakukan dengan cepat dan mudah

- Setiap kali fungsionalitas apa pun ditambahkan ke sistem, pengujian dapat dijalankan dan masalah yang telah diperkenalkan oleh kode baru dapat segera ditangkap.

Masalah dengan pengembangan tes-pertama



- Pemrogram lebih memilih pemrograman daripada pengujian dan terkadang mereka mengambil jalan pintas saat menulis tes. Misalnya, mereka mungkin menulis tes yang tidak lengkap yang tidak memeriksa semua kemungkinan pengecualian yang mungkin terjadi.
- Beberapa tes bisa sangat sulit untuk ditulis secara bertahap. Misalnya, dalam antarmuka pengguna yang kompleks, seringkali sulit untuk menulis pengujian unit untuk kode yang mengimplementasikan 'logika tampilan' dan alur kerja antar layar.
- Sulit untuk menilai kelengkapan serangkaian tes. Meskipun Anda mungkin memiliki banyak pengujian sistem, set pengujian Anda mungkin tidak memberikan cakupan yang lengkap.

Pemrograman pasangan



- Pemrograman berpasangan melibatkan pemrogram yang bekerja berpasangan, mengembangkan kode bersama.
- Ini membantu mengembangkan kepemilikan bersama atas kode dan menyebarkan pengetahuan ke seluruh tim.
- Ini berfungsi sebagai proses peninjauan informal karena setiap baris kode dilihat oleh lebih dari 1 orang.
- Ini mendorong refactoring karena seluruh tim dapat mengambil manfaat dari peningkatan kode sistem.

Pemrograman pasangan



- Dalam pemrograman berpasangan, pemrogram duduk bersama di komputer yang sama untuk mengembangkan perangkat lunak.
- Pair dibuat secara dinamis sehingga semua anggota tim saling bekerja sama selama proses pengembangan.
- Berbagi pengetahuan yang terjadi selama pemrograman berpasangan sangat penting karena mengurangi risiko keseluruhan proyek ketika anggota tim pergi.
- Pemrograman berpasangan belum tentu tidak efisien dan ada beberapa bukti yang menunjukkan bahwa pasangan yang bekerja bersama lebih efisien daripada 2 pemrogram yang bekerja secara terpisah.



Manajemen proyek yang gesit

Manajemen proyek yang gesit



- Tanggung jawab utama manajer proyek perangkat lunak adalah mengelola proyek sehingga perangkat lunak dikirimkan tepat waktu dan sesuai anggaran yang direncanakan untuk proyek tersebut.
- Pendekatan standar untuk manajemen proyek adalah plan-driven. Manajer menyusun rencana untuk proyek yang menunjukkan apa yang harus disampaikan, kapan harus disampaikan dan siapa yang akan mengerjakan pengembangan hasil proyek.
- Manajemen proyek tangkas memerlukan pendekatan yang berbeda, yang disesuaikan dengan pengembangan bertahap dan praktik yang digunakan dalam metode tangkas.

Scrum



-Scrum adalah metode tangkas yang berfokus pada pengelolaan pengembangan berulang daripada praktik tangkas tertentu.

-Ada tiga fase dalam Scrum.

- Fase awal adalah fase perencanaan garis besar di mana Anda menetapkan tujuan umum untuk proyek dan merancang arsitektur perangkat lunak.
- Ini diikuti oleh serangkaian siklus sprint, di mana setiap siklus mengembangkan peningkatan sistem.
- Fase penutupan proyek menyelesaikan proyek, melengkapi dokumentasi yang diperlukan seperti kerangka bantuan sistem dan manual pengguna dan menilai pelajaran yang didapat dari proyek.

Terminologi scrum (a)



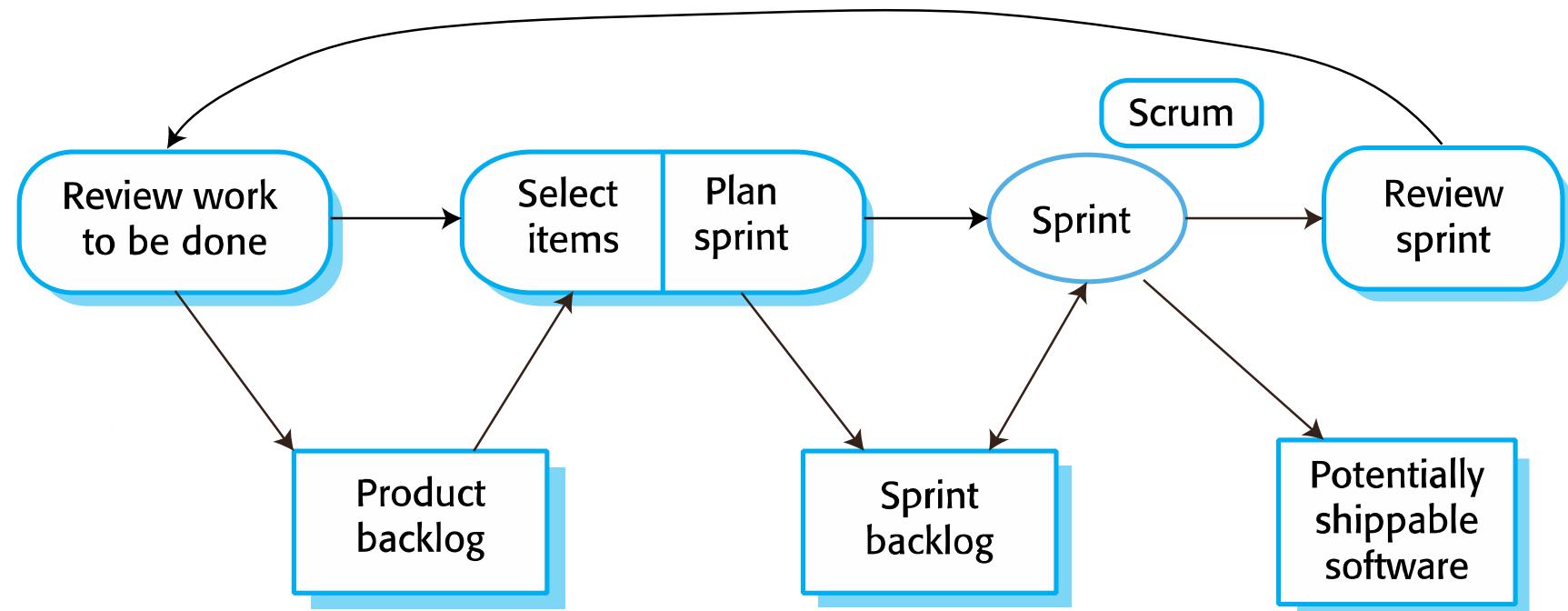
Istilah scrum	Definisi
Tim pengembangan	Sekelompok pengembang perangkat lunak yang mengatur diri sendiri, yang tidak boleh lebih dari 7 orang. Mereka bertanggung jawab untuk mengembangkan perangkat lunak dan dokumen proyek penting lainnya.
Berpotensi dikirim peningkatan produk	Peningkatan perangkat lunak yang dihasilkan dari sprint. Idenya adalah bahwa ini harus 'berpotensi dikirim' yang berarti bahwa itu dalam keadaan jadi dan tidak ada pekerjaan lebih lanjut, seperti pengujian, yang diperlukan untuk memasukkannya ke dalam produk akhir. Dalam praktiknya, ini tidak selalu dapat dicapai.
jaminan produk	Ini adalah daftar item 'yang harus dilakukan' yang harus ditangani oleh tim Scrum. Mereka mungkin definisi fitur untuk perangkat lunak, persyaratan perangkat lunak, cerita pengguna atau deskripsi tugas tambahan yang diperlukan, seperti definisi arsitektur atau dokumentasi pengguna.
Pemilik produk	Seorang individu (atau mungkin kelompok kecil) yang tugasnya adalah untuk mengidentifikasi fitur atau persyaratan produk, memprioritaskan ini untuk pengembangan dan terus meninjau backlog produk untuk memastikan bahwa proyek terus memenuhi kebutuhan bisnis yang kritis. Pemilik Produk dapat menjadi pelanggan tetapi juga dapat menjadi manajer produk di perusahaan perangkat lunak atau perwakilan pemangku kepentingan lainnya.

Terminologi scrum (b)



Istilah scrum	Definisi
Scrum	Rapat harian tim Scrum yang meninjau kemajuan dan memprioritaskan pekerjaan yang harus diselesaikan hari itu. Idealnya, ini harus menjadi pertemuan tatap muka singkat yang mencakup seluruh tim.
ScrumMaster	ScrumMaster bertanggung jawab untuk memastikan bahwa proses Scrum diikuti dan memandu tim dalam penggunaan Scrum secara efektif. Dia bertanggung jawab untuk berinteraksi dengan seluruh perusahaan dan untuk memastikan bahwa tim Scrum tidak dialihkan oleh gangguan dari luar. Pengembang Scrum bersikeras bahwa ScrumMaster tidak boleh dianggap sebagai manajer proyek. Namun, yang lain mungkin tidak selalu mudah untuk melihat perbedaannya.
Sprint	Sebuah iterasi pengembangan. Sprint biasanya berdurasi 2-4 minggu.
Kecepatan	Perkiraan seberapa banyak upaya backlog produk yang dapat dilakukan tim dalam satu sprint. Memahami kecepatan tim membantu mereka memperkirakan apa yang dapat dicakup dalam sprint dan memberikan dasar untuk mengukur peningkatan kinerja.

Siklus scrum sprint



Siklus sprint Scrum



- Sprint memiliki panjang yang tetap, biasanya 2-4 minggu.
- Titik awal perencanaan adalah product backlog, yaitu daftar pekerjaan yang harus dilakukan pada proyek.
- Tahap seleksi melibatkan semua tim proyek yang bekerja dengan pelanggan untuk memilih fitur dan fungsionalitas dari product backlog yang akan dikembangkan selama sprint.

Siklus lari cepat



- Setelah ini disepakati, tim mengatur diri mereka sendiri untuk mengembangkan perangkat lunak.
- Selama tahap ini tim diisolasi dari pelanggan dan organisasi, dengan semua komunikasi disalurkan melalui apa yang disebut 'Scrum master'.
- Peran master Scrum adalah untuk melindungi tim pengembangan dari gangguan eksternal.
- Di akhir sprint, pekerjaan yang dilakukan ditinjau dan dipresentasikan kepada pemangku kepentingan. Siklus sprint berikutnya kemudian dimulai.

Kerja tim di Scrum



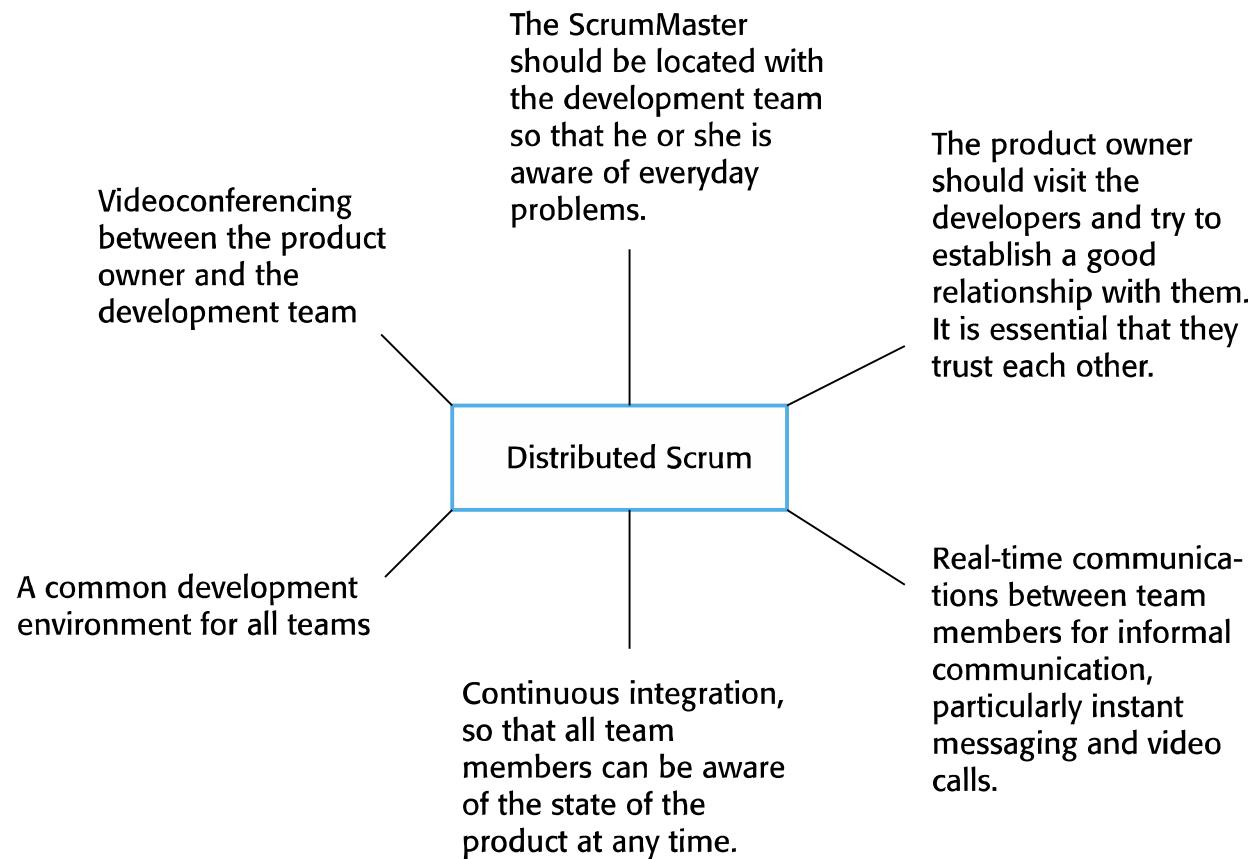
- 'Scrum master' adalah fasilitator yang mengatur pertemuan harian, melacak backlog pekerjaan yang harus diselesaikan, mencatat keputusan, mengukur kemajuan terhadap backlog dan berkomunikasi dengan pelanggan dan manajemen di luar tim.
- Seluruh tim menghadiri pertemuan harian singkat (Scrums) di mana semua anggota tim berbagi informasi, menggambarkan kemajuan mereka sejak pertemuan terakhir, masalah yang muncul dan apa yang direncanakan untuk hari berikutnya.
 - Ini berarti bahwa setiap orang dalam tim mengetahui apa yang sedang terjadi dan, jika masalah muncul, dapat merencanakan kembali pekerjaan jangka pendek untuk mengatasinya.

Manfaat Scrum



- Produk dipecah menjadi satu set potongan yang dapat dikelola dan dimengerti.
- Persyaratan yang tidak stabil tidak menghambat kemajuan.
- Seluruh tim memiliki visibilitas segalanya dan akibatnya komunikasi tim ditingkatkan.
- Pelanggan melihat pengiriman kenaikan tepat waktu dan mendapatkan umpan balik tentang cara kerja produk.
- Kepercayaan antara pelanggan dan pengembang dibangun dan budaya positif diciptakan di mana setiap orang mengharapkan proyek berhasil.

Scrum Terdistribusi





Menskalakan metode tangkas

Menskalakan metode tangkas



- Metode tangkas telah terbukti berhasil untuk proyek berukuran kecil dan menengah yang dapat dikembangkan oleh tim kecil yang berlokasi bersama.
- Kadang-kadang dikatakan bahwa keberhasilan metode ini datang karena komunikasi yang lebih baik yang dimungkinkan ketika semua orang bekerja sama.
- Meningkatkan metode tangkas melibatkan perubahan ini untuk mengatasi proyek yang lebih besar dan lebih lama di mana ada beberapa tim pengembangan, mungkin bekerja di lokasi yang berbeda.

Memperluas dan meningkatkan



- 'Scaling up' berkaitan dengan penggunaan metode tangkas untuk mengembangkan sistem perangkat lunak besar yang tidak dapat dikembangkan oleh tim kecil.
 - 'Scaling out' berkaitan dengan bagaimana metode tangkas dapat diperkenalkan di seluruh organisasi besar dengan pengalaman pengembangan perangkat lunak selama bertahun-tahun.
- Saat menskalakan metode tangkas, penting untuk mempertahankan dasar-dasar tangkas:
- Perencanaan yang fleksibel, rilis sistem yang sering, integrasi berkelanjutan, pengembangan yang didorong oleh pengujian, dan komunikasi tim yang baik.

Masalah praktis dengan metode tangkas



- Informalitas pengembangan tangkas tidak sesuai dengan pendekatan hukum untuk definisi kontrak yang umum digunakan di perusahaan besar.
- Metode tangkas paling tepat untuk pengembangan perangkat lunak baru daripada pemeliharaan perangkat lunak. Namun sebagian besar biaya perangkat lunak di perusahaan besar berasal dari pemeliharaan sistem perangkat lunak yang ada.
- Metode tangkas dirancang untuk tim kecil yang berlokasi bersama namun banyak pengembangan perangkat lunak sekarang melibatkan tim terdistribusi di seluruh dunia.

Masalah kontrak



- Sebagian besar kontrak perangkat lunak untuk sistem kustom didasarkan pada spesifikasi, yang menetapkan apa yang harus diterapkan oleh pengembang sistem untuk pelanggan sistem.
- Namun, ini menghalangi spesifikasi dan pengembangan interleaving seperti norma dalam pengembangan tangkas.
- Diperlukan kontrak yang membayar waktu pengembang daripada fungsionalitas.
 - Namun, ini dipandang sebagai risiko tinggi banyak departemen hukum saya karena apa yang harus disampaikan tidak dapat dijamin.

Metode tangkas dan pemeliharaan perangkat lunak



-Sebagian besar organisasi menghabiskan lebih banyak untuk memelihara perangkat lunak yang ada daripada yang mereka lakukan untuk pengembangan perangkat lunak baru. Jadi, jika metode tangkas ingin berhasil, mereka harus mendukung pemeliharaan serta pengembangan asli.

-Dua masalah utama:

- Apakah sistem yang dikembangkan menggunakan pendekatan tangkas dapat dipelihara, dengan penekanan pada proses pengembangan untuk meminimalkan dokumentasi formal?
- Dapatkah metode tangkas digunakan secara efektif untuk mengembangkan sistem dalam menanggapi permintaan perubahan pelanggan?

-Masalah mungkin timbul jika tim pengembangan asli tidak dapat dipertahankan.

Pemeliharaan tangkas



-Masalah utama adalah:

- Kurangnya dokumentasi produk
- Menjaga pelanggan terlibat dalam proses pengembangan
- Menjaga kesinambungan tim pengembang

-Pengembangan tangkas bergantung pada tim pengembangan yang mengetahui dan memahami apa yang harus dilakukan.

-Untuk sistem yang tahan lama, ini adalah masalah nyata karena pengembang asli tidak akan selalu bekerja pada sistem.

Metode tangkas dan digerakkan oleh rencana



-Sebagian besar proyek menyertakan elemen proses yang digerakkan oleh rencana dan tangkas. Memutuskan saldo tergantung pada:

- Apakah penting untuk memiliki spesifikasi dan desain yang sangat detail sebelum beralih ke implementasi? Jika demikian, Anda mungkin perlu menggunakan pendekatan berbasis rencana.
- Apakah strategi pengiriman tambahan, di mana Anda mengirimkan perangkat lunak kepada pelanggan dan mendapatkan umpan balik yang cepat dari mereka, realistik? Jika demikian, pertimbangkan untuk menggunakan metode tangkas.
- Seberapa besar sistem yang sedang dikembangkan? Metode tangkas paling efektif ketika sistem dapat dikembangkan dengan tim kecil yang ditempatkan bersama yang dapat berkomunikasi secara informal. Ini mungkin tidak mungkin untuk sistem besar yang membutuhkan tim pengembangan yang lebih besar sehingga pendekatan berbasis rencana mungkin harus digunakan.

Prinsip tangkas dan praktik organisasi



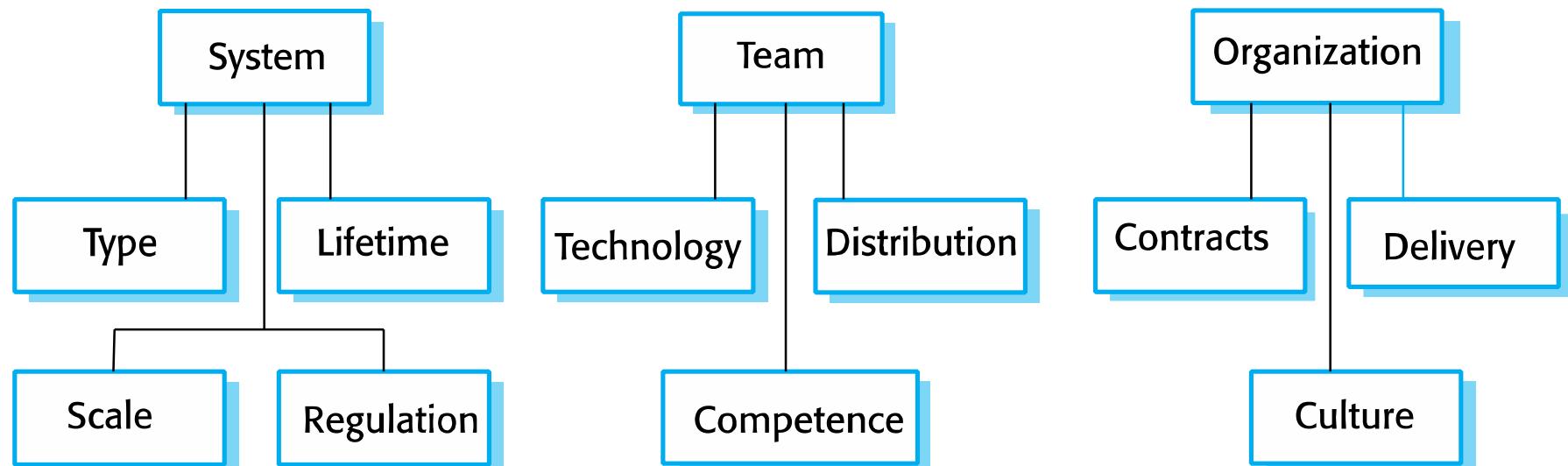
Prinsip	Praktek
Keterlibatan pelanggan	<p>Ini tergantung pada memiliki pelanggan yang bersedia dan mampu menghabiskan waktu dengan tim pengembangan dan yang dapat mewakili semua pemangku kepentingan sistem. Seringkali, perwakilan pelanggan memiliki tuntutan lain pada waktu mereka dan tidak dapat memainkan peran penuh dalam pengembangan perangkat lunak.</p> <p>Di mana ada pemangku kepentingan eksternal, seperti regulator, sulit untuk mewakili pandangan mereka kepada tim tangkas.</p>
Rangkullah perubahan	<p>Memprioritaskan perubahan bisa sangat sulit, terutama dalam sistem yang memiliki banyak pemangku kepentingan. Biasanya, setiap pemangku kepentingan memberikan prioritas yang berbeda untuk perubahan yang berbeda.</p>
Pengiriman tambahan	<p>Pengulangan yang cepat dan perencanaan jangka pendek untuk pengembangan tidak selalu sesuai dengan siklus perencanaan jangka panjang dari perencanaan bisnis dan pemasaran. Manajer pemasaran mungkin perlu mengetahui fitur produk apa beberapa bulan sebelumnya untuk mempersiapkan kampanye pemasaran yang efektif.</p>

Prinsip tangkas dan praktik organisasi



Prinsip	Praktek
Pertahankan kesederhanaan	Di bawah tekanan dari jadwal pengiriman, anggota tim mungkin tidak punya waktu untuk melakukan penyederhanaan sistem yang diinginkan.
Orang tidak memproses	Anggota tim individu mungkin tidak memiliki kepribadian yang cocok untuk keterlibatan intens yang khas dari metode tangkas, dan karena itu mungkin tidak berinteraksi dengan baik dengan anggota tim lainnya.

Faktor gesit dan berbasis rencana



Masalah sistem



-Seberapa besar sistem yang dikembangkan?

- Metode tangkas paling efektif untuk tim yang relatif kecil yang dapat berkomunikasi secara informal.

-Jenis sistem apa yang sedang dikembangkan?

- Sistem yang membutuhkan banyak analisis sebelum implementasi membutuhkan desain yang cukup rinci untuk melakukan analisis ini.

-Berapa umur sistem yang diharapkan?

- Sistem yang tahan lama memerlukan dokumentasi untuk mengomunikasikan niat pengembang sistem kepada tim dukungan.

-Apakah sistem tunduk pada regulasi eksternal?

- Jika suatu sistem diatur, Anda mungkin akan diminta untuk membuat dokumentasi terperinci sebagai bagian dari kasus keamanan sistem.

Orang dan tim



-Seberapa baik para desainer dan programmer dalam tim pengembangan?

- Kadang-kadang dikatakan bahwa metode tangkas membutuhkan tingkat keterampilan yang lebih tinggi daripada pendekatan berbasis rencana di mana pemrogram hanya menerjemahkan desain terperinci ke dalam kode.

-Bagaimana tim pengembangan diatur?

- Dokumen desain mungkin diperlukan jika tim didistribusikan.

-Teknologi pendukung apa yang tersedia?

- Dukungan IDE untuk visualisasi dan analisis program sangat penting jika dokumentasi desain tidak tersedia.

Masalah organisasi



- Organisasi rekayasa tradisional memiliki budaya pengembangan berbasis rencana, karena ini adalah norma dalam rekayasa.
- Apakah praktik organisasi standar untuk mengembangkan spesifikasi sistem yang terperinci?
- Akankah perwakilan pelanggan tersedia untuk memberikan umpan balik tentang peningkatan sistem?
- Bisakah pengembangan tangkas informal masuk ke dalam budaya organisasi dokumentasi terperinci?

Metode tangkas untuk sistem besar



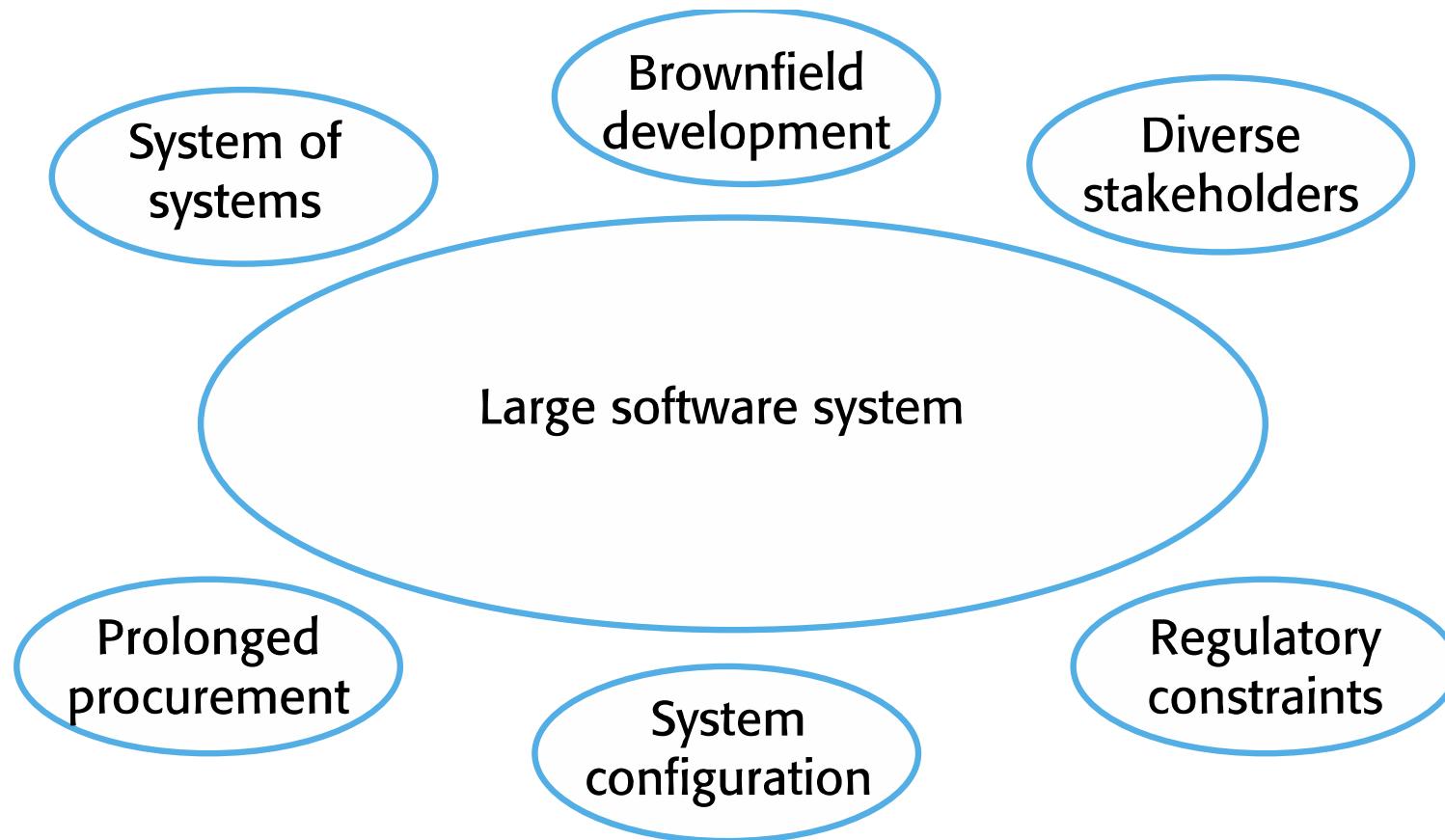
- Sistem yang besar biasanya merupakan kumpulan dari sistem komunikasi yang terpisah, di mana tim yang terpisah mengembangkan setiap sistem. Seringkali, tim ini bekerja di tempat yang berbeda, terkadang di zona waktu yang berbeda.
- Sistem besar adalah 'sistem brownfield', yaitu mereka termasuk dan berinteraksi dengan sejumlah sistem yang ada. Banyak persyaratan sistem yang berkaitan dengan interaksi ini sehingga tidak benar-benar memberikan fleksibilitas dan pengembangan tambahan.
- Di mana beberapa sistem terintegrasi untuk membuat sistem, sebagian besar pengembangan berkaitan dengan konfigurasi sistem daripada pengembangan kode asli.

Pengembangan sistem besar

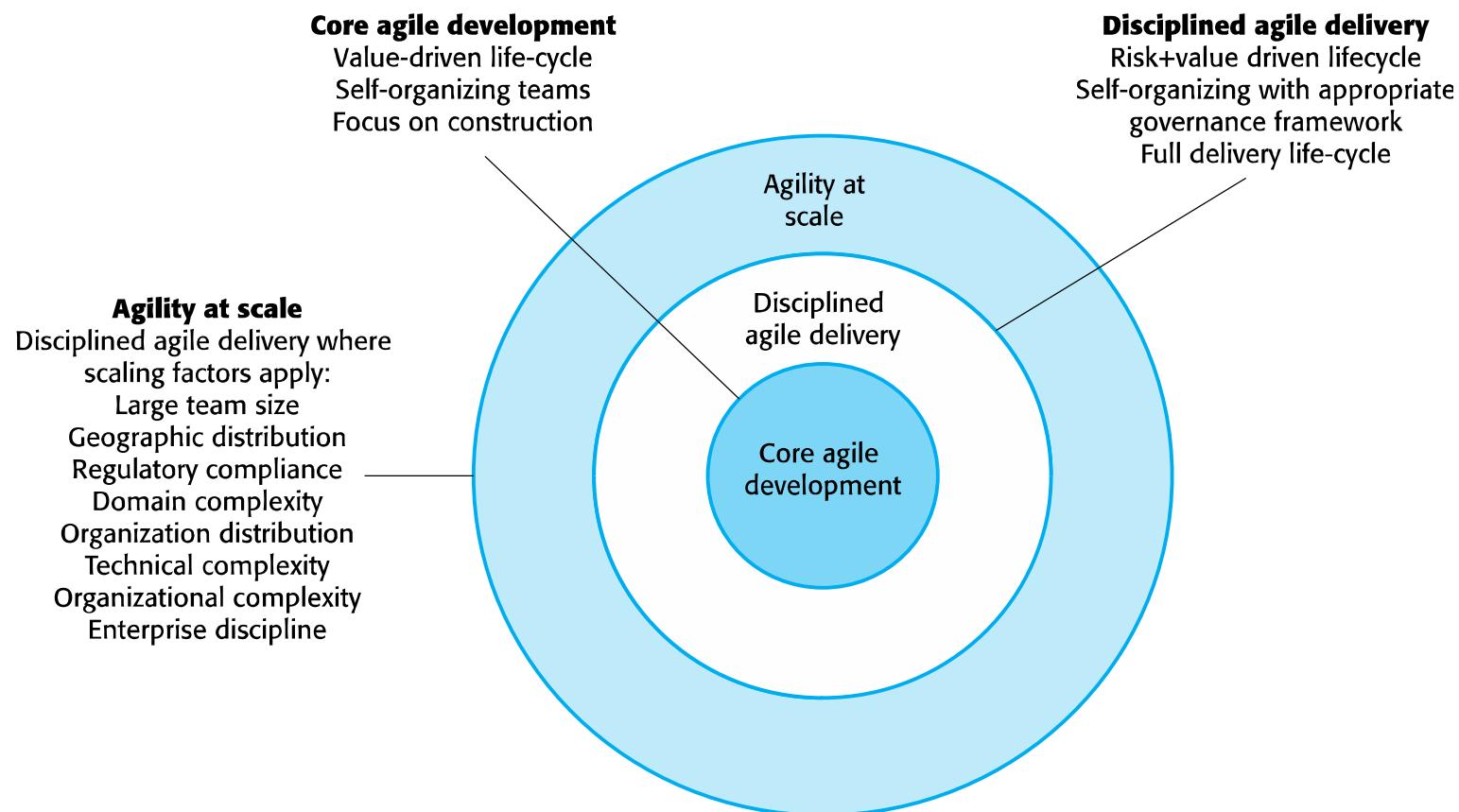


- Sistem besar dan proses pengembangannya sering dibatasi oleh aturan dan regulasi eksternal yang membatasi cara pengembangannya.
- Sistem yang besar memiliki waktu pengadaan dan pengembangan yang lama. Sulit untuk mempertahankan tim yang koheren yang tahu tentang sistem selama periode itu karena, mau tidak mau, orang beralih ke pekerjaan dan proyek lain.
- Sistem besar biasanya memiliki beragam pemangku kepentingan. Praktis tidak mungkin melibatkan semua pemangku kepentingan yang berbeda ini dalam proses pembangunan.

Faktor dalam sistem besar



Model skala kelincahan IBM



Meningkatkan ke sistem besar



- Pendekatan yang sepenuhnya inkremental untuk rekayasa persyaratan tidak mungkin dilakukan.
- Tidak boleh ada satu pemilik produk atau perwakilan pelanggan.
- Untuk pengembangan sistem yang besar, tidak mungkin hanya berfokus pada kode sistem.
- Mekanisme komunikasi lintas tim harus dirancang dan digunakan.
- Integrasi berkelanjutan praktis tidak mungkin. Namun, sangat penting untuk mempertahankan pembuatan sistem yang sering dan rilis sistem secara teratur.

Scrum multi-tim



-*Replikasi peran*

- Setiap tim memiliki Pemilik Produk untuk komponen kerja mereka dan ScrumMaster.

-*Arsitek produk*

- Setiap tim memilih arsitek produk dan arsitek ini berkolaborasi untuk merancang dan mengembangkan arsitektur sistem secara keseluruhan.

-*Perataan rilis*

- Tanggal rilis produk dari masing-masing tim diselaraskan sehingga dihasilkan sistem yang dapat dibuktikan dan lengkap.

-*Scrum dari Scrum*

- Ada Scrum of Scrums harian di mana perwakilan dari setiap tim bertemu untuk membahas kemajuan dan merencanakan pekerjaan yang harus dilakukan.

Metode tangkas di seluruh organisasi



- Manajer proyek yang tidak memiliki pengalaman metode tangkas mungkin enggan menerima risiko pendekatan baru.
- Organisasi besar sering kali memiliki prosedur dan standar kualitas yang diharapkan untuk diikuti oleh semua proyek dan, karena sifat birokrasinya, ini cenderung tidak sesuai dengan metode tangkas.
- Metode tangkas tampaknya bekerja paling baik ketika anggota tim memiliki tingkat keterampilan yang relatif tinggi. Namun, dalam organisasi besar, kemungkinan besar ada berbagai keterampilan dan kemampuan.
- Mungkin ada resistensi budaya terhadap metode tangkas, terutama di organisasi-organisasi yang memiliki sejarah panjang menggunakan proses rekayasa sistem konvensional.

Poin-poin penting



-Metode Agile adalah metode pengembangan tambahan yang berfokus pada pengembangan perangkat lunak yang cepat, rilis perangkat lunak yang sering, mengurangi biaya proses dengan meminimalkan dokumentasi dan menghasilkan kode berkualitas tinggi.

-Praktik pengembangan tangkas meliputi:

- Cerita pengguna untuk spesifikasi sistem
- Rilis perangkat lunak yang sering,
- Peningkatan perangkat lunak berkelanjutan
- Pengembangan uji-pertama
- Partisipasi pelanggan dalam tim pengembangan.

Poin-poin penting



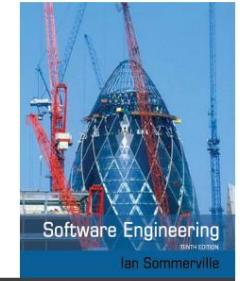
-Scrum adalah metode tangkas yang menyediakan kerangka kerja manajemen proyek.

- Ini berpusat pada serangkaian sprint, yang merupakan periode waktu tetap ketika peningkatan sistem dikembangkan.

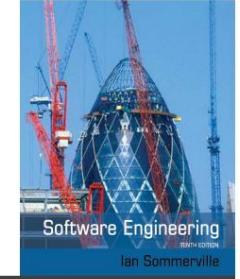
-Banyak metode pengembangan praktis yang merupakan campuran dari pengembangan berbasis rencana dan tangkas.

-Menskalakan metode tangkas untuk sistem besar itu sulit.

- Sistem besar membutuhkan desain awal dan beberapa dokumentasi dan praktik organisasi mungkin bertentangan dengan informalitas pendekatan tangkas.



Bab 4 – Rekayasa Persyaratan



Topik yang dibahas

Persyaratan fungsional dan non-fungsional

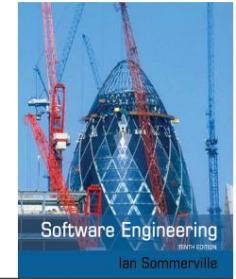
Proses rekayasa persyaratan

Persyaratan elisitasi

Spesifikasi persyaratan

Validasi persyaratan

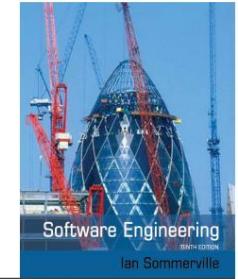
Persyaratan berubah



Rekayasa persyaratan

Proses menetapkan layanan yang pelanggan dibutuhkan dari suatu sistem dan kendala di mana ia beroperasi dan dikembangkan.

Persyaratan sistem adalah deskripsi layanan sistem dan batasan yang dihasilkan selama proses rekayasa persyaratan.

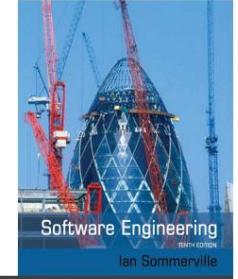


Apa itu persyaratan?

Ini dapat berkisar dari pernyataan abstrak tingkat tinggi dari layanan atau batasan sistem hingga spesifikasi fungsional matematis yang terperinci.

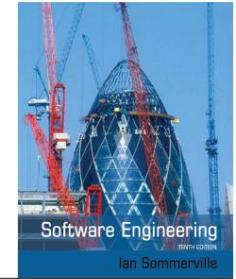
Ini tidak dapat dihindari karena persyaratan dapat berfungsi ganda fungsi

- ÿ Mungkin menjadi dasar untuk tawaran kontrak - oleh karena itu harus terbuka untuk interpretasi;
- ÿ Dapat menjadi dasar untuk kontrak itu sendiri - oleh karena itu harus didefinisikan secara rinci;
- ÿ Kedua pernyataan ini dapat disebut persyaratan.



Abstraksi persyaratan (Davis)

“Jika sebuah perusahaan ingin memberikan kontrak untuk proyek pengembangan perangkat lunak yang besar, ia harus mendefinisikan kebutuhannya dengan cara yang cukup abstrak sehingga solusi tidak ditentukan sebelumnya. Persyaratan harus ditulis sehingga beberapa kontraktor dapat menawar kontrak, menawarkan, mungkin, cara berbeda untuk memenuhi kebutuhan organisasi klien. Setelah kontrak diberikan, kontraktor harus menulis definisi sistem untuk klien secara lebih rinci sehingga klien memahami dan dapat memvalidasi apa yang akan dilakukan perangkat lunak. Kedua dokumen ini dapat disebut sebagai dokumen persyaratan untuk sistem.”



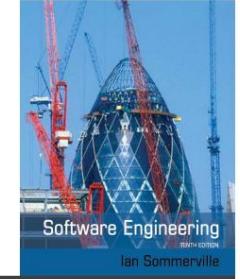
Jenis persyaratan:

Persyaratan pengguna

- ŷ Pernyataan dalam bahasa alami ditambah diagram layanan yang disediakan sistem dan kendala operasionalnya. Ditulis untuk pelanggan.

Persyaratan sistem

- ŷ Sebuah dokumen terstruktur yang menguraikan deskripsi rinci tentang **fungsi sistem, layanan dan kendala operasional.**
Mendefinisikan apa yang harus dilaksanakan sehingga dapat menjadi bagian dari kontrak antara klien dan kontraktor.



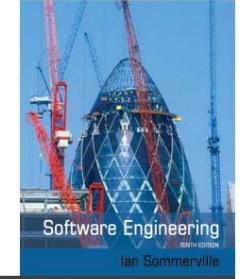
Persyaratan pengguna dan sistem (contoh perawatan)

Definisi requirement specification

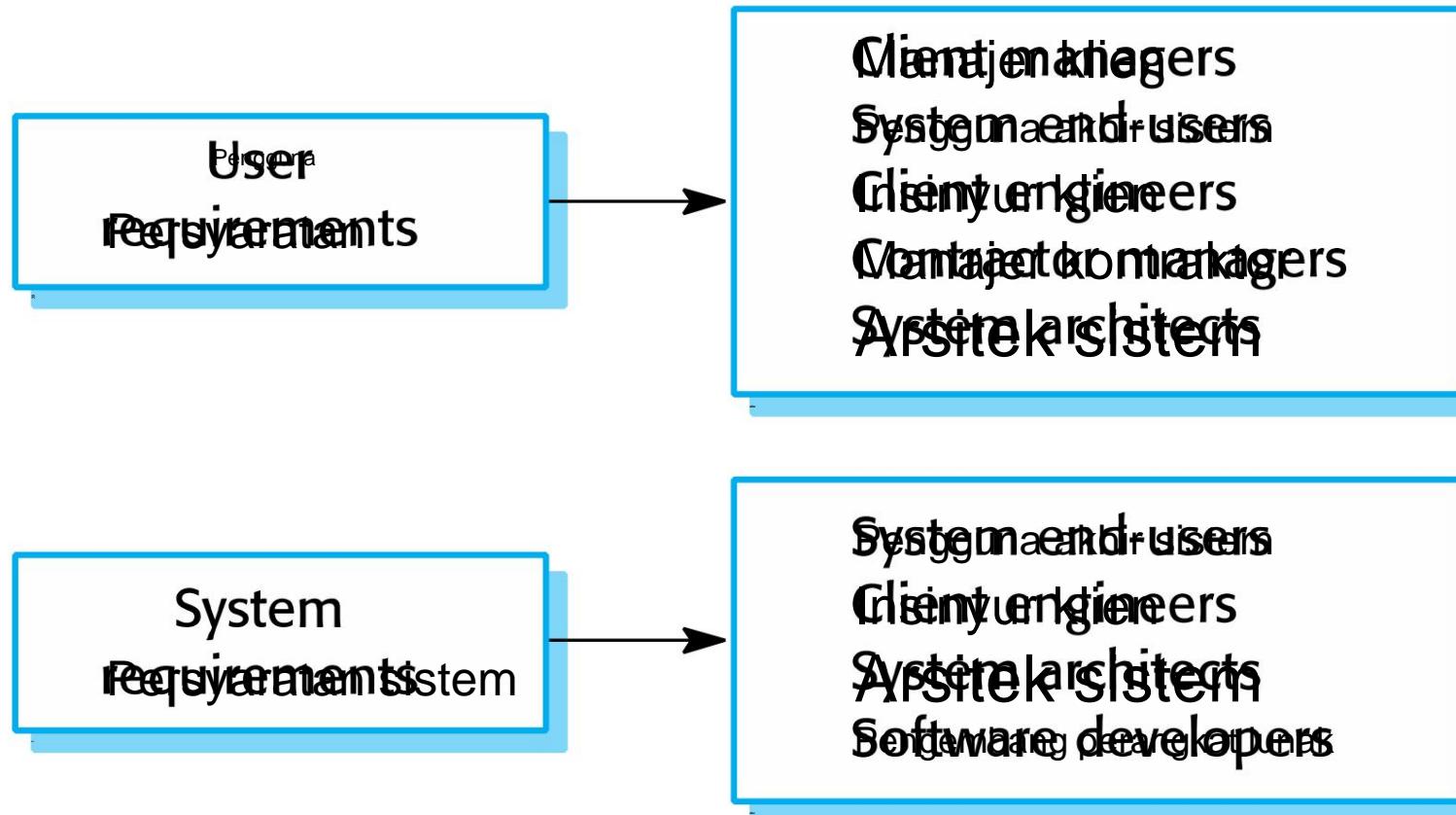
1. Sistem Medis adalah sistem yang dapat melaporkan hanya jika ada empat laporan seperti jumlah obat yang diberikan pada pasien dalam satuan kardus selama dua minggu.

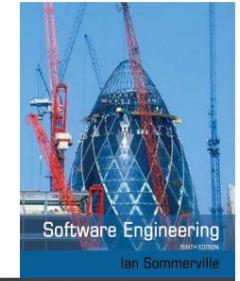
Sistemika persyaratan spesifikasi

- 1.1 Pada hari kerja setiap bulan, angka rata-rata obat yang dibutuhkan, jumlah obat yang diberikan pada pasien seharusnya tidak melebihi 1750 di the last two weeks.
- 1.2 Sistem akan melaporkan bahwa rata-rata jumlah obat yang diberikan pada pasien selama 750 di the last two weeks.
- 1.3 Laporan harus berisi tentang obat-obatan klinik dan status obat antiviral dengan bantuan teknologi, jumlah obat resep, jumlah obat yang tidak diperlukan dan jumlah obat yang diberikan pada pasien.
- 1.4 Jika obat tersedia dalam beberapa jenis (misalnya, 200mg/200mg) seharusnya hanya dibuat untuk satu dosis.
- 1.5 Akses ke obat yang diberikan shall be restricted hanya kepada user yang terdaftar dan mempunyai hak akses control list.



Pembaca dari berbagai jenis spesifikasi persyaratan



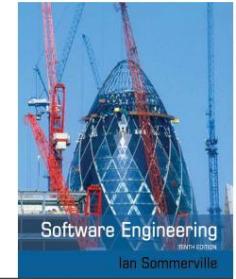


Pemangku kepentingan sistem

Setiap orang atau organisasi yang terpengaruh oleh sistem dalam beberapa cara dan siapa yang memiliki kepentingan yang sah

Jenis pemangku kepentingan

- ÿ Pengguna akhir
- ÿ Manajer sistem
- ÿ Pemilik sistem
- ÿ Pemangku kepentingan eksternal



Contoh : Pemangku kepentingan dalam sistem Mentcare

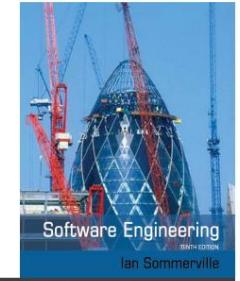
Pasien yang informasinya dicatat dalam sistem.

Dokter yang bertanggung jawab untuk menilai dan mengobati pasien.

Perawat yang mengoordinasikan konsultasi dengan dokter dan memberikan beberapa perawatan.

Resepsionis medis yang mengelola janji temu pasien.

Staf TI yang bertanggung jawab untuk menginstal dan memelihara sistem.

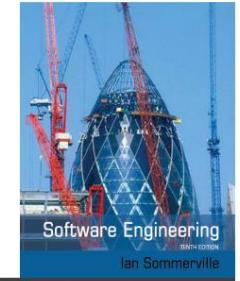


Pemangku kepentingan dalam sistem Mentcare

Seorang manajer etika medis yang harus memastikan bahwa sistem memenuhi pedoman etika saat ini untuk perawatan pasien.

Manajer perawatan kesehatan yang memperoleh informasi manajemen dari sistem.

Staf rekam medis yang bertanggung jawab untuk memastikan bahwa sistem informasi dapat dipelihara dan dipelihara, dan bahwa prosedur penyimpanan catatan telah diterapkan dengan benar.



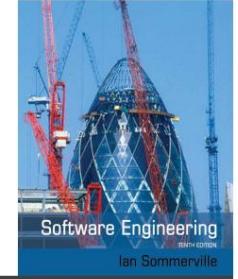
Metode dan persyaratan tangkas

Banyak metode tangkas berpendapat bahwa menghasilkan detail persyaratan sistem adalah buang-buang waktu karena persyaratan berubah begitu cepat.

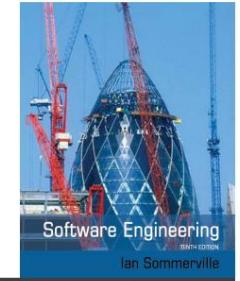
Oleh karena itu, dokumen persyaratan selalu keluar tanggal.

Metode Agile biasanya **menggunakan rekayasa persyaratan tambahan dan dapat mengekspresikan persyaratan sebagai 'cerita pengguna'** (**dibahas** dalam Bab 3).

Ini praktis untuk sistem bisnis tetapi bermasalah untuk sistem yang memerlukan analisis pra-pengiriman (misalnya sistem kritis) atau sistem yang dikembangkan oleh beberapa tim.



Persyaratan fungsional dan non-fungsional



Persyaratan fungsional dan non-fungsional

Persyaratan fungsional

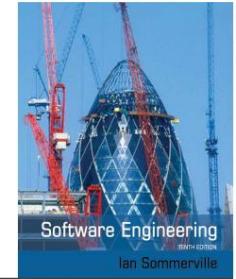
- ÿ Pernyataan **layanan yang harus disediakan sistem**, bagaimana sistem harus bereaksi terhadap input tertentu dan bagaimana sistem harus berperilaku dalam situasi tertentu.
- ÿ Dapat menyatakan apa yang tidak boleh dilakukan oleh sistem.

Persyaratan non-fungsional

- ÿ **Kendala pada layanan atau fungsi yang ditawarkan oleh sistem** seperti kendala waktu, kendala pada proses pengembangan, standar, dll.
- ÿ Sering diterapkan pada sistem secara keseluruhan daripada individu fitur atau layanan.

Persyaratan domain

- ÿ Kendala pada sistem dari domain operasi



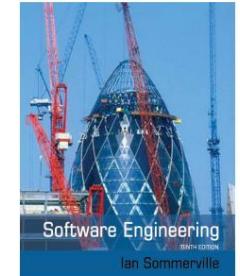
Persyaratan fungsional

Menjelaskan fungsionalitas atau layanan sistem.

Tergantung pada jenis perangkat lunak, pengguna yang diharapkan, dan jenis sistem tempat perangkat lunak tersebut digunakan.

Persyaratan pengguna fungsional mungkin merupakan pernyataan tingkat tinggi tentang apa yang harus dilakukan sistem.

Persyaratan sistem fungsional harus menggambarkan layanan sistem secara rinci.

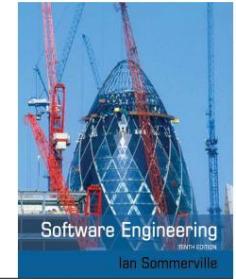


Contoh : Sistem Mentcare: kebutuhan fungsional

Seorang pengguna harus dapat mencari daftar janji temu untuk semua klinik.

Sistem akan menghasilkan setiap hari, untuk setiap klinik, a daftar pasien yang diharapkan untuk menghadiri janji hari itu.

Setiap anggota staf yang menggunakan sistem harus diidentifikasi secara unik dengan 8 digit nomor karyawannya.



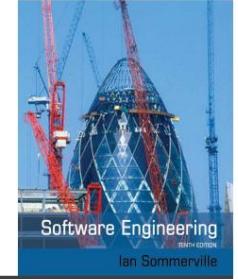
Ketidaktepatan persyaratan

Masalah muncul ketika persyaratan fungsional tidak dinyatakan secara tepat.

Persyaratan ambigu dapat ditafsirkan berbeda cara oleh pengembang dan pengguna.

Pertimbangkan istilah 'pencarian' dalam persyaratan 1

- ÿ Niat pengguna – mencari nama pasien di semua janji temu di semua klinik;
- ÿ Interpretasi pengembang – mencari nama pasien di klinik individu. Pengguna memilih klinik kemudian mencari.



Kelengkapan dan konsistensi persyaratan

Pada prinsipnya, persyaratan harus lengkap dan konsisten.

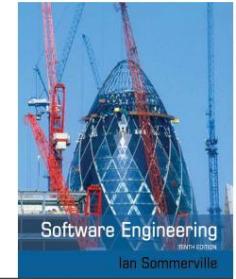
Lengkap

- ÿ Mereka harus mencakup deskripsi semua fasilitas yang diperlukan.

Konsisten

- ÿ Seharusnya tidak ada konflik atau kontradiksi dalam deskripsi dari fasilitas sistem.

Dalam praktiknya, karena sistem dan lingkungan kompleksitas, tidak mungkin untuk menghasilkan dokumen persyaratan yang lengkap dan konsisten.

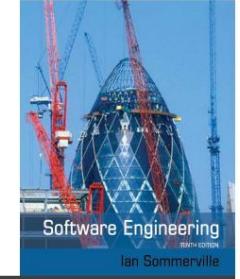


Persyaratan non-fungsional

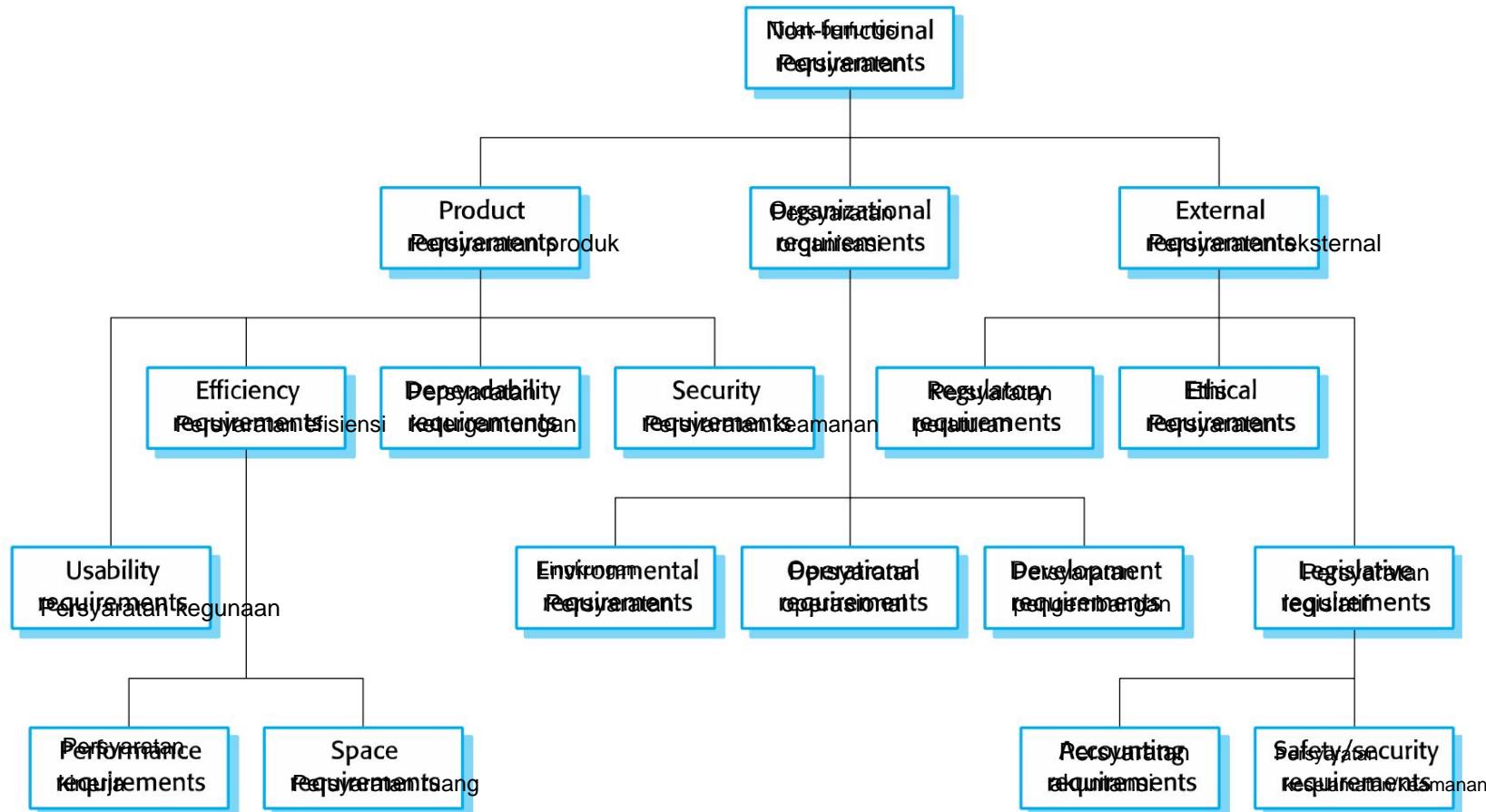
Ini mendefinisikan properti sistem dan kendala misalnya **keandalan, waktu respons, dan persyaratan penyimpanan**. **Kendalanya adalah kemampuan perangkat I/O, representasi sistem, dll.**

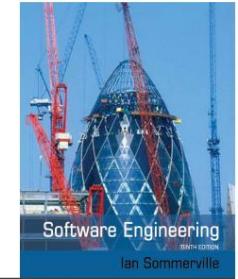
Persyaratan proses juga dapat ditentukan dengan mengamanatkan IDE, bahasa pemrograman, atau metode pengembangan tertentu.

Persyaratan non-fungsional mungkin lebih penting daripada persyaratan fungsional. Jika ini tidak terpenuhi, sistem mungkin tidak berguna.



Jenis kebutuhan nonfungsional



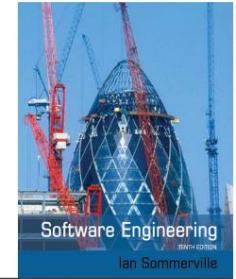


Implementasi persyaratan non-fungsional

Persyaratan non-fungsional dapat mempengaruhi arsitektur keseluruhan sistem daripada komponen individu.

ÿ Misalnya, untuk memastikan bahwa persyaratan kinerja terpenuhi, Anda mungkin harus mengatur sistem untuk meminimalkan komunikasi antar komponen.

Persyaratan non-fungsional tunggal, seperti persyaratan keamanan, dapat menghasilkan sejumlah persyaratan fungsional terkait yang mendefinisikan layanan sistem yang diperlukan. ÿ Ini juga dapat menghasilkan persyaratan yang membatasi persyaratan yang ada.



Klasifikasi non-fungsional

Persyaratan produk

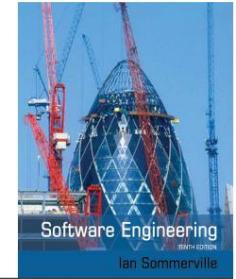
- ŷ **Persyaratan yang menentukan bahwa produk yang dikirim harus berperilaku dengan cara tertentu** misalnya kecepatan eksekusi, keandalan, dll.

Persyaratan organisasi

- ŷ **Persyaratan yang merupakan konsekuensi dari kebijakan dan prosedur organisasi**, misalnya standar proses yang digunakan, persyaratan implementasi, dll.

Persyaratan eksternal

- ŷ **Persyaratan yang muncul dari faktor-faktor yang berada di luar sistem dan proses pengembangannya**, misalnya persyaratan interoperabilitas, persyaratan legislatif, dll.



Contoh kebutuhan nonfungsional dalam Sistem perawatan jiwa

Persyaratan produk

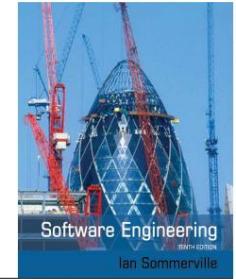
Sistem Mentcare harus tersedia untuk semua klinik selama jam kerja normal (Senin–Jumat, 0830–17.30). Waktu henti dalam jam kerja normal tidak boleh melebihi lima detik dalam satu hari.

Persyaratan organisasi

Pengguna sistem Mentcare harus mengotentikasi diri mereka sendiri menggunakan kartu identitas otoritas kesehatan mereka.

Persyaratan eksternal

Sistem akan menerapkan ketentuan privasi pasien sebagaimana diatur dalam HStan-03-2006-priv.



Tujuan dan persyaratan

Persyaratan non-fungsional mungkin sangat sulit untuk dinyatakan secara tepat dan persyaratan yang tidak tepat mungkin sulit untuk diverifikasi.

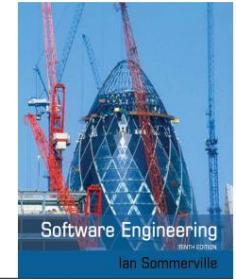
Tujuan

ÿ Tujuan umum pengguna seperti kemudahan penggunaan.

Persyaratan non-fungsional yang dapat diverifikasi

ÿ Suatu pernyataan yang menggunakan beberapa ukuran yang dapat diuji secara objektif.

Sasaran sangat membantu pengembang karena menyampaikan maksud dari pengguna sistem.



Persyaratan kegunaan

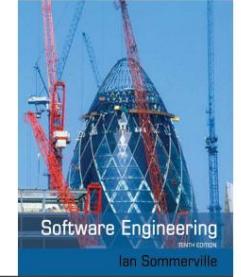
Sistem harus mudah digunakan oleh staf medis dan harus diatur sedemikian rupa sehingga kesalahan pengguna diminimalkan.
(Sasaran)

Staf medis harus dapat menggunakan semua fungsi sistem setelah empat jam pelatihan. Setelah pelatihan ini, jumlah rata-rata kesalahan yang dibuat oleh pengguna berpengalaman tidak boleh melebihi dua per jam penggunaan sistem. (Persyaratan non-fungsional yang dapat diuji)

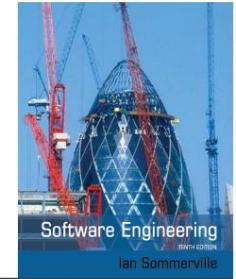


Metrik untuk menentukan persyaratan nonfungsional

Properti	Ukuran
Kecepatan	Transaksi yang diproses/detik Waktu respons pengguna/acara Waktu penyegaran layar
Ukuran	Mbytes Jumlah chip ROM
Kemudahan penggunaan	Waktu pelatihan Jumlah bingkai bantuan
Keandalan	Berarti waktu untuk gagal Probabilitas ketidaktersediaan Tingkat terjadinya kegagalan Ketersediaan
kekokohan	Saatnya memulai kembali setelah gagal Persentase kejadian yang menyebabkan kegagalan Probabilitas korupsi data pada kegagalan
Portabilitas	Persentase pernyataan yang bergantung pada target Jumlah sistem target



Proses rekayasa persyaratan



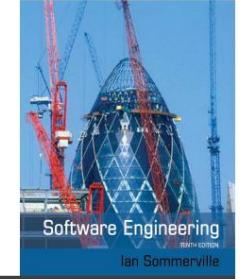
Proses rekayasa persyaratan

Proses yang digunakan untuk RE sangat bervariasi tergantung pada domain aplikasi, orang yang terlibat dan organisasi yang mengembangkan persyaratan.

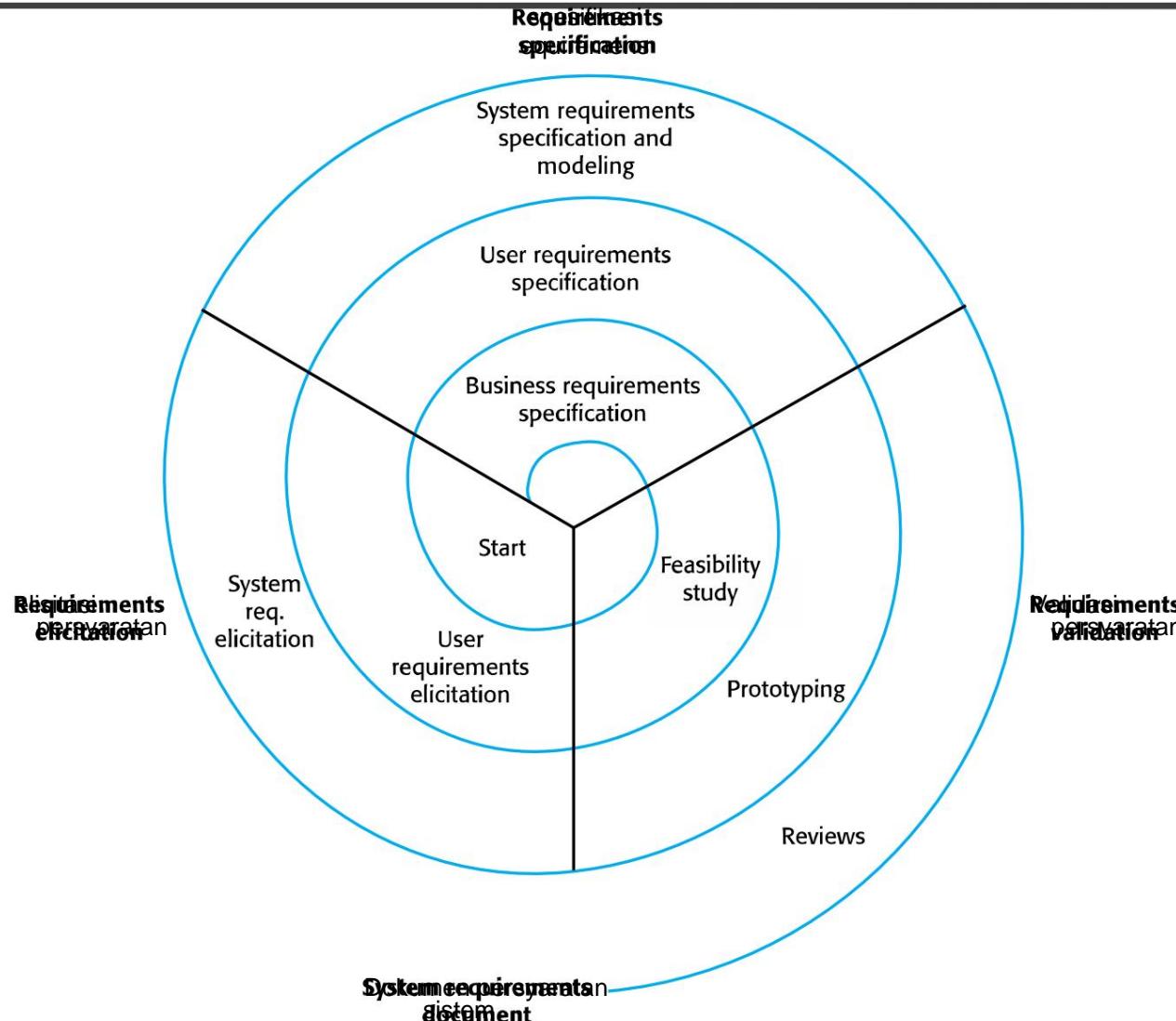
Namun, ada sejumlah aktivitas umum yang umum untuk semua proses

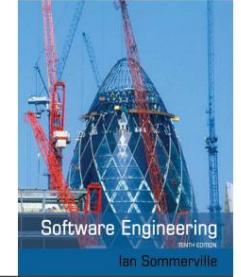
- ÿ Persyaratan elisitasi;
- ÿ Analisis kebutuhan;
- ÿ Validasi persyaratan;
- ÿ Manajemen kebutuhan.

Dalam praktiknya, RE adalah aktivitas berulang di mana ini proses disisipkan.

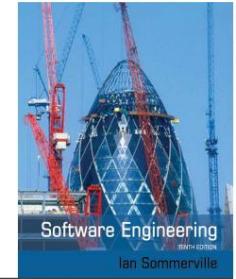


Pandangan spiral dari proses rekayasa persyaratan





Persyaratan elitisasi

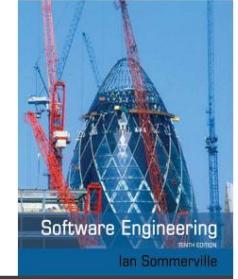


Persyaratan elisitasi dan analisis

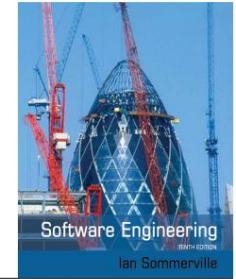
Kadang-kadang disebut elisitasi persyaratan atau penemuan kebutuhan.

Melibatkan staf teknis yang bekerja dengan pelanggan untuk mencari tahu tentang domain aplikasi, layanan yang harus disediakan sistem, dan kendala operasional sistem.

Dapat melibatkan pengguna akhir, manajer, insinyur yang terlibat dalam pemeliharaan, pakar domain, serikat pekerja, dll. Ini disebut *pemangku kepentingan*.



Persyaratan elisitasi

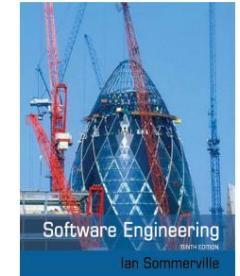


Persyaratan elisitasi

Insinyur perangkat lunak bekerja dengan berbagai sistem pemangku kepentingan untuk mengetahui tentang domain aplikasi, layanan yang harus disediakan sistem, kinerja sistem yang diperlukan, kendala perangkat keras, sistem lain, dll.

Tahapannya meliputi:

- ÿ Penemuan kebutuhan,
- ÿ Persyaratan klasifikasi dan organisasi,
- ÿ Prioritas dan negosiasi persyaratan,
- ÿ Spesifikasi kebutuhan.



Masalah elisitasi persyaratan

Pemangku kepentingan tidak tahu apa yang sebenarnya mereka inginkan.

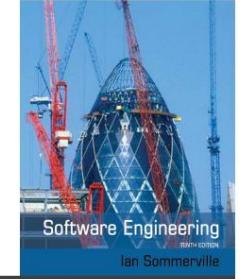
Pemangku kepentingan mengungkapkan persyaratan dalam istilah mereka sendiri.

Pemangku kepentingan yang berbeda mungkin memiliki persyaratan
yang saling bertentangan.

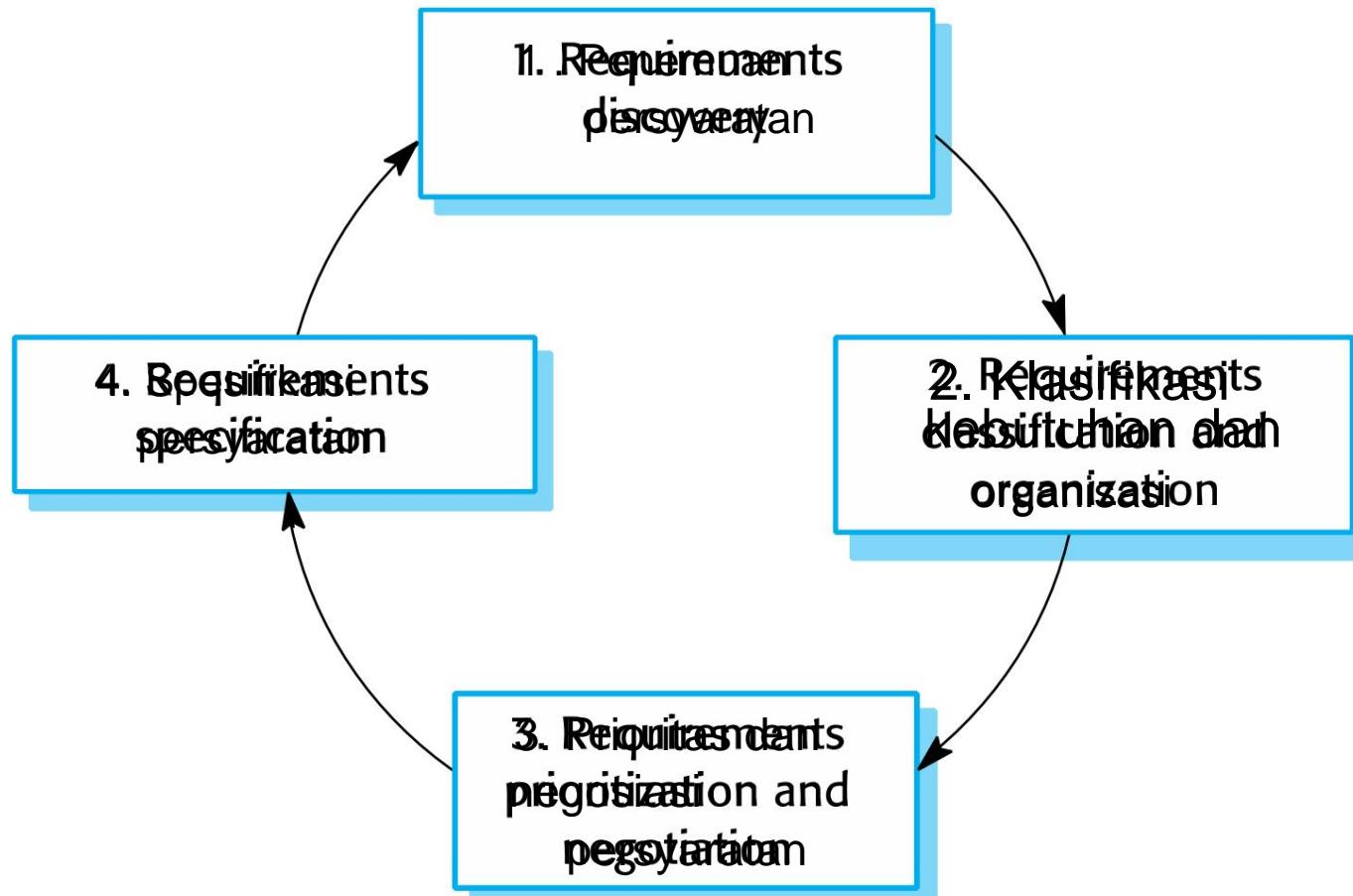
Faktor organisasi dan politik dapat mempengaruhi
persyaratan sistem.

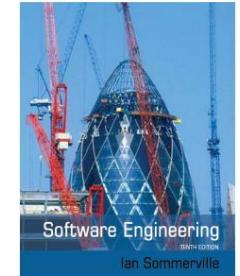
Persyaratan berubah selama proses analisis.

Pemangku kepentingan baru dapat muncul dan
lingkungan bisnis dapat berubah.



Proses elisitasi dan analisis persyaratan





Proses kegiatan

Penemuan persyaratan

- ÿ Berinteraksi dengan pemangku kepentingan untuk menemukan kebutuhan mereka.
- Persyaratan domain juga ditemukan pada tahap ini.

Klasifikasi dan organisasi persyaratan

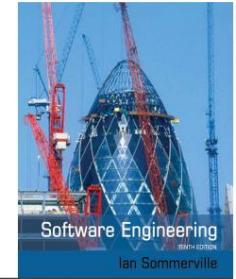
- ÿ Mengelompokkan persyaratan terkait dan mengurnya menjadi koheren cluster.

Prioritas dan negosiasi

- ÿ Memprioritaskan persyaratan dan menyelesaikan konflik persyaratan.

Spesifikasi persyaratan

- ÿ Persyaratan didokumentasikan dan dimasukkan ke putaran berikutnya spiralnya.

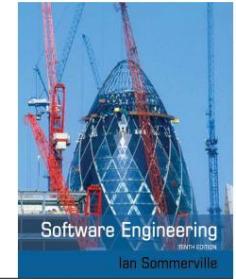


Penemuan persyaratan

Proses pengumpulan informasi tentang sistem yang dibutuhkan dan yang ada dan menyaring kebutuhan pengguna dan sistem dari informasi ini.

Interaksi dengan pemangku kepentingan sistem dari manajer hingga regulator eksternal.

Sistem biasanya memiliki berbagai pemangku kepentingan.



Wawancara

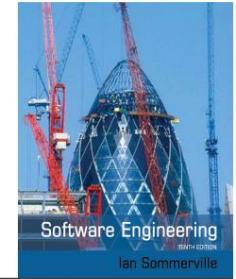
Wawancara formal atau informal dengan pemangku kepentingan adalah bagian dari sebagian besar proses EBT.

Jenis-jenis wawancara

- ÿ Wawancara tertutup berdasarkan daftar pertanyaan yang telah ditentukan sebelumnya
- ÿ Wawancara terbuka di mana berbagai isu dieksplorasi dengan pemangku kepentingan.

Wawancara yang efektif

- ÿ Bersikap terbuka, hindari gagasan yang sudah terbentuk sebelumnya tentang persyaratan dan bersedia mendengarkan pemangku kepentingan.
- ÿ Minta orang yang diwawancarai untuk memulai diskusi menggunakan a pertanyaan batu loncatan, proposal persyaratan, atau dengan bekerja sama pada sistem prototipe.



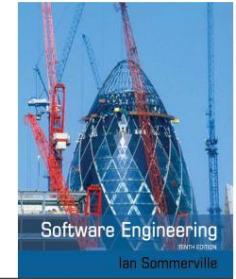
Wawancara dalam praktek

Biasanya merupakan campuran wawancara tertutup dan terbuka.

Wawancara baik untuk mendapatkan pemahaman menyeluruh tentang apa yang dilakukan pemangku kepentingan dan bagaimana mereka dapat berinteraksi dengan sistem.

Pewawancara harus berpikiran terbuka tanpa gagasan sebelumnya tentang apa yang harus dilakukan sistem

Anda perlu meminta penggunaan untuk berbicara tentang sistem dengan menyarankan persyaratan daripada hanya menanyakan apa yang mereka inginkan.

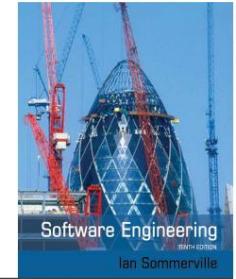


Masalah dengan wawancara

Spesialis aplikasi dapat menggunakan bahasa untuk menggambarkan pekerjaan mereka yang tidak mudah dipahami oleh para insinyur persyaratan.

Wawancara tidak baik untuk memahami persyaratan domain

- ÿ Persyaratan insinyur tidak dapat memahami terminologi domain tertentu;
- ÿ Beberapa pengetahuan domain begitu familiar sehingga orang merasa sulit untuk mengartikulasikan atau berpikir bahwa itu tidak layak untuk diartikulasikan.



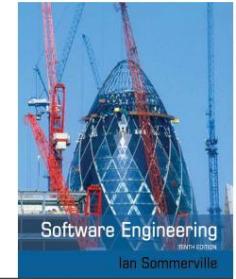
Etnografi

Seorang ilmuwan sosial menghabiskan banyak waktu untuk mengamati dan menganalisis bagaimana orang benar-benar bekerja.

Orang tidak perlu menjelaskan atau mengartikulasikan pekerjaan mereka.

Faktor sosial dan organisasi yang penting mungkin diamati.

Studi etnografi telah menunjukkan bahwa pekerjaan biasanya lebih kaya dan lebih kompleks daripada yang disarankan oleh model sistem sederhana.



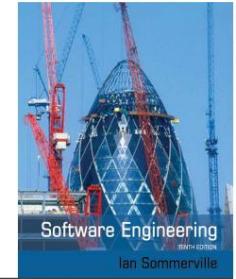
Lingkup etnografi

Persyaratan yang diturunkan dari cara orang benar-benar bekerja daripada cara saya yang definisi prosesnya menyarankan bahwa mereka seharusnya bekerja.

Persyaratan yang berasal dari kerjasama dan kesadaran akan aktivitas orang lain.

- ÿ Kesadaran tentang apa yang orang lain lakukan mengarah pada perubahan dalam cara-cara di mana kita melakukan sesuatu.

Etnografi efektif untuk memahami yang ada proses tetapi tidak dapat mengidentifikasi fitur baru yang harus ditambahkan ke sistem.



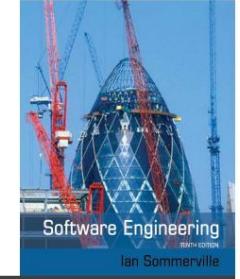
Etnografi terfokus

Dikembangkan dalam proyek yang mempelajari kontrol lalu lintas udara proses

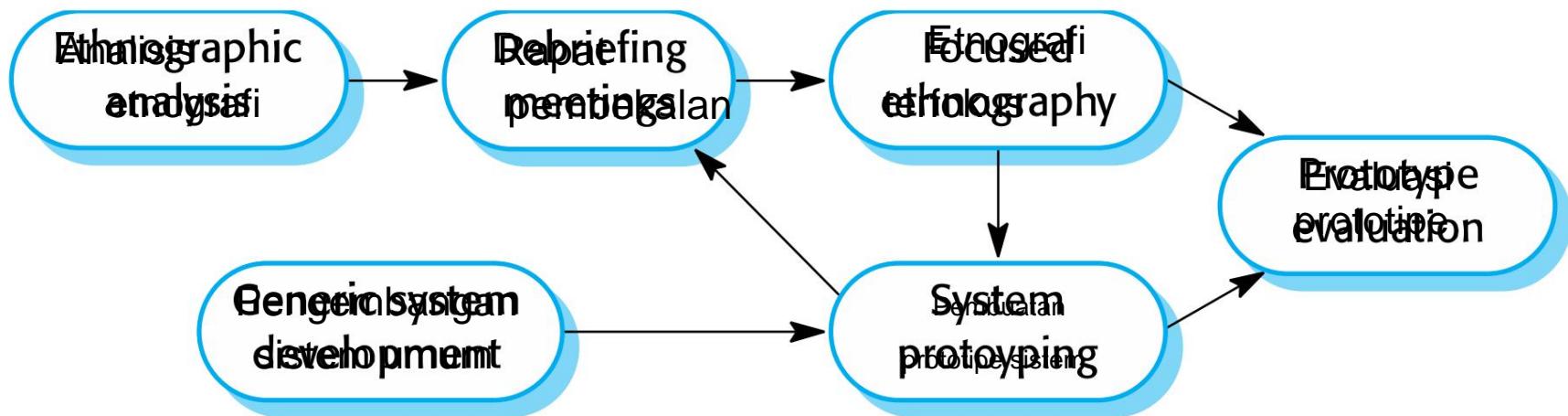
Menggabungkan etnografi dengan prototyping

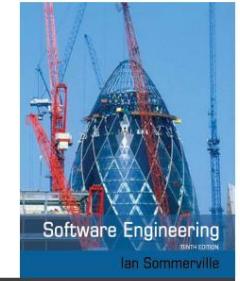
Pengembangan prototipe menghasilkan pertanyaan yang tidak terjawab yang memfokuskan analisis etnografi.

Masalah dengan etnografi adalah bahwa ia mempelajari praktik-praktek yang ada yang mungkin memiliki beberapa dasar sejarah yang tidak lagi relevan.



Etnografi dan pembuatan prototipe untuk analisis kebutuhan



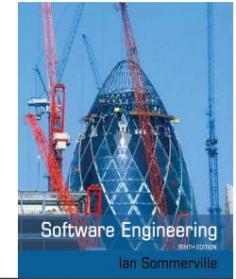


Cerita dan skenario

Skenario dan cerita pengguna adalah contoh nyata tentang bagaimana sebuah sistem dapat digunakan.

Cerita dan skenario adalah deskripsi tentang bagaimana suatu sistem dapat digunakan untuk tugas tertentu.

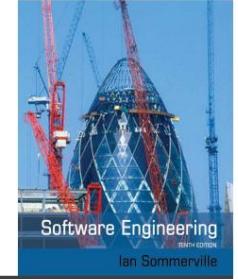
Karena didasarkan pada situasi praktis, pemangku kepentingan dapat berhubungan dengan mereka dan dapat mengomentari situasi mereka sehubungan dengan cerita.



Berbagi foto di dalam kelas (iLearn)

Jack adalah seorang guru sekolah dasar di Ullapool (sebuah desa di utara Skotlandia). Dia punya memutuskan bahwa proyek kelas harus difokuskan di sekitar industri perikanan di daerah tersebut, dengan melihat sejarah, perkembangan dan dampak ekonomi dari penangkapan ikan. Sebagai bagian dari ini, siswa diminta untuk mengumpulkan dan berbagi kenang-kenangan dari kerabat, menggunakan arsip koran dan mengumpulkan foto-foto lama yang berkaitan dengan komunitas nelayan dan nelayan di daerah tersebut. Murid menggunakan wiki iLearn untuk mengumpulkan cerita memancing dan SCRAPAN (situs sumber sejarah) untuk mengakses arsip surat kabar dan foto. Namun, Jack juga membutuhkan situs berbagi foto karena dia ingin siswa saling mengambil dan mengomentari foto satu sama lain dan mengunggah pindaian foto lama yang mungkin mereka miliki di keluarga mereka.

Jack mengirim email ke grup guru sekolah dasar, di mana dia menjadi anggotanya untuk melihat apakah ada yang bisa merekomendasikan sistem yang sesuai. Dua guru membalas dan keduanya menyarankan agar dia menggunakan KidsTakePics, situs berbagi foto yang memungkinkan guru memeriksa dan memoderasi konten. Karena KidsTakePics tidak terintegrasi dengan iLearn layanan otentikasi, ia membuat seorang guru dan akun kelas. Dia menggunakan iLearn setup service untuk menambahkan KidsTakePics ke layanan yang dilihat oleh siswa di kelasnya sehingga ketika mereka login, mereka dapat langsung menggunakan sistem untuk mengupload foto dari perangkat mobile dan komputer kelas mereka.

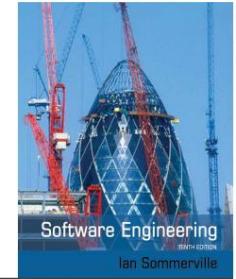


Skenario

Bentuk cerita pengguna yang terstruktur

Skenario harus mencakup

- ÿ Deskripsi situasi awal; ÿ Deskripsi aliran normal peristiwa; ÿ Sebuah deskripsi tentang apa yang bisa salah; ÿ Informasi tentang aktivitas bersamaan lainnya; ÿ Deskripsi keadaan saat skenario selesai.

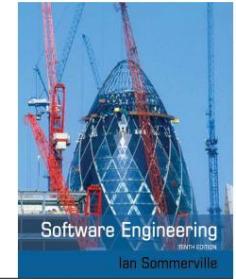


Mengunggah foto iLearn)

Asumsi awal: Seorang pengguna atau sekelompok pengguna memiliki satu atau lebih foto digital untuk diunggah ke situs berbagi gambar. Ini disimpan di komputer tablet atau laptop. Mereka telah berhasil masuk ke KidsTakePics.

Normal : Pengguna memilih unggah foto dan mereka diminta untuk memilih foto yang akan diunggah ke komputer mereka dan untuk memilih nama proyek di mana foto akan disimpan. Mereka juga harus diberikan pilihan untuk memasukkan kata kunci yang harus dikaitkan dengan setiap foto yang diunggah. Foto yang diunggah diberi nama dengan membuat konjungsi nama pengguna dengan nama file foto di komputer lokal.

Setelah pengunggahan selesai, sistem secara otomatis mengirim email ke moderator proyek yang meminta mereka untuk memeriksa konten baru dan membuat pesan di layar kepada pengguna bahwa ini telah dilakukan.



Mengunggah foto

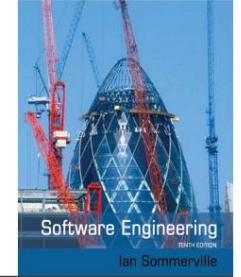
Apa yang bisa salah:

Tidak ada moderator yang terkait dengan proyek yang dipilih. Sebuah email secara otomatis dihasilkan ke administrator sekolah meminta mereka untuk mencalonkan moderator proyek. Pengguna harus diberi tahu bahwa mungkin ada penundaan dalam membuat foto mereka terlihat.

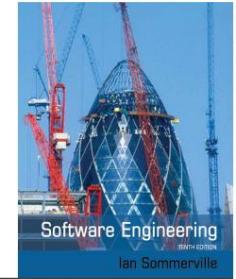
Foto dengan nama yang sama telah diunggah oleh pengguna yang sama. Pengguna harus ditanya apakah mereka ingin mengunggah ulang foto dengan nama yang sama, mengganti nama foto, atau membatalkan pengunggahan. Jika mereka memilih untuk mengunggah ulang foto, aslinya akan ditimpas. Jika mereka memilih untuk mengganti nama foto, nama baru akan dibuat secara otomatis dengan menambahkan nomor ke nama file yang ada.

Aktivitas **Iain**: Moderator dapat masuk ke sistem dan dapat menyetujui foto saat diunggah.

Status **sistem saat selesai**: Pengguna masuk. Foto-foto yang dipilih telah diunggah dan diberi status 'menunggu moderasi'. Foto dapat dilihat oleh moderator dan pengguna yang mengunggahnya.



Spesifikasi kebutuhan



Spesifikasi kebutuhan

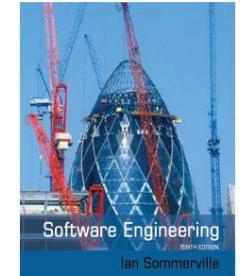
Proses menuliskan kebutuhan pengguna dan sistem dalam dokumen persyaratan.

Persyaratan pengguna harus dapat dimengerti oleh pengguna akhir dan pelanggan yang tidak memiliki latar belakang teknis.

Persyaratan sistem adalah persyaratan yang lebih rinci dan dapat mencakup lebih banyak informasi teknis.

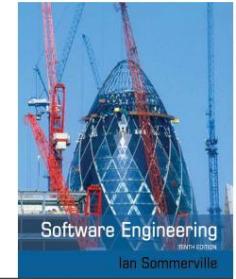
Persyaratan dapat menjadi bagian dari kontrak untuk pengembangan sistem

↳ Oleh karena itu penting bahwa ini selengkap mungkin.



Cara menulis spesifikasi persyaratan sistem

Notasi	Keterangan
Bahasa alami	Persyaratan ditulis menggunakan kalimat bernomor dalam bahasa alami. Setiap kalimat harus mengungkapkan satu persyaratan.
Bahasa alami terstruktur	Persyaratan ditulis dalam bahasa alami pada formulir atau templat standar. Setiap bidang memberikan informasi tentang aspek persyaratan.
Bahasa deskripsi desain	Pendekatan ini menggunakan bahasa seperti bahasa pemrograman, tetapi dengan fitur yang lebih abstrak untuk menentukan persyaratan dengan mendefinisikan model operasional sistem. Pendekatan ini sekarang jarang digunakan meskipun dapat berguna untuk spesifikasi antarmuka.
Notasi grafis	Model grafis, dilengkapi dengan penjelasan teks, digunakan untuk menentukan kebutuhan fungsional sistem; UML use case dan diagram urutan biasanya digunakan.
Spesifikasi matematika	Notasi ini didasarkan pada konsep matematika seperti mesin atau set keadaan hingga. Meskipun spesifikasi yang tidak ambigu ini dapat mengurangi ambiguitas dalam dokumen persyaratan, sebagian besar pelanggan tidak memahami spesifikasi formal. Mereka tidak dapat memeriksa apakah itu mewakili apa yang mereka inginkan dan enggan menerimanya sebagai kontrak sistem

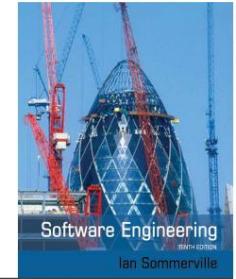


Persyaratan dan desain

Pada prinsipnya, persyaratan harus menyatakan apa yang harus dilakukan sistem dan desain harus menjelaskan bagaimana hal itu dilakukan.

Dalam praktiknya, persyaratan dan desain tidak dapat dipisahkan

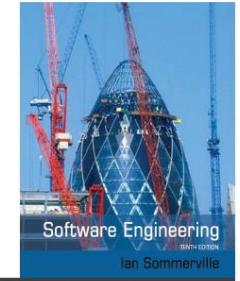
- ÿ Arsitektur sistem dapat dirancang untuk menstrukturkan Persyaratan;
- ÿ Sistem dapat saling beroperasi dengan sistem lain yang menghasilkan persyaratan desain;
- ÿ Penggunaan arsitektur tertentu untuk memenuhi persyaratan non-fungsional mungkin merupakan persyaratan domain.
- ÿ Ini mungkin merupakan konsekuensi dari persyaratan peraturan.



Spesifikasi bahasa alami

Persyaratan ditulis sebagai kalimat bahasa alami dilengkapi dengan diagram dan tabel.

Digunakan untuk keperluan menulis karena ekspresif, intuitif dan universal. Ini berarti bahwa persyaratan dapat dipahami oleh pengguna dan pelanggan.



Pedoman persyaratan menulis

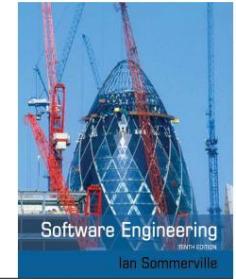
Ciptakan format standar dan gunakan untuk semua persyaratan.

Gunakan bahasa secara konsisten. Gunakan harus untuk persyaratan wajib, harus untuk persyaratan yang diinginkan.

Gunakan penyorotan teks untuk mengidentifikasi bagian-bagian penting dari persyaratan.

Hindari penggunaan jargon komputer.

Sertakan penjelasan (alasan) mengapa suatu persyaratan diperlukan.



Masalah dengan bahasa alami

Kurang jelas

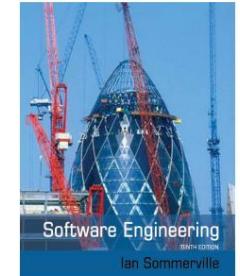
Presisi sulit tanpa membuat dokumen sulit untuk
Baca.

Kebingungan persyaratan

↳ Persyaratan fungsional dan non-fungsional cenderung tercampur.

Persyaratan penggabungan

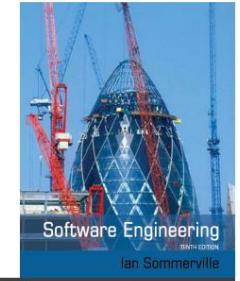
↳ Beberapa persyaratan yang berbeda dapat diungkapkan bersama-sama.



Contoh persyaratan untuk sistem perangkat lunak pompa insulin

3.2 Sistem harus mengukur gula darah dan mengirimkan insulin, jika diperlukan, setiap 10 menit. (*Perubahan gula darah relatif lambat sehingga pengukuran yang lebih sering tidak diperlukan; pengukuran yang lebih jarang dapat menyebabkan kadar gula tinggi yang tidak perlu.*)

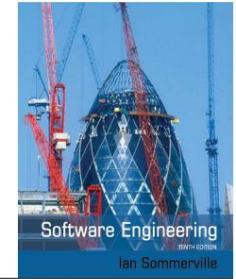
3.6 Sistem harus menjalankan rutin swa-uji setiap menit dengan kondisi yang akan diuji dan tindakan terkait yang ditentukan dalam Tabel 1.
(rutin swauji dapat menemukan masalah perangkat keras dan perangkat lunak dan mengingatkan pengguna pada fakta bahwa operasi normal dapat menjadi tidak mungkin.)



Spesifikasi terstruktur

Pendekatan persyaratan penulisan di mana kebebasan penulis persyaratan terbatas dan persyaratan ditulis dengan cara yang standar.

Ini bekerja dengan baik untuk beberapa jenis persyaratan misalnya persyaratan untuk sistem kontrol tertanam tetapi terkadang terlalu kaku untuk menulis persyaratan sistem bisnis.



Spesifikasi berbasis formulir

Definisi fungsi atau entitas.

Deskripsi input dan dari mana asalnya.

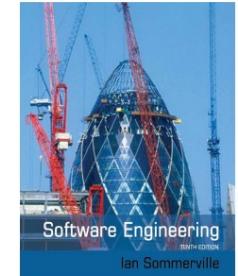
Deskripsi keluaran dan ke mana mereka pergi.

Informasi tentang informasi yang dibutuhkan untuk komputasi dan entitas lain yang digunakan.

Deskripsi tindakan yang akan dilakukan.

Kondisi sebelum dan sesudah (jika sesuai).

Efek samping (jika ada) dari fungsi tersebut.



A structured specification of a requirement for an insulin pump

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

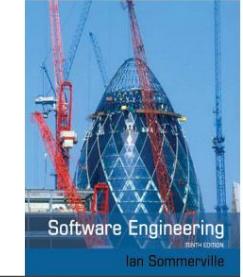
Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r_2); the previous two readings (r_0 and r_1).

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

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.



Spesifikasi terstruktur dari persyaratan untuk pompa insulin

Action

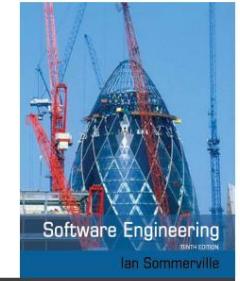
Requirements

Two open biosimulation settings so that the relative effect of sugar_level can be computed.

Pre-condition

The insulin reservoir contains at least the maximum allowed amount of insulin during a single dose infusion again. -----v

Postcondition **Sickles effects** Tidak ada. **None.**

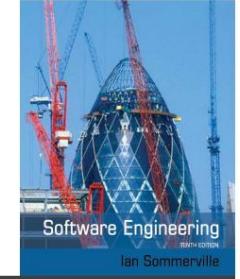


Spesifikasi tabel

Digunakan untuk melengkapi bahasa alami.

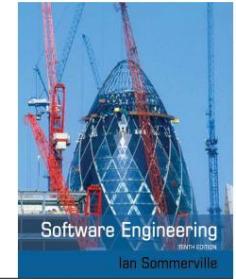
Sangat berguna ketika Anda harus menentukan sejumlah kemungkinan tindakan alternatif.

Misalnya, sistem pompa insulin mendasarkan perhitungan pada tingkat perubahan kadar gula darah dan spesifikasi tabel menjelaskan bagaimana menghitung kebutuhan insulin untuk skenario yang berbeda.



Spesifikasi tabel perhitungan untuk pompa insulin

Kondisi	Tindakan
Kadar gula turun ($r_2 < r_1$)	CompDose = 0
Kadar gula stabil ($r_2 = r_1$)	CompDose = 0
Tingkat gula meningkat dan tingkat bertambah berkurang $((r_2 - r_1) < (r_1 - r_0))$	CompDose = 0
Tingkat gula meningkat dan tingkat meningkat stabil meningkat $((r_2 - r_1) (r_1 - r_0))$	<p>Putaran = CompDose $((r_2 - r_1)/4)$</p> <p>Jika dibulatkan hasilnya = 0 maka = Dosis Komprehensif</p> <p>Dosis Minimum</p>



Gunakan kasus

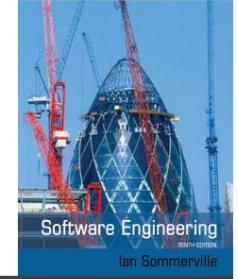
Use-case adalah sejenis skenario yang termasuk dalam UML.

Use case mengidentifikasi aktor dalam interaksi dan yang menggambarkan interaksi itu sendiri.

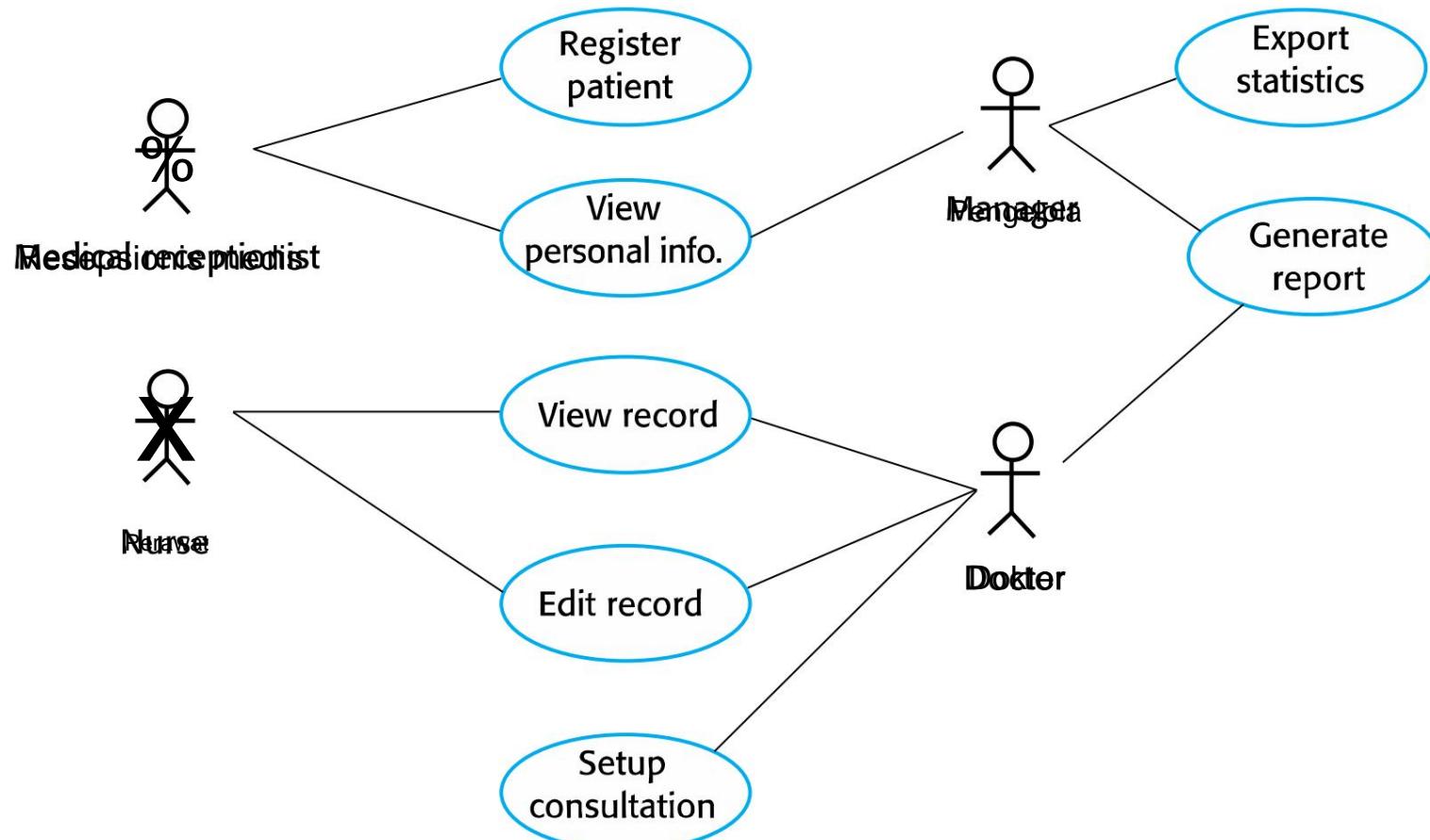
Seperangkat kasus penggunaan harus menggambarkan semua kemungkinan interaksi dengan sistem.

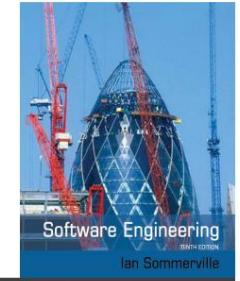
Model grafis tingkat tinggi dilengkapi dengan deskripsi tabel yang lebih rinci (lihat Bab 5).

Diagram urutan UML dapat digunakan untuk menambahkan detail ke use-cases dengan menunjukkan urutan pemrosesan event dalam sistem.



Gunakan kasus untuk sistem Mentcare



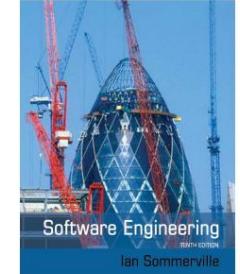


Dokumen persyaratan perangkat lunak

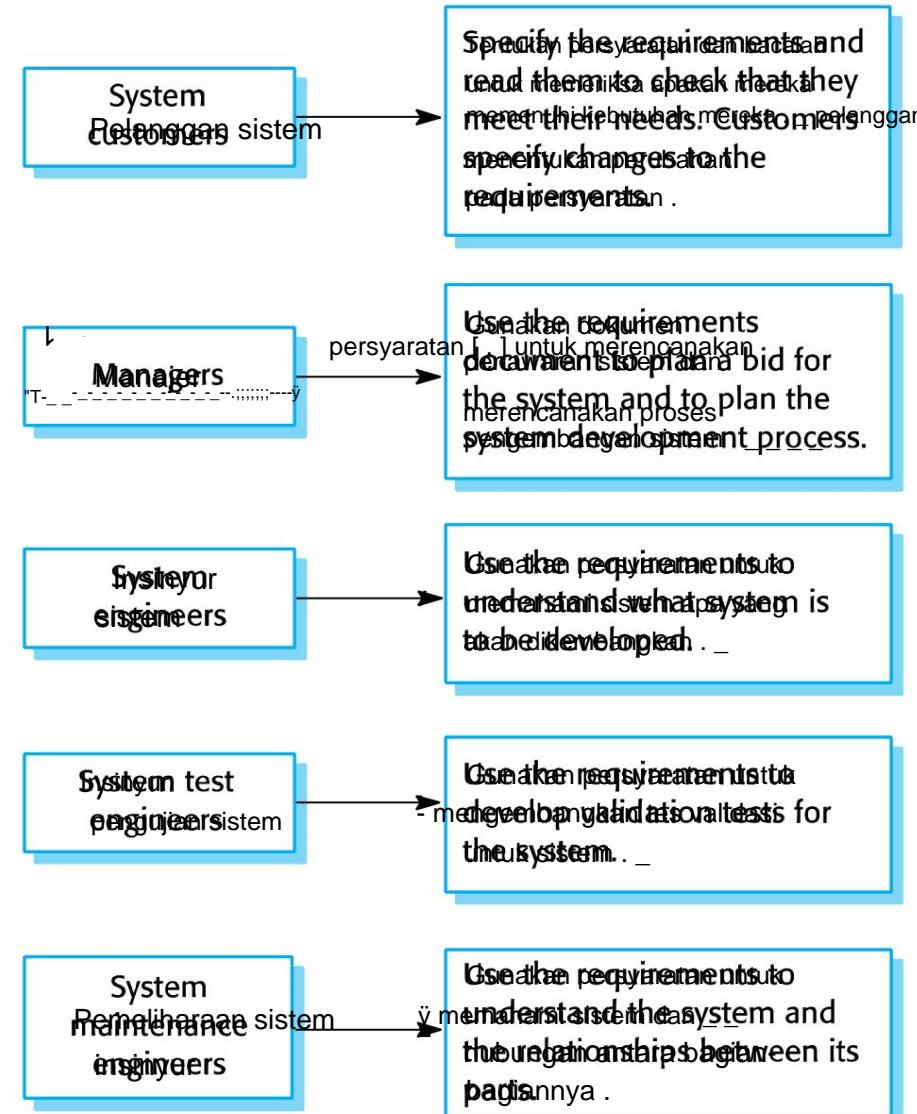
Dokumen persyaratan perangkat lunak adalah resmi pernyataan tentang apa yang diperlukan dari pengembang sistem.

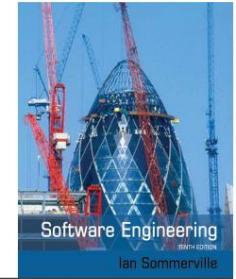
Harus mencakup definisi kebutuhan pengguna dan spesifikasi persyaratan sistem.

Ini BUKAN dokumen desain. Sejauh mungkin, itu harus mengatur APA yang harus dilakukan sistem daripada BAGAIMANA seharusnya melakukannya.



Pengguna dokumen persyaratan



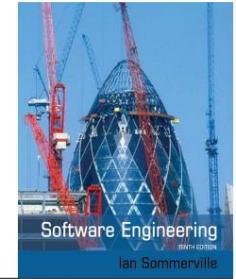


Variabilitas dokumen persyaratan

Informasi dalam dokumen persyaratan tergantung pada jenis sistem dan pendekatan pengembangan yang digunakan.

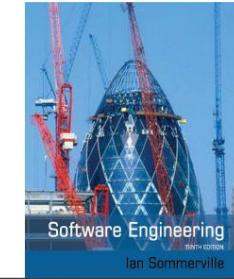
Sistem yang dikembangkan secara bertahap akan, biasanya, memiliki lebih sedikit detail dalam dokumen persyaratan.

Standar dokumen persyaratan telah dirancang misalnya standar IEEE. Ini sebagian besar berlaku untuk persyaratan untuk proyek rekayasa sistem besar.



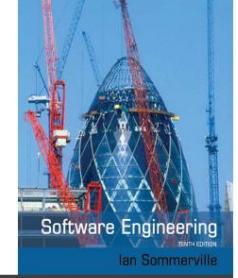
Struktur dokumen persyaratan

Bab	Keterangan
Kata pengantar	Ini harus mendefinisikan pembaca yang diharapkan dari dokumen dan menjelaskan riwayat versinya, termasuk alasan pembuatan versi baru dan ringkasan perubahan yang dibuat di setiap versi.
pengantar	Ini harus menggambarkan kebutuhan sistem. Ini harus menjelaskan secara singkat fungsi sistem dan menjelaskan cara kerjanya dengan sistem lain. Dia juga harus menjelaskan bagaimana sistem cocok dengan bisnis secara keseluruhan atau tujuan strategis organisasi yang menugaskan perangkat lunak.
Glosarium	Ini harus mendefinisikan istilah teknis yang digunakan dalam dokumen. Anda harus tidak membuat asumsi tentang pengalaman atau keahlian pembaca.
Persyaratan pengguna definisi	Di sini, Anda menjelaskan layanan yang disediakan untuk pengguna. yang tidak berfungsi persyaratan sistem juga harus dijelaskan dalam bagian ini. Ini deskripsi dapat menggunakan bahasa alami, diagram, atau notasi lain yang dimengerti oleh pelanggan. Standar produk dan proses yang harus diikuti harus ditentukan.
Sistem arsitektur	Bab ini harus menyajikan gambaran umum tingkat tinggi dari sistem yang diantisipasi arsitektur, menunjukkan distribusi fungsi di seluruh modul sistem. Komponen arsitektur yang digunakan kembali harus disorot.

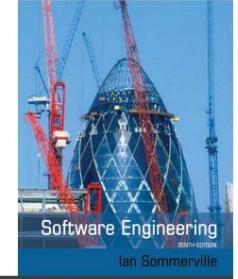


Struktur dokumen persyaratan

Bab	Keterangan
Sistem Persyaratan spesifikasi	Ini harus menjelaskan persyaratan fungsional dan nonfungsional secara lebih rinci. Jika perlu, detail lebih lanjut juga dapat ditambahkan ke persyaratan nonfungsional. Antarmuka ke sistem lain dapat ditentukan.
Model sistem	Ini mungkin termasuk model sistem grafis yang menunjukkan hubungan antara komponen sistem dan sistem serta lingkungannya. Contoh dari model yang mungkin adalah model objek, model aliran data, atau model data semantik.
Evolusi sistem	Ini harus menggambarkan asumsi mendasar yang menjadi dasar sistem, dan setiap perubahan yang diantisipasi karena evolusi perangkat keras, perubahan kebutuhan pengguna, dan seterusnya. Bagian ini berguna untuk perancang sistem karena dapat membantu mereka menghindari keputusan desain yang akan membatasi kemungkinan perubahan di masa depan pada sistem.
Lampiran	Ini harus memberikan informasi rinci dan spesifik yang terkait dengan aplikasi yang sedang dikembangkan; misalnya, deskripsi perangkat keras dan database. Persyaratan perangkat keras menentukan konfigurasi minimal dan optimal untuk sistem. Persyaratan database menentukan organisasi logis dari data yang digunakan oleh sistem dan hubungan antar data.
Indeks	Beberapa indeks ke dokumen dapat disertakan. Serta alfabet normal indeks, mungkin ada indeks diagram, indeks fungsi, dan sebagainya.



Validasi persyaratan

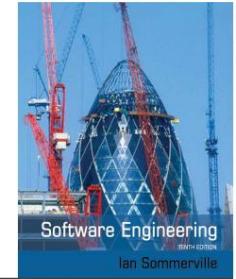


Validasi persyaratan

Berkaitan dengan mendemonstrasikan bahwa persyaratan mendefinisikan sistem yang benar-benar diinginkan pelanggan.

Biaya kesalahan persyaratan tinggi sehingga validasi sangat penting

- ÿ Memperbaiki kesalahan persyaratan setelah pengiriman dapat memakan biaya hingga 100 kali lipat biaya untuk memperbaiki kesalahan implementasi.



Pemeriksaan persyaratan

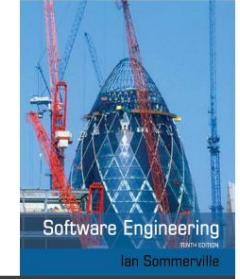
Validitas. Apakah sistem menyediakan fungsi-fungsi yang dukungan terbaik kebutuhan pelanggan?

Konsistensi. Apakah ada konflik persyaratan?

Kelengkapan. Apakah semua fungsi yang dibutuhkan oleh pelanggan disertakan?

Realisme. Dapatkah persyaratan diimplementasikan dengan anggaran dan teknologi yang tersedia?

Verifikasi. Apakah persyaratannya bisa dicek?



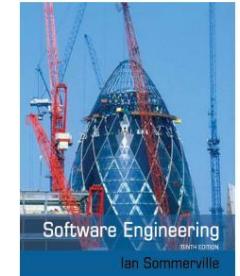
Teknik validasi persyaratan

Tinjauan persyaratan

ÿ Analisis manual yang sistematis dari persyaratan.

Prototipe

ÿ Menggunakan model sistem yang dapat dieksekusi untuk memeriksa persyaratan.



Tinjau cek

Verifikasi

ÿ Apakah persyaratan tersebut dapat diuji secara realistik?

Dapat dipahami

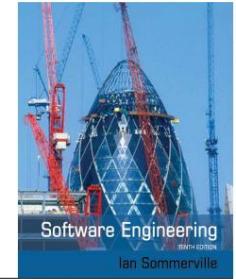
ÿ Apakah persyaratan dipahami dengan benar?

Ketertelusuran

ÿ Apakah asal persyaratan dinyatakan dengan jelas?

Kemampuan beradaptasi

ÿ Dapatkah persyaratan diubah tanpa dampak besar pada persyaratan lain?



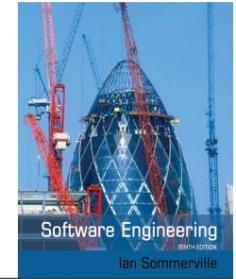
Poin-poin penting

Persyaratan untuk sistem perangkat lunak menetapkan apa yang sistem harus melakukan dan menentukan batasan pada operasi dan implementasinya.

Persyaratan fungsional adalah pernyataan layanan yang harus disediakan sistem atau deskripsi tentang bagaimana beberapa komputasi harus dilakukan.

Persyaratan non-fungsional sering membatasi sistem yang sedang dikembangkan dan proses pengembangan yang digunakan.

Mereka sering berhubungan dengan sifat-sifat yang muncul dari sistem dan karena itu berlaku untuk sistem secara keseluruhan.

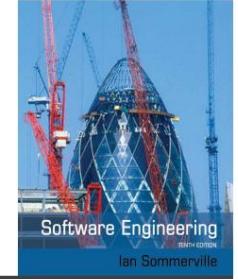


Poin-poin penting

Proses rekayasa persyaratan adalah proses berulang yang mencakup elisitasi, spesifikasi, dan validasi persyaratan.

Persyaratan elisitasi adalah proses berulang yang dapat direpresentasikan sebagai spiral kegiatan – penemuan persyaratan, klasifikasi dan organisasi persyaratan, negosiasi persyaratan dan dokumentasi persyaratan.

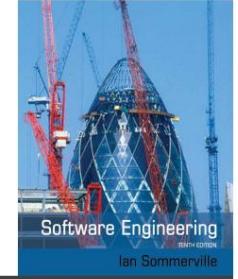
Anda dapat menggunakan berbagai teknik untuk memperoleh persyaratan termasuk wawancara dan etnografi. Cerita dan skenario pengguna dapat digunakan untuk memfasilitasi diskusi.



Poin-poin penting

Spesifikasi persyaratan adalah proses mendokumentasikan secara formal persyaratan pengguna dan sistem dan membuat dokumen persyaratan perangkat lunak.

Dokumen persyaratan perangkat lunak adalah pernyataan persyaratan sistem yang disepakati. Itu harus diatur sehingga pelanggan sistem dan pengembang perangkat lunak dapat menggunakannya.



Poin-poin penting

Validasi persyaratan adalah proses memeriksa persyaratan untuk validitas, konsistensi, kelengkapan, realisme, dan keterverifikasi.

Perubahan bisnis, organisasi, dan teknis pasti mengarah pada perubahan persyaratan untuk sistem perangkat lunak. Manajemen persyaratan adalah proses mengelola dan mengendalikan perubahan ini.



Bab 5 – Pemodelan Sistem



Topik yang dibahas

- ✧ Model konteks
- ✧ Model interaksi
- ✧ Model struktural
- ✧ Model perilaku
- ✧ Rekayasa berbasis model

Pemodelan sistem



- ✧ Pemodelan sistem adalah proses mengembangkan model abstrak dari suatu sistem, dengan masing-masing model menyajikan pandangan atau perspektif yang berbeda dari sistem itu.
- ✧ Pemodelan sistem kini berarti mewakili sistem menggunakan beberapa jenis notasi grafis, yang sekarang hampir selalu didasarkan pada notasi dalam Unified Modeling Language (UML).
- ✧ **Pemodelan sistem membantu analis untuk memahami fungsionalitas sistem dan model yang digunakan untuk berkomunikasi dengan pelanggan .**

Model sistem yang ada dan yang direncanakan



- ✧ **Model dari sistem yang ada digunakan selama rekayasa kebutuhan.** Mereka membantu memperjelas apa yang dilakukan sistem yang ada dan dapat digunakan sebagai dasar untuk mendiskusikan kekuatan dan kelemahannya. Ini kemudian mengarah pada persyaratan untuk sistem baru.
- ✧ Model sistem baru digunakan selama rekayasa persyaratan untuk membantu menjelaskan persyaratan yang diusulkan kepada pemangku kepentingan sistem lainnya. Insinyur menggunakan model ini untuk mendiskusikan proposal desain dan untuk mendokumentasikan sistem untuk implementasi.
- ✧ Dalam proses rekayasa model-driven, dimungkinkan untuk menghasilkan implementasi sistem yang lengkap atau sebagian dari model sistem.



Perspektif sistem

- ✧ Perspektif eksternal, di mana Anda memodelkan konteks atau lingkungan sistem.
- ✧ Perspektif interaksi, di mana Anda memodelkan interaksi antara sistem dan lingkungannya, atau antara komponen sistem.
- ✧ Perspektif struktural, di mana Anda memodelkan organisasi sistem atau struktur data yang diproses oleh sistem.
- ✧ Perspektif perilaku, di mana Anda memodelkan perilaku dinamis sistem dan bagaimana responsnya terhadap peristiwa.

Jenis diagram UML



- ✧ Diagram aktivitas, yang menunjukkan aktivitas yang terlibat dalam suatu proses atau dalam pemrosesan data.
- ✧ Gunakan diagram kasus, yang menunjukkan interaksi antara sistem dan lingkungannya.
- ✧ Diagram urutan, yang menunjukkan interaksi antara aktor dan sistem dan antara komponen sistem.
- ✧ Diagram kelas, yang menunjukkan kelas-kelas objek dalam sistem dan asosiasi antara kelas-kelas ini.
- ✧ State diagram, yang menunjukkan bagaimana sistem bereaksi terhadap kejadian internal dan eksternal.

Penggunaan model grafis



- ✧ Sebagai sarana untuk memfasilitasi diskusi tentang sistem yang ada atau yang diusulkan
 - Model yang tidak lengkap dan salah tidak masalah karena perannya mendukung diskusi.
- ✧ Sebagai cara untuk mendokumentasikan sistem yang ada
 - Model harus merupakan representasi akurat dari sistem tetapi tidak harus lengkap.
- ✧ Sebagai deskripsi sistem rinci yang dapat digunakan untuk menghasilkan implementasi sistem
 - Model harus benar dan lengkap.



Model konteks

Model konteks



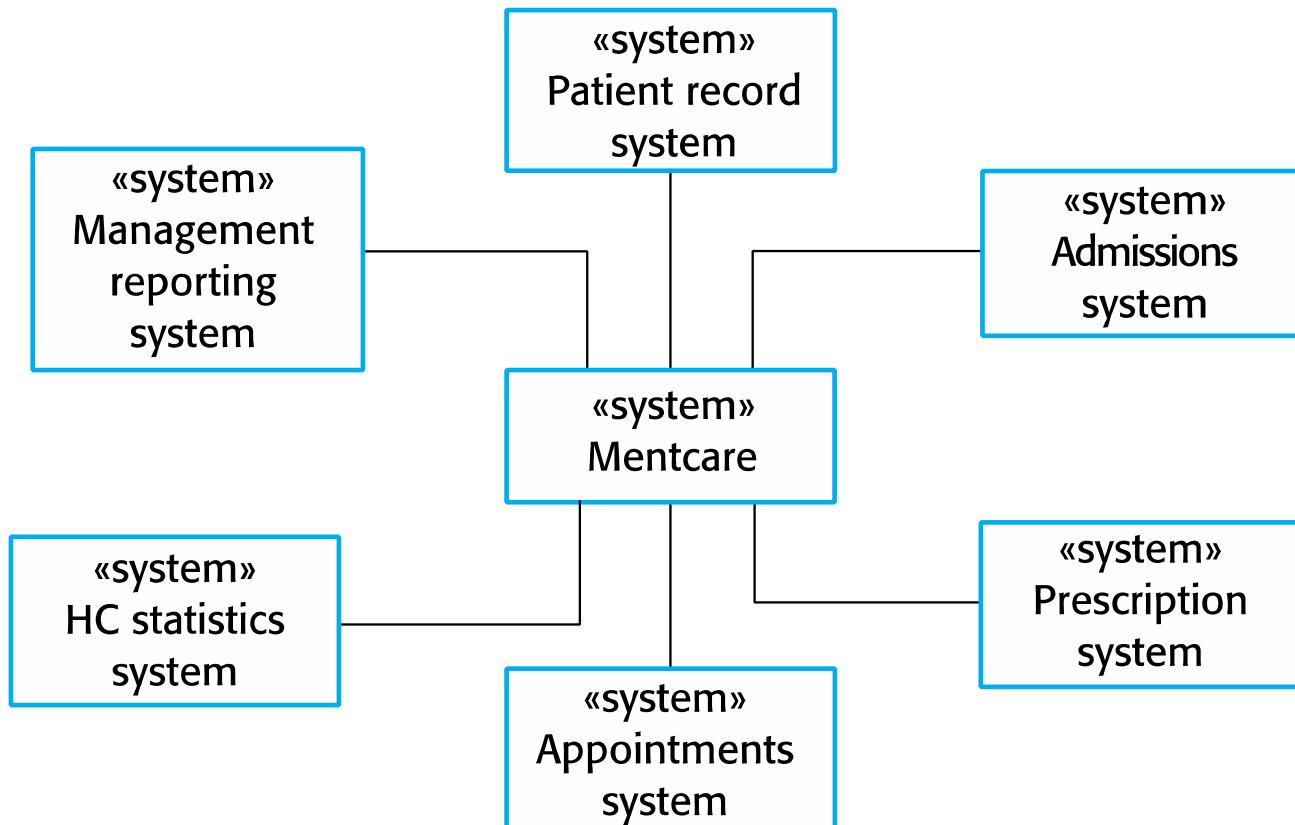
- ✧ Model konteks digunakan untuk mengilustrasikan konteks operasional suatu sistem - model tersebut menunjukkan apa yang ada di luar batas sistem.
- ✧ Kekhawatiran sosial dan organisasi dapat mempengaruhi keputusan tentang di mana memposisikan batas-batas sistem.
- ✧ Model arsitektur menunjukkan sistem dan hubungannya dengan sistem lain.

Batas sistem



- ✧ Batas sistem ditetapkan untuk menentukan apa yang ada di dalam dan apa yang ada di luar sistem.
 - Mereka menunjukkan sistem lain yang digunakan atau bergantung pada sistem yang dikembangkan.
- ✧ Posisi batas sistem memiliki efek mendalam pada persyaratan sistem.
- ✧ Mendefinisikan batas sistem adalah penilaian politik
 - Mungkin ada tekanan untuk mengembangkan batasan sistem yang meningkatkan/menurunkan pengaruh atau beban kerja dari berbagai bagian organisasi.

Konteks sistem

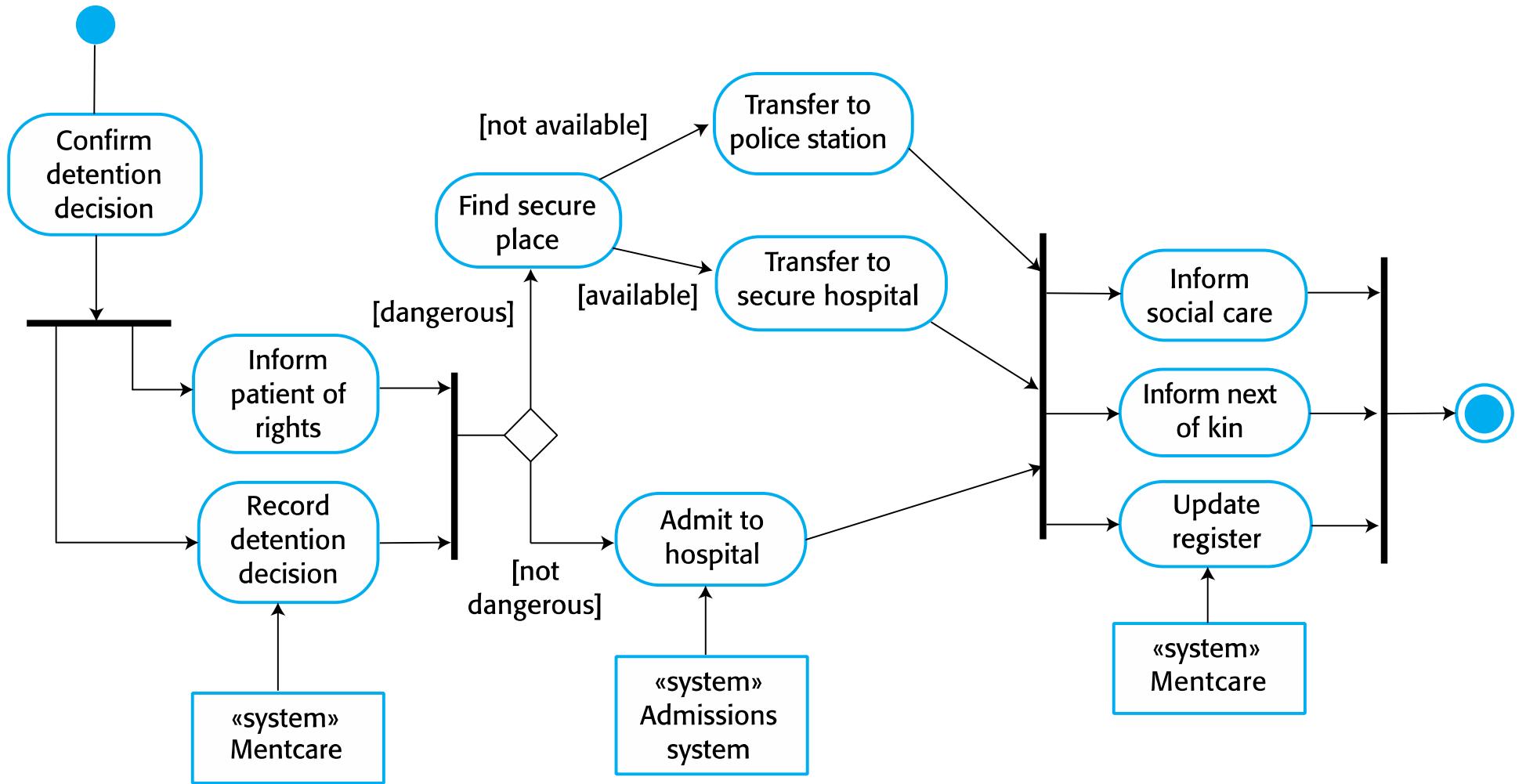


Perspektif proses



- ✧ Model konteks hanya menunjukkan sistem lain di lingkungan, bukan bagaimana sistem yang dikembangkan digunakan di lingkungan itu.
- ✧ Model proses mengungkapkan bagaimana sistem yang dikembangkan digunakan dalam proses bisnis yang lebih luas.
- ✧ Diagram aktivitas UML dapat digunakan untuk mendefinisikan model proses bisnis.

Model proses penahanan tidak sukarela





Model interaksi

Model interaksi



- ✧ Pemodelan interaksi pengguna penting karena membantu mengidentifikasi kebutuhan pengguna.
- ✧ Pemodelan interaksi sistem-ke-sistem menyoroti masalah komunikasi yang mungkin timbul.
- ✧ Pemodelan interaksi komponen membantu kita memahami jika struktur sistem yang diusulkan cenderung memberikan kinerja dan ketergantungan sistem yang diperlukan.
- ✧ Gunakan diagram kasus dan diagram urutan dapat digunakan untuk pemodelan interaksi .

Gunakan pemodelan kasus



- ✧ Use case awalnya dikembangkan untuk mendukung elisitasi kebutuhan dan sekarang dimasukkan ke dalam UML.
- ✧ Setiap use case mewakili tugas diskrit yang melibatkan interaksi eksternal dengan sistem.
- ✧ Aktor dalam use case dapat berupa orang atau sistem lain.
- ✧ Direpresentasikan secara diagram untuk memberikan gambaran umum tentang use case dan dalam bentuk textual yang lebih detail.

Kasus penggunaan transfer-data



- ❖ Sebuah kasus penggunaan dalam sistem Mentcare



Deskripsi tabel dari kasus penggunaan 'Transfer data'



MHC-PMS: Transfer data	
Aktor	Resepsionis medis, sistem catatan pasien (PRS)
Keterangan	Resepsionis dapat mentransfer data dari sistem Mentcase ke database catatan pasien umum yang dikelola oleh otoritas kesehatan. Informasi yang ditransfer dapat berupa informasi pribadi yang diperbarui (alamat, nomor telepon, dll.) atau ringkasan diagnosis dan pengobatan pasien.
Data	Informasi pribadi pasien, ringkasan perawatan
Rangsangan	Perintah pengguna dikeluarkan oleh resepsionis medis
Tanggapan	Konfirmasi bahwa PRS telah diperbarui
Komentar	Resepsionis harus memiliki izin keamanan yang sesuai untuk mengakses informasi pasien dan PRS.

Gunakan kasus dalam sistem Mentcare yang melibatkan peran 'Resepsionis Medis'

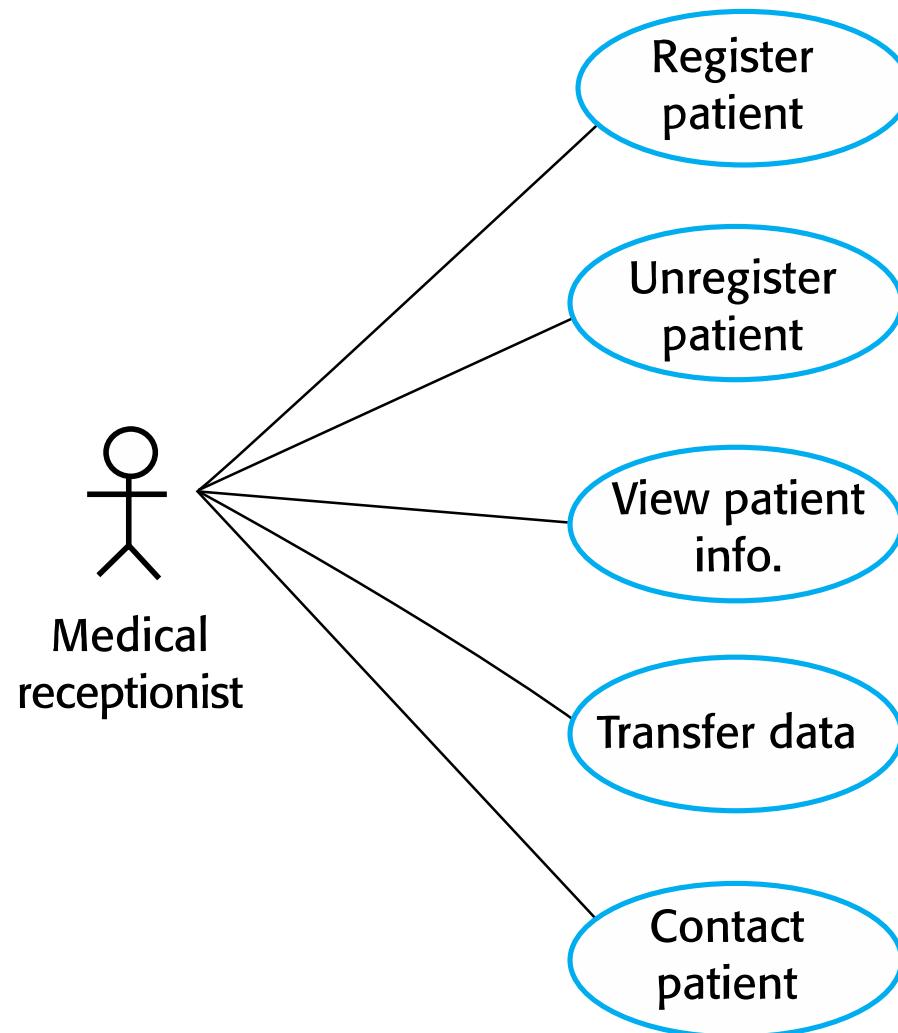
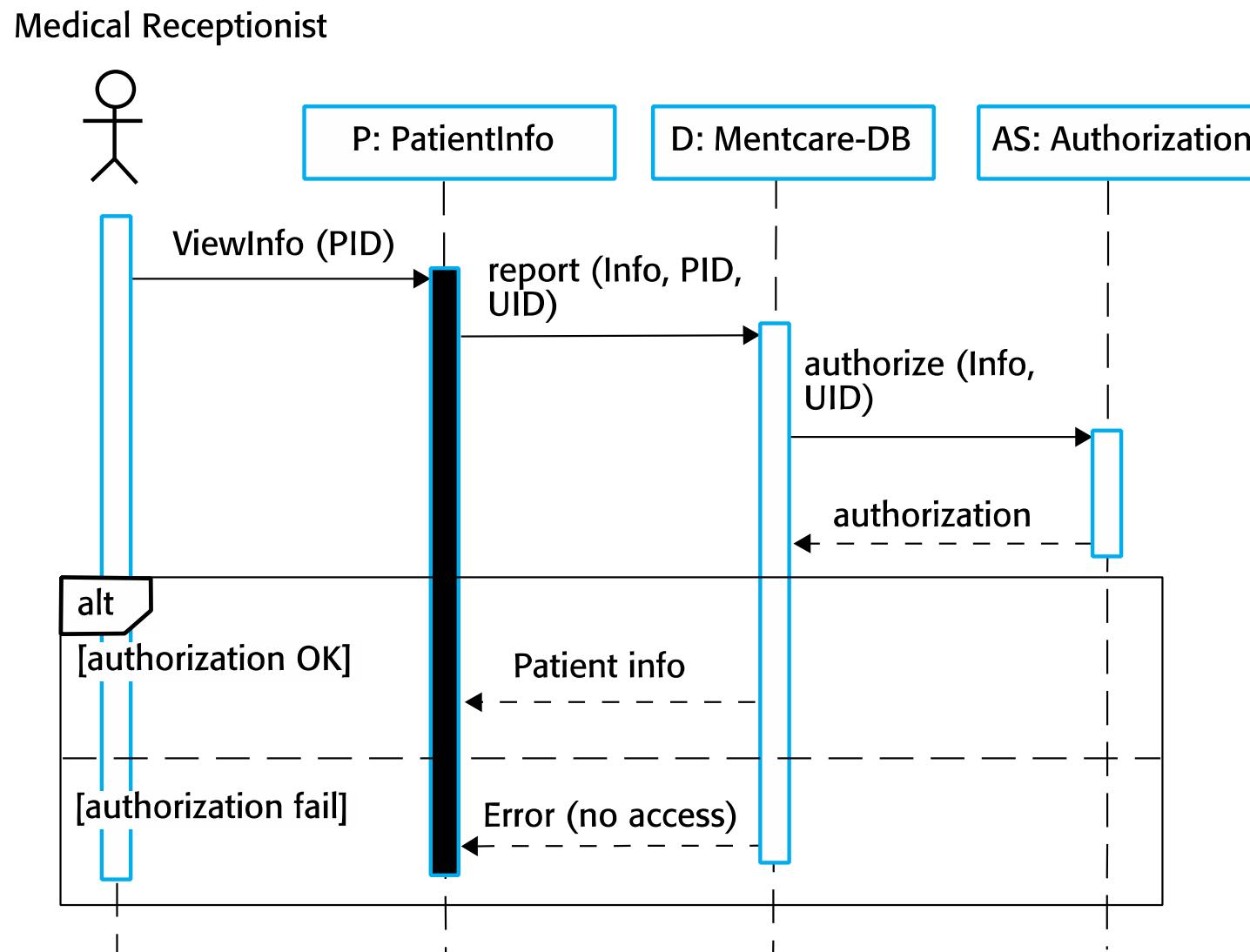


Diagram urutan



- ✧ Diagram urutan adalah bagian dari UML dan digunakan untuk memodelkan interaksi antara aktor dan objek dalam suatu sistem.
- ✧ Sequence diagram menunjukkan urutan interaksi yang terjadi selama use case atau use case instance tertentu.
- ✧ Objek dan aktor yang terlibat terdaftar di sepanjang bagian atas diagram, dengan garis putus-putus yang ditarik secara vertikal darinya.
- ✧ Interaksi antar objek ditunjukkan oleh panah beranotasi.

Diagram urutan untuk Melihat informasi pasien



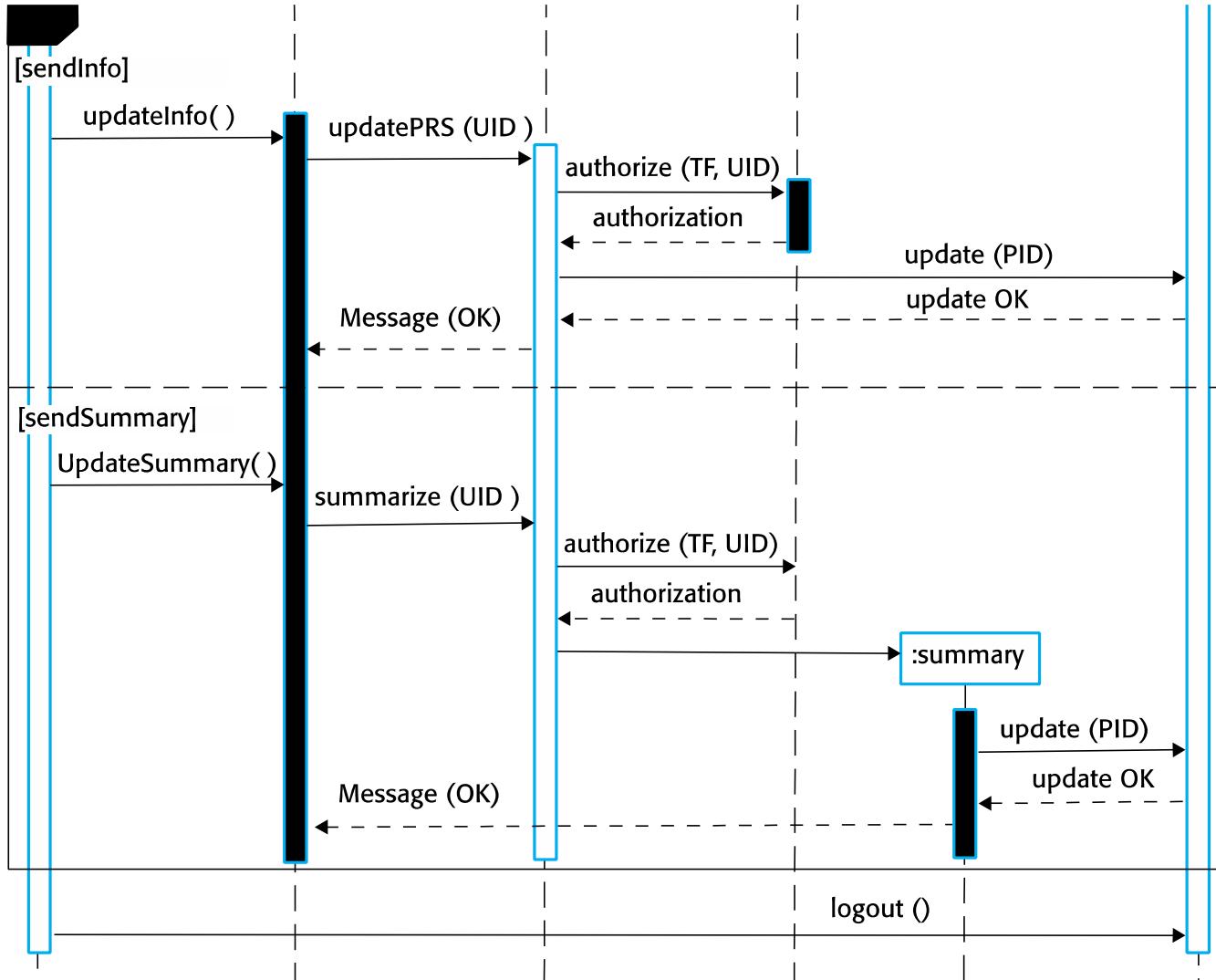
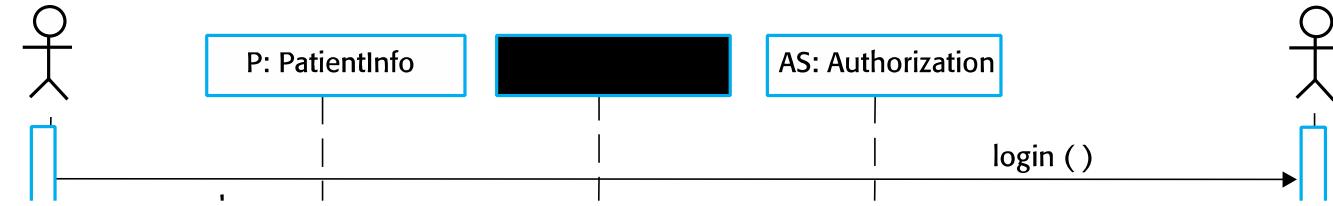


Diagram urutan untuk Transfer Data



Model struktural

Model struktural



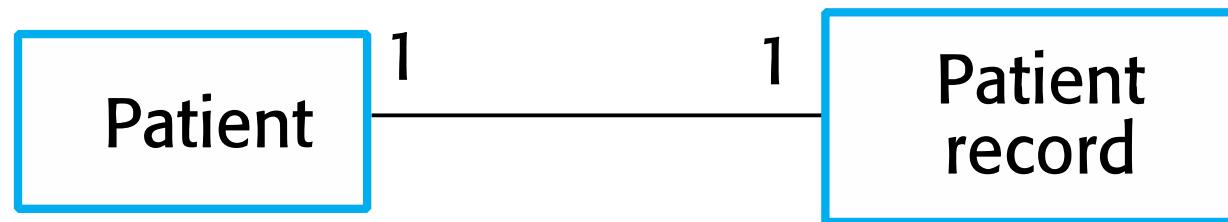
- ✧ Model struktural perangkat lunak menampilkan organisasi sistem dalam hal komponen yang membentuk sistem itu dan hubungannya.
- ✧ Model struktural dapat berupa model statis, yang menunjukkan struktur desain sistem, atau model dinamis, yang menunjukkan organisasi sistem saat dijalankan.
- ✧ Anda membuat model struktural suatu sistem ketika Anda mendiskusikan dan merancang arsitektur sistem.

diagram kelas

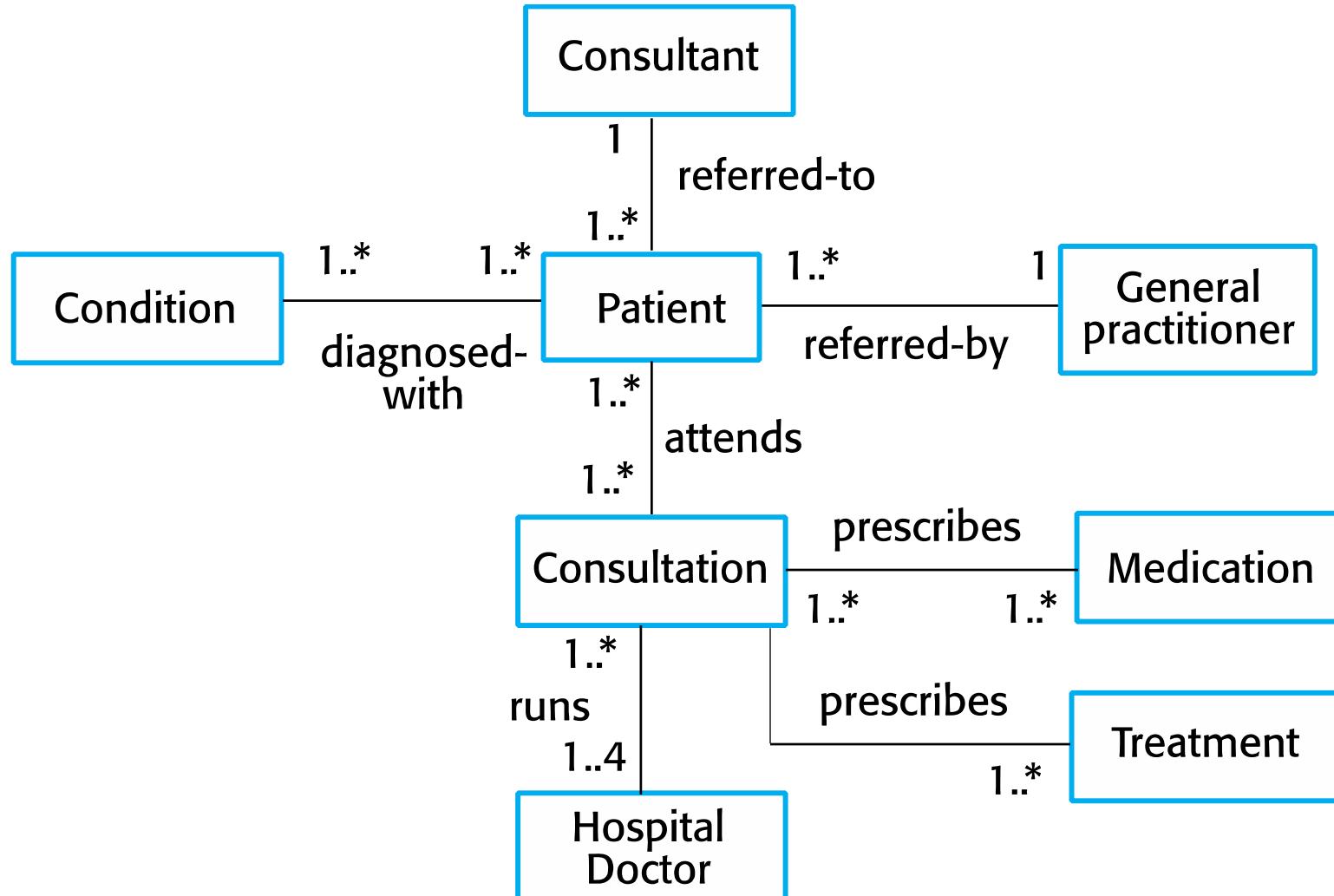


- ✧ Diagram kelas digunakan ketika mengembangkan model sistem berorientasi objek untuk menunjukkan kelas-kelas dalam suatu sistem dan asosiasi antara kelas-kelas ini.
- ✧ Kelas objek dapat dianggap sebagai definisi umum dari satu jenis objek sistem.
- ✧ Asosiasi adalah hubungan antar kelas yang menunjukkan bahwa ada beberapa hubungan antara kelas-kelas tersebut.
- ✧ Saat Anda mengembangkan model selama tahap awal proses rekayasa perangkat lunak, objek mewakili sesuatu di dunia nyata, seperti pasien, resep, dokter, dll.

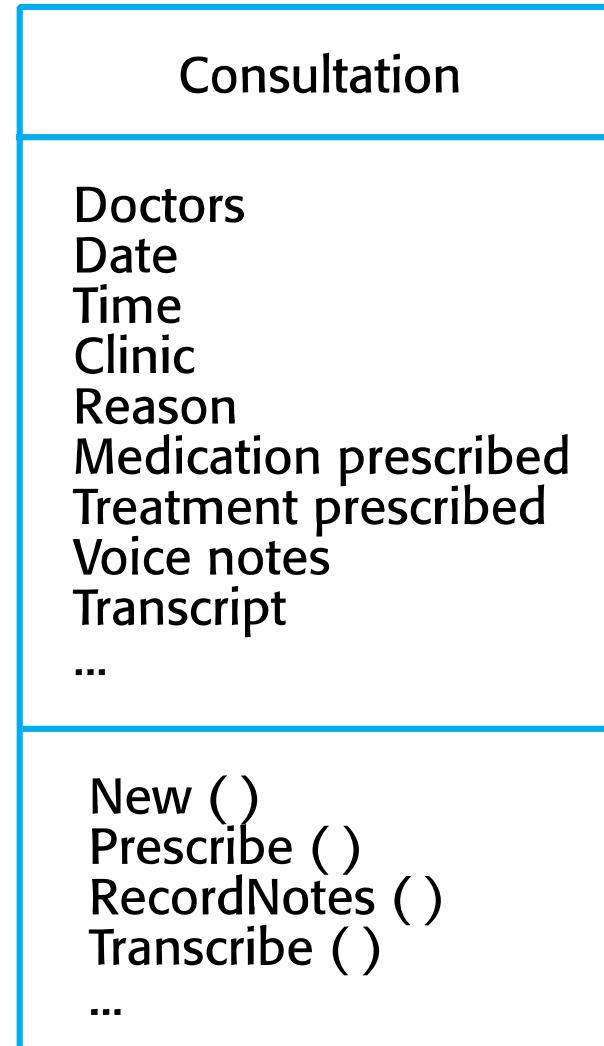
Kelas dan asosiasi UML



Kelas dan asosiasi di MHC-PMS



Kelas Konsultasi



Generalisasi



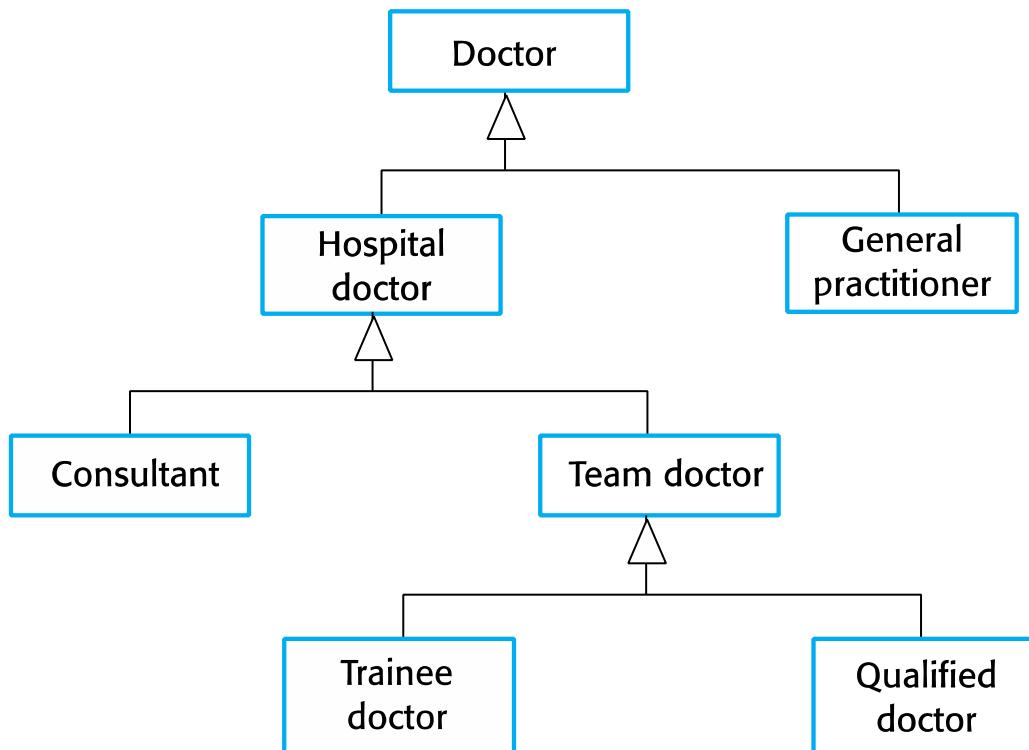
- ✧ Generalisasi adalah teknik sehari-hari yang kita gunakan untuk mengelola kompleksitas.
- ✧ Daripada mempelajari karakteristik terperinci dari setiap entitas yang kami alami, kami menempatkan entitas ini di kelas yang lebih umum (hewan, mobil, rumah, dll.) dan mempelajari karakteristik kelas-kelas ini.
- ✧ Hal ini memungkinkan kita untuk menyimpulkan bahwa anggota yang berbeda dari kelas ini memiliki beberapa karakteristik umum misalnya tupai dan tikus adalah hewan penggerat.

Generalisasi

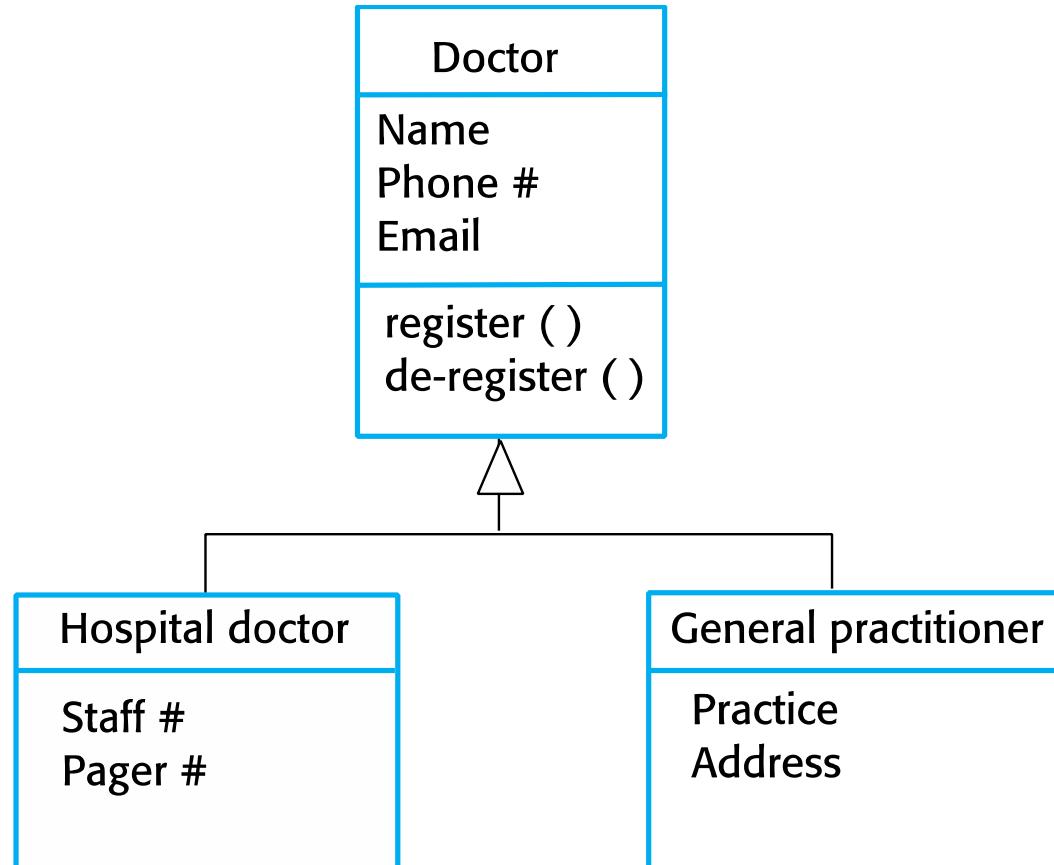


- ✧ Dalam pemodelan sistem, seringkali berguna untuk memeriksa kelas-kelas dalam suatu sistem untuk melihat apakah ada ruang lingkup untuk generalisasi. Jika perubahan diusulkan, maka Anda tidak perlu melihat semua kelas dalam sistem untuk melihat apakah mereka terpengaruh oleh perubahan tersebut.
- ✧ Dalam bahasa berorientasi objek, seperti Java, generalisasi diimplementasikan menggunakan mekanisme pewarisan kelas yang dibangun ke dalam bahasa tersebut.
- ✧ Dalam generalisasi, atribut dan operasi yang terkait dengan kelas tingkat yang lebih tinggi juga terkait dengan kelas tingkat yang lebih rendah.
- ✧ Kelas tingkat bawah adalah subkelas yang mewarisi atribut dan operasi dari superkelasnya . Kelas-kelas tingkat yang lebih rendah ini kemudian menambahkan atribut dan operasi yang lebih spesifik.

Hirarki generalisasi



Hirarki generalisasi dengan detail tambahan

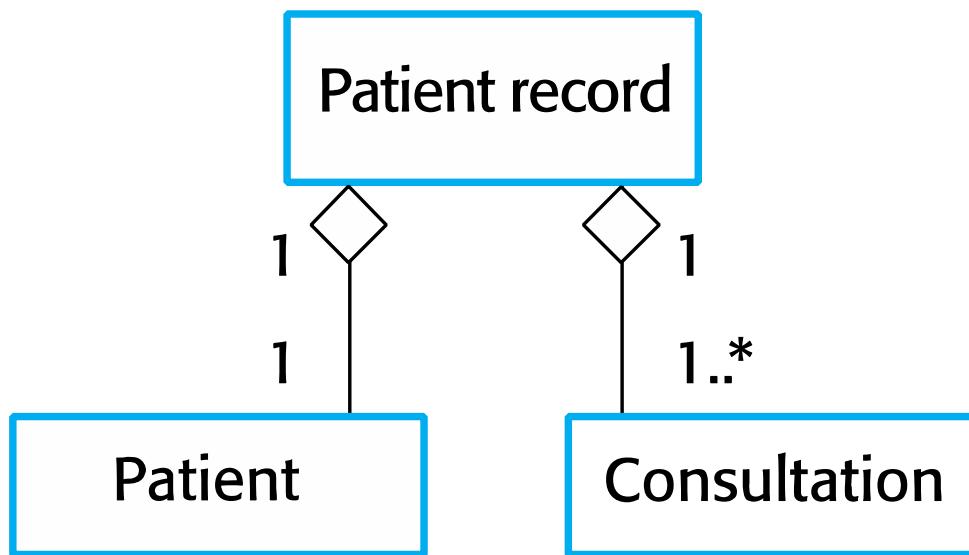




Model agregasi kelas objek

- ✧ Model agregasi menunjukkan bagaimana kelas yang merupakan koleksi terdiri dari kelas lain.
- ✧ Model agregasi mirip dengan bagian-hubungan dalam model data semantik.

Asosiasi agregasi





Model perilaku



Model perilaku

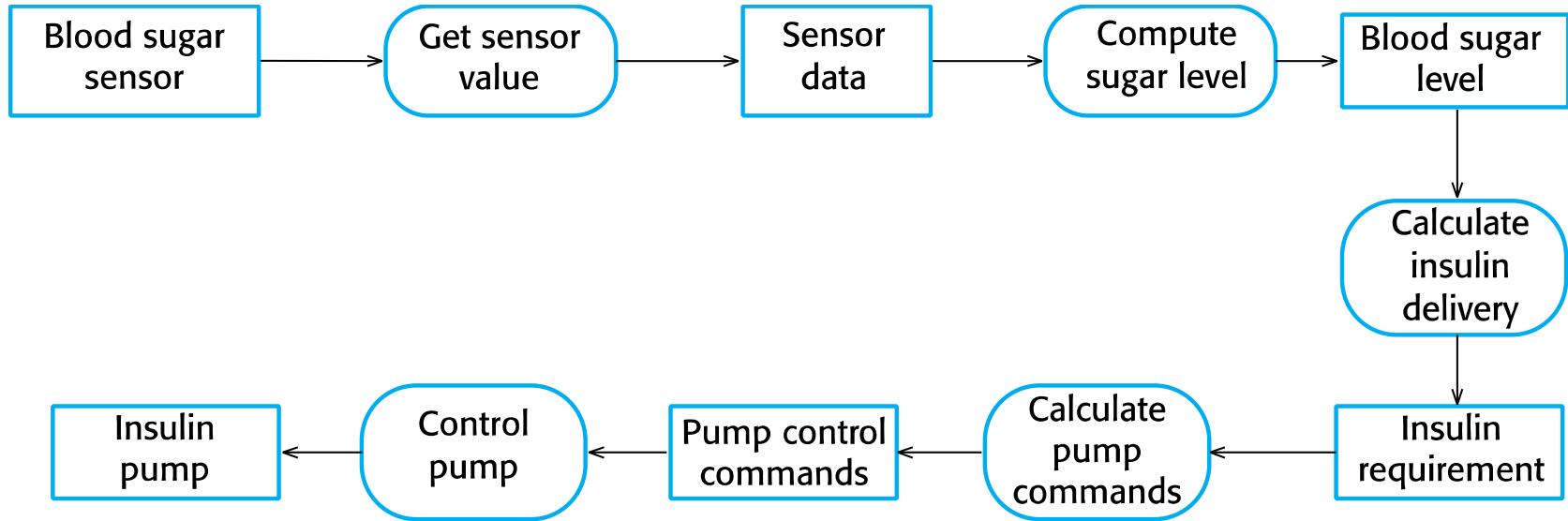
- ✧ Model perilaku adalah model perilaku dinamis dari suatu sistem saat dijalankan. Mereka menunjukkan apa yang terjadi atau apa yang seharusnya terjadi ketika sistem merespon stimulus dari lingkungannya.
- ✧ Anda dapat menganggap rangsangan ini sebagai dua jenis:
 - **Data** Beberapa data datang yang harus diproses oleh sistem.
 - **Peristiwa** Beberapa peristiwa terjadi yang memicu pemrosesan sistem. Peristiwa mungkin memiliki data terkait, meskipun hal ini tidak selalu terjadi.

Pemodelan berbasis data



- ✧ Banyak sistem bisnis adalah sistem pemrosesan data yang terutama didorong oleh data. Mereka dikendalikan oleh input data ke sistem, dengan pemrosesan peristiwa eksternal yang relatif sedikit.
- ✧ Model berbasis data menunjukkan urutan tindakan yang terlibat dalam memproses data input dan menghasilkan output terkait.
- ✧ Mereka sangat berguna selama analisis persyaratan karena dapat digunakan untuk menunjukkan pemrosesan ujung ke ujung dalam suatu sistem.

Model aktivitas operasi pompa insulin

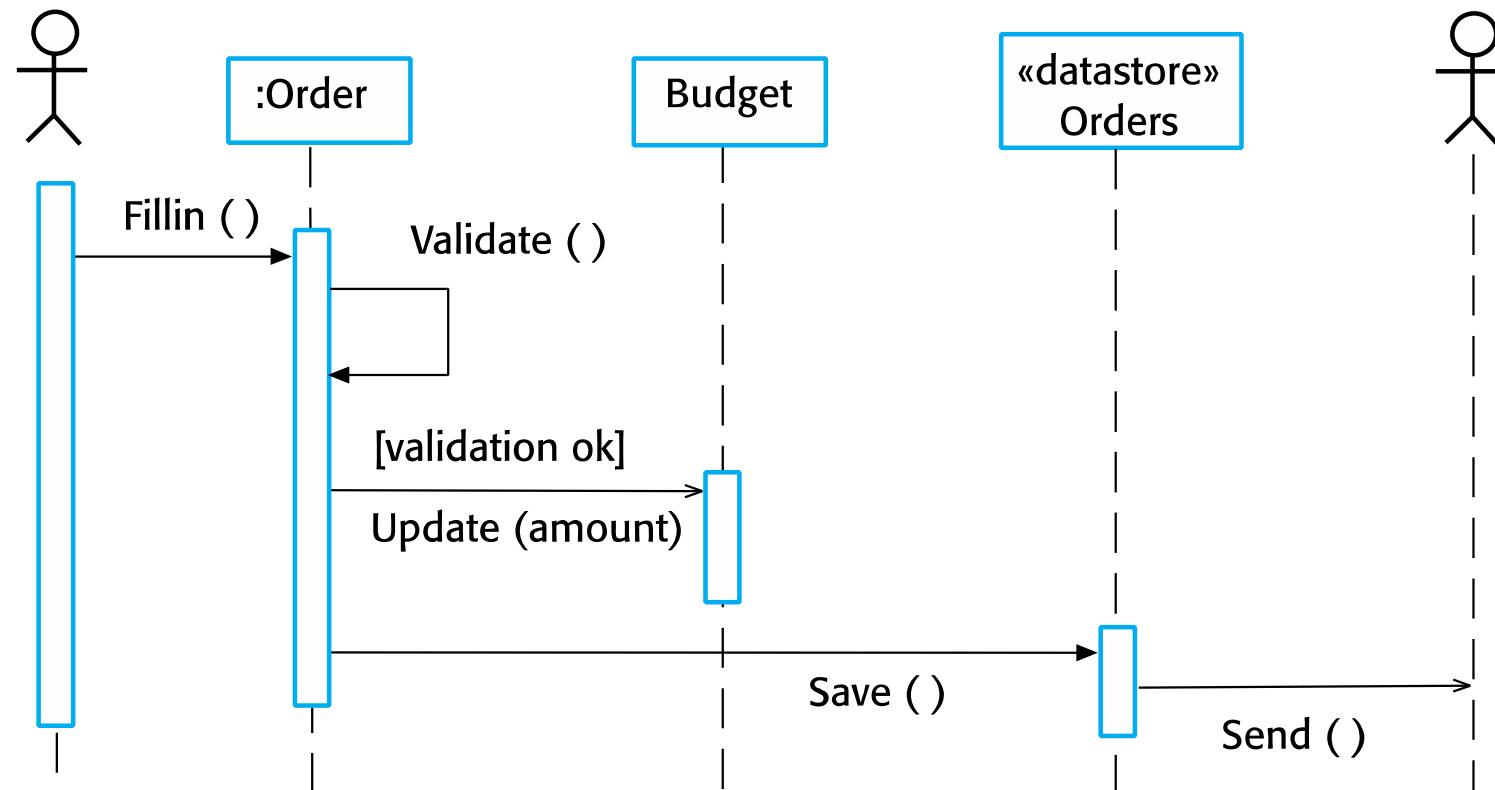


Proses pemesanan



Purchase officer

Supplier



Pemodelan berdasarkan peristiwa



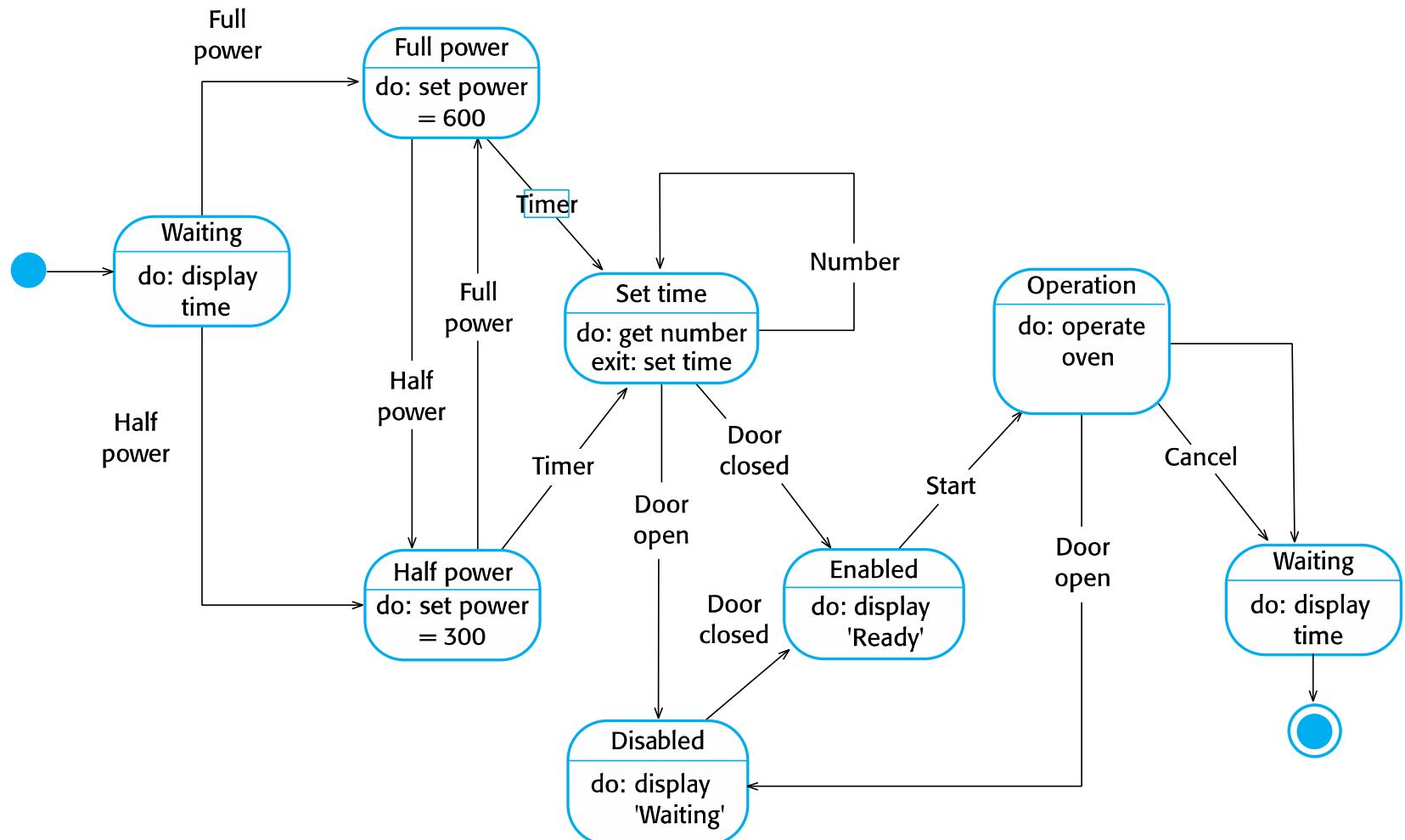
- ✧ Sistem waktu nyata sering kali digerakkan oleh peristiwa, dengan pemrosesan data yang minimal. Misalnya, sistem pengalihan telepon rumah merespons kejadian seperti 'penerima tidak terhubung' dengan menghasilkan nada panggil.
- ✧ Pemodelan berbasis peristiwa menunjukkan bagaimana sistem merespons peristiwa eksternal dan internal.
- ✧ Ini didasarkan pada asumsi bahwa suatu sistem memiliki jumlah keadaan yang terbatas dan bahwa peristiwa (stimuli) dapat menyebabkan transisi dari satu keadaan ke keadaan lainnya.

Model mesin negara



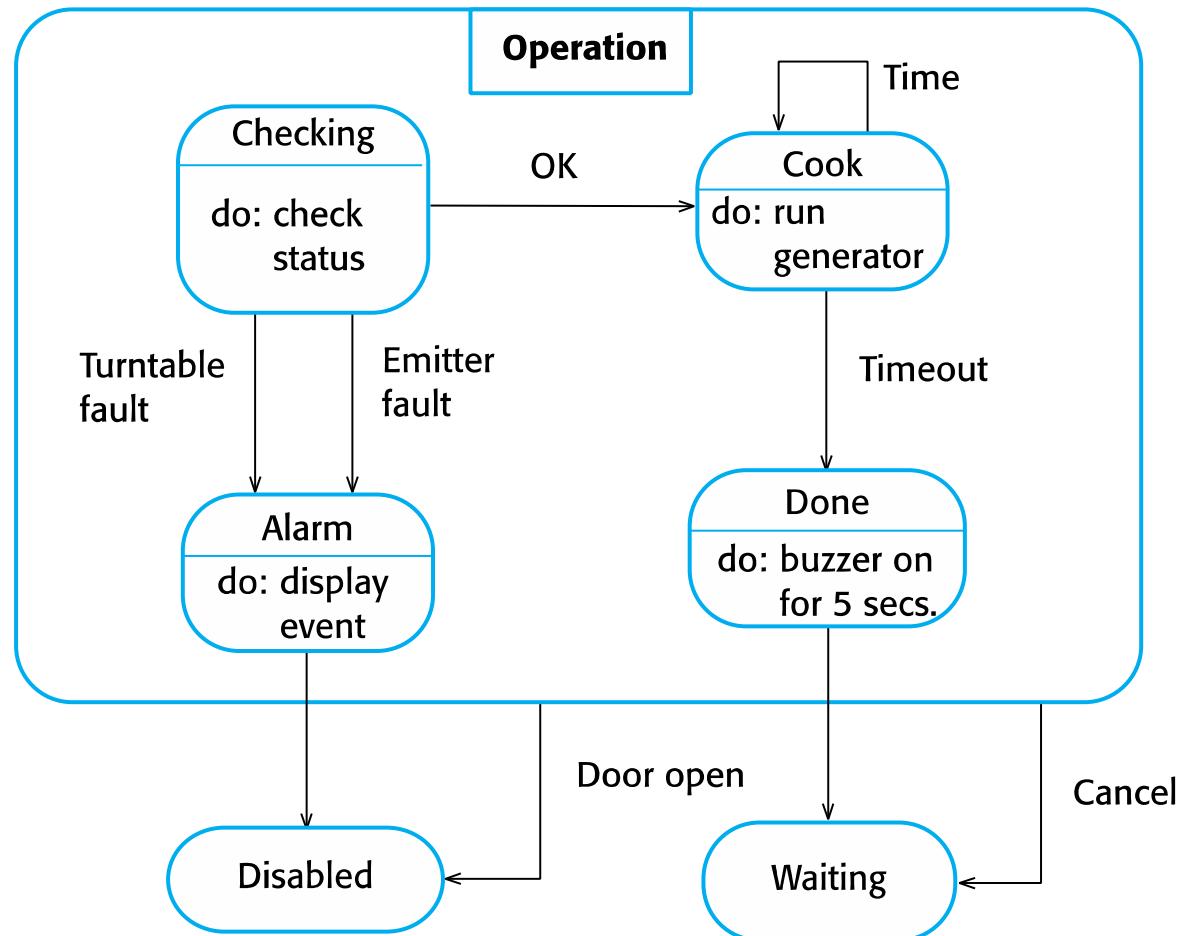
- ✧ Ini memodelkan perilaku sistem dalam menanggapi peristiwa eksternal dan internal.
- ✧ Mereka menunjukkan respons sistem terhadap rangsangan sehingga sering digunakan untuk memodelkan sistem waktu nyata.
- ✧ Model mesin status menunjukkan status sistem sebagai simpul dan peristiwa sebagai busur di antara simpul-simpul ini. Ketika suatu peristiwa terjadi, sistem berpindah dari satu keadaan ke keadaan lain.
- ✧ Statechart merupakan bagian integral dari UML dan digunakan untuk merepresentasikan model state machine.

Diagram keadaan oven microwave





Operasi oven microwave



Keadaan dan rangsangan untuk oven microwave

(a)



Negara	Keterangan
Menunggu	Oven sedang menunggu input. Layar menunjukkan waktu saat ini.
Setengah kekuatan	Daya oven diatur ke 300 watt. Layar menunjukkan 'Setengah daya'.
Kekuatan penuh	Daya oven diatur ke 600 watt. Layar menunjukkan 'Kekuatan penuh'.
Atur waktu	Waktu memasak diatur ke nilai input pengguna. Layar menunjukkan waktu memasak yang dipilih dan diperbarui saat waktu telah diatur.
	Operasi oven dinonaktifkan untuk keselamatan. Lampu oven bagian dalam menyala. Tampilan menunjukkan 'Belum siap'.
Diaktifkan	Operasi oven diaktifkan. Lampu oven bagian dalam mati. Tampilan menunjukkan 'Siap memasak'.
Operasi	Oven sedang beroperasi. Lampu oven bagian dalam menyala. Tampilan menunjukkan penghitung waktu mundur. Setelah selesai memasak, bel berbunyi selama lima detik. Lampu oven menyala. Tampilan menunjukkan 'Memasak selesai' saat bel berbunyi.

Keadaan dan rangsangan untuk oven microwave

(b)



Rangsangan	Keterangan
Setengah kekuatan	Pengguna telah menekan tombol setengah daya.
Kekuatan penuh	Pengguna telah menekan tombol daya penuh.
pengatur waktu	Pengguna telah menekan salah satu tombol pengatur waktu.
Nomor	Pengguna telah menekan tombol numerik.
Pintu terbuka	Saklar pintu oven tidak tertutup.
Pintu tertutup	Saklar pintu oven tertutup.
Awal	Pengguna telah menekan tombol Start.
Membatalkan	Pengguna telah menekan tombol Batal.



Rekayasa berbasis model

Rekayasa berbasis model



- ✧ Model-driven engineering (MDE) adalah **pendekatan untuk pengembangan perangkat lunak di mana model daripada program adalah output utama dari proses pengembangan .**
- ✧ Program yang dijalankan pada platform perangkat keras/perangkat lunak kemudian dihasilkan secara otomatis dari model.

Penggunaan rekayasa berbasis model



✧ kelebihan

- Memungkinkan sistem untuk dipertimbangkan pada tingkat abstraksi yang lebih tinggi
- Menghasilkan kode secara otomatis berarti lebih murah untuk mengadaptasi sistem ke platform baru.

✧ Kontra

- Model untuk abstraksi dan belum tentu tepat untuk diimplementasikan.
- Penghematan dari menghasilkan kode mungkin sebanding dengan biaya pengembangan penerjemah untuk platform baru.

Arsitektur yang digerakkan oleh model



- ✧ Arsitektur berbasis model (MDA) adalah pendahulu dari rekayasa berbasis model yang lebih umum
- ✧ MDA adalah pendekatan yang berfokus pada **model untuk desain dan implementasi perangkat lunak yang menggunakan subset model UML untuk menggambarkan suatu sistem** .
- ✧ Model pada tingkat abstraksi yang berbeda dibuat. Dari model independen platform tingkat tinggi, pada prinsipnya dimungkinkan untuk menghasilkan program kerja tanpa intervensi manual.

Jenis model



✧ Model independen komputasi (CIM)

- Ini memodelkan abstraksi domain penting yang digunakan dalam suatu sistem. CIM terkadang disebut model domain.

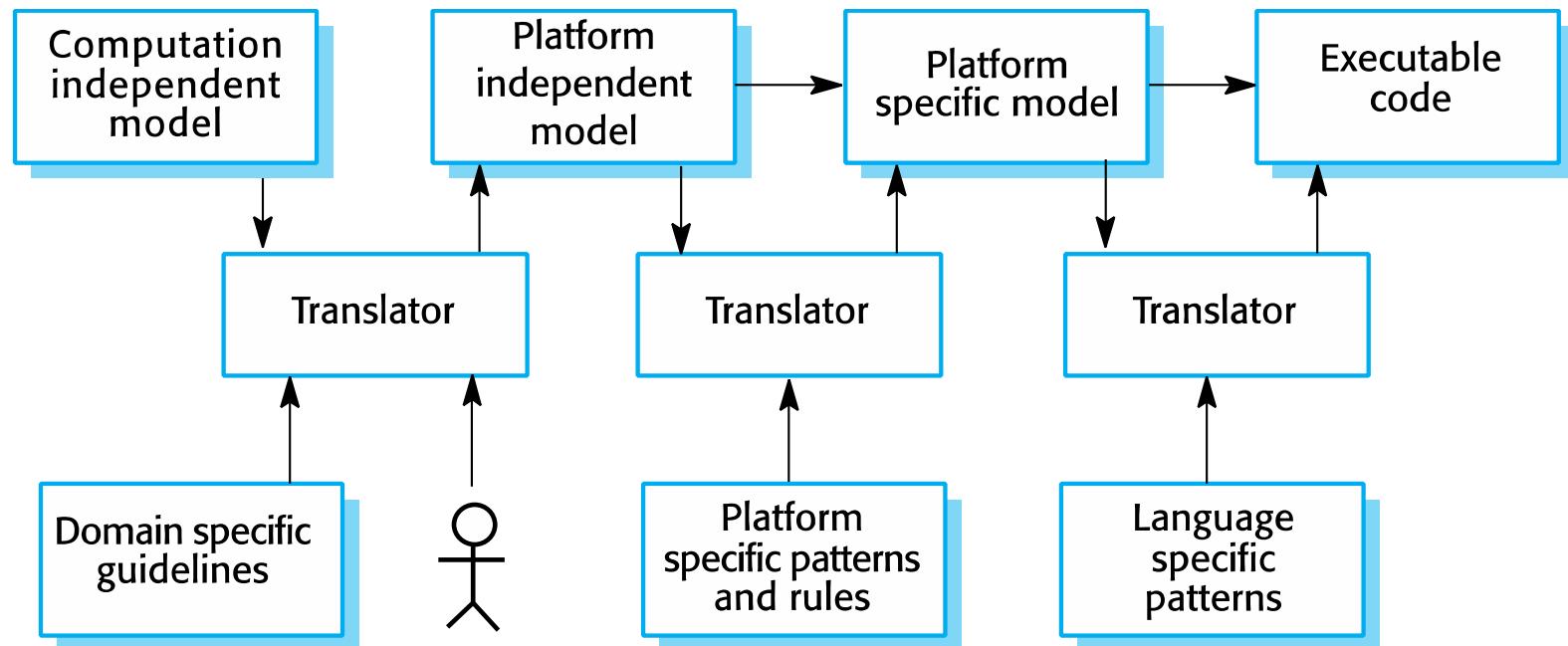
✧ Model independen platform (PIM)

- Ini memodelkan operasi sistem tanpa mengacu pada implementasinya. PIM biasanya dijelaskan menggunakan model UML yang menunjukkan struktur sistem statis dan bagaimana merespons kejadian eksternal dan internal.

✧ Model khusus platform (PSM)

- Ini adalah transformasi model platform-independen dengan PSM terpisah untuk setiap platform aplikasi. Pada prinsipnya, mungkin ada lapisan PSM, dengan setiap lapisan menambahkan beberapa detail spesifik platform.

transformasi MDA





Poin-poin penting

- ✧ Model adalah tampilan abstrak dari sistem yang mengabaikan detail sistem. Model sistem komplementer dapat dikembangkan untuk menunjukkan konteks, interaksi, struktur, dan perilaku sistem .
- ✧ Model konteks menunjukkan bagaimana sistem yang dimodelkan diposisikan dalam lingkungan dengan sistem dan proses lain.
- ✧ Use case diagram dan sequence diagram digunakan untuk menggambarkan interaksi antara pengguna dan sistem dalam sistem yang dirancang. Use case menggambarkan interaksi antara sistem dan aktor eksternal; diagram urutan menambahkan lebih banyak informasi ini dengan menunjukkan interaksi antara objek sistem.
- ✧ Model struktural menunjukkan organisasi dan arsitektur suatu sistem. Diagram kelas digunakan untuk mendefinisikan struktur statis kelas dalam suatu sistem dan asosiasinya.



Poin-poin penting

- ✧ Model perilaku digunakan untuk menggambarkan perilaku dinamis dari sistem yang dijalankan. Perilaku ini dapat dimodelkan dari perspektif data yang diproses oleh sistem, atau oleh peristiwa yang merangsang tanggapan dari suatu sistem.
- ✧ Diagram aktivitas dapat digunakan untuk memodelkan pemrosesan data, di mana setiap aktivitas mewakili satu langkah proses.
- ✧ Diagram keadaan digunakan untuk memodelkan perilaku sistem dalam menanggapi peristiwa internal atau eksternal.
- ✧ Rekayasa berbasis model adalah pendekatan untuk pengembangan perangkat lunak di mana sistem direpresentasikan sebagai seperangkat model yang dapat secara otomatis diubah menjadi kode yang dapat dieksekusi.

DIAGRAM ALIRAN DATA

1. KONSEP PERANCANGAN TERSTRUKTUR

Pendekatan perancangan terstruktur dimulai dari awal tahun 1970. Pendekatan terstruktur dilengkapi dengan alat-alat (*tools*) dan teknik *teknik* yang dibutuhkan dalam pengembangan sistem, sehingga hasil akhir dari sistem yang dikembangkan akan diperoleh sistem yang didefinisikan dengan baik dan jelas.

Melalui pendekatan terstruktur, permasalahan yang kompleks di organisasi dapat memecahkan dan hasil dari sistem akan mudah untuk dipelihara, fleksibel, memuaskan pemakainya, memiliki dokumentasi yang baik, tepat waktu, sesuai dengan anggaran biaya pengembangan, dapat meningkatkan produktivitas dan akan lebih baik. kesalahan)

2. DIAGRAM ALIRAN DATA (DFD)

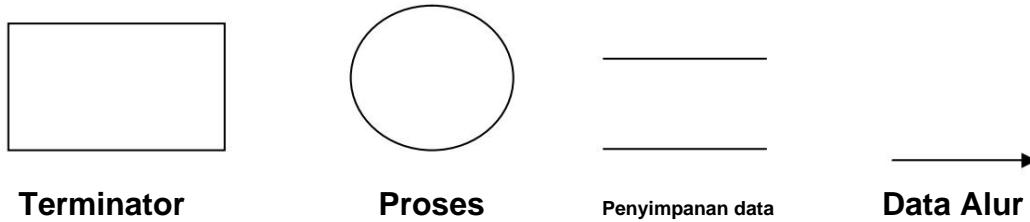
Data Flow Diagram (DFD) adalah alat pembuatan model yang memungkinkan sistem profesional untuk menggambarkan suatu sistem sebagai suatu jaringan fungsional yang sesuai dengan alur data, baik secara manual maupun komputerisasi. DFD ini sering disebut juga dengan nama Bubble chart, Bubble diagram, model proses, diagram alur kerja, atau model fungsi.

DFD ini adalah salah satu alat pembuatan model yang sering digunakan, khususnya bila fungsi-fungsi sistem merupakan bagian yang lebih penting dan kompleks dari data yang dimanipulasi oleh sistem. Dengan kata lain, DFD adalah alat pembuatan model yang memberikan penekanan hanya pada fungsi sistem.

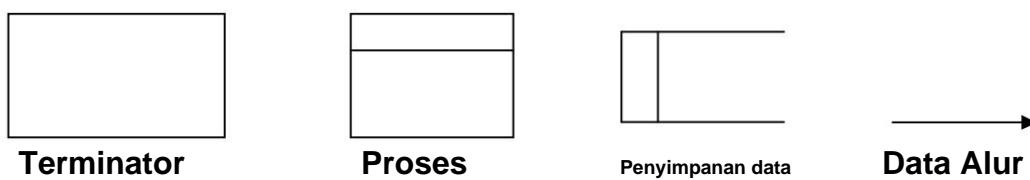
DFD ini merupakan alat perancangan sistem yang berorientasi pada alur data dengan konsep dekomposisi dapat digunakan untuk penggambaran analisis maupun rancangan sistem yang mudah dikomunikasikan oleh sistem profesional kepada pemakai maupun pembuat program.

3. DIAGRAM ALIRAN DATA KOMPONEN

Menurut Yourdan dan DeMarco



Menurut Gene dan Serson

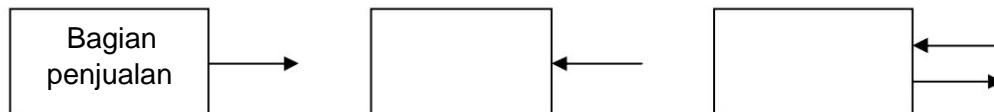


3.1. Komponen Terminator / Entitas Luar

Terminator mewakili entitas eksternal yang berkomunikasi dengan sistem yang sedang dikembangkan. biasanya terminator dikenal dengan nama entitas luar (*external entity*).

Terdapat dua jenis terminator :

1. Terminator Sumber (*source*) : merupakan terminator yang menjadi sumber.
2. Terminator Tujuan (*sink*) : merupakan terminator yang menjadi tujuan data/sistem informasi.



Terminator Sumber Terminator Tujuan T. Tujuan & Sumber

Terminator dapat berupa orang, sekelompok orang, organisasi, departemen di dalam organisasi, atau perusahaan yang sama tetapi di luar kendali sistem yang sedang dibuat modelnya.

Terminator dapat juga berupa departemen, divisi atau sistem di luar sistem yang berkomunikasi dengan sistem yang sedang dikembangkan.

Komponen terminator ini perlu **diberi nama** sesuai dengan dunia luar yang berkomunikasi dengan sistem yang dibuat modelnya, dan biasanya menggunakan **kata benda**, misalnya **Bagian Penjualan, Dosen, Mahasiswa.**

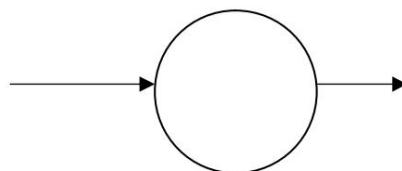
- Ada tiga hal penting yang harus diingat tentang terminator :
1. Terminator merupakan bagian/lingkungan luar sistem. Alur data yang menghubungkan terminator dengan berbagai sistem, menunjukkan hubungan sistem dengan dunia luar.
 2. Sistem profesional tidak dapat mengubah isi atau cara kerja organisasi, atau prosedur yang berkaitan dengan terminator.
 3. Hubungan yang ada antar terminator yang satu dengan yang lain tidak digambarkan pada DFD.

3.2. Komponen Proses

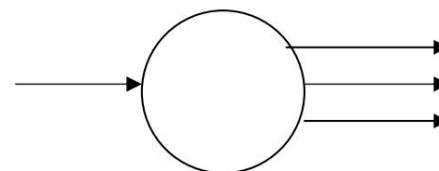
Komponen proses menggambarkan bagian dari sistem yang mentransformasikan input menjadi output.

Proses diberi nama untuk menjelaskan proses/kegiatan apa yang sedang/akan dilaksanakan. Pemberian nama proses dilakukan dengan menggunakan **kata kerja transitif** (kata kerja yang membutuhkan obyek), seperti **Menghitung Gaji, Mencetak KRS, Menghitung Jumlah SKS.**

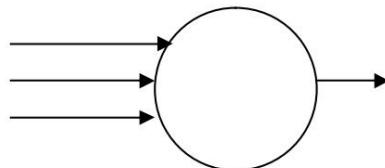
Ada empat kemungkinan yang dapat terjadi dalam proses terkait dengan input dan output :



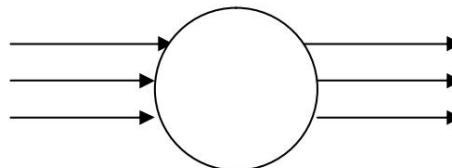
1 masukan & 1 keluaran



1 masukan & banyak keluaran



Banyak masukan & 1 keluaran

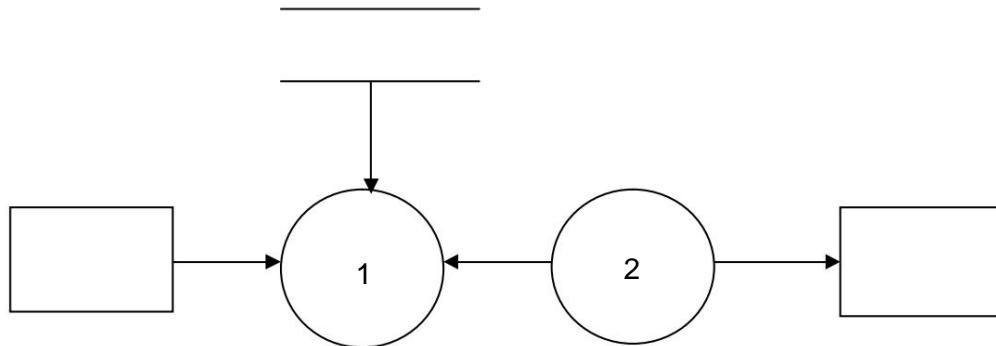


Banyak masukan & banyak keluaran

Ada beberapa hal yang perlu diperhatikan tentang proses : Proses harus memiliki input dan output. Proses dapat melihat dengan terminator komponen, data menyimpan atau proses melalui alur data.

Sistem/bagian/divisi/departemen yang sedang dianalisis oleh sistem profesional yang digambarkan dengan komponen proses.

Berikut ini merupakan suatu contoh proses yang salah :



Gambar 1. Contoh proses

Umumnya kesalahan proses di DFD adalah : 1.

Proses memiliki input tetapi tidak menghasilkan output.

Kesalahan ini disebut dengan **black hole** (lubang hitam), karena data masuk ke dalam proses dan lenyap tidak berbekas seperti dimasukkan ke dalam lubang hitam (*lihat proses 1*).

2. Proses menghasilkan keluaran tetapi tidak pernah menerima masukan.

Kesalahan ini disebut dengan **keajaiban** (ajaib), karena keluaran ajaib yang dihasilkan tanpa pernah menerima masukan (*lihat proses 2*).

3.3. Komponen Penyimpanan Data

Komponen ini digunakan untuk membuat model paket data dan **diberi nama** dengan **kata benda jamak**, misalnya **Mahasiswa**.

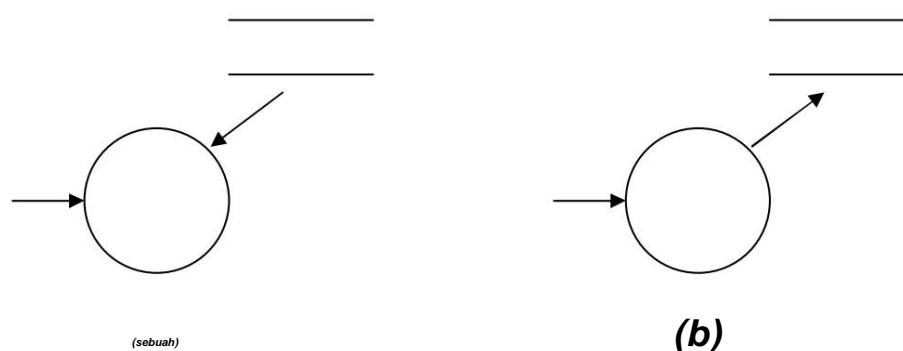
Penyimpanan data ini biasanya berkaitan dengan penyimpanan penyimpanan, seperti file atau database yang berkaitan dengan penyimpanan secara komputerisasi, misalnya file disket, file harddisk, file pita jenis. Penyimpanan data juga berkaitan dengan penyimpanan secara manual seperti buku alamat, folder file, dan agenda.

Suatu tempat penyimpanan data dengan alur data **hanya pada proses komponen**, tidak dengan komponen DFD lainnya. Alur data yang menghubungkan penyimpanan data dengan suatu proses memiliki sebagai berikut : - **Alur data dari data store** yang berarti membaca atau pengaksesan satu paket data tunggal, lebih dari satu paket data, sebagian dari satu paket data, atau sebagian dari lebih satu paket data untuk suatu proses (*lihat gambar 2 (a)*).

- **Alur data ke data store** yang berarti sebagai pengupdatean data, seperti menambah satu paket data baru atau lebih, menghapus satu paket atau lebih, atau mengubah/memodifikasi satu paket data atau lebih (*lihat gambar 2 (b)*).

Pada pengertian pertama dengan jelas bahwa penyimpanan data tidak berubah, jika suatu paket data/informasi berpindah dari penyimpanan data ke suatu proses.

Sebaliknya pada pengertian kedua penyimpanan data berubah sebagai hasil alur yang memasuki penyimpanan data. Dengan kata lain, proses alur data bertanggung jawab terhadap perubahan yang terjadi pada penyimpanan data.



Gambar 2. Implementasi penyimpanan data

3.4. Komponen Aliran Data / Alur Data

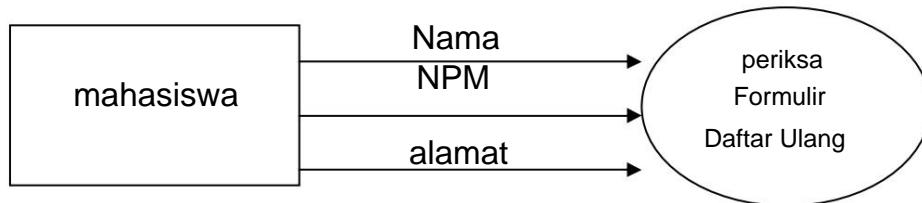
Suatu aliran data / alur data yang digambarkan dengan anak panah, yang menunjukkan arah menuju ke dan keluar dari suatu proses. Alur data ini digunakan untuk menjelaskan data atau paket data/informasi dari satu bagian sistem ke bagian lainnya.

Selain menunjukkan arah, alur data pada model yang dibuat oleh sistem profesional dapat merepresentasikan bit, karakter, pesan, formulir, bilangan real, dan macam-macam informasi yang berkaitan dengan komputer. Alur data juga dapat merepresentasikan data/informasi yang tidak berkaitan dengan komputer.

Alur data perlu **diberi nama** sesuai dengan data/informasi yang dimaksud, biasanya pemberian nama pada alur data dilakukan dengan menggunakan **kata benda**, contohnya **Laporan Penjualan**.

Ada empat konsep yang perlu diperhatikan dalam penggambaran alur data, yaitu : **1. Konsep Paket Data (Packets of Data)**

Jika dua data atau lebih dari *suatu sumber yang sama* menuju ke *tujuan yang sama* dan memiliki hubungan, dan harus dianggap sebagai satu alur data tunggal, karena data itu mengalir bersama-sama sebagai satu paket.



(a) Konsep paket data yang salah

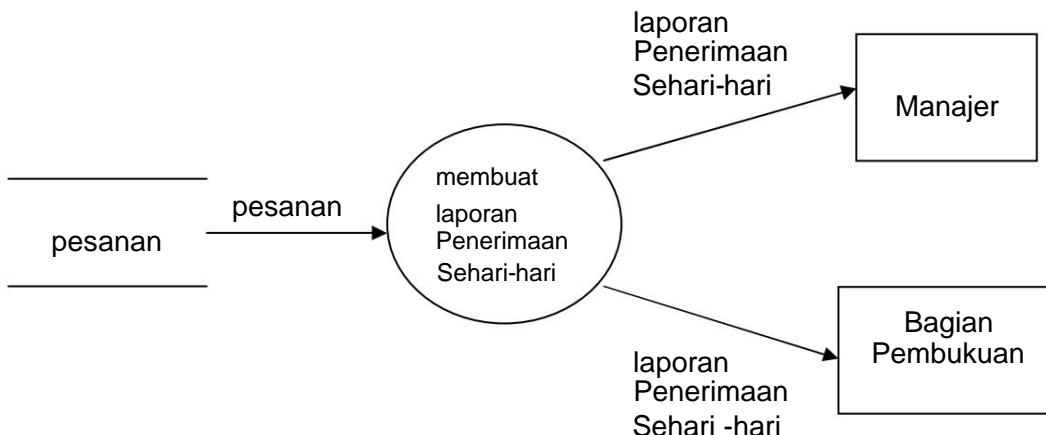


(b) Konsep paket data yang benar

Gambar 3. Konsep paket data

2. Konsep Alur Data Menyebar (Diverging Data Flow)

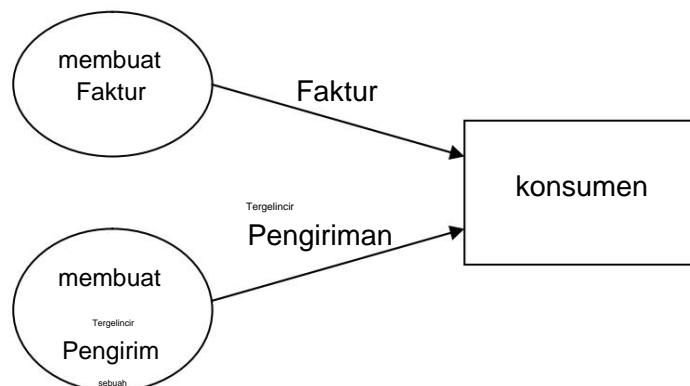
Alur data menunjukkan jumlah tembusan paket data yang berasal dari *sumber yang sama* menuju *tujuan yang berbeda*, atau paket data yang kompleks dibagi menjadi beberapa elemen data yang dikirim ke tujuan yang berbeda, atau alur data ini membawa paket data yang memiliki nilai yang berbeda. yang akan dikirim ke tujuan yang berbeda.



Gambar 4. Konsep alur data menyebar

3. Konsep Alur Data Mengumpul (**Converging Data Flow**)

Beberapa alur data yang berbeda sumber bergabung bersama menuju *tujuan yang sama*.

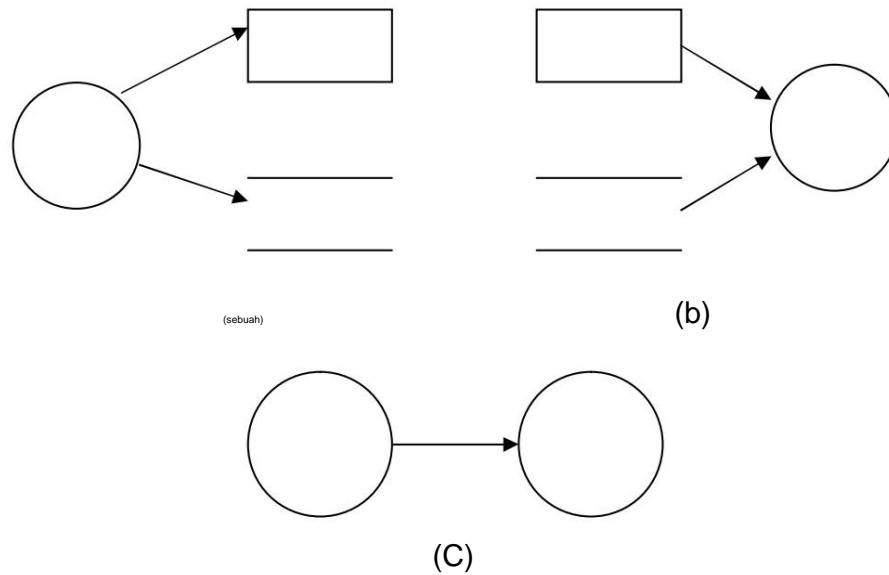


Gambar 5. Konsep alur data mengumpul

4. Konsep Sumber atau Tujuan Alur Data

Semua alur data harus **minimal mengandung satu proses**. Maksud kalimat ini adalah :

- q Suatu alur data yang dihasilkan dari suatu proses dan menuju ke suatu *data store* dan/atau *terminator* (*lihat gambar 6 (a)*). q Suatu alur data yang dihasilkan dari suatu *data store* dan/atau *terminator* dan menuju ke suatu proses (*lihat gambar 6 (b)*). q Suatu alur data yang dihasilkan dari suatu proses dan menuju ke suatu proses (*lihat gambar 6 (c)*).



Gambar 6. Konsep sumber atau tujuan alur data

4. DIAGRAM ALIRAN DATA BENTUK

Terdapat dua bentuk DFD, yaitu **Diagram Alur Data Fisik**, dan **Diagram Alur data Logika**. Diagram alur data yang diterapkan pada bagaimana proses dari sistem, sedangkan diagram alur data yang diterapkan di sistem.

4.1. Diagram Alur Data Fisik (DADF)

DADF lebih tepat digunakan untuk menggambarkan sistem yang ada (sistem yang lama). Penekanan dari DADF adalah bagaimana proses-proses dari sistem diterapkan (dengan cara apa, oleh siapa dan dimana), termasuk proses-proses manual.

Untuk memperoleh gambaran bagaimana sistem yang ada diterapkan, DADF harus memuat :

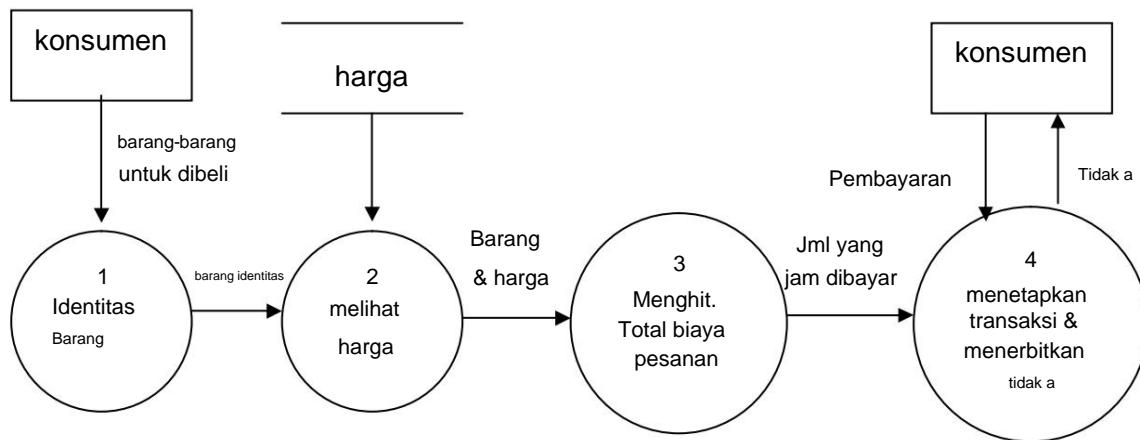
1. Proses-proses manual juga digambarkan.

2. Nama yang cukup dari alur data harus memuat keterangan yang terinci untuk menunjukkan bagaimana pemakai memahami sistem kerja.
3. Simpanan data dapat menunjukkan simpanan non komputer.
4. Nama dari simpanan data harus menunjukkan jenis penerapannya apakah secara manual atau komputerisasi. Secara manual misalnya dapat menunjukkan buku katatan, meja pekerja. Sedang cara komputerisasi misalnya menunjukkan file urut, file database.

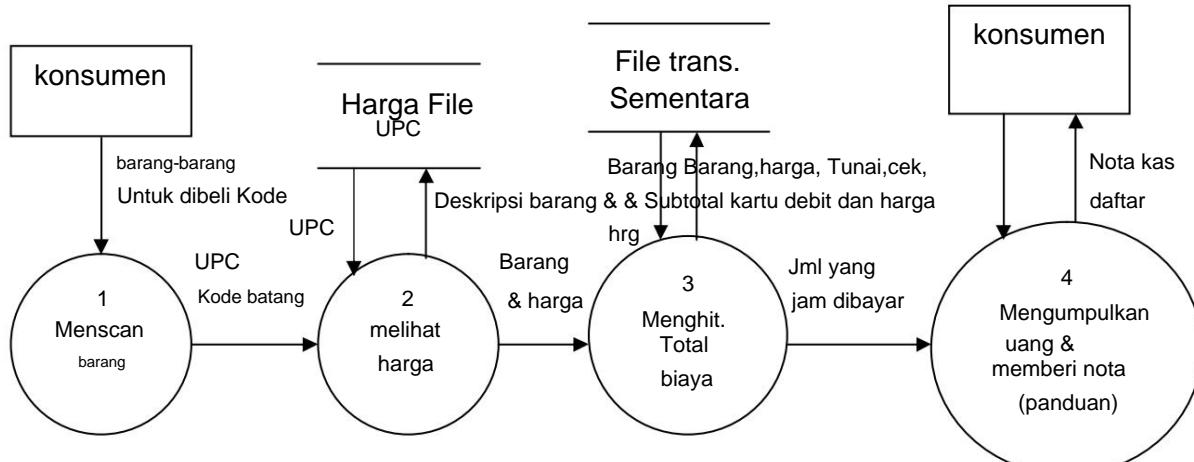
5. Proses harus menunjukkan nama dari pemroses, yaitu orang, departemen, sistem komputer, atau nama program komputer yang mengakses proses tersebut.

4.2. Diagram Alur Data Logika (DADL)

DADL lebih tepat digunakan untuk menggambarkan sistem yang akan diusulkan (sistem yang baru). Untuk sistem komputerisasi, penggambaran DADL hanya menunjukkan kebutuhan proses dari sistem yang logis secara logika, biasanya proses-proses yang digambarkan hanya merupakan proses-proses secara komputer saja.



(a) Diagram Alur Data Logika



(b) Diagram Alur Data Fisik

Gambar 7. DADF dan DADL

5. DIAGRAM ALIRAN DATA SYARAT-SYARAT PEMBUATAN

Syarat pembuatan DFD ini akan membantu sistem profesional untuk menghindari pembentukan DFD yang salah atau DFD yang tidak lengkap atau tidak konsisten secara logika. Beberapa syarat pembuatan DFD dapat membantu sistem profesional untuk membentuk DFD yang benar, menyenangkan untuk dilihat dan mudah dibaca oleh pemakai.

Syarat-syarat pembuatan DFD ini adalah :

1. Pemberian nama untuk setiap komponen DFD
2. Pemberian nomor pada komponen proses
3. Penggambaran DFD sesering mungkin agar enak dilihat
4. Penghindaran penggambaran DFD yang rumit
5. Pemastian DFD yang dibentuk itu konsisten secara logika

5.1. Pemberian Nama untuk Tiap DFD

Sebagaimana yang telah dijelaskan sebelumnya, terminator komponen mewakili lingkungan luar dari sistem, tetapi memiliki pengaruh terhadap sistem yang sedang dikembangkan ini. Maka agar pemakai mengetahui dengan lingkungan mana saja sistem mereka berhubungan, komponen terminator ini harus diberi nama sesuai dengan lingkungan luar yang mempengaruhi sistem ini. Biasanya terminator komponen diberi nama dengan kata benda.

Selanjutnya adalah komponen proses. Komponen proses ini mewakili fungsi sistem yang akan dilaksanakan atau menunjukkan bagaimana fungsi sistem yang dilaksanakan oleh seseorang, sekelompok orang atau mesin. Maka jelaslah bahwa komponen ini perlu diberi nama yang tepat, agar siapa pun yang membaca DFD khususnya pemakai akan merasa yakin bahwa DFD yang dibentuk ini adalah model yang akurat.

Pemberian nama pada komponen proses lebih baik menunjukkan aturan-aturan yang akan dilaksanakan oleh seseorang dibandingkan dengan memberikan nama atau identitas orang yang akan melaksanakannya. Ada dua alasan mengapa bukan nama atau identitas orang (yang melaksanakan fungsi sistem) yang digunakan sebagai proses nama, yaitu : 1. Orang tersebut mungkin diganti oleh orang lain saat mendatang, sehingga bila setiap kali ada pergantian orang yang melaksanakan fungsi tersebut, maka sistem yang dibentuk harus diubah lagi.

2. Orang tersebut mungkin tidak menjalankan satu fungsi sistem saja, melainkan beberapa fungsi sistem yang berbeda. Daripada menggambarkan beberapa proses dengan nama yang sama tetapi artinya berbeda, lebih baik merencanakan dengan tugas/fungsi sistem yang sebenarnya akan dilaksanakan.

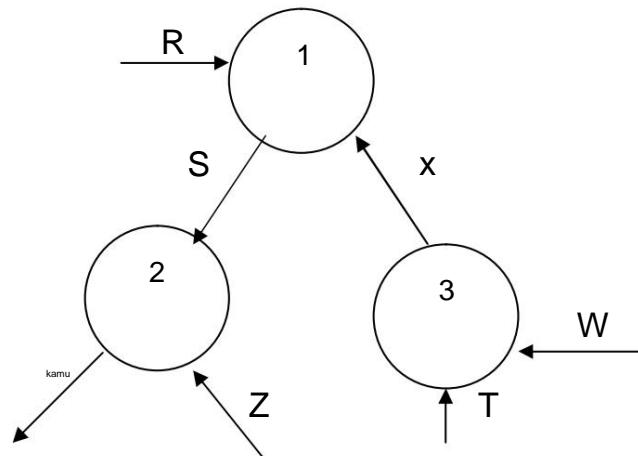
Karena tugas nama untuk komponen proses lebih baik menunjukkan/fungsi sistem yang akan dilaksanakan, maka lebih baik mempersesembahkan nama ini menggunakan kata kerja transitif.

Pemberian nama komponen untuk penyimpanan data menggunakan kata benda, karena penyimpanan data menunjukkan data apa yang disimpan untuk kebutuhan sistem dalam melaksanakan tugas. Jika sistem sewaktu-waktu data tersebut untuk melaksanakan tugas, karena data tersebut tetap ada, sistem menyimpannya.

Beginu pula untuk komponen alur data, namanya lebih baik diberikan dengan menggunakan kata benda. Karena alur data ini menunjukkan data dan informasi yang dibutuhkan dan dikeluarkan oleh sistem saat pelaksanaan.

5.2. Pemberian Nomor pada Komponen Proses

Biasanya sistem profesional memberikan nomor dengan bilangan terurut pada komponen proses sebagai referensi. Tidak jadi masalah nomor-nomor proses ini diberikan. Nomor proses dapat diberikan dari kiri ke kanan, atau dari atas ke bawah, atau dapat pula dilakukan dengan pola-pola tertentu selama pemberian nomor ini tetap konsisten pada nomor yang digunakan.



Gambar 8. Contoh Pemberian nomor pada proses

Nomor-nomor proses yang diberikan terhadap komponen proses ini tidak boleh dikatakan bahwa proses tersebut dilaksanakan secara berurutan. Pemberian nomor ini agar suatu pembacaan proses dalam suatu diskusi akan lebih mudah dengan hanya menyebutkan prosesnya saja jika dibandingkan dengan menyebutkan nama prosesnya, jika nama prosesnya panjang dan sulit.

Maksud pemberian nomor pada proses yang lebih penting lagi adalah untuk menunjukkan referensi terhadap skema penomoran secara hierarki pada levelisasi DFD. Dengan kata lain, jumlah proses ini merupakan dasar pemberian nomor pada levelisasi DFD (*lihat gambar 11*).

5.3. Penggambaran DFD sesering mungkin

Penggambaran DFD dapat dilakukan berkali-kali sampai secara teknik DFD itu benar, dapat diterima oleh pemakai, dan sudah cukup rapi sehingga profesional sistem tidak merasa malu untuk menunjukkan DFD kepada atasannya dan pemakai.

Dengan kata lain, penggambaran DFD ini dilakukan sampai terbentuk DFD yang enak dilihat, dan mudah dibaca oleh pemakai dan sistem profesional lainnya. Keindahan penggambaran DFD tergantung pada standar-standar yang diminta oleh organisasi tempat profesional sistem itu bekerja dan perangkat lunak yang dipakai oleh sistem profesional dalam membuat DFD.

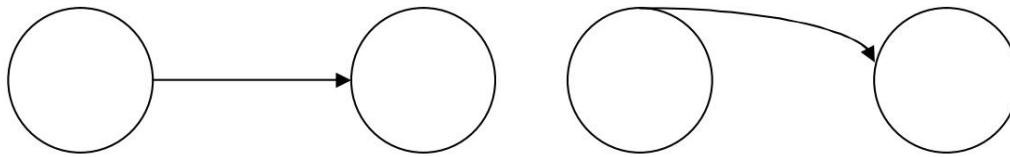
Penggambaran yang enak untuk dilihat dapat dilakukan dengan memperhatikan hal-hal berikut ini :

Ukuran *dan bentuk proses*.

Beberapa pemakai kadang-kadang merasa bingung bila ukuran proses yang berbeda dengan proses yang lain. Mereka akan mengira bahwa proses dengan ukuran yang lebih besar akan dilupakan lebih penting dari proses yang lebih kecil. Hal ini sebenarnya hanya karena nama proses itu lebih panjang dibandingkan dengan proses yang lain. Jadi, sebaiknya proses yang digambarkan memiliki ukuran dan bentuk yang sama.

Alur *data dan alur data lurus*.

Data alur dapat digambarkan dengan gambar atau hanya garis lurus. Mana yang lebih enak dilihat tergantung siapa yang akan melihat DFD tersebut.



(Sebuah). Alur data dengan garis lurus (b). Alur data dengan **gambar 9**

DFD dengan gambar tangan dan gambar menggunakan mesin.

DFD dapat digambarkan secara manual atau dengan menggunakan bantuan mesin, tergantung pilihan pengguna atau sistem profesional.

5.4. Penghindaran Penggambaran DFD yang rumit

Tujuan DFD adalah untuk membuat model fungsi yang harus dilaksanakan oleh suatu sistem dan interaksi antar fungsi. Tujuan lainnya adalah agar model yang dibuat itu mudah dibaca dan dipahami tidak hanya oleh sistem profesional yang membuat DFD, tetapi juga pemakai yang berpengalaman dengan subyek yang terjadi. Hal ini berarti DFD harus mudah dipahami, dibaca, dan menyenangkan untuk dilihat.

Pada banyak masalah, DFD yang dibuat tidak memiliki terlalu banyak proses dengan penyimpanan data, alur data, dan terminator yang berkaitan dengan proses tersebut dalam satu diagram.

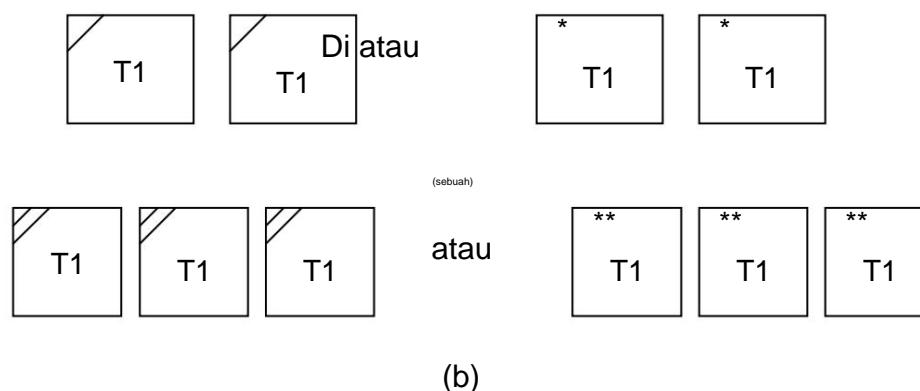
Bila terlalu banyak proses, terminator, data store, dan alur data yang digambarkan dalam satu DFD, maka ada kemungkinan terjadi banyak persilangan alur data dalam DFD tersebut. Persilangan alur data ini menyebabkan pemakai akan sulit membaca dan memahami DFD yang terbentu. Jadi semakin sedikit adanya persilangan data pada DFD, maka semakin baik DFD yang dibentuk oleh sistem yang profesional.

Persilangan alur data ini dapat dihindari dengan menggambarkan DFD secara bertingkat (levelisasi DFD), atau dengan menggunakan duplikat terhadap komponen DFD.

Komponen DFD yang **dapat menggunakan duplikat** hanya **komponen store** dan **terminator**. Pemberian duplikat ini juga tidak dapat diberikan sesuka hati sistem yang membuat DFD, tetapi semakin sedikit penggunaan duplikat, semakin baik DFD yang terbentuk.

Pemberian duplikat terhadap penyimpanan data dilakukan dengan memberikan simbol garis lurus (x) atau asterik (*), sedangkan untuk terminator menggunakan simbol garis miring (/) atau asterik (*).

Banyaknya mempersempitakan simbol duplikat pada duplikat yang digunakan tergantung pada banyaknya duplikat yang digunakan.



Gambar 10. Contoh penggunaan simbol duplikat pada komponen terminator (a) Satu duplikat yang digunakan (b) Dua duplikat yang digunakan

5.5. Penggambaran DFD yang Konsisten

Penggambaran DFD harus konsisten terhadap kelompok DFD lainnya. Sistem profesional menggambarkan DFD berdasarkan tingkat DFD dengan tujuan agar DFD yang dibuatnya mudah dibaca dan dipahami oleh pemakai sistem. Hal ini sesuai dengan salah satu tujuan atau syarat membuat DFD.

6. PENGGAMBARAN DFD

Tidak ada aturan baku untuk menggambarkan DFD. Tapi dari berbagai referensi yang ada, secara garis besar langkah untuk membuat DFD adalah :

- Identifikasi terlebih dahulu semua entitas luar yang terlibat di

sistem.

2. Identifikasi semua input dan output yang terlibat dengan entitas luar.

3. Buat Diagram Konteks (*diagram konteks*)

Diagram ini adalah diagram level tertinggi dari DFD yang menggambarkan hubungan sistem dengan lingkungan luarnya. Caranya :

- q Tentukan nama sistemnya. q
- Tentukan batasan sistemnya. q
- Tentukan terminator apa saja yang ada dalam sistem. q
- Tentukan apa yang diterima/diberikan terminator dari/ke sistem. q
- Gambarkan diagram konteks.

4. Buat Diagram Level Zero

Diagram ini adalah dekomposisi dari diagram konteks.

Caranya :

- q Tentukan proses utama yang ada pada sistem. q
- Tentukan apa yang diberikan/diterima masing-masing proses ke/dari sistem sambil memperhatikan konsep keseimbangan (alur data yang keluar/masuk dari suatu level harus sama dengan alur data yang masuk/keluar pada level berikutnya). q Jika diperlukan, munculkan data store (master) sebagai sumber maupun tujuan alur data.
- q Gambarkan diagram level nol.
 - kendala perpotongan data arus
 - Beri nomor pada proses utama (nomor tidak menunjukkan urutan proses).

5. Buat Diagram Level Satu

Diagram ini merupakan dekomposisi dari diagram level nol.

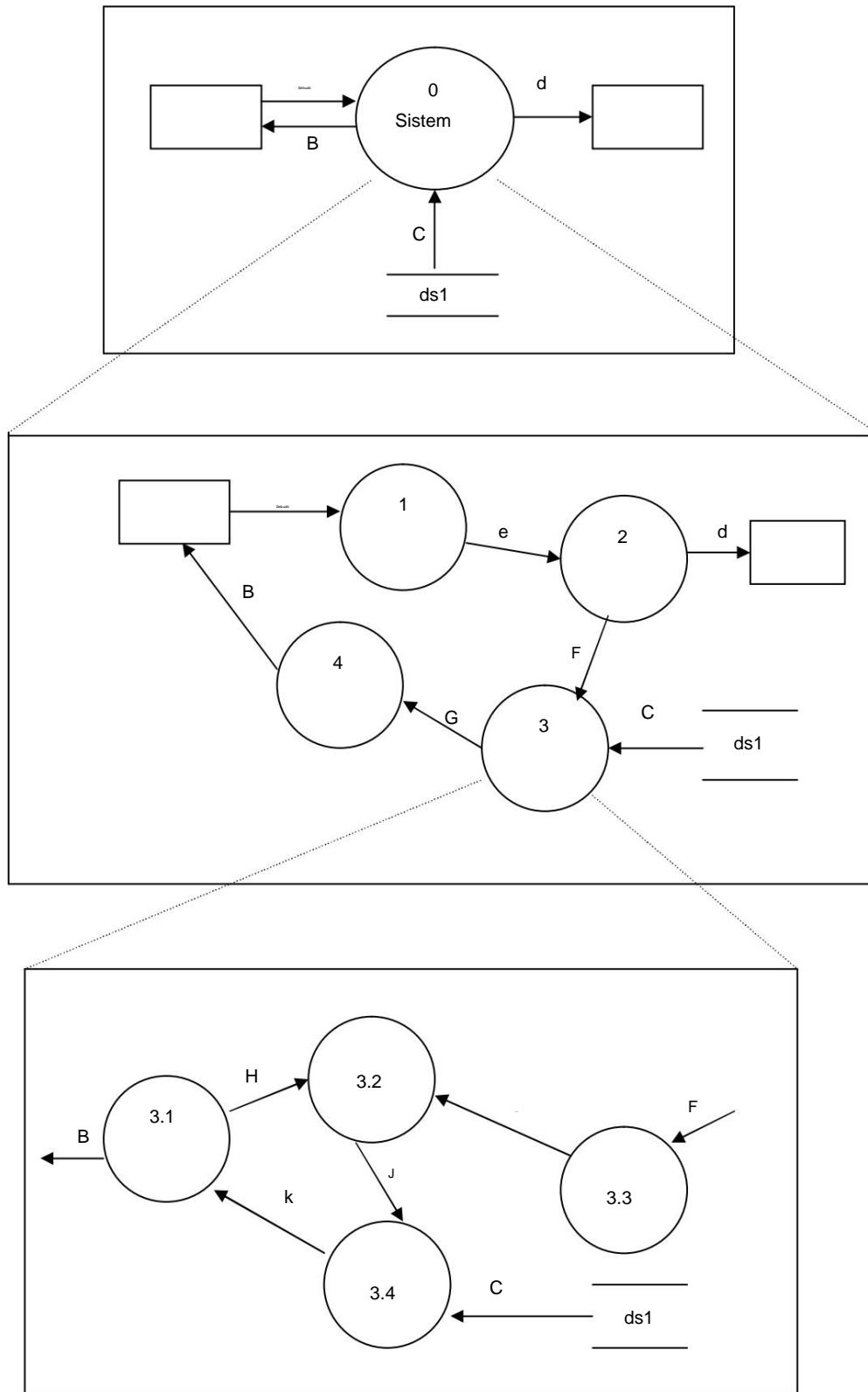
Caranya :

- q Tentukan proses yang lebih kecil (sub-proses) dari proses utama yang ada di level nol. q Tentukan apa yang diberikan/diterima masing-masing sub proses ke/dari sistem dan perhatikan konsep keseimbangan. q Jika diperlukan, munculkan data store (transaksi) sebagai sumber maupun tujuan alur data.
 - q Gambarkan DFD level Satu
 - mengatasi perpotongan data arus.
 - Beri nomor pada masing-masing sub-proses yang menunjukkan dekomposisi dari proses sebelumnya.
- Contoh : 1.1, 1.2, 2.1

6. DFD Level Dua, Tiga, «

Diagram ini merupakan dekomposisi dari level sebelumnya.

Proses dekomposisi dilakukan sampai dengan proses siap ke dalam program. Aturan yang digunakan sama dengan level satu.



Gambar 11. Levelisasi DFD