# Menu

[ Menu 1](#_Toc154914699)

[ **SOFTWARE ENGINEERING** 7](#_Toc154914700)

[1. History 7](#_Toc154914701)

[2. Basic Concepts 7](#_Toc154914702)

[Software costs 7](#_Toc154914703)

[Frequently asked questions about software engineering 8](#_Toc154914704)

[Frequently asked questions about software engineering 8](#_Toc154914705)

[Essential attributes of good software 8](#_Toc154914706)

[Software engineering 9](#_Toc154914707)

[Importance of software engineering 9](#_Toc154914708)

[Software process activities 9](#_Toc154914709)

[3. General Issues That Affect Most Software 9](#_Toc154914710)

[Software engineering diversity 9](#_Toc154914711)

[4. Classification 10](#_Toc154914712)

[Software products 10](#_Toc154914713)

[Product specification 10](#_Toc154914714)

[Application types 10](#_Toc154914715)

[Software engineering fundamentals 10](#_Toc154914716)

[Internet software engineering 11](#_Toc154914717)

[Web-based software engineering 11](#_Toc154914718)

[Web software engineering 11](#_Toc154914719)

[5. Software Industry In Vietnam 11](#_Toc154914720)

[Key points 11](#_Toc154914721)

[** REQUIREMENTS FOR SOFTWARE ENGINEERS 12**](#_Toc154914722)

[1. SOFTWARE ENGINEERING ETHICS - RULES & REGULATIONS 12](#_Toc154914723)

[Issues of professional responsibility 12](#_Toc154914724)

[ACM/IEEE Code of Ethics 12](#_Toc154914725)

[Rationale for the code of ethics 12](#_Toc154914726)

[The ACM/IEEE Code of Ethics 12](#_Toc154914727)

[Ethical principles 13](#_Toc154914728)

[Ethical dilemmas 13](#_Toc154914729)

[2. REQUIRED KNOWLEDGE, SKILLS & ATTITUDE 13](#_Toc154914730)

[ **SOFTWARE PROCESSES** 14](#_Toc154914731)

[1. WHAT IS SOFTWARE PROCESS? 14](#_Toc154914732)

[Code-and-Fix 14](#_Toc154914733)

[The software process 14](#_Toc154914734)

[Software process descriptions 14](#_Toc154914735)

[2. SOFTWARE PROCESS CLASSIFICATION 14](#_Toc154914736)

[Plan-driven and agile processes 14](#_Toc154914737)

[3. PROCESS ACTIVITIES 15](#_Toc154914738)

[Process activities 15](#_Toc154914739)

[Software specification 15](#_Toc154914740)

[The requirements engineering process 15](#_Toc154914741)

[Software design and implementation 15](#_Toc154914742)

[Software implementation 15](#_Toc154914743)

[Software validation 15](#_Toc154914744)

[Stages of testing 16](#_Toc154914745)

[Testing stages 16](#_Toc154914746)

[Software evolution 16](#_Toc154914747)

[System evolution 16](#_Toc154914748)

[4. THE WATERFALL MODEL 16](#_Toc154914749)

[Waterfall model phases 16](#_Toc154914750)

[Waterfall model benefits 17](#_Toc154914751)

[Waterfall model problems 17](#_Toc154914752)

[Coping with change 17](#_Toc154914753)

[Reducing the costs of rework 17](#_Toc154914754)

[5. THE INCREMENTAL DEVELOPMENT 17](#_Toc154914755)

[Incremental delivery 17](#_Toc154914756)

[Incremental development 17](#_Toc154914757)

[Incremental development benefits 18](#_Toc154914758)

[Incremental development problems 19](#_Toc154914759)

[When not to use? 19](#_Toc154914760)

[6. PROTOTYPE 20](#_Toc154914761)

[Software prototyping 20](#_Toc154914762)

[The process of prototype development 20](#_Toc154914763)

[Prototype development 20](#_Toc154914764)

[Benefits of prototyping 20](#_Toc154914765)

[Problem of Prototyping 20](#_Toc154914766)

[Throw-away prototypes 20](#_Toc154914767)

[7. SCRUM 21](#_Toc154914768)

[3 Roles 21](#_Toc154914769)

[3 Artifacts 21](#_Toc154914770)

[Burndown Chart 22](#_Toc154914771)

[4 Events 22](#_Toc154914772)

[Sprint Planning (8 hours / sprint) 22](#_Toc154914773)

[Daily Scrum (15 mins) 22](#_Toc154914774)

[Sprint Review (4 hours / sprint) 22](#_Toc154914775)

[Sprint Retrospective (3 hours / sprint) 22](#_Toc154914776)

[3 pillars 23](#_Toc154914777)

[Sprint Process 23](#_Toc154914778)

[** REQUIREMENTS ENGINEERING 24**](#_Toc154914779)

[1. WHAT IS REQUIREMENT? 24](#_Toc154914780)

[Requirements engineering 24](#_Toc154914781)

[A Requirement: 24](#_Toc154914782)

[2. TYPES OF REQUIREMENT 24](#_Toc154914783)

[User and system requirements 24](#_Toc154914784)

[Readers of different types of requirements specification 24](#_Toc154914785)

[Functional and non-functional requirements 25](#_Toc154914786)

[3. FUNCTIONAL REQUIREMENTS 25](#_Toc154914787)

[Mentcare system: functional requirements 25](#_Toc154914788)

[Requirements imprecision 25](#_Toc154914789)

[Requirements completeness and consistency 25](#_Toc154914790)

[4. NON-FUNCTIONAL REQUIREMENTS 26](#_Toc154914791)

[Types of nonfunctional requirement 26](#_Toc154914792)

[Non-functional classifications 26](#_Toc154914793)

[Non-functional requirements implementation 26](#_Toc154914794)

[Goals and requirements 26](#_Toc154914795)

[Usability requirements 27](#_Toc154914796)

[Metrics for specifying nonfunctional requirements 27](#_Toc154914797)

[5. REQUIREMENTS ENGINEERING PROCESS 28](#_Toc154914798)

[Requirements elicitation and analysis 28](#_Toc154914799)

[Requirements elicitation 28](#_Toc154914800)

[Problems of requirements elicitation 28](#_Toc154914801)

[The requirements elicitation and analysis process 28](#_Toc154914802)

[Process activities 28](#_Toc154914803)

[Requirements discovery 29](#_Toc154914804)

[System stakeholders 29](#_Toc154914805)

[Stakeholders in the Mentcare system 29](#_Toc154914806)

[Stakeholders in the Mentcare system 29](#_Toc154914807)

[Requirements discovery 29](#_Toc154914808)

[6. METHODS FOR REQUIREMENTS DISCOVERY 30](#_Toc154914809)

[Interviewing 30](#_Toc154914810)

[Interviews in practice 30](#_Toc154914811)

[Questions during interview 30](#_Toc154914812)

[Questions during interview 30](#_Toc154914813)

[Questions during interview 30](#_Toc154914814)

[Questions during interview 31](#_Toc154914815)

[Ethnography 31](#_Toc154914816)

[Scope of ethnography 31](#_Toc154914817)

[Stories and scenarios 31](#_Toc154914818)

[7. REQUIREMENTS SPECIFICATION 32](#_Toc154914819)

[Requirements specification 32](#_Toc154914820)

[Ways of writing a system requirements specification 32](#_Toc154914821)

[The software requirements document 32](#_Toc154914822)

[Users of a requirements document 33](#_Toc154914823)

[Natural language specification 33](#_Toc154914824)

[Guidelines for writing requirements 33](#_Toc154914825)

[Problems with natural language 33](#_Toc154914826)

[Structured specifications 33](#_Toc154914827)

[Form-based specifications 33](#_Toc154914828)

[Tabular specification 34](#_Toc154914829)

[Use cases 34](#_Toc154914830)

[Requirements validation 34](#_Toc154914831)

[Requirements checking 34](#_Toc154914832)

[Requirements validation techniques 34](#_Toc154914833)

[ **UML** 35](#_Toc154914834)

[1. What is UML ? 35](#_Toc154914835)

[Why UML? 35](#_Toc154914836)

[2. USE-CASE MODELING 35](#_Toc154914837)

[Objectives 35](#_Toc154914838)

[What Is Use-Case Modeling? 35](#_Toc154914839)

[ ACTIVITY DIAGRAM 37](#_Toc154914840)

[Activity Diagrams 37](#_Toc154914841)

[Activities 37](#_Toc154914842)

[Actions 37](#_Toc154914843)

[Transitions 37](#_Toc154914844)

[Initial Nodes 37](#_Toc154914845)

[Final Nodes 37](#_Toc154914846)

[Decision nodes 37](#_Toc154914847)

[Decision nodes > Guard conditions 37](#_Toc154914848)

[Merge nodes 38](#_Toc154914849)

[Synch states 38](#_Toc154914850)

[Fork Nodes 38](#_Toc154914851)

[Join Nodes 38](#_Toc154914852)

[Swimlanes 38](#_Toc154914853)

[ **SOFTWARE DESIGN** 39](#_Toc154914854)

[A general model of the design process 39](#_Toc154914855)

[Design activities 39](#_Toc154914856)

[ **CONFIGURATION MANAGEMENT** 40](#_Toc154914857)

[1. CM ACTIVITIES 41](#_Toc154914858)

[Configuration management activities 41](#_Toc154914859)

[Multi-version systems 41](#_Toc154914860)

[2. CHANGE MANAGEMENT 41](#_Toc154914861)

[The change management process 42](#_Toc154914862)

[Factors in change analysis 42](#_Toc154914863)

[3. VERSION MANAGEMENT 42](#_Toc154914864)

[Codelines and baselines 42](#_Toc154914865)

[Baselines 42](#_Toc154914866)

[Version control systems 43](#_Toc154914867)

[Centralized version control 43](#_Toc154914868)

[Repository Check-in/Check-out 43](#_Toc154914869)

[Distributed version control 44](#_Toc154914870)

[Benefits of distributed version control 44](#_Toc154914871)

[Open source development 44](#_Toc154914872)

[Version management systems 44](#_Toc154914873)

[Version management systems 45](#_Toc154914874)

[Storage management using deltas 45](#_Toc154914875)

[Branching and merging 45](#_Toc154914876)

[4. SYSTEM BUILDING 45](#_Toc154914877)

[Build platforms 45](#_Toc154914878)

[Development, build, and target platforms 45](#_Toc154914879)

[System building 46](#_Toc154914880)

[Build system functionality 46](#_Toc154914881)

[5. RELEASE MANAGEMENT 46](#_Toc154914882)

[Release tracking 46](#_Toc154914883)

[Release reproduction 46](#_Toc154914884)

[Release planning 46](#_Toc154914885)

[Factors influencing system release planning 47](#_Toc154914886)

[Release components 47](#_Toc154914887)

[Key points 47](#_Toc154914888)

# SOFTWARE ENGINEERING

• The economies of ALL developed nations are dependent on software.

• More and more systems are software controlled.

• Software engineering is concerned with theories, methods, and tools for professional software development.

• Expenditure on software represents a significant fraction of GNP in all developed countries.

## History

1968: Nato conference on software engineering. Discuss what was then called the ‘software crisis’.

Early 1970s. Development of the notions of structured programming. Development of Pascal programming language. Development of Smalltalk languages which

introduced notions of object-oriented development.

Late 1970s. Early use of software design methods. Development of first programming environments.

Early 1980s. Development of the Ada programming language which included notions of structured programming and information hiding. Proposals for software

engineering environments. CASE tools introduced to support design methods. Development of algorithmic approaches to software costing and estimation. Publication of

the 1st edition of this book as the first student textbook on software engineering.

Late 1980s. Increased use of object-oriented programming through languages such as C++ and Objective-C. Introduction of object-oriented design methods. Extensive

use of CASE tools.

Early 1990s. Object-oriented development becomes a mainstream development technique. Commercial tools to support requirements engineering become available.

Late 1990s. Java is developed and released in the mid-1990s. The UML is proposed, integrating several separately developed notations for representing object-oriented

systems.

Early 2000s. Use of the UML becomes widespread. Increasing use of scripting languages such as Python and PERL for software development. C# developed as a

competitor to Java.

## Basic Concepts

### Software costs

* Software costs often dominate computer system costs. The costs of software on a PC are often greater than the hardware cost.
* Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times development costs.
* Software engineering is concerned with cost-effective software development.

### Frequently asked questions about software engineering

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số

Mô tả được tạo tự động

### Frequently asked questions about software engineering

Ảnh có chứa văn bản, đồ điện tử, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động

### Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số Mô tả được tạo tự độngEssential attributes of good software

### Software engineering

• Software engineering is an engineering discipline that is concerned

with all aspects of software production from the early stages of system

specification through to maintaining the system after it has gone into

use.

• Engineering discipline

• Using appropriate theories and methods to solve problems bearing in mind

organizational and financial constraints.

• All aspects of software production

• Not just technical process of development. Also project management and the

development of tools, methods etc. to support software production.

### **Importance of software engineering**

• More and more, individuals and society rely on advanced software systems. We need to be ble to produce reliable and trustworthy systems economically and quickly.

• It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.

### Software process activities

• Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.

• Software development, where the software is designed and programmed.

• Software validation, where the software is checked to ensure that it is what the customer requires.

• Software evolution, where the software is modified to reflect changing customer and market requirements.

## General Issues That Affect Most Software

• Heterogeneity

Increasingly, systems are required to operate as distributed systems across

networks that include different types of computer and mobile devices.

• Business and social change

Business and society are changing incredibly quickly as emerging economies

develop and new technologies become available. They need to be able to

change their existing software and to rapidly develop new software.

• Security and trust

As software is intertwined with all aspects of our lives, it is essential that we

can trust that software.

• Scale

Software has to be developed across a very wide range of scales, from very

small embedded systems in portable or wearable devices through to Internetscale, cloud-based systems that serve a global community.

### Software engineering diversity

• There are many different types of software system and there is no

universal set of software techniques that is applicable to all of these.

• The software engineering methods and tools used depend on the type

of application being developed, the requirements of the customer and

the background of the development team.

## Classification

### Software products

• Generic products

Stand-alone systems that are marketed and sold to any customer who wishes

to buy them.

Examples – PC software such as graphics programs, project management tools;

CAD software; software for specific markets such as appointments systems for

dentists.

• Customized products

Software that is commissioned by a specific customer to meet their own

needs.

Examples – embedded control systems, air traffic control software, traffic

monitoring systems.

### Product specification

• Generic products

The specification of what the software should do is owned by the software developer and decisions on software change are made by the developer.

• Customized products

The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.

### Application types

• Stand-alone applications

These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.

• Interactive transaction-based applications

Applications that execute on a remote computer and are accessed by users from their

own PCs or terminals. These include web applications such as e-commerce

applications.

• Embedded control systems

These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.

• Batch processing systems

These are business systems that are designed to process data in large batches. They

process large numbers of individual inputs to create corresponding outputs.

• Entertainment systems

These are systems that are primarily for personal use and which are intended to

entertain the user.

• Systems for modeling and simulation

These are systems that are developed by scientists and engineers to model physical

processes or situations, which include many, separate, interacting objects.

• Data collection systems

These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.

• Systems of systems

These are systems that are composed of a number of other software systems.

### Software engineering fundamentals

• Some fundamental principles apply to all types of software system,

irrespective of the development techniques used:

• Systems should be developed using a managed and understood development

process. Of course, different processes are used for different types of software.

• Dependability and performance are important for all types of system.

• Understanding and managing the software specification and requirements

(what the software should do) are important.

• Where appropriate, you should reuse software that has already been

developed rather than write new software.

### Internet software engineering

• The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.

• Web services allow application functionality to be accessed over the web.

• Cloud computing is an approach to the provision of computer services where applications run remotely on the ‘cloud’.

• Users do not buy software buy pay according to use.

### Web-based software engineering

• Web-based systems are complex distributed systems but the fundamental principles of software engineering discussed previously are as applicable to them as they are to any other types of system.

• The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software in the same way that they apply to other types of software system.

### Web software engineering

• Software reuse

Software reuse is the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from pre-existing software components and systems.

• Incremental and agile development

Web-based systems should be developed and delivered incrementally. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.

• Service-oriented systems

Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services.

• Rich interfaces

Interface development technologies such as AJAX and HTML5 have emerged that support the creation of rich interfaces within a web browser.

## Software Industry In Vietnam

### Key points

• Software engineering is an engineering discipline that is concerned with all aspects of software production.

• Essential software product attributes are maintainability, dependability and security, efficiency and acceptability.

• The high-level activities of specification, development, validation and evolution are part of all software processes.

• The fundamental notions of software engineering are universally applicable to all types of system development.

• There are many different types of system and each requires appropriate software engineering tools and techniques for their development.

• The fundamental ideas of software engineering are applicable to all types of software system.

# REQUIREMENTS FOR SOFTWARE ENGINEERS

## SOFTWARE ENGINEERING ETHICS - RULES & REGULATIONS

• Software engineering involves wider responsibilities than simply the application of technical skills.

• Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals.

• Ethical behaviour is more than simply upholding the law but involves following a set of principles that are morally correct.

### Issues of professional responsibility

• Confidentiality

• Engineers should normally respect the confidentiality of their employers or

clients irrespective of whether or not a formal confidentiality agreement has

been signed.

• Competence

• Engineers should not misrepresent their level of competence. They should not

knowingly accept work which is outwith their competence.

• Intellectual property rights

• 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.

• Computer misuse

• 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

• The professional societies in the US have cooperated to produce a code of ethical practice.

• Members of these organisations sign up to the code of practice when they join.

• The Code contains eight Principles related to the behaviour 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.

### Rationale for the code of ethics

• Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems.

• Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected 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

• Disagreement in principle with the policies of senior management.

• Your employer acts in an unethical way and releases a safety-critical

system without finishing the testing of the system.

• Participation in the development of military weapons systems or

nuclear systems.

## REQUIRED KNOWLEDGE, SKILLS & ATTITUDE

Khi một công nghệ mới bắt đầu lăn bánh,

nếu bạn không là một phần

của guồng quay đó,

bạn sẽ bị đào thải.

- Stewart Brand –

# SOFTWARE PROCESSES

## WHAT IS SOFTWARE PROCESS?

### Code-and-Fix

• Really Bad

• Really Common

• Advantages

• No Overhead

• No Expertise

• Disadvantages

• No means of assessing progress

• Difficult to coordinate multiple programmers

• Useful for “hacking” single-use/personal-use programs: start with empty program and debug until it works

### 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.

## SOFTWARE PROCESS CLASSIFICATION

### 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 plandriven and agile approaches.

• There are no right or wrong software processes.

## 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.

### Software specification

• The process of establishing what services are required and the constraints on the system’s operation and development.

• Requirements engineering process

• Feasibility study

• Is it technically and financially feasible to build the system?

• 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

### The requirements engineering process

Ảnh có chứa văn bản, biểu đồ, Phông chữ, hàng

Mô tả được tạo tự động

### 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.

### Software 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

Ảnh có chứa văn bản, Phông chữ, hàng, ảnh chụp màn hình

Mô tả được tạo tự động

### 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.

### 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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, hàng

Mô tả được tạo tự động

## THE WATERFALL MODEL

Ảnh có chứa văn bản, Phông chữ, ảnh chụp màn hình, biểu đồ

Mô tả được tạo tự động

### 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 benefits

• The waterfall model is consistent with other engineering process models and documentation is produced at each phase.

• This makes the process visible so managers can monitor progress against the development plan.

### 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 wellunderstood 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.

## 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 be altered to incorporate the change.

## THE INCREMENTAL DEVELOPMENT

### 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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, biểu đồ

Mô tả được tạo tự động

### 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.

• 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 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.

• 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.

• Iterative development can also be difficult when a replacement system is being developed.

• Users want all of the functionality of the old system and are often unwilling to experiment with an incomplete new system. Therefore, getting useful customer feedback is difficult.

• 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.

### When not to use?

• There are some types of system where incremental development is not the best approach. These are very large systems where development may involve teams working in different locations, some embedded systems where the software depends on hardware development and some critical systems where all the requirements must be analyzed to check for interactions that may compromise the safety or security of the system.

• These systems, of course, suffer from the same problems of uncertain and changing requirements. Therefore, to address these problems and get some of the benefits of incremental development, a process may be used in which

a system prototype is developed iteratively and used as a platform for experiments with the system requirements and design. With the experience gained from the prototype, definitive requirements can then be agreed.

## PROTOTYPE

### 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.

### The process of prototype development

Ảnh có chứa văn bản, Phông chữ, ảnh chụp màn hình, biểu đồ

Mô tả được tạo tự động

### 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

### Benefits of prototyping

• Improved system usability.

• A closer match to users’ real needs.

• Improved design quality.

• Improved maintainability.

• Reduced development effort.

### Problem of Prototyping

• A general problem with prototyping is that the prototype may not necessarily be used in the same way as the final system. The tester of the prototype may not be typical of system users. The training time during prototype evaluation may be insufficient.

• If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times. When provided with better response in the final system, they may use it in a different way.

### 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.

## SCRUM

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, Phông chữ

Mô tả được tạo tự động

### 3 Roles

* Product Owner
  + • Sets the vision
  + • Represents the voice of the customer
  + • Define the features of the product
  + • Prioritize features
  + • Adjust features and priority every iteration, as needed
  + • Accept or reject work results
* Scrum Master
  + • Represents management to the project
  + • Responsible for enacting Scrum values and practices
  + • Removes impediments
  + • Ensure that the team is fully functional and productive
  + • Enable close cooperation across all roles and functions
  + • Shield the team from external interferences
* Development Team
  + • 7 +/- 2
  + • Cross functional
  + • Full time
  + • Self organizing
  + • Empowered
  + • Trust

### 3 Artifacts

* Product Backlog
  + • Contains all potential features, prioritized as an absolute ordering by business value.
  + • It is therefore the “What” that will be built, sorted by importance.
  + • It contains rough estimates of both business value and development effort.
  + • Those estimates help the Product Owner to gauge the timeline and, to a limited extent prioritize.
  + • Responsible by Product Owner, including content, availability, and ordering.
* Sprint Backlog
  + • The Sprint Backlog is the set of Product Backlog items selected for the Sprint, plus a plan for delivering the product Increment and realizing the Sprint Goal.
  + • It is a forecast by the Development Team about what functionality will be in the next Increment and the work needed to deliver that functionality into a “Done” Increment.
  + • It makes visible all of the work that the Development Team identifies as necessary to meet the Sprint Goal.
* Increment
  + • The Increment is the sum of all the Product Backlog items completed during a Sprint and the value of the increments of all previous Sprints.
  + • At the end of a Sprint, the new Increment must be “Done”, which means it must be in useable condition and meet the Scrum Team’s definition of “Done”.
  + • It must be in useable condition regardless of whether the Product Owner decides to actually release it.

### Burndown Chart

• Displays the remaining effort of the team for a given period of time.

• Is not a standard artifact in scrum.



### 4 Events

• Sprint Planning

• Daily Scrum

• Sprint Review

• Sprint Retrospective

### Sprint Planning (8 hours / sprint)

• The work to be performed in the Sprint is planned at the Sprint Planning. This plan is created by the collaborative work of the entire Scrum Team.

• Sprint Planning is time-boxed to a maximum of eight hours for a one-month Sprint. For shorter Sprints, the event is usually shorter. The Scrum Master ensures that the event takes place and that attendants understand its purpose. The Scrum Master teaches the Scrum Team to keep it within the time-box.

• Sprint Planning answers the following:

• What can be done this Sprint?

• How will the chosen work get done?

### Daily Scrum (15 mins)

• The Daily Scrum is a 15-minute time-boxed event for the Development Team to synchronize activities and create a plan for the next 24 hours. This is done by inspecting the work since the last Daily Scrum and forecasting the work that could be done before the next one. The Daily Scrum is held at the same time and place each day to reduce complexity. During the meeting, the Development Team members explain:

• What did I do yesterday that helped the Development Team meet the Sprint Goal?

• What will I do today to help the Development Team meet the Sprint Goal?

• Do I see any impediment that prevents me or the Development Team from meeting the Sprint Goal?

• The Development Team uses the Daily Scrum to inspect progress toward the Sprint Goal and to inspect how progress is trending toward completing the work in the Sprint Backlog.

• The Scrum Master ensures that the Development Team has the meeting and teaches the Development Team to keep the Daily Scrum within the 15-minute time-box.

• The Scrum Master enforces the rule that only Development Team members participate in the Daily Scrum.

### Sprint Review (4 hours / sprint)

• A Sprint Review is held at the end of the Sprint to inspect the Increment and adapt the Product Backlog if needed. During the Sprint Review, the Scrum Team and stakeholders collaborate about what was done in the Sprint.

• This is an informal meeting, not a status meeting, and the presentation of the Increment is intended to elicit feedback and foster collaboration.

• This is a four-hour time-boxed meeting for one-month Sprints.

• The Scrum Master ensures that the event takes place and that attendants understand its purpose. The Scrum Master teaches all to keep it within the time-box.

### Sprint Retrospective (3 hours / sprint)

• The Sprint Retrospective is an opportunity for the Scrum Team to inspect itself and create a plan for improvements to be enacted during the next Sprint.

• The Sprint Retrospective occurs after the Sprint Review and prior to the next Sprint Planning. This is a three-hour time-boxed meeting for one-month Sprints.

• The Scrum Master encourages the Scrum Team to improve, within the Scrum process framework, its development process and practices to make it more effective and enjoyable for the next Sprint. During each Sprint Retrospective, the Scrum Team plans ways to increase product quality by adapting the definition of “Done” as appropriate.

• By the end of the Sprint Retrospective, the Scrum Team should have identified improvements that it will implement in the next Sprint.

### 3 pillars

• Transparency: Significant aspects of the process must be visible to those responsible for the outcome.

• Inspection: Scrum users must frequently inspect Scrum artifacts and progress toward a Sprint Goal to detect undesirable variances.

• Adaptation: If an inspector determines that one or more aspects of a process deviate outside acceptable limits, and that the resulting product will be unacceptable, the process or the material being processed must be adjusted. An adjustment must be made as soon as possible to minimize further deviation.

### Sprint Process

Ảnh có chứa văn bản, ảnh chụp màn hình, Đồ họa, thiết kế đồ họa

Mô tả được tạo tự động

• Must be useable

• Fully tested

• Documented

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu tượng, Đồ họa

Mô tả được tạo tự động

# REQUIREMENTS ENGINEERING

## WHAT IS REQUIREMENT?

• A statement of a service the system must do OR

• A statement of a constraint the system must satisfy

### 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.

### 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.

## 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

### Readers of different types of requirements specification

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, hàng

Mô tả được tạo tự động

### 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.

• “Functional requirements deal with the What, non functional requirements deal with the How” (Ariel Schlesinger)

## 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.

### 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 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 ross 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.

### Types of nonfunctional requirement

Ảnh có chứa văn bản, biểu đồ, ảnh chụp màn hình, hàng

Mô tả được tạo tự động

### 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.

### 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.

### 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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số

Mô tả được tạo tự động

## REQUIREMENTS ENGINEERING PROCESS

• 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.

### 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

• 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

Ảnh có chứa văn bản, Phông chữ, ảnh chụp màn hình, biểu đồ

Mô tả được tạo tự động

### 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.

### 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

### 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.

### Requirements discovery

Used methods:

• Interviewing

• Ethnography

• Stories and scenarios

## METHODS FOR REQUIREMENTS DISCOVERY

### 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 use to talk about the system by suggesting requirements rather than simply asking them what they want.

### Questions during interview

Identify problems

• What problems do you run into in your day-to-day work? Is there a standard way of solving it, or do you have a workaround?

• Why is this a problem?

• How do you solve the problem today?

• How would you ideally like to solve the problem?

### Questions during interview

The user environment

• Who will be the users of the system?

• What level of education or training do the users have?

• What computer skills do the users have?

• Are users familiar with this type of IT system?

• What technical platforms do they use today?

• Do you know of any plans for future systems or platforms?

• What other IT systems does the organization use today that the new system will need to link to?

• What are your expectations regarding system usability?

• What training needs do you expect for the future system?

• What kind of documentation do you expect?

### Questions during interview

Summary of problem

• So, as I understand it, you are experiencing the following problems/needs (describe the interviewee’s problems and needs in your own words – often you will discover that you do not share the same image. It is very very

common to not understand each other even if at first you think you do).

• Just to confirm, have I correctly understood the problems you have with the current solution?

• Are there any other problems you’re experiencing? If so, what are they?

### Questions during interview

Identify non-functional requirements

• What are your expectations for system availability?

• What are your expectations for system performance?

• Who will manage system support and maintenance?

• What are the organization’s support requirements?

• What are the organization’s system maintenance requirements?

• What are the organization’s security requirements?

• What are the organization’s requirements for installation and configuration?

• How will the system be distributed?

• Are there any legal requirements or other regulatory requirements that need to be met?

• Can you think of any additional requirements that we should know about?

### Ethnography

• It is spending 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 I 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.

• The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.

### 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.

## 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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số

Mô tả được tạo tự động

### 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

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, Song song

Mô tả được tạo tự động

### 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.

### 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.

### 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.

### 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.

• UML sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

### 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. Covered in Chapter 2.

• Test-case generation

• Developing tests for requirements to check testability

# UML

## What is UML ?

• UML - Unified Modeling Language

• Standard & graphical language

❏ Used for software as well as non software

• UML is a language (notation) for modeling Object Oriented system

• Used for making software blue print

• specifying, visualizing, constructing, documenting the artifacts of software system

### Why UML?

• Graphical notation

• A picture is worth a thousand words

• Standard communication language

• Provides multiple diagrams for capturing different Architectural Views

• Promotes component reusability

UML is a standard language for visualizing, specifying, constructing, and documenting software systems

## USE-CASE MODELING

### Objectives

• When you complete this module, you should be able to:

• Define actor, use case, and use-case model

• List the benefits of use cases

• Explain how use cases fit into a requirements management process and the software development lifecycle

### What Is Use-Case Modeling?

• Links stakeholder needs to software requirements.

• Defines clear boundaries of a system.

• Captures and communicates the desired behavior of the system.

• Identifies who or what interacts with the system.

• Validates/verifies requirements.

• Is a planning instrument.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số

Mô tả được tạo tự động

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, biểu đồ

Mô tả được tạo tự động

# ACTIVITY DIAGRAM

• An activity diagram in the use-case model can be used to capture the activities and actions

performed in a use case.

• It is essentially a flow chart, showing flow of control from one activity or action to another.

• It consists of actions, nodes and transitions between activities and states.

### Activity Diagrams

• Model business workflows.

• Identify candidate use cases, through the examination of business

workflows.

• Identify pre- and post-conditions for use cases.

• Model workflows between/within use cases.

### Activities

• An Activity is the process being modeled.

• An Activity is a unit of work that needs to be carried out.

• Any Activity takes time.

### Actions

• An Action is a step in the overall activity.

• The work can be documented as Actions in the activity.

### Transitions

• With arrows indicating direction, the transition lines on an activity diagram show the sequential flow of actions in the modeled activity.

• The arrow will always point to the next action in the activity's sequence.

### Initial Nodes

• The initial node clearly shows the starting point for the action sequence within an activity diagram.

• The initial node is drawn as a solid circle with a transition line (arrow) that connects it to the first action in the activity's sequence of actions.

• There can be more than one initial node. In this case, invoking the activity starts multiple flows, one at each initial node.

• There can be only one transition line connecting the initial node to an action.

### Final Nodes

• Final node indicates that the activity's action sequence or a flow has reached its end.

• There are two types of final nodes: activity final node and flow final node.

• Activity final node is drawn as a circle surrounding a smaller solid circle.

• Flow final node is drawn as a circle with a cross inside.

• An activity final node stops all flows in an activity and terminates the entire activity.

• A flow final node terminates a path through an activity, but not the entire activity.

• Every activity diagram should have at least one final node symbol.

• It is possible for an activity diagram to show multiple final nodes. In other words, the activity may terminate in different manners.

### Decision nodes

• A decision node shows where the exit transition from a action may branch in alternative directions depending on a condition.

• A decision is drawn as a diamond on an activity diagram.

• Since a decision will have at least two different outcomes, the decision symbol will have multiple transition lines connecting to different actions.

### Decision nodes > Guard conditions

• A guard condition explicitly tells when to follow a transition line to the next action.

• Guard condition text is always placed in brackets. For example, [guard condition text].

• The [else] guard is commonly used in activity diagrams to mean "if none of the other guarded transition lines matches the actual condition," then follow the [else] transition line.

### Merge nodes

• A merge node brings together alternate flows into a single output flow.

• A merge node using the same diamond icon with multiple paths pointing to it, but with only one transition line coming out of it.

• A merge does not synchronize multiple concurrent flows.

### Synch states

• Certain action sequences can be done in parallel.

• Parallel action sequences are officially named synch states.

• A synch state is where a transition forks into multiple paths or multiple paths are joined into a single transition.

### Fork Nodes

• A fork is where a path splits.

• A fork node splits incoming flow into multiple concurrent flows.

### Join Nodes

• A join is where multiple concurrent paths meet.

• A join synchronizes multiple inflows and produces a single outflow. The outflow from a join cannot execute until all inflows have been received.

### Swimlanes

Swimlane is used to model the activity's procedural flow of control between the objects (persons, organizations, or other responsible entities) that actually execute the action.

# SOFTWARE DESIGN

• Software design: Design a software structure that realises the specification;

• The activities of design and implementation are closely related and may be inter-leaved.

### A general model of the design process

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, biểu đồ

Mô tả được tạo tự động

### Design activities

• Architectural design, where you identify the overall structure of the system, the principal components (sometimes called sub-systems 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.

# CONFIGURATION MANAGEMENT

• Software systems are constantly changing during development and use.

• Configuration management (CM) is concerned with the policies, rocesses and tools for managing changing software systems.

• You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.

• CM is essential for team projects to control changes made by different developers.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số

Mô tả được tạo tự động

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số

Mô tả được tạo tự động

## CM ACTIVITIES

• Change management

• Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.

• Version management

• Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.

• System building

• The process of assembling program components, data and libraries, then compiling these to create an executable system.

• Release management

• Preparing software for external release and keeping track of the system versions that have been released for customer use.

### Configuration management activities

Ảnh có chứa văn bản, biểu đồ, Phông chữ, hàng

Mô tả được tạo tự động

### Multi-version systems

• For large systems, there is never just one ‘working’ version of a system.

• There are always several versions of the system at different stages of development.

• There may be several teams involved in the development of different system versions.

## CHANGE MANAGEMENT

• Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.

• Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.

• The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.

### The change management process

Ảnh có chứa văn bản, biểu đồ, ảnh chụp màn hình, hàng

Mô tả được tạo tự động

### Factors in change analysis

• The consequences of not making the change

• The benefits of the change

• The number of users affected by the change

• The costs of making the change

• The product release cycle

## VERSION MANAGEMENT

• Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.

• It also involves ensuring that changes made by different developers to these versions do not interfere with each other.

• Therefore version management can be thought of as the process of managing codelines and baselines.

### Codelines and baselines

A codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions.

• Codelines normally apply to components of systems so that there are different versions of each component.

• A baseline is a definition of a specific system.

• The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

### Baselines

• Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.

• Baselines are important because you often have to recreate a specific version of a complete system.

• For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in

their system that have to be repaired.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số

Mô tả được tạo tự động

### Version control systems

• Version control (VC) systems identify, store and control access to the different versions of components. There are two types of modern version control system

• Centralized systems, where there is a single master repository that maintains all versions of the software components that are being developed. Subversion is a widely used example of a centralized VC system.

• Distributed systems, where multiple versions of the component repository exist at the same time. Git is a widely-used example of a distributed VC system.

### Centralized version control

• Developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.

• When their changes are complete, they check-in the components back to the repository.

• If several people are working on a component at the same time, each check it out from the repository. If a component has been checked out, the VC system warns other users wanting to check out that component that it has been checked out by someone else.

### Repository Check-in/Check-out

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, biểu đồ

Mô tả được tạo tự động

### Distributed version control

• A ‘master’ repository is created on a server that maintains the code produced by the development team.

• Instead of checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on their computer.

• Developers work on the files required and maintain the new versions on their private repository on their own computer.

• When changes are done, they ‘commit’ these changes and update their private server repository. They may then ‘push’ these changes to the project repository.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số

Mô tả được tạo tự động

### Benefits of distributed version control

• It provides a backup mechanism for the repository.

• If the repository is corrupted, work can continue and the project repository can be restored from local copies.

• It allows for off-line working so that developers can commit changes if they do not have a network connection.

• Project support is the default way of working.

• Developers can compile and test the entire system on their local machines and test the changes that they have made.

### Open source development

• Distributed version control is essential for open source development.

• Several people may be working simultaneously on the same system without any central coordination.

• As well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.

• It is then up to the open-source system ‘manager’ to decide when to pull these changes into the definitive system.

### Version management systems

• Version and release identification

• Managed versions are assigned identifiers when they are submitted to the system.

• Storage management

• To reduce the storage space required by multiple versions of components that differ only slightly, version management systems usually provide storage management facilities.

• Change history recording

• All of the changes made to the code of a system or component are recorded and listed.

### Version management systems

• Independent development

• The version management system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.

• Project support

• A version management system may support the development of several projects, which share components.

### Storage management using deltas

### Branching and merging

• Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.

• This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.

• At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made.

• If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.

## SYSTEM BUILDING

• System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.

• System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.

• The configuration description used to identify a baseline is also used by the system building tool.

### Build platforms

• The development system, which includes development tools such as compilers, source code editors, etc.

• Developers check out code from the version management system into a private workspace before making changes to the system.

• The build server, which is used to build definitive, executable versions of the system.

• Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.

• The target environment, which is the platform on which the system executes.

### Development, build, and target platforms

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, Phông chữ

Mô tả được tạo tự động

### System building

Ảnh có chứa văn bản, biểu đồ, hàng, Phông chữ

Mô tả được tạo tự động

### Build system functionality

• Build script generation

• Version management system integration

• Minimal re-compilation

• Executable system creation

• Test automation

• Reporting

• Documentation generation

## RELEASE MANAGEMENT

• A system release is a version of a software system that is distributed to customers.

• For mass market software, it is usually possible to identify two types of release: major releases which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported.

• For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.

### Release tracking

• In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.

• When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.

• This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.

• Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

### Release reproduction

• To document a release, you have to record the specific versions of the source code components that were used to create the executable code.

• You must keep copies of the source code files, corresponding executables and all data and configuration files.

• You should also record the versions of the operating system, libraries, compilers and other tools used to build the software.

### Release planning

• As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.

• Release timing

• If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.

• If system releases are too infrequent, market share may be lost as customers move to alternative systems

### Factors influencing system release planning

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, số

Mô tả được tạo tự động

### Release components

• As well as the the executable code of the system, a release may also include:

• configuration files defining how the release should be configured for particular installations;

• data files, such as files of error messages, that are needed for successful system operation;

• an installation program that is used to help install the system on target hardware;

• electronic and paper documentation describing the system;

• packaging and associated publicity that have been designed for that release.

### Key points

• Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.

• The main configuration management processes are change management, version management, system building and release management.

• Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these in a new version of a system.

• Version management involves keeping track of the different versions of software components as changes are made to them.

• System building is the process of assembling system components into an executable program to run on a target computer system.

• Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.

• System releases include executable code, data files, configuration files and documentation. Release management involves making decisions on system release dates, preparing all information for distribution and documenting each system release.