BBM 382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 1

Assoc. Prof. Dr. Ayça KOLUKISA TARHAN
Dr. Tuğba ERDOĞAN

BBM 382 & 384 – Course homepage

https://tinyurl.com/t6qu5vf

	BBM382			Reading & UML Asgn. (see below)
Week #	Date	Content	Ch's	Due dates
1	February 27, 2020	Introduction, Waterfall SD	1, 2, 22, 23	
2	March 5, 2020	Prj. Mgt & Planning & Risk Mgt.		
3	March 12, 2020	Software Requirements and UML	4	RA-1
4	March 19, 2020	Exercise in class-System: BILSIS	4	
5	March 26, 2020	System Modeling	5,6	RA-2
6	April 2, 2020	Architectural Design	6	
7	April 9, 2020	Design & Imp. (UML modeling) / **UML assignment (see below)**	7	RA-3
8	April 16, 2020	MIDTERM		
9	April 23, 2020	Software Testing 1	8	
10	April 30, 2020	Software Testing 2	9	UML Assignment
11	May 7, 2020	Software Evolution and Maintenance	9	
12	May 14, 2020	Change/Configuration Management	24	RA-4
13	May 21, 2020	Quality Management	3	
14	May 28, 2020	Software Process (Others)		

Components	Midterm Exam	25.00%
	Final Exam	40%
	UML assignment	15.00%
	Reading Assignments (4 RAs)	10.00%
	Attendance	10.00%

Textbook: Ian Sommerville, ***Software Engineering***, Addison-Wesley; 10th edition, 2015.

Translation of 10th Ed (in Turkish):
https://www.nobelyayin.com/detay.asp?u=14776

CHAPTER 1 – INTRODUCTION

1.1 PROFESSIONAL SOFTWARE DEVELOPMENT

Software in Modern World

- We can’t run the modern world without software.
  - National infrastructures and utilities are controlled by computer-based systems
  - Most electrical products include a computer and controlling software
  - Industrial manufacturing and distribution is completely computerized, as is the financial system.
  - Entertainment, including the music industry, computer games, and film and television, is software intensive

Software in Modern World: Examples

Challenges

- Characteristics of software:
  - Intangible (abstract)
  - Changeable
- Characteristics of software development:
  - Human-intensive
  - Multi-disciplinary

Software Failures – Main Reasons

- ***Increasing demands***
  - Systems have to be built and delivered more quickly;
  - Larger, even more complex systems are required;
  - Systems have to have new capabilities that were previously thought to be impossible.
  - Existing software engineering methods cannot cope and new software engineering techniques have to be developed to meet these new demands.

- ***Low expectations***
  - It is relatively easy to write computer programs without using software engineering methods and techniques.
  - Many companies do not use software engineering methods.
  - Consequently, their software is often more expensive and less reliable than it should be.

We need better software engineering education and training to address this problem.

Software Failures – Expectations!

Software Engineering is NOT Programming!

Programming Vs. Engineering

Programming	Software Engineering
Personal activity (instrument)	Team activity (orchestra)
One aspect of software development	Large systems must be developed similar to other engineering practices
Concerned about accomplishing the objective of the program itself	Concerned about the entire solution, its feasibility, and future use

History of Software Engineering

- The notion of ‘software engineering’ was first proposed in 1968 at a conference held to discuss what was then called the ‘software crisis’ (Naur and Randell, 1969).
  - It became clear that individual approaches to program development did not scale up to large and complex software systems. These were unreliable, cost more than expected, and were delivered late.
- Throughout the 1970s and 1980s, a variety of new software engineering techniques and methods were developed, such as structured programming, information hiding and object-oriented development. Tools and standard notations were developed and are now extensively used.
- See http://www.SoftwareEngineering-9.com/Web/History/

Professional Software Development

- Professional software, intended for use by someone apart from its developer, is usually developed by teams rather than individuals. It is maintained and changed throughout its life.

- Software engineering is intended to support professional software development, rather than individual programming. It includes techniques that support program specification, design, and evolution.
  - None of which are normally relevant for personal software development.

Professional Software Development

- Many people think that software is simply another word for computer programs. However, software is not just the programs themselves but also all associated documentation and configuration data that is required to make these programs operate correctly.
  - A professionally developed software system usually consists of system documentation, which describes the structure of the system; user documentation, which explains how to use the system; and web-sites for users to download recent product information.

- This is one of the important differences between professional and amateur software development.

Software Products

- Software engineers are concerned with developing software products (i.e., software which can be sold to a customer).

- There are two kinds of software products:
  - *Generic products*
  - *Customized (or bespoke) products*

Kinds of Software Products

- *Generic products* These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them.
  - Examples include software for PCs such as databases, word processors, games, drawing packages, and project-management tools. It also includes some specific purpose applications such as library information systems, accounting systems, or systems for maintaining dental records.

- *Customized (or bespoke) products* These are systems that are commissioned by a particular customer. A software contractor develops the software especially for that customer.
  - Examples of this type of software include control systems for specific-purpose electronic devices, systems written to support a particular business process, and air traffic control systems.

Kinds of Software Products

- An important difference between these types of software is
  - In *generic products*, the organization that develops the software controls the software specification (*e.g. MS Word*)
  - For *custom products*, the specification is usually developed and controlled by the organization that is buying the software (*e.g. Submit system*)

- The mix of two is also possible!
  - Systems are being built with a generic product as a base, which is then adapted to suit the requirements of a customer.
    - Enterprise Resource Planning (ERP) systems, such as the SAP system, are the best examples.

Quality

- When we talk about the quality of professional software, we have to take into account that the software is used and changed by people apart from its developers.
  - Quality is therefore not just concerned with what the software does; it has to include the software's behavior while it is executing and the structure and organization of the system programs and associated documentation.

- This is so-called quality or non-functional software attributes.
  - Examples of these attributes are the *software's response time to a user query* and the *understandability of the program code*.
  - The specific set of attributes that you might expect from a software system obviously depends on its application domain. Therefore, a banking system must be secure, an interactive game must be responsive, a telephone switching system must be reliable, and so on.

Essential Attributes of Good Software

Product characteristics	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
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	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
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.

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 - Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work to organizational and financial constraints so they look for solutions within these constraints.*
  - *All aspects of software production - Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software production.*

Engineering Discipline

- Engineering is about getting repeatable results of the required quality within the schedule and budget.
  - This often involves making compromises—engineers cannot be perfectionists.
  - People writing programs for themselves, however, can spend as much time as they wish on program development.

- In general, software engineers adopt a systematic and organized approach to their work, as this is often the most effective way to produce high-quality software.
  - However, engineering is all about selecting the most appropriate method for a set of circumstances so a more creative, less formal approach to development may be effective in some circumstances.
  - Less formal development is particularly appropriate for the development of web-based systems, which requires a blend of software and graphical design skills.

Software Engineering is Important

Software engineering is important for two reasons:

- More and more, individuals and society rely on advanced software systems. We need to be able 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 systems, the majority of costs are the costs of changing the software after it has gone into use.

Software Engineering vs. Computer Science

- Computer science is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software.
- Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers.
- Computer science theory, however, is often most applicable to relatively small programs. Elegant theories of computer science cannot always be applied to large, complex problems that require a software solution.

Software Engineering vs. System Engineering

- System engineering is concerned with all aspects of the development and evolution of complex systems where software plays a major role.
- System engineering is therefore concerned with hardware development, policy and process design and system deployment, as well as software engineering.
- System engineers are involved in specifying the system, defining its overall architecture, and then integrating the different parts to create the finished system. They are less concerned with the engineering of the system components (hardware, software, etc.)

The BIGGER picture!

General Issues of Software

- *Heterogeneity* Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices. As well as running on general-purpose computers, software may also have to execute on mobile phones. You often have to integrate new software with older legacy systems written in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity.

General Issues of Software

- *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. Many traditional software engineering techniques are time consuming and delivery of new systems often takes longer than planned. They need to evolve so that the time required for software to deliver value to its customers is reduced.

- *Security and trust* As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote soft- ware systems accessed through a web page or web service interface. We have to make sure that malicious users cannot attack our software and that information security is maintained.

Software Engineering Diversity

- Software engineering is a systematic approach to the production of software that takes into account practical cost, schedule, and dependability issues, as well as the needs of software customers and producers.
  - How this systematic approach is actually implemented varies dramatically depending on the organization developing the software, the type of software, and the people involved in the development process.
  - There are no universal software engineering methods and techniques that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years.

Types of Software Applications

- There are many different types of application including:
  - *Stand-alone applications*
  - *Interactive transaction-based applications*
  - *Embedded control systems*
  - *Batch processing systems*
  - *Entertainment systems*
  - *Systems for modeling and simulation*
  - *Data collection systems*
  - *Systems of systems*

Software Engineering Diversity

- You use different software engineering techniques for each type of system because the software has quite different characteristics.
  - For example, an embedded control system in an automobile is safety-critical and is burned into ROM when installed in the vehicle. It is therefore very expensive to change. Such a system needs very extensive verification and validation so that the chances of having to recall cars after sale to fix software problems are minimized. User interaction is minimal (or perhaps nonexistent) so there is no need to use a development process that relies on user interface prototyping.
  - For a web-based system, an approach based on iterative development and delivery may be appropriate, with the system being composed of reusable components. However, such an approach may be impractical for a system of systems, where detailed specifications of the system interactions have to be specified in advance so that each system can be separately developed.

Software Engineering Diversity

- Nevertheless, there are software engineering fundamentals that apply to all types of software systems:
  - They should be developed using a managed and understood development process. The organization developing the software should plan the development process and have clear ideas of what will be produced and when it will be completed. Of course, different processes are used for different types of software.
  - Dependability and performance are important for all types of systems. Software should behave as expected, without failures and should be available for use when it is required. It should be safe in its operation and, as far as possible, should be secure against external attack. The system should perform efficiently and should not waste resources.

CHAPTER 2 – SOFTWARE PROCESSES

Software Process

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product:

- 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.

- There are also supporting process activities such as documentation and software configuration management

Software Process

- A structured set of activities required to develop a software system.
  - Development of software from scratch in a standard programming language
  - Development by extending and modifying existing systems or by configuring and integrating generic (off-the-shelf) software or system components

Ref. http://www.chambers.com.au/glossary/activity.php

Ref. http://www.dkgcreative.com.au/web-design-company-web-developer/web-design-process/

Software Process

- Different types of systems need different development processes.
  - For example, *real-time software* in an aircraft has to be completely specified before development begins.
  - In *e-commerce systems*, the specification and the program are usually developed together.

- Consequently, these generic activities may be organized in different ways and described at different levels of detail depending on the type of software being developed.
  - There are many different types of software. There is no universal software engineering method or technique that is applicable for all of these.

Process Descriptions

- When we describe and discuss processes, we usually talk about the <u>activities</u> in these processes
  - e.g., specifying a data model, designing a user interface, and the ordering of these activities.

- Anology:

Sufle receipe

(process desc.)

&

Sufle cooking

(process)

Keto Cheese Souffle Recipe

Ingredients:

1 tbsp + 4 tbsp butter	salt + pepper
1 tbsp coconut flour	1 drop tobasco
3 tbsp heavy cream	1 sprinkle nutmeg
2 eggs	1 oz. cheddar cheese

Instructions:

1. Preheat oven to 375 degrees.
2. Use 1 tbsp of butter to coat souffle mold and aluminum collar.
3. Heat butter in sauce pan on medium heat.
4. Add coconut flour and blend with spatula to form a roux. Cook until foamy and bubbly, then remove from heat.
5. Once the roux stops foaming, add the heavy cream. Heat with a whisk until the sauce thickens, approximately 3 minutes.
6. Separate egg yolks from egg whites. Place egg whites in a large mixing bowl and set to the side.
7. Add egg yolks to white sauce and heat with whisk. Stir in salt, pepper, tobasco and nutmeg. Set to the side.
8. Heat egg whites with a blender. When it starts to foam, add a pinch of salt. Continue beating egg whites until you get a fluffy, slightly stiff consistency. (They should have a sheen to them).
9. Stir a heaping spoon full of the egg whites to the white sauce.
10. Take the white sauce and pour it into the side of the bowl of egg whites. Fold the white sauce into the egg whites, adding the cheddar cheese as you go.
11. Pour souffle mixture into the mold.
12. Place collar around the mold.
13. Bake for 45 minutes.

ketoresegirl.com

Software Process Descriptions:

Example

(Personal Software Process)

PSP0 Process Script

Purpose	To guide the development of module-level programs
Entry Criteria	- Problem description - PSP0 Project Plan Summary form - Time and Defect Recording logs - Defect Type standard - Stopwatch (optional)

Step	Activities	Description
1	Planning	- Produce or obtain a requirements statement. - Estimate the required development time. - Enter the plan data in the Project Plan Summary form. - Complete the Time Recording log.
2	Development	- Design the program. - Implement the design. - Compile the program, and fix and log all defects found. - Test the program, and fix and log all defects found. - Complete the Time Recording log.
3	Postmortem	Complete the Project Plan Summary form with actual time, defect, and size data.

Exit Criteria	- A thoroughly tested program - Completed Project Plan Summary form with estimated and actual data - Completed Time and Defect Recording logs

Software Process Descriptions

- Process descriptions may also include:
  - Products, which are the outcomes of a process activity;
    - For example, the outcome of the activity of architectural design may be a model of the software architecture.
  - Roles, which reflect the responsibilities of the people involved in the process;
    - Examples of roles are project manager, configuration manager, programmer, etc.
  - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.
    - For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition might be that the UML models describing the architecture have been reviewed.

Process Standardization

- Software processes can be improved by <u>process standardization</u> where the diversity in software processes across an organization is reduced.
  - This leads to improved communication and a reduction in training time, and makes automated process support more economical.
  - Standardization is also an important first step in introducing new software engineering methods and techniques and good software engineering practice.

2.1 SOFTWARE PROCESS MODELS

Software Process Models

- A software process model is an abstract representation of a software process.
  - Each process model represents a process from a particular perspective, and thus provides only partial information about that process.
    - For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities.
    - In this section, we introduce a number of very general process models and present these from an architectural perspective.
      - That is, we see the framework of the process but not the details of specific activities.

Software Process Models

- These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process that can be used to explain different approaches to software development.
  - You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

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
- **Reuse-oriented software engineering**
  - The system is assembled from existing components
  - May be plan-driven or agile

Software Process Models (..continued)

- The waterfall model
  - Plan-driven model. / This approach takes the fundamental process activities of specification, development, validation, and evolution, and represents them as separate process phases in sequence.
- Incremental development
  - May be plan-driven or agile. / This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
- Reuse-oriented software engineering
  - May be plan-driven or agile. / This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

The Waterfall Model

Because of the cascade from one phase to another, this model is known as the 'waterfall model.'

In a plan-driven process—in principle, you must plan and schedule all of the process activities before starting work on them.

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 Phases

- *Requirements analysis and definition.* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
- *System and software design*. The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
- *Implementation and unit testing.* The software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.
- *Integration and system testing.*
- *Operation and maintenance*.

Waterfall Model Phases

- *Requirements analysis and definition.*
- *System and software design.*
- *Implementation and unit testing.*
- *Integration and system testing.* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
- *Operation and maintenance.* The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

Waterfall Model Outputs

The Waterfall Model

- In principle, the result of each phase is one or more documents that are approved ('signed off'). The following phase should not start until the previous phase has finished.
  - In practice, these stages overlap and feed information to each other.
  - During design, problems with requirements are identified. During coding, design problems are found and so on.

- The software process is not a simple linear model but involves feedback from one phase to another. Documents produced in each phase may then have to be modified to reflect the changes made.

The Waterfall Model - Benefits

- The waterfall model is consistent with other engineering process models.
- Documentation is produced at each phase.
  - This makes the process visible so managers can monitor progress against the development plan.
- As is easier to use a common management model in other engineering projects for the whole project, software processes based on the waterfall model are still commonly used.

Waterfall Model - Problems

- It is difficult to accommodate change after the process is underway.
  - A phase has to be complete before moving onto the next phase.
- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
  - Few business systems have stable requirements.
- For the majority of systems this model does not offer significant cost-benefits over other approaches to system development.

Waterfall Model - Usage

- In principle, the waterfall model should only be used <u>when the requirements are well understood and unlikely to change radically</u> during system development.

- 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.

Richard H. Thayer, Software System Engineering: A Tutorial, IEEE Computer, 2002.

Read & Summarize in max 2 A4 pages.

DO NOT COPY AND PASTE!

Submit your summary via Submit System

Due Date: 12.March.2020 23:59 (no extension)

READING & SUMMARY ASSIGNMENT - 1

FOR STUDENT READING…

More on Software Quality Attributes

software quality attributes.

- accessibility
- accountability
- accuracy
- adaptability
- administrability
- affordability
- agility
- auditability
- availability
- credibility
- standards compliance
- process capabilities
- compatibility
- composability
- configurability
- correctness
- customizability
- degradability
- demonstrability
- dependability
- deployability
- distributability
- durability
- evolvability
- extensibility
- fidelity
- flexibility
- installability
- integrity
- interchangeability
- interoperability
- learnability
- maintainability
- manageability
- mobility
- modularity
- nomadicity
- operability
- portability
- precision
- predictability
- recoverability
- relevance
- reliability
- repeatability
- reproducibility
- responsiveness
- reusability
- robustness
- safety
- scalability
- seamlessness
- serviceability (a.k.a. supportability)
- securability
- simplicity
- stability
- survivability
- sustainability
- tailorability
- testability
- timeliness
- understandability
- usability

Types of Software Applications

- There are many different types of application including:
  - *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.
    - Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, etc.

Types of Software Applications

- *Stand-alone applications*
- *Interactive transaction-based applications* These are applications that execute on a remote computer and that are accessed by users from their own PCs or terminals.
  - These include web applications such as e-commerce applications where you can interact with a remote system to buy goods and services.
  - This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing.
  - Interactive applications often incorporate a large data store that is accessed and updated in each transaction.

Types of Software Applications

- *Stand-alone applications*
- *Interactive transaction-based 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.
  - Examples of embedded systems include the software in a mobile (cell) phone, software that controls anti-lock braking in a car, and software in a microwave oven to control the cooking process.

Types of Software Applications

- *Stand-alone applications*
- *Interactive transaction-based applications*
- *Embedded control systems*
- *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.
  - Examples of batch systems include periodic billing systems, such as phone billing systems, and salary payment systems.

Types of Software Applications

- *Stand-alone applications*
- *Interactive transaction-based applications*
- *Embedded control systems*
- *Batch processing systems*
- *Entertainment systems* These are systems that are primarily for personal use and which are intended to entertain the user.
  - Most of these systems are games of one kind or another.
  - The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.

Types of Software Applications

- *Stand-alone applications*
- *Interactive transaction-based applications*
- *Embedded control systems*
- *Batch processing systems*
- *Entertainment systems*
- *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.
  - These are often computationally intensive and require high-performance parallel systems for execution.

Types of Software Applications

- *Stand-alone applications*
- *Interactive transaction-based applications*
- *Embedded control systems*
- *Batch processing systems*
- *Entertainment systems*
- *Systems for modeling and simulation*
- *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.
  - The software has to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location.

Types of Software Applications

- *Stand-alone applications*
- *Interactive transaction-based applications*
- *Embedded control systems*
- *Batch processing systems*
- *Entertainment systems*
- *Systems for modeling and simulation*
- *Data collection systems*
- *Systems of systems* These are systems that are composed of a number of other software systems.
  - Some of these may be generic software products, such as a spreadsheet program. Other systems in the assembly may be specially written for that environment.

Case Studies – Three types of systems *

*(* Refer to case studies in Sec 1.3 for more details)*

- *An embedded system* This is a system where the software controls a hardware (medical) device and is embedded in that device. Issues in embedded systems typically include physical size, responsiveness, power management, etc. / Insulin pump control system
- *An information system* This is a system whose primary purpose is to manage and provide access to a database of information. Issues in information systems include security, usability, privacy, and maintaining data integrity. / Medical records system
- *A sensor-based data collection system* This is a system whose primary purpose is to collect data from a set of sensors and process that data in some way. The key requirements of such systems are reliability, even in hostile environmental conditions, and maintainability. / Wilderness weather station

ID	WBS	Task Name	Start	Finish	Predec
1	**1**	**ATM SİSTEMİ GELİŞTİRME PROJESİ**	**Mon 15.12.08**	**Tue 03.08.10**	
2	**1.1**	**PROJE YÖNETİMİ**	**Mon 15.12.08**	**Mon 02.08.10**	
3	**1.1.1**	**Proje planlama**	**Mon 15.12.08**	**Mon 29.12.08**	
4	1.1.1.1	Kestirimlerin yapılması	Mon 15.12.08	Mon 22.12.08	
5	1.1.1.2	Proje planının geliştirilmesi	Mon 22.12.08	Wed 24.12.08	4
6	1.1.1.3	Proje planının gözden geçirilmesi	Wed 24.12.08	Thu 25.12.08	5
7	1.1.1.4	Proje planının günlenmesi	Thu 25.12.08	Fri 26.12.08	6
8	1.1.1.5	Proje planının yapılandırma kontrolu altına alınması	Fri 26.12.08	Mon 29.12.08	7
9	1.1.1.6	Proje planı	Mon 29.12.08	Mon 29.12.08	8
10	**1.1.2**	**Proje izleme ve kontrol**	**Mon 29.12.08**	**Mon 02.08.10**	**9**
11	1.1.2.1	Başlangıç toplantısı	Mon 29.12.08	Tue 30.12.08	
12	1.1.2.2	Izleme toplantisi-1	Mon 02.02.09	Mon 02.02.09	
13	1.1.2.3	Gelisme raporu-1	Mon 02.02.09	Mon 02.02.09	12
14	1.1.2.4	Izleme toplantisi-2	Mon 02.03.09	Mon 02.03.09	
15	1.1.2.5	Gelisme raporu-2	Mon 02.03.09	Mon 02.03.09	14
16	1.1.2.6	…	Mon 06.04.09	Mon 06.04.09	
17	1.1.2.7	…	Mon 06.04.09	Mon 06.04.09	16
18	1.1.2.8	Izleme toplantisi-18	Mon 05.07.10	Mon 05.07.10	
19	1.1.2.9	Gelisme raporu-18	Mon 05.07.10	Mon 05.07.10	18
20	1.1.2.10	Kapanış toplantısı	Mon 02.08.10	Mon 02.08.10	
21	1.1.2.11	Kapanış raporu	Mon 02.08.10	Mon 02.08.10	20
22	**1.2**	**İŞ GEREKSİNİMLERİNİN BELİRLENMESİ**	**Mon 15.12.08**	**Fri 06.02.09**	
23	1.2.1	İş gereksinimlerinin incelenmesi	Mon 15.12.08	Fri 23.01.09	
24	1.2.2	İş gereksinimleri belgesinin geliştirilmesi	Mon 26.01.09	Fri 30.01.09	23
25	1.2.3	İş gereksinimleri belgesinin gözden geçirilmesi	Mon 02.02.09	Mon 02.02.09	24
26	1.2.4	İş gereksinimleri belgesinin günlenmesi	Tue 03.02.09	Thu 05.02.09	25
27	1.2.5	İş gereksinimleri belgesinin yapılandırma kontrolu altına alınması	Fri 06.02.09	Fri 06.02.09	26
28	1.2.6	İş gereksinimleri belgesi	Fri 06.02.09	Fri 06.02.09	27
29	**1.3**	**SİSTEM GEREKSİNİMLERİNİN VE MİMARİSİNİN BELİRLENMESİ**	**Mon 09.02.09**	**Fri 01.05.09**	**22**
30	**1.3.1**	**Sistem gereksinimleri analizi**	**Mon 09.02.09**	**Fri 03.04.09**	
32	1.3.2	Sistem gereksinimleri belgesi	Fri 03.04.09	Fri 03.04.09	30
33	**1.3.3**	**Sistem mimarisinin belirlenmesi**	**Mon 06.04.09**	**Fri 01.05.09**	**30**
35	1.3.4	Sistem mimari dokumanı	Fri 01.05.09	Fri 01.05.09	33
36	**1.4**	**DONANIM GELİŞTİRME**	**Mon 04.05.09**	**Fri 26.06.09**	**29**
38	**1.5**	**YAZILIM GELİŞTİRME**	**Mon 04.05.09**	**Tue 29.06.10**	**29**
39	**1.5.1**	**Çevrim-1: ATM Kart Yazılım Paketi Geliştirme**	**Mon 04.05.09**	**Fri 09.04.10**	
40	**1.5.1.1**	**Çevrim-1: Yazılım gereksinimlerinin belirlenmesi**	**Mon 04.05.09**	**Fri 12.06.09**	
41	1.5.1.1.1	Yazılım gereksinimleri analizi	Mon 04.05.09	Fri 29.05.09	
42	1.5.1.1.2	Yazılım gereksinimleri belgesinin hazırlanması	Mon 01.06.09	Fri 05.06.09	41
43	1.5.1.1.3	Yazılım gereksinimleri belgesinin gozden gecirilmesi	Mon 08.06.09	Mon 08.06.09	42
44	1.5.1.1.4	Yazılım gereksinimleri belgesinin gunlenmesi	Tue 09.06.09	Thu 11.06.09	43
45	1.5.1.1.5	Yazılım gereksinimleri belgesinin yapılandırma kontrolu altına alınması	Fri 12.06.09	Fri 12.06.09	44
46	1.5.1.1.6	Yazılım gereksinimleri belgesi	Fri 12.06.09	Fri 12.06.09	45
47	**1.5.1.2**	**Çevrim-1: Yazılım tasarımının yapılması**	**Mon 15.06.09**	**Fri 07.08.09**	**40**
48	1.5.1.2.1	Yazılım ust duzey tasarımının yapılması	Mon 15.06.09	Fri 19.06.09	
49	1.5.1.2.2	Yazılım ust duzey tasarımının gozden gecirilmesi	Mon 22.06.09	Mon 22.06.09	48
50	1.5.1.2.3	Yazılım ust duzey tasarımının gunlenmesi	Tue 23.06.09	Tue 23.06.09	49
51	1.5.1.2.4	Yazılım detay tasarımının yapılması	Wed 24.06.09	Tue 21.07.09	50
52	1.5.1.2.5	Yazılım detay tasarımının gozden gecirilmesi	Wed 22.07.09	Tue 28.07.09	51
53	1.5.1.2.6	Yazılım detay tasarımının gunlenmesi	Wed 29.07.09	Thu 30.07.09	52
54	1.5.1.2.7	Yazılım tasarımının belgelendirilmesi	Fri 31.07.09	Tue 04.08.09	53
55	1.5.1.2.8	Yazılım tasarım belgesinin gozden gecirilmesi	Wed 05.08.09	Wed 05.08.09	54
56	1.5.1.2.9	Yazılım tasarım belgesinin gunlenmesi	Thu 06.08.09	Thu 06.08.09	55
57	1.5.1.2.10	Yazılım tasarım belgesinin yapılandırma kontrolu altına alınması	Fri 07.08.09	Fri 07.08.09	56
58	1.5.1.2.11	Yazılım tasarım belgesi	Fri 07.08.09	Fri 07.08.09	57
59	**1.5.1.3**	**Çevrim-1: Yazılım gerçekleştirme ve birim test**	**Mon 10.08.09**	**Fri 19.02.10**	**47**
60	1.5.1.3.1	Yazılım kodlama	Mon 10.08.09	Fri 22.01.10	
61	1.5.1.3.2	Yazılım kodu	Fri 22.01.10	Fri 22.01.10	60
62	1.5.1.3.3	Yazılım birim test	Mon 25.01.10	Fri 19.02.10	61
63	1.5.1.3.4	Yazılım birim test raporu	Fri 19.02.10	Fri 19.02.10	62
64	1.5.1.4	Çevrim-1: Yazılım tümleştirme ve test	Mon 22.02.10	Fri 19.03.10	59
65	1.5.1.5	Yazılım tümleştirme test raporu	Fri 19.03.10	Fri 19.03.10	64
66	1.5.1.6	Yazılım günleme	Mon 22.03.10	Fri 26.03.10	65
67	1.5.1.7	Çevrim-1: Yazılım işlevsel test	Mon 29.03.10	Fri 02.04.10	66
68	1.5.1.8	Yazılım işlevsel test raporu	Fri 02.04.10	Fri 02.04.10	67
69	1.5.1.9	Yazılım günleme	Mon 05.04.10	Fri 09.04.10	68

Project: Project1
Date: Sun 11.01.09

Task | Progress | Summary | External Tasks | Deadline
Split | Milestone | Project Summary | External Milestone

ID	WBS	Task Name	Start	Finish	Predec
70	**1.5.2**	**Çevrim-2: Kredi Kart Yazılım Paketi Geliştirme**	**Mon 04.05.09**	**Fri 04.06.10**	
71	**1.5.2.1**	**Çevrim-2: Yazılım gereksinimlerinin belirlenmesi**	**Mon 04.05.09**	**Fri 26.06.09**	
72	1.5.2.1.1	Yazılım gereksinimleri analizi	Mon 04.05.09	Fri 12.06.09	
73	1.5.2.1.2	Yazılım gereksinimleri belgesinin hazırlanması	Mon 15.06.09	Fri 19.06.09	72
74	1.5.2.1.3	Yazılım gereksinimleri belgesinin gozden gecirilmesi	Mon 22.06.09	Mon 22.06.09	73
75	1.5.2.1.4	Yazılım gereksinimleri belgesinin gunlenmesi	Tue 23.06.09	Thu 25.06.09	74
76	1.5.2.1.5	Yazılım gereksinimleri belgesinin yapılandırma kontrolu altına alınması	Fri 26.06.09	Fri 26.06.09	75
77	1.5.2.1.6	Yazılım gereksinimleri belgesi	Fri 26.06.09	Fri 26.06.09	76
78	**1.5.2.2**	**Çevrim-2: Yazılım tasarımının yapılması**	**Mon 29.06.09**	**Fri 21.08.09**	**71**
79	1.5.2.2.1	Yazılım ust duzey tasarımının yapılması	Mon 29.06.09	Fri 03.07.09	
80	1.5.2.2.2	Yazılım ust duzey tasarımının gozden gecirilmesi	Mon 06.07.09	Mon 06.07.09	79
81	1.5.2.2.3	Yazılım ust duzey tasarımının gunlenmesi	Tue 07.07.09	Tue 07.07.09	80
82	1.5.2.2.4	Yazılım detay tasarımının yapılması	Wed 08.07.09	Tue 04.08.09	81
83	1.5.2.2.5	Yazılım detay tasarımının gozden gecirilmesi	Wed 05.08.09	Tue 11.08.09	82
84	1.5.2.2.6	Yazılım detay tasarımının gunlenmesi	Wed 12.08.09	Thu 13.08.09	83
85	1.5.2.2.7	Yazılım tasarımının belgelendirilmesi	Fri 14.08.09	Tue 18.08.09	84
86	1.5.2.2.8	Yazılım tasarım belgesinin gozden gecirilmesi	Wed 19.08.09	Wed 19.08.09	85
87	1.5.2.2.9	Yazılım tasarım belgesinin gunlenmesi	Thu 20.08.09	Thu 20.08.09	86
88	1.5.2.2.10	Yazılım tasarım belgesinin yapılandırma kontrolu altına alınması	Fri 21.08.09	Fri 21.08.09	87
89	1.5.2.2.11	Yazılım tasarım belgesi	Fri 21.08.09	Fri 21.08.09	88
90	**1.5.2.3**	**Çevrim-2: Yazılım gerçekleştirme ve birim test**	**Mon 24.08.09**	**Fri 16.04.10**	**78**
91	1.5.2.3.1	Yazılım kodlama	Mon 24.08.09	Fri 19.03.10	
92	1.5.2.3.2	Yazılım kodu	Fri 19.03.10	Fri 19.03.10	91
93	1.5.2.3.3	Yazılım birim test	Mon 22.03.10	Fri 16.04.10	92
94	1.5.2.3.4	Yazılım birim test raporu	Fri 16.04.10	Fri 16.04.10	93
95	1.5.2.4	Çevrim-2: Yazılım tümleştirme ve test	Mon 19.04.10	Fri 14.05.10	90
96	1.5.2.5	Yazılım tümleştirme test raporu	Fri 14.05.10	Fri 14.05.10	95
97	1.5.2.6	Yazılım günleme	Mon 17.05.10	Fri 21.05.10	96
98	1.5.2.7	Çevrim-2: Yazılım işlevsel test	Mon 24.05.10	Fri 28.05.10	97
99	1.5.2.8	Yazılım işlevsel test raporu	Fri 28.05.10	Fri 28.05.10	98
100	1.5.2.9	Yazılım günleme	Mon 31.05.10	Fri 04.06.10	99
101	1.5.3	Çevrim-1 ve Çevrim-2 tümleştirme ve test	Mon 07.06.10	Fri 11.06.10	39;70
102	1.5.4	Yazılım günleme	Mon 14.06.10	Wed 16.06.10	101
103	1.5.5	Yazılım işlevsel test	Thu 17.06.10	Wed 23.06.10	102
104	1.5.6	Yazılım günleme	Thu 24.06.10	Mon 28.06.10	103
105	1.5.7	Yazılım kodunun yapılandırma kontrolü altına alınması	Tue 29.06.10	Tue 29.06.10	104
106	1.5.8	Yazılım kodu	Tue 29.06.10	Tue 29.06.10	105
107	1.6	SİSTEM TÜMLEŞTİRME VE TEST	Wed 30.06.10	Tue 13.07.10	36;38
108	**1.7**	**SİSTEM GÜNLEME**	**Wed 14.07.10**	**Tue 27.07.10**	**107**
109	1.7.1	Donanım günleme	Wed 14.07.10	Tue 27.07.10	
110	1.7.2	Yazılım günleme	Wed 14.07.10	Tue 27.07.10	
111	1.8	SİSTEM KABUL TEST	Wed 28.07.10	Tue 03.08.10	108
112	1.9	SİSTEM TESLİM	Tue 03.08.10	Tue 03.08.10	111

Project: Project1
Date: Sun 11.01.09

Task | Progress | Summary | External Tasks | Deadline
Split | Milestone | Project Summary | External Milestone

ID		WBS	Task Name	Duration	Start	Finish	Work	Predecessors	Resource Names
1		**1**	**ATM SİSTEMİ GELİŞTİRME PROJESİ**	**427 days**	**Mon 15.12.08**	**Tue 03.08.10**	**8.676 hrs**		
2		**1.1**	**PROJE YÖNETİMİ**	**426 days**	**Mon 15.12.08**	**Mon 02.08.10**	**260 hrs**		
3		**1.1.1**	**Proje planlama**	**10 days**	**Mon 15.12.08**	**Mon 29.12.08**	**164 hrs**		
4		1.1.1.1	Kestirimlerin yapılması	5 days	Mon 15.12.08	Mon 22.12.08	100 hrs		PY[50%];K1;K2
5		1.1.1.2	Proje planının geliştirilmesi	2 days	Mon 22.12.08	Wed 24.12.08	16 hrs	4	PY
6		1.1.1.3	Proje planının gözden geçirilmesi	1 day	Wed 24.12.08	Thu 25.12.08	32 hrs	5	KG;PY;YY;UY
7		1.1.1.4	Proje planının günlenmesi	1 day	Thu 25.12.08	Fri 26.12.08	8 hrs	6	PY
8		1.1.1.5	Proje planının yapılandırma kontrolu altına alınması	1 day	Fri 26.12.08	Mon 29.12.08	8 hrs	7	YY
9		1.1.1.6	Proje planı	0 days	Mon 29.12.08	Mon 29.12.08	0 hrs	8	YY
10		**1.1.2**	**Proje izleme ve kontrol**	**416 days**	**Mon 29.12.08**	**Mon 02.08.10**	**96 hrs**	**9**	
11		1.1.2.1	Başlangıç toplantısı	1 day	Mon 29.12.08	Tue 30.12.08	16 hrs		PE;UY
12		1.1.2.2	Izleme toplantisi-1	1 day	Mon 02.02.09	Mon 02.02.09	16 hrs		PE;UY
13		1.1.2.3	Gelisme raporu-1	0 days	Mon 02.02.09	Mon 02.02.09	0 hrs	12	PY
14		1.1.2.4	Izleme toplantisi-2	1 day	Mon 02.03.09	Mon 02.03.09	16 hrs		PE;UY
15		1.1.2.5	Gelisme raporu-2	0 days	Mon 02.03.09	Mon 02.03.09	0 hrs	14	PY
16		1.1.2.6	…	1 day	Mon 06.04.09	Mon 06.04.09	16 hrs		PE;UY
17		1.1.2.7	…	0 days	Mon 06.04.09	Mon 06.04.09	0 hrs	16	PY
18		1.1.2.8	Izleme toplantisi-18	1 day	Mon 05.07.10	Mon 05.07.10	16 hrs		PE;UY
19		1.1.2.9	Gelisme raporu-18	0 days	Mon 05.07.10	Mon 05.07.10	0 hrs	18	PY
20		1.1.2.10	Kapanış toplantısı	1 day	Mon 02.08.10	Mon 02.08.10	16 hrs		PE;UY
21		1.1.2.11	Kapanış raporu	0 days	Mon 02.08.10	Mon 02.08.10	0 hrs	20	PY
22		**1.2**	**İŞ GEREKSİNİMLERİNİN BELİRLENMESİ**	**40 days**	**Mon 15.12.08**	**Fri 06.02.09**	**632 hrs**		
23		1.2.1	İş gereksinimlerinin incelenmesi	30 days	Mon 15.12.08	Fri 23.01.09	480 hrs		K1;K2
24		1.2.2	İş gereksinimleri belgesinin geliştirilmesi	5 days	Mon 26.01.09	Fri 30.01.09	80 hrs	23	K1;K2
25		1.2.3	İş gereksinimleri belgesinin gözden geçirilmesi	1 day	Mon 02.02.09	Mon 02.02.09	16 hrs	24	PE;Müşteri
26		1.2.4	İş gereksinimleri belgesinin günlenmesi	3 days	Tue 03.02.09	Thu 05.02.09	48 hrs	25	K1;K2
27		1.2.5	İş gereksinimleri belgesinin yapılandırma kontrolu altına alınması	1 day	Fri 06.02.09	Fri 06.02.09	8 hrs	26	YY
28		1.2.6	İş gereksinimleri belgesi	0 days	Fri 06.02.09	Fri 06.02.09	0 hrs	27	YY
29		**1.3**	**SİSTEM GEREKSİNİMLERİNİN VE MİMARİSİNİN BELİRLENMESİ**	**60 days**	**Mon 09.02.09**	**Fri 01.05.09**	**960 hrs**	**22**	
30		**1.3.1**	**Sistem gereksinimleri analizi**	**40 days**	**Mon 09.02.09**	**Fri 03.04.09**	**640 hrs**		
32		1.3.2	Sistem gereksinimleri belgesi	0 days	Fri 03.04.09	Fri 03.04.09	0 hrs	30	YY
33		**1.3.3**	**Sistem mimarisinin belirlenmesi**	**20 days**	**Mon 06.04.09**	**Fri 01.05.09**	**320 hrs**	**30**	
35		1.3.4	Sistem mimari dokumani	0 days	Fri 01.05.09	Fri 01.05.09	0 hrs	33	YY
36		**1.4**	**DONANIM GELİŞTİRME**	**40 days**	**Mon 04.05.09**	**Fri 26.06.09**	**800 hrs**	**29**	
38		**1.5**	**YAZILIM GELİŞTİRME**	**302 days**	**Mon 04.05.09**	**Tue 29.06.10**	**5.304 hrs**	**29**	
39		**1.5.1**	**Çevrim-1: ATM Kart Yazılım Paketi Geliştirme**	**245 days**	**Mon 04.05.09**	**Fri 09.04.10**	**2.344 hrs**		
40		**1.5.1.1**	**Çevrim-1: Yazılım gereksinimlerinin belirlenmesi**	**30 days**	**Mon 04.05.09**	**Fri 12.06.09**	**320 hrs**		
41		1.5.1.1.1	Yazılım gereksinimleri analizi	20 days	Mon 04.05.09	Fri 29.05.09	160 hrs		PYE
42		1.5.1.1.2	Yazılım gereksinimleri belgesinin hazırlanması	5 days	Mon 01.06.09	Fri 05.06.09	80 hrs	41	K1;K2
43		1.5.1.1.3	Yazılım gereksinimleri belgesinin gozden gecirilmesi	1 day	Mon 08.06.09	Mon 08.06.09	24 hrs	42	KG;PYE;Müşteri
44		1.5.1.1.4	Yazılım gereksinimleri belgesinin gunlenmesi	3 days	Tue 09.06.09	Thu 11.06.09	48 hrs	43	K1;K2
45		1.5.1.1.5	Yazılım gereksinimleri belgesinin yapılandırma kontrolu altına alınması	1 day	Fri 12.06.09	Fri 12.06.09	8 hrs	44	YY
46		1.5.1.1.6	Yazılım gereksinimleri belgesi	0 days	Fri 12.06.09	Fri 12.06.09	0 hrs	45	YY

ID		WBS	Task Name	Duration	Start	Finish	Work	Predecessors	Resource Names
47		**1.5.1.2**	**Çevrim-1: Yazılım tasarımının yapılması**	**40 days**	**Mon 15.06.09**	**Fri 07.08.09**	**424 hrs**	**40**	
48		1.5.1.2.1	Yazılım ust duzey tasarımının yapılması	5 days	Mon 15.06.09	Fri 19.06.09	80 hrs		K1;K2
49		1.5.1.2.2	Yazılım ust duzey tasarımının gozden gecirilmesi	1 day	Mon 22.06.09	Mon 22.06.09	32 hrs	48	PY;K1;K2;Müşteri
50		1.5.1.2.3	Yazılım ust duzey tasarımının gunlenmesi	1 day	Tue 23.06.09	Tue 23.06.09	16 hrs	49	K1;K2
51		1.5.1.2.4	Yazılım detay tasarımının yapılması	20 days	Wed 24.06.09	Tue 21.07.09	160 hrs	50	PYE
52		1.5.1.2.5	Yazılım detay tasarımının gozden gecirilmesi	5 days	Wed 22.07.09	Tue 28.07.09	40 hrs	51	PYE
53		1.5.1.2.6	Yazılım detay tasarımının gunlenmesi	2 days	Wed 29.07.09	Thu 30.07.09	16 hrs	52	PYE
54		1.5.1.2.7	Yazılım tasarımının belgelendirilmesi	3 days	Fri 31.07.09	Tue 04.08.09	48 hrs	53	K1;K2
55		1.5.1.2.8	Yazılım tasarım belgesinin gozden gecirilmesi	1 day	Wed 05.08.09	Wed 05.08.09	16 hrs	54	KG;PYE
56		1.5.1.2.9	Yazılım tasarım belgesinin gunlenmesi	1 day	Thu 06.08.09	Thu 06.08.09	8 hrs	55	PYE
57		1.5.1.2.10	Yazılım tasarım belgesinin yapılandırma kontrolu altına alınması	1 day	Fri 07.08.09	Fri 07.08.09	8 hrs	56	YY
58		1.5.1.2.11	Yazılım tasarım belgesi	0 days	Fri 07.08.09	Fri 07.08.09	0 hrs	57	YY
59		**1.5.1.3**	**Çevrim-1: Yazılım gerçekleştirme ve birim test**	**140 days**	**Mon 10.08.09**	**Fri 19.02.10**	**1.120 hrs**	**47**	
60		1.5.1.3.1	Yazılım kodlama	120 days	Mon 10.08.09	Fri 22.01.10	960 hrs		PYE
61		1.5.1.3.2	Yazılım kodu	0 days	Fri 22.01.10	Fri 22.01.10	0 hrs	60	YY
62		1.5.1.3.3	Yazılım birim test	20 days	Mon 25.01.10	Fri 19.02.10	160 hrs	61	PYE
63		1.5.1.3.4	Yazılım birim test raporu	0 days	Fri 19.02.10	Fri 19.02.10	0 hrs	62	YY
64		1.5.1.4	Çevrim-1: Yazılım tümleştirme ve test	20 days	Mon 22.02.10	Fri 19.03.10	320 hrs	59	K1;K2
65		1.5.1.5	Yazılım tümleştirme test raporu	0 days	Fri 19.03.10	Fri 19.03.10	0 hrs	64	K1;K2
66		1.5.1.6	Yazılım günleme	5 days	Mon 22.03.10	Fri 26.03.10	40 hrs	65	PYE
67		1.5.1.7	Çevrim-1: Yazılım işlevsel test	5 days	Mon 29.03.10	Fri 02.04.10	80 hrs	66	K1;K2
68		1.5.1.8	Yazılım işlevsel test raporu	0 days	Fri 02.04.10	Fri 02.04.10	0 hrs	67	K1;K2
69		1.5.1.9	Yazılım günleme	5 days	Mon 05.04.10	Fri 09.04.10	40 hrs	68	PYE
70		**1.5.2**	**Çevrim-2: Kredi Kart Yazılım Paketi Geliştirme**	**285 days**	**Mon 04.05.09**	**Fri 04.06.10**	**2.664 hrs**		
71		**1.5.2.1**	**Çevrim-2: Yazılım gereksinimlerinin belirlenmesi**	**40 days**	**Mon 04.05.09**	**Fri 26.06.09**	**400 hrs**		
72		1.5.2.1.1	Yazılım gereksinimleri analizi	30 days	Mon 04.05.09	Fri 12.06.09	240 hrs		PYE
73		1.5.2.1.2	Yazılım gereksinimleri belgesinin hazırlanması	5 days	Mon 15.06.09	Fri 19.06.09	80 hrs	72	K1;K2
74		1.5.2.1.3	Yazılım gereksinimleri belgesinin gozden gecirilmesi	1 day	Mon 22.06.09	Mon 22.06.09	24 hrs	73	KG;PYE;Müşteri
75		1.5.2.1.4	Yazılım gereksinimleri belgesinin gunlenmesi	3 days	Tue 23.06.09	Thu 25.06.09	48 hrs	74	K1;K2
76		1.5.2.1.5	Yazılım gereksinimleri belgesinin yapılandırma kontrolu altına alınması	1 day	Fri 26.06.09	Fri 26.06.09	8 hrs	75	YY
77		1.5.2.1.6	Yazılım gereksinimleri belgesi	0 days	Fri 26.06.09	Fri 26.06.09	0 hrs	76	YY
78		**1.5.2.2**	**Çevrim-2: Yazılım tasarımının yapılması**	**40 days**	**Mon 29.06.09**	**Fri 21.08.09**	**424 hrs**	**71**	
79		1.5.2.2.1	Yazılım ust duzey tasarımının yapılması	5 days	Mon 29.06.09	Fri 03.07.09	80 hrs		K1;K2
80		1.5.2.2.2	Yazılım ust duzey tasarımının gozden gecirilmesi	1 day	Mon 06.07.09	Mon 06.07.09	32 hrs	79	PY;K1;K2;Müşteri
81		1.5.2.2.3	Yazılım ust duzey tasarımının gunlenmesi	1 day	Tue 07.07.09	Tue 07.07.09	16 hrs	80	K1;K2
82		1.5.2.2.4	Yazılım detay tasarımının yapılması	20 days	Wed 08.07.09	Tue 04.08.09	160 hrs	81	PYE
83		1.5.2.2.5	Yazılım detay tasarımının gozden gecirilmesi	5 days	Wed 05.08.09	Tue 11.08.09	40 hrs	82	PYE
84		1.5.2.2.6	Yazılım detay tasarımının gunlenmesi	2 days	Wed 12.08.09	Thu 13.08.09	16 hrs	83	PYE
85		1.5.2.2.7	Yazılım tasarımının belgelendirilmesi	3 days	Fri 14.08.09	Tue 18.08.09	48 hrs	84	K1;K2
86		1.5.2.2.8	Yazılım tasarım belgesinin gozden gecirilmesi	1 day	Wed 19.08.09	Wed 19.08.09	16 hrs	85	KG;PYE
87		1.5.2.2.9	Yazılım tasarım belgesinin gunlenmesi	1 day	Thu 20.08.09	Thu 20.08.09	8 hrs	86	PYE
88		1.5.2.2.10	Yazılım tasarım belgesinin yapılandırma kontrolu altına alınması	1 day	Fri 21.08.09	Fri 21.08.09	8 hrs	87	YY
89		1.5.2.2.11	Yazılım tasarım belgesi	0 days	Fri 21.08.09	Fri 21.08.09	0 hrs	88	YY

ID		WBS	Task Name	Duration	Start	Finish	Work	Predecessors	Resource Names
90		**1.5.2.3**	**Çevrim-2: Yazılım gerçekleştirme ve birim test**	**170 days**	**Mon 24.08.09**	**Fri 16.04.10**	**1.360 hrs**	**78**	
91		1.5.2.3.1	Yazılım kodlama	150 days	Mon 24.08.09	Fri 19.03.10	1.200 hrs		PYE
92		1.5.2.3.2	Yazılım kodu	0 days	Fri 19.03.10	Fri 19.03.10	0 hrs	91	YY
93		1.5.2.3.3	Yazılım birim test	20 days	Mon 22.03.10	Fri 16.04.10	160 hrs	92	PYE
94		1.5.2.3.4	Yazılım birim test raporu	0 days	Fri 16.04.10	Fri 16.04.10	0 hrs	93	YY
95		1.5.2.4	Çevrim-2: Yazılım tümleştirme ve test	20 days	Mon 19.04.10	Fri 14.05.10	320 hrs	90	K1;K2
96		1.5.2.5	Yazılım tümleştirme test raporu	0 days	Fri 14.05.10	Fri 14.05.10	0 hrs	95	K1;K2
97		1.5.2.6	Yazılım günleme	5 days	Mon 17.05.10	Fri 21.05.10	40 hrs	96	PYE
98		1.5.2.7	Çevrim-2: Yazılım işlevsel test	5 days	Mon 24.05.10	Fri 28.05.10	80 hrs	97	K1;K2
99		1.5.2.8	Yazılım işlevsel test raporu	0 days	Fri 28.05.10	Fri 28.05.10	0 hrs	98	K1;K2
100		1.5.2.9	Yazılım günleme	5 days	Mon 31.05.10	Fri 04.06.10	40 hrs	99	PYE
101		1.5.3	Çevrim-1 ve Çevrim-2 tümleştirme ve test	5 days	Mon 07.06.10	Fri 11.06.10	120 hrs	39;70	K1;K2;KG
102		1.5.4	Yazılım günleme	3 days	Mon 14.06.10	Wed 16.06.10	24 hrs	101	PYE
103		1.5.5	Yazılım işlevsel test	5 days	Thu 17.06.10	Wed 23.06.10	120 hrs	102	K1;K2;KG
104		1.5.6	Yazılım günleme	3 days	Thu 24.06.10	Mon 28.06.10	24 hrs	103	PYE
105		1.5.7	Yazılım kodunun yapılandırma kontrolü altına alınması	1 day	Tue 29.06.10	Tue 29.06.10	8 hrs	104	YY
106		1.5.8	Yazılım kodu	0 days	Tue 29.06.10	Tue 29.06.10	0 hrs	105	YY
107		1.6	SİSTEM TÜMLEŞTİRME VE TEST	10 days	Wed 30.06.10	Tue 13.07.10	400 hrs	36;38	KG;K1;K2;K3;K4
108		**1.7**	**SİSTEM GÜNLEME**	**10 days**	**Wed 14.07.10**	**Tue 27.07.10**	**160 hrs**	**107**	
109		1.7.1	Donanım günleme	10 days	Wed 14.07.10	Tue 27.07.10	80 hrs		PDE
110		1.7.2	Yazılım günleme	10 days	Wed 14.07.10	Tue 27.07.10	80 hrs		PYE
111		1.8	SİSTEM KABUL TEST	5 days	Wed 28.07.10	Tue 03.08.10	160 hrs	108	KG;K1;K3;Müşteri
112		1.9	SİSTEM TESLİM	0 days	Tue 03.08.10	Tue 03.08.10	0 hrs	111	

BBM-382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 2

Assoc.Prof.Dr. Ayça KOLUKISA TARHAN

Dr. Tuğba ERDOĞAN

CHAPTER 22 – PROJECT MANAGEMENT

What is a Project?

- *"A project is a <u>temporary</u> endeavour undertaken to create a <u>unique</u> product or service"*

  [Project Management Body of Knowledge-PMBOK, 2013]

  - "Temporary"
    - Has a start and an end date
  - "Unique"
    - Has specific attributes for its customers

Example: Civil Engineering

Example: Software Projects

- Developing MS Vista (a new operating system)
  - *Generic (commercial-off-the-shelf - COTS) software*
- Developing Airline Ticket Reservation System
  - *Custom software*

Product of the Sw.Eng.Lab. Project

- System to be developed: Online Bookstore System

What is Project Management?

- *"Project management is the application of knowledge, skills, tools, and techniques to project activities in order to meet project requirements"* [PMBOK, 2013]

- Requires alignment among:
  - The parameters of time, cost, quality, etc.
  - The needs of different stakeholders

Project Management Process

[PMBOK, 2013]

- *"Project Management is accomplished through the appropriate application of five process groups which are:*
  - *Initiating, Planning, Executing, Monitoring and Controlling, and Closing."*

- *"Managing a project typically includes:*
  - *Identifying requirements,*
  - *Addressing the various needs, concerns, and expectations of the stakeholders as the project is planned and carried out,*
  - *Balancing the competing project constraints including (but not limited to):*
    - *Scope, Quality, Schedule, Budget, Resources, and Risk."*

Project Management Process Groups
[PMBOK, 2013]

Figure 3-1. Project Management Process Groups

Project Management Processes with respect to time
[PMBOK, 2013]

Figure 3-2. Process Groups Interact in a Phase or Project

Software Project Management

- Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software.

- Project management is needed because software development is always subject to budget and schedule constraints that are set by the organisation developing the software.

Success criteria

- Deliver the software to the customer at the agreed time.
- Keep overall costs within budget.
- Deliver software that meets the customer’s expectations.
- Maintain a happy and well-functioning development team.

Software management distinctions

- The product is intangible.
  - Software cannot be seen or touched. Software project managers cannot see progress by simply looking at the artifact that is being constructed.
- Many software projects are 'one-off' projects.
  - Large software projects are usually different in some ways from previous projects. Even managers who have lots of previous experience may find it difficult to anticipate problems.
- Software processes are variable & organization specific.
  - We still cannot reliably predict when a particular software process is likely to lead to development problems.

We need management activities!

- Project planning
  - Project managers are responsible for planning. estimating and scheduling project development and assigning people to tasks.
- Reporting
  - Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software.
- Risk management
  - Project managers assess the risks that may affect a project, monitor these risks and take action when problems arise.
- People management
  - Project managers have to choose people for their team and establish ways of working that leads to effective team performance
- Proposal writing
  - The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out.

People management - Teamwork

- Most software engineering is a group activity.
  - The development schedule for most non-trivial software projects is such that they cannot be completed by one person working alone.
- A good group is cohesive and has a team spirit.
  - The people involved are motivated by the success of the group as well as by their own personal goals.
- Group interaction is a key determinant of group performance.
- Flexibility in group composition is limited.
  - Managers must do the best they can

with available people.

People management - Group cohesiveness

- In a cohesive group, members consider the group to be more important than any individual in it.

- The advantages of a cohesive group are:
  - Group quality standards can be developed by the group members.
  - Team members learn from each other and get to know each other's work; Inhibitions caused by ignorance are reduced.
  - Knowledge is shared. Continuity can be maintained if a group member leaves.
  - Refactoring and continual improvement is encouraged. Group members work collectively to deliver high quality results and fix problems, irrespective of the individuals who originally created the design or program.

Risk management

- Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project.

- A risk is a probability that some adverse circumstance will occur
  - Project risks affect schedule or resources;
  - Product risks affect the quality or performance of the software being developed;
  - Business risks affect the organisation developing or procuring the software.

Examples of common project, product, and business risks

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

The risk management process

- Risk identification - Identify project, product and business risks;
- Risk analysis - Assess the likelihood and consequences of these risks;
- Risk planning - Draw up plans to avoid or minimise effects of the risk;
- Risk monitoring - Monitor the risks throughout the project.

Risk identification

- May be a team activities or based on the individual project manager's experience.
- A checklist of common risks may be used to identify risks in a project
  - Technology risks
  - People risks
  - Organisational risks
  - Requirements risks
  - Estimation risks
  - Tool risks

Examples of different risk types

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. (1) Reusable software components contain defects that mean they cannot be reused as planned. (2)
People	It is impossible to recruit staff with the skills required. (3) Key staff are ill and unavailable at critical times. (4) Required training for staff is not available. (5)
Organizational	The organization is restructured so that different management are responsible for the project. (6) Organizational financial problems force reductions in the project budget. (7)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
Estimation	The time required to develop the software is underestimated. (12) The rate of defect repair is underestimated. (13) The size of the software is underestimated. (14)

Risk analysis

- Assess probability and consequences of each risk.
  - Probability may be:
    - insignificant, low, moderate, high or very high.
  - Risk consequences might be:
    - catastrophic, serious, tolerable or insignificant.

Risk types and examples

HIGH IMPACT !!!

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (7).	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project (3).	High	Catastrophic
Key staff are ill at critical times in the project (4).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused. (2).	Moderate	Serious
Changes to requirements that require major design rework are proposed (10).	Moderate	Serious
The organization is restructured so that different management are responsible for the project (6).	High	Serious
The database used in the system cannot process as many transactions per second as expected (1).	Moderate	Serious
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	Tolerable
The rate of defect repair is underestimated (13).	Moderate	Tolerable
The size of the software is underestimated (14).	High	Tolerable
Code generated by code generation tools is inefficient (8).	Moderate	Insignificant

Risk Analysis: An Example

Risk Assessment Matrix				
Impact of Risk (Consequence)	Major	Medium	High	Extreme
	Moderate	Medium	Medium	High
	Minor	Low	Medium	Medium
Seriousness of Risk = Probability x Impact		Unlikely (0-33%)	Moderately Likely (33%-66%)	Highly Likely (66%-100%)
		Probability of Risk (Likelihood)		

Risk planning

- Consider each risk and develop a strategy to manage that risk.
  - Avoidance strategies
    - The probability that the risk will arise is reduced.
  - Minimisation strategies
    - The impact of the risk on the project or product will be reduced.
  - Contingency plans
    - If the risk arises, contingency plans are plans to deal with that risk.

Strategies to help manage risk

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.

Risk monitoring

- Assess each identified risks regularly to decide whether or not it is becoming less or more probable.
- Also assess whether the effects of the risk have changed.
- Each key risk should be discussed at management progress meetings.

Risk indicators

Risk type	Potential indicators
Technology	Late delivery of hardware or support software; many reported technology problems.
People	Poor staff morale; poor relationships amongst team members; high staff turnover.
Organizational	Organizational gossip; lack of action by senior management.
Tools	Reluctance by team members to use tools; complaints about CASE tools; demands for higher-powered workstations.
Requirements	Many requirements change requests; customer complaints.
Estimation	Failure to meet agreed schedule; failure to clear reported defects.

Key Points

- Good project management is essential if software engineering projects are to be developed on schedule and within budget.
- Software management is distinct from other engineering management. Software is intangible. Projects may be novel or innovative with no body of experience to guide their management. Software processes are not as mature as traditional engineering processes.
- Risk management is now recognized as one of the most important project management tasks.
- Risk management involves identifying and assessing project risks to establish the probability that they will occur and the consequences for the project if that risk does arise. You should make plans to avoid, manage or deal with likely risks if or when they arise.

CHAPTER 22 – PROJECT PLANNING

Project planning

- Project planning involves:
  - breaking down the work into parts,
  - assigning these parts to project team members,
  - anticipating problems that might arise, and
  - preparing tentative solutions to those problems.

- The *project plan*, which is created at the start of a project, is used:
  - to show <u>how the work will be done</u> to the project team and customers, and
  - <u>to help assess progress</u> on the project.

Planning stages

- At the proposal stage,

  → when you are bidding for a contract to develop or provide a software system.

- During the project startup phase,

  → when you have to plan who will work on the project,
  how the project will be broken down into increments,
  how resources will be allocated across company, etc.

- Periodically throughout the project,

  → when you modify your plan in the light of experience gained and information from monitoring the progress of the work.

Software pricing

- Estimates are made to discover the cost of producing a software system.
  - You take into account: hardware, software, travel, training and effort costs.
- There is <u>not a simple relationship</u> between the development cost and the price charged to the customer.
- Broader organisational, economic, political and business considerations influence the price charged.

Plan-driven development

- *Plan-driven or plan-based development* is an approach to software engineering where the development process is planned in detail.
  - Plan-driven development is based on engineering project management techniques and is the 'traditional' way of managing large software development projects.
  - A project plan is created that records the work to be done, who will do it, the development schedule and the work products.
  - Managers use the plan to support project decision making and as a way of measuring progress.

Plan-driven development – pros and cons

- The arguments in favor of a plan-driven approach are that:
  - Early planning allows organizational issues (availability of staff, other projects, etc.) to be closely taken into account, and
  - Potential problems and dependencies are discovered before the project starts, rather than once the project is underway.

- The principal argument against plan-driven development is that:
  - Many early decisions have to be revised because of changes to the environment in which the software is to be developed and used.

Project plans

- In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.

- Plan sections:
  - Introduction
  - Project organization
  - Risk analysis
  - Hardware and software resource requirements
  - Work breakdown
  - Project schedule
  - Monitoring and reporting mechanisms

The planning process

- Project planning is an iterative process that starts when you create an initial project plan during the project startup phase.

- Plan changes are inevitable.
  - As more information about the system and the project team becomes available during the project, you should regularly revise the plan to reflect requirements, schedule and risk changes.
  - Changing business goals also leads to changes in project plans. As business goals change, this could affect all projects, which may then have to be re-planned.

The project planning process

Milestones and deliverables

- *Milestones* are points in the schedule against which you can assess progress, for example, the handover of the system for testing.
- *Deliverables* are work products that are delivered to the customer, e.g. a requirements document for the system.

YAZILIM GELİŞTİRME	Mon 04.05.09	Tue 29.06.10	29
Çevrim-1: ATM Kart Yazılım Paketi Geliştirme	Mon 04.05.09	Fri 09.04.10	
Çevrim-1: Yazılım gereksinimlerinin belirlenmesi	Mon 04.05.09	Fri 12.06.09	
Yazılım gereksinimleri analizi	Mon 04.05.09	Fri 29.05.09	
Yazılım gereksinimleri belgesinin hazırlanması	Mon 01.06.09	Fri 05.06.09	41
Yazılım gereksinimleri belgesinin gözden geçirilmesi	Mon 08.06.09	Mon 08.06.09	42
Yazılım gereksinimleri belgesinin güncenmesi	Tue 09.06.09	Thu 11.06.09	43
Yazılım gereksinimleri belgesinin yapılandırma kontrolü altına alınması	Fri 12.06.09	Fri 12.06.09	44
Yazılım gereksinimleri belgesi	Fri 12.06.09	Fri 12.06.09	45
Çevrim-1: Yazılım tasarımının yapılması	Mon 15.06.09	Fri 07.08.09	40
Yazılım üst düzey tasarımının yapılması	Mon 15.06.09	Fri 19.06.09	
Yazılım üst düzey tasarımının gözden geçirilmesi	Mon 22.06.09	Mon 22.06.09	48
Yazılım üst düzey tasarımının güncenmesi	Tue 23.06.09	Tue 23.06.09	49
Yazılım detay tasarımının yapılması	Wed 24.06.09	Tue 21.07.09	50
Yazılım detay tasarımının gözden geçirilmesi	Wed 22.07.09	Tue 28.07.09	51
Yazılım detay tasarımının güncenmesi	Wed 29.07.09	Thu 30.07.09	52
Yazılım tasarımının belgelendirilmesi	Fri 31.07.09	Tue 04.08.09	53
Yazılım tasarım belgesinin gözden geçirilmesi	Wed 05.08.09	Wed 05.08.09	54
Yazılım tasarım belgesinin güncenmesi	Thu 06.08.09	Thu 06.08.09	55
Yazılım tasarım belgesinin yapılandırma kontrolü altına alınması	Fri 07.08.09	Fri 07.08.09	56
Yazılım tasarım belgesi	Fri 07.08.09	Fri 07.08.09	57

Project scheduling

- *Project scheduling* is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed.
- You estimate the calendar time needed to complete each task, the effort required and who will work on the tasks that have been identified.
- You also have to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be.

Project scheduling activities

- Split project into tasks and estimate time and resources required to complete each task.
- Organize tasks concurrently to make optimal use of workforce.
- Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- Dependent on project managers intuition and experience.

Scheduling problems

- Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- Productivity is not proportional to the number of people working on a task.
- Adding people to a late project makes it later because of communication overheads.
- The unexpected always happens. Always allow contingency in planning.

Schedule representation

- Graphical notations are normally used to illustrate the project schedule.
- These show the project breakdown into tasks. Tasks should not be too small. They should take about *a week or two*.
- Bar charts are the most commonly used representation for project schedules. They show the schedule as activities or resources against time.

Tasks, durations, and dependencies

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Activity bar chart

Staff allocation chart

Example:
ATM System Software Development Project Plan

- Full plan is available in the share area of course lecture notes.

Example: ATM System Software Development
(Tasks and dependencies, Schedule, Effort, Resources)

ID		WBS	Task Name	Duration	Start	Finish	Work	Predecessors	Resource Names
1		1	**ATM SİSTEMİ GELİŞTİRME PROJESİ**	**427 days**	**Mon 15.12.08**	**Tue 03.08.10**	**8.676 hrs**		
2		1.1	**PROJE YÖNETİMİ**	**426 days**	**Mon 15.12.08**	**Mon 02.08.10**	**260 hrs**		
3		1.1.1	**Proje planlama**	**10 days**	**Mon 15.12.08**	**Mon 29.12.08**	**164 hrs**		
4		1.1.1.1	Kestirimlerin yapılması	5 days	Mon 15.12.08	Mon 22.12.08	100 hrs		PY[50%];K1;K2
5		1.1.1.2	Proje planının geliştirilmesi	2 days	Mon 22.12.08	Wed 24.12.08	16 hrs	4	PY
6		1.1.1.3	Proje planının gözden geçirilmesi	1 day	Wed 24.12.08	Thu 25.12.08	32 hrs	5	KG;PY;YY;UY
7		1.1.1.4	Proje planının günlenmesi	1 day	Thu 25.12.08	Fri 26.12.08	8 hrs	6	PY
8		1.1.1.5	Proje planının yapılandırma kontrolu altına alınması	1 day	Fri 26.12.08	Mon 29.12.08	8 hrs	7	YY
9		1.1.1.6	Proje planı	0 days	Mon 29.12.08	Mon 29.12.08	0 hrs	8	YY
10		1.1.2	**Proje izleme ve kontrol**	**416 days**	**Mon 29.12.08**	**Mon 02.08.10**	**96 hrs**	**9**	
11		1.1.2.1	Başlangıç toplantısı	1 day	Mon 29.12.08	Tue 30.12.08	16 hrs		PE;UY
12		1.1.2.2	İzleme toplantisi-1	1 day	Mon 02.02.09	Mon 02.02.09	16 hrs		PE;UY
13		1.1.2.3	Gelisme raporu-1	0 days	Mon 02.02.09	Mon 02.02.09	0 hrs	12	PY
14		1.1.2.4	İzleme toplantisi-2	1 day	Mon 02.03.09	Mon 02.03.09	16 hrs		PE;UY
15		1.1.2.5	Gelisme raporu-2	0 days	Mon 02.03.09	Mon 02.03.09	0 hrs	14	PY
16		1.1.2.6	...	1 day	Mon 06.04.09	Mon 06.04.09	16 hrs		PE;UY
17		1.1.2.7	...	0 days	Mon 06.04.09	Mon 06.04.09	0 hrs	16	PY
18		1.1.2.8	İzleme toplantisi-18	1 day	Mon 05.07.10	Mon 05.07.10	16 hrs		PE;UY
19		1.1.2.9	Gelisme raporu-18	0 days	Mon 05.07.10	Mon 05.07.10	0 hrs	18	PY
20		1.1.2.10	Kapanış toplantısı	1 day	Mon 02.08.10	Mon 02.08.10	16 hrs		PE;UY
21		1.1.2.11	Kapanış raporu	0 days	Mon 02.08.10	Mon 02.08.10	0 hrs	20	PY
22		1.2	**İŞ GEREKSİNİMLERİNİN BELİRLENMESİ**	**40 days**	**Mon 15.12.08**	**Fri 06.02.09**	**632 hrs**		
23		1.2.1	İş gereksinimlerinin incelenmesi	30 days	Mon 15.12.08	Fri 23.01.09	480 hrs		K1;K2
24		1.2.2	İş gereksinimleri belgesinin geliştirilmesi	5 days	Mon 26.01.09	Fri 30.01.09	80 hrs	23	K1;K2
25		1.2.3	İş gereksinimleri belgesinin gözden geçirilmesi	1 day	Mon 02.02.09	Mon 02.02.09	16 hrs	24	PE;Müşteri
26		1.2.4	İş gereksinimleri belgesinin günlenmesi	3 days	Tue 03.02.09	Thu 05.02.09	48 hrs	25	K1;K2
27		1.2.5	İş gereksinimleri belgesinin yapılandırma kontrolu altına alınması	1 day	Fri 06.02.09	Fri 06.02.09	8 hrs	26	YY
28		1.2.6	İş gereksinimleri belgesi	0 days	Fri 06.02.09	Fri 06.02.09	0 hrs	27	YY
29		1.3	**SİSTEM GEREKSİNİMLERİNİN VE MİMARİSİNİN BELİRLENMESİ**	**60 days**	**Mon 09.02.09**	**Fri 01.05.09**	**960 hrs**	**22**	
30		1.3.1	**Sistem gereksinimleri analizi**	**40 days**	**Mon 09.02.09**	**Fri 03.04.09**	**640 hrs**		
32		1.3.2	Sistem gereksinimleri belgesi	0 days	Fri 03.04.09	Fri 03.04.09	0 hrs	30	YY
33		1.3.3	**Sistem mimarisinin belirlenmesi**	**20 days**	**Mon 06.04.09**	**Fri 01.05.09**	**320 hrs**	**30**	
35		1.3.4	Sistem mimari dokumani	0 days	Fri 01.05.09	Fri 01.05.09	0 hrs	33	YY
36		1.4	**DONANIM GELİŞTİRME**	**40 days**	**Mon 04.05.09**	**Fri 26.06.09**	**800 hrs**	**29**	
38		1.5	**YAZILIM GELİŞTİRME**	**302 days**	**Mon 04.05.09**	**Tue 29.06.10**	**5.304 hrs**	**29**	
39		1.5.1	**Çevrim-1: ATM Kart Yazılım Paketi Geliştirme**	**245 days**	**Mon 04.05.09**	**Fri 09.04.10**	**2.344 hrs**		
40		1.5.1.1	**Çevrim-1: Yazılım gereksinimlerinin belirlenmesi**	**30 days**	**Mon 04.05.09**	**Fri 12.06.09**	**320 hrs**		
41		1.5.1.1.1	Yazılım gereksinimleri analizi	20 days	Mon 04.05.09	Fri 29.05.09	160 hrs		PYE
42		1.5.1.1.2	Yazılım gereksinimleri belgesinin hazırlanması	5 days	Mon 01.06.09	Fri 05.06.09	80 hrs	41	K1;K2
43		1.5.1.1.3	Yazılım gereksinimleri belgesinin gozden gecirilmesi	1 day	Mon 08.06.09	Mon 08.06.09	24 hrs	42	KG;PYE;Müşteri
44		1.5.1.1.4	Yazılım gereksinimleri belgesinin gunlenmesi	3 days	Tue 09.06.09	Thu 11.06.09	48 hrs	43	K1;K2
45		1.5.1.1.5	Yazılım gereksinimleri belgesinin yapılandırma kontrolu altına alınması	1 day	Fri 12.06.09	Fri 12.06.09	8 hrs	44	YY
46		1.5.1.1.6	Yazılım gereksinimleri belgesi	0 days	Fri 12.06.09	Fri 12.06.09	0 hrs	45	YY

Example: ATM System Software Development
(Tasks and dependencies – Gantt Chart)

ID	WBS	Task Name	Duration	Start	Finish	Work	Predecessors	Resource Names
1		1 ATM SISTEMI GELİŞTİRME PROVESİ	427 days	Mon 15.12.09	Tue 02.06.10	3,075 hrs		
2	1.1	PROJE YÖNETİMİ	426 days	Mon 15.12.09	Mon 02.06.10	268 hrs		
3	1.1.1	Proje planlaması	10 days	Mon 15.12.09	Mon 28.12.09	164 hrs		
4	1.1.1.1	Kaynakların yapılması	5 days	Mon 15.12.09	Mon 28.12.09	100 hrs	PY[50%],K1,K2	
5	1.1.1.2	Proje planının geliştirilmesi	5 days	Mon 15.12.09	Mon 28.12.09	16 hrs	PY	
6	1.1.1.3	Proje planının gözden geçirilmesi	1 day	Wed 24.12.09	Thu 25.12.09	32 hrs	KG,PY,PY,UY	
7	1.1.1.4	Proje planının güncelleme	1 day	Thu 25.12.09	Fri 26.12.09	8 hrs	PY	
8	1.1.1.5	Proje planının paydaşlarına kontrolü altına alınması	1 day	Fri 26.12.09	Mon 28.12.09	8 hrs	YY	
9	1.1.1.6	Proje planı	0 days	Mon 28.12.09	Mon 28.12.09	0 hrs	YY	
10	1.1.2	Proje izleme ve kontrol	415 days	Mon 28.12.09	Mon 02.06.10	96 hrs		
11	1.1.2.1	Başlangıç toplantısı	1 day	Mon 28.12.09	Tue 29.12.09	16 hrs	PE,UY	
12	1.1.2.2	İzleme toplantısı-1	1 day	Mon 02.02.09	Mon 02.02.09	16 hrs	PE,UY	
13	1.1.2.3	Gelişme raporu-1	0 days	Mon 02.02.09	Mon 02.02.09	0 hrs	PY	
14	1.1.2.4	İzleme toplantısı-2	1 day	Mon 02.03.09	Mon 02.03.09	16 hrs	PE,UY	
15	1.1.2.5	Gelişme raporu-2	0 days	Mon 02.03.09	Mon 02.03.09	0 hrs	PY	
16	1.1.2.6	...	1 day	Mon 06.04.09	Mon 06.04.09	16 hrs	PE,UY	
17	1.1.2.7	...	0 days	Mon 06.04.09	Mon 06.04.09	0 hrs	PY	
18	1.1.2.8	İzleme toplantısı-19	1 day	Mon 06.07.10	Mon 06.07.10	16 hrs	PE,UY	
19	1.1.2.9	Gelişme raporu-19	0 days	Mon 06.07.10	Mon 06.07.10	0 hrs	PY	
20	1.1.2.10	Kapanış toplantısı	1 day	Mon 02.06.10	Mon 02.06.10	16 hrs	PE,UY	
21	1.1.2.11	Kapanış raporu	0 days	Mon 02.06.10	Mon 02.06.10	0 hrs	PY	
22	1.2	İŞ GEREKSİNİMLERİNİN BELİRLENMESİ	30 days	Mon 15.12.09	Fri 06.03.09	320 hrs		
23	1.2.1	İş gereksinimlerinin modelenmesi	10 days	Mon 15.12.09	Fri 29.01.09	400 hrs	K1,K2	
24	1.2.2	İş gereksinimleri belgesinin geliştirilmesi	5 days	Mon 26.01.09	Fri 30.01.09	80 hrs	K1,K2	
25	1.2.3	İş gereksinimleri belgesinin gözden geçirilmesi	1 day	Mon 02.02.09	Mon 02.02.09	16 hrs	PE,Müşteri	
26	1.2.4	İş gereksinimleri belgesinin güncelleme	5 days	Tue 03.02.09	Thu 05.02.09	40 hrs	K1,K2	
27	1.2.5	İş gereksinimleri belgesinin paydaşlarına kontrolü altına alınması	1 day	Fri 06.02.09	Fri 06.02.09	8 hrs	YY	
28	1.2.6	İş gereksinimleri belgesi	0 days	Fri 06.02.09	Fri 06.02.09	0 hrs	YY	
29	1.3	SİSTEM GEREKSİNİMLERİNİN VE MİMARİSİNİN BELİRLENMESİ	40 days	Mon 09.02.09	Fri 01.05.09	960 hrs		
30	1.3.1	Sistem gereksinimleri analizi	30 days	Mon 09.02.09	Fri 03.04.09	640 hrs		
31	1.3.2	Sistem gereksinimleri belgesi	0 days	Fri 03.04.09	Fri 03.04.09	0 hrs	YY	
32	1.3.3	Sistem mimarisinin belirlenmesi	30 days	Mon 06.04.09	Fri 01.05.09	320 hrs		
33	1.3.4	Sistem mimari dokümanı	0 days	Fri 01.05.09	Fri 01.05.09	0 hrs	YY	
34	1.4	DONANIM GELİŞTİRME	40 days	Mon 04.05.09	Fri 26.06.09	960 hrs		
35	1.5	YAZILIM GELİŞTİRME	302 days	Mon 04.05.09	Tue 29.06.10	8,208 hrs		
36	1.5.1	Çevrim-1: ATM Kart Yazılım Paketi Geliştirme	245 days	Mon 04.05.09	Fri 09.04.10	2,344 hrs		
40	1.5.1.1	Çevrim-1: Yazılım gereksinimlerinin belirlenmesi	30 days	Mon 04.05.09	Fri 12.06.09	320 hrs		
41	1.5.1.2	Çevrim-1: Yazılım tasarımının yapılması	40 days	Mon 15.06.09	Fri 07.08.09	424 hrs	40	
42	1.5.1.3	Çevrim-1: Yazılım gerçekleştirme ve birim test	140 days	Fri 07.08.09	Fri 19.03.10	1,120 hrs	37	
44	1.5.1.4	Çevrim-1: Yazılım bütünleştirme ve test	20 days	Mon 22.02.10	Fri 19.03.10	320 hrs	50	K1,K2
45	1.5.1.5	Yazılım bütünleştirme test raporu	0 days	Fri 19.03.10	Fri 19.03.10	0 hrs	44	K1,K2
46	1.5.1.6	Yazılım güncelleme	5 days	Mon 22.03.10	Fri 26.03.10	40 hrs	45	PYE
47	1.5.1.7	Çevrim-1: Yazılım işlevsel test	5 days	Mon 29.03.10	Fri 02.04.10	80 hrs	46	K1,K2
48	1.5.1.8	Yazılım işlevsel test raporu	0 days	Fri 02.04.10	Fri 02.04.10	0 hrs	47	K1,K2
49	1.5.1.9	Yazılım güncelleme	5 days	Mon 05.04.10	Fri 09.04.10	40 hrs	48	PYE
50	1.5.2	Çevrim-2: Kredi Kart Yazılım Paketi Geliştirme	285 days	Mon 04.05.09	Fri 04.06.10	2,664 hrs		
101	1.5.3	Çevrim-1 ve Çevrim-2 bütünleştirme ve test	5 days	Mon 07.06.10	Fri 11.06.10	120 hrs	99,100	K1,K2,K3
102	1.5.4	Yazılım güncelleme	3 days	Mon 14.06.10	Wed 16.06.10	24 hrs	101	PYE
103	1.5.5	Yazılım işlevsel test	5 days	Thu 17.06.10	Wed 23.06.10	120 hrs	102	K1,K2,K3
104	1.5.6	Yazılım güncelleme	3 days	Thu 24.06.10	Mon 28.06.10	24 hrs	103	PYE
105	1.5.7	Yazılım kaynaklarının paydaşlarına kontrolü altına alınması	1 day	Tue 29.06.10	Tue 29.06.10	8 hrs	104	YY
106	1.5.8	Yazılım kodu	0 days	Tue 29.06.10	Tue 29.06.10	0 hrs	105	YY
107	1.6	SİSTEM TÜMLEŞTİRME VE TEST	10 days	Wed 30.06.10	Tue 13.07.10	400 hrs	30,106	KG,K1,K2,K3,K4
108	1.7	SİSTEM GÜNLEME	10 days	Wed 14.07.10	Tue 27.07.10	160 hrs	107	
109	1.7.1	Donanım güncelleme	10 days	Wed 14.07.10	Tue 27.07.10	80 hrs		PEE
110	1.7.2	Yazılım güncelleme	10 days	Wed 14.07.10	Tue 27.07.10	80 hrs		PYE
111	1.8	SİSTEM KABUL TEST	5 days	Wed 28.07.10	Tue 03.08.10	160 hrs	108	KG,K1,K3,Müşteri
112	1.9	SİSTEM TESLİM	0 days	Tue 03.08.10	Tue 03.08.10	0 hrs	111	

Key points

- The price charged for a system does not just depend on its estimated development costs; it may be adjusted depending on the market and organizational priorities.
- Plan-driven development is organized around a complete project plan that defines the project activities, the planned effort, the activity schedule and who is responsible for each activity.
- Project scheduling involves the creation of graphical representations the project plan. Bar charts show the activity duration and staffing timelines, are the most commonly used schedule representations.

FOR STUDENT READING…

Project plan supplements

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

Factors affecting software pricing

Factor	Description
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.

Estimation techniques

- Organizations need to make software effort and cost estimates. There are two types of technique that can be used to do this:
  - *Experience-based techniques* The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.
  - *Algorithmic cost modeling* In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved.

Experience-based approaches

- Experience-based techniques rely on judgments based on experience of past projects and the effort expended in these projects on software development activities.
- Typically, you identify the deliverables to be produced in a project and the different software components or systems that are to be developed.
- You document these in a spreadsheet, estimate them individually and compute the total effort required.
- It usually helps to get a group of people involved in the effort estimation and to ask each member of the group to explain their estimate.

Algorithmic cost modelling

- Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers:
  - Effort = A ´ $Size^B$ ´ M
  - A is an organisation-dependent constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes.
- The most commonly used product attribute for cost estimation is code size.
- Most models are similar but they use different values for A, B and M.

Estimation accuracy

- The size of a software system can only be known accurately when it is finished.
- Several factors influence the final size
  - Use of COTS and components;
  - Programming language;
  - Distribution of system.
- As the development process progresses then the size estimate becomes more accurate.
- The estimates of the factors contributing to B and M are subjective and vary according to the judgment of the estimator.

Estimate uncertainty

The COCOMO 2 model

- An empirical model based on project experience.
- Well-documented, ‘independent’ model which is not tied to a specific software vendor.
- Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- COCOMO 2 takes into account different approaches to software development, reuse, etc.

COCOMO 2 models

- COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.
- The sub-models in COCOMO 2 are:
  - Application composition model. Used when software is composed from existing parts.
  - Early design model. Used when requirements are available but design has not yet started.
  - Reuse model. Used to compute the effort of integrating reusable components.
  - Post-architecture model. Used once the system architecture has been designed and more information about the system is available.

COCOMO estimation models

Application composition model

- Supports prototyping projects and projects where there is extensive reuse.
- Based on standard estimates of developer productivity in application (object) points/month.
- Takes CASE tool use into account.
- Formula is
  - PM = (NAP ´ (1 - %reuse/100)) / PROD
  - PM is the effort in person-months, NAP is the number of application points and PROD is the productivity.

Application-point productivity

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NAP/month)	4	7	13	25	50

Early design model

- Estimates can be made after the requirements have been agreed.
- Based on a standard formula for algorithmic models
  - $PM = A \times Size^{B} \times M$ where
  - $M = PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED$;
  - A = 2.94 in initial calibration, Size in KLOC, B varies from 1.1 to 1.24 depending on novelty of the project, development flexibility, risk management approaches and the process maturity.

Multipliers

- Multipliers reflect the capability of the developers, the non-functional requirements, the familiarity with the development platform, etc.
  - RCPX - product reliability and complexity;
  - RUSE - the reuse required;
  - PDIF - platform difficulty;
  - PREX - personnel experience;
  - PERS - personnel capability;
  - SCED - required schedule;
  - FCIL - the team support facilities.

The reuse model

- Takes into account black-box code that is reused without change and code that has to be adapted to integrate it with new code.
- There are two versions:
  - Black-box reuse where code is not modified. An effort estimate (PM) is computed.
  - White-box reuse where code is modified. A size estimate equivalent to the number of lines of new source code is computed. This then adjusts the size estimate for new code.

Reuse model estimates 1

- For generated code:
  - PM = (ASLOC * AT/100)/ATPROD
  - ASLOC is the number of lines of generated code
  - AT is the percentage of code automatically generated.
  - ATPROD is the productivity of engineers in integrating this code.

Reuse model estimates 2

- When code has to be understood and integrated:
  - ESLOC = ASLOC * (1-AT/100) * AAM.
  - ASLOC and AT as before.
  - AAM is the adaptation adjustment multiplier computed from the costs of changing the reused code, the costs of understanding how to integrate the code and the costs of reuse decision making.

Post-architecture level

- Uses the same formula as the early design model but with 17 rather than 7 associated multipliers.
- The code size is estimated as:
  - Number of lines of new code to be developed;
  - Estimate of equivalent number of lines of new code computed using the reuse model;
  - An estimate of the number of lines of code that have to be modified according to requirements changes.

The exponent term

- This depends on 5 scale factors (see next slide). Their sum/100 is added to 1.01
- A company takes on a project in a new domain. The client has not defined the process to be used and has not allowed time for risk analysis. The company has a CMM level 2 rating.
  - Precedenteness - new project (4)
  - Development flexibility - no client involvement - Very high (1)
  - Architecture/risk resolution - No risk analysis - V. Low .(5)
  - Team cohesion - new team - nominal (3)
  - Process maturity - some control - nominal (3)
- Scale factor is therefore 1.17.

Scale factors used in the exponent computation in the post-architecture model

Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; extra-high means that the client sets only general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; extra-high means a complete and thorough risk analysis.
Team cohesion	Reflects how well the development team knows each other and work together. Very low means very difficult interactions; extra-high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organization. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.

Multipliers

- Product attributes
  - Concerned with required characteristics of the software product being developed.
- Computer attributes
  - Constraints imposed on the software by the hardware platform.
- Personnel attributes
  - Multipliers that take the experience and capabilities of the people working on the project into account.
- Project attributes
  - Concerned with the particular characteristics of the software development project.

The effect of cost drivers on effort estimates

Exponent value	**1.17**
System size (including factors for reuse and requirements volatility)	128,000 DSI
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2,306 person-months**

The effect of cost drivers on effort estimates

Exponent value	**1.17**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

Project duration and staffing

- As well as effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required.
- Calendar time can be estimated using a COCOMO 2 formula
  - $TDEV = 3 \times (PM)^{(0.33+0.2*(B-1.01))}$
  - PM is the effort computation and B is the exponent computed as discussed above (B is 1 for the early prototyping model). This computation predicts the nominal schedule for the project.
- The time required is independent of the number of people working on the project.

Staffing requirements

- Staff required can’t be computed by diving the development time by the required schedule.
- The number of people working on a project varies depending on the phase of the project.
- The more people who work on the project, the more total effort is usually required.
- A very rapid build-up of people often correlates with schedule slippage.

Key points

- Estimation techniques for software may be experience-based, where managers judge the effort required, or algorithmic, where the effort required is computed from other estimated project parameters.
- The COCOMO II costing model is an algorithmic cost model that uses project, product, hardware and personnel attributes as well as product size and complexity attributes to derive a cost estimate.

BBM-382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 3

Assoc.Prof.Dr. Ayça KOLUKISA TARHAN

Dr. Tuğba ERDOĞAN

CHAPTER 4 – REQUIREMENTS ENGINEERING

Waterfall Development Model - Revisited

Challenges - Revisited

- Characteristics of software:
  - Intangible (abstract)
  - Changeable

- Characteristics of software development:
  - Human-intensive
  - Multi-disciplinary

Business Domain

Technical Domain

Business Domains and Requirements

Education

Tourism

Development Team:
WHAT TO DEVELOP?

1) What are customer needs?
2) **How will software behave** to satisfy these needs?

Shopping

Requirements

- Requirements are the most critical aspect of the software development cycle.
  - Studies point to a more than 60% failure rate for software projects in the U.S., with poor requirements as one of the top five reasons.
  - Studies also show a high percentage of project schedule overruns, with 80% due to creeping requirements.

- Managing requirements must be an integral part of an organization's overall development process, and is vital to mitigating risk of cost overruns and delays on large development projects.

Standish Group Bi-Annual Chaos Study

MODERN RESOLUTION FOR ALL PROJECTS

	2011	2012	2013	2014	2015
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

The Modern Resolution (OnTime, OnBudget, with a satisfactory result) of all software projects from FY2011–2015 within the new CHAOS database. Please note that for the rest of this report CHAOS Resolution will refer to the Modern Resolution definition not the Traditional Resolution definition.

Requirements engineering

- *Requirements Engineering* is:
  - the process of establishing the services that the customer requires from a system, and
  - the constraints under which it operates and is developed.

- The *requirements* themselves 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 specification**, e.g., of a mathematical function

- This is inevitable as requirements may serve a *dual function*
  - the basis for a bid for a contract - therefore must be open to interpretation;
  - contract itself - therefore must be defined in detail;

Both statements may be called requirements.

A spiral view of the requirements engineering process

Stages (Levels) of requirements

- 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 and 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.
  - Written for systems and software developers.
- Typically, a user requirement is detailed into more than one system requirements.

Example: Flight Booking System (FBS)

- Scope of Development:
  - A Flight Booking System (FBS) that serves different functionalities for reservation agents and reservation managers shall be designed and developed.

Example: FBS – User Requirements

- **User requirement**
  - **(1) A reservation manager shall be able to manipulate flights (enter new ones, update existing flight information, delete flights, etc.), and generate inventory reports of flights.**
- **System requirements**
  - **(1.1)** A reservation manager should be able to manipulate flights (enter new ones, update existing flight information, delete flights, etc.), and generate inventory reports of flights. The inventory report should contain a summary of all fights in the system that still have unsold seats (either economy or economy), and the total number of unsold seats of business class and economy class in all flights.
  - **(1.2)** Flight information (to be manipulated by reservation managers) should include the following: airline code (e.g., AC), flight number (e.g.., 321), departure airport, departure terminal number, departure date and time, arrival airport, arrival terminal number, arrival date and time, cost of business class and cost of economy class ticket.
  - **(1.3)** The flight number is a 3-digit number, prefixed with 0's if less the actual number is less than 100.
  - **(1.4)** The airline code must be a two-letter code defined in http://en.wikipedia.org/wiki/IATA_airline_designator .
  - **(1.5)** The airport location must be one of airport with a three-letter code defined in http://en.wikipedia.org/wiki/List_of_airports_by_IATA_code.

Types of 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 also state what the system should <u>not</u> 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.

Example: FBS – Functional and Non-functional requirements

- Functional requirement:
  - **A reservation manager shall be able to log into the system with his/her username and password.**

Software Behavior

- Non-functional requirements:
  - The system shall enable log in of its users within 5 sec
    - ***Performance requirement***
  - The system shall enable secure log in of its users.
    - ***Security requirement***

Constraints on Software Behavior

Example: FBS – Usability (non-functional) requirement

- User need:
  - The system shall be easy to use by flight booking staff and shall be organized in such a way that user errors are minimized.

- Verifiable non-functional requirement:
  - Flight booking staff shall be able to use all the system functions after six hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

In-class exercise

- Write performance non-functional requirements for the FBS under these items:
  - User need
  - Verifiable non-functional requirement

- Time: 5 minutes

Types of Requirements (cont'd)

- **Domain requirements**
  - Constraints on the system from its operational domain
  - If not satisfied, the system may be unworkable
    - Requirements are expressed in the language of the application domain, and this is often not understood by engineers developing the system.
    - Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

- **Process requirements**
  - Requirements that are specified mandating a particular IDE, programming language, development method or standard.

Example: FBS – Domain and Process requirements

- Domain requirement:
  - When there is a natural disaster (e.g. strong storm or tornado) on a specific location and date as reported by Weather Monitoring System, all flights that depart from/arrive to that location on that date will be cancelled.

- Process requirements:
  - The system shall be developed by following Waterfall Development Model.
  - The system shall be a simple client-server, web-based application.
  - The system shall be developed with Java.

Specification of Requirements

- Requirements should be :
  - Unambigious:
    - They should not be interpreted in different ways by developers and users.
  - Complete:
    - They should include descriptions of all facilities required.
  - Consistent:
    - There should be no conflicts or contradictions in the descriptions of the system facilities.

The software requirements document

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

Users of a requirements document

System customers	Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.
Managers	Use the requirements document to plan a bid for the system and to plan the system development process.
System engineers	Use the requirements to understand what system is to be developed.
System test engineers	Use the requirements to develop validation tests for the system.

Requirements document variability

- Information in requirements document depends on type of system and the approach to development used, e.g.
  - A systems developed by Waterfall Model are required to have all details in its requirements document.
  - A system developed incrementally (e.g. by iterations) will typically have less detail in its requirements document.

- Requirements documents standards have been designed, e.g. **IEEE standard 830.**
  - These are mostly applicable to the requirements for large systems engineering projects.

Requirements engineering (RE) process

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

In-class exercise

- **Discuss with your friend and write down how you will conduct these for the FBS:**
  - Requirements elicitation and analysis
  - Requirements specification
  - Requirements validation
  - Requirements management

- **Time: 5 minutes**

A spiral view of the requirements engineering process - revisited

1) Requirements elicitation and analysis

- Technical staff works 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, domain experts, engineers, trade unions, etc.
  - These are called ***stakeholders***.
- Starts with *requirement discovery*

Requirements discovery

- It is the process of:
  - Gathering information about the required and existing systems, and
  - Distilling the user and system requirements from this information.
- Requires interaction with system stakeholders from managers to external regulators.

Problems of requirements elicitation and analysis

- 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.

System context

- From an early stage of system specification, we should decide on the boundaries of the system under development, e.g.:
  - What functionality will be included in the system,
  - What is provided (e.g. inputs) by the system's environment,
  - Manual versus automated functionality of business processes.

- We can use a *context model* to represent this information (see Chapter 5).

Example: FBS – Context model

A spiral view of the requirements engineering process

2) Requirements specification

- *Requirements specification* is the process of writing 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 requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify requirements by defining an operational model of 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 functional requirements for 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

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, and ***may*** for optional 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.

Example: FBS requirements – Natural Language

1. A reservation manager *shall* be able to use the system to create accounts for reservation agents.
2. The system *shall* enable a reservation manager to log in with his/her username and password.
3. The reservation manager *shall* be able to manipulate flights (enter new ones, update existing flight information, delete flights, etc.), and generate inventory reports of flights.
4. When a flight is canceled by the reservation manager, all itineraries that are reserved or booked and include that flight *shall* be canceled as well.
5. The reservation manager *shall* be able to view information of a reservation agent and confirm a ticket for him/her.
6. The reservation manager *shall* be able to log out at any stage during his/her session.

Example: FBS requirements – Structured Natural Language (e.g. Tabular structure)

Use case: Login	
Actors	Traveler
Precondition	Traveler is not logged in
Post-condition	Traveler is logged in
Main (happy) path:	**1.** **User** enters her/his username and password and presses on the Login button **2.** **System** checks the username and password and shows the app's main window (page) **3.** **User** will be able to use the feature afterwards
Alternative path:	1. If the username and password combination is invalid, the system will show a message indicating it 2. User will be able to enter another username and password

- Interaction between user(s) and the software system to be developed!
- Only outside behavior of the systems as observed by the user(s)!

In-class exercise

- **Write the structured tabular requirements for the feature "Search for Flights"**
- **Time: 5 minutes**

Use case: Login	
Actors	Traveler
Precondition	Traveler is not logged in
Post-condition	Traveler is logged in
Main (happy) path:	1. User enters her/his username and password and presses on the Login button 2. System checks the username and password and shows the app's main window (page) 3. User will be able to use the feature afterwards
Alternative path:	1. If the username and password combination is invalid, the system will show a message indicating it 2. User will be able to enter another username and password

Scenarios (or user stories)

- *Scenarios* are real-life examples of how a system can be used.

- They 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.

Example: FBS Scenarios (in Natural Language)

- Scenario: Search for flights & create reservation
  - When a reservation agent wants to create and book a travel itinerary, he/she first shall search for flight information. A list of available flight options shall be shown with departure/arrival and cost information. Once the agent selected an itinerary from the list, he/she shall have the option to reserve it.
- Scenario: Buy a ticket
  - Once the reservation agent reserved an itinerary, he/she shall have the option to book it by providing payment information via credit card. If the credit card information is not on-file for the agent, he/she shall be prompted to enter the credit card information. Once the credit card is validated, the reservation agent shall be shown with the actual ticket information.

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 requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify requirements by defining an operational model of 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 functional requirements for 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

Scope of FBS Requirements by Use-Case Diagram

Use cases

- *Use-cases* are a scenario based technique in UML (Unified Modeling Language), which identify actors in an interaction and describe externally observable interaction between the system and its actors.

- A set of use cases should describe all possible interactions with the system (e.g. Use-case diagram).
  - High-level graphical model supplemented by more detailed tabular description (see Chapter 5).
  - Activity diagrams or sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system (see Chapter 5).

Exercise for HOME

- System: student registration management system (like BILSIS2)
- Draw the use-case diagram

Example: FBS – Details of "Search for Flights" use-case description by Activity Diagram

Example: FBS – Details of "Buy a Ticket" use-case description by Activity Diagram

A spiral view of the requirements engineering process

3) Requirements validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.

- Requirements error costs are high so validation is very important.
  - Fixing a requirements error after delivery may cost **up to 100 times** the cost of fixing an implementation error.

Relative cost factor to find and fix defects

Checklist for requirements validation

- 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 verified (e.g. tested)?

Requirements validation techniques

- Requirements review
  - Systematic manual analysis of the requirements.
- Prototyping
  - Using an executable model of the system to check requirements (see Chapter 2).
- Test-case generation
  - Developing tests for requirements to check testability.

Requirements review

- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal.
- Good communications between developers, customers and users can resolve problems at an early stage.

Checklist for requirements review

- 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?

4) Requirements management

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.

- New requirements emerge as a system is being developed and after it has gone into use.
  - We keep track of individual requirements and maintain *links between dependent requirements* so that you can assess the impact of requirements changes.
  - We establish a *formal process* for making change proposals and linking these to system requirements.

Requirements evolution

Changing requirements

- **The business and technical environment of the system always changes after installation.**
  - New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change and new legislation and regulations may be introduced that the system must necessarily abide by.

- **The people who pay for a system and the users of that system are rarely the same people.**
  - System customers impose requirements due to organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

Changing requirements

- Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
  - The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements change management

Requirements change management

- *Problem analysis and change specification*
  - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
- *Change analysis and costing*
  - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
- Change implementation
  - Requirements document and, where necessary, system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

Traceability of Requirements to Requirements

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			D		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1								R
3.2							R	

D: “dependent”
R: “relational”

Traceability of Requirements to Test Cases

Requirement Traceability Matrix											
	Test Case ID	TC_1	TC_2	TC_3	TC_4	TC_5	TC_6	TC_7	TC_8	TC_9	TC_10	# Test Cases for respective Requirement
Req. ID												
Req_1		X		X			X					3
Req_2			X			X						2
Req_3				X								1
Req_4					X		X					2
Req_5						X		X				2
Req_6							X					1
Req_7						X		X				2
Req_8									X			1
Req_9										X		1
Req_10											X	1

©TestingClub.blogspot.com

Requirements Engineering – Wrap up

http://www.mks.com/solutions/discipline/rm/requirements-engineering

FOR STUDENT READING…

Key points

- *Requirements engineering* process is an iterative process including requirements elicitation, specification and validation.
- Requirements elicitation and analysis 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, scenarios, use-cases and ethnography.

Key points

- Requirements specification is the process of writing the user and system requirements in a requirements document.
- 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.

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.

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.
- 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.

User and system requirements

User requirement definition

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

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall automatically 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, 20 mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Readers of different types of requirements specification

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.

Functional requirements for the MHC-PMS

- 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.
- Ex: Consider the term ‘search’ in the first requirement
  - 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, it is very difficult 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.

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

Types of nonfunctional requirements

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.

Examples of nonfunctional requirements in the MHC-PMS

Product requirement
The MHC-PMS 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.

Non-functional classifications

- Organisational requirements
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
  - Requirements that 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 MHC-PMS

Organizational requirement
Users of the MHC-PMS 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.

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 elicitation and analysis process

Requirements elicitation and analysis: 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.

Stakeholders in the MHC-PMS

- 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 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 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.

The structure of a requirements document

Chapter	Description
System requirements specification	This should describe functional and nonfunctional requirements in more detail. Interfaces to other systems may be defined.
System models	This might include graphical system models (object models, data-flow models, or semantic data models) showing relationships between system components and system and its environment.
System evolution	This should describe 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 application being developed; for example, hardware and database descriptions. Hardware requirements define minimal and optimal configurations for system. Database requirements define logical organization of data used by system and relationships between data.
Index	Several indexes to 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.

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 requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify requirements by defining an operational model of 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 functional requirements for 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

A) Natural language specification

- Requirements are written as natural language sentences supplemented by diagrams and tables.
  - It is expressive, intuitive and universal.
  - Requirements can be understood by users and customers.

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.

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.

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.)

B) 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.

B-1) 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 (r2), the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered.
Destination	Main control loop.
Action	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requirements	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition	r0 is replaced by r1 then r1 is replaced by r2.
Side effects	None.

B-2) 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

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.

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

Scenario for collecting medical history in MHC-PMS

Initial assumption: The *patient* has seen a *medical receptionist* who has created a record in system and collected patient's personal information (name, address, age, etc.). A *nurse* is logged on to system and is collecting medical history.

Normal: *Nurse* searches for the patient by family name. If there is more than one patient with same surname, given name (first name in English) and date of birth are used to identify patient.

Nurse chooses the menu option to add medical history.

Nurse then follows a series of prompts from the system to enter information about consultations elsewhere

- on mental health problems (free text input),
- existing medical conditions (nurse selects conditions from menu),
- medication currently taken (selected from menu),
- allergies (free text),
- and home life (form).

Scenario for collecting medical history in MHC-PMS

What can go wrong: Patient's record does not exist or cannot be found. *Nurse* should create a new record and record personal information.

Patient conditions or medication are not entered in menu. *Nurse* should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. *Nurse* should enter free text recording the patient's inability/unwillingness to provide information. System should print standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to patient.

Other activities: Record may be consulted but not edited by other staff while information is being entered.

System state on completion: User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.

Use cases for the MHC-PMS

The structure of a requirements document

Chapter	Description
Preface	This should define expected readership of document and describe its version history, including a rationale for creation of a new version and a summary of changes made in each version.
Introduction	This should describe need for the system. It should briefly describe system's functions and explain how it will work with other systems. It should also describe how system fits into the overall business or strategic objectives of organization commissioning software.
Glossary	This should define technical terms used in document. You should not make assumptions about experience or expertise of reader.
User requirements definition	Here, you describe services provided for user. 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 anticipated system architecture, showing distribution of functions across system modules. Architectural components that are reused should be highlighted.

Requirements management planning

- Establishes the level of requirements management detail that is required.
- Requirements management decisions:
  - *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
  - *A change management process* This is the set of activities that assess impact and cost of changes.
  - *Traceability policies* These policies define relationships between each requirement and between requirements and system design that should be recorded.
  - *Tool support* Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

BBM-382 – SOFTWARE ENGINEERING
SPRING 2020
Lecture 4

Assoc.Prof.Dr. Ayça KOLUKISA TARHAN
Dr. Tuğba ERDOĞAN

Before we start

- **What is requirement?**
  - **Requirement, functional**
    - **A statement of some <u>function</u> or <u>feature</u> that should be implemented in a system**
  - **Requirement, non functional**
    - **A statement of a <u>constraint</u> that applies to a system**

Functional (A) of non-functional (B) ?

The system sends an email to the customer when she places a new order.

functional

Functional (A) of non-functional (B)?

The mail should be send not later than 12 hours after the order has been placed.

non-functional

Functional requirements frequently describe

Non-functional requirements

- Non-functional requirement relates to quality attributes: e.g., **performance, learnability, availability**
- functional requirement: "when the user presses the green button the Options dialog appears”:
  - **performance**: how quickly the dialog appears;
  - **availability**: how often this function may fail, and how quickly it should be repaired;
  - **learnability**: how easy it is to learn this function.

Exercise: Functional or Non-functional

- The system shall be able to process at least 40 executing jobs at a time.

  non-functional, performance

- The system shall provide the means for the resource provider to see on which project his resource is working.

  functional

- The system shall provide the means for the system admin to perform his actions on a computer with Windows XP, Mac OS X or Linux.

  non-functional, portability

- If one of the resource disappears while it was performing a job, the system should requeue the job.

  functional

- The (de-)installation of the software needed by resource providers should not require a computer expert.

  non-functional, installability

From the book

- Read it online!

5

System modeling

Objectives

The aim of this chapter is to introduce some types of system model that may be developed as part of the requirements engineering and system design processes. When you have read the chapter, you will:

- understand how graphical models can be used to represent software systems;
- understand why different types of model are required and the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- have been introduced to some of the diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- be aware of the ideas underlying model-driven engineering, where a system is automatically generated from structural and behavioral models.

Contents

5.1 Context models
5.2 Interaction models
5.3 Structural models
5.4 Behavioral models
5.5 Model-driven engineering

UML

- 14 diagrams in total

Part 1 Introduction to Software Engineering

Chapter 1 Introduction
Chapter 2 Software processes
Chapter 3 Agile software development
Chapter 4 Requirements engineering
Chapter 5 System modeling
Chapter 6 Architectural design
Chapter 7 Design and implementation
Chapter 8 Software testing
Chapter 9 Software evolution

Unified Modeling Language - UML

- UML has become the de facto standard for modeling software applications and is growing in popularity in modeling other domains. Its roots go back to three distinct methods:
  - Booch Method -- Grady Booch
  - Object Modeling Technique -- James Rumbaugh
  - Objectory -- Ivar Jacobson
- Known as the Three Amigos, Booch, Rumbaugh, and Jacobson kicked off what became the first version of UML, in 1994.
  - In 1997, UML was accepted by the Object Management Group (OMG) & released as UML v1.1.
- Since then, UML has gone through several revisions and refinements leading up to the current 2.5.1 (2017) release.
  - https://www.omg.org/spec/UML/About-UML/
  - Specification is extensive in terms of page count (796 pages)

UML Basics – 1

- UML is a language, it has both syntax and semantics.
  - When you model a concept in UML, there are rules regarding how the elements can be put together and what it means when they are organized in a certain way.
  - UML is intended not only to be a pictorial representation of a concept, but also to tell you something about its context.

- You can apply UML in any number of ways; common uses include:
  - Designing software
  - Communicating software or business processes
  - Capturing details about a system for requirements or analysis
  - Documenting an existing system, process, or organization

UML Basics – 2

- The basic building block of UML is a *diagram*.
  - A diagram graphically represents things, and the relationships between these things.
    - These things can be representations of real-world objects, pure software constructs, or a description of the behavior of some other object.

UML Rules of Thumb

- Nearly everything in UML is optional
  - UML provides a language to capture information that varies greatly depending on the domain of the problem, and you don't need to use every part of UML in every model you create.
  - At times there is more than one way to convey the same information; use what is familiar to your audience.
- UML models are rarely complete
  - As a consequence of everything being optional, it is common for a UML model to be missing some details about a system. The trick is to not miss key details that could impact your system design.
- UML is designed to be open to interpretation
  - While the UML specification does a good job of laying down the groundwork for a modeling language, it is critical that within an organization or group of users you establish how and when to use a language feature.
    - E.g. an aggregation relationship to indicate a C++ pointer and a composition relationship to indicate a C++ reference.
  - It is a good practice to put together a document on modeling guidelines.

Requirements document in practice

SEVENTH FRAMEWORK PROGRAMME

INFORMATION AND COMMUNICATION TECHNOLOGIES

Project:

Accessibility Assessment Simulation Environment for New Applications Design and Development (ACCESSIBLE, Grant Agreement No. 224145)

accessible

Applications Design and Development

Deliverable number and title:

D 2.2b – User needs and System Requirements Specification

Lead beneficiary:	SUN
WP, no, title and activity type	WP2 – User needs, benchmarking, accessibility and simulation requirements
Contributing Task (s)	T2.1, T2.2
Dissemination level	RE
Delivery date	September 2009
Status	Final Draft
File name and size	[illegible]

Final Draft Page 1 of 174- SUN

Page 1 of 174-

Requirements document

What should a user be able to do?

We need a better way

- **Goal:** specify how different parties (users, administrators, external systems, …) can interact with our system
  - Recall: unambiguous, realistic, verifiable and evolvable
- **Plain-text** does not work
  - A more **structured way** is needed

Simple example

- What can you do with a telephone?

http://globestaronline.com/panasonic-kx-ts500mx-corded-telephone/#.UrF1wPTuLy0

Simple example

- What can you do with a telephone?

Basic use case diagram

Simple example

- What can you do with a telephone?

Basic use case diagram

What is an *Actor*?

- **A class of entities** (human or computer), falling **beyond the system boundaries**, **interacting with the system**
  - Actor can be an **external software system**
  - One might have **multiple actors in a use case**: caller/callee
  - One user might be represented by **multiple actors**
    - doctor can be a patient

What is a *Use Case*?

Place a call

- **Use case:** a contract between the stakeholders of a system about its behavior
  - Stakeholder: a person, group or organization with an interest in a project
- Usually, a large **end-to-end** process comprising several (inter)actions
  - Internal operations are NOT use cases
  - Individual methods are NOT use cases
  - BUT it is one process: specification usually involves multiple use cases

Is placing a call large?

Place a call

- How do you make a phone call?

a) The user picks up the telephone hook connected to the telephone line “A”.

b) If the line is free, the user receives a dial tone sent by the line .

c) The user dials number “B”.

d) The call request is forwarded to the switch center.

e) If line “B” is not busy, the call request is forwarded to “B” and a tone is sent to “A”.

f) “B”’s telephone rings.

Main Scenario

g) If somebody at “B” picks up the hook, the ringing tone at “A” is stopped and a telephone connection will commence.

What if the “if” conditions do not hold?

- If the line is free, the user receives a dial tone sent by the line.
  - b-1) If the telephone line is engaged in a conversation, the user will be connected to the same conversation.
  - b-2) If the user does not dial a number for a certain amount of time, a permanent tone is emitted by the switch center, no further call will be accepted and the user has to replace the hook.

Alternatives

Use case description

- **Pre-condition:** when a use case is available to its user
  - The telephone set is connected to the telephone line “A”, it is on-hook and there is no incoming call (it is not ringing).
- **Trigger**: action that initiates the use case
  - The user picks up the telephone hook connected of the telephone set (connected to line “A”) and dials number “B”.
- **Guarantee** (**post-condition**): what does the user achieve through the use case
  - A communication between “A” and “B” will commence
- **Main scenario**
- **Alternatives**

Exercise

Single
Day Return
5 Return ticket
Weekend Return
Railrunner 4-11 (incl.) years
Press one of the blue boxes on the screen
User presses a "Day Return" button is a
a) Trigger ✓
b) Pre-condition
c) Guarantee (post-condition)
Other tickets
To Belgium / Luxemburg / France
To Germany
Nederlands
English
Stop
Clear all

Summary so far

What is an *actor*?

- A **class of entities** (human or computer), falling **beyond the system boundaries**, **interacting** with the system

- NB:
  - Actor can be an **external software system**
  - One might have **multiple actors in a use case:** caller/callee
  - **One user** might be represented by **multiple actors**
    - doctor can be a patient

Use case description

- **Pre-condition:** when a use case is available to its user
  - The telephone set is connected to the telephone line "A", it is on-hook and there is no incoming call (it is not ringing).
- **Trigger:** action that initiates the use case
  - The user picks up the telephone hook connected of the telephone set (connected to line "A") and dials number "B".
- **Guarantee (post-condition)**: what does the user achieve through the use case
  - A communication between "A" and "B" will commence
- **Main scenario**
- **Alternatives**

Formally

- Dependency: A use case depends on another use case for realizing its goal.
  - The target use case is always used by the source.

- What if the use case is used only sometimes?
  - Alternatives in the use case
  - Extension

Example

	<<include>> (A → B)	<<extend>> (A ← B)
Read	A includes B	B extends A
A can operate without B	no	yes
B is aware of A	no	yes

Which model is correct?

- Digital learning environment supports students' learning through quizzes. Given your knowledge of this kind of systems which of the following models would be correct?

Flight Booking System (FBS)

(see "FBS User Requirements and Use-case Modeling.pdf")

- A Flight Booking System (FBS) that serves different functionalities for reservation agents and reservation managers shall be designed and developed.

FBS – Use-Case Diagram

Tabular description of use-cases

Use case: Login	
Actors	Traveler
Precondition	Traveler is not logged in
Post-condition	Traveler is logged in
Main (happy) path:	1. User enters her/his username and password and presses on the Login button 2. System checks the username and password and shows the app's main window (page) 3. User will be able to use the feature afterwards
Alternative path:	1. If the username and password combination is invalid, the system will show a message indicating it 2. User will be able to enter another username and password

Exercise

- System: student information system (like BILBSIS2)
- Design:
  - The context diagram
  - The use-case diagram
  - Three tabular descriptions for three use-cases
  - Three activity diagrams for three other use-cases

For student reading

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.

Real vs. Model – Beytepe Kampüsü

- Real – Satellite View

- Model – Vector Map

Existing and planned system models

- Models of the existing system are used during requirements engineering, re-engineering and reverse-engineering .
  - They help analyst clarify what 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.
  - Re-engineering: Systematic starting over and reinventing the way a firm, a software, or a business process, work (code refactoring in our context)
  - Reverse engineering: taking apart an object to see how it works in order to duplicate or enhance the object. The practice, taken from older industries, is now frequently used on computer hardware and software
  - Re-engineering ≠ Reverse engineering

Existing and planned system models

- Models of the new system are used during requirements engineering to help explain proposed requirements to other system stakeholders.
  - Engineers use these models to discuss design proposals and to document system for implementation.
  - In a model-driven engineering process, it is possible to generate a complete or partial system implementation from system model.

Stakeholders of a Company

http://upload.wikimedia.org/wikipedia/commons/4/4c/Stakeholder_%28en%29.png

System perspectives

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

System modeling metaphore

http://www.slideshare.net/jcabot/mde-20-pragmatic-model-verification-and-other-stories-habilitation-public-lecture

FBS – User Requirements (READ LATER)

- The system shall be a simple, stand-alone application.
  - "Stand-alone" means that it shall not be developed as a web-based on-line application, but a one which runs as an application on a single computer.
- A reservation manager shall be able to use the system to create accounts for reservation agents. The system shall enable a reservation manager to log in with his/her username and password. The reservation manager shall be able to manipulate flights (enter new ones, update existing flight information, delete flights, etc.), and generate inventory reports of flights. When a flight is canceled by the reservation manager, all itineraries that are reserved or booked and include that flight shall be canceled as well. The reservation manager shall be able to view information of a reservation agent and confirm a ticket for him/her. The reservation manager shall be able to log out at any stage during his/her session.

FBS – User Requirements (..cont'd)

- A reservation agent (traveler) shall be able to buy a ticket by using the system. An agent shall be able to register himself/herself with the reservation site by entering personal profile information. The system shall enable a reservation agent to log in with his/her username and password. Once the agent is logged in, he/she shall be shown with a list of travel itineraries along with the status of each itinerary (reserved, booked, confirmed, or canceled). A travel itinerary is a travel arrangement with one or more flights, e.g., an example travel itinerary might consist of the following three flights: Calgary-Vancouver, Vancouver-Los Angeles and Los Angeles-New York. The reservation agent shall create a reservation and pay for it to book the flight. The agent shall cancel a previously created reservation in his/her itinerary list, but the cancellation of an itinerary cannot be rolled back.

FBS – User Requirements (..cont'd)

- When a reservation agent wants to create and book a travel itinerary, he/she first shall search for flight information. A list of available flight options shall be shown with departure/arrival and cost information. Once the agent selected an itinerary from the list, he/she shall have the option to reserve it. Once the reservation agent reserved an itinerary, he/she shall have the option to book it by providing payment information via credit card. If the credit card information is not on-file for the agent, he/she shall be prompted to enter the credit card information.

FBS – User Requirements (..cont'd)

- Once the credit card is validated, the reservation agent shall be shown with the actual ticket information. The ticket number shall be unique for each agent in each flight. If the credit card number is invalid, an error message shall be displayed to the agent and the agent shall be asked to re-enter the credit card information or cancel the booking process. If a reserved itinerary is not paid for by 5 minutes from the time of reservation, it cannot be booked anymore and it shall be canceled by the system. Cancellation after payment shall not be permitted. The reservation agent shall be able to log out at any stage during his/her session. If the agent is logged out during the preparation of an itinerary, the next time when he/she is logged in again, the reservation agent shall be able to continue with the previous itinerary-preparation process.

FBS – Context Diagram

Context diagram of FlightBooking System

- Example of APIs…

UML

- 14 diagrams in total

Part 1 Introduction to Software Engineering

Chapter 1 Introduction
Chapter 2 Software processes
Chapter 3 Agile software development
Chapter 4 Requirements engineering
Chapter 5 System modeling
Chapter 6 Architectural design
Chapter 7 Design and implementation
Chapter 8 Software testing
Chapter 9 Software evolution

In-class exercise

- Write the tabular description of use-case “View My Tickets”
- Time: 5 minutes

FBS – Details of "Search for Flights" use-case desc. by Activity Diagram

FBS – Details of "Buy a Ticket" use-case description by Activity Diagram

In-class exercise

- Draw the Activity Diagram for use-case “Create Account”
- Time: 5 minutes

BBM-382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 5

Assoc. Prof. Dr. Ayça KOLUKISA TARHAN
Dr. Tuğba Erdoğan

Chapter 5 – System Modeling

From the book

- Read it online!

5

System modeling

Objectives

The aim of this chapter is to introduce some types of system model that may be developed as part of the requirements engineering and system design processes. When you have read the chapter, you will:

- understand how graphical models can be used to represent software systems;
- understand why different types of model are required and the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- have been introduced to some of the diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- be aware of the ideas underlying model-driven engineering, where a system is automatically generated from structural and behavioral models.

Contents

5.1 Context models
5.2 Interaction models
5.3 Structural models
5.4 Behavioral models
5.5 Model-driven engineering

Chapter 5 – System Modeling - Topics covered

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

UML

- 14 diagrams in total

Part 1 Introduction to Software Engineering

Chapter 1 Introduction
Chapter 2 Software processes
Chapter 3 Agile software development
Chapter 4 Requirements engineering
Chapter 5 System modeling
Chapter 6 Architectural design
Chapter 7 Design and implementation
Chapter 8 Software testing
Chapter 9 Software evolution

UML

- Four types of diagrams may be used for interaction modelling.
- 14 diagrams in total

Interaction models – need to keep UI in mind

- Users will INTERACT with the system using the GUI (UI)
- Prototyping of GUI would be useful
- Use cases:

Example: FBS – Details of "Search for Flights" use-case description by Activity Diagram

Design of Interaction models based on UI prototypes

Sequence diagram for Search for Flights use-case

The Model-View-Controller (MVC) pattern

View (UI)

Controller
(brain, business logic)

Model

The Model-View-Controller (MVC) pattern

In-class exercise

- Draw the sequence diagram for use-case “View My Tickets”
- Time: 5 minutes

In-class exercise

- Draw the sequence diagram for use-case “View My Tickets”
- Let’s use online tools such as www.draw.io
- Or http://www.visual-paradigm.com/download/community.jsp

Sequence diagram for Search for Flights use-case

- Important points:
  - All the method names on the message MUST exists in the class definitions
  - **IN CLASS DIAGRAMS**

Sequence diagram for Search for Flights use-case

- Important points:
  - All the method names on the message MUST exists in the class definitions
  - **LATER IN THE SOURCE CODE**

3) 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.

Interaction models - Sequence diagrams

- Sequence diagrams are part of UML and are used to model interactions between actors and objects within a system.
- A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.

Sequence diagram for Search for Flights use-case

- Important points:
  - Object creation messages…

Sequence diagram

- Important points:
  - All the method names on the message MUST exists in the class definitions
  - **IN CLASS DIAGRAMS**

Sequence diagram for Login use-case

- Solution is next…

Sequence diagram for Login use-case

Sequence diagram for Login use-case

Sequence diagram for Login use-case

Sequence diagrams: learn more

https://www.google.com.tr/search?q=Sequence+diagram&espv=2&biw=1269&bih=864&site=web

Google

Sequence diagram

All Images Videos Books Apps More Search tools

About 8,020,000 results (0.43 seconds)

Images for Sequence

Report images

Sequence diagrams: learn more

- **This sequence diagram shows the flow of a search method. The user will enter a keyword or name of a product in the Search textbox on a Web form**

Chapter 5 – System Modeling - Topics covered

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

UML

- 14 diagrams in total

Part 1 Introduction to Software Engineering

Chapter 1 Introduction
Chapter 2 Software processes
Chapter 3 Agile software development
Chapter 4 Requirements engineering
Chapter 5 System modeling
Chapter 6 Architectural design
Chapter 7 Design and implementation
Chapter 8 Software testing
Chapter 9 Software evolution

Structural models - 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 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.*

Adding methods and attributes for classes

Classes and associations in FlightBooking System

- Zoomed next…

Class diagram of FlightBooking

Class diagram of FlightBooking

4) 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.
- You create structural models of a system when you are discussing and designing the system architecture.

- 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.

In-class exercise

- System: student information system (like BILSIS2)
- Derive the classes and draw the class diagram
- Time: 5 minutes

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 (trees, 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. Çam, Kavak, Köknar -> Ağaç

tree

A generalization hierarchy with added detail

- In object-oriented languages, such as Java, generalization is implemented using class inheritance mechanisms built into language.
- Attributes and operations associated with higher-level classes (super-classes) are also associated with lower-level classes (subclasses.)
- Subclasses inherit attributes and operations from their superclasses and add more specific attributes and operations.

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.

Class diagram: learn more online

Class diagram: learn more online

Chapter 5 – System Modeling - Topics covered

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

UML

- 14 diagrams in total

Part 1 Introduction to Software Engineering

Chapter 1 Introduction
Chapter 2 Software processes
Chapter 3 Agile software development
Chapter 4 Requirements engineering
Chapter 5 System modeling
Chapter 6 Architectural design
Chapter 7 Design and implementation
Chapter 8 Software testing
Chapter 9 Software evolution

Data-driven modeling using activity diagrams

DATA

5) Behavioral models

- Behavioral models are models of 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 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 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 analysis of requirements as they can be used to show end-to-end processing in a system.

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 diagram of Flight class

- Only for state-dependent classes/objects

OCL

In-class exercise

- Draw the state diagram for class Ticket
- Time: 5 minutes

State chart of a land-line phone

State charts for Control and Embedded Systems in Matlab Simulink (can be directly executed – converted to code)

State charts: a cruise control system

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.
- State-charts are an integral part of the UML and are used to represent state machine models.

Chapter 5 – System Modeling - Topics covered

- Context models
- Interaction models
- Structural models
- Behavioral models
- Model-Driven engineering (MDE)

Model-Driven Engineering (MDE)

- Model-driven engineering (MDE) is an approach to software development where models rather than programs are principal outputs of development process.
- Programs that execute on a hardware/software platform are then generated automatically from models.
- Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Usage of Model-Driven Engineering (MDE)

- MDE is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.
- 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 costs of developing translators for new platforms.

A metaphor for Model-Driven Engineering

Ref:http://www.theenterprisearchitect.eu/blog/2009/08/05/a-metaphor-for-model-driven-engineering/

Executable UML

- The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible.
- This is possible using a subset of UML 2, called Executable UML or xUML.

Books

https://www.google.com.tr/search?q=Executable+UML&oq=Executable+UML&aqs=chrome..69i57j0l5.47

Google

Executable UML

All | Images | Videos | News | Books | More | Search tools

About 7,070 results (0.60 seconds)

Executable UML: A Foundation for Model-driven Architecture

https://books.google.com.tr/books?isbn=0201748045

Stephen J. Mellor, Marc J. Balcer - 2002 - Preview - More editions

This timely new book, "Executable UML: A Foundation for Model-Driven Architecture, " thoroughly introduces, documents, and explains this important new technology.

Model Driven Architecture with Executable UML - Volume 1

https://books.google.com.tr/books?isbn=0521537711

Chris Raistrick - 2004 - Preview

CD-ROM contains: iUMLite software, documentation, tutorials and example databases.

Executable UML: How to Build Class Models

https://books.google.com.tr/books?id=MrdQAAAAMAAJ

Leon Starr - 2002 - Snippet view

The practical guide to developing class models with Executable UML. In "Executable UML," Leon Starr shows how to build precise class models that form the foundation of rigorous software specifications.

Model-Driven Development with Executable UML

https://books.google.com.tr/books?isbn=0470535997

Dragan Milicev - 2009 - Preview - More editions

Tools for ‘Executable UML‘

Tools for ‘Executable UML‘

- Many!

modeling-languages.com/list-of-executable-uml-tools/

It looks like Executable UML is getting increasingly popular (again) maybe because the new Executable UML standards (fuml and Alf) we have now available. Executable UML aims at defining UML models with a behavioral specification precise enough to be effectively executed. In its purest state, Executable UML eliminates the need of programming the software system.

I'm basing my opinion on the common question I've been asked more and more during the last months: which tools out there provide support for Executable UML? In fact, I don't have a clear idea myself but I though I could just try to write down what I know and then hope that you, my dear readers, will help me to complete/update the list (thanks Ed for doubling the length of the lists with your suggestions!) . So, here we go (in no particular order). For each tool we provide the name and URL, whether the tool is free, commercial or whatever and if the tool supports the recent Executable UML standards or its own kind of executable UML.

Name	License	Standard?	Comments
fUML Ref. Implementation	Open source	Yes	Reference implementation that can assist in evaluating vendor implementations conformance with the specification.
Cameo Simulation Toolkit	Commercial	Yes	Model execution framework based on OMG fUML and W3C SCXML standards. Offered as an extension of MagicDraw
LieberLieber AM|USE 2.0	Commercial	Yes	Offers both state machine and activity model execution. Integrated in Sparx Enterprise

Feedback

END OF CHAPTER 5

FOR STUDENTS

State diagram of Ticket class

- Only for state-dependent classes/objects

Example: Embedded software to control a microwave oven

Context models - 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.
  - Social and organisational concerns may affect the decision on where to position system boundaries.

2) Process process models

- Context models simply show the other systems in the environment, not how the system under development (SuD) is used in that environment.

- Process models reveal how SuD is used in broader business processes.
  - UML activity diagrams may be used to define business process models.

Process model of involuntary detention

Interaction models - Use case modeling

- Use cases were developed originally to support requirements elicitation and now incorporated into 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 use case and in a more detailed textual form.

A use case in MHC-PMS

Use cases in MHC-PMS involving role 'Medical Receptionist'

Tabular description of Transfer data use-case

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

Sequence diagram for View Patient Info

Sequence diagram for Transfer Data

Classes and associations in MHC-PMS

Order processing

State diagram of a microwave oven

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.

Microwave oven operation

Model-Driven Architecture (MDA)

- 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 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 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

Types of model

- A computation independent model (CIM)
  - These model the important domain abstractions used in a system. CIMs are sometimes called domain models.

Types of model

- A platform independent model (PIM)
  - These model the operation of 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.

Types of model

- *Platform specific models (PSM)*
  - These are transformations of 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.

Multiple platform-specific models

BBM-382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 6

Assoc. Prof. Dr. Ayça KOLUKISA TARHAN
Dr. Tuğba ERDOĞAN

Chapter 6 – Architectural Design

From the book

- Read it online!

6

Architectural design

Objectives

The objective of this chapter is to introduce the concepts of software architecture and architectural design. When you have read the chapter, you will:

- understand why the architectural design of software is important;
- understand the decisions that have to be made about the system architecture during the architectural design process;
- have been introduced to the idea of architectural patterns, well-tried ways of organizing system architectures, which can be reused in system designs;
- know the architectural patterns that are often used in different types of application system, including transaction processing systems and language processing systems.

Contents

6.1 Architectural design decisions
6.2 Architectural views
6.3 Architectural patterns
6.4 Application architectures

Chapter 6-Topics covered

- Review of OO and class diagrams (basics of SW architecture)
- On overview of Architectural design
- Architectural views
- Architectural patterns
- Application architectures

Class diagrams (zoomed next…)

Class diagrams

Class diagrams

From Class diagrams to SW architecture

Architectural “layers” (tiers)

Showing SW architecture in UML

- 14 diagrams in total

UML Package Diagram - Example

https://www.uml-diagrams.org/package-diagrams-overview.html

UML Component Diagram - Example

https://creately.com/blog/diagrams/uml-diagram-types-examples/

UML Deployment Diagram: Example

http://www.conceptdraw.com/How-To-Guide/uml-deployment-diagram

Bank Server

<<Database>>

CustomerDB

Mortgage Application

IMortgage Application

Real Estate Server

Listing

<<Storage>>

MultipleListings

IListing

a PC

BuyerInterface

TCP/IP

TCP/IP

UML Deployment Diagram: 4-Tier Architecture

https://conceptdraw.com/a661c3/preview--UML%20deployment%20diagram%20template

Chapter 6-Topics covered

- Review of OO and class diagrams
- On overview of Architectural design
- Architectural views
- Architectural patterns
- Application architectures

Software architecture versus ‘regular’ architecture

Software architecture

- The design process for identifying the major system components making up a system and their control and communication is architectural design.
  - An early stage of the design process - Represents the link between specification and design processes.
  - Often carried out in parallel with some specification activities.
  - The output of this design process is a description of the system/software architecture.

Note that sub-system = component = module in this context.

A general model of the design process (Ch. 2)

- The diagram suggests that the stages of the design process are sequential.
- In fact, design process activities are interleaved. Feedback from one stage to another and consequent design rework is inevitable in all design processes.

Design activities (Ch.2)

- *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.
- *Interface design*, interfaces between system components.
- *Component design*, you take each system component and design how it will operate.
- *Database design*, you design the system data structures and how these are to be represented in a database.

The architecture of a packing robot control system

Architectural Design identifies the major system components making up a system and their control and communication.

https://www.youtube.com/watch?v=cf7M4XnIzU8

Architectural abstraction

- Architecture in the small is concerned with the architecture of individual programs.
  - At this level, we are concerned with the way that an individual program is decomposed into components.

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

Architectural representations

- Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
  - criticized to be very abstract because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
  - However, useful for communication with stakeholders and for project planning.

Architectural representations

- By hand

- Using software tools

Architectural representations

- By hand

Architectural design decisions

- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.
  - Is there a generic application architecture that can be used?
  - How will the system be distributed?
  - What architectural styles are appropriate?
  - What approach will be used to structure the system?
  - How will the system be decomposed into modules?
  - What control strategy should be used?
  - How will the architectural design be evaluated?
  - How should the architecture be documented?

Architecture and system characteristics

- Performance
  - Localize critical operations and minimize communications.
- Security
  - Use a layered architecture with critical assets in the inner layers.

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

Software in automotive and aviation

Sources: IEEE Automotive designline
http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code

Automotive software

spectrum.ieee.org/cars-that-think/transportation/systems/bmw-fixes-vehicle-software-flaw-that-would've-let-hackers-in

IEEE.org | IEEE Xplore Digital Library | IEEE Standards | IEEE Spect

IEEE SPECTRUM

Follow on:

MIT PROFESSIONAL EDUCATION
Advanced Study Program

Engineering Topics | Special Reports | Blogs | Multimedia | The Ma

Cars That Think | Transportation | Systems

BMW Fixes Software Flaw That Would've Let Hackers Unlock Doors

By Willie Jones
Posted 6 Feb 2015 | 18:00 GMT

Jeep Hacking 101

Miller and Valasek reveal some of the details of their car hacking at Blackhat
8 Aug 2015

Architecture and system characteristics

- Availability
  - Include redundant components and mechanisms for fault tolerance.
- Maintainability
  - Use fine-grain, replaceable components.

An important topic in software architecture: Cohesion and Coupling

- Cohesion is the degree to which the elements in a design unit (package, class etc.) are logically related, or "belong together".
- Coupling is the degree to which the elements in a design are connected.

Cohesion and Coupling: examples from code

Coincidental cohesion is when parts of a module are grouped arbitrarily; the only relationship between the parts is that they have been grouped together.

Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module

https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

Cohesion and Coupling: examples from code

https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

Cohesion and Coupling: examples from code

https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

Cohesion and Coupling

- A good architecture:
  - Maximizes the **cohesion** of each module
    - Goal: the contents of each module are strongly inter-related
    - High cohesion means the subcomponents really do belong together
  - Minimizes **coupling** between modules:
    - Goal: modules don't need to know much about one another to interact
    - Low coupling makes future change easier

Cohesion and Coupling

Conway's Law:

- *"The structure of a software system reflects the structure of the organisation that built it"*

- "Bir yazılım sisteminin yapısı onun inşa eden firmanın yapısını yansıtır"

Cohesion and Coupling

- Correspondence of socio-technical relationship and dependencies

Cohesion and Coupling

- Goal is to: Maximize Cohesion and Minimize coupling

Cohesion and Coupling: Tools

There are many tools to do “dependency analysis”

Cohesion and Coupling: Tools

There are many tools to do “dependency analysis”

A good book on SW architecture

https://leanpub.com/visualising-software-architecture/read

Store Read Write Support Blog

Search Leanpub

The Art of Visualising Software Architecture

Table of Contents

- 1. Introduction
  - 1.1 Past, present and future
  - 1.2 About this book
- 2. Draw one or more diagrams
  - 2.1 Where do we start?
  - 2.2 Some examples
  - 2.3 Common problems
  - 2.4 The hidden assumptions of diagrams
- 3. Creating a shared vocabulary
  - 3.1 Common abstractions over a common notation
  - 3.2 Static structure
  - 3.3 Software systems
  - 3.4 Containers
  - 3.5 Components
  - 3.6 Components vs classes
  - 3.7 Non-OO components
  - 3.8 Modules and subsystems?

The Art of Visualising Software Architecture

Communicating software architecture with sketches, diagrams and the C4 model

Simon Brown

$0.00 $20.00

Add Ebook to Cart

Let’s watch now!

- https://www.youtube.com/watch?v=0o9_zjZeJuE
- 38 minutes

Watch later

- https://vimeo.com/167722769

Chapter 6-Topics covered

- On overview of Architectural design
- Architectural views
- Architectural patterns
- Application architectures

Architectural views

- What views or perspectives are useful when designing and documenting a system's architecture?
- What notations should be used for describing architectural models?
- Each architectural model only shows one view or perspective of the system.
  - It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network.

4 + 1 view model of software architecture
(introduced by Philippe Kruchten)

https://www.researchgate.net/publication/238381956_A_41_View_Model_of_Software_Architecture

4 + 1 view model of software architecture

For both design and documentation, you usually need to present multiple views of the software architecture:

- A logical view, which shows the key abstractions in the system as objects or object classes.
- A process view, which shows how, at run-time, the system is composed of interacting processes.
- A development view, which shows how the software is decomposed for development.
- A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
- Related use cases or scenarios (+1)

Use of architectural models

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

Chapter 6-Topics covered

- On overview of Architectural design
- Architectural views
- Architectural patterns
- Application architectures

Architectural patterns

- In software engineering, a pattern is a general repeatable solution to a commonly occurring problem in software engineering.
- Patterns are a means of representing, sharing and reusing knowledge.
- There are different types of pattern in SW eng:
  - Architectural patterns
  - Design patterns
  - Test patterns
  - Performance patterns

Architectural patterns

- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when they are not useful.
- Patterns may be represented using tabular and graphical descriptions.
- Review of several Architectural patterns:
  - 1 - Model-View-Controller (MVC) pattern
  - 2 - Layered architecture pattern
  - 3 - Repository architecture pattern
  - 4 - Client-server architecture pattern
  - 5 - Pipe and filter architecture pattern

1 - Model-View-Controller Pattern

Web application architecture using the MVC pattern

Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. Model component manages data and associated operations on that data. View component defines and manages how data is presented to user. Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to View and Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when data model and interactions are simple.

2 - Layered architecture pattern

User interface

User interface management
Authentication and authorization

Core business logic/application functionality
System utilities

System support (OS, database etc.)

Layered architecture pattern

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

Layered architecture pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Architecture of android

The architecture of a library system

3 - Repository architecture pattern

A repository architecture for an IDE

Repository architecture pattern

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an efficient data sharing mechanism.

Repository architecture pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

4 - Client-server architecture pattern

- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

A client–server architecture for a film library

Client–server architecture pattern

Name	Client-server
Description	The functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	Servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

5 - Pipe and filter architecture pattern

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

An example of the pipe and filter architecture

Pipe and filter architecture pattern

Name	Pipe and filter
Description	The processing of data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

Chapter 6-Topics covered

- On overview of Architectural design
- Architectural views
- Architectural patterns
- Application architectures

FOR STUDENT REVIEW…

Application architectures

- Application systems are designed to meet an organizational need.
- As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.

Use of application architectures

- As a starting point for architectural design.
- As a design checklist.
- As a way of organising the work of the development team.
- As a means of assessing components for reuse.
- As a vocabulary for talking about application types.

Examples of application types

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

Application type examples

- Focus here is on transaction processing and language processing systems.
- Transaction processing systems
  - E-commerce systems;
  - Reservation systems.
- Language processing systems
  - Compilers;
  - Command interpreters.

2 - Transaction processing systems

- Process user requests for information from a database or requests to update the database.
- From a user perspective a transaction is:
  - Any coherent sequence of operations that satisfies a goal;
  - For example - find the times of flights from London to Paris.
- Users make asynchronous requests for service which are then processed by a transaction manager.

The structure of transaction processing applications

Software architecture of an ATM system

Information systems architecture

- Information systems have a generic architecture that can be organized as a layered architecture.
- These are transaction-based systems as interaction with these systems generally involves database transactions.
- Layers include:
  - The user interface
  - User communications
  - Information retrieval
  - System database

Layered information system architecture

Architecture of the MHC-PMS

Web-based information systems

- Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
  - For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
  - In an e-commerce system, the application-specific layer includes additional functionality supporting a ‘shopping cart’ in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

Server implementation

- These systems are often implemented as multi-tier client server/architectures (discussed in Chapter 18)
  - The web server is responsible for all user communications, with the user interface implemented using a web browser;
  - The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
  - The database server moves information to and from the database and handles transaction management.

4 - Language processing systems

- Accept a natural or artificial language as input and generate some other representation of that language.
- May include an interpreter to act on the instructions in the language that is being processed.
- Used in situations where the easiest way to solve a problem is to describe an algorithm or describe the system data
  - Meta-case tools process tool descriptions, method rules, etc and generate tools.

The architecture of a language processing system

Compiler components

- A lexical analyzer, which takes input language tokens and converts them to an internal form.
- A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- A syntax analyzer, which checks the syntax of the language being translated.
- A syntax tree, which is an internal structure representing the program being compiled.
- A semantic analyzer that uses information from syntax tree and symbol table to check the semantic correctness of the input language text.
- A code generator that 'walks' the syntax tree and generates abstract machine code.

A pipe and filter compiler architecture

A repository architecture for a language processing system

Learn more

https://www.google.com.tr/search?q=android+architecture&espv=2&biw=1920&bih=935&source=lnms&tbm=isch&sa=X&ved=0ahUKE
Google
software architecture
All
Images
Videos
Books
News
More
Search tools
Diagram
Layers
Icon
SOA
Simple

Key points

- Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- The architecture of a system may be designed around one of more architectural patterns or ‘styles’.
  - These capture the essence of an architecture and can be instantiated in different ways.
  - Discussed later in this lecture.

Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- The architecture of a system may be designed around one of more architectural patterns or ‘styles’.
  - These capture the essence of an architecture and can be instantiated in different ways.
  - Discussed later in this lecture.

Advantages of explicit architecture

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Product-line architectures may be developed.

4 + 1 view model of software architecture

functionality that the system provides to end-users

from a programmer's perspective and is concerned with software management

A small set of use cases. Describe sequences of interactions between objects, and between processes.

dynamic aspects of the system, explains the system processes and how they communicate

depicts the system from a system engineer's point of view

Software sizes

Software sizes

Software sizes

BBM-382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 7

Assoc. Prof. Dr. Ayça KOLUKISA TARHAN
Dr. Tuğba ERDOĞAN

Chapter 7 – Design and Implementation

From the book

✧ Read it online!

7

Design and implementation

Objectives

The objectives of this chapter are to introduce object-oriented software design using the UML and highlight important implementation concerns. When you have read this chapter, you will:

- understand the most important activities in a general, object-oriented design process;
- understand some of the different models that may be used to document an object-oriented design;
- know about the idea of design patterns and how these are a way of reusing design knowledge and experience;
- have been introduced to key issues that have to be considered when implementing software, including software reuse and open-source development.

Contents

7.1 Object-oriented design using the UML
7.2 Design patterns
7.3 Implementation issues
7.4 Open source development

Chapter 7 – Design and Implementation

Part 1 Introduction to Software Engineering

Chapter 1 Introduction
Chapter 2 Software processes
Chapter 3 Agile software development
Chapter 4 Requirements engineering
Chapter 5 System modeling
Chapter 6 Architectural design
Chapter 7 Design and implementation
Chapter 8 Software testing
Chapter 9 Software evolution

4.1 Functional and non-functional requirements
4.2 The software requirements document
4.3 Requirements specification
4.4 Requirements engineering processes
4.5 Requirements elicitation and analysis
4.6 Requirements validation
4.7 Requirements management

Use-case diagrams

5.1 Context models
5.2 Interaction models
5.3 Structural models
5.4 Behavioral models
5.5 Model-driven engineering

6.1 Architectural design decisions
6.2 Architectural views
6.3 Architectural patterns
6.4 Application architectures

7.1 Object-oriented design using the UML
7.2 Design patterns
7.3 Implementation issues
7.4 Open source development

Chapter 7 – Design and Implementation

- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open-source development

More about UML

✧ Latest version:

https://www.omg.org/spec/UML/

RESOURCE HUB · OMG SPECIFICATIONS · PROGRAMS · MEMBERSHIP · MEMBERS AREA

ABOUT THE SPECIFICATION

Title:	Unified Modeling Language
Acronym:	UML®
Version:	2.5.1
Description:	A specification defining a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.
Publication Date:	December 2017
IPR Mode:	RF-Limited
Keywords:	Abstraction Action Sequence Action State Activity Graph Architecture Association Class Diagram Collaboration Diagram Component Diagram Control Flow Data Flow Deployment Diagram Execution Implementation Pins Procedure
Categories:	Modeling Software Engineering
Issues Reported:	https://issues.omg.org/issues/spec/UML/2.5.1
Issues Fixed:	https://issues.omg.org/issues/spec/UML/2.5.1/fixed
Report an Issue:	https://issues.omg.org/issues/create-new-issue?specification=UML%232.5.1

SPECIFICATION DOCUMENTS

NORMATIVE DOCUMENTS

Click the URL links to download the document.

DOCUMENT NUMBER	EXPLANATION	FORMAT	URL
formal/17-12-05	Specification	PDF	UML/2.5.1/PDF

More about UML

More about UML

Before 95' - Fragmentation ▶ 95' - Unification ▶ 98' - Standardization ▶ 99' - Industrialization

More about UML

Part 1 Introduction to Software Engineering

Chapter 1 Introduction
Chapter 2 Software processes
Chapter 3 Agile software development
Chapter 4 Requirements engineering
Chapter 5 System modeling
Chapter 6 Architectural design
Chapter 7 Design and implementation
Chapter 8 Software testing
Chapter 9 Software evolution

✧ 14 diagrams in total

Lots of UML books: a few of them

Lots of UML books: a few of them

Lots of UML books: by programming language

Fig. 3 Histogram of UML books published based on programming language focus

Reference: V. Garousi, "Classification and trend analysis of UML books", Software & Systems Modeling, 2012.

Lots of UML books: by programming language

✧ One book PDF in our website:

UML for Java Programmers		
PDF:		
https://drive.google.com/file/d/0B6dKdxaNjBENRjRqUDZUby1xMFE/view?usp=sharing		

Fig. 3 Histogram of UML books published based on programming language focus

Lots of UML books: by programming language

Listing 3-1
UI.java

```
public class UI implements
  WithdrawlUI, DepositUI, TransferUI
{
  private Screen itsScreen;
  private MessageLog itsMessageLog;

  public void displayMessage(String message)
  {
    itsMessageLog.logMessage(message);
    itsScreen.displayMessage(message);
  }
}
```

Figure 3-8
ATM Class Diagram

Chapter 7 – Design and Implementation

- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open-source development

Design patterns

- In software engineering, a pattern is a general repeatable solution to a commonly occurring problem in software engineering.
- Patterns are a means of representing, sharing and reusing knowledge.
- There are different types of pattern in SW engineering:
  - Architectural patterns
  - Design patterns
  - Test patterns
  - Performance patterns
  - Process patterns
  - …

Design patterns: An example
Multiple displays using the Observer pattern

The Observer pattern

- Name
  - Observer.
- Description
  - Separates the display of object state from the object itself.
- Problem description
  - Used when multiple displays of state are needed.

Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

Pattern elements

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

REMINDER:
Multiple displays using the Observer pattern

A UML model of the Observer pattern

Design patterns

✧ Levels of design

Design patterns: many of them

Design pattern catalog

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	• Factory Method	• Adapter	• Interperter
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Momento • Observer • State • Strategy • Vistor

GoF – Who are they ?

- *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides* (Gang of Four)
- They published a book "Design Pattens by GoF" in 1994 which was a major milestone in OO development
- This book is considered the "bible" for Design Patterns
- It describes 23 patterns for good OO design

Design patterns: many of them

✧ C: Creational, S: Structural, B: Behavioral

THE 23 GANG OF FOUR DESIGN PATTERNS

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

Design patterns: We just spent a few minutes!

✧ There are courses and many books on Design patterns…

Chapter 7 – Design and Implementation

- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open-source development

Implementation issues

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

Implementation issues

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

Code Reuse:
A bit of history

- ✧ Ad hoc code reuse has been practiced from the earliest days of programming.
- ✧ Programmers have always reused sections of code, functions, and procedures.
- ✧ Software reuse as a recognized area of study in software engineering, however, dates only from 1968 when Douglas McIlroy of Bell Laboratories proposed basing the software industry on reusable components.

Reuse

- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse of software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

A spectrum of reuse

Types of reuse

- Concerning motivation and driving factors, reuse can be:
  - Opportunistic - While getting ready to begin a project, the team realizes that there are existing components that they can reuse.
  - Planned - A team strategically designs components so that they'll be reusable in future projects.
- Reuse can be categorized further:
  - Internal reuse - A team reuses its own components. This may be a business decision, since the team may want to control a component critical to the project.
  - External reuse - A team may choose to license a third-party component. Licensing a third-party component typically costs the team 1 to 20 percent of what it would cost to develop internally. The team must also consider the time it takes to find, learn and integrate the component.

Reuse levels

- The abstraction level
  - At this level, you don’t reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Use versus reuse

Reuse costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Reuse: costs versus benefits

✧ Do the Reuse only when:

Costs < Benefits

Book on Reuse: Many…

Implementation issues

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

Configuration management

- Configuration management is the name given to the general process of managing a changing software system.
- The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
- See also Chapter 25 of the book

Configuration management activities

- Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Configuration management: Version management

https://en.wikipedia.org/wiki/List_of_version_control_software

Français
Українська
Edit links

Client-server model [edit]

In the client-server model, developers use a shared single repository.

Open source [edit]

- Concurrent Versions System (CVS) – originally built on RCS, licensed under the GPL.
  - CVSNT – cross-platform port of CVS that allows case insensitive file names among other changes
  - OpenCVS – CVS clone under the BSD license, with emphasis put on security and source code correctness
- Subversion (SVN) – versioning control system inspired by CVS[4]
- Vesta – build system with a versioning file system and support for distributed repositories

Proprietary [edit]

- Filesentral – GUI based version control aimed primarily at the non - programmers demographic
- AccuRev – source configuration management tool with integrated issue tracking based on "Streams" that efficiently mana
- Autodesk Vault – Version control tool specifically designed for Autodesk applications managing the complex relationships
- CADES - Designer productivity and version control system by International Computers Limited.
- Dimensions CM - software change and configuration management system developed by Serena Software that includes
- IBM Rational ClearCase – SCC compliant configuration management system by IBM Rational Software
- IBM Configuration Management Version Control (CMVC) – version control system, no longer available.
- IBM Rational Team Concert – Collaboration and application lifecycle management platform by IBM Rational Software
- IC Manage Global Design Platform (GDP) – design data management for IC design and Perforce infrastructure support.
- MKS Integrity
- Panvalet - Around since the 1970s, source and object control for IBM mainframe computers.
- Perforce – Free for up to 20 users.
- PVCS – originally Polytron Version Control System, developed by Don Kinzer at Polytron, first released in 1985. Now owr
- Quma Version Control System

Configuration management: Version management

subversion.apache.org

SUBVERSION®

THE APACHE SOFTWARE FOUNDATION http://www.apache.org/

About Subversion
- News
- Features
- Documentation
- FAQ
- Roadmap
- Security
- Quick Start

Getting Subversion
- Source Download
- Binary Packages

Apache™ Subversion®

"Enterprise-class centralized version control for the masses"

Welcome to **subversion.apache.org**, the online home of the Apache Subversion™ software project. Subversion is an open source version control system. Founded in 2000 by CollabNet, Inc., the Subversion project and software have seen incredible success over the past decade. Subversion has enjoyed and continues to enjoy widespread adoption in both the open source arena and the corporate

https://git-scm.com

git --everything-is-local

Search entire site...

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.

Learn Git in your browser for free with Try Git.

Configuration management: Version management

✧ Version control graphs…

Configuration management: Version management

- Code churn: defined as lines added, modified or deleted to a file from one version to another. We dimension it by the following:
  - User
  - Date/Time
  - Changeset
  - Portfolio Project
  - Path

Configuration management: Version management

- Studying code growth and churn
- http://www.easterbrook.ca/steve/2009/07/growth-of-code-complexity/

Configuration management: Version management

- Code churn versus quality: more changes would USUALLY results in more bugs!
- https://docs.codeclimate.com/docs/churn

Configuration management: Version management

✧ Many tools to manage large code bases and monitor quality…

CODE CLIMATE Repos

Worker Feed Classes Smells Trends Security

Summary of February 11th - 17th

42 files changed, 1,054 insertions, 316 deletions

D → B Quality::CanonicalLocator has improved. about a month ago

D → F Quality::Worker::Jobs::ScanVulnerabilities has gotten worse. about a month ago

One class/module was added. about a month ago

B Quality::Worker::Jobs::CompareVulnerabilities

B → C Quality::Worker::Jobs::SnapshotTree has gotten worse. about a month ago

Summary of February 4th - 10th

38 files changed, 799 insertions, 527 deletions

A → D Quality::ConstantDefinition has gotten worse. about a month ago

A → D Quality::MethodDefinition has gotten worse. about a month ago

Classes by Rating

2.67 GPA

Hotspots

F Quality::Worker::Jobs::CompareSnapshots

F Quality::Worker::Jobs::ScanVulnerabilities

F Quality::Worker::Jobs::FinalizeSnapshot

D Quality::ConstantDefinition

D Quality::MethodDefinition

Configuration management: System integration

- **Continuous Integration** (CI) is a development practice that requires developers to **integrate** code into a shared repository several times a day.
- Each check-in is then verified by an automated build, allowing teams to detect problems early.

Configuration management: System integration

https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Comparison of continuous integration software

From Wikipedia, the free encyclopedia

Continuous integration software supports continuous integration, a set of software engineering practices that speed the delivery of software by decreasing integration times.

Comparison [edit]

Name	Platform	License	Windows builders	Java builders	Other builders	Notification	IDE integration	Other integr
AnthillPro	Cross-platform	Proprietary	MSBuild,[1] NAnt,[2] Visual Studio[3]	Ant,[4] Maven 1, 2, & 3[5]	Shell script, Batch script, Cross-platform command-line, Groovy, Make,[6] RTC Jazz,[7] TFS Build,[8] Custom Script Interpreter	E-mail, XMPP/Jabber, RSS, Systray	Eclipse, Visual Studio	(many)
Apache Continuum	JDK, Servlet Container	Apache 2.0	Unknown	Maven 1 & 2 & 3	Shell script[10]	Mail, Jabber and Google Talk, MSN, IRC, report deployment with wagon	Unknown	Unknow
Apache Gump	Python	Apache 2.0	Unknown	Ant, Maven 1	Unknown	E-mail	Unknown	Unknow
			Visual Studio					

Configuration management: System integration

Configuration management: System integration

CruiseControl.NET
CONTINUOUS INTEGRATION SERVER

Dashboard > local > AutomationTest > 2011-03-10 17:08:29 (Failed)

Latest
Next
Previous

Build Report
View Build Log
Automation Test Report

Recent Builds

2011-03-24 17:37:47 (25)
2011-03-24 15:50:00 (24)
2011-03-24 15:42:11 (23)
2011-03-24 14:16:41 (22)
2011-03-24 13:12:30 (21)
2011-03-24 09:59:23 (20)
2011-03-24 09:38:48 (Failed)
2011-03-24 09:13:59 (Failed)
2011-03-24 09:10:00 (Failed)
2011-03-23 15:55:14 (Failed)
Show All

BUILD FAILED

Project: AutomationTest
Date of build: 2011-03-10 17:08:29
Running time: 00:06:53
Integration Request: Build (ForceBuild) triggerec

Modifications since last build (0)

AutomationTest Report - 17:08:37
TestName: TestList
Duration: 00:05:01.5830189
StartTime: 10-03-2011 17:10:18
EndTime: 10-03-2011 17:15:20
Overall RresultTest:

Project	Passed Tests	Failed Tests	Coverage
17:08:37	5	2	71.4 %

TestName	Duration	TestResult
AddGroupOfProductTest	00:01:04.0657139	
UpdateGroupOfProductTest	00:00:40.0217943	
DeleteGroupOfProductTest	00:00:34.6941032	
AddUnicodeTest	00:00:39.7084492	
DeleteUnicodeTest	00:00:32.8812365	
ActiveVectorTest	00:00:44.7838262	
InActiveVectorTest	00:00:45.2628482	

Configuration management: Issue tracking

- Support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

- There are usually four types of issues:
  - Bugs
  - Feature requests
  - Support requests
  - Source-code patches

Configuration management: Issue tracking

✧ For Android, as of Mar. 21, 2016, 47376 issues were reported!

https://code.google.com/p/android/issues/list

vgarousi@gmail.com ▾ | My favorites ▾ | Profile | Sign out

android

Android Open Source Project - Issue Tracker

Search projects

Project Home | Issues

New issue | Search | Open issues ▾ | for | Search | Advanced search | Search tips | Subscriptions

1 - 100 of 47376 Next › | List | Grid

ID ▾	Status ▾	Priority ▾	Owner ▾	Summary + Labels ▾	Stars ▾	Reporter ▾	Opened ▾
55288	Unassigned	----	to...@android.com	'Check for updates now' throws 'Connection failed' error	83	vinovar...@gmail.com	May 2013
56157	Unassigned	----	c...@google.com	Running tests - feature/change request	26	werner...@gmail.com	Jun 2013
60423	New	----	esteban...@google.com	Traceview - tabular view doesn't show times split by caller/callee	23	vs...@google.com	Sep 2013
60424	New	----	esteban...@google.com	Traceview - timeline at the top needs to be more fine grained	16	vs...@google.com	Sep 2013
60425	New	----	esteban...@google.com	traceview - the graphical and tabular views need to be linked	20	vs...@google.com	Sep 2013
60427	New	----	esteban...@google.com	Traceview - restrict the amount by which the flamechart can be zoomed/panned	15	vs...@google.com	Sep 2013
63135	New	----	r...@google.com	No option for setting gravity of text in textView to "justify" option	11	sandeepi...@gmail.com	Dec 2013
64594	New	----	joeferna...@google.com	documentation in Sample says "POST body must be a JavaBean", which is not reflected in the code.	12	andrewne...@gmail.com	Jan 2014
66006	New	----	di...@android.com	call on emulator disconnect emulator from ADB	11	zktk...@gmail.com	Feb 2014

Configuration management: Issue tracking

✧ A SIMPLE issue handling process for projects

Configuration management: Issue tracking

✧ A MORE COMPLEX issue handling process for projects

Configuration management: Issue tracking

✧ Let us see some Assigned issues

Configuration management: Issue tracking

- ✧ Let us see some Assigned issues
- ✧ Not fixed as of Mar. 21, 2016!

android

Android Open Source Project - Issue Tracker

Search projects

Project Home | Issues

New issue | Search | Open issues | for status=Assigned | Search | Advanced search | Search tips | Subscriptions

1 - 100 of 3206 Next › | List Grid

ID	Status	Priority	Owner	Summary + Labels	Stars	Reporter	Opened
66388	Assigned	---	y...@google.com	Use the manifest merger api rather than a custom implementation of it	8	vs...@google.com	Feb 2014
73204	Assigned	---	ddr...@google.com	External Documentation (for API) does not work	19	Ronald.J...@gmail.com	Jul 2014
74208	Assigned	---	alr...@google.com	navigator: Show all source sets, not just the currently active one	15	vs...@google.com	Jul 2014
81643	Assigned	---	trevorjohns@google.com	No sample preview images in Samples wizard	5	katherin...@google.com	Dec 2014
81778	Assigned	---	di...@android.com	PANIC: Could not open: Nexus_5_API_21_x86_Google_APIs	6	fredsa@google.com	Dec 2014
81826	Assigned	---	tnor...@google.com	StackOverflowError at java.security.AccessController.doPrivileged	4	tobias.preuss	Dec 2014
82458	Assigned	---	b...@google.com	cannot run emulator: missing sdk/tools\emulator-x86_64.exe	2	snkumar...@gmail.com	Dec 2014
91423	Assigned	---	tnor...@google.com	Refactoring Package name doesn't change applicationID	1	imoro...@google.com	Dec 2014
91463	Assigned	---	tnor...@google.com	New projects are created as broken	1	vaclav.k...@gmail.com	Dec 2014
93290	Assigned	---	j...@google.com	Do Not Ask People To Edit Non-Editable Files	2	mmur...@commonsware.com	Jan 2015
98328	Assigned	---	rslo...@google.com	sdk folder missing from android studio linux download	7	DavidRus...@gmail.com	Jan 2015

Configuration management: Issue tracking

- Issues keep coming!
- Date: Mar. 21, 2016

android

Android Open Source Project - Issue Tracker

Search projects

Project Home | Issues

New issue | Search | Open issues | for | Search | Advanced search | Search tips | Subscriptions

1 - 100 of 47358 Next › List Grid

ID	Status	Priority	Owner	Summary + Labels	Stars	Reporter	Opened
204608	New	Medium	----	in dual sim phone multiple data can be activated by menu and switching of sim is not possible on home screen directly	1	kanavk...@gmail.com	moments ago
204607	New	----	gcabil...@google.com	Support for Java 8 bytecode (libraries) by Jill	1	far...@gmail.com	52 minutes ago
204606	New	----	to...@android.com	Instant run can't support multidex	1	ad...@chenzhongjin.cn	71 minutes ago
204605	New	----	----	Snackbar NPE in SwipeDismissBehavior - Support Library v23.2.1	1	kn...@synyx.de	77 minutes ago
204604	New	Small	----	Battery usage by WiFi on Nexus 5 with Android Marshmallow	1	saulphil...@gmail.com	100 minutes ago
204603	New	Small	----	smart lock not working properly	1	VUT...@gmail.com	101 minutes ago
204601	New	----	to...@android.com	Visualization of some settings in module settings are not correct	1	andr...@curasoftgmbh.de	117 minutes ago
204600	NeedsInfo	----	dnaga...@google.com	[developer preview] SDK version is still set to 23, should be 24	1	saraswat...@gmail.com	2 hours ago
204598	New	Medium	jlaurid...@google.com	AVD Manager: Edit Device button is easily missed	1	cherylpo...@google.com	3 hours ago
204597	New	Medium	----	Basic OS Operation: "Undo" Option	1	bizsk...@gmail.com	3 hours ago
204595	New	Medium	j...@google.com	Switching productFlavors uses previous flavor code	1	andhiew...@gmail.com	4 hours ago

Configuration management: Issue tracking

✧ Popularity of issues

Configuration management: Issue tracking

✧ Many tools

Configuration management: Issue tracking

✧ Many tools

Bug List - Mozilla Firefox

Bugzilla of ERICSSON - Bug List

Wed May 23 2012 13:10:39 IST

28 bugs found.

ID	Sev	Pri	OS	Assignee	Status	Resolution	Summary
296	nor	High	Ubun	nayandeep.saha@ericsson.com	RESO	INVA	Actual Start Date and Actual End Date Field in Sub Project Page is Taking Future Date
297	min	High	Ubun	nayandeep.saha@ericsson.com	RESO	INVA	Actual end date can be entered while the status of a sub project is open
299	min	High	Ubun	nayandeep.saha@ericsson.com	RESO	INVA	Sub project is creating duplicate except different Sub project id
310	min	High	Ubun	nayandeep.saha@ericsson.com	RESO	INVA	Defect Created date is taking Future Date
311	min	High	Ubun	nayandeep.saha@ericsson.com	RESO	INVA	Defect Created date is taking Future Date
313	maj	High	Ubun	nayandeep.saha@ericsson.com	RESO	INVA	Time Sheet updation is not working properly
295	min	High	Ubun	gopal.krishna.mishra@eriss...	CLOS	FIXE	The Function Defect Closed By is not working Properly
309	nor	High	Ubun	gopal.krishna.mishra@ericss...	CLOS	FIXE	'Defect Closed By' and 'Defect Closed Date' Both are not a mandatory field
288	min	High	Ubun	nayandeep.saha@ericsson.com	CLOS	FIXE	Function 'Open Link in New Tab' or 'Open Link in New Window' is not working properly
293	nor	High	Ubun	nayandeep.saha@ericsson.com	CLOS	FIXE	All The Date fields in sub project are taking invalid Date entry while Creating or updating
300	nor	High	Ubun	nayandeep.saha@ericsson.com	CLOS	FIXE	All The Date fields in Task are taking invalid Date entry while Creating or updating
302	nor	High	Ubun	nayandeep.saha@ericsson.com	CLOS	FIXE	All The Date fields in Defect logging are taking invalid Date entry while Creating or updating
303	nor	High	Ubun	nayandeep.saha@ericsson.com	CLOS	FIXE	Issue in creation of sub project with respect to planned start date and planned end date
304	nor	High	Ubun	nayandeep.saha@ericsson.com	CLOS	FIXE	Issue in creation of Task with respect to planned start date and planned end date
305	nor	High	Ubun	nayandeep.saha@ericsson.com	CLOS	FIXE	Issue in creation of Defect with respect to Defect start date and defect end date
306	nor	High	Ubun	nayandeep.saha@ericsson.com	CLOS	FIXE	Issue in creation of Report with respect to start date and end date
307	min	High	Ubun	nayandeep.saha@ericsson.com	CLOS	FIXE	If we are putting more than 106 character in Remark field of Defect Logging Page it is giving system error

Configuration management: Issue tracking

- Studies trends of issues in projects
- IC: Issues closed
- IO: Issues opened

Garousi V. (2009) Evidence-Based Insights about Issue Management Processes: An Exploratory Study. In: Wang Q., Garousi V., Madachy R., Pfahl D. (eds) Trustworthy Software Development Processes. ICSP 2009. Lecture Notes in Computer Science, vol 5543. Springer, Berlin, Heidelberg

Configuration management: Issue tracking

- Studies the trends of issues in projects

Implementation issues

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

Host-target development

- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- **Mobile SW development**

Host-target development

- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- **Embedded SW development**

Host-target development

- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- **Embedded SW development in defense**

Host-target development

- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Development platform tools

- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- Project support tools that help you organize the code for different development projects.

Integrated development environments (IDEs)

- Software development tools are often grouped to create an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

Chapter 7 – Design and Implementation

- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open-source development

For student review

Open-source development

- Open-source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- Open-source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open-source systems

- There are many major open-source products, e.g., the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- Other important Open-source products are Java, the Apache web server and the MySQL database management system.

A closer look at Linux

Linux

https://linuxator.wordpress.com/2008/07/22/5-things-you-didnt-know-about-linux-kernel-code-metrics/

A closer look at Linux

Linux

A closer look at Linux: More…

Linux

Open-source issues

When you develop a new software:

- ✧ Should the product that is being developed make use of Open-source components?
- ✧ Should an Open-source approach be used for the software's development?

Lots of places to search for Open-source SW (code)

https://code.openhub.net

BLACKDUCK | Open HUB

Go to

Welcome to the Black Duck Open Hub Code Search, the world's largest, most comprehensive code search engine!

A free public code search engine for 21,372,664,482 lines of open source code.

Check out our FAQ for more info.

Search open source code

Search

Show example searches

Don't see your code? Add your project to The Black Duck Open Hub

https://code.openhub.net/search?s=convert%20date

BLACKDUCK | Op

convert date

Search

Go to Open Hub ho

Filter Code Results

Definitions:

- Method (2,192)
- Function Definition (368)
- Function Declaration (22)
- Field (16)
- Constructor (6)

2 more

Projects:

- elenas_inbox (6,033)
- tosback2-data (4,152)
- RailsDiff (3,612)
- jquery-builder (3,200)
- uClinux-dist (ADI Blackfin...

95 more

Languages:

- Other Languages (244,647)
- JavaScript (136,147)
- C (90,083)
- PHP (88,600)
- Java (80,710)

32 more

File Extensions:

- js (136,138)
- php (88,612)
- c (83,514)
- java (80,710)

Code Results

Results 1 - 10 of about 894,308 results found for 'convert date' in 12

Keep current filters: (Deselect all Filters)

File: convert-date.php Project: matt-cc

```
29    $row = sql_command("select date, time from calendar where id = $x");
30    $date = split(",", $row[date]);
31    $tme = split(",", $row[time]);
32    $tm = split(":", $tme[0]);
33    if ($minutes != "") {
34       $minutes = $tm[1];
```

```
43       if (strcmp($monthnames[$d], $date[0]) == 0 ) $month = $d;
44    }
45    $day = $date[1];
46    $year = $date[2];
47    echo "<tr><td>$month</td><td>$day</td><td>$year</td><td>$hour</td><td>$minutes</td>";
48    $time = mktime($hour, $minutes, 0, $month, $day, $year);
49    echo "<td>$time</td>\n";
50    sql_query("update calendar set datetime = '$time' where id = $x");
```

Code Location: http://matt-cc.go...

File Path: myPHPCalendar/convert-date.php

Copyright: (C) 2000 David van der Bokke

File: convert-date.tag Project: interchange

```
2 convert date to a specified format
3 __END__
4
5 __NAME__ synopsis
6 <row>
```

```
119    Current date
120    </entry>
121    <entry>
122    Text to display if the date value to convert is empty
123    </entry>
124
125 </row>
126 <row>
```

```
157 [convert-date format="%B %e, %Y"]20080701[/convert-date]
158 ]]></programlisting>
159 yields <literal>July 1, 2008</literal>.
160   END
```

Code Sign v2.4.2 | Copyright © 2014 Black Duck Software, Inc., All Rights Reserved.

Lots of places to search for Open-source SW (code)

Open-source business

- More and more product companies are using an Open-source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the Open-source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

Open-source licensing

- A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
  - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an Open-source software license.
  - Some Open-source developers believe that if an Open-source component is used to develop a new system, then that system should also be Open-source.
  - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

License models

- The GNU General Public License (GPL). This is a so-called ‘reciprocal’ license that means that if you use Open-source software that is licensed under the GPL license, then you must make that software Open-source.
- The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to Open-source code without having to publish the source of these components.
- The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to Open-source code. You can include the code in proprietary systems that are sold.

License models

License management

- ✧ Establish a system for maintaining information about open-source components that are downloaded and used.
- ✧ Be aware of the different types of licenses and understand how a component is licensed before it is used.
- ✧ Be aware of evolution pathways for components.
- ✧ Educate people about Open-source.
- ✧ Have auditing systems in place.
- ✧ Participate in the Open-source community.

END OF Chapter 7 – Design and Implementation

- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open-source development

BBM-382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 8

Assoc. Prof. Dr. Ayça KOLUKISA TARHAN
Dr. Tuğba ERDOĞAN

Reference:

Textbook Ch8-Software Testing

Lecture Notes by Vahid Garousi

From the book

- Read it online!

8

Software testing

Objectives

The objective of this chapter is to introduce software testing and software testing processes. When you have read the chapter, you will:

- understand the stages of testing from testing, during development to acceptance testing by system customers;
- have been introduced to techniques that help you choose test cases that are geared to discovering program defects;
- understand test-first development, where you design tests before writing code and run these tests automatically;
- know the important differences between component, system, and release testing and be aware of user testing processes and techniques.

Contents

8.1 Development testing
8.2 Test-driven development
8.3 Release testing
8.4 User testing

Need for SW testing-Examples of SW Failures

ComputerWeekly.com

News | IT Management | Industry Sectors | Technology Topics | Blogs | Multimed

NEWS

Lack of software testing to blame for Terminal 5 fiasco, BA executive tells MPs

Home > Applications > News

Applications

Jaguar recalls 18,000 cars over cruise control software fault

Car system upgrade needed, but no hardware affected

By Leo King | Computerworld UK | Published 10:23, 24 October 11

Examples of Software Failures

FOX BUSINESS
THE POWER TO PROSPER™

WHAT'S HOT Retirement European

Home | Video | News | Investing | Technology | Personal Finance | Home Office | Travel

Honda Recalls 2.5M Vehicles Due to Software Problem

Published August 05, 2011 / Reuters

Print

Email

Honda Motor Co Ltd will recall 2.49 million cars, small SUVs and minivans worldwide, including its popular Accord sedan, to repair a software problem that could damage the automatic transmission.

Toyota software bugs unlike those in flaky PCs

by Brooke Crothers | February 18, 2010 8:39 AM PST

Follow

The electronic issues dogging the Toyota Prius signal that cars are increasingly susceptible to software bugs. Cars, however, are not PCs on wheels and have a different set of problems than that crash-prone computer on your desktop, according to an expert at Edmunds.com.

Brain storming: How will you test the Win Calculator?

- Test plan
- How many inputs (test cases)?
- So MANY possibilities
- How do we know if we have tested the system “enough”?
- How do we know if the outputs are correct!?
- If the system changes, shall we test everything AGAIN?
- Tedious!

Brain storming: How will you test the Notepad?

- Test-case Design: Exploratory
- Test-case Design: Criteria-based
- Test Automation
- Test Evaluation

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- Types of Test Activities
- Types of Testing
- Test Levels/ Test Organization
- Testing can be Manual or Automatic
- What is a good test case?
- Exhaustive (Complete) Testing
- Test Adequacy and Coverage: When can we say testing is enough?
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels

Definitions

- **SW Project Management:** The discipline of managing projects to achieve quality within time constraints and budget.
- **SW Quality (Engineering):** The discipline of specifying, assuring, monitoring, and controlling the quality of software products.
- **SW Verification:** Did we build the product right?
  - Ensures that the product satisfies or matches the original design (low-level checking).
  - This is usually done through white-box testing.
- **SW Validation:** Did we built the right product?
  - Checks that the product design satisfies or fits the intended usage (high-level checking).
  - This is usually done through black-box testing.

Definitions

- **SW Testing:** Techniques to execute programs with the intent of finding as many defects as possible and/or gaining sufficient confidence in the software system under test.
  - ***Program testing can show the presence of bugs, but never their absence"*** **(Edsger Dijkstra, 1930-2002)**

- **SW Inspections:** Techniques aimed at *systematically* verifying software artifacts (design documents, code, tests, etc) with the intent of finding as many defects as possible, as early as possible.

What SW testing is and is NOT

- Testing is to execute a program with the intent of finding as many defects as possible and/or gaining sufficient confidence in the software system under test
- Testing is not cut-and-fit improvement of a user interface or requirements elicitation by prototyping.
- Testing is not the verification of an analysis or design model by syntax checkers or simulation.
- Testing is not the scrutiny of documentation or code by humans in inspections, reviews, or walkthroughs.
- Testing is not static analysis of code using a language translator or code checker.
- Testing is not the use of dynamic analyzers to identify memory leaks or similar problems.
- Testing is not debugging, although successful testing should lead to debugging.

Goals and Limitations of Testing

- Dijkstra, 1972:
  - *"Program testing can be used to show the presence of bugs, but never to show their absence"*
- Thus:
  - No absolute certainty can be gained from testing
  - Testing should be integrated with other SW quality activities, e.g., code reviews and inspections
  - Main goal of testing: demonstrate the software can be depended upon, i.e., *sufficient dependability*
  - We can only gain <u>sufficient</u> confidence in the System Under Test (SUT), but not full confidence!

Basic Testing Definitions

- **Error:** the very root cause. Errors are committed by people
- **Fault:** A fault is the result of a human error in the software documentation, code, etc.
- **Failure:** A failure occurs when a fault executes
- (Many people use the above three terms inter-changeably. We should not do so!)
- **Incident:** Consequences of failures – Failure occurrence may or may not be apparent to the user
- **Defect:** any of the above, usually a fault, a.k.a. bug
  - E.g., people say: “my code has a bug!”
- The fundamental chain of SW dependability threats:

Basic Testing Definitions: a Software Example

- Cube root of 7 in Calculator App:
- The correct expected value: ⟶

- A human SW tester runs the app manually and gets:

- She does not notice the fault in this case.
- Let's discuss the possible errors, faults and failures in this case.

Introduction to Software Testing- Outline

- Definitions
- **Why is SW testing important?**
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- Types of Test Activities
- Types of Testing
- Test Levels/ Test Organization
- Testing can be Manual or Automatic
- What is a good test case?
- Exhaustive (Complete) Testing
- Test Adequacy and Coverage: When can we say testing is enough?
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels

Why is SW testing important?

- A study by (the American) NIST in 2002: The annual national cost of inadequate testing is as much as $59 Billion US!
  - The report is titled: “The Economic Impacts of Inadequate Infrastructure for Software Testing”
  - http://www.nist.gov/public_affairs/releases/n02-10.htm

Software Errors Cost U.S. Economy $59.5 Billion Annually

NIST Assesses Technical Needs of Industry to Improve Software-Testing

www.turkishtestingboard.org

TTB

Turkish Testing Board

Your gateway to global Software Testing community

ISTQB International Software Testing Qualifications Board www.istqb.org

Anasayfa | Hakkımızda | Sertifikasyon | Eğitim | Sertifika Kazananlar | TestIstanbul Konferansları | TSQR | Haberler | İletişim | English

ISTQB Büyümeye Devam Ediyor

TTB'nin yönetiminde yer aldığı dünyanın en büyük Yazılım Test ve Kalite Organizasyonu ISTQB 70 ülkede 240.000'den fazla bilişim profesyonelini sertifikalandırmıştır.

daha fazlası için

Uluslararası Akredite Eğitim Şirketleri

ISTQB Uluslararası Sertifikalı Yazılım Test Uzmanı Eğitimleri ISTQB tarafından akredite edilmiş eğitim kurumları tarafından verilmektedir. ISTQB tarafından onaylanmış "Accredited Training Provider" logolu eğitim kurumları:

tested by keytorc "excellence in software testing"

ISTQB Accredited Training Provider

Dünya Yazılım Test ve Kalite Sektörü İstanbul'da Buluşmaya Devam Ediyor...

TEST ISTANBUL

Future of Testing: New Techniques and Methodologies
23-24 May,2013

Software Test Engineers at Microsoft

Ankara Test Günleri

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- **Dealing with SW Faults**
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- Types of Test Activities
- Types of Testing
- Test Levels/ Test Organization
- Testing can be Manual or Automatic
- What is a good test case?
- Exhaustive (Complete) Testing
- Test Adequacy and Coverage: When can we say testing is enough?
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels

Dealing with SW Faults / Software Quality Engineering (SQE)

Motto from Google!

googletesting.blogspot.com

Google Testing Blog

If it ain't broke, you're not trying hard enough

RPF: Google's Record Playback Framework

Thursday, November 17, 2011 5:26 AM

By Jason Arbon

At GTAC, folks asked how well the Record/Playback (RPF) works in the Browser Integrated Test Environment (BITE). We were originally skeptical ourselves, but figured somebody should try. Here is some anecdotal data and some background on how we started measuring the quality of RPF.

The idea is to just let users use the application in the browser, record their actions, and save them as a javascript to play back as a regression test or repro later. Like most test tools, especially code-generating ones, it works most of the time but its not perfect. Po Hu had an early version working, and decided to test this out on a real world product. Po, the developer of RPF, worked with the chrome web store team to see how an early version would work for them. Why chrome web store? It is a website with lots of data-driven UX, authentication, file upload, and it was changing all the time and breaking existing Selenium scripts: a pretty hard web testing problem, only targeted the chrome browser, and most importantly they were sitting 20 feet from

Search

powered by Google

Tags

Whittaker (57)
GTAC (33)
Copeland (26)
Conferences (14)
TotT (12)
Misko (8)
chromeos (2)
test (2)
testing (2)
Alberto has gone off the deep end (1)
Coverage (1)

Testing Process Overview

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- **What is a Test Case and Test Suite (Test Set)?**
- Types of Test Activities
- Types of Testing
- Test Levels/ Test Organization
- Testing can be Manual or Automatic
- What is a good test case?
- Exhaustive (Complete) Testing
- Test Adequacy and Coverage: When can we say testing is enough?
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels

What is a Test Case and Test Suite (Test Set)?

- A **test case** is set of inputs and the expected outputs for a unit/module/system under test
- A **test suite (test set)** is set of test cases (at least one)
- Without the expected outputs, a test case is NOT complete.
- For Example, for Calculator:

TC#	Unit under test	Input 1	Input 2	Expected output
1	+	5	6	11
2	+	16	0	16
3	/	5	2	2.5
4	/	4	0	Cannot divide
	…			
	So many more test cases!			

Test Case and Test Suite (Test Set)

- The above definitions hold for test cases in any level: unit, integration or system testing
- Example, for Microsoft Word:

TC#	Feature under test	Inputs	Expected output
1	Open	Valid DOCX file	File opened properly
2	Open	Invalid DOCX file	Error message
3	Open	RTF file	File opened properly
	…		
	So many more test cases!		

Test Case and Test Suite (Test Set)

- The definitions hold for test cases in any level, unit, integration or system testing

```
int abs(int x){
    if (x>0)
        return x;
    else
        return -x;
}
```

- Example, for the abs function:

TC#	Unit (method) under test	Inputs	Expected output
1	abs	A +ve, 3	3
2	abs	A –ve, -5	5
	Are two test cases enough? What about 0?		

- The art and science of deriving efficient but effective number of test cases is called “test-case design”.
- “Less is more!” - Less test cases, but maximizing the chance of covering faults.

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- **Types of Test Activities**
- Types of Testing
- Test Levels/ Test Organization
- Testing can be Manual or Automatic
- What is a good test case?
- Exhaustive (Complete) Testing
- Test Adequacy and Coverage: When can we say testing is enough?
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels

Types of Test Activities

- **Test-case Design:** Exploratory (Human-based)
  - Design test values based on domain knowledge of the program and human knowledge of testing, exploratory testing
- **Test-case Design:** Criteria-based
  - Design test values to satisfy coverage criteria, covering all lines of code
- **Test Automation**
  - Embed test values into executable scripts
- **Test Execution**
  - Run tests on the software and record the results
- **Test Evaluation**
  - Evaluate results of testing, report to developers

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- Types of Test Activities
- **Types of Testing**
- Test Levels/ Test Organization
- Testing can be Manual or Automatic
- What is a good test case?
- Exhaustive (Complete) Testing
- Test Adequacy and Coverage: When can we say testing is enough?
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels

Types of Testing

- **Functional Testing: testing functional requirements**
  - Functional testing is checking correct functionality of a system

- **Non-functional Testing: testing non-functional requirements**
  - **Performance testing**
  - Load testing
  - Endurance testing
  - Localization/Internationalization testing
  - Recovery testing
  - Security testing
  - Stress testing
  - Installation testing
  - Compatibility testing
  - Usability testing
  - Scalability testing
  - Reliability testing
  - …ilities!

quality time

Editors: Nancy Eickelmann ▪ Motorola ▪ nancy.eickelmann@motorola.com
Jane Hoffman Hayes ▪ University of Kentucky ▪ hayes@cs.uky.edu

Software's Secret Sauce: The "-ilities"

Jeffrey Voas

If beauty is in the eye of the beholder, then quality must be as well. We live in a world where beauty to one is a complete turnoff to another. Software quality is no different. We have the developer's perspective, the end users' perspective, the testers' perspective, and so forth. So, for instance, if the software complies with the requirements, is that quality software? Many would say yes, but then, what if the software isn't fit for any reasonable purpose? Or is quality software simply software that was developed according to particular standards and regulations?

As you can see, meeting the requirements might be different from being fit for a purpose, which can also be different from complying with rules and regulations on how to develop and deploy the software. Yet we can think of all three perspectives as ways to determine how to judge and assess software quality.

To everything there's a season

when added. To use an analogy, an -ility in an application or system is like a condiment on an entrée: not valuable as a stand-alone item, but capable of significantly enhancing the flavor when added properly. Examples of these flavor-enhancing software -ilities include maintainability, reliability, usability, efficiency, adaptability, availability, security, portability, scalability, safety, fault tolerance, testability, usability, reusability, and sustainability.

So, if we look at the issue of the software meeting its requirements, and if those requirements are solely functional and prescribe no -ilities, then clearly the software can meet the requirements but could be unable to fulfill any reasonable purpose. Furthermore, the -ilities will unlikely be associated in any way with how the software is developed, because development standards rarely drill down low enough to provide the guidance needed to attain particular degrees of any -ility.

Measure for measurement

Note that degree is important. For example, a little salt on a steak usually tastes great.

Test Levels / Test Organization

- Since we can have defects in different level and stages of the SW process, we need **testing at different levels and stages**
  - Unit testing (also called module or component testing)
  - Integration testing
  - System testing
  - Non-functional (e.g., performance) testing
  - Acceptance testing
  - Installation testing
  - …

- One typical test process comes next…

Test Organization (just one typical test process)

Pfleeger, 1998

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- Types of Test Activities
- Types of Testing
- Test Levels/ Test Organization
- **Testing can be Manual or Automated**
- What is a good test case?
- Exhaustive (Complete) Testing
- Test Adequacy and Coverage: When can we say testing is enough?
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels

Testing can be Manual or Automated

Manual Testing

- Clever test-case design
- Interaction with system inspiration for new tests
- Human oracle
- Single test case execution
- Limited data

Automated Testing

- Specs, models, & code used to derive test cases
- Automated oracle needed
- Test execution easily repeatable
- Massive input data possible

Clearly,
we need both
and
both have
advantages/
disadvantages

Manual Testing: Exploratory Software Testing

- **Human tester**
  - using brain, and fingers
  - to create scenarios that
  - will cause software either to fail or to fulfill its mission

- **Take advantage of human cleverness**
  - No scripts: Exploratory Testing
  - Or manual scripted testing
  - Record test findings as you go

The Problem with Manual Testing

- Not very repeatable
  - Either cannot remember exactly how it was last tested (what input value did I use a month ago to fail the software?)
  - Or, too expensive to document detailed procedures. Perhaps you have seen “thick” manual test scripts still some companies use.
- Very effort intensive
  - How long will it take to re-run all the tests?
  - How often will you actually do it?

Automated testing

- Using a test software (tool) to test another software
- Developing test code (script) to test other piece of code (application code)
- How much do you know about it?

Why Automate Tests?

- It gives us rapid feedback (pass/fail)!
- Repeatable
- Much faster than human testers, but needs up-front investment
- How do we know where defects are?
  - We remember where we put them!
  - Defect (fault) localizations

```
int abs(int x){
    if (x>0)
        return x;
    else
        return -x;
}

int test_abs_positive(){
    if (abs(5)==5)
        return 1;
    else
        return 0;
}

int test_abs_negative(){
    if (abs(-4)==4)
        return 1;
    else
        return 0;
}
```

Increasing size and complexity for automated test systems

Note There were more than a million test cases written for Microsoft Office 2007.

- "We designed a test system that probably is as complicated as the system itself."

Jeff Feldstein, Manager, Software Development

Cisco Systems, San Jose, California

Google Tech Talks, November 9, 2005

(a very good talk)

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- Types of Test Activities
- Types of Testing
- Test Levels/ Test Organization
- Testing can be Manual or Automated
- What is a good test case?
- **Exhaustive (Complete) Testing**
- Test Adequacy and Coverage: When can we say testing is enough?
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels

Exhaustive (Complete) Testing

- Exhaustive testing, i.e., testing a software system using all the possible inputs, is most of the time impossible.

- Examples:
  - **A program that computes the factorial function (n!=n.(n-1).(n-2)…1)**
    - **Exhaustive testing = running the program with 0, 1, 2, …, 100, … as an input!**
  - **A compiler (e.g., javac)**
    - **Exhaustive testing = running the (Java) compiler with any possible (Java) program (i.e., source code)**

- Techniques used to reduce the number of inputs
  - Testing criteria group input elements into (equivalence) classes
  - One input is selected in each class (notion of **test data coverage**)

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- Types of Test Activities
- Types of Testing
- Test Levels/ Test Organization
- Testing can be Manual or Automatic
- What is a good test case?
- Exhaustive (Complete) Testing
- **Test Adequacy and Coverage: When can we say testing is enough?**
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels

Test Adequacy: When can we say testing is enough?

Functional (black-box)

- All use cases
- All decision paths
- All menu items
- All states
- All boundary values ...

Structural (white-box)

- All methods
- All statements
- All code branches (if/else)
- …

A test-adequacy criterion leads to **test obligations**.
Test coverage: percentage of obligations that are met.

Test Coverage and Types of Testing

Types of testing: Whether test cases are generated based on:

- **<u>Specification</u> ⇒ Black-Box Testing**
- **<u>Implementation</u> ⇒ White (glass)-Box Testing**

Black- vs. White (glass)-Box Testing

Missing functionality:
Cannot be revealed by white-box techniques

Unexpected functionality:
Cannot be revealed by black-box techniques

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- Types of Test Activities
- Types of Testing
- Test Levels/ Test Organization
- Testing can be Manual or Automatic
- What is a good test case?
- Exhaustive (Complete) Testing
- Test Adequacy and Coverage: When can we say testing is enough?
- **Testing throughout the Software Development Life Cycle**
- Testing Maturity Levels

Testing throughout the SDLC – in Waterfall

- SDLC: Software Development Life Cycle
- Much of the life-cycle development artifacts provides a rich source of test data
- Identifying test requirements and test cases early helps shorten the development time
- They may help reveal faults
- It may also help identify early low testable specifications or design

Testing throughout the Lifecycle – in Rational Unified Process (RUP)

Testing throughout the Lifecycle – in the "V" model (half of this is about testing!)

Testing throughout the Lifecycle – in TDD

(Test-Driven Development)

In TDD, test cases are actually the requirements of the system.

Introduction to Software Testing- Outline

- Definitions
- Why is SW testing important?
- Examples of Software Failures
- Dealing with SW Faults
- Testing Process Overview
- What is a Test Case and Test Suite (Test Set)?
- Types of Test Activities
- Types of Testing
- Test Levels/ Test Organization
- Testing can be Manual or Automatic
- What is a good test case?
- Exhaustive (Complete) Testing
- Test Adequacy and Coverage: When can we say testing is enough?
- Testing throughout the Software Development Life Cycle
- Testing Maturity Levels
- **Testing Roles**

Testing roles

Also, called QA in many companies

Google

SE 550-Software Engineering

Black-box (Functional) Testing

Vahid Garousi

Basic Principles

- Based on the definition of a program's specification, as opposed to its structure (code, interface)
- The notion of complete coverage can also be applied to functional (Black-box) testing
- Rigorous specifications have another benefit, they help Black-box testing, e.g., categorize inputs, derive expected outputs
- In other words, they help test case generation and test oracles

Black-box Testing: Different Techniques

- **Equivalence Class Partitioning**
- **Boundary-Value Analysis**
- **Category-Partition**
- **Decision Tables**
- **Cause-Effect Graphs**

Black-box Testing: Outline

- **Equivalence Class Partitioning**
- **Boundary-Value Analysis**
- **Category-Partition**
- **Decision Tables**
- **Cause-Effect Graphs**

Equivalence Class Testing

- Motivation: we would like to have a sense of complete testing and we would hope to avoid test redundancy
- Equivalence classes: partitions of the input set in which input data have the same effect on the program (e.g., the result in the same output)
- Entire input set is covered: completeness

A SUT's input set

- Disjoint classes: to avoid redundancy
- Test cases: one element of each equivalence class
- But equivalence classes have to be chosen wisely …
- Guessing the likely system behavior might be needed

Companies are using Equivalence Class Testing!

http://proquest.safaribooksonline.com/9780735624252/firstchapter#X2ludGVybmFsX0ZsYXNoUmVhZGVyP3h

This Book

How We Test Software at Microsoft®

Search

Contents

- Table of Contents
- Index
- Errata

Safari

HTML VIEW | FULL SCREEN

Chapter 5. Functional Testing Techniques > Equivalence Class Partitioning

URL | SHOW SEARCH TERMS | ?

5.2. Equivalence Class Partitioning

The first functional testing technique testers should learn to master is *equivalence class partitioning* (ECP). Understanding how to use this technique is important because it provides a foundation for many other techniques and approaches to software testing. The ECP technique is simply a tool that enables the tester to evaluate input or output variables systematically for each parameter in a feature. But to be most effective equivalence class partitioning requires us to perform a comprehensive analysis of the variable data for each parameter in the context of the specific system. So, before designing the ECP tests, we must meticulously decompose and model the variable data for each input and/or output

Figure 5-2. A C# implementation of the Next Date program.

Example of Equivalence Classes

- `NextDate` is a function with three variables: `month`, `day`, `year`. It returns the date of the day after the input date. Limitation: years 1812-2020
- Treatment Summary:
  - If it is not the last day of the month, the next date function will simply increment the day value.
  - At the end of a month, the next day is 1 and the month is incremented.
  - At the end of the year, both the day and the month are reset to 1, and the year incremented.
  - Finally, the problem of leap year makes determining the last day of a month interesting.
- Definition of leap years (http://en.wikipedia.org/wiki/Leap_years): The Gregorian calendar adds a 29th day to February in all years evenly divisible by 4, except for centennial years (those ending in -00) which are not evenly divisible by 400. Thus 1600, 2000 and 2400 are leap years but 1700, 1800, 1900, 2100, 2200 and 2300 are not.
- Only the year 1900 falls in the 1812-2020 period

`NextDate` Equivalence Classes

- M1 = {month | month has 30 days}
- M2 = {month | month has 31 days}
- M3 = {month | month is February (the impact of leap years)}

- D1 = {day | 1<= day <= 28}
- D2 = {day | day = 29}= 29
- D3 = {day | day = 30}=30
- D4 = {day | day = 31}=31

- Y1 = {year | year = 1900}= 1900
- Y2 = {year | 1812 <= year <= 2012 AND (year != 1900) AND (year mod 4 = 0)}
- Y3 = {year | (1812 <= year <= 2012 AND year mod 4 != 0)}

Weak/Strong Equivalence Class Testing

- For an example SUT, suppose there are three input variables from three domains: A, B, C
- $A = A_1 \cup A_2 \cup A_3 \cup \ldots \cup A_m$ where $a_i \in A_i$
- $B = B_1 \cup B_2 \cup B_3 \cup \ldots \cup B_n$ where $b_i \in B_i$
- $C = C_1 \cup C_2 \cup C_3 \cup \ldots \cup C_o$ where $c_i \in C_i$
- Weak Equivalence Class Testing: Choosing one variable value form each equivalence class (one a_i, b_i, and c_i) such that all classes are covered. # of test cases?
  - max (|A|, |B|, |C|)
- Strong Equivalence Class Testing: Is based on the Cartesian product of the partition subsets ($A \times B \times C$), i.e., testing all interactions of all equivalence classes. # of test cases?
  - |A| x |B| x |C|
- Examples next…

Example of Weak Equivalence Class Testing (WECT)

- *For an example SUT, suppose there are three input variables from three domains: A, B, C*
- $A = A_1 \cup A_2 \cup A_3$ *where* $a_i \in A_i$
- $B = B_1 \cup B_2 \cup B_3 \cup B_4$ *where* $b_i \in B_i$
- $C = C_1 \cup C_2$ *where* $c_i \in C_i$
- *Number of WETCs needed=Max size w.r.t. the number of equivalence classes of {A, B, C}*
- *4 WETCs are enough.*

Test Case	A	B	C
WETC1	a1	b1	c1
WETC2	a2	b2	c2
WETC3	a3	b3	c1
WETC4	a1	b4	c2

Example of Strong Equivalence Class Testing (SECT)

- *|A| (number of equivalence classes) = 3*
- *|B| = 4*
- *|C| = 2*

- *# of test cases=3x4x2=24*

Test Case	A	B	C
SETC1	a1	b1	c1
SETC2	a1	b1	c2
SETC3	a1	b2	c1
SETC4	a1	b2	c2
SETC5	a1	b3	c1
SETC6	a1	b3	c2
SETC7	a1	b4	c1
SETC8	a1	b4	c2
SETC9	a2	b1	c1
SETC10	a2	b1	c2
SETC11	a2	b2	c1
SETC12	a2	b2	c2
SETC13	a2	b3	c1
SETC14	a2	b3	c2
SETC15	a2	b4	c1
SETC16	a2	b4	c2
SETC17	a3	b1	c1
SETC18	a3	b1	c2
SETC19	a3	b2	c1
SETC20	a3	b2	c2
SETC21	a3	b3	c1
SETC22	a3	b3	c2
SETC23	a3	b4	c1
SETC24	a3	b4	c2

`NextDate` Equivalence Classes

- M1 = {month | month has 30 days}
- M2 = {month | month has 31 days}
- M3 = {month | month is February (the impact of leap years)}

- D1 = {day | 1<= day <= 28}
- D2 = {day | day = 29}
- D3 = {day | day = 30}
- D4 = {day | day = 31}

- Y1 = {year | year = 1900}
- Y2 = {year | 1812 <= year <= 2012 AND (year != 1900) AND (year mod 4 = 0)}
- Y3 = {year | (1812 <= year <= 2012 AND year mod 4 != 0)}

NextDate

Weak Equivalence Class Testing (WECT) - Test Cases

- Number of WECT test cases=maximum partition size (D)=4

Test Case ID	Month	Day	Year	Output
WETC1	6	14	1900	6/15/1900
WETC2	7	29	1912	7/30/1912
WETC3	2	30	1913	Invalid Input date (not possible)
WETC4	5	31	1900	Invalid Input date

NextDate

Strong Equivalence Class Testing (SECT) - Test Cases

- *Number of SECT test cases= partition size (D) x partition size (M) x partition size (Y) = 3x4x3=36 test cases*

Test Case ID	Month	Day	Year	Expected Output
SE1	6	14	1900	6/15/1900
SE2	6	14	1912	6/15/1912
SE3	6	14	1913	6/15/1913
SE4	6	29	1900	6/30/1900
SE5	6	29	1912	6/30/1912
SE6	6	29	1913	6/30/1913
SE7	6	30	1900	7/1/1900
SE8	6	30	1912	7/1/1912
SE9	6	30	1913	7/1/1913
SE10	6	31	1900	ERROR
SE11	6	31	1912	ERROR
SE12	6	31	1913	ERROR
SE13	7	14	1900	7/15/1900
SE14	7	14	1912	7/15/1912
SE15	7	14	1913	7/15/1913
SE16	7	29	1900	7/30/1900
SE17	7	29	1912	7/30/1912
SE18	7	29	1913	7/30/1913
SE19	7	30	1900	7/31/1900
SE20	7	30	1912	7/31/1912
SE21	7	30	1913	7/31/1913
SE22	7	31	1900	8/1/1900
SE23	7	31	1912	8/1/1912
SE24	7	31	1913	8/1/1913
SE25	2	14	1900	2/15/1900
SE26	2	14	1912	2/15/1912
SE27	2	14	1913	2/15/1913
SE28	2	29	1900	ERROR
SE29	2	29	1912	3/1/1912
SE30	2	29	1913	ERROR
SE31	2	30	1900	ERROR
SE32	2	30	1912	ERROR
SE33	2	30	1913	ERROR
SE34	2	31	1900	ERROR
SE35	2	31	1912	ERROR
SE36	2	31	1913	ERROR

Equivalence Class Testing

`NextDate` Example - Discussion

- If error conditions are a high priority, we should extend strong equivalence class testing to include invalid classes
  - **E.g., year | year < 1812 or year > 2012**
- Equivalence Class Testing is appropriate when input data defined in terms of ranges and sets of discrete values
- We can also consider extra equivalence classes to consider the “invalid” situations, e.g., years before 1812 or after 2012

Exercise

- Use equivalence-class testing to develop an automated black-box (functional) test suite to test the "*Send Email*" feature (use case) of the Google Mail. See the screenshot below.

Part B: Black-box Testing

3- (20 total marks) Suppose you want to develop an automated black-box (functional) test suite to test the "Send Email" feature (use case) of the Google Mail. See the screenshot below.

- (15 marks) Using appropriate reasoning, choose an appropriate black-box test-case design method (e.g. equivalence class, cause-effect graph, decision table, etc.) and use that method to derive test cases for this feature. Show the details of your solution.
- (5 marks) Describe a strategy that you will use to develop an automated test oracle for the assertion/verification phase of your test code.

- Was in exam a few years ago!

FOR STUDENT READING

Black-box Testing: Outline

- Equivalence Class Partitioning
- **Boundary-Value Analysis**
- Category-Partition
- Decision Tables
- Cause-Effect Graphs

Boundary-Value Analysis

Motivations

- In equivalence class testing, we partitioned input domains into equivalence classes, on the assumption that the behavior of the program is “similar” for all input values of a equivalence class
- But the above assumption may not be true in all cases as…
- Some typical programming errors happen to be at the <u>boundary</u> between different equivalence classes
- This is what boundary value testing focuses on
- Simpler but complementary to equivalence class testing

http://proquest.safaribooksonline.com/9780735624252/firstchapter#X2ludGVybmFsX0ZsYXNoUmVhZGVyP3h

Safari Books Online

HTML VIEW | FULL SCREEN

Chapter 5. Functional Testing Techniques > Boundary Value Analysis

URL | SHOW SEARCH TERMS | ?

This Book

How We Test Software at Microsoft®

Search

Contents

- Table of Contents
- Index
- Errata

Copyright
Acknowledgments
Introduction
Pt. I. About Microsoft
Pt. II. About Testing
Ch. 4. A Practical Approach to Test Case Design
Ch. 5. Functional Testing Techniques

5.3. Boundary Value Analysis

Boundary value analysis (BVA) is perhaps the best-known functional test technique. Unfortunately, it is also typically misused because many testers assume it to be relatively easy or trivial. Historical evidence demonstrates that a surprising number of problems occur at boundaries of linear variables, so we must carefully analyze the boundary conditions of linear variable data to avoid overlooking boundary class defects. When BVA is used in conjunction with equivalence class partitioning, the BVA functional technique can be very effective in systematically analyzing the boundary values of linear variables for independent input and output parameters. BVA or boundary testing is especially useful for detecting the following types of errors:

- Incorrect artificial constraints of a data type
- Erroneously assigned relational operators
- Wrapping of data types
- Problems with looping structures
- Off-by-one errors

Boundary-Value Analysis

Basic Ideas

- Assume a function F, with two variables x1 and x2
- (Possibly unstated) boundaries: a <= x1 <= b, c <= x2 <= d
- In some programming languages, strong typing allows the specification of such intervals
- In boundary-value analysis (and testing), the focus is on the boundary of the input space for identifying test cases
- The rationale is that errors tend to occur near extreme values of input variables

Boundary-Value Analysis

Basic Ideas

- Approach: Setting values for input variable at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum.
- Convention for the above notions:
  - min, min+, nom, max-, max
- A usual strategy for all input variables: Holding the values of all but one variable at their nominal values, letting one variable assume its extreme value
- Examples next…

Boundary-Value Analysis

Input Domain of Function F

- *Assume a function F, with two variables x1 and x2, domains X1 and X2*
- *(Possibly unstated) boundaries: a <= x1 <= b, c <= x2 <= d*

Boundary-value Analysis

Test Cases

- Test set ={ <$x1_{nom}$, $x2_{min}$>, <$x1_{nom}$, $x2_{min+}$>, <$x1_{nom}$, $x2_{nom}$>, <$x1_{nom}$, $x2_{max-}$>, <$x1_{nom}$, $x2_{max}$>, <$x1_{min}$, $x2_{nom}$,>, <$x1_{min+}$, $x2_{nom}$,>, <$x1_{max-}$, $x2_{nom}$>, <$x1_{max}$, $x2_{nom}$>}.
- Number of test cases=9

Boundary-Value Analysis

General Case and Limitations

- A function with n variables will require 4n + 1 test cases
- Works well with variables that represent bounded physical quantities
- No consideration of the nature of the function and the meaning of variables
- An elementary technique that is a extendable to robustness testing (next)
  - "Robustness is defined as the degree to which a system operates correctly in the presence of *exceptional inputs* or *stressful environmental conditions*." [IEEE Standard 610.12.1990]
  - The goal of robustness testing is to develop test cases and test environments where a system's robustness can be assessed.

Boundary-Value Analysis: The relationship with Robustness Testing

- In Robustness Testing, We also look at the behavior of the system when the variable extremes are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min-)
- This is an extension of the basic boundary-value testing.
- A function with n variables will require 6n + 1 test cases

Boundary-Value Analysis

Worst Case Testing (WCT)

- Boundary-value makes the common assumption that failures, most of the time, originate from one fault
- What happens when more than one variable has an extreme value?
- The idea comes from electronics in circuit analysis
- Cartesian product of {min, min+, nom, max-, max}
- *Clearly more thorough than boundary-value analysis, but with much more effort: 5^n test cases (n: number of variables)*
- *WCT is a good strategy when physical variables have numerous interactions, and where failure is costly*

Boundary-Value Analysis

Even further: Robust Worst-Case Testing for 2 variables

- 7^n test cases (n: number of variables)

Black-box Testing: Outline

- **Equivalence Class Partitioning**
- **Boundary-Value Analysis**
- **Category-Partition Testing**
- **Decision Tables**
- **Cause-Effect Graphs**

Category-Partition Testing

Steps

- The system is divided into individual “functions” that can be independently tested
- The method identifies the *parameters* of each “function” and, for each parameter, identifies distinct *categories*
- Besides parameters, *environment characteristics,* under which the function operates (characteristics of the system state), can also be considered
- *Categories* are major properties or characteristics for each parameter
- The *categories* are further subdivided into *choices* in the same way as equivalence partitioning is applied (possible “values”)
- A broader idea compared to Equivalence-Class Partitioning. (see the notes area)
- In essence: Category-partition testing integrates equivalence class testing, boundary analysis, and adds a few refinements

Category-Partition Testing

An Example

- Specification: The program prompts the user for a positive integer in the range 1 to 20 and then for a string of characters of that length.
- The program then prompts for a character and returns either the position in the string at which the character was first found or a message indicating that the character was not present in the string.
- The user has the option to search for more characters.

Category-Partition Testing Example

Parameters and Categories

- Parameters: Three
  - integer x (length), the string a, and the character c
- Categories:
  - For the length x, the categories are “in-range” (1-20) or “out-of-range”
  - Categories for the string a: minimal, maximal, intermediate length
  - Categories for character c: character appears at the beginning, middle, end of string, or does not occur in the string

Category-Partition Testing Example

Choices

- For Integer x, out-of-range: 0, 21
- For Integer x, in-range: 1, 2-19, 20
- For String a: 1, 2-19, 20
- For Character c: first, middle, last, does not occur

- Note: sometimes we can have only one choice in each category

Category-Partition Testing Example

Formal Test Specifications with Constraints

Recall the definition of dependent and independent variables from statistics

x (length):

Choices

1)	0	[error (out of range)]
2)	1	[property stringOk, MinLength]
3)	2-19	[property stringOk, MidLength]
4)	20	[property stringOk, MaxLength]
5)	21	[error (out of range)]

independent

a (string):

Choices

1)	Of length 1	[if stringOk and MinLength]
2)	Of length 2-19	[if stringOk and MidLength]
3)	Of length 20	[if stringOk and MaxLength]

dependent

c (character):

Choices

1)	At first position in string	[if stringOk]
2)	At last position in string	[if stringOk and not MinLength]
3)	In middle of string	[if stringOk and not MinLength]
4)	Not in string	[if stringOk]

Category-Partition Testing

Specifying constraints among choices, in General

- *Properties,* and *Selectors* are associated with choices

3 properties (definitions)

Category A

Choices

ChoiceA1 [property X, Y, Z]

ChoiceA2

Category B

A selector

ChoiceB1

ChoiceB2 [if X and Z]

- E.g., in the above abstract example, ChoiceA2 and ChoiceB2 would not be combined into a test frame

Category-Partition Testing Example

Test Frames, Data and Cases

- The number depends on the inter-choice constraint dependencies
- Without constraints: 5*3*4 = 60 test frames/cases (i.e., using SECT)
- We have reduced to 12 using constraints

x (length):

1)	*0*	*[error (out of range)]*
2)	*1*	*[property stringOk, MinLength]*
3)	*2-19*	*[property stringOk, MidLength]*
4)	*20*	*[property stringOk, MaxLength]*
5)	*21*	*[error (out of range)]*

a (string):

1)	*Of length 1*	*[if stringOk and MinLength]*
2)	*Of length 2-19*	*[if stringOk and MidLength]*
3)	*Of length 20*	*[if stringOk and MaxLength]*

c (character):

1)	*At first position in string*	*[if stringOk]*
2)	*At last position in string*	*[if stringOk and not MinLength]*
3)	*In middle of string*	*[if stringOk and not MinLength]*
4)	*Not in string*	*[if stringOk]*

x1	***x = 0***
x2a1c1	***x = 1, a = 'A', c = 'A'***
x2a1c4	***x = 1, a = 'A', c = 'B'***
x3a2c1	***x = 7, a = 'ABCDEFG', c = 'A'***
x3a2c2	***x = 7, a = 'ABCDEFG', c = 'G'***
x3a2c3	***x = 7, a = 'ABCDEFG', c = 'D'***
x3a2c4	***x = 7, a = 'ABCDEFG', c = 'X'***
x4a3c1	***x = 20, a = 'ABCDEFGHIJKLMNOPQRST', c = 'A'***
x4a3c2	***x = 20, a = 'ABCDEFGHIJKLMNOPQRST', c = 'T'***
x4a3c3	***x = 20, a = 'ABCDEFGHIJKLMNOPQRST', c = 'J'***
x4a3c4	***x = 20, a = 'ABCDEFGHIJKLMNOPQRST', c = 'X'***
x5	***x = 21***

Category-Partition Testing

Steps (II)

- The *constraints* operating between choices are then identified, i.e., how the occurrence of one choice can affect the existence of another
  - E.g., in the array sorting example, if Len = 0, then the rest does not matter
- *Test frames* (or test specifications) are then generated which are defined as the allowable combinations of choices in the categories
- Test frames are then converted into *test data*

Criteria Using Choices

- All Combinations (AC): This is what was shown in the previous example, what is typically done when using category-partition. One value for every choice of every parameter must be used with one value of every (possible) choice of every other category.
- Each choice (EC): This is a weaker criterion. One value from each choice for each category must be used at least in one test case.
  - **Should use a "greedy approach"**
  - **How many test cases for our example? 7 TCs**

x1	x = 0
x2a1c1	x = 1, a = 'A', c = 'A'
~~x2a1c4~~	~~x = 1, a = 'A', c = 'B'~~
x3a2c1	x = 7, a = 'ABCDEFG', c = 'A'
~~x3a2c2~~	~~x = 7, a = 'ABCDEFG', c = 'G'~~
~~x3a2c3~~	~~x = 7, a = 'ABCDEFG', c = 'D'~~
~~x3a2c4~~	~~x = 7, a = 'ABCDEFG', c = 'X'~~
~~x4a3c1~~	~~x = 20, a = 'ABCDEFGHIJKLMNOPQRST', c = 'A'~~
x4a3c2	x = 20, a = 'ABCDEFGHIJKLMNOPQRST', c = 'T'
x4a3c3	x = 20, a = 'ABCDEFGHIJKLMNOPQRST', c = 'J'
x4a3c4	x = 20, a = 'ABCDEFGHIJKLMNOPQRST', c = 'X'
x5	x = 21

Conclusions

- Identifying parameters and environments conditions, and categories, heavily relies on the experience of the tester
- Makes testing decisions explicit (e.g., constraints), open for review
- Combines boundary-value analysis, robustness testing, and equivalence-class partitioning
- Once the first step (i.e., specifying the constraints) is completed, the technique is straightforward and can be automated
- The criteria for test-case reduction makes it useful for practical testing
- Source:

P. Ammann and J. Offutt, "Using Formal Methods to Derive Test Frames in Category-Partition Testing," *Safety, Reliability, Fault Tolerance, Concurrency, and Real Time Security: Proc. 9th Ann. Conf. Computer Assurance (COMPASS '94)*, pp. 69–79. Los Alamitos, Calif.: IEEE Computer Soc. Press, 1994.

Black-box Testing: Outline

- **Equivalence Class Partitioning**
- **Boundary-Value Analysis**
- **Category-Partition Testing**
- **Decision Tables**
- **Cause-Effect Graphs**

Decision Tables - Motivations

- To help express *test requirements* in a directly usable form
- Easy to understand and support the systematic derivation of test cases
- To support automated or manual generation of test cases
- A particular response or response subset is to be selected for testing by evaluating many related conditions
- Ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions, e.g., control systems

Decision Tables - An Example

For a program identifying the type of a triangle given its edge (side) sizes

Equilateral Isosceles Scalene

Conditions

Decisions

Actions

c1: a, b, c triangle?	N					Y			
c2: a = b?	–			Y				N	
c3: a = c?	–		Y		N		Y		N
c4: b = c?	–	Y	N	Y	N	Y	N	Y	N
a1: not a triangle?	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

Decision Tables - Structure

c1: a, b, c triangle?	N					Y			
c2: a = b?	–			Y				N	
c3: a = c?	–		Y		N		Y		N
c4: b = c?	–	Y	N	Y	N	Y	N	Y	N
a1: not a triangle?	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

- There is a condition section which lists *conditions* and combinations thereof
- Conditions express relationships among *decision variables*
- There is an Action section which lists *responses* to be produced when corresponding combinations of conditions are true
- A limitation: Resultant actions are determined by the *current* values of decision variables!
- Actions are *independent* of input order and the order in which conditions are evaluated
- Actions may appear more than once but each combination of conditions is unique

Decision Tables - Table Structure

Conditions	c1	True		True	Fals		
	c2	True		False	True		False
	c3	T	F	–	T	F	–
Actions	a1	X	X		X		
	a2	X				X	
	a3		X		X	X	
	a4			X			X

Rule

Decision Tables - An Example

Can also be shown as a conventional Truth Table

c1: a, b, c triangle?	N					Y				
c2: a = b?	–			Y					N	
c3: a = c?	–		Y		N			Y		N
c4: b = c?	–	Y	N	Y	N	Y	N	Y	N	N
a1: not a triangle?	X									
a2: Scalene										X
a3: Isosceles					X		X	X		
a4: Equilateral		X								
a5: Impossible			X	X		X				

conditions											
c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	-	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	-	-	F	T	T	T	T	T	T	T	T
c4: a = b?	-	-	-	T	T	T	T	F	F	F	F
c5: a = c?	-	-	-	T	T	F	F	T	T	F	F
c6: b = c?	-	-	-	T	F	T	F	T	F	T	F
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

Decision Tables - An Example

Test Cases

conditions											
c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	-	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	-	-	F	T	T	T	T	T	T	T	T
c4: a = b?	-	-	-	T	T	T	T	F	F	F	F
c5: a = c?	-	-	-	T	T	F	F	T	T	F	F
c6: b = c?	-	-	-	T	F	T	F	T	F	T	F
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

A test rule

11 test rules…

One test case for each test rule from the corresponding truth table.

Test Case ID	a	b	c	Expected Output
TC1	4	1	2	Not a triangle
TC 2	1	4	2	Not a triangle
TC 3	1	2	4	Not a triangle
TC 4	5	5	5	Equilateral
TC 5	?	?	?	Impossible
TC 6	?	?	?	Impossible
TC 7	2	2	3	Isosceles
TC 8	?	?	?	Impossible
TC 9	2	3	2	Isosceles
TC 10	3	2	2	Isosceles
TC 11	3	4	5	Scalene

Decision Tables - Ideal Usage Conditions

- One of several distinct responses (i.e., actions) is to be selected according to distinct cases of input variables
- These cases can be modeled by mutually-exclusive Boolean expressions on the input variables
- The response to be produced does not depend on the order in which input variables are set or evaluated (e.g., events are received)
- The response does not depend on prior inputs or outputs

Decision Tables - Explosion of Condition Variants

- For n conditions, there may be at most 2^n *variants* (unique combinations of conditions and actions)
- Our triangle Example…

conditions											
c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	-	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	-	-	F	T	T	T	T	T	T	T	T
c4: a = b?	-	-	-	T	T	T	T	F	F	F	F
c5: a = c?	-	-	-	T	T	F	F	T	T	F	F
c6: b = c?	-	-	-	T	F	T	F	T	F	T	F

- But, fortunately, there are usually much fewer *explicit* variants in SW systems…
- "Don't care" values in decision tables help reduce the number of those variants
- "Don't care" can correspond to several cases:
  - The inputs are necessary but have no effect
  - The inputs may be omitted

Black-box Testing: Outline

- **Equivalence Class Partitioning**
- **Pair-wise and n-way testing**
- **Boundary-Value Analysis**
- **Category-Partition Testing**
- **Decision Tables**
- **Cause-Effect Graphs**

Cause-Effect Graphs

Car Insurance Renewal Example

Cause-Effect Graphs

Definition

- Graphical technique that helps derive decision tables
- Aim at supporting interaction with domain experts and the reverse engineering of specifications, for the purpose of testing.
- Identify causes (conditions on inputs, stimuli) and effects (outputs, changes in system state)
- Causes have to be stated in such a way to be either true or false (Boolean expression)
- Can be used to explicitly specify constraints (environmental, external) on causes and effects
- They can help select more “significant” subset of input-output combinations and build smaller decision tables

Cause-Effect Graphs

Structure of

- A node is drawn for each cause and effect
- Nodes placed on opposite side of a sheet
- An edge from a cause to an effect indicates that the cause is a necessary condition for the effect
- If a single effect has two or more causes, the logical relationship of the causes is annotated by symbols for logical *and* (^) and logical *or* ($\vee$) placed between lines
- A cause whose negation is necessary is shown by a logical *not* (~)
- A single cause may be necessary for many effects; a single effect may have many necessary causes.
- Intermediate nodes may be used to simplify the graph and its construction

Cause-Effect Graphs

Notations in

If A then B

AND: If (A and B) then C

OR: If (A or B) then C

NAND: If not (A and B) then C

NOR: If (neither A nor B) then C

NOT: If (not A) then B

From “Cause-Effect Graphs” to “Decision Tables”

Car Insurance Renewal Example

Identical

	Condition Section		Action Section		
Variant	**Claims**	**Age**	**Premium Increase $**	**Send Warning**	**Cancel**
1	**0**	**25-**	**50**	**No**	**No**
2	**0**	**26+**	**25**	**No**	**No**
3	**1**	**25-**	**100**	**Yes**	**No**
4	**1**	**26+**	**50**	**No**	**No**
5	**2 to 4**	**25-**	**400**	**Yes**	**No**
6	**2 to 4**	**26+**	**200**	**Yes**	**No**
7	**5+**	**Any**	**0**	**No**	**Yes**

Cause-Effect Graphs

Additional Constraints

Exactly one of A and B must be true

At most one of A and B may be true
(E should be remembered!)

At least one of A and B must be true

A masks B, i.e., A => NOT B

A requires B, i.e., A => B

Motivations: They are key to minimize test cases (logic minimization), but one must be very careful. They need to be reviewed carefully.

Discussion

- The cause-effect graph can be used to generate all *possible* combinations of causes and checking whether the effect corresponds to the specification
- It provides a test oracle and specifies constraints on outputs (effects), helping detecting wrong system states and action combinations
- If the graph is too large, for each admissible combination of effects, we should find some combinations of causes that cause that combination of effects by tracing back through the graph
- Because of additional constraints on a cause-effect graph, it can be more restrictive than straight decision tables

Basic Testing Definitions: a non-Software Example

IEEE TRANSACTIONS ON RELIABILITY, VOL. 46, NO. 4, 1997 DECEMBER

Defect, Fault, Error, . . . , or Failure?

Behrooz Parhami, *Fellow IEEE*
University of California, Santa Barabara

- An automobile brake system with a weak joint in the brake fluid piping (e.g., caused by a design flaw) is defective.
- Error: design flaw (human error)
- (Potential) Fault: weak joint in the piping
- A careful testing or inspection activity on the automobile can reveal the fault.
- However, the driver does not automatically notice the fault while driving.

Basic Testing Definitions: a non-Software Example

- A failure happens if either during normal driving or a testing activity, the fault reveals itself.
- If the weak-joint breaks down, the brake system then have a failure.
- It could be that the weak-joint breaks down in one break and the other 3 breaks work, the driver or tester may not notice the fault. This will be a hidden fault. A failure has happened, but it is also hidden.
- Incident: the break system fails and an accident happens.

BBM-382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 11

Assist.Prof.Dr. Ayça TARHAN
Dr. Tuğba ERDOĞAN

Chapter 9 – Software Evolution

From the book

- Read it online!

9

Software evolution

Objectives

The objectives of this chapter are to explain why software evolution is an important part of software engineering and to describe software evolution processes. When you have read this chapter, you will:

- understand that change is inevitable if software systems are to remain useful and that software development and evolution may be integrated in a spiral model;
- understand software evolution processes and influences on these processes;
- have learned about different types of software maintenance and the factors that affect maintenance costs; and
- understand how legacy systems can be assessed to decide whether they should be scrapped, maintained, reengineered, or replaced.

Contents

9.1 Evolution processes
9.2 Program evolution dynamics
9.3 Software maintenance
9.4 Legacy system management

Topics covered

- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about software change

Software change

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

- A key problem for all organizations is implementing and managing change to their existing software systems.

Evolution of Android

Evolution of Android

[1] Source: http://www.ohloh.net, http://en.wikipedia.org/wiki/Android_version_history (February 2013)

Evolution of Linux

Linux distro timeline
Version 7.2 by NPL (nonplus@gmail.com)
For the latest version, visit ldd-files.org
Feel free to modify and spread. Mail me for updates, corrections and source (Inkscape) files
Based on "Linea del tiempo Distribuciones Linux" by A. Sandmann (marcelo@cripes.cl)
Additional info: distrowatch.com/wikipedia.org
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
GNU/Linux
Debian
TAMU
MCC Interim
SLS
Slackware
S.u.S.E
Jurix
Yggdrasil
Stampede
Enoch
Beehive
Gentoo
Sorcerer
Source Mage
Vector
Lunar
Libranet
Storm
Astaro
Linuxware
Progeny
Xandros
Corel
Linspire
GuadaLinux
Ubuntu
Xubuntu
Edubuntu
Kubuntu
MEPIS
SimplyMEPIS
Dream Studio Linux
Symphony OS
KNOPPIX
Knoppix
Morphix
LinEx
Yoper
Puppy
Fardus
Kate Linux
KateOS
SLAX
Ministack
Zenwalk
Frugalware
SUSE
SuSE
Sun JDS
openSUSE
RR4 / RR64
Sabayon
Kurotoa
VidaLinux
Arch
CRUX
Rock Linux
Linux From Scratch
dyne:bolic
Ark
ISO
Caldera
SCO
Redmond
Lycoris
Conectiva
Mandrake
Virtual
PCLinuxOS
Mandriva
CentOS
Unitad Linux
Scientific
White Box
SpeedFox
rPath
Foresight
FoX
Ekaaty
Fedora Core
BLAG
PLD
SELinux
EnGarde
Red Flag
Peanut
Yellow Dog
Turbolinux
DLD / Delix
Red Hat
zLinux

Importance of evolution

- Organizations have huge investments in their software systems - they are critical business assets.
- To maintain the value of these assets to the business, they must be changed and updated.
- The majority of the software budget in large companies is devoted to **changing and evolving existing software** rather than developing new software.

Evolution and servicing

- Evolution
  - The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.
- Servicing
  - At this stage, software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in software's environment. No new functionality is added.
- Phase-out
  - The software may still be used but no further changes are made to it.

Timeline Microsoft Office for Windows
Office 2016
Office 2013
Office 2010
Office 2007
Office 2003
Office XP
Office 2000
Office 97
Office 95
Office 4
1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026
(Spent) standard support
(Remaining) standard support
(Spent) extended support
(Remaining) extended support
Updated 2017-04-24

Office
Office XP
Office 2000
Office 97
Office 95
Office 4
1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006
(Spent) standard support
(Remaining) standard support
(Spent) extended support
(Remaining) extended support

Evolution processes

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

Change identification and evolution processes

The software evolution process

Studying software evolution

- Evolution of the Unix code-base
- https://www.youtube.com/watch?v=S7JB0mhrGCQ

Unix versions

2015 IEEE/ACM 38th IEEE International Conference on Software Engineering

The Evolution of C Programming Practices: A Study of the Unix Operating System 1973–2015

Diomidis Spinellis
dds@aueb.gr

Panos Louridas
louridas@aueb.gr

Maria Kechagia
mkechagia@aueb.gr

Department of Management Science and Technology
Athens University of Economics and Business
Patision 76, GR-104 34 Athens, Greece

Table 1: Analysed Unix Releases

Name	Mean File Date	LoC
Research-V3	1973-08-30	5,963
Research-V5	1974-11-27	27,429
Research-V6	1975-06-03	49,630
BSD-1	1977-12-09	55,378
Bell-32V	1978-12-13	163,643
Research-V7	1979-01-31	157,003
BSD-2	1979-05-06	88,048
BSD-3	1979-08-18	269,646
BSD-4	1980-06-22	373,881
BSD-4.1 snap	1981-04-09	447,540
BSD-4.1c 2	1989-01-03	564,883
BSD-4.2	1990-06-23	683,428
386BSD-0.0	1991-05-13	655,191
386BSD-0.1	1991-06-08	768,469
BSD-4.3	1991-12-03	1,117,677
BSD-4.3 Tahoe	1992-06-21	1,315,451
BSD-4.3 Net 2	1993-07-02	1,445,124
FreeBSD 1.0	1993-07-02	3,515,680
FreeBSD 1.1	1993-09-17	1,519,925
FreeBSD 1.1.5	1993-11-16	1,698,792
BSD-4.3 Net 1	1993-12-25	179,292
BSD-4.4 Lite1	1994-01-15	2,436,446
BSD-4.4 Lite2	1994-01-20	2,487,503
BSD-4.4	1994-01-27	2,828,398
BSD-4.3 Reno	1994-02-26	1,217,391
FreeBSD 2.0	1994-10-09	6,152,630
FreeBSD 2.0.5	1995-01-12	2,154,728
FreeBSD 2.1.0	1995-02-19	2,204,647
FreeBSD 2.1.5	1995-05-20	2,254,479
FreeBSD 2.1.6	1995-05-31	2,266,688
FreeBSD 2.2.5	1996-05-05	2,615,395
FreeBSD 2.2.6	1996-08-07	2,679,569
FreeBSD 2.2.7	1996-09-16	2,710,214
FreeBSD 3.0.0	1997-07-04	3,371,194
FreeBSD 3.2.0	1997-08-26	3,527,404
FreeBSD 3.3.0	1998-03-19	3,575,250
FreeBSD 3.4.0	1998-04-22	3,656,716
FreeBSD 3.5.0	1998-05-17	3,684,268
FreeBSD 4.0.0	1999-03-29	4,422,666
FreeBSD 4.1.0	1999-05-30	4,632,772
FreeBSD 4.1.1	1999-06-29	4,689,328
FreeBSD 4.2.0	1999-07-16	4,717,764
FreeBSD 4.3.0	1999-09-27	4,786,370
FreeBSD 5.0.0	2001-11-02	5,434,960
FreeBSD 4.4.0	2002-02-03	4,881,244
FreeBSD 5.1.0	2002-03-22	5,490,790
FreeBSD 4.6.0	2002-07-03	5,050,687
FreeBSD 5.2.0	2002-08-23	5,706,097
FreeBSD 4.7.0	2002-10-05	5,184,702
FreeBSD 5.3.0	2004-10-16	6,093,719
FreeBSD 5.4.0	2005-04-20	6,126,108
FreeBSD 6.0.0	2005-10-09	6,330,668
FreeBSD 7.1.0	2005-12-09	7,401,793
FreeBSD 7.2.0	2006-01-19	7,501,567
FreeBSD 6.1.0	2006-04-30	6,424,602
FreeBSD 6.2.0	2006-11-14	6,530,096
FreeBSD 8.0.0	2007-02-21	8,016,942
FreeBSD 8.1.0	2007-05-08	8,152,027
FreeBSD 8.2.0	2007-07-22	8,358,507
FreeBSD 6.3.0	2007-11-24	6,614,077
FreeBSD 7.0.0	2007-12-22	7,231,515
FreeBSD 9.0.0	2008-09-22	9,230,038
FreeBSD 9.1.0	2009-02-01	9,497,551
FreeBSD 9.2.0	2009-06-09	9,674,294
FreeBSD 9.3.0	2009-10-29	10,048,538
FreeBSD 10.0.0	2010-11-09	10,767,581

Studying software evolution

- Visualizing the commit history of the Eclipse IDE project
- https://vimeo.com/1130828

Studying software evolution

- At each time-step (usually a month) developers are clustered by the files they modify. A histogram at the bottom shows the volume and type of file committed.
- http://www.michaelogawa.com/research/storylines/

Studying software evolution

- For one single SW engineer:

Change implementation

Change implementation

- Iteration of the development process where revisions to system are designed, implemented and tested.
- A critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.
  - During the program understanding phase, you have to understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.

Urgent change requests

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

(Emergency repair process)

Topics covered

- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about software change

Program evolution dynamics

- “Software systems are grown, not built”.

Program evolution dynamics

- **Types of software systems:**

S-Type	Conforms to an exact specification. **Specification based** → mathematical criteria.
P-Type	Consistent with a single external paradigm. **Paradigmatic domain** → stable assumptions. v.g. Unicode, Physics simulation
E-Type	Actively used and embedded in the real world. **Evolutionary systems** → constantly adapting to change.

Program evolution dynamics

- *Program evolution dynamics* is the study of the processes of system change.
- After several major empirical studies, Lehman and Belady proposed that there were a number of 'laws' which applied to all systems as they evolved.
- There are sensible observations rather than laws. They are applicable to large systems developed by large organisations.

1060 PROCEEDINGS OF THE IEEE, VOL. 68, NO. 9, SEPTEMBER 1980

Programs, Life Cycles, and Laws of Software Evolution

MEIR M. LEHMAN, SENIOR MEMBER, IEEE

Lehman's Laws of Software Evolution

Law	Description
Continuing change	A program that is used in a real-world environment must necessarily change, or else become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors is approximately invariant for each system release.
Organizational stability	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.

Lehman's Laws of Software Evolution

Law	Description
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will decline unless they are modified to reflect changes in their operational environment.
Feedback system	Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Exercise

- Discuss Lehman's laws of software evolution for a typical banking mobile app

- Provide detailed examples when needed

Topics covered

- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about software change

Software maintenance

- Modifying a program after it has been put into use.
- The term is mostly used for changing custom software. Generic software products are said to evolve to create new versions.
- Maintenance does not normally involve major changes to the system's architecture.
- Changes are implemented by modifying existing components and adding new components to the system.

Types of maintenance

- Adaptive –
  - modifying the system to cope with changes in the software environment (DBMS, OS)
- Perfective –
  - implementing new or changed user requirements which concern functional enhancements to the software
- Corrective –
  - diagnosing and fixing errors, possibly ones found by users
- Preventive –
  - increasing software maintainability or reliability to prevent problems in the future

Figure 9.8 Maintenance effort distribution

Maintenance costs

- Usually greater than development costs (2* to 100* depending on the application).
- Affected by both technical and non-technical factors.
- Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- "Ageing software" can have high support costs (e.g. old languages, compilers etc.).

Figure 9.9 Development and maintenance costs

Maintenance cost factors

- Team stability
  - Maintenance costs are reduced if the same staff are involved with them for some time.
- Contractual responsibility
  - The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
- Staff skills
  - Maintenance staff are often inexperienced and have limited domain knowledge.
- Program age and structure
  - As programs age, their structure is degraded and they become harder to understand and change.

Maintenance prediction

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

Maintainability Index (MI) in Visual Studio

Code Metrics Results

Filter: None Min: Max:

Hierarchy	Maintainabilit...	Cyclomatic Co...	Depth of Inheri...	Class Coupling	Lines of Code
CodeMetricsSampleApp (Debug)	87	55	2	7	61
CodeMetricsSampleApp	87	55	2	7	61
BaseClass	97	7	1	2	7
IBase	100	6	0	0	0
Program	65	42	2	5	54
Go() : void	89	1		1	2
Main() : void	41	25		2	33
Method1() : void	57	10		2	11
Method2() : void	100	1		0	1
Method3() : void	74	2		1	4
Method4() : void	89	1		0	1
Method5() : void	98	1		1	1

Maintainable Method

The Maintainability Index rates the relative ease of maintaining code. This index is a value between 0 and 100 based on the number of lines of source code, its Halstead Volume, and its Cyclomatic Complexity. A rating in the following range indicates that the code has good maintainability: 60 - 100.

Maintainability Index : 100, **Lines of Code** : 0, **Halstead Volume** : 0, **Cyclomatic Complexity** : 1

```
1 reference | maintainability 100
21  private static void MyMethod()
22  {
23
24  }
25
```

Maintainability Index (MI) in Visual Studio

Maintenance prediction

Change prediction

- Predicting the number of changes requires and understanding of the relationships between a system and its environment.
- Tightly coupled systems require changes whenever the environment is changed.
- Factors influencing this relationship are
  - Number and complexity of system interfaces
  - Number of inherently volatile system requirements
  - The business processes where the system is used

Complexity metrics

- Predictions of maintainability can be made by assessing the complexity of system components.
- Studies have shown that **most maintenance effort is spent on a relatively small number of system components**.
- Complexity depends on
  - Complexity of control structures (if, then, else, loops)
  - Complexity of data structures (objects)
  - Object, method (procedure) and module size

Process metrics

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

System re-engineering / refactoring

- Re-structuring or re-writing part or all of a legacy system without changing its functionality.
- Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.
- Also called "refactoring"

Advantages of reengineering / refactoring

- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.

The reengineering process

Reengineering process activities

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

Figure 9.12 Reengineering approaches

Reengineering cost factors

- The quality of the software to be reengineered.
- The tool support available for reengineering.
- The extent of the data conversion which is required.
- The availability of expert staff for reengineering.
  - This can be a problem with old systems based on technology that is no longer widely used.

Preventative maintenance by refactoring

- Refactoring is the process of making improvements to a program to slow down degradation through change.
  - You can think of refactoring as 'preventative maintenance' that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
- When you refactor a program, you should not add functionality but rather concentrate on program improvement.

Refactoring and reengineering

- Re-engineering takes place after a system has been maintained for some time and maintenance costs are increasing. You use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable.
- Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.
- One of the main reasons to do refactoring is to identify and remove 'Bad smells' in program code (next)

Refactoring

- A very active discussion among expert programmers

Refactoring

- In Turkey

www.kurumsaljava.com/yazilim-metotlari/refactoring/

Kurumsal Java

Java Enterprise Architecture

ANA SAYFA | HAKKIMDA | İÇERİK | İLETİŞİM | KOD KATA | MEDYA | YAZILIM METO

Refactoring

Refactoring Metotları

- Kataların Anası
- Parametrik Konstruktör (Parameterize Constructor)
- Yeni Sınıf Oluşturma (Extract Class)
- Koşullu Mantığın Komut İle Değiştirilmesi (Replace Conditional Dispatcher With Command)
- Metodu Metot Nesnesine Dönüştürme (Replace Method with Method Object)
- Refactoring Oturumu (Video)
- Geçici Değişkenlerin Sorguya Dönüştürülmesi (Replace Temp with Query)
- Parametrele Değer Atamasının Kaldırılması (Remove Assignments to Parameters)
- Yeni Metot Oluşturma (Extract Method)
- Koşulları Parçalarına Ayırma (Reverse Conditional)

Paylaş

‘Bad smells’ in program code

- Duplicate code
  - The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
- Long methods
  - If a method is too long, it should be redesigned as a number of shorter methods.
- Switch (case) statements
  - These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.

‘Bad smells’ in program code

- Data clumping
  - Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.
- Speculative generality
  - This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Bad smells

https://en.wikipedia.org/wiki/Code_smell

Not logged in Talk Contributions

Article Talk Read Edit View history Search

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item

Code smell

From Wikipedia, the free encyclopedia

Code smell, also known as **bad smell**, in computer programming code, refers to any symptom in the source that possibly indicates a deeper problem.[1] According to Martin Fowler,[2] "a code smell is a surface indication corresponds to a deeper problem in the system". Another way to look at smells is with respect to principles an are certain structures in the code that indicate violation of fundamental design principles and negatively impa Code smells are usually not bugs—they are not technically incorrect and do not currently prevent the progran Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of the future. Bad code smells are an important reason for technical debt.[1] Robert C. Martin calls a list of code s system" for software craftsmanship.[4]

Often the deeper problem hinted by a code smell can be uncovered when the code is subjected to a short fe where it is refactored in small, controlled steps, and the resulting design is examined to see if there are any fu that indicate the need of more refactoring. From the point of view of a programmer charged with performing re smells are heuristics to indicate when to refactor, and what specific refactoring techniques to use. Thus, a cod for refactoring.

A 2015 empirical study[1] of half a million code commits to 200 open source software projects found that

- most bad smells affecting a piece of code are already present since its creation, rather than being introdu evolutionary code changes
- although code smells are often introduced while adding new features and enhancing existing ones, refact also add bad smells

Topics covered

- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about software change

Legacy systems

- What is a legacy software? A system built many years ago, e.g., could be from 1950's

Legacy systems

Some Fun with Windows 95

- https://www.youtube.com/watch?v=8ucCxtgN6sc

Legacy systems: Not too old, but should be maintained!

Legacy system management

- Organisations that rely on legacy systems must choose a strategy for evolving these systems
  - Scrap the system completely and modify business processes so that it is no longer required;
  - Continue maintaining the system;
  - Transform the system by re-engineering to improve its maintainability;
  - Replace the system with a new system.
- The strategy chosen should depend on the system quality and its business value.

Deciding when to reengineer a legacy system

Figure 9.13 An example of a legacy system assessment

Legacy system categories

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

Business value assessment

- Assessment should take different viewpoints into account
  - System end-users;
  - Business customers;
  - Line managers;
  - IT managers;
  - Senior managers.
- Interview different stakeholders and collate results.

Issues in business value assessment

- The use of the system
  - If systems are only used occasionally or by a small number of people, they may have a low business value.
- The business processes that are supported
  - A system may have a low business value if it forces the use of inefficient business processes.
- System dependability
  - If a system is not dependable and the problems directly affect business customers, the system has a low business value.
- The system outputs
  - If the business depends on system outputs, then the system has a high business value.

System quality assessment

- Business process assessment
  - How well does the business process support the current goals of the business?
- Environment assessment
  - How effective is the system's environment and how expensive is it to maintain?
- Application assessment
  - What is the quality of the application software system?

Business process assessment

- Use a viewpoint-oriented approach and seek answers from system stakeholders
  - Is there a defined process model and is it followed?
  - Do different parts of the organisation use different processes for the same function?
  - How has the process been adapted?
  - What are the relationships with other business processes and are these necessary?
  - Is the process effectively supported by the legacy application software?

- Example - a travel ordering system may have a low business value because of the widespread use of web-based ordering.

Factors used in environment assessment

Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?

Factors used in environment assessment

Factor	Questions
Support requirements	What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licenses? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system? Is hardware emulation required?

Factors used in application assessment

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?

Factors used in application assessment

Factor	Questions
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

System measurement

- You may collect quantitative data to make an assessment of the quality of the application system
  - The number of system change requests;
  - The number of different user interfaces used by the system;
  - The volume of data used by the system.

FOR STUDENT READING

Key points

- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.
- Lehman's laws, such as the notion that change is continuous, describe a number of insights derived from long-term studies of system evolution.

Key points

- There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.
- Software re-engineering is concerned with re-structuring and re-documenting software to make it easier to understand and change.
- Refactoring, making program changes that preserve functionality, is a form of preventative maintenance.
- The business value of a legacy system and the quality of the application should be assessed to help decide if a system should be replaced, transformed or maintained.

Program evolution dynamics

- **Types of software systems:**

S-Type	Conforms to an exact specification. **Specification based** → mathematical criteria.

Fig. 1. *S*-programs.

Program evolution dynamics

- **Types of software systems:**

P-Type

Consistent with a single external paradigm.
Paradigmatic domain → stable assumptions.
v.g. Unicode, Physics simulation

Fig. 2. *P*-programs.

Program evolution dynamics

- **Types of software systems:**

E-Type

Actively used and embedded in the real world.
Evolutionary systems → constantly adapting to change.

Fig. 4. *E*-programs.

Metrics used

2015 IEEE/ACM 38th IEEE International Conference on Software Engineering

The Evolution of C Programming Practices: A Study of the Unix Operating System 1973–2015

Dionisis Spinellis
dds@aueb.gr

Panos Louridas
louridas@aueb.gr

Maria Kechagia
mkechagia@aueb.gr

Department of Management Science and Technology
Athens University of Economics and Business
Patision 76, GR-104 34 Athens, Greece

Table 2: Derived Aggregate Metrics, Factors Affecting them, and Hypotheses they Affect

Name	Description	Language	Ergonomics	Guidelines	CPU & Storage	Conventions	Tools	H1	H2	H3	H4	H5	H6	H7
FILINE	Mean file length (lines)			↵	↵			↗						
FISTMT	Mean file functionality (statements)			↵	↵			↗						
LICHAR	Mean line length (characters)		↵	↵				↗						
FULINE	Mean function length (lines)		↵	↵				↗					↗	
IDCHAR	Mean identifier length		↵	↵			↵	↗						
STNEST	Mean statement nesting		↵	↵	↵		↵						↗	
DSTATIC	Internally visible declaration density	↵		↵			↵		↗					
DCONST	`const` keyword density	↵		↵			↵			↗				
DENUM	`enum` keyword density	↵		↵			↵			↗				
DUNS	`unsigned` keyword density	↵		↵						↗				
DINL	`inline` keyword density	↵					↵			↗				
DNOAL	`noalias` keyword density	↵					↵			↗				
DSIGN	`signed` keyword density	↵					↵			↗				
DREG	`register` keyword density	↵					↵				↗			
DREST	`restrict` keyword density	↵					↵			↗				
DVOID	`void` keyword density	↵					↵			↗				
DVOL	`volatile` keyword density	↵					↵			↗				
DGOTO	`goto` keyword density			↵									↗	
INDEV	Indentation spaces standard deviation			↵		↵	↵					↗		
FMINC	Formatting inconsistency			↵		↵	↵					↗		
INSPC	Mean indentation spaces		↵	↵		↵								↗
DSTMT	Statement density		↵	↵		↵								↗
DCMCHAR	Comment character density		↵	↵										↗
CMCHAR	Mean comment size		↵	↵										↗
DCM	Comment density		↵	↵										↗
DCPIF	C preprocessor conditional statement density			↵									↗	
DCPINC	C preprocessor include statement density			↵	↵		↵		↗					
DCPSTMT	C preprocessor non-include statement density	↵		↵	↵		↵						↗	
DKLUDGE	Kludge word density					↵	↵							↗

Metrics used

2015 IEEE/ACM 18th IEEE International Conference on Software Engineering

The Evolution of C Programming Practices: A Study of the Unix Operating System 1973–2015

Diomidis Spinellis dds@aueb.gr
Panos Louridas louridas@aueb.gr
Maria Kechagia mkechagia@aueb.gr
Department of Management Science and Technology
Athens University of Economics and Business
Patision 76, GR-104 34 Athens, Greece

FILINE 0.91

FISTMT 0.90

FMINC 0.81

INDEV 0.89

INSPC 0.77

LICHAR 0.93

Name	Description
FILINE	Mean file length (lines)
FISTMT	Mean file functionality (statements)
LICHAR	Mean line length (characters)
FULINE	Mean function length (lines)
IDCHAR	Mean identifier length
STNEST	Mean statement nesting
DSTATIC	Internally visible declaration density
DCONST	`const` keyword density
DENUM	`enum` keyword density
DUNS	`unsigned` keyword density
DINL	`inline` keyword density
DNOAL	`noalias` keyword density
DSIGN	`signed` keyword density
DREG	`register` keyword density
DREST	`restrict` keyword density
DVOID	`void` keyword density
DVOL	`volatile` keyword density
DGOTO	`goto` keyword density
INDEV	Indentation spaces standard deviation
FMINC	Formatting inconsistency
INSPC	Mean indentation spaces
DSTMT	Statement density
DCMCHAR	Comment character density
CMCHAR	Mean comment size
DCM	Comment density
DCPIF	C preprocessor conditional statement density
DCPINC	C preprocessor include statement density
DCPSTMT	C preprocessor non-include statement density
DKLUDGE	Kludge word density

Change is inevitable

- The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!
- Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- Systems MUST be changed if they are to remain useful in an environment.

Lehman's Laws of Software Evolution

1060 PROCEEDINGS OF THE IEEE, VOL. 68, NO. 9, SEPTEMBER 1980

Programs, Life Cycles, and Laws of Software Evolution

MEIR M. LEHMAN, SENIOR MEMBER, IEEE

BBM-382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 10

Assoc. Prof. Dr. Ayça KOLUKISA TARHAN
Dr. Tuğba ERDOĞAN

CHAPTER 25 – CONFIGURATION MANAGEMENT

What is Cofiguration Management (CM)?

- Roots in Latin:
  - *"com" (together)* & *"figurare" (stucture)*
- ISO 10007-2003: QMS -- Guidelines for CM
  - *"interrelated functional and physical characteristics of a product defined in product configuration information"*

Why do we need CM?

- Because software changes frequently, systems, can be thought of as a set of versions, each of which has to be maintained and managed.
  - Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems.
- Configuration management (CM) is concerned with the policies, processes 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.

Software Versions

CM Activities

1) 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 if the changes should be implemented.

2) 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.

3) SYSTEM BUILDING

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

4) RELEASE MANAGEMENT

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

CM Activities

Configuration Identification

CI	ID	Rev
LCP s/w	5467	3.9
LCP db	456	3.2
LCP display	374	4.1
RTU f/w	123	1.3
RTU cfg	9283	1.6
RTU logic	2837	1.9
IBP logic	8263	2.9

CM Terminology – SCI and Version

- Software Configuration Item (SCI)
  - Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control.
    - There are often different versions of a configuration item.
    - Configuration items have a unique name.
- Version
  - An instance of a configuration item that differs, in some way, from other instances of that item.
    - Versions always have a unique identifier, which is often composed of the configuration item name plus a version number.

CM Terminology – Baseline and Codeline

- Baseline
  - A baseline is a collection of component versions that make up a system.
    - Baselines are controlled, which means that the versions of the components making up the system cannot be changed.
    - This means that it should always be possible to recreate a baseline from its constituent components.
- Release
  - A version of a system that has been released to customers (or other users in an organization) for use.

More on CM terminology

- **Version**

  Specific version of a configuration item

  ECR_BRD V0.1.doc

  ECR_BRD V0.2.doc

- **Baseline**

  A specification or product that has been formally reviewed and agreed to by responsible management, that thereafter serves as the basis for further development, and can be changed only through formal change control procedures

  ECR_BRD V1.0.doc

  ECR_BRD V2.0.doc

More on CM terminology

- Example: Releases of Open Source Java Build Tools

An Open Source Example

(1) 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.

(1) Change Management

Change Request Form

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

Project Change Request Form

Change request number:	Requested by:	Date requested:

Change Request Description

Change description:

Business or technical justification for change request:

Priority:
☐ Top ☐ High ☐ Medium ☐ Low

Impact of not making the change:

Change Impact Analysis

Impact on project requirements:

Impact on project risk:

Impact on project schedule:

Impact on project budget projection:

Impact on project configuration:

Alternatives:

Recommendation:

Change Request Resolution

Change request decision: ☐ Approved ☐ On hold ☐ Denied	Decision date:

Decision made by:
☐ Project manager ☐ Project sponsor ☐ Executive sponsor ☐ Other

Reason for decision:

Change Request Tracking (updates to project baseline)

Requirements document	Yes	N/A	By:	Date:	Comments:
Schedule (WBS)	Yes	N/A	By:	Date:	Comments:
Risk management plan	Yes	N/A	By:	Date:	Comments:
Configuration management plan	Yes	N/A	By:	Date:	Comments:
Communication plan	Yes	N/A	By:	Date:	Comments:

Example: Change Request in Github

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

base: firstBranch ··· compare: master ✓ Able to merge. These branches can be automatically merged.

firstBranch needs updates

Write Preview

Leave a comment

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Styling with Markdown is supported

Create pull request

Reviewers

No reviews—requ

Assignees

tekinonlayn

Labels

None yet

Projects

None yet

Milestone

No milestone

(2) 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.

Codelines and Baselines

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.

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.
- Independent development
  - It 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.

Version Management Systems

Storage Management

Check-in and check-out from a version repository

Codeline Branches and Merges

Example: Branching and Merging in Github

- Branching

Example: Branching and Merging in Github

- Merging

(3) 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.

(3) System Building

Development, build, and target platforms

File identification

- Modification timestamps
  - The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.
- Source code checksums
  - The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by 1 character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.

Continuous integration

Daily building

- The development organization sets a delivery time (say 2 p.m.) for system components.
  - If developers have new versions of the components that they are writing, they must deliver them by that time.
  - A new version of the system is built from these components by compiling and linking them to form a complete system.
  - This system is then delivered to the testing team, which carries out a set of predefined system tests
  - Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

(4) 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 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.

Release components

- As well as 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.

FOR STUDENT REVIEW…

CM terminology

Term	Explanation
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Configuration control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier, which is often composed of the configuration item name plus a version number.
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it should always be possible to recreate a baseline from its constituent components.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.

CM terminology

Term	Explanation
Mainline	A sequence of baselines representing different versions of a system.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.

A partially completed change request form (a)

Change Request Form

Project: SICSA/AppProcessing **Number:** 23/02
Change requester: I. Sommerville **Date:** 20/01/09
Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek **Analysis date:** 25/01/09
Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

A partially completed change request form (b)

Change Request Form

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium
Change implementation:
Estimated effort: 2 hours
Date to SGA app. team: 28/01/09 **CCB decision date:** 30/01/09
Decision: Accept change. Change to be implemented in Release 1.2
Change implementor: **Date of change:**
Date submitted to QA: **QA decision:**
Date submitted to CM:
Comments:

Derivation history

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2009
//
// © St Andrews University 2009
//
// Modification history
// Version Modifier   Date        Change       Reason
// 1.0     J. Jones   11/11/2009  Add header   Submitted to CM
// 1.1     R. Looek   13/11/2009  New field    Change req. R07/02
```

Change management and agile methods

- In some agile methods, customers are directly involved in change management.
- The propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
- Changes to improve the software improvement are decided by the programmers working on the system.
- Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

Codeline branches

- 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.

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.

Build system functionality

- Build script generation
- Version management system integration
- Minimal re-compilation
- Executable system creation
- Test automation
- Reporting
- Documentation generation

Minimizing recompilation

- Tools to support system building are usually designed to minimize the amount of compilation that is required.
- They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- A unique signature identifies each source and object code version and is changed when the source code is edited.
- By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

Timestamps vs checksums

- Timestamps
  - Because source and object files are linked by name rather than an explicit source file signature, it is not usually possible to build different versions of a source code component into the same directory at the same time, as these would generate object files with the same name.
- Checksums
  - When you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible and different versions of a component may be compiled at the same time.

Agile building

- Check out the mainline system from the version management system into the developer's private workspace.
- Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.
- Make the changes to the system components.
- Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

Agile building

- Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.
- Build the system on the build server and run the tests. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
- If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

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.

Factors influencing system release planning

Factor	Description
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Lehman's fifth law (see Chapter 9)	This 'law' suggests that if you add a lot of new functionality to a system; you will also introduce bugs that will limit the amount of functionality that may be included in the next release. Therefore, a system release with significant new functionality may have to be followed by a release that focuses on repairing problems and improving performance.

Factors influencing system release planning

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Customer change proposals	For custom systems, customers may have made and paid for a specific set of system change proposals, and they expect a system release as soon as these have been implemented.

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.

Key points

- 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.

BBM-382 – SOFTWARE ENGINEERING

SPRING 2020

Lecture 11

Assoc. Prof. Dr. Ayça KOLUKISA TARHAN
Dr. Tuğba ERDOĞAN

CHAPTER 24 – QUALITY MANAGEMENT

Topics covered

- Software quality
- Software standards
- Reviews and inspections
- Software measurement and metrics

Software Quality Management

- What is Software Quality?

- How is Software Quality managed?

What is Sofware Quality?

How is Software Quality managed?

Quality of Measurement Results

Software Enginnering Culture & Ethics

Software Process Management

Process Automation

Software Products Measurement

Software Quality Assurance

Software Quality Management Techniques

Process Assessment Methods

Process Measurement Techniques

Quality Improvement

Reviews and Audits

Verification & Validation

Software Life Cycle Models

Process Infrastucture

Process Assessment Models

Value and Cost of Quality

Notations for Process Definition

Software Quality Requirements

Software Life Cycle Processes

Software Quality Measurement

Process Adaptation

Defect Characterization

Models for Process Implementation and Change

Process Measurement

Models and Quality Characteritics

Software Information Models

Quality of Professional Software

- When we talk about the quality of professional software, we have to take into account that the software is used and changed by people apart from its developers.
  - Quality is therefore not just concerned with what the software does; it has to include the software's behavior while it is executing and the structure and organization of the system programs and associated documentation.
- This is so-called quality or non-functional software attributes.
  - Examples of these attributes are the *software's response time to a user query* and the *understandability of the program code*.
  - The specific set of attributes that you might expect from a software system obviously depends on its application domain.
    - E.g., a banking system must be secure, an interactive game must be responsive, a telephone switching system must be reliable, and so on.

Quality – Some Definitions

- The word "quality" is derived from the latin "qualitas", meaning "of what"
  - Its use goes back to antiquity
  - The morfem "-tas" was used to create "nature" or "character"

- Conformance to requirements (Crosby, 1979)
  - Producer perspective

- Fitness for use (Juran, 1951)
  - Customer-oriented view

- The degree to which a set of inherent characteristics fulfills the requirements, i.e. needs or expectations that are stated, generally implied or obligatory. (ISO 9000)

Software Quality is a bit Different (Gillies)

- Software has no physical existence
- The lack of knowledge of client needs at the start
- The change of client needs over time
- The rapid rate of change in both hardware and software
- The high expectations of customers, particularly with respect to adaptability
- No distinction between ‘design quality’ and ‘manufacturing quality’

Software quality management

- Concerned with ensuring that the required level of quality is achieved in a software product.

- Quality management provides an independent check on the software development process.
  - It checks project deliverables to ensure they are consistent with organizational standards and goals

- The quality team should be independent from the development team so that they can take an objective view of the software.
  - This allows them to report on software quality without being influenced by software development issues.

Quality management and software development

Software Quality Management

- **Software quality management** is a management process, the goal of which is to develop and manage the quality of a software to make sure the product satisfies the user. (Wikipedia)

  - The goals of SQM (software quality management) is to make sure a product is in good levels of quality and meets the regulations and standards the customer needs from the product.

Product-based

Process-based

Software quality management

- Three principal concerns:
  - At the organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software.
  - At the project level, quality management involves the application of specific quality processes and checking that these planned processes have been followed.
  - At the project level, quality management is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

Quality planning

- A quality plan sets out the desired product qualities and how these are assessed and defines the most significant quality attributes.
- The quality plan should define the quality assessment process.
- It should set out which organisational standards should be applied and, where necessary, define new standards to be used.

- Quality plan structure:
  - Product introduction; Product plans; Process descriptions; Quality goals; Risks and risk management.
- Quality plans should be short, succinct documents
  - If they are too long, no-one will read them.

Product-based Quality:

Software quality attributes

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

Quality conflicts

- It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance.
- The quality plan should therefore define the most important quality attributes for the software that is being developed.
- The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

Example Scenario (Weigers)

- *"Hi, Phil, this is Maria again. I have a question about the new employee system you programmed. As you know, this system runs on our mainframe, and every department has to pay for its disk storage and CPU processing charges every month. It looks like the new system's files are using about twice as much disk space as the old system did. Even worse, the CPU charges are nearly three times what they used to be for a session. Can you tell me what's going on, please?"*

- *"Sure, Maria," said Phil. "Remember that you wanted this system to store much more data about each employee than the old system did, so naturally the database is much larger. Therefore, you're paying more for disk space usage each month. Also, you and the other product champions requested that the new system be much easier to use than the old one, so we designed that nice graphical user interface. However, the GUI consumes a lot more computer horsepower than the old system's simple character-mode display. So that's why your per-session processing charges are so much higher. The new system's a lot easier to use, isn't it?"*

Elaboration on Example Scenario

- Users naturally focus on specifying their functional requirements, but there is more to software success than just delivering the right functionality.

- Users also have the expectations about how well the product will work:
  - How easy it is to use,
  - How quickly it runs,
  - How often it fails,
  - How it handles exception conditions,
  - Etc.

Elaboration on Example Scenario (cont'd)

- Customers generally do not present their quality expectations explicitly.

- Questions that explore the customer's implicit expectations can lead to quality goal statements and design criteria that help the developers create a fully satisfactory product.

Difficulties with Quality Attributes

✧ Lack of awareness in the need of their specification

✧ Lack of know-how on;

- Categories and meanings,
- How to select and set relationships (trade-offs),
- How to specify,
- How to evaluate.

→ Without a precise specification of software quality, it is not possible to evaluate and achieve the quality of the sofware !!

Process and product quality

- **The quality of a developed product is influenced by the quality of the production process.**
  - This is important in software development as some product quality attributes are hard to assess.

- However, there is a very complex and poorly understood relationship between software processes and product quality.
  - The application of individual skills and experience is particularly important in software development;
  - External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality.

Process-based quality

Software standards

- Standards define the required attributes of a product or process. They play an important role in quality management.
- Standards may be international, national, organizational or project standards.
- Product standards define characteristics that all software components should exhibit.
  - e.g. a common programming style.
- Process standards define how the software process should be enacted.

Product and process standards

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Product standards - example: Software Quality Attributes

- Augment the description of the product's functionality by describing the product's characteristics in various dimensions that are important either to users or developers.

- E.g.;
  - Internal quality (developer perspective)
  - External quality (tester perspective)
  - Quality in use (user perspective)

ISO/IEC 25000 Quality Model Framework

- Process quality contributes to improving product quality, and product quality contributes to improving quality in use
- Evaluating quality in use can provide feedback to improve a product, and evaluating a product can provide feedback to improve a process

ISO 25010 Product Quality Attributes

Targets are set and results are evaluated via metrics defined for quality sub-attributes

Process standards - example: ISO 9001 standards framework

- An international set of standards that can be used as a basis for developing quality management systems.
- ISO 9001, the most general of these standards, applies to organizations that design, develop and maintain products, including software.
- It sets out general quality principles, describes quality processes in general and lays out the organizational standards and procedures that should be defined.
  - These should be documented in an organizational quality manual.

ISO 9001 core processes

Product Delivery Processes

- Business Acquisition
- Design and Development
- Test
- Production and Delivery
- Service and Support

Supporting Processes

- Business Management
- Supplier Management
- Inventory Management
- Configuration Management

ISO 9001 and quality management

ISO 9001 certification

- Quality standards and procedures should be documented in an organisational quality manual.
- An external body may certify that an organisation's quality manual conforms to ISO 9000 standards.
- Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

How to Achieve Quality?

- Traditional Approach: Defect detection
  - "Quality Control"
  - Achieving quality by targeting the product/service that is produced

- Modern Approach: Defect prevention
  - "Quality Assurance"
  - Achieving quality by targeting the system that produces the product/service

Software Errors, Faults and Failures

- *Bug/defect/fault* as a consequence of *human error*
  - results in non-conformance to requirements
  - may manifest as *failure* in running software

Reference: Galin, D. Software Quality Assurance: From Theory to Implementation, 2003.

When are defects introduced?

- The majority of defects are introduced in earlier phases.
  - Requirements are the top factor in a project's success or failure.

Phase	% of defects	% effort to fix
Requirements	56	82
Design	27	13
Code	7	1
Other	10	4

Cost of fixing defects

- Relative cost of fixing defects
  - benchmark: cost at requirements phase = 1

Phase found	Cost ratio
Requirements	1
Design	3 – 5
Coding	10
Unit / integration testing	15 - 40
System / acceptance testing	30 – 70
Production	40 - 1000

Quality Through Product/Service

- It is assumed that defects appear accidentally. When a defect is observed, the production line is <u>not</u> stopped, and defective parts are removed later.

- Forces workers to <u>not repeat the error</u>.
  - Workers may hide the problems.
  - Management may directly criticize workers.

Quality Through Process

- It is assumed that defects appear as a result of a problem in the production system. When a defect is detected, the production line is stopped. The reason of the defect is identified and removed (so that the same defect does not appear again).

- Forces workers to find defect and improve the production system.
  - It allows managers and workers to take responsibility of problems together.

Quality Assurance and Quality Control

- Quality Control
  - → For the purpose of defect detecting, at a certain time

- Quality Assurance
  - → For the purpose of defect prevention, over a period of time

- If we improve the quality of the development process, we both improve the product quality and reduce development time and cost.
  - "Quality is free"

Software Quality Assurance

- It is the cost-effective way of achieving quality during software development and maintenance.

- For the quality built into the software product, it aims to assure that processes applied and work-products produced within the software development life-cycle satisfy their specified requirements.

- It enables prevention and early detection of defects by planning, executing, and tracking a number of activites.

Scope of SQA Activities

- Software Quality Assurance (SQA) involves setting and executing mechanisms to ensure process and product quality throughout software development and maintenance.
  - Defining software processes, specifying standards to be used, setting up development environment, identfying SQA activities and their criteria
- Verification and Validation (V&V) involves assuring the quality of the work-products of the software development process against their specifications with the purpose of detecting non-conformances.
- Testing inovelves detecting defects by executing the code.

SQA includes V&V, and V&V includes Testing

SQA > V&V > Testing

Quality Through Product: Methods

- Static Methods: Applied by not executing the code (please refer to IEEE 1028:2008)
  - Management review
  - Technical review
  - Inspection
  - Walk-through
  - Audit
- Dynamic methods: Applied by executing the code
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing

V – Model of Software Development

Reviews and inspections

- A group examines part or all of a process or system and its documentation to find potential problems.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

- <u>There are different types of review with different objectives</u>
  - Inspections for defect removal (product);
  - Reviews for progress assessment (product and process);
  - Quality reviews (product and standards).

The software review process

Program inspections

- These are peer reviews where engineers examine the source of a system with the aim of discovering anomalies and defects.
- Inspections do not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Inspection checklists

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- Examples: Initialisation, constant naming, loop termination, array bounds, etc.

An inspection checklist (a)

Fault class	Inspection check
Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	• Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?

An inspection checklist (b)

Fault class	Inspection check
Interface faults	• Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	• Have all possible error conditions been taken into account?

Predictor and control measurements

Software metric

- Any type of measurement which relates to a software system, process or related documentation
  - Lines of code in a program, the Fog index, the number of reported faults in a delivered software product, number of person-days required to develop a component
- Allow the software and the software process to be quantified.
- May be used to predict product attributes or to control the software process.
- Product metrics can be used for general predictions or to identify anomalous components.

Relationships between internal and external software quality attributes

Static software product metrics

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.

Static software product metrics

Software metric	Description
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Some Examples

- Cyclomatic complexity

If G is the control flowgraph of program P and G has e edges and n nodes

$V(G) = e - n + 2$

$V(G)$ is the number of linearly independent paths in G.

Here $e = 16$, $n = 13$, $V(G) = 5$

More simply, if d is the number of decision nodes in G than

$V(G) = d + 1$

- Fan-in & Fan-out

The CK object-oriented metrics suite

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.

The CK object-oriented metrics suite

Object-oriented metric	Description
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Software component analysis

- System component can be analyzed separately using a range of metrics.
- The values of these metrics may then compared for different components and, perhaps, with historical measurement data collected on previous projects.
- Anomalous measurements, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

FOR STUDENT REVIEW…

Scope of quality management

- Quality management is particularly important for large, complex systems. The quality documentation is a record of progress and supports continuity of development as the development team changes.
- For smaller systems, quality management needs less documentation and should focus on establishing a quality culture.

Software quality

- Quality, simplistically, means that a product should meet its specification.

- This is problematical for software systems
  - There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
  - Some quality requirements are difficult to specify in an unambiguous way;
  - Software specifications are usually incomplete and often inconsistent.

Software fitness for purpose

- The focus may be ‘fitness for purpose’ rather than specification conformance.
  - Have programming and documentation standards been followed in the development process?
  - Has the software been properly tested?
  - Is the software sufficiently dependable to be put into use?
  - Is the performance of the software acceptable for normal use?
  - Is the software usable?
  - Is the software well-structured and understandable?

Importance of standards

- Encapsulation of best practice - avoids repetition of past mistakes.
- They are a framework for defining what quality means in a particular setting i.e. that organization's view of quality.
- They provide continuity - new staff can understand the organisation by understanding the standards that are used.

Problems with standards

- They may not be seen as relevant and up-to-date by software engineers.
- They often involve too much bureaucratic form filling.
- If they are unsupported by software tools, tedious form filling work is often involved to maintain the documentation associated with the standards.

Standards development

- Involve practitioners in development. Engineers should understand the rationale underlying a standard.
- Review standards and their usage regularly. Standards can quickly become outdated and this reduces their credibility amongst practitioners.
- Detailed standards should have specialized tool support. Excessive clerical work is the most significant complaint against standards.
  - Web-based forms are not good enough.

Quality reviews

- A group of people carefully examine part or all of a software system and its associated documentation.
- Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

Reviews and agile methods

- The review process in agile software development is usually informal.
  - In Scrum, for example, there is a review meeting after each iteration of the software has been completed (a sprint review), where quality issues and problems may be discussed.
- In extreme programming, pair programming ensures that code is constantly being examined and reviewed by another team member.
- XP relies on individuals taking the initiative to improve and re-factor code. Agile approaches are not usually standards-driven, so issues of standards compliance are not usually considered.

Agile methods and inspections

- Agile processes rarely use formal inspection or peer review processes.
- Rather, they rely on team members cooperating to check each other's code, and informal guidelines, such as 'check before check-in', which suggest that programmers should check their own code.
- Extreme programming practitioners argue that pair programming is an effective substitute for inspection as this is, in effect, a continual inspection process.
- Two people look at every line of code and check it before it is accepted.

Software measurement and metrics

- Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- This allows for objective comparisons between techniques and processes.
- Although some companies have introduced measurement programs, most organisations still don’t make systematic use of software measurement.
- There are few established standards in this area.

Use of measurements

- To assign a value to system quality attributes
  - By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
- To identify the system components whose quality is sub-standard
  - Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

Metrics assumptions

- A software property can be measured.
- The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- This relationship has been formalised and validated.
- It may be difficult to relate what can be measured to desirable external quality attributes.

Product metrics

- A quality metric should be a predictor of product quality.

- Classes of product metric
  - Dynamic metrics which are collected by measurements made of a program in execution;
    - Dynamic metrics help assess efficiency and reliability
  - Static metrics which are collected by measurements made of the system representations;
    - Static metrics help assess complexity, understandability, and maintainability.

Dynamic and static product metrics

- Dynamic metrics are closely related to software quality attributes
  - It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).
- Static metrics have an indirect relationship with quality attributes
  - You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

Problems with measurement in industry

- It is impossible to quantify the return on investment of introducing an organizational metrics program.
- There are no standards for software metrics or standardized processes for measurement and analysis.
- In many companies, software processes are not standardized and are poorly defined and controlled.
- Most work on software measurement has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS.
- Introducing measurement adds additional overhead to processes.

The process of product measurement

Measurement surprises

- Reducing the number of faults in a program leads to an increased number of help desk calls
  - The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase;
  - A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

Key points

- Software quality management is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability and so on.
- SQM includes defining standards for processes and products and establishing processes to check that these standards have been followed.
- Software standards are important for quality assurance as they represent an identification of ‘best practice’.
- Quality management procedures may be documented in an organizational quality manual, based on the generic model for a quality manual suggested in the ISO 9001 standard.

Key points

- Reviews of the software process deliverables involve a team of people who check that quality standards are being followed.
- In a program inspection or peer review, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions
- Software measurement can be used to gather data about software and software processes.
- Product quality metrics are particularly useful for highlighting anomalous components that may have quality problems.