CS 6300 Software Development Process

Jie Wu, Jack

Spring 2024

1 SOFTWARE ENGINEERING

1.1 Lecture 1 Introduction and Overview

1.1.1 What is Software Engineering

Bultan: software engineering is the discipline that investigates program development (efficiency and reliability)

Cleland-Huang: software engineering is the systematic application of methods to build software in a rigorous way (find a solution that balances all of the stakeholder needs)

Prasaol: software engineering is the whole process of creating software using engineering principles

Burnett: software engineering differs from programming in that it is about the entire life cycle and is concerned about quality (things long after the shipping of the software is the largest piece of software development)

Xie: software engineering is for better software productivity and quality

Emmerich: software engineering is fundamentally a venue-creating creativity involving social processes

Zeller: software engineering is the act of many people working together and putting together many versions of large and complex systems

Sullivan: software engineering is the activity of envisioning and realizing valuable new functions with sufficient and justifiable confidence that the resulted system will have all of the critical quality attributes that are necessary for the system to be a success, it is not only for the software components of engineering systems but for the system overall

Penix: software engineering is the art and practice of building software systems, it is important because you need to trade off different aspects (performance, scalability, reliability)

Meduidovic: software engineering is the set of methods and principles and techniques that have been developed to enable us to build large software systems that outpaces a small team of engineer's ability to understand, construct, and maintain over time, it is important because software is everywhere around us and the way we build and maintain is is something that determines almost a basic quality of life

Maletic: software engineering is about building and constructing very large-scale high-quality systems, it is important because software is ubiquitous

Briand: software engineering is the engineering discipline of developing software-based systems, usually embedded into larger systems composed of hardware, humans, and business

Visser: software engineering is about the ability to put big systems together so that they work reliably

1.1.2 The Software Crisis

The software crisis in the 60s resulted from

rising demand for software: moving from hardware dominated to software dominated

increased amount of development effort: needed due to increase in product complexity, the effort of an individual developer that can get the job done is called *programming effort*, for the rest it is called *software engineering effort*

slow developers' productivity growth: software size and complexity grow exponentially, while developer's productivity grows much slower

The NATO software engineering conference that was held in January 1969 is considered to be the birth of software engineering.

1.1.3 Software Development

A **software process** is a way of breaking down an otherwise unmanageable task into smaller steps that can be tackled individually. Having a software process is of fundamental importance for the following reason: for non-trivial systems you can't do it by just sitting down and developing it. You have to break down the complexity in a systematic and formal way.

1.1.4 Software Process

Software process is not just one but multiple processes. In this course we will focus on the following four main software processes:

waterfall: in the process we go from one phase to the other

evolutionary prototyping: start with an initial prototype and evolve it based on the feedback from the customer

rational unified process/unified software process: heavily based on the use of UML

agile: sacrifice the discipline a bit to be more flexible and more able to account for changes in requirements

1.1.5 Preliminary Questions

Professional software engineers produce 50–100 LOC/day.

1.1.6 Software Phases

Software processes are normally characterized by several phases called **software phases**. Only one phase mainly focuses on coding. The other are meant to support other parts of software development. They are

requirements engineering: talk to the stakeholders whoever we are building the software for

design: high-level structure

implementation: coding

verification and validation: make sure the code behaves as intended

maintenance: adding new functionality, eliminating bugs, responding to reported problems

1.1.7 Tools of the Trade

Tools and automation are fundamental in software engineering, for improving not only efficiency but also effectiveness. They can be used to help developers' productivity keep up pace with the complexity of software. Examples include

development: punch cards → IDEs

language: machine code → high-level languages

debugging: print lines → symbolic debuggers

We will use three main kinds of tools: (1) IDE (2) version control software or VCS (3) coverage and verification tools.

1.2 Lecture 2 Life Cycle Models

1.2.1 Introduction with Barry Boehm

A **software lifecycle** is a sequence of decisions that determine the history of a software. It is important to understand which models are good for which situations. Small projects are usually appropriate for agile, while larger projects may require a more rigorous approach. Criticality of the software also plays an important role when choosing a model. If the defect is loss of comfort or fund then agile is fine, which is not suitable if it is loss of life. So does the expected variability in the requirements. If you have a lot of unpredictable changes, then you don't want to spend lots of time writing plans and documents. You may even use multiple lifecycle models within a single project.

1.2.2 Traditional Software Phases

requirements engineering → design → implementation → verification and validation → maintenance.

1.2.3 Requirements Engineering

Requirements engineering or RE is the process of establishing the needs of stakeholders that are to be solved by software. It is important because the cost of correcting an error depends on the number of subsequent decisions that are based on it. Therefore errors made in understanding requirements have the potential for greatest cost, because many other design decisions depend on them and many other follow-up decisions depend on them. Traditional requirements engineering goes through a set of steps

elicitation: collection of requirements from stakeholders and other sources

analysis: study and deeper understanding of the collected requirements

specification: the collected requirements are suitably represented, organized, and saved so that they can be shared

validation: make sure the specified requirements are complete, consistent, no redundant and so on

management: accounting for changes made to requirements during the lifetime of the project

The above 5 steps form more of an iterative instead of sequential process.

1.2.4 Design

Design is the phase where software requirements are analyzed in order to produce a description of the internal structure and organization of the system. Traditional software design consists of the following design activities which result in a set of design products

architectural design: produces system structure

abstract specification: produces software specification

interface design: produces interface specification

component design: produces component specification

data structure: produces data structure specification

algorithm design: produces algorithm specification

This is just one possible list of activities, but the core idea is that we go from a high-level view of the system (architectural design) to a low-level view (algorithm design).

1.2.5 Implementation

Implementation realizes the design of the system. There are four fundamental pillars that can affect the way in which the software is constructed

reduction of complexity: aims to build software that is easier to understand and use

anticipation of diversity: takes into account that software construction may change in various ways over time, and in many cases the software evolves in unexpected ways thus we have to be able to anticipate some of the changes

structuring for validation: also called design for testability, makes the software easily testable during the subsequent verification and validation

use of (external) standards: makes the software conform to internal (e.g. coding standards or naming standards) and external standards (e.g. regulations)

1.2.6 Verification and Validation

Verification and validation check that the software system meets its specifications and fulfills its intended purpose. More precisely we can look at verification and validation independently

verification: did we build the system right? can be performed at different levels

- **unit level:** test the individual units work as expected
- **integration level:** test the interaction between different units
- **system level:** test the system as a whole, the level we apply validation or other testing techniques like stress testing or robustness testing

validation: did we build the right system?

1.2.7 Maintenance

Maintenance is the activity that sustains the software product as it evolves throughout its lifecycle. It is important because there might be (1) environment change (2) additional feature request (3) bug report. In response to these development organizations perform three kinds of maintenance activities

corrective maintenance: eliminate bugs

perfective maintenance: accommodate feature request, in some cases just to improve the software

adaptive maintenance: take care of the environment changes

After these activities have been performed the software developer produces a new version of the application and the new maintenance cycle begins. Maintenance is expensive and one of the reasons is **regression testing**—the activity of retesting software after it has been modified to make sure that the modification works as expected and does not introduce any unforeseen effect. The error discovered in regression testing is called **regression error**.

1.2.8 Software Process Model

Software process model, also called **software lifecycle model**, is a prescriptive model of what should happen from the beginning to the end of the software development process. The main function of the lifecycle model is to

- determine the order of the different activities
- establish the transition criteria between activities

1.2.9 Waterfall Process

The **waterfall model** is the grandfather of all lifecycle models, in which the project progresses in an orderly sequence of steps: software concept → requirements analysis → architectural design → detailed design → coding and debugging → system testing. The waterfall model performs well for

- software products in which there is a stable product definition
- the domain is well known
- the technology involved is well understood

The waterfall model helps identify errors in the early local stages of the projects. The main advantage and disadvantage of the waterfall model are

advantage: it allows developers to find errors early

disadvantage: it is not flexible and normally it is difficult to fully specify requirements at the beginning (requirement changes, the developers are not domain experts, the technology used are novel and evolving), hence it is less ideal for most real projects

1.2.10 Spiral Process

The **spiral model** was first described by Barry Boehm in his 1986 paper titled *A spiral Model of Software Development and Enhancement*. One main characteristic of the paper is that it describes the spiral model using a diagram. The spiral model is an incremental risk-oriented lifecycle model that has four main phases

determine objectives: the requirements are gathered

identify and resolve risks: the risks and alternative solutions are identified, a prototype is produced

development and test: software and tests for the software are produced

plan the next iteration: the output of the project is evaluated, and the next iteration is planned

A software project will go through these four phases in an iterative way, in which we learn more and more of the software and account for more and more risks. There are several advantages of using the spiral model

risk reduction: extensive risk analysis does reduce the chance of project failure

incremental functionality: functionality can be added at a later phase

early production: software is produced early in the software lifecycle, and at the end of any iteration there is a product, so that we can get early user feedback

The main disadvantages of the spiral model are that

- the risk analysis requires a highly specific expertise, and the whole success of the process is highly dependent on risk analysis
- the model is way more complex than other models thus is costly to implement

1.2.11 Evolutionary Prototyping Process

The **evolutionary prototyping model** works in four main phases

1. initial concept
2. design and implement initial prototype
3. refine prototype until acceptable
4. complete and release prototype

Therefore when developing a software using evolutionary prototyping, the software is continually refined and built. Hence it is an ideal process when not all requirements are well understood which is a very common situation. In this model the developers start by developing the parts of the system that they understand instead of working on developing a whole system. The partial system is then shown to the customer, and the customer's feedback is used to drive the next iteration of the refinement phase. Finally when the customer agrees that the prototype is good enough, the developers will complete the remaining work and release the prototype as the final product.
The main advantage and disadvantage of the evolutionary prototyping model are

advantage: immediate feedback, thus the risk of implementing a wrong system is minimized

disadvantage: it is difficult to plan in advance how long the development will take because we don't know how many iterations will be needed, and it easily becomes an excuse to do the kind of cut-and-fix approach in which we hack something together, fix the main issues based on the customer feedback, and continue this way until the final product is working but not of high quality

In fact there are many different kinds of prototyping models, and the evolutionary prototyping model is just one of them. For example the throwaway prototyping is another kind of prototyping model in which the prototype is just used to gather requirements but is thrown away at the end of the requirement gathering process instead of evolving into the final product.

1.2.12 Rational Unified Process

The rational unified process or RUP is based on UML. RUP works in an iterative way, which means that it performs different iterations and at each iteration it performs four phases. In each one of these four phases we perform standard software engineering activities, and we do them to different extents based on the phase in which we are.

inception: determine the scope and domain, perform cost and budget estimates

operation/elaboration: domain analysis, basic architecture design

construction: bulk of the development, most implementation

transition: the system goes from development into production so that it becomes available to users

1.2.13 Agile Process

This is a group of software development methods based on highly iterative and incremental development. We will focus on test driven development or TDD which is based on the iteration of the following three main phases

red/fail phase: write test cases that encode the requirements, which have not been coded up therefore must fail the tests

green/pass phase: write just enough code to make the test cases pass

refactor phase: as we repeat step 2 the structure of the code deteriorates, thus the code is refactored extensively in this step

1.2.14 Choosing A Model

Because process models define the master plan for a software development project, the model choice has as much influence over a project's success as any other major planning decision. Choosing the right model depends on

requirement understanding: if collecting all the requirements is difficult then we should follow a more flexible model

expected lifetime: quick project for a specific purpose vs. multi-year project to maintain

risk: more traditional models require good understanding of the domain and technology

schedule constraint: the amount of time and resource available

interaction with management/customer: many models rely on continuous customer feedback

expertise: some model requires specific expertise

1.2.15 Choosing A Model Quiz

The waterfall model is most suitable to developing a control system software, because the requirements and domain of a control system are usually well understood and we do not expect the requirements to change dramatically over time. However we should choose the spiral or evolutionary prototyping model instead if we expect mid-course corrections, because it is very expensive for a waterfall model to make changes during the course of the development especially changes that involve requirements. In summary iterative models work better in changing environment.

1.2.16 Lifecycle Documents

The documents that we produce are used for different purposes such as

- communicating details of the software system to different stakeholders
- ensuring the correct implementation of the system
- facilitating maintenance

There are standardized documents provided by the IEEE but they are kind of heavy-weight. Thus we will use the light-weight documents that were created for this course by modifying the standard ones for the projects of this course.

1.2.17 Classic Mistakes: People

heroics: too much emphasis on can-do attitudes, the idea that one person can do everything and can make a difference in the whole project encourages extreme risk taking and discourages cooperation

bad work environment: not create the right work environment which plays a big role in productivity

poor people management: lack of leadership or leadership that is exercised using the wrong means, note that adding people to a project that is behind schedule never works, because bringing new people up to speed would cause further delay

1.2.18 Classic Mistakes: Process

There are many types, for example

scheduling issue: unable to come up with a realistic schedule, because we underestimate the effort involved in different parts of the project or overestimate the ability of the people involved

planning issue: insufficient planning, abandoned planning due to pressure

failure: often unforeseen failures such as the failures on the constructor's end that might lead to low quality or late deliverable

1.2.19 Classic Mistakes: Product

There are many types, for example

gold plating: gold plating of requirements means that it is very common for projects to have more requirements than they actually need, e.g. marketing might want to add more features than what are needed by the user

feature creep: adding more and more features to a product that were not initially planned and are not needed in most cases, there is evidence that an average project experiences about a 25% growth in the number of features over its lifetime

research ≠ development: projects that strain the limits of computer science are better classified as research instead of development and thus should be managed accordingly e.g. highly unpredictable schedule

1.2.20 Classic Mistakes: Technology

There are many types, for example

silver-bullet syndrome: too much reliance on the advertised benefits of previously unused technology

switching tools: switch or add tools in the middle of a project, it is OK to upgrade a tool but new tools may not always have a steep learning curve

no version control: lack of an automated version control system

1.2.21 Classic Mistakes Quiz

Adding people to a late project is a people mistake, because it takes time to bring new people up to speed and having more people makes the communication more difficult.

1.3 Lecture 4 Version Control

1.3.1 Interview with John Britton

According to John Britton the most immediately obvious benefit of keeping revisions is that you can go back. The other benefit is being able to collaborate with multiple people.
In John Britton's opinion the main important characteristics of Git are (1) open-sourced (2) it is a distributed version control system so that you can track revisions of your software that doesn't have any central repository.

1.3.2 Version Control System Introduction

Version control system or VCS is a system that allows developers to manage multiple revisions of the same unit of information (documents, source files etc.). A VCS allows multiple actors to cooperate and share. VCS is useful for

enforce discipline: because it manages the process by which the control of items is passed from one person to another

archive versions: allows developers to archive versions

maintain historical information: allows developers to maintain lots of historical information about the archived versions

enable collaboration: a central repository of versions and associated information enables collaboration

recover from accidental deletions or edits:

conserve disk space: save disk space on both source control clients and on the server because there is only one central point where copies of different versions are stored, in addition VCS often uses efficient algorithms to store these changes

1.3.3 Essential Actions

add: add a file to the central repository

commit: commit locally made changes to the central repository so that they become visible to others

update: retrieve the changes others have committed

1.3.4 Example Workflow

Janet add + commit → Brad update → Brad commit → Janet update.

1.3.5 Don'ts in VCS

There are two kinds of resources that you don't want to add to a VCS

derived file: e.g. .exe as the source files are already in the repository

bulky binary file:

Another mistake is to save many local version files before modification. No local copies when using a VCS.

1.3.6 Two Main Types of VCS

VCS is mainly classified into two main types

centralized VCS: there is a single centralized repository

decentralized VCS: every user has a local repository to which they can commit their changes, when they are satisfied with the local version they can push the version to a centralized repository, the advantages are

- it allows users to take a snapshot for their own record without being visible to others
- users can push their local repository to multiple centralized repositories

1.3.7 Introduction to Git

Git is a distributed version control system that was initially designed and developed by Linus Torvalds for maintaining the Linux kernel.

1.3.8 Git Workflow

There are four fundamental elements in Git, which are

workspace: your local directory

index: also called stage

local repository: also referred to as head

remote repository:

A file in workspace can be in three possible states

committed: the latest changes are safely stored in local repository

locally modified: changes have not been saved to local repository

staged: the file is part of the index i.e. tagged to be considered in the next commit

The workflow of different Git commands for making changes are

clone: create a local copy of remote repository in workspace

add: add a file in the workspace to index, after which the file is staged

commit: all the files that are staged are released to local repository, for old files stage and commit can be done in one single command commit +a; for new files we have to manually add

push: push changes in local repository to remote repository

The workflow of different Git commands for retrieving changes are

fetch: get file from remote repository to local repository but not yet to workspace, it allows us to compare files before getting the latest version

diff head: get the difference between the file in workspace and the one in local repository, and based on the difference decide whether to merge or not

diff: get the difference between the file in workspace and the one staged for commit, it tells what could still be added to the stage for further commit

merge: take changes in local repository and get them to workspace

pull: similar to commit +a it is a shortcut for performing both fetch and merge

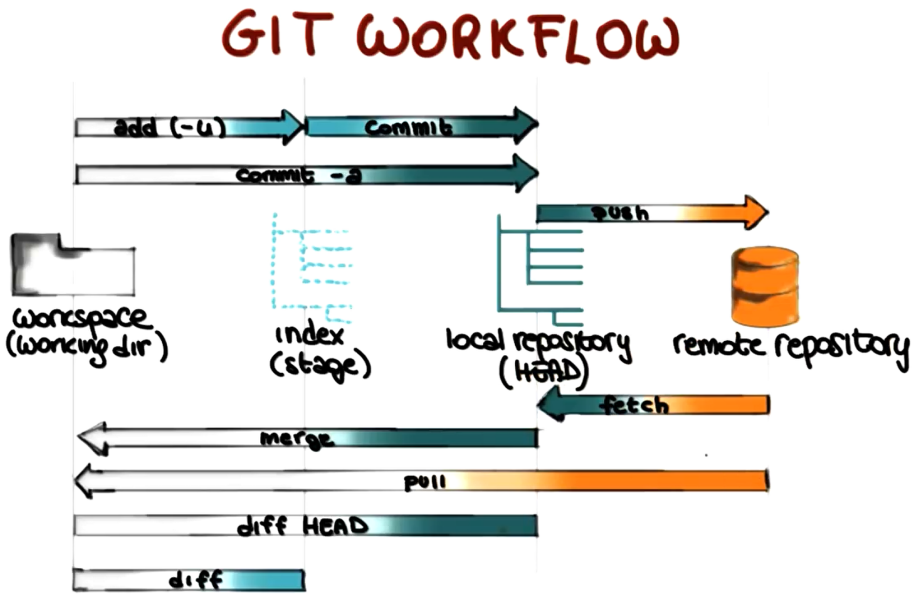


Figure 1: Git workflow

1.3.9 Git Recap: Local Repositories

create: mkdir test_project → cd test_project → git init

modify: git add new.txt → git commit (-a or -m) → git mv or git rm

inspect: git log, git status to see what files are changed, git diff filename to see the changes, git show to show the last commit

1.3.10 Git Recap: Remote Repositories

copy: git clone repository to create a local copy of the remote repository and link it to repository (referred to as origin), where repository can be a URL of file, http, ssh etc.

receive change: git pull

send change: git push

1.4 Lecture 5 Requirements Gathering

1.4.1 Choosing Good Questions Quiz

- The client should not need to know how many lines of code make up the program's source file. In forming requirements we should avoid implementation specific questions that do not directly interfere with the user.
- What OS should the program run on is relevant in some situations. However we should not write any OS specific code unless absolutely needed. Instead we should strive for making our code platform independent whenever possible.

1.4.2 Average Sentence Length Requirements

coding: language, compiler, way of execution

input: input format (file path), option (delimiter, upper bound)

output: output format (rounded to the nearest integer), option

2 REQUIREMENTS AND PROTOTYPING

2.1 Lecture 1 Requirements Engineering

2.1.1 Interview with Jane Cleland-Huang

According to Jane software requirements provide us a description of the functionality that the software has to deliver in order to satisfy its stakeholders. Requirements can be represented formally in the form of a set of "shall" statements or state transition diagrams, or informally in the form of user stories or use cases.
Requirements engineering is called engineering because it covers a number of different activities, including

- working with stakeholders to elicit or proactively discover their requirements are
- analyzing those requirements to understand the tradeoffs because different stakeholders caring about different things
- specifying the requirements
- validating the requirements are met

The requirement management process encompasses change management. Getting the requirements right includes finding the right stakeholders.

2.1.2 General RE Definition

Generally speaking requirements engineering or RE in short is the process of establishing the services that the customer requires from the software system. RE also has to do with the constraints under which the system operates and is developed. The final result of RE is a software requirements specification (SRS), which focuses on what the system is intended to do, not how it will do it—the focus of design.

2.1.3 Software Intensive Systems

Software is an abstract description of a set of computations that becomes concrete and therefore useful only when we run the software on some hardware and in the context of some human activity that it can support. It means that when we say software what we really mean is software intensive system, which is a combination of three things

$$\text{software intensive system} = \text{software} + \text{hardware} + \text{context}$$

We usually take hardware and context for granted in this equation, but they need to be explicitly considered when building a system.

2.1.4 Software Quality

Software itself does not define the quality of the software system. Instead software quality is a function of both the software and its purpose. Hence a software system can be of low quality not only because it does not work well but also because it does not fulfill its purpose which happens quite often. In other words we can define the quality of software in terms of fitness for purpose, and RE is mostly about identifying the purpose of the software.

2.1.5 Identifying Purpose

Identifying the purpose of a software system means defining the requirements for the system, which is extremely difficult because

- the inherent complexity of the purpose/requirements
- it is hard to figure out what people really want ("often people don't know what they want until you show it to them")
- requirements change over time
- multiple stakeholders often have conflicting requirements which can be hard to reconcile

2.1.6 Completeness and Pertinence

Two particularly relevant and common problems of requirements are

completeness: it is often extremely difficult to identify all requirements

pertinence: relevance of the requirements, to avoid the completeness problem developers often end up collecting a lot of irrelevant albeit not conflicting requirements

2.1.7 Pertinence Quiz

That the list shall be stored as a linked list is how to do it instead of what to do, thus should not be part of the requirements.

2.1.8 Irrelevant Requirements Quiz

Irrelevant requirements are harmful because

- they can introduce inconsistency
- they can waste project resources

2.1.9 Best Practices

In general requirements documents are difficult to read, and often the stakeholders are short on time overwhelmed by the amount of information they are given. So they give in to the pressure and sign.

2.1.10 RE Definition Breakdown

Requirements engineering (RE) is a set of activities concerned with identifying and communicating the purpose of a software-intensive system, and the context in which it will be used. Hence RE acts as the bridge between the real-world needs of users, customers, and other constituencies affected by a software system, and the capabilities and opportunities afforded by software-intensive technologies. From this traditional definition of RE we can infer that

- communication is as important as analysis
- software quality can be assessed in terms of fitness-for-purpose
- designers need to know the context—how and where the system will be used
- requirements are partly about what is needed (capabilities) and partly about what is possible (opportunities), we need to compromise between these two
- we need to identify all stakeholders (other constituencies) not just the customer and user

2.1.11 Defining Requirements

At a high level we have two domains:

machine domain: characterized by computer (hardware) and program (OS, libraries etc.)

application domain: characterized by domain properties (assumptions made on the domain) and requirements (what we would like to achieve by delivering the system)

The intersection of machine domain and application domain, the bridge between the two, is what we call the **specification**, which is a description of what the system that we are building should do to meet the requirements. In this shared domain there are two types of phenomena:

application domain: events in the real world that machine can directly sense (e.g. button pushed)

machine domain: actions in the real world that machine can directly cause (e.g. screen turned on)

2.1.12 Functional and Non-Functional Requirements

Functional requirements have to do with the functionality of the system, which have generally well-defined satisfaction criteria. **Non-functional requirements** refer to system qualities such as security, accuracy, cost, interoperability etc, which do not always have clear satisfaction criteria (e.g. fast elevator) hence we need to refine them so that they become verifiable.

2.1.13 User and System Requirements

user requirements: written for customers, often in natural language, no technical details

system requirements: written for developers, contain detailed functional and non-functional requirements, clearly and more rigorously specified

We need to define both because they serve different purposes.

2.1.14 Requirement Origins

There are many possible sources of requirements:

stakeholder: anybody who is affected by the system

application domain: there are constraints that are characteristics of the application domain which will affect the functionality of the system (e.g. regulations)

documentation: everything that refers to the functionality of the system that we are building

2.1.15 Elicitation Problems

They are

thin spread of domain knowledge: knowledge is rarely available in an explicit form, moreover knowledge is often distributed across many sources, what is even worse there are often conflicts between the knowledge gathered from different sources

knowledge is tacit: people often find it hard to describe knowledge that they regularly use, let alone pass it to someone else

limited observability: identifying requirements through observation is often difficult as the problem owners might be too busy to perform the task that we need to observe, or they might be doing lots of other things together with the task that we need to observe, moreover the presence of an observer might change the problem since it is very typical for human subjects to improve or modify an aspect of their behavior under observation

bias: the information that we collect might be biased because (1) people might not feel free to share or just don't want to share what we need to know

2.1.16 Traditional Techniques

They are

background reading: collect information by reading existing documents, especially appropriate when one is not familiar with your organization for which the requirements are being collected, one disadvantage is that written documents are often out of sync with each other and with the reality

hard data and samples: decide which hard data to collect and choose the sample of the population for which to collect such data, hard data include forms, invoices, financial statement etc.

interview: can be structured in which case there is an agenda of open questions, or can be open-ended in which case there is no preset agenda

advantage: can collect a rich set of information because they allow for uncovering opinions as well as hard facts, moreover they can probe in-depth through followup questions

disadvantage: require special skills that are difficult to master and experience

survey: can quickly collect information from a large number of people and can be administered remotely, however it tends to severely constrain the information the user can provide and might miss opportunities to collect unforeseen relevant information

meeting: generally used for summarization of findings and collection of feedback, should have clearly stated objectives and is planned carefully but does not always happen in practice

2.1.17 Other Techniques

They can be divided into three main groups:

collaborative techniques: created to support incremental development of complex systems with large diverse user populations, the most well-known one is brainstorming

social approaches: explore social science to better collect information from stakeholders and environment, one example is ethnographic techniques based on the idea of collecting information on the participants by observing them in their original environment

cognitive techniques: leverage cognitive science to discover expert knowledge

2.1.18 Modeling Requirements

what to model: depends on which aspects of the requirements we want to focus on (e.g. enterprise, information and behaviors, system qualities), often they are orthogonal to one another hence are complimentary

how to model it: different models can be used to provide views of the requirements from different perspectives, e.g. UML diagrams provide graphical depiction, one extremely popular model is *goal modeling*, the main idea is to start with the main goal of the system and then keep refining it by decomposing it into sub-goals, the refinement continues until we get to goals that can be operationalized and represent the basic units of functionality of the system

2.1.19 Analyzing Requirements

There are three types of analysis

verification: developers will study the requirements to check whether they are correct, whether they accurately reflect the customer needs, developers can also check the completeness and pertinence of the requirements

validation: stakeholders assess whether the collected requirements define the system that they really want, stakeholders can assess the requirements by interacting with a prototype of the system if the requirements engineering process being used involves early prototyping

risk analysis: aims to identify and analyze the main risks involved with the development

2.1.20 Requirements Prioritization

When we are unable to satisfy all the requirements due to limited resources, we need to prioritize them by classifying them into one of three classes: (1) mandatory (2) nice to have (3) superfluous.

2.1.21 Requirements Engineering Process

RE is an iterative process consisting of four steps: elicitation → negotiation → modeling → analysis.

2.1.22 Software Requirements Specification

Software requirements specification or SRS is important because it represents a common ground between analysts and stakeholders and is a way to communicate requirements to others. IEEE defined a standard that divides the document in predefined sections. We will use a simplified version of the IEEE SRS format that includes three main sections: (1) introduction (2) user requirements (3) system requirements.
A few important characteristics that requirements should have are

simple: each requirement should express one specific piece of functionality

testable: untestable requirements are useless

organized: related requirements should be grouped, more abstract requirements should contain more details, and priorities should be clearly indicated when present

numbered: so that they can be traced

2.2 Lecture 2 Object-Oriented Software Engineering and UML

2.2.1 Object Orientation Introduction

Object orientation means

precedence of data over function: data items become the center of development activities

information hiding/encapsulation: encapsulation and segregation of data behind well-defined and ideally stable interfaces in order to hide the design and implementation decisions

inheritance: allows the reuse of object definition by incremental refinement

One motivation behind object orientation is to make code more maintainable, because the rest of the system doesn't need to be concerned about how the implementation details or the design are defined. Therefore any change that happens behind this wall.

2.2.2 Objects and Classes

An **object** is a computing unit organized around a collection of state or **instance variables** that define the state of an object. In addition, each object has associated with it a set of **operations** or methods that operate on such state. In traditional object orientation we say that operations are invoked by sending a message to the appropriate object, which is what we normally call method implication.
A **class** is basically a template from which new objects, which is what we call instances of the class, can be created. The fact that a class that allows us to create as many objects as we want can further reuse also contributes to make the code more readable, understandable, and therefore ultimately more maintainable.

2.2.3 Benefits of OO

The motivation behind adopting object orientation include

reduce maintenance cost: reduce long-term maintenance cost by limiting the effect of changes, because encapsulation and information hiding makes it easier to modify parts of the system without affecting the rest of the system

improve development process: by favoring code and design reuse

enforce good design: such as the ones that we saw in encapuslation, information hiding, high cohesion, low coupling

2.2.4 OO Benefits Quiz

Normally the fact of designing real-world entities, which is one characteristic of the object oriented approaches, does increase understandability. Hence increased understandability is another benefit of object orientation.

2.2.5 OO Analysis History

The use of object orientation led to what we call OOAD—object oriented analysis and design. OOAD is a software engineering approach whose main characteristics is to model a software system as a group of interacting objects. In this lesson we will focus on the first part of OOAD—object oriented analysis, which is a requirements analysis technique that concentrates on modeling real world objects. This transition from a function-centric world to a data-centric world was initiated by James Rumbaugh, who in the 90s developed an integrated approach to object oriented modeling with three main aspects:

data: the object model, using an extended version of entity relationship diagrams to describe classes and inheritance

function: the functional model, data flow diagrams were used to represent the functional aspects of the system, where each function was then becoming a method in a class

control: representing the dynamic aspects of a system, using state machines to represent how a system would evolve going from one state to the other based on what happened to the system

These models together represented what was called the *Object Modeling Technique* or OMT. OMT combined with contributions from several people, in particular Jacobson and Booch, evolved into what we call the **Unified Modeling Language** or UML. UML extends OMT by providing more diagrams and a broader view of a system from multiple perspectives.

2.2.6 OO Analysis

Traditional analysis and design techniques were functionally oriented. Conversely object oriented analysis is primarily concerned with data objects, that is during the analysis phase we define a system first in terms of the data types and their relationships, and the functions or methods are defined only later and associated with specific objects.
The basic idea of object oriented analysis is to go from real world objects to a set of requirements, and we can describe this as a four-step process:

1. obtain/prepare a texture description of the problem to be solved

2. underline the nouns in the description to identify classes

3. underline the adjectives of the nouns to identify the attributes of the classes
4. underline the active verbs in the description to define the operations of the classes

2.2.7 Running Example: Course Management System

The textual description of the system is as follows

1. The registrar sets up the curriculum for a semester using a scheduling algorithm.
2. One course may have multiple course offerings.
3. Each course offering has a number, a location, and a time associated with it.
4. Students select 4 primary courses and 2 alternative courses by submitting a registration form.
5. Students might use the course management system to add or drop courses for a period of time after registration.
6. Professors use the system to receive their course offering rosters.
7. Users of the registration system are assigned passwords which are used for login validation.

2.2.8 UML Structural Diagrams

UML structural diagrams represent the static characteristics of a system. This is in contrast with dynamic models which represent the behaviors of a system instead. One kind of UML structural diagrams is **class diagram**, which represents a static structural view of the system, describing (1) the classes and their structure (2) the relationships among classes.

2.2.9 Class Diagram: Class

In a class diagram, a class is represented as a rectangle with three parts:

class name: should be named using the vocabulary of the domain, the normal naming standard requires that class names are singular nouns starting with a capital letter

attributes: a set of attributes for a class defines a state for the class, in addition to name we can define type and initial value for an attribute

operations: normally represented by a name with a list of arguments and the result type

The plus and minus sign before an attribute or an operation depicts the visibility of this class member, with

minus sign: private to the class i.e. only instances of this class can access it, which allows us to enforce the information hiding principle

plus sign: visible outside the class, which allows us to define an interface for the class

We can also use ellipses to indicate that there are more attributes or more operations, but we just don't want to list them now.

2.2.10 Class Diagram: Attributes

Attributes represent the structure of a class, the individual data items that compose the state of the class. Attributes may be found by

- examining class definitions
- studying requirements
- applying domain knowledge—domain knowledge tends to be fairly important to identify things which might not be provided in the description of the system that is incomplete

2.2.11 Class Diagram: Operations

Operations represent the behavior of a class, which may be found by examining interactions among entities in the description.

2.2.12 Class Diagram: Relationships

Relationships describe interactions between objects. There are three main types of relationships:

dependency: X uses Y

association/aggregation: X contains a Y

generalization: X is a Y, this is the relationship that expresses inheritance

2.2.13 Class Diagram: Dependency

Dependency is represented by a dash arrow pointing from a client to a supplier. An argument of an operation also represent a dependency.

2.2.14 Class Diagram: Association

Association is represented by a solid line, which allows adornments:

label: clarify the nature of the relationship

arrow: clarify the direction of the relationship

multiplicity: define the number of instances of one class that are related to one instance of the other

Aggregation is a relationship between two classes in which one represents a larger class like a whole which consists of smaller classes like the parts of this whole. We use a solid line with a diamond end to represent aggregation with the diamond pointing to the larger class.

2.2.15 Class Diagram: Generalization

Generalization is a relationship between a general class, which we normally call super-class, and a more specific class, a class that refines the super-class and what we normally call sub-class. It is represented with a solid line with a big arrow head at the end pointing to the super-class.

2.2.16 Class Diagram: Creation Tips

- understand the problem
- choose good class names (to make it easier to create the mapping between the real-world objects and the entities in the model)
- concentrate on the WHAT not the HOW
- start with a simple diagram
- refine it iteratively until it is complete

2.2.17 UML Structural Diagrams: Component Diagram

A **component diagram** is a static view of the components in a system and their relationships which can be used to represent an architecture. More precisely in a component diagram

node: represent a component which consists of one or more classes with a well-defined interface

edge: indicate relationships between the components, which can be read as component A uses services of component B, as far as edges are concerned there are two kinds of edges:

- the first kind of dashed edges which were part of the original UML definition and indicate use
- the second kind of edges have a richer representation: a lollipop indicates a provided interface while a socket indicates a required interface

In a component diagram components or relationships can be annotated.

2.2.18 UML Diagrams: Deployment Diagram

A **deployment diagram** provides a static deployment view of a system, and unlike previous diagram it is about the physical allocation of components to computational units. In a deployment diagram

node: correspond to a computational unit, also display which components are deployed on this node

edge: indicate communication between computational units

2.2.19 UML Behavioral Diagrams: Use Case

UML behavioral diagrams deal with the behavior, the dynamic aspects of the system, rather than the static ones. One fundamental UML behavioral diagram is **use case diagram**. A **use case**, also called scenario, script, or user story, represents

- the sequence of interactions of outside entities, normally called actors, with the system that we're modeling
- the system actions that yield an observable result of values to the actors

The basic notation of use case consists

name: use case

role name: actor

solid line: is the actor of

2.2.20 Use Case: Actor

An **actor** represents an entity from the outside world, which can be a human or a device, that interacts with the system. Note that

- an entity can play more than one role
- more than one entity can play the same role
- an actor may appear in more than one use case

2.2.21 Build a Use Case Diagram

To build a use case diagram we need to add use cases for actors, and show how the actors interact with the system through the use cases. The behavior of a use case can be specified by describing its flow of events, which should be described from an actor's point of view. The description should detail what the system must provide to the actor when the use case is executed. In particular it should describe

- how the use case starts and ends
- normal flow of events
- possible alternative flow of events when there are multiple ways of accomplishing one action
- exceptional flow of events

The description can be provided in two main ways:

informal: a textual description of the flow of events in natural language

formal: use pre and post conditions, pseudo code to indicate the steps, or sequence diagrams

2.2.22 Role of Use Cases

requirement elicitation: it is much easier to describe what the system should do if we think about the system in terms of scenarios of usage rather than trying to describe the whole functionality of the system at once

architectural analysis: use cases are the starting point for the analysis of the architecture of the system since it can help identify the main blocks of the system, thus can help define the initial architecture

user prioritization: use cases can be used to prioritize users so as to define which part of the system should be built in which order

planning: knowing which pieces of functionality to build and in which order facilitates better plan of the development of the system

testing: an early description of the main pieces of functionality of the system and the interaction between the actors and the system allows defining test cases even before writing the code and defining the system

2.2.23 Use Case Diagram: Creation Tips

- use name that communicates purpose
- define one atomic behavior per use case
- define flow of events clearly
- provide only essential details
- factor common behaviors
- factor variants

2.2.24 UML Behavioral Diagrams: Sequence Diagram

A **sequence diagram** is an interaction diagram that emphasizes how objects communicate and the time ordering of the messages between objects. The steps needed to build such a sequence diagram are:

1. place the objects that participate in the interaction at the top of the diagram along the x-axis, with objects that initiate the interaction at the left and increasingly more subordinate objects to the right, reflecting the way the events will flow for the majority of the interactions in the system
2. add object lifeline—a vertical line that shows the existence of an object over a period of time, normally represented with a dashed line except for the leftmost object for which it is a solid line
3. place messages that these objects send and receive along the y-axis in order of increasing time from top to bottom, maybe also a number to further clarify the sequence
4. add the focus of control which is a tall thin rectangle that shows the period of time an object is performing an action either directly or indirectly

2.2.25 UML Behavioral Diagrams: State Transition Diagram

A **state transition diagram** is defined for each relevant class in the system and basically shows the possible live history of a given class. It describes

- the possible states of the class as defined by the values of the class attributes
- events that cause a transition from one state to another
- actions that result from a state change

In a state transition diagram states are represented by ovals with a name; transitions are marked by a solid arrow labeled with events that trigger the transitions. Note that not all events will cause a state transition. Events may produce actions. They may also have attributes which are analogous to parameters in a method call, and Boolean conditions that guard the state transition.
States may also be associated with activities and actions:

activity: operation performed by an object when it is in a given state that takes time to complete

action: instantaneous operation performed by an object, can be triggered when the object reaches a given state, when the object exits that state, or when a specific event occurs—a shortcut for any event that will cause a state transition bringing the object back into the same state

If we have several actions and activities, it is worthwhile clarifying their ordering, with

1. the actions on the incoming transition performed first
2. if there is an entry action, that is the next action that would be performed
3. then we have activity and event actions as appropriate
4. exit actions

UML structural diagrams: class diagram → component diagram → deployment diagram

UML behavioral diagrams: use case diagram, sequence diagram, state transition diagram

2.3 Lecture 3 Android

2.3.1 Android Introduction

Android is an operating system designed for mobile devices. It is running customized version of the Linux kernel, and in order to be able to run application it is powered by a Java-based virtual machine called the Dalvik VM. The Dalvik VM is optimized low processing power and low memory environments. Therefore Android applications, or Android apps as we normally call them, are Java-based applications that run on Dalvik.

2.3.2 Basic Architecture of Android

The architecture of Android by layers from the highest to the lowest are:

apps: applications written in Java and compiled into byte code, packaged and installed in Android, e.g. contacts, phone, browser, game

application framework: provides the services that are needed by Android apps to run, e.g. windows manager, telephony manager, location manager, resource manager

libraries and Android runtime: the Android native libraries provide APIs for various functionalities, e.g. SQLite API, webkit API, SSL API, OpenGL API
the Android runtime consists of the Dalvik VM and the core Java libraries which are the Android version of the standard Java libraries also called JDKs

Linux kernel: the core of the Android operating system, a standard Linux kernel with some customizations made by Google, the kernel interacts with the hardware and includes all the essential hardware drivers to access the network, the file system, the screen, and so on

2.3.3 Android App Components

Slightly different from traditional applications, an Android application is a collection of several components of different types—activities, services, content providers, and broadcast receivers. These four components are connected by intents and tied together by an Android manifest, which is a .xml file that declares the components and some properties of your application.

2.3.4 Activity

In Android an **activity** is a single screen with a user interface. Activities are independent units. But they can work together to form a cohesive whole and can be invoked by other applications. From the programmatic standpoint we can create an activity by simply extending the activity class, which is part of the Android system.

2.3.5 Service

A **service** is an application component that performs a usually long running operation in the background while not interacting with the user e.g. music playing service, download service. Therefore unlike an activity services do not provide a user interface. Similar to activities from the programmatic standpoint the way to create a service is simply to extend the service class.
A service is not a way to offload some work that should be logically done by the application. But it is a way for the app to tell the system about something that it wants to continue doing in the background even when the user leaves the application. It is also a way for the application to expose some of this functionality e.g. the music playing functionality. For example a navigation app will bind to the music player service to be able to stop the audio temporarily when making an announcement.

2.3.6 Content Provider

The **content provider** provides a structure interface (insert, update, delete, query) to a set of data. Android contains an implementation of the SQLite database manager. So very often apps rely on a SQLite database to store the data and a content provider to give it the access to such data. In some cases a provider can be used for accessing data within an application, in which case the data will be application data, such as a list of items to be bought for a shopping list management application. More generally a provider can be used by an application also to share data with other applications, e.g. the address book application could use a provider to give the user contacts to other applications such as the telephone application and the messaging application etc.

2.3.7 Broadcast Receiver

The **broadcast receiver** can register with the Android system for a specific event of interest. The Android system will then notify the broadcast receiver every time an event of that kind occurs. A music player could use a broadcast receiver to register for the event of incoming phone calls, because it wants to suspend playing music every time there is a phone call incoming. Similarly it could also register for the end of the phone call because it might want to resume playing music every time a phone call is ended.

2.3.8 Intents

The four components can be connected by using intents. An **intent** is an abstract description of an operation to be performed and it consists of two main parts

action: the action to be performed, which is indicated as a string

data: the data on which the action will operate

For example a phone call intent has call as an action and phone number from the contact database as the data. Intents are useful because they provide developers with a way to perform late binding between different applications to connect applications that were not initially meant to be connected. That is they allow for binding at runtime to otherwise completely decoupled applications. There are three different ways in which the binding can happen:

direct binding with activity: contacts app → telephone app

direct binding with service: message app → play audio service

broadcast: contacts app broadcasts call intent, user will be provided with a choice of which app to use to call when there are multiple activities that could handle that intent

2.3.9 Manifest

In a manifest is a XML file that

- declares all of the steady components of your app—activities, services, and content providers, broadcast receivers can be defined in either statically in the manifest or dynamically in runtime so they don't necessarily have to be in the manifest
- declares all the permissions required for your app to work
- specifies the entry point for the app, that is which activity should be launched when the application is executed
- declares the version of the app, which is used for checking updates
- declares the lowest Android SDK version for which the app is valid, checked at installation time

2.3.10 Android Activity Lifecycle

Because Android apps run in a multitasking environment, the runtime system routinely suspends, stops, and even kills applications and activities. When the Android system needs to suspend or kill an activity, it does so by calling some special methods in the activity. Therefore when we develop an Android app, we must be aware of what these methods are and suitably implement them. When another activity comes into the foreground, two things may happen:

onPause(): it may lose its focus but still be visible in the background, this may be a case in which the new activity that is being created is an activity that doesn't take up the whole screen, or a case in which the new activity is a transparent activity, a paused activity is completely alive in the sense that it maintains all state information and remains attached to the window manager

onStop(): the activity is finishing or being destroyed by the system, this may be a case in which there is not enough memory to run both the new activity and the original activity, or more often a case in which the new activity being launched is completely obscuring the original activity, similar to the paused state in the stopped state an activity still retains all of state information but it is more likely to be killed or destroyed by the system

Two scenarios can happen from these two states:

killed: the app with higher priority needs memory, if the user happens to go back to the activity the system will recreate it from scratch using onCreate(), this is the reason why it is very important that when your application goes into the paused or stopped state you suitably save the state of the application

destroyed: the activity is finishing or is being completely destroyed by the system, in which case the onDestroy() method is called and the activity is shut down for good

returned: the activity is not killed nor destroyed and the user simply returns to the activity, in which case the system will call onRestart() on the activity and the activity will go back to the running state

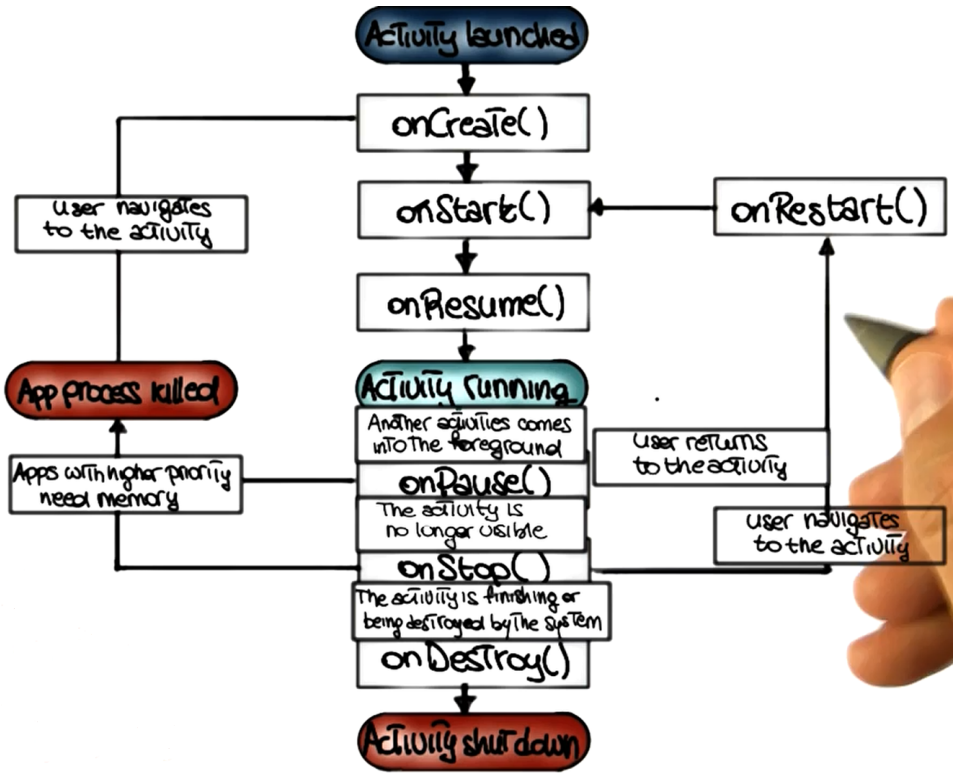


Figure 2: Android Activity Lifecycle

3 SOFTWARE ARCHITECTURE

3.1 Lecture 1 Software Architecture

3.1.1 Interview with Nenad Medvidovic

- Many design decisions will typically not really impact the success of your system and the long term well-being of your system. But there are other design decisions that will potentially impact how the system operates. These design decisions are architectural design decisions, which are the principal design decisions in your system. All the other design decisions you could tag with being important are below this very important or highly important threshold. A system could be successful and poorly architected.

- The non-architectural design decisions are on the average much easier to make. Yet the scale of the consequences of making such a change can vary from very minor and highly localized to very important and system wide.

- **Architectural erosion** refers to the situation where continuous change in architecture albeit based on local optimization ends up with a software system that is distant from the original one, and the global optimal behavior of the system is badly compromised manifested by the compromised structural soundness and the flawed non-functional properties (security, reliability, usability, maintenability).

3.1.2 What is Software Architecture?

Two seminal definitions of software architecture (SWA) are:

Perry & Wolf: software architecture is comprised of

elements (what): the processes, data, and connectors that compose a software architecture

form (how): the set of properties of and relationships among these elements

rationale (why): the justification for the elements and their relationships

Shaw & Garland: software architecture is a level of design that involves

- a description of elements from which these systems are built
- the interactions among those elements
- the patterns that guide their composition
- the constraints on these patterns

In fact there are many more alternative definitions of software architecture.

3.1.3 A General Definition of SWA

We define a SWA as the set of principal design decisions about the system, where principal implies a degree of importance that grants a design decision architectural status. In some cases the distinction between these two kinds of design decisions is clear, while in other cases it is much fuzzier and it depends on the context. The bottom line is that if you believe that something is an important design decision, that becomes an architectural decision. In this spirit we can see a software architecture as the blueprint for a software system that we can use to construct and evolve the system. This blueprint encompasses every facet of the system under development—structure, behavior, interaction, and non-functional properties.
Software architecture also has a temporal aspect. A SWA is not defined at once but iteratively over time. At any point in time there is a SWA but it will change over time, because design decisions are made, unmade, and changed over a system's lifetime.

3.1.4 Prescriptive vs. Descriptive Architecture

A **prescriptive architecture** captures the design decisions that are made prior to the system's construction. This is what we normally call the *as-conceived software architecture*. Conversely a **descriptive architecture** describes how the system has actually been built. So it's based on observing the system as it is and extracting the architecture from the observation. This is what we normally call the *as-implemented software architecture*. Often these two architectures end up being different.

3.1.5 Architectural Evolution

Ideally when a system evolves, its prescriptive architecture should be modified first. Unfortunately in software engineering this rarely ever happens. In practice the system and therefore it's descriptive architecture are often directly modified, and these two architectures start diverging. This happens for a number of reasons:

- developers' sloppiness
- short deadlines
- lack of documented prescriptive architecture

3.1.6 Architectural Degradation

architectural drift: the introduction of architectural design decisions that are orthogonal to a system's prescriptive architecture, that is they are not included in or encompassed by or implied by the prescriptive architecture, so that the architecture becomes unnecessarily complex

architectural erosion: the introduction of architectural design decisions that violate a system's prescriptive architecture, that is they don't comply with the prescriptive architecture, resulting in a poor architecture

3.1.7 Architectural Recovery

There are two main options for dealing with a degraded architecture:

- keep frantically tweaking the code which normally leads to disaster, because you don't exactly know what you are changing and therefore you are basically stabbing in the dark
- try to determine the software system architecture from its implementation level artifacts and try to fix it, this is normally called **architectural recovery**

3.1.8 Real World Example

One example is Linux kernel. Different from its prescriptive architecture, in its descriptive architecture pretty much everything talks to everything else which is in general not a good thing. In addition there are several things that don't make sense including (1) the library calls both the file system and the network interface (2) the file system calls the kernel initialization code.

3.1.9 More Example

Another example is the architecture of iRods system, a data grid system built by a biologist for storing and accessing big.
A more well-known example is Hadoop. In the descriptive architecture of Hadoop 61 out of 67 components in the system have circular/mutual dependencies.

3.1.10 Final Example

The final example is bash, a Unix shell also called `sh`. There are two design problems of the component:

lack of cohesion within the components: only a few connections exist between the sub-components

high coupling with outside components: the components has many connections with outside components

3.1.11 Architectural Design Quiz

The ideal characteristics of an architectural design include

scalability: the ability to handle the growth of the software system (e.g. adding new web servers)

high cohesion: the elements within a module are strongly related

low coupling: different modules are independent of each other

A system with high cohesion and low coupling is easier to understand and maintain.

3.1.12 Architectural Elements

A software system's architecture typically is not, and should not be, a uniform monolith. It should be a composition and interplay of different elements. There are three main types of elements in an architecture

processing elements: elements that implement the business logic and perform transformations on data

data elements: also called information or state, are those elements that contain the information that is used and transformed by the processing elements

interaction elements: the glue that holds the different pieces of the architecture together

Processing and data elements are contained in the system components, whereas interaction elements are maintained and controlled by the system connectors. Components and connectors together form a system's configuration, which models components, connectors and their relationships.

3.1.13 Components, Connectors, and Configurations

software component: an architectural entity that

- encapsulates a subset of the system's functionality and/or data, basically components typically provide application specific services
- restricts access to that subset via an explicitly defined interface, in addition a component can also have explicitly defined dependencies on its required execution environment

software connector: an architectural entity that effects and regulates interactions among components, thus connectors typically provide application independent interaction facilities
in many software systems connectors might simply be procedure calls or shared data accesses, however in complex systems interactions might become more important and challenging than functionality

architectural configuration/topology: a set of specific associations between the components and connectors of a software system's architecture

3.1.14 Configuration Example

A configuration can also have hierarchically decomposable components. A component diagram as discussed in UML before can also be used to represent an architectural configuration.

3.1.15 Deployment Architectural Perspective

Deploying a system involves mapping components and connectors to specific hardware elements. The deployment view of an architecture can be critical in assessing whether the system will be able to satisfy its requirement, because this map allows you to discover and assess other characteristics of your system that you might not have considered. For example

memory: is there enough memory available to run the system?

power: is the power consumption larger than what the device can handle?

bandwidth: does the system have enough network bandwidth to enable the required interactions?

3.1.16 Architectural Styles

There are certain design choices that when applied in a given context regularly result in solutions with superior properties. Architectural styles capture exactly these solutions. Shaw and Garlan define an architectural style as a family of systems in terms of

- a pattern of structural organization
- a vocabulary of components and connectors
- with constraints on how they can be combined

In summary an **architectural style** is a named collection of architectural design decisions applicable in a given context. It is important to study and know these architectural styles because

- it allows us to avoid reinventing the wheel
- it allows us to choose the right solution to a known problem
- in some cases it even allows us to discover even more advanced styles if we know the basic ones

3.1.17 Types of Architectural Styles

pipes and filters: an architectural style in which a chain of processing elements is arranged so that the output of each element is the input of the next one, usually with some buffering in between consecutive elements, e.g. Unix pipes

event driven: typically consists of event emitters and event consumers, event consumers are notified when events of interest occur and have the responsibility of reacting to those events, e.g. GUI in which widgets generate events and listeners listen to those events and react to them

publish-subscribe: an architectural style in which senders of messages called publishers publish messages without knowledge of who will receive such messages, while subscribers express interest in one or more tags and will only receive messages of interest according to such tags, e.g. twitter

client-server: the server provides the resources and functionality, and the client initiates contact with the server and requests the use of those resources and functionality, e.g. email

peer-to-peer: a type of decentralized and distributed network system in which individual nodes in the network called peers act as independent agents that are both suppliers and consumers of resources, this is in contrast to the centralized client-server model

representational state transfer (REST): a hybrid architectural style for distributed hypermedia systems that is derived from several other network based architectural styles and characterized by uniform connector interface, e.g. WWW

3.1.18 Architectural Styles Quiz

Android OS: heavily based on the generation and handling of events thus is mostly an event-driven system, also contains some elements of publish-subscribe which can register for events of interest

Skype: mainly peer-to-peer with minimal elements of client-server

WWW: based on REST, derived from other network based architectural styles such as client-server

Dropbox: client-server

3.1.19 P2P Architectures

Two representative examples are (1) Napster and (2) Skype.

3.1.20 Napster

In its first incarnation Napster was a peer-to-peer file sharing system mostly used to illegally share MP3s. A typical sequence of events for Napster is:

1. Peer A starts by registering with the peer & content directory, so does Peer B later
2. Peer A requests a song
3. the peer & content directory looks up its index and finds that Peer B has the song, it then sends Peer A a handle that Peer A can use to connect directly to Peer B
4. Peer A sends the request to Peer B, and Peer B starts sending the content to Peer A

The peer & content directory is a single point of failure, and is likely to cause problem when the number of peers grows too large.
Therefore Napster is a hybrid architecture with both client-server and peer-to-peer elements. Actually in real world nontrivial architectures it is very common to see multiple styles used in the same system.

3.1.21 Skype

Skype is a much more decentralized system than Napster. Only the login server adopts the client-server architecture. The skeleton of the peer-to-peer architecture of Skype is composed of super nodes, which are highly reliable nodes with high bandwidth that are not behind a firewall and that runs Skype regularly—nodes that shut down Skype occasionally will not qualify as super nodes. Note that super nodes are not owned by Skype. They are regular nodes that get promoted by Skype to super nodes and that know about each other. A typical sequence of events for Skype is:

1. Peer A contacts its super node A
2. based on its knowledge of the Skype network super node A routes the communication from Peer A to super node B of Peer B
3. super node B in turn routes the communication to Peer B

If the link between super nodes A and B were to go down, then Skype will automatically reroute the communication through super node C which will in turn reroute it to super node B. Hence Peer A and B will still be connected. This is exactly what happens when you are talking over Skype, the quality of the communication degrades and you are reconnected. Although this architecture is more effective than the Napster's one, it is not without problems. A few years ago Microsoft published a critical patch that required a reboot to be installed, which caused a large number of super nodes went down roughly at the same time. Skype's algorithm for determining super nodes didn't have enough nodes to work with, so the whole system crashed. Actually more recently Skype ditched peer-to-peer super nodes altogether.

3.1.22 Takeaway Message

- a great architecture is a ticket to a successful project
- a great architecture reflects a deep understanding of the problem domain
- a great architecture is likely to combine aspects of several simpler architectures

When defining a software architecture, you should innovate only as much as you need to and reuse as much as you can. By doing so you will be able to avoid reinventing the wheel and suitably combine existing solutions appropriately to come up with an effective overall solution.

3.2 Lecture 2 A Tale of Analysis and Design

3.3 Lecture 3 Design Patterns

3.3.1 Lesson Overview

Design patterns can support design activities by providing general reusable solutions to commonly occurring design problems. Similar to architectural styles design patterns can help developers build better designed systems by reusing design solutions that worked well in the past and by building on those solutions.

3.3.2 History of Design Patterns

1977: Christopher Alexander introduces the idea of patterns: successful solutions to problems

1987: Cunningham and Beck leveraged Alexander's idea in the context of an OO language

1987: Gamma's dissertation on the importance of patterns and how to capture them

1992: Jim Coplien compiled a catalog of C++ items which are some sort of patterns and listed this catalog of patterns in his book titled *Advanced C++ Programming Styles and Idioms*

1994: the gang of 4 (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) collaborated to publish the famous book *Design Patterns: Elements of Reusable Object Oriented Software*

3.3.3 Patterns Catalog

There are five main classes of patterns contained in the book:

fundamental patterns: the basic patterns

creational patterns: the patterns that support object creation

structural patterns: patterns that help compose objects i.e. put objects together

behavioral patterns: patterns mostly focusing on realizing interactions among different objects

concurrency patterns: more related to concurrency aspects

3.3.4 Pattern Format

We will focus on four essential elements of a design pattern:

name:

intent:

applicability: the list of situations or contexts in which the pattern is applicable

structure: the classes or the objects involved in the pattern, in addition to that structure also describes the relationships, responsibilities, and collaborations among these classes or objects

sample code: examples that illustrate the use of patterns

3.3.5 Factory Method Pattern

intent: it allows for creating objects without specifying their class by invoking what we call a *factory method*—a method whose main goal is to create class instances

applicability: applicable in cases in which

- a class cannot anticipate the type of object it must create, i.e. the type of an object is unknown at compile time (e.g. frameworks which only know about interfaces and abstract classes)
- a class wants its subclasses to specify the type of objects it creates
- a class needs control over the creation of its objects (e.g. when there is a limit on the number of objects that can be created)

structure: using the UML notation the structure includes three classes: the Creator, the ConcreteCreator, and the Product

participant: the classes are:

Creator: provides the interface for the factory model

ConcreteCreator: provides the method for creating actual objects

Product: the object created by the factory method

3.3.6 Factory Method Pattern Example

```
int imageType = getImageType(inputStream);
switch(imageType){
    case ImageReaderFactory.GIF
        return new GifReader(inputStream);
    case ImageReaderFactory.JPEG
        return new JpegReader(inputStream);
}
```

3.3.7 Strategy Pattern

Strategy pattern allows for defining a family of algorithms, encapsulating them into separate classes with each algorithm in one class and making these classes interchangeable, but providing a common interface for all the encapsulated algorithms.

intent: allow for switching between different algorithms for accomplishing a given task, e.g. choosing different sorting algorithms to meet different space-time tradeoff requirements

applicability: not only when we have different variants of an algorithm, but also when we have many related classes that differ only in their behavior

structure: there are three types of participants: the Context, the Algorithm, the ConcreteStrategy

participant: the classes are:

Context: interface to the outside world, maintains a reference to current algorithm and allows for updating this reference at runtime

Algorithm/Strategy: common interface for the different algorithms

ConcreteStrategy: actual implementation of the algorithm/strategy

3.3.8 Strategy Pattern Example

program: input = text file ⇒ output = filtered file

filter: four choices

- no filtering
- only words that start with "t"
- only words longer than 5 characters
- only words that are palindromes

Filter is selected as follows:

```
context.changeStrategy(new StartWithT());
context.filter(filename);
context.changeStrategy(new LongerThan5());
context.filter(filename);
context.changeStrategy(new Palindrome());
context.filter(filename);
```

The context.filter method is implemented as follows:

```
StringTokenizer words = new StringTokenizer(buffer);
while (words.hasMoreTokens()){
    String word = words.nextToken();
    if (strategy.check(word)){
        System.out.println(word);
    }
}
```

and the strategy.check method is implemented as follows:

```
class StartWithT implements CheckStrategy{
    public boolean check(String s){
        if (s == null || s.length() == 0){
            return false;
        }
        return s.charAt(0) == 't';
    }
}
```

3.3.9 Other Common Patterns

visitor pattern: a way of separating an algorithm from an object structure on which it operates, a practical result of this separation is the ability to add new operations to existing object structures without modifying the structures, the typical usage is when you are visiting a graph or a set of objects

decorator pattern: a wrapper that adds functionality to a class, it not only reproduces the functionality of the original class but also adds some functionality implemented using the services of the original class, a nice property of the decorator pattern is that it is stackable—you add decorators on decorators on decorators

iterator pattern: access elements of a collection without knowing the underlying representation

observer pattern: notify dependents when an object of interest changes by registering the dependents to let the system know that they are interested in changes in this object, a push notification instead of a pull notification, e.g. a folder in a file system

proxy pattern: a surrogate controls access to an object, all requests to the object and all responses from the object will go through the proxy, in some sense the proxy allows masking some of the functionalities of the object behind the proxy

3.3.10 Choosing a Pattern

One approach follows the following sequence of steps:

1. understand design context
2. examine the patterns catalog
3. identify and study related patterns
4. apply suitable pattern

However there are also pitfalls of using patterns:

- selecting wrong patterns to make the design worse
- abusing patterns ending up with a more complicated design

3.3.11 Choosing a Pattern Quiz

To design a class that can have only one instance we can use the factory method pattern:

```
public class Singleton{
    private static Singleton instance;
    private Singleton(){ }
    public static Singleton factory(){
        if (instance == null){
            instance = new Singleton();
        }
    }
}
```

3.3.12 Negative Design Patterns

Negative design pattern, also called anti-patterns and bad smells, are basically guidelines on how not to do things. They enable recurring design defects to be avoided.

3.4 Lecture 4 Unified Software Process

3.4.1 History

In 1997 Rational defined six best practices for modern software engineering:

1. develop in an iterative way with risk as the primary iteration driver
2. managing requirements, updating them, and keeping traceability information between them and other software artifacts
3. employ a component-based architecture, having a high-level design that focuses on cooperating cohesive and highly decoupled components
4. model software visually, using UML visual diagrams extensively to make artifacts easier to understand and agreed upon among stakeholders
5. continuously verify quality after each iteration
6. change management and control

These six practices were the starting point for the development of the Rational Unified Process or RUP.

3.4.2 Key Features of RUP

There are three key features of RUP:

- the RUP is a software development model, hence it defines an order of phases that should be followed in the software development process, and prescribes transition criteria i.e. when to go from one phase to the next
- the RUP is component based, hence a software system is defined and built as a set of software components, and there must be well-defined interfaces between these components

- the RUP is tightly related to UML, it relies extensively on UML for its notation and with respect to its basic principles

The three main distinguishing aspects of the RUP are (1) use-case driven (2) architecture-centric (3) iterative and incremental.

3.4.3 UML Quiz

The difference between use case and use case model is that, a use case model is a set of use cases. Use case diagrams are used for (1) prioritizing requirements (2) requirement elicitation (3) test case design without code (4) effort estimate (5) allowing customers to assess requirements.

3.4.4 Use-Case Driven

Generally speaking a system performs a sequence of actions in response to user input. Use cases capture this interaction and answer the question "what is the system supposed to do for each user?" In the RUP use cases are used to support each of its five phases: requirements engineering → design → implementation → verification and validation → maintenance.

3.4.5 Architecture-Centric

While use cases define the functionality, architecture defines the form—how the system should be structured to provide the functionality. In the RUP a software architecture is defined through the following iterative process:

1. create a rough outline of the system independently from the functionality
2. use key use cases to define main subsystems of the architecture
3. refine the architecture using additional use cases

3.4.6 Iterative and Incremental

The lifetime of a RUP consists of a series of cycles which can also be called increments. Each cycle results in a product release which can be internal or external. More precisely each cycle terminates with a product release that includes a complete set of artifacts for the project, including code, manuals, use cases, test cases, non-functional specification etc.
Each cycle involves all of the five main phases of software development and is divided into four phases: inception → elaboration → construction → transition. Within these individual phases there may be multiple iterations. Each iteration corresponds to a group of use cases that are selected so as to deal with the most important risks first, and then continue in the following iterations with less and less risky ones so that each iteration extends the functionality beyond the previous iteration.

3.4.7 Cycle Example

Each cycle focuses on a subset of use cases and the final product for that cycle will be a product that realizes those use cases. Nonetheless there is a little bit of overlap among cycles.

3.4.8 Phases within A Cycle

The four phases within a cycle are related to traditional activities of software development as follows:

business modeling: mainly performed in the inception phase and a bit in the elaboration phase

requirements engineering: starts in the inception phase, mainly performed in the elaboration phase, and continue albeit to a lesser extent throughout the remaining phases up until the end of the transition phase

analysis and design: mainly performed in the elaboration phase, but a considerable amount also continues in the construction phase, it then phases out with very little in the transition phase

implementation: mainly performed in the construction phase

test: performed throughout most phases with peaks in some specific point e.g. at the end of some iterations

deployment: mainly performed in the transition phase, which is the phase that has to do with deployment and then maintenance

3.4.9 Iterations

In almost every iteration developers perform the following activities:

identify relevant use cases: identify which pieces of functionality this iteration will develop

create design: the set of use cases plus the architectural guidelines will result in a design for the selected use cases

implement the design: result in a set of software components

verify code against use cases: make sure that the components satisfy the use cases

release a product: in many cases the release will be just an internal release or maybe just go to some of the stakeholders so that they can provide feedback on that

3.4.10 Iterative Approach Quiz

The benefits of an iterative approach are:

- give developers early feedback which
    1. increases the project tempo
    2. keeps developers focused as it is easier to keep focused with a short-term deadline
    3. provides developers with immediate reward
    4. minimizes the risk of developing wrong system
- accommodate evolving requirements—incorporate new requirements and realize a few requirements at a time starting from the most risky ones

It doesn't improve planning though because the number of iterations is hard to predict thus it is difficult to make a natural plan with an iterative approach.

3.4.11 Inception Phase

Since phases are fundamental aspects of the RUP, we will discuss for each phase

- what it is
- what it produces
- how is the result of the phase suppose to be assessed
- what are the consequences of this assessment

The inception phase goes from the idea of the product to the vision of the end product. It involves delimiting the project scope and making the business case for the product presented. Specifically this phase answers three main questions:

- what are the major users/actors and what will the system do for them, this will produce an initial use-case model where only a few use-cases are represented and described
- what could be an architecture for the system, this will produce an initial architecture that describes the most crucial subsystems
- what is the plan and how much will it cost, this will identify the main risks for the project and also produce a rough plan with estimates for resources, initial planning for the phases and dates and milestones

Specifically the inception phase generates several deliverables:

vision document: general vision of the core project's requirements, key features, and main constraints

initial use-case model: an initial set of use cases that will be later refined

initial project glossary: describe the main terms using the project and their meaning

initial business case: include business context and success criteria

initial project plan: phases, iterations, roles of the participants, schedule, and initial estimates

risk assessment: describe the main risks and countermeasures for these risks

prototype (optional): prototypes to address some specific risks, or to show some specific aspect of the system of which we are unsure to the stakeholders etc.

Evaluation criteria that conclude the inception phase include:

- stakeholder concurrence on the scope, definition, and cost/schedule estimates for the projects
- requirements understanding as evidenced by the fidelity of the primary use cases identified
- the credibility of the cost/schedule estimates, priorities, risks and their countermeasures, and the development process followed
- the depth and breadth of any prototype that was developed

If the outcome is considered to be inadequate with respect to one or more of these criteria, the project may be canceled or considerably re-thought.

3.4.12 Elaboration Phase

There are four main goals and activities for the elaboration phase:

- analyze problem domain
- establish architectural foundation
- eliminate the highest risk elements i.e. address the most critical use cases
- refine the plan of activities and estimates of resources

The artifacts produced by this phase include:

almost complete use-case model: all use cases and actors identified and most use case descriptions developed

supplementary requirements: all the requirements that are not associated with a use case, including in particular all non-functional requirements such as security, reliability, maintainability etc.

software architecture: refine the initial architecture produced by the inception phase until we get a software architecture which is complete

design model, test cases, executable prototype: lower-level design for the system

revised project plan and risk assessment: refine the various estimates and pieces of information in the project plan, update risk assessment document

preliminary user manual: describe how the system can be used and should be used

The evaluation criteria for the elaboration phase include:

- are vision and architecture stable?
- does the prototype show that the major risks identified have been resolved or at least addressed?
- is the project plan sufficiently detailed and accurate?
- do all stakeholders agree that the vision can be achieved with the current plan?
- is the actual resource expenditure vs. the planned expenditure acceptable?

Again if the outcome is considered to be inadequate with respect to one or more of these criteria, the project may be canceled or considerably re-thought.

3.4.13 Construction Phase

The construction phase is basically the phase in which most of the actual development occurs. All the features considered are developed and thoroughly tested. In general the construction phase is the phase in which there is a shift in emphasis from intellectual property development to product development, from ideas to products.
The construction phase produces the following deliverables:

- all the use cases realized with traceability information from the use cases to the artifacts
- software product integrated on all the needed platforms

- complete system tests results
- user manual
- complete set of artifacts including design, code, test case etc.

Roughly speaking we can consider the product that is produced at the end of this phase a typical *beta release*.
The evaluation criteria for the construction phase include:

- is the product stable and mature enough to be deployed to users?
- are the stakeholders ready for the transition into the user community?
- are we ready to go from development to production?
- are the actual resource expenditures vs. the planned expenditures still acceptable?

Unless we can answer all these questions in a positive way, the transition phase might be postponed by one release.

3.4.14 Transition Phase

Transition phase has mainly to do with deployment and maintenance of a system. The main activities in the transition phase are:

- maintenance after deployment leading to new release, including
  - corrective maintenance for bug reports
  - perfective maintenance for feature requests
  - adaptive maintenance for environment change (OS patch/upgrade)
- training customer services and providing help-line assistance
- a new cycle may start

The transition phase produces the following deliverables:

- a complete project with all the artifacts
- product in use
- lessons learned
- plan for next release

The evaluation criteria for the transition phase include:

- is the user satisfied?
- are the actual resource expenditures vs. the planned expenditures still acceptable?

Problems with this milestone might lead to further maintenance of the system and a new release to address the issues.

3.4.15 Phases and Iterations

It is very important to understand the relation between the RUP and the traditional software engineering activities. Although there is normally one main phase for each activity, the activities really span multiple phases. This allows you, in subsequent iterations, to address problems that came up in previous iterations.

4 SOFTWARE VERIFICATION AND VALIDATION

4.1 Lecture 1 General Concepts

4.1.1 Failure, Fault, and Error

failure: an observable incorrect behavior of the software, conceptually related to the behavior of the program rather than its code

fault: also called bug, an incorrect piece of code, a necessary but not sufficient condition for the occurrence of a failure

error: the cause of a fault, usually a human error

4.1.2 Verification

The four mainstream ways to verify a software system are:

testing: also called dynamic verification, exercising the system to try to make it fail, a test case is a pair of input i from the input domain D and expected output o from the output domain O, a test suite is a set of test cases

static verification: try to identify specific classes of problems in the program, unlike testing it considers all possible inputs for the program, i.e. it is complete

inspection: also called reviews or walkthroughs, unlike the previous techniques inspections are a human intensive activity, more precisely they are a manual group activity in which several people from the organization that developed the software look at the code or other artifacts and try to identify defects in these artifacts, inspections have been shown to be quite effective in practice and that is the reason why they are used quite widely in the industry

formal proof of correctness: given a formal specification that formally defines and specifies the expected behavior of the program, a formal proof of correctness proves that the program actually implements the program specification and it does that through a sophisticated mathematical analysis of the specifications and of the code

Although we will discuss all four approaches, we will spend most of our time on software testing since software testing is the most popular and most used approach in industry.

4.1.3 Pros and Cons of Approaches

testing: the pro and con are

pros: it does not generate false positives

cons: it is incomplete because it can consider only a tiny fraction of the program's behaviors

static verification: the pro and con are

pro: it considers all program behaviors

con: it considers not only the possible behaviors but also the impossible behaviors hence it can generate false positives

inspection: the pro and con are

pro: systematic, resulted in a thorough analysis of the code

con: informal, subjective depending on the people performing the inspection

formal proof of correctness: the pro and con are

pro: provide strong guarantees

con: need a complete mathematical description of the expected behavior which is rarely available, and it is complex to build one and expensive to prove that the program corresponds to a specification (mathematics specialist)

4.1.4 Testing Introduction

Testing means executing the program on a tiny sample of the input domain. The two important aspects of testing are:

dynamic technique: the program must be executed in order to perform testing

optimistic approximation: it is done under the assumption that the behavior with any other input is consistent with the behavior shown for the selected subset of input data

As Goodenough and Gerhart's claim "a test is successful if the program fails" implies, testing cannot prove the absence of errors but only reveal their presence. If a set of tests does not produce any failure, we are either in the extremely unlikely case of a correct program, or in the very likely situation of a bad set of tests that are not able to reveal failures of the program.

4.1.5 Testing Granularity Levels

unit testing: the testing of the individual units or modules in isolation

integration testing: the testing of the interactions among different modules, performed according to different strategies depending on the order in which the modules are integrated, and on whether we integrate one module at a time or multiple modules together all at once (big bang integration testing)

system testing: the testing of the complete system and include both functional and non-functional testing

functional test: the test that aims to verify the functionality provided by the system

non-functional test: the test that targets at non-functional properties of the system including performance tests, load tests, robustness tests, in general non-functional tests will try to assess different qualities of the system such as reliability, maintainability, usability

acceptance testing: the validation of the software against the customer requirements

regression testing: the type of testing or retesting performed every time that we change our system to ensure that the changes behave as intended and that the unchanged code is not negatively affected by the modification by these changes (regression error), regression testing is one of the main causes why software maintenance is so expensive, hence developers try to automate as much as possible regression testing

4.1.6 Alpha and Beta Testing

All the testing levels above are developers' testing—testing performed either within the testing organization, or by somebody who is doing like third-party testers on behalf of the testing organization. After developers' testing we have

alpha testing: the testing performed by distributing a software system ready to be released to a set of users that are internal to the organization that developed the software

beta testing: the software is released to a selected subset of users outside the organization that developed the software

4.1.7 Black- and White-Box Testing Introduction

black-box testing: the testing that considers the software as a closed box, the testing that

- based on a description of the software specification
- cover as much specified behavior as possible
- cannot reveal errors due to implementation details

white-box testing: the testing that looks at the code, the testing that

- based on the code
- cover as much coded behavior as possible
- cannot reveal errors due to missing paths (a missing path is a part of a software specification that is not implemented)

4.1.8 White-Box Testing Example

Black-box testing and white-box testing are complimentary techniques.

4.2 Lecture 2 Black-Box Testing

4.2.1 Overview

There are several advantages in using black-box/functional testing:

- focus on the input domain of the software, so that we can use it to make sure that we cover the important behaviors of the software
- no need for the code, so that we can perform early test design
- catch logic defects because we derive test cases from the description of what the software should do i.e. its logic
- applicable at all granularity levels

4.2.2 Systematic Functional Testing Approach

Black-box testing starts from a functional specification of the software. The final result of black-box testing is a set of test cases, a set of actual inputs and corresponding outputs. A systematic approach to derive test cases from a functional specification is to simplify the overall problem by dividing the process into elementary steps, in particular we will perform three main steps:

1. identify independently testable features
2. identify relevant inputs
3. derive test case specifications that can be used to generate test cases

Proceeding by this steps has many advantages:

- it allows for the decoupling different activities
- it allows for dividing brain-intensive steps from steps that can be automated
- it allows for monitoring the testing process (e.g. whether you are generating too many test cases)

It does not make sense to just try to devise test cases for all the features of the software at once. A much better way is to identify independently testable features and consider one of them at a time when generating tests.

4.2.3 Test Data Selection

The problem of identifying relevant inputs for some software or some feature of it is called *test data selection*. We will focus on two different ways of doing it.

4.2.4 Why Not Random Testing

Random testing has several advantages:

- it picks inputs uniformly
- it makes no preferences, all inputs are considered equal
- it eliminates designer bias—the developer might develop code assuming a given behavior of the user and we may write tests making the same assumptions

The disadvantage is that bugs are very scarce in the input domain, thus it is very unlikely that just by picking randomly we will be able to get to the bugs.

4.2.5 Partition Testing

Failing inputs are generally very sparse in the input domain, but they tend to be dense in some parts of the domain. To leverage this the input domain is split into partitions—areas of the domain that are treated homogeneously by the software. Partition testing consists of

- identify partitions of the input domain
- select inputs from each partition

By doing so we can dramatically increase our chance to reveal faults in the code.

4.2.6 Partition Testing Example

- Without testing negative string size is a kind of designer bias.
- Treating different parts of the input independently also helps partitioning.
- Select representative instead of random value from each partition.

4.2.7 Boundary Values

The basic idea is that errors tend to occur at the boundary of a (sub)domain, because they are the cases that are less understood by the developers, e.g. the last iteration of a loop or a special integer like zero. For our split program example possible boundary values for the subdomain {size > 0 and str with length in [size, size $\times 2$]} are size $= 1$ and size $=$ MAXINT and a string with length size.

4.2.8 Deriving Test Case Specifications

Test case specification defines how the values should be put together when testing the system. One possible way of combining the parameters is simply to take the Cartesian product of the values for the first parameter and the values for the second parameter.

4.2.9 Category-Partition Method

The category-partition method is a method for going from a specification to a set of test cases by following six steps:

1. identify independently testable features
2. identify categories
3. partition categories into choices
4. identify constraints among choices
5. produce and evaluate test case specifications
6. generate test cases from test case specifications

4.2.10 Identify Categories

Categories are characteristics of each input element for which we leverage our domain knowledge, e.g. in our split program example (length, content) for str and value for size. The characteristics are somehow subjective thus there is more than one possibility.

4.2.11 Partition Categories into Choices

Choices are interesting subdomains for each category. For our split program example

- 0 and size $- 1$ are interesting cases of the length of str
- spaces and special non-printable characters are interesting cases of the content of str
- $= 0$, < 0, MAXINT are interesting cases of the value of size

4.2.12 Identify Constraints among Choices

We identify these constraints in order to

- eliminate meaningless combinations of inputs
- reduce the number of test cases

There are three types of properties:

if: that the content of str contains special character is possible only if the length of str is nonzero

error: size < 0

single: size = MAXINT is used in only one combination

4.2.13 Produce and Evaluate Test Case Specifications

The step of producing and evaluating test case specifications can be completely automated given the results of the previous steps which produces *test frames*—the specification of a test. Test frames are normally identified by a sequence number and specify the characteristics of the inputs for that test. One advantage of this approach is that we can easily use it to assess how many test cases we will generate with the current list of category choices and constraints. If the number is too large then we can add additional constraints and reduce it.

4.2.14 Generate Test Cases from Test Case Specifications

This step mainly consists of simple instantiation of test frames and its final result is a set of concrete tests.

4.2.15 Model-Based Testing

The step of identifying relevant inputs and combining them to generate test case specifications can also be done through the construction of a model, where a model is an abstract representation of the software under test. There are many possible models that we can use and we will focus on a specific kind of model.

4.2.16 Finite State Machines

At a high level a **state machine** is a graph in which

nodes: represent states of the system

edges: represent transitions between states

edge labels: represent events and actions

The steps of building a finite state machine from a specification are:

1. identify the system's boundaries, and the input and output to the system
2. identify within the system's boundaries the relevant states and transitions

4.2.17 Finite State Machines Example

The way to go from the finite state machine representation to a set of test cases is to cover the behaviors represented by the finite state machine, and we can decide how to cover them. For example we may want to cover all the states, or all the transitions. For the latter we can first cover all the states and look for transitions that are not covered, then generate another test case to cover them or extend an existing one. The bottom line here is that it is much harder to build a set of test cases that will cover the behavior of an informal description. But by going through a model, a finite state machine for that description, we can in a much easier way see what the behaviors of interest of the system are and try to cover them. That is again in the spirit of breaking down a complex problem into smaller steps that we can better manage, which results in a more efficient and effective testing.

4.2.18 Finite State Machine Considerations

applicability: testing based on final state machines is a very general approach that we can apply in a number of contexts, and in particular if you are working with UML you have state machines for free—state charts are a special kind of state machine

abstraction is key: the bigger the system the more you have to abstract if you want to represent it with a model, and in particular with the final state machine, the more you represent the more complex your system is going to be and the more thorough your testing is going to be but also more expensive; the less you represent the less expensive testing is going to be but also testing might not be as thorough as it would be otherwise, so you have to find the right balance

many other approaches: decision tables, flow graphs, and even historical models—models that can guide your testing based on the problems that occurred in your system in the past

4.3 Lecture 3 White-Box Testing

4.3.1 Overview

There is one basic assumption behind the idea of white-box/structural testing—executing the faulty statement is a necessary condition for revealing a fault. The main advantages of white-box testing are that

- since it is based on the code, the quality of the testing can be (1) measured objectively and (2) measured automatically
- it can be used to compare test suites
- it allows for covering the coded behavior—if there is some mistake in the code and is not obvious by looking at the specification of the code, white-box testing might be able to catch it

There are many different kinds of white-box testing such as (1) control-flow based (2) data-flow based (3) fault based. In this lesson we will focus on control-flow based white-box testing.

4.3.2 Coverage Criteria Introduction

Coverage criteria are defined in terms of test requirements, which are the entities in the code that we need to execute in order to satisfy the criteria. Normally when we apply a set of coverage criteria the result is a set of test specifications and test cases.

4.3.3 Statement Coverage

The criterion **statement coverage** is characterized by two aspects

test requirements: statements in the program, i.e. the goal of branch coverage is to execute all the statements in the program

coverage measure: how we measure coverage for that criterion, a good measure is the ratio of the number of executed statements over the total number of statements in the program

Statement coverage is the most widely used type of coverage criterion in industry. The typical coverage target is 80–90%.

4.3.4 Control Flow Graphs

A control flow graph or CFG is just a representation for the code. It represents statements with nodes and the flow of control within the code with edges.

4.3.5 Branch Coverage

Branch coverage is sometimes also called decision coverage.

test requirements: branches in the program, i.e. the goal of branch coverage is to execute all of the branches in the program

coverage measure: the ratio of the number of executed branches over the total number of branches in the program

In a CFG branches are outgoing edges from a decision point. Note however that that 100% coverage does not provide any guarantee of finding the problems in the code. It is only that by testing more thoroughly we have more chances of finding a problem in the code. Nonetheless we tested more thoroughly when we go from statement coverage to branch coverage. This pertains to the concept of **test criteria subsumption**—one test criterion subsumes another criterion when all the test suites that satisfy that criterion will also satisfy the other one. If we identify a test width that achieves 100% branch coverage, the same test width will also achieve, necessarily, 100% statement coverage, hence branch coverage subsumes statement coverage. Because branch coverage is a stronger criterion than statement coverage as there is no way to cover all the branches without covering all the statements. What it also means is that in general it is more expensive to achieve branch coverage than to achieve statement coverage, because achieving branch coverage requires the generation of a larger number of test cases. Conversely it is not true that any test suite satisfying statement coverage will also satisfy branch coverage (e.g. no statement for a branch).

4.3.6 Condition Coverage

We can make each condition true and false instead of just considering the whole predicate, which is **condition coverage**.

test requirements: individual conditions in the program

coverage measure: the ratio of the number of conditions that are both true and false over the total number of conditions in the program

4.3.7 Branch and Condition Coverage

Condition coverage has with no relationship of subsumption with either branch coverage or statement coverage, which means that the criteria are not comparable. Because it is possible that despite the fact that we are exercising all possible values for the conditions, the overall predicate is always true resulting in $< 100\%$ branch coverage. Therefore branch coverage and condition coverage are normally considered together, and the resulted criterion is called **branch and condition coverage**.

test requirements: branches and individual conditions in the program

coverage measure: computed considering both coverage measures

4.3.8 Test Criteria Subsumption

Branch and condition coverage subsumes both branch coverage and condition coverage.

4.3.9 B&C Coverage Quiz

Testing all true-false combinations of the conditions inside each predicate is called **multiple condition coverage**, which is expensive to the point of being impractical.

4.3.10 MC/DC Coverage

Modified condition/decision coverage or MC/DC is often required for safety critical applications e.g. FAA. The key idea is to test only the important combinations of conditions so as to limit the testing costs. The way in which it works is by extending branch and condition coverage with the requirement that each condition should affect the decision outcome independently. Thus MC/DC subsumes branch and condition coverage.
As an example consider a AND b AND c, for which we only need to consider the following four cases:

a	b	c	outcome
true	true	true	true
false	true	true	false
true	false	true	false
true	true	false	false

4.3.11 Other Criteria

path coverage: all the paths in the program, incredibly expensive

data-flow coverage: covering the statement in which the content of some memory location is modified and the statement in which the content of the same memory location is used

mutation coverage: evaluate the goodness of tests by generating enough variation of the program (e.g. $k > 0$ to $k \geq 0$) to assess how good are the tests at distinguishing the original program and the mutants, because we change the code based on the way we expect to introduce errors in the code, the more the tests can identify mutants the better they are at identifying real faults

Note that

- Since MC/DC is a smarter way of doing multiple condition coverage, exercising a subset of the elements of the multiple condition coverage, multiple condition coverage subsumes MC/DC.

- Path coverage subsumes branch coverage because we cover all the paths in the program we necessarily cover all the branches. However it doesn't subsume multiple condition coverage, MC/CD, or branch and condition coverage, because they have additional requirements involving the conditions of the predicate that path coverage does not have.
- Data-flow coverage criteria and mutation coverage criteria have really no relation with the other criteria, because they look at different aspects of the code thus are not comparable.

These criteria can be grouped into two categories:

practical: including MC/DC, branch and condition coverage, branch coverage, condition coverage, statement coverage, data-flow coverage, they are not too expansive to be used in real scenarios

theoretical: including multiple condition coverage, path coverage, mutation coverage, they are useful in theory from the conceptual standpoint but are too expansive to be applicable in practice

4.3.12 Review Quiz

All the coverage criteria are just approximation of a complete test or exhaustive testng. Thus test is a best effort kind of activity. Test can only reveal issues. It can only reveal problems. It can never show the absence of problems.
Any non-trivial program contains dead or unreachable code that no matter how well we test our system we will never be able to exercise. Because for example defensive programming, or developing for future releases there might be pieces of code that are added but are not activated. If there is some unreachable code, we will never be able to reach 100% code coverage, and we need to take that into account when we try to achieve a given coverage target.

4.3.13 Summary

objective: white-box testing works on a formal model, the code itself are models derived from the code, so we don't need to make subjective decision

comparable: coverage criteria allows us to compare different test suites, because we can measure the coverage achieved by a test suite

two classes: there are two broad classes of coverage criteria—practical criteria that we can use vs. theoretical criteria that are interesting from a conceptual standpoint

fully automatable: there are tools that can instrument it automatically and measure the level of coverage that can be achieved with your test

4.4 Lecture 4 Agile Development Methods

4.4.1 Lesson Overview

The agile development process, also called test-driven development, is better suited for the context in which changes are the norm and we need to adapt fast. In particular we will discuss two processes that apply the principles of agile software development and that are commonly used in industry—Extreme Programming, also called XP, and Scrum.

4.4.2 Cost of Change

As Boehm said the cost of change grows exponentially with time, we should discover errors early before they become expensive, which in turn means doing a lot of upfront planning. Because models are cheaper to modify in code, we are willing to make large investments in upfront analysis and design models, and only after we have built and checked these models we are going to go ahead and build the code. In other words we are following a waterfall mentality. However the cost of change becomes flatter in the past 30 years due to greater and cheaper computational power.

4.4.3 Agile Software Development

There are a few interesting consequences if the cost of change is flat:

- upfront work becomes liability because we pay for speculative work, some of which is likely to be wrong and some of which we are likely to undo
- if there is ambiguity and volatility in requirements, then it is good to delay

In summary there is value in waiting, because time answers questions and removes uncertainty. This and other considerations led to the birth of agile software development. In a nutshell agile methods aim at flat cost and a decrease in traditional overhead by following a set of important principles:

- focus on the code rather than the design to avoid unnecessary changes
- focus on people rather than process and make sure to reward people
- based on an iterative approach to deliver working software quickly and to be able to evolve just as quickly based on feedback
- involve customer throughout the development process for customer feedback
- the expectation that requirements will change
- the mentality of simplicity—simple design, simple code etc.

4.4.4 Extreme Programming (XP)

According to Kent Beck XP is a lightweight methodology for small to medium sized teams developing software in the face of vague or rapidly-changing requirements. XP is

lightweight: it doesn't overburden the developers with an invasive process thus process is kept to a minimum

humanistic: people—developers and customers—are at the center of the process

discipline: it includes practices that we need to follow

software development: software development is a key point of the whole method

In XP we need to adopt a mentality of sufficiency.

4.4.5 XP's Values, Principles, and Practices

The important values of XP are

communication: XP tries to keep the right communication flowing and it uses several practices to achieve that, its practices are based on and require information and in general share the information e.g. pair programming, user stories, customer involvement

simplicity: look for the simplest thing that works

feedback: occurs at different levels and is used to drive changes e.g. developers write test cases, estimate new stories from customers, and work together with customers to develop functional system test

courage: the courage to throw away code if it doesn't work, to change it if you find a way to improve it, to fix it if you find a problem, now that we can build and test systems very quickly we can be much braver than what we were before

XP's practices to achieve these values are (1) incremental planning (2) small releases (3) simple design (4) test first (5) refactoring (6) pair programming (7) continuous integration (8) on-site customer etc.

4.4.6 Incremental Planning

Incremental planning is based on the idea that requirements are recorded on use cases/stories that the customer provides. It consists of the following steps:

1. select user stories for current release, which stories exactly to include depends on the time available and on the priority
2. break stories into tasks, i.e. identify specific development tasks that need to be performed in order to realize the user stories
3. plan release
4. develop, integrate, test in an iterative way
5. release software
6. evaluate system and start planning for new release

4.4.7 Small Releases

Instead of having a big release at the end of a long development cycle, we try to release very often. There are many advantages to small releases and to release often:

sooner business value delivery: getting business value sooner in turn increases customer confidence and customer satisfaction

rapid feedback: enable adapting quickly to changes in the requirements

quick adaptation: to changes in the requirements

risk reduction: quick adaptation to requirement changes avoids going down the wrong path, and we know we are late right away

sense of accomplishment: for the developers

4.4.8 Simple Design

We should avoid creating a huge, complicated, possibly cumbersome design at the beginning of the project. We should have just enough design to meet the requirements, which means no duplicated functionality, and fewest possible classes and methods in general.

4.4.9 Test-First Development

Create unit tests for each such piece of functionality even before the functionality is implemented.

4.4.10 Refactoring

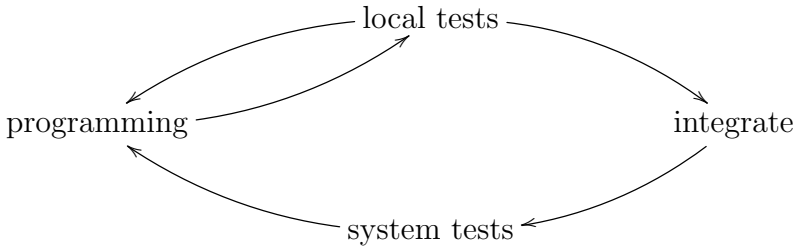
Refactoring means restructure a piece of code so that it becomes simple and maintainable. Developers are expected to refactor as soon as opportunities for improvement are found, which can happen before or after adding some code. Note that we should refactor on demand, and the goal is to keep the code simple and maintainable and not to over do it.

4.4.11 Pair Programming

Pair programming means that all production code is written with two people looking at one machine. They are working with one keyboard and one mouse by playing different roles at different times instead of just interfering and writing on each other's code. The two developers alternate between the role of programming and strategizing, where strategizing means thinking whether the wode would work, or what are the tests that are not there might not work, or whether the code can be made simpler, more maintainable, more efficient. There are studies that suggest that development productivity with pair programming is similar to that of two people working independently rather than cutting their productivity in half.

4.4.12 Continuous Integration

Continuous integration means means integrating and testing every few hours or a day at most to prevent problems from piling up or being discovered too late. Continuous integration is a cycle that starts with the developers' programming. Once the developers have a stable version of the code they will run the local tests. If the local tests fail, the developers will go back to programming to fix their code. This mini cycle will continue until all the local tests pass. At that point the developers can integrate their code with the code of other developers, and then they can run test for the integrated system. If the system tests fail, the developers will have to go back and modify their programming and again going through the cycle of running the local tests, integrating, and running the systems tests. Conversely if all the systems tests pass, then the code is good to go and it is integrated into the system, and it will be the problem of some other developers. If we do this every few hours or every day, we can find problems very early and we can avoid the situations in which we have many different changes coming from many different developers.



4.4.13 On-site Customer

On-site customer means literally the customer is an actual member of the team. That is the customer will sit with the team, and will bring requirements to the team and discuss the requirements with them. The typical objection to this practice is the fact that it is just impossible in the real world. The answer to that objection is that if the system is not worth the time of one customer then maybe the system is not worth building.

4.4.14 Requirements Engineering

In XP requirements are expressed as user stories. These are written by customers on cards. The development team then take these users stories and break them down into implementation tasks. These implementation tasks are then used as a basis for scheduling cost estimates. Given these estimates and based on their priorities the customer will choose the stories that will be included in the next release. At this point the corresponding card will be developed. There might be 50 to 100 user stories for a project with a few month's duration.

4.4.15 Testing Strategy

The basic principle is that testing is Coded confidence. Both the running and the checking of the tests has to be automated for all of this to work. There are two types of tests:

unit tests: created by the programmers looking at the task cards, check that the code has correctly implemented the described functionality, special cases, and even refactoring

system tests: also called acceptance tests, involve the customer providing test cases for their stories, and then the development team transforms them into actual automated tests

Unit tests run very quickly and run very frequently, while system tests run longer and run less frequently. They run every time the system is integrated according to the continuous integration cycle above.

4.4.16 Testing Strategy Quiz

- It is not true that because of pair programming, XP requires twice the number of developers.
- XP is based on subsequent iterations of the same cycle of story cards $\rightarrow$ selected user cards $\rightarrow$ task cards

4.4.17 Scrum

Scrum is another agile development process. There are three main kinds of actors in Scrum:

product owner/customer: responsible for expressing the product backlog—the list of things that have to be done, analogous to the user stories to be realized in XP—and to prioritize them by value

development team: responsible for delivering shippable increments to estimate the backlog items, normally self-organized consisting of four to nine people

Scrum master: the manager or supervisor, responsible for the overall Scrum process, remove obstacles, facilitate events, helps communications etc.

4.4.18 High-Level Scrum Process

product backlog: the single source of requirements for the process ordered by value, risk, priority, and necessity, it is a living list in the sense that backlog items can be added or removed by the product owner

sprint planning: the next increment/sprint is defined, the backlog items of interest are selected based on the order which are then converted into tasks and estimated

sprint backlog: the set of backlog items that will be completed during the next sprint

sprint: the actual iteration of the Scrum process, with a main part that lasts 2 to 4 weeks, and within this main part there are many daily Scrums that last 24 hours

- **daily scrum:** typically characterized by a 50-minute meeting at the beginning of the day for the team to sync, involving (1) discussing the accomplishments since the last meeting (2) producing a to-do list for the next meeting (3) an obstacle analysis proposing possible solutions
- **sprint review and retrospective:** at the end of the 2- to 4-week cycle, the sprint review normally consists of a 4-hour meeting, in which the product owner assesses the accomplishment for the specific sprint and the team discusses issues that were encountered and solved, there is typically a demo of the deliverable for that sprint, the product owner will also discuss the backlog, and together with the team they will decide what to do next
  conversely the retrospective focuses more on the process, the goal is discussing possible process improvements, to identify them, and if promising improvements are identified try to plan how to implement those improvements in the next iterations

potential shippable product increment: if the product increment is good enough as it reach the state in which it can be actually shipped, then the sprint will result in a release that can be deployed and used in production

Hence XP and Scrum are fairly similar, both implementing and enforcing the ideas, values, practices, and characteristics of the agile development process in general.

4.5 Lecture 5 Software Refactoring

4.5.1 Lesson Overview

Software refactoring is the process of taking a program and transforming it to make it easier to understand, make it more maintainable, and in general to improve its design. A bad smell in software is the indication that there might be something wrong with the code that might call for the application of refactoring.

4.5.2 Introduction

Refactoring is the process of applying transformation to a program so as to obtain a refactored program with an improved design but the same functionality as the original program. The main goal of refactoring is to keep the program readable, understandable, and maintainable as we evolve it, and to do this by eliminating small problems soon so that you can avoid big trouble later. A key feature of refactoring is that it is behavior preserving. Unfortunately in general there are no guarantees of behavior preserving, but what we can do is to test the code. We may just have to rerun the test cases after refactoring.

4.5.3 Reasons to Refactor

There are three main reasons behind refactoring:

requirements change: when the requirements change, we often need to change our design accordingly

design needs to be improved: need to add a new feature, want to make the code more maintainable, and also in general programmers don't come up with the best design the first time

sloppiness/laziness: create a method/class to recycle certain functionality instead of copy-and-paste

Even renaming a class is a refactoring.

4.5.4 History of Refactoring

Refactoring is especially important in the context of object-oriented languages, probably because the object-oriented features are well suited to make designs flexible and reusable, and encapsulation and information hiding make it easier to modify something without changing the functionality. In more recent times refactoring is becoming increasing popular due to agile development. Because refactoring is one of the practices that help making changes less expensive and therefore adapt to changing requirements and changing environments more quickly.
One of the first example of a specific discussion of refactoring is Opdyke's PhD thesis in 1990 discussing refactoring for SmallTalk. One of the milestones in the history of refactoring is Martin Fowler's book titled *Improving the Design of Existing Code* which we will follow in this lesson.

4.5.5 Types of Refactorings

There are many types of refactorings. We will elaborate on (1) collapse hierarchy (2) consolidate conditionals (3) decompose conditionals (4) extract method (5) extract class (6) inline class.

4.5.6 Collapse Hierarchy

After you apply a number of refactorings that move methods and fields up and down the class hierarchy, a subclass might become too similar to its superclass and might not be adding much value to the system. In this case it is a good idea to merge the classes together. That is exactly what the collapse hierarchy refactoring does.

4.5.7 Consolidate Conditional Expression

Sometimes the code contains a series of conditional checks in which each check is different, yet the resulting action is the same. In these cases the code could be improved by combining the conditionals using for example AND and OR as connectors so as to have a single conditional check with a single result. At that point you can also extract that conditional into a method and replace the conditional with a call. Extracting a condition and having a method instead of a condition can clarify your code by explaining why you are doing a given check rather than how you are doing it. For example

$$\text{if (seniority} < 2 \parallel \text{monthsDisabled} > 12 \parallel \text{isPartTime)}$$

can be consolidated into

$$\text{if (notEligibleForDisability())}$$

4.5.8 Decompose Conditionals

A conditional, if it is too complex, might tell you what happens but obscure why it happens. To address this issue we can

- transform the condition into a method and then replace the condition with a call to that method
- modify the then and else part of the conditional by extracting the corresponding code, making it into a method, and having a call to the method only in the then and else part of the conditional

For example

```
if (date.before(summer_start) || data.after(summer_end))
    charge = quantity × winterRate + winterSourceCharge;
else
    charge = quantity × summerRate;
```

can be refactored into

```
if (notSummer(date))
    charge = winterCharge(quantity);
else
    charge = summerCharge(quantity);
```

4.5.9 Extract Class

When a software system evolves, we might end up with classes that really do the work of more than one class because we keep adding functionality to the class. Therefore they are too big and too complicated. In this case it is normally a good idea to split the class into two by creating a new class and move there the relevant fields and methods from the original class so as to have two classes, each one implementing a piece of the functionality.

4.5.10 Inline Class

Inline class is the reverse of extract class, and this is kind of a general situation in the sense that it is often the case that the refactoring also has a reverse refactoring that does exactly the opposite. In this case the motivation for the refactoring is that during system evolution we might end up with one or more classes that do not do much. In this case what you want to do is to take the class that is not doing that much and move its features into another class and then delete the original class.

4.5.11 Extract Method

Extract method is one of the most commonly used refactoring. The starting point is a method that is too long and contains cohesive code fragments that really serve a single very specific purpose. What we can do in this case is to create a method using that code fragment and to replace the code fragment with a call to that method.

4.5.12 Exact Method Refactoring Quiz

It is appropriate to apply extract method refactoring when

- when there is duplicated code in two or more methods, in this case we want to factor it out and have the two methods called a third method that is created using the refactoring

- a method is highly coupled with a class other than the one where it is defined, in this case we extract part of the method coupled with the other class and move the extracted method to the class where it actually belongs to

When a class is too large, normally we use the extract class refactoring.

4.5.13 Refactoring Risks

- More complex refactorings may also introduce subtle faults which are normally called regression errors. One way to avoid that is to run tests every time you make a refactoring.
- Refactoring should be performed when it is needed. Do not over do it.
- We should be particularly careful when we are refactoring systems that are in production, because if you introduce a problem for a system in production then you have to issue a new version of the code.

4.5.14 Cost of Refactoring

manual work: in many cases refactoring involves quite a bit of manual work if you are doing some manual refactoring, and how much that costs depends on how well the operations on the source code are supported

test development and maintenance: refactoring relies heavily on testing and we might have to develop test cases specifically to check our refactoring, even if we have an existing test from e.g. some agile context, as we refactor we might need to update our test, hence refactoring not only requires the development of test cases but also maintaining the test cases

documentation maintenance: applying refactoring may involve changes in interfaces, names etc. and when we make this kind of changes we might need to update the documentation

4.5.15 When Not to Refactor

when the code is broken: refactoring is not a way to fix a code because by definition refactoring should maintain its functionality, if your code does not compile or does not run in a stable way it is probably better to rewrite it

when a deadline is close: refactoring may take a long time and therefore may introduce risks of being late for the deadline, and we don't want to introduce problems that may take time to fix right before a deadline

when there is no reason to: we should refactor on demand

Nonetheless bad smells tell us when to refactor without an indication that will tell us that it is time to refactor the code.

4.5.16 Bad Smells

Bad smells or code smells are symptoms in the code of a program that might indicate deeper problems. Bad smells are usually not bugs and don't prevent the program from functioning. They however indicate weaknesses in the design of the system that may make the code less maintainable, harder to understand etc. Just like refactorings there are also many possible different bad smells.

4.5.17 Bad Smell Examples

duplicated code: the same fragment of code or code structure replicated in more than one place, for which we can use extract method refactoring

long method: the longer the procedure the more difficult to understand and maintain, for which we can use extract method refactoring, decompose conditional refactoring etc.

large class: a class that contains too many fields and too many methods is just too complex to understand, for which we can use extract class refactoring

shotgun surgery: every time we make some change to the system we have to make many little changes all over the place to many different classes implying too much coupling between these classes and too little cohesion within these classes, for which we can use move method/field refactoring or inline class refactoring

feature envy: a method that seems more interested in a class other than the one it belongs to e.g. using a lot of public fields or calling a lot of public methods of another class, for which we can use extract method refactoring followed by move method refactoring to reduce the coupling

4.5.18 Bad Smell Quiz

- The fact the program takes too long to execute is not really a bad smell.
- That the method is too long is a typical example of bad smell.
- That every time we modify one method we also need to modify some other method is a typical example of bad smell.