SOFTWARE ENGINEERING

CO3001

LECTURE 2&3 – REQUIREMENTS ENGINEERING

Anh Nguyen Duc
Jingyue Li
Daniela Soares Cruzes

NTNU

Adapted from https://iansommerville.com/software-engineering-book/slides/

TOPICS COVERED

- ✓ Requirements engineering & Requirements
- ✓ Requirements elicitation
- ✓ Requirements specification
- ✓ Requirements validation
- ✓ Requirements management
- ✓ Goal oriented requirement engineering
- ✓ Quality of requirements
- ✓ Non-functional requirements
- ✓ Requirement traceability

TOPICS COVERED

- ✓ Requirements engineering & Requirements
- ✓ Requirements elicitation
- ✓ Requirements specification
- ✓ Requirements validation
- ✓ Requirements management
- ✓ Goal oriented requirement engineering
- ✓ Quality of requirements
- ✓ Non-functional requirements
- ✓ Requirement traceability

REQUIREMENTS

EFFECTS OF INADEQUATE REQUIREMENTS DEVELOPMENT – AIRBUS

- ✓ **Requirement**: *Reverse thrust may only be used <u>when the airplane is landed</u>.*
- ✓ **Translation**: *Reverse thrust may only be used <u>while the wheels are rotating</u>.*
- ✓ **Implementation**: *Reverse thrust may only be used <u>while the wheels are rotating fast enough</u>.*

- **Situation**: Rainstorm – aquaplaning
- **Result**: Crash due to overshooting the runway!
- **Problem**: Erroneous in the requirement phase

The Ariane 5 accident – 1

- Single root cause failure!
- **The ”bug”: attitude deviation** stored as 2-byte integer (max value 65,535) instead of 4-byte (max value 4,294,967,295)
- SW module was reused from Ariane 4
- Insufficient V&V of detailed requirements: larger attitude deviation tolerated in Ariane 5 than in Ariane 4
- Ariane 5 production cost 10 years and $7 billion; luckily, no victims because it was unmanned.

65,53**5** = 00000000 00000000 11111111 11111111

65,53**6** = 00000000 00000001 00000000 00000000

OTHER SOFTWARE RELATED ACCIDENTS & INCIDENTS

- **Accidents & incidents**
  - Alarm flooding, **power distribution** failure, BP Grangemouth Scotland, 29th May - 10th June 2000
  - failure in a data bus + faulty logic in the software => engine power loss, **Airbus** A340-642, 2005
  - failed accelerometer + software bug => Faulty airspeed metering, **Boeing** 777-200, 2005
  - Software bug => shutdown of radio communication between **ATC and aircraft**, 2004. disrupted 800 flights
  - Safety-related software flaws => recall of 200,000 **pacemakers** in 1990-2000
  - **radiotherapy machines** attacked by computer **viruses**, 2005
  - Buggy software incorporate computer + connection between corporate and control systems networks => shutdown of the **nuclear power plant**, USA 2008
- Many software failures are actually ”bugs” in the **detailed requirement specifications,** i.e., poor understanding of the very detailed requirements

Korean Air 747 in Guam, 200 deaths (1997): incorrect configuration of ”minimum altitude” warning system

How the customer explained it
How the project leader understood it
How the engineer designed it
How the programmer wrote it
How the sales executive described it
How the project was documented
What operations installed
How the customer was billed
How the helpdesk supported it
What the customer really needed

REQUIREMENTS ENGINEERING

- ✓ The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.

WHAT IS A REQUIREMENT?

Requirement engineering = establishing the ***services*** that the customer requires from a system and the ***constraints*** under which it operates and is developed.

- Requirement = the descriptions of
  - the system services
  - and constraints
- It may range
  - from a high-level abstract statement
  - to a detailed mathematical functional specification.
- May serve a dual function
  - The basis for a bid for a contract - must be open to interpretation;
  - The basis for the contract itself - must be in detail;

STAKEHOLDERS

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

FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

- Functional requirements
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.
- Non-functional requirements
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.

  - Domain requirements
    - Constraints on the system from the domain of operation

NON-FUNCTIONAL REQUIREMENTS IMPLEMENTATION

- Non-functional requirements may be more critical than functional requirements.
  - If these are not met, the system may be useless.
- Non-functional requirements may affect the overall architecture of a system
  - rather than the individual components
  - cross-cutting concern
- A single non-functional requirement
  - may generate a number of related functional requirements
  - and may also generate requirements that restrict existing requirements.

NON-FUNCTIONAL REQUIREMENTS

- Define system properties and constraints
  - Properties: reliability, response time and storage requirements.
  - Constraints: I/O device capability, system representations, etc.

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

NON-FUNCTIONAL REQUIREMENTS IMPLEMENTATION

- Non-functional requirements may affect the overall architecture of a system
  - rather than the individual components
  - cross-cutting concern

- A single non-functional requirement
  - may generate a number of related functional requirements
  - and may also generate requirements that restrict existing requirements.

REQUIREMENTS ENGINEERING PROCESS

REQUIREMENTS ENGINEERING PROCESSES

- Processes to “generate” all require
- Generic activities common to all pr
  - Requirements elicitation;
  - Requirements analysis;
  - Requirements validation;
  - Requirements management.
- In practice, RE is an iterative activi

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

The RE process:

A SPIRAL VIEW OF THE REQUIREMENTS ENGINEERING PROCESS

A AGILE VIEW OF THE REQUIREMENTS ENGINEERING PROCESS

Iterative Enhancement Model

REQUIREMENT ELICITATION

REQUIREMENTS ELICITATION AND ANALYSIS

- ~ requirements elicitation or requirements discovery.

- Work with customers to find out:
  - the application domain, the services and the operational constraints (system performance, hardware constraints, etc.).
- May involve
  - end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called stakeholders.

PROBLEMS OF REQUIREMENTS ELICITATION

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

THE REQUIREMENTS ELICITATION AND ANALYSIS PROCESS

REQUIREMENTS DISCOVERY

- To gather information about the required and existing systems and distil the user and system requirements from this information.

- Main concerns:
  - Stakeholders
  - Discovery techniques/approaches/…

DISCOVERY TECHNIQUE - INTERVIEWING

- Part of most RE processes.
- Types of interview
  - Closed vs Open => mixed?
- Be effective
  - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
  - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

DISCOVERY TECHNIQU

- Observational technique
  - used to understand operat
    derive support requiremen
- How
  - A social scientist spends a considerable time observing and analysing how people actually work.
  - People do not have to explain or articulate their work.
  - Social and organisational factors of importance may be observed.

STORIES AND SCENARIOS

- ✓ Scenarios and use
  of how a system c
- ✓ Stories and scenar
  system may be us
- ✓ Because they are
  stakeholders can r
  comment on their
  story.

REQUIREMENT ELICITATION TECHNIQUES

	Interviews	Workshops	Focus groups	Observations	Questionnaires	System interface analysis	User interface analysis	Document analysis
Mass-market software	x		x		x			
Internal corporate software	x	x	x	x		x		x
Replacing existing system	x	x		x		x	x	x
Enhancing existing system	x	x				x	x	x
New application	x	x				x		
Packaged software implementation	x	x		x		x		x
Embedded systems	x	x				x		x
Geographically distributed stakeholders	x	x			x			

Suggested elicitation techniques by project characteristics

REQUIREMENTS SPECIFICATION

REQUIREMENTS SPECIFICATION

- The process of writing down the user and system requirements in a requirements document.

- Notes:
  - User requirements have to be understandable by end-users and customers who do not have a technical background.
  - System requirements are more detailed requirements and may include more technical information.
  - The requirements may be part of a contract for the system development

WAYS OF WRITING A SYSTEM REQUIREMENTS SPECIFICATION

Notation	Description
Natural language	Sentences in natural language. Each sentence should express one requirement.
Structured natural language	Natural language statements on a standard form or template
Design description languages	Uses a language like a programming language, but with more abstract features
Graphical notations	Graphical models, supplemented by text annotations (best for functional requirements); UML use case and sequence diagrams are commonly used.
Mathematical specifications	Based on mathematical concepts such as finite-state machines or sets; Can reduce the ambiguity but hard to understand (and hard to check manually)

NATURAL LANGUAGE SPECIFICATION

- Used for writing requirements because it is expressive, intuitive and universal.
  - The requirements can be understood by users and customers.

- Problems
  - Lack of clarity: Precision is difficult without making the document difficult to read.
  - Requirements confusion: Functional and non-functional requirements tend to be mixed-up.
  - Requirements amalgamation: Several different requirements may be expressed together.

An insulin pump control system

An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system, which collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

People who suffer from diabetes use the system. Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolises glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.

EXAMPLE REQUIREMENTS FOR THE INSULIN PUMP SOFTWARE SYSTEM

- ✓ Req 3.2. The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes.

- ✓ Req 3.6. The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1.

STRUCTURED SPECIFICATIONS

- Writing on a standard form or template:
  - Name
  - Inputs, outputs
  - The information needed for the computation
  - Action
  - Pre and post conditions (if appropriate)
  - The side effects (if any)

- This works well for some types of requirements e.g. requirements for embedded control system

A
RE

Insulin Pump/Control Software/SRS/3.3.2	
Function	Compute insulin dose: safe sugar level
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2); the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered
Destination	Main control loop.
Action	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requirements	Two previous readings so that the rate of change of sugar level can be computed
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin

TABULAR SPECIFICATION

- Particularly useful when you have to define a number of possible alternative courses of action.
- Example:

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

USE CASES

- Use-cases are a kind of scenario
  - identify the actors in an interaction and which describe the interaction itself
  - Included in the UML
- A set of use cases should describe all possible interactions with the system.
- UML sequence diagrams may be used to add detail to use-cases
  - show the sequence of event processing in the system

USE CASE DIA

- Actors
- Use cases
- Association
- System boundary boxes

Partial use case diagram for the Chemical Tracking System (CTS)

NTNU

USE CASE TEMPLATE AND EXAMPLE

ID and Name:	UC-4 Request a Chemical		
Created By:	Lori	Date Created:	8/22/13
Primary Actor:	Requester	Secondary Actors:	Buyer, Chemical Stockroom, Training Database
Description:	The Requester specifies the desired chemical to request by entering its name or chemical ID number or by importing its structure from a chemical drawing tool. The system either offers the Requester a container of the chemical from the chemical stockroom or lets the Requester order one from a vendor.		
Trigger:	Requester indicates that he wants to request a chemical.		
Preconditions:	PRE-1. User's identity has been authenticated. PRE-2. User is authorized to request chemicals. PRE-3. Chemical inventory database is online.		
Postconditions:	POST-1. Request is stored in the CTS. POST-2. Request was sent to the Chemical Stockroom or to a Buyer.		

USE CASE TEMPLATE AND EXAMPLE (CONT')

ID and Name:	UC-4 Request a Chemical
Normal Flow:	**4.0 Request a Chemical from the Chemical Stockroom** 1. Requester specifies the desired chemical. 2. System lists containers of the desired chemical that are in the chemical stockroom, if any. 3. System gives Requester the option to View Container History for any container. 4. Requester selects a specific container or asks to place a vendor order (see 4.1). 5. Requester enters other information to complete the request. 6. System stores the request and notifies the Chemical Stockroom.
Alternative Flows:	**4.1 Request a Chemical from a Vendor** 1. Requester searches vendor catalogs for the chemical (see 4.1.E1). 2. System displays a list of vendors for the chemical with available container sizes, grades, and prices. 3. Requester selects a vendor, container size, grade, and number of containers. 4. Requester enters other information to complete the request. 5. System stores the request and notifies the Buyer.
Exceptions:	**4.1.E1 Chemical Is Not Commercially Available**

DATA FLOW DIAGRAM

- ✓ How data moves through a system

NTNU

DECISION TABLE/DECISION TREE

- ✓ Complex logics

Requirement Number					
Condition	**1**	**2**	**3**	**4**	**5**
User is authorized	F	T	T	T	T
Chemical is available	—	F	T	T	T
Chemical is hazardous	—	—	F	T	T
Requester is trained	—	—	—	F	T
Action					
Accept request			X		X
Reject request	X	X		X	

STATE-TRANSITION DIAGRAM

- ✓ A set of complex state changes in natural language creates a high probability of overlooking a permitted state change or including a disallowed change.

THE SOFTWARE REQUIREMENTS DOCUMENT

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

REQUIREMENT VALIDATION

REQUIREMENTS VALIDATION

- ✓ Requirements error can be costly !

Cost to Fix Requirements Error (in Ratios)			
Development Discipline/Phase	**Composite of Studies (NASA)***	**NASA Software#**	**Davis' Composite+**
Requirements	1 (baseline)	1 (baseline)	1 (baseline)
Design	5x	8x	2.5x – 5x
Code	10x	16x	5x – 10x
Test	50.5x	21x	Unit Test: 10x – 20x Acceptance Test: 25x – 50x
Post-Deployment	n/a	29x	100x – 200x

* NASA reviewed McGibbon (2003), Pavlina (2003), Cigital (2003), Rothman (2002), Hoffman (2001), Rothman (2000), and Boehm (1981) and assessed the median of their data, presented here.
# NASA analyzed its own software projects to assess the relevance of reviewed studies.
# Leffingwell & Widrig reviewed GTE, TRW, IBM, and Davis (2003), who himself reviewed several studies. The results of Davis' work are presented here.

REQUIREMENTS CHECKING

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

REQUIREMENTS VALIDATION TECHNIQUES

- ✓ Requirements reviews
  - Systematic manual analysis of the requirements.
- ✓ Prototyping
  - Using an executable model of the system to check requirements.
- ✓ Test-case generation
  - Developing tests for requirements to check testability.

REQUIREMENTS MANAGEMENT

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

CHANGING REQUIREMENTS

- The business and technical environment of the system always changes after installation.
- The people who pay for a system and the users of that system are rarely the same people.
- Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

REQUIREMENTS MANAGEMENT

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

REQUIREMENTS MANAGEMENT PLANNING

- ✓ Establishes the level of requirements management detail that is required.

- ✓ Requirements management decisions:
  - Requirements identification
  - A change management process
  - Traceability policies
  - Tool support

REQUIREMENTS CHANGE MANAGEMENT

- Deciding if a requirements change should be accepted
  - Problem analysis and change specification
  - Change analysis and costing
  - Change implementation

RESPONSE TO CHANGE: AGILE APPROACH!

SUMMARY

- ✓ Requirements: what the system should do and constraints on its operation and implementation.
- ✓ Functional requirements = the services
- ✓ Non-functional requirements = constraints (development & use)
  - apply to the system as a whole.
- ✓ The software requirements document (i.e. SRS) is an agreed statement of the system requirements.
- ✓ The RE process is an iterative process
  - requirements elicitation, specification and validation.

GOAL ORIENTED REQUIREMENT ENGINEERING

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

WHAT IS GORE [3]?

- GORE is approach and modeling method to
  - Identify goals and sub-goals of software systems
  - Identify who and how to achieve the goals
  - Identify domain hypothesis and conflicts
- Similar principle to use cases and user stories
- Can be applicable to cyber-physical systems and user-system interaction systems

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- ✓ Anticipating obstacles
- ✓ Identifying domain hypothesis
- ✓ Handling conflicts

GOAL MODEL

- A goal is an objective the system under consideration should achieve
- Goal is condition or situation that can be achieved or not. Goal is used to define the functional requirements of the system.
- Goals provide a precise criterion for sufficient completeness of a requirements specification

GOAL MODEL

- Hierarchy and relationship of goals
- Goals can be of different types

GOAL MODEL USING BEHAVIOR AND SOFT GOAL TYPES

- Soft goal
  - Satisfaction cannot be established in a clear-cut sense
  - Useful for comparing alternative goal refinements
- Hard/behavioral goal
  - Satisfaction can be established through verification

GOAL MODEL USING BEHAVIOR GOAL TYPES

Informal temporal patterns

Achieve[TargetCondition]
Cease[TargetCondition]

Maintain[GoodCondition]
Avoid[BadCondition]

Improve[TargetCondition]
Increase[TargetQuantity]
Reduce[TargetQuantity]
Maximise[ObjectiveFunction]
Minimise[ObjectiveFunction]

- Achieve: Some target property to be eventually satisfied
- Maintain: Some target property to be permanently satisfied
- Avoid: Some target property to be permanently denied
- …

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- ➢ Anticipating obstacles
- ➢ Identifying domain hypothesis
- ➢ Handling conflicts

ELICITING GOALS

Each sub-goal is refined into finer-grained goals until we reach a software requirement associated with a single agent and expectation (assumption) on the environment.

CASE STUDY: TRAIN TRANSPORT SAFETY

NTNU

SOFT AND BEHAVIOR GOALS

ELICITING NEW GOALS THROUGH “HOW” QUESTIONS

CONTINUE ELICITING NEW GOALS THROUGH "HOW" QUESTIONS

NTNU

GOALS REFINEMENT

Mean-ends **relationship** ( )
Decomposition **relationship** ( )
Contribution **relationship**( )
Dependency **relationship** ( )

ELICITING NEW GOALS THROUGH “WHY” QUESTIONS

- ✓ High level goals are identified by asking WHY questions

AATC SOFT GOALS

How

Why

LINKS:

+ : Contribute
- : Conflict
Double line : OR
Black circle : AND
… : More sub-goals

++ : Contribute strongly
-- : Conflict strongly

NTNU

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- ➢ Anticipating obstacles
- ➢ Identifying domain hypothesis
- ➢ Handling conflicts

OBJECT MODEL

- Agent: independent active object, e.g.,
  - Speed/Acceleration control system
  - On-board train controller
  - Tracking systems

- Agent has entities
  - E.g., tracking systems have several sensors

AGENT RESPONSIBILITY MODEL

- A **goal** has to be placed under the responsibility of an **agent**.

NTNU

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- ➢ Anticipating obstacles
- ➢ Identifying domain hypothesis
- ➢ Handling conflicts

OPERATION MODEL

- ✓ Operation is input-output relation over objects
- ✓ Defines state transitions
- ✓ Characterized by pre-, post-, and trigger conditions

OPERATION MODEL – TRAIN CONTROL SYS.

- ✓ Operation: SendCommandMessage
- ✓ Input: Train ID
- ✓ Output: Command to the train
- ✓ Pre-condition: Message not sent
- ✓ Post-condition: Message sent to correct train

State transition

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- Anticipating obstacles
- Identifying domain hypothesis
- Handling conflicts

ANTICIPATING OBSTACLES

- Obstacles (Possible failures that may negatively impact goal achievement)
  - CommandMsgNotIssued
  - CommandMsgIssuedLate
  - CommandMsgSentToWrongTrain

IDENTIFY DOMAIN HYPOTHESIS

- **Domain property**: A descriptive statement about the problem world.
- **Domain invariants:** Should hold invariably, regardless of how the system behaves. This usually corresponds to some physical laws. E.g. *The light is either on or off*.
- **Domain hypothesis**: To satisfy a service request, an infrastructure/environment to perform the service must be available.

IDENTIFYING DOMAIN HYPOTHESIS

✓ Domain hypothesis related to the goal: *Every track segment leading to a switch is ended with a gate*.

HANDLING GOAL CONFLICTS

CAN GORE HELP DERIVE BETTER REQUIREMENTS ?

- ✓ "Requirements definition must say why a system is needed, based on current or foreseen conditions, which may be internal operations or an external market. It must say what system features will serve and satisfy this context. And it must say how the system is to be constructed. [1]"

Goal models

Agent models

Responsibility models

Operation models

Domain hypotheses

SUMMARY (CONT.)

- ✓ Requirements elicitation and analysis = iterative process
  - requirements discovery, classification and organization, negotiation and requirements documentation.
- ✓ Techniques for requirements elicitation
  - interviews, scenarios, use-cases and ethnography, etc.
- ✓ Requirements validation = checking the requirements
  - for validity, consistency, completeness, realism and verifiability.
- ✓ Business, organizational and technical changes inevitably
  - => changes to the requirements for a software system.
- ✓ Requirements management = managing and controlling the requirement changes.

- Electronic Healthcare Record (EHR) system is a repository of patient data in digital form, stored and exchanged securely, and accessible by multiple authorised users. As a normal flow, medication will be first prescribed by authorized nurses to a patient, after that the medication is dispensed and then taken by the patient. EHR ServerRX deployed in a London hospital, required all the patients to wear a bracelet ID for identification purposes. Nurses had to scan this bracelet before dispensing any medicine in order to reduce the risk of administering the medication to the wrong patient.
- Describe at least three sub goals to the goals
- https://app.diagrams.net/#G1pL-o8b58wxjIz1BakfxWQo-B6EWvYL8W

EXERCIS

GOAL MODEL
Achieve [Prescribed Medication Taken By Patient]
Achieve [Patient ID Verified]
Achieve [Dispensed Medication Taken By Patient]
Achieve [Prescribed Medication Dispensed When Patient ID Verified]

Maintain [Medication in stock]
Avoid [Medicine Overdose]
Achieve [Prescribed Medication Taken By Patient]
Achieve [Patient ID Verified]
Achieve [Prescribed Medication Dispensed when Patient ID Verified]
Achieve [Dispensed Medication Taken By Patient]
Nurse
Patient Wears Identification Bracelet
Achieve [Patient Identification Bracelet Scanned]
Achieve [Prescribed Medication In Patient Drawer]
Achieve [Patient Drawer Opened When Patient Bracelet Scanned
Nurse
Nurse
Electronic Dispensing System

QUALITY OF REQUIREMENTS

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

EXAMPLE OF NIGHTMARE REQUIREMENTS

- ✓ The system shall perform at the maximum rating at all times except that in emergencies it shall be capable of providing up to 125% rating unless the emergency condition continues for more than 15 minutes in which case the rating shall be reduced to 105% but in the event that only 95% can be achieved then the system shall activate a reduced rating exception and shall maintain the rating within 10% of the stated values for a minimum of 30 minutes.

More than one requirement in a paragraph
Rambling: like a novel
Vague terms: emergencies

ANOTHER EXAMPLE OF NIGHTMARE REQUIREMENTS

- ✓ The system shall provide general word processing facilities which shall be easy to use by untrained staff and shall run on a thin Ethernet Local Area Network wired into the overhead ducting with integrated interface cards housed in each system together with additional memory if that should be necessary.

More than one requirement in a paragraph
Rambling: like a novel
Let-out clauses: if that should be necessary
Vague words: be easy to use, additional memory

CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION*

- Complete
- Unambiguous
- Consistent
- Correct
- ✓ Verifiable
- ✓ Traceable
- ✓ Ranked for importance and/or stability
- ✓ Modifiable

* IEEE Std 830-1998

COMPLETENESS

- ✓ Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations (IEEE Std).

- ✓ All possible situations must be covered

  "If X then….", "If Y then…." Must also consider what will happen "If neither X nor Y…"

COMPLETENESS (CONT')

- Automatic door opener

  If the door is closed and a person is detected, then send signal Open_Door.

  If no person is detected after 10 sec., send signal Close_Door.

- What are missing?

UNAMBIGUOUS

- ✓ A software requirement is unambiguous, if and only if, every requirement stated therein has only one interpretation (IEEE Std).
- Ambiguity: Requirement with terms or statements that can be interpreted in different ways.

UNAMBIGUOUS (CONT')

- ✓ Requirement of autonomous cruise control (ACC) system is:

The maximum speed of a vehicle in a busy road shall be 10 mph.

Ambiguity: What type of vehicle do you mean?
What is busy road?

CONSISTENT

- ✓ A software requirement is internally consistent if, and only if, no subset of individual requirements described in is conflict.

CONSISTENT (CONT')

- Two requirements of the PROFIBUS network are in conflict
  - The PROFIBUS shall have a short reaction time of 60ms
  - PROFIBUS used in hazardous area shall have a low power dissipation of 3.6W

Conflict: Fast PROFIBUS has to have a high power

MAIN TYPES OF CONFLICTS

- ✓ Three main types of conflicts in software requirements
  - Specific characteristics of real-world objects
  - The logical or temporal conflict between two actions
  - Different terms for describing the same real-world object

- Consistency is a challenge since we, at least in the general case, need a complete overview of all requirements that are related to the same event, function, or parameter.

CORRECT

- ✓ Possible incorrect requirements
  - Forward referencing
  - Opacity
  - Noise
  - Etc.

REQUIREMENTS IMPRECISION

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

- For the term ‘search’ in requirement 1
  - User intention – search for a patient name across all appointments in all clinics;
  - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

FORWARD REFERENCING

- ✓ Requirement items that make use of problem world domain features that are not yet defined.

E, C, and D need to be mapped to a requirement item

FORWARD REFERENCING (CONT')

✓ A requirement of autonomous cruise control (ACC) system is:

ACC system shall maintain the preset speed of an ego-vehicle if there is no forward vehicle.

- Missing inference
  - Who sets the preset speed?
  - What is the value of the preset speed?

OPACITY

- ✓ Requirement items for which rational or dependencies are invisible.

Multiple unrelated concept mapping. A is not related to B

OPACITY (CONT')

- ✓ Requirement:

  Each time the freight train (Freight trains are trains used for carrying goods) doors are closed, the passengers must all seated.

- Opacity
  - There is no visible relationship between the freight trains and passengers.

NOISE

- Requirement items that yield no information on problem world features.

X refers to a concept undefined in the domain

NOISE (CONT')

✓ Requirement: The train system shall guarantee safe transportation of all passengers on their residence.

- Noise
  - A residence is an unknown concept within the train domain
  - The train can only transport passengers to the train station and not to their residence

CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION*

- ✓ Complete
- ✓ Unambiguous
- ✓ Consistent
- ✓ Correct
- ➢ Verifiable
- ✓ Traceable
- ✓ Ranked for importance and/or stability
- ✓ Modifiable

* IEEE Std 830-1998

TESTABLE REQUIREMENT

- ✓ If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement is not testable.

TESTABILITY CONCERNS

- Testability touches upon two areas of concern
  - How test-friendly is the requirement?
  - How easy is it to test the implementation?

- These two concerns are not independent and need to be considered together

THREE WAYS OF CHECKING GOAL ACHIEVEMENT

- ✓ Executing a test
  - Give input
  - Observe and check the output
  - Involves the computer system and peripherals
- ✓ Run experiments
  - Input / output but also involves the users
- ✓ Inspect the code and other artifacts
  - Evaluation based on documents

WHEN TO USE WHAT

T: Run test on code
E: Experiment
I: Inspect

NTNU

TO MAKE REQUIREMENTS TESTABLE

- ✓ A requirement needs to be stated in a precise way
- ✓ For some requirements, this is in place right from the start

When the ACC system is turned on, the "Active" light on the dashboard shall be turned on.

- ✓ For some other requirements, we need to change it to get a testable version

The system shall be easy to use.

REQUIREMENT TESTABILITY CHECKLIST

- ✓ Modifying phrases
- ✓ Vague words
- ✓ Pronouns with no reference
- ✓ Passive voice
- ✓ Negative requirements
- ✓ Assumptions and comparisons
- ✓ Indefinite pronouns
- ✓ Boundary values
- ✓ …

MODIFYING PHRASES

- The meaning of the following words and phrases are subject to interpretation and make requirement optional:
  - as appropriate
  - if practical
  - as required
  - to the extent necessary / practical

- “Shall be considered" only requires the contractor to think about it
- “And/or” mandatory/optional?
- “Shall/should/may”, should not be used for priority

VAGUE WORDS

- Vague words inject confusion, e.g.,
  - Manage, handle, process, necessary, appropriate
  - Usually, approximately, systematically, quickly
- Use words that express what the system must do
- ***Requirement***: The system shall process ABC data to the extent necessary to store it in an appropriate form for future access.
- ***Correction***: The system shall edit ABC data.

PRONOUNS WITH NO REFERENCE

- ***Requirement***: It shall be displayed.
- When this occurs, the writer is usually relying on a nearby requirement in the requirements document for the meaning of "it."
- As requirements are assigned for implementation, they are often reordered and regrouped, and the defining requirement is no longer nearby.

PASSIVE VOICE

- Requirements should be written in an active voice, which clearly shows X does or provides Y
- **Requirement with passive voice**: Z shall be calculated
- **Correction by using active voice**: The cruise control system shall calculate Z.

NEGATIVE REQUIREMENTS

- ✓ *Everything* outside the system is what the system does not do
- ✓ Testing would have to continue forever to prove that the system does not do something
- ✓ State what the system does

Negative requirement: Dangerous characters should not be used (blacklist).

Correction: Only characters A, B, C are allowed to be used (whitelist).

ASSUMPTIONS AND COMPARISONS

- ✓ **Requirement**: the system shall increase throughput by 15%.
- ✓ Sounds testable, but isn't
- ✓ The assumption is "over current system throughput"
- ✓ By comparing to another system, the meaning of the requirement changes when the other system changes

INDEFINITE PRONOUNS

- ✓ Indefinite pronouns are “stand-in” for unnamed people or things
- ✓ Their meaning is subject to interpretation

- All
- Another
- Any
- Anybody
- Anything
- Each
- Either
- Every
- Everybody
- Everyone
- Everything
- Few
- Many
- Most
- Much

BOUNDARY VALUES

- ✓ Vacation requests of up to 5 days do not require approval. Vacation requests of 5 to 10 days require supervisor approval. Vacation requests of 10 days or longer require management approval.

HOW EASY IS IT TO TEST THE IMPLEMENTATION?

- ✓ Some requirements are more difficult to test than others
- ✓ The volume of tests needed, e.g.,
  - Response time or storage capacity
- ✓ Type of event to be tested, e.g.,
  - Error handling or safety mechanisms
- ✓ The required state of the system before testing, e.g.,
  - A rare failure state or a specific transaction history

HOW EASY IS IT TO TEST THE IMPLEMENTATION? (CONT')

- ✓ Four concerns related to implementation and testability
  - Autonomy of the system under test
  - Observability of the testing progress
  - Re-test efficiency
  - Test restartability

AUTONOMY OF THE SYSTEM UNDER TEST

- How many other systems are needed to test this requirement?
- Requirement: “If the door is closed and a person is detected then send signal Open_Door”
- Can be tested using a simulated actuator
  - Simulate a “person detected” signal on the sensor and check if an Open_Door signal is sent to the actuator
- More difficult/costly to be tested if the actual sensors and actuators must be installed for running tests.

OBSERVABILITY

- ✓ How easy is it to observe the progress of the test execution?

  This is important for tests that do not produce output – e.g., the requirement is only concerned with an internal state change or update of a database.

- ✓ How easy to observe results of the test?

  Important for tests where the output is dependent on an internal state or database content.

RE-TEST EFFICIENCY

- How easy is it to perform the “test – check – change – re-test” cycle?

- This includes
  - Observe the test result and run only failed tests
  - Run only tests that are related to the change

TEST RESTARTABILITY

- How easy is it to
  - Stop the test temporarily
  - Study current state and output
  - Start the test again from the point where it was stopped

INVOLVE TESTERS IN REQUIREMENT PHASE

- ✓ That a requirement is testable does not necessarily mean that it is ***easy*** to test

- ✓ In order to have testable requirements it is important that

  The testers are involved right from the start of the project. It is difficult to add testability later.

- ✓ The tests are integrated parts of the requirement

NON-FUNCTIONAL REQUIREMENTS

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

WHAT ARE NON-FUNCTIONAL REQUIREMENTS (NFR)

- Many definitions can be found of NFR
- A NFR is basically an attribute of or a constraint on a system [1]
  - Attributes
    - **-ilities**: understandability, reliability, portability, flexibility, availability, maintainability, scalability, …
    - **-ities**: security, simplicity, clarity, ubiquity, integrity, safety, modularity, …
    - **-ness**: user-friendliness, robustness, timeliness, responsiveness, …
  - Constraint
    - **Physical**
    - **Environmental**
    - **Legal**
    - **Cultural**

NTNU

ISO/IEC 25010 PRODUCT QUALITY MODEL

The ISO model is a factor – criteria – metrics model. Only the first two levels are shown in the diagram.

MAKE RELIABILITY MEASURABLE

The quality concepts
(abstract notions of quality properties)

Measurable Quantities
(define some metrics)

Counts taken from design/execution
(realization of the metrics)

NTNU

RELIABILITY REQUIREMENTS SPECIFICATION

- **Maturity**:
  MTTF (Mean Time To Failure, the expected time to failure for a non-repairable system) > 500 hrs

- ✓ **Availability:**
  The system shall meet or exceed 99.99% uptime

- ✓ **Recoverability:**
  In case of an error, the time needed to get the system up and running again (MTTR) shall not exceed one hour

RELIABILITY TESTS

- ✓ MTTF calculation
  - Assume you tested 3 identical systems starting from time 0 until all of them failed
  - The first system failed at 10 hours, the second failed at 12 hours and the third failed at 13 hours
  - The MTTF is the average of the three failure times, which is (10+12+13)/3 = 11.6667 hours

- How to verify MTTF > 500 hrs?
  - Use 10 PCs. Test for two weeks (40 hr per week) => TTT (Total Test Time) = 800
  - Failed more than once? Satisfied/Not satisfied

TOTAL TEST TIME IN LAB VS. USAGE TIME

- We need to distinguish
  - Total test time (TTT) in the lab
  - Usage time (UT) of the system in real life

- The usage time is related to
  - How often is the system used at the users' site?
  - Number of users each time the system is used

TOTAL TEST TIME IN LAB VS. USAGE TIME (CONT')

A simple example:

- ✓ We have TTT = 800 hrs in the lab.
  - Use 10 PCs. Test for two weeks (40 hrs per week)
- ✓ The system will be used for only 20 hrs per week – e.g., for accounting purposes
- ✓ The 800hrs in the lab. TTT is equivalent to 1600 hrs of usage time
- ✓ Failed twice? Satisfy/Not satisfy MTTF > 500 hrs?

EXERCISE : AVAILABILITY REQUIREMENTS

- If the system shall meet or exceed the following availability, what is the maximum downtime per year to satisfy such availability requirements?

maximum downtime = (total time per year) x (1-availability)

Availability requirement	Maxium downtime per year?
99%	3.65 days/year
99.9%	8.76 hours/year
99.999%	5 minutes/year
99.9999%	31 seconds/ year

PERFORMANCE

Response speed and amount of resources used

- Software products
- Hardware facilities
- Materials (e.g., print paper, diskettes)

Criteria

- **Time behavior:** Response and processing times and throughput rates
- **Resource utilization:** Amount and types of resources used
- **Capability**: Minimum/maximum limits

PERFORMANCE REQUIREMENT EXAMPLES

- ✓ Authorization of an ATM withdrawal request shall take no more than 2.0 seconds.

Response time

- ✓ Webpages shall fully download in an average of 3 seconds or less over 30 megabits/second Internet connection.

Resource utilization

- ✓ The on-line banking systems should support minimum 10.000 credit card transactions proceed per second.

Capability

USABILITY

Help users achieve specified goals with

- Effectiveness
- Efficiency
- Satisfaction

Criteria

- **User interface aesthetics**
- **Easy to operate**
- **Easy to learn**

Usability criteria are subjective

Tests should have a strong component of subjectivism

SETTING REQUIREMENTS – THE MBO METHOD

- ✓ The method used in the example is based on T. Gilb's ideas of MbO – Management by Objectives [5]
- ✓ We start with the requirement – e.g., the system shall be easy to learn
- ✓ We then follow up with "what do you mean by…" until we reach something that is observable and thus testable

REQUIREMENTS – FIRST STEP

- The first step is to apply the MbO for each criterion

- Examples
  - Learn its application
    What do we mean by “learn application.”

  - Is the software product liked by the user
    What do you mean by “beautiful / pleasing / like”?

REQUIREMENTS – SECOND STEP

- ✓ Learn application: After a two weeks course and after two weeks use of the system, a normal user can solve a set of standardized problems

- ✓ Pleasing/like: Score high on a likeability scale – e.g., 90 % score 7 or higher on a scale from 1 to 10 – after a two weeks course and two weeks of real use

We need to develop a course, a set of standardized problems, and a likeability questionnaire.

USER INVOLVEMENT IN SETTING REQUIREMENTS

When we use MbO or other related techniques for setting requirements, we will in most cases have a situation where:

- ✓ The user will have to participate in the tests
- ✓ There will be a strong link between requirement and test
- ✓ In many cases, ***passing the test by the user will be the requirement***

NFR REQUIREMENTS – TO REMEMBER

- ✓ Some NFR requirements are conflict with each other
- ✓ Trade-off of NFR requirements will impact system design and implementation

CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION*

- ✓ Complete
- ✓ Unambiguous
- ✓ Consistent
- ✓ Correct
- ✓ Verifiable
- ➢ Ranked for importance and/or stability
- ➢ Traceable
- ➢ Modifiable

* IEEE Std 830-1998

RANKED FOR IMPORTANCE AND/OR STABILITY

- ✓ If, and only if, each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.

- ✓ Rank and distinguish classes of requirements
  - Essential
  - Conditional
  - Optional

REQUIREMENT TRACEBILITY

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

REQUIREMENTS TRACEABILITY

"Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction, i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases."

Gotel and Finkelstein

BIDIRECTIONAL TRACEABILITY

PURPOSES OF REQUIREMENT TRACEABILITY

- ✓ Forward traceability
  - Ensure all requirements are implemented
  - Change impact analysis

- ✓ Backward traceability
  - Avoid “gold plating”
  - Change impact analysis
  - Defect impact analysis
  - Root cause analysis of defects

HOW TO IMPLEMENT TRACEABILITY?

- ✓ Traceability matrix

Project Name

Unique No.	Requirement	Source of Requirement	Software Reqs. Spec / Functional Req. Doc.	Design Spec.	Program Module	Test Case(s)	Successful Test Verification	Modification of Requirement	Remarks

EXAMPLE OF TRACEABILITY MATRIX

Requirement Source	Product Requirements	HLD Section #	LLD Section #	Code Unit	UTS Case #	STS Case #	User Manual
Business Rule #1	R00120 Credit Card Types	4.1 Parse Mag Strip	4.1.1 Read Card Type	Read_Card _Type.c Read_Card _Type.h	UT 4.1.032 UT 4.1.033 UT 4.1.038 UT 4.1.043	ST 120.020 ST 120.021 ST 120.022	Section 12
			4.1.2 Verify Card Type	Ver_Card _Type.c Ver_Card _Type.h Ver_Card _Types. dat	UT 4.2.012 UT 4.2.013 UT 4.2.016 UT 4.2.031 UT 4.2.045	ST 120.035 ST 120.036 ST 120.037 ST 120.037	Section 12
Use Case #132 step 6	R00230 Read Gas Flow	7.2.2 Gas Flow Meter Interface	7.2.2 Read Gas Flow Indicator	Read_Gas _Flow.c	UT 7.2.043 UT 7.2.044	ST 230.002 ST 230.003	Section 21.1.2
	R00231 Calculate Gas Price	7.3 Calculate Gas price	7.3 Calculate Gas price	Cal_Gas_ Price.c	UT 7.3.005 UT 7.3.006 UT 7.3.007	ST 231.001 ST 231.002 ST 231.003	Section 21.1.3

HOW TO IMPLEMENT TRACEABILITY? (CONT')

- ✓ Trace tagging

SRS

R00104 The system shall c
time prior to the actual dispe
requests cancellation.

SDS

SDS Identifier	SRS Tag	Component Name
7.01.032	R00104	Cancel_ Transaction

UTS

Test Case #	SDS Identifier	Test C
23476	7.01.032	Cancel_Befor
23477	7.01.032	Cancel_After_
23478	7.01.032	Cancel_After_

CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION*

- ✓ Complete
- ✓ Unambiguous
- ✓ Consistent
- ✓ Correct
- ✓ Verifiable
- ✓ Ranked for importance and/or stability
- ✓ Traceable
- ➢ Modifiable

* IEEE Std 830-1998

MODIFIABLE

- If, and only if, a requirement structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style.
- Redundancy can easily lead to low modifiability
- Boilerplate can help improve modifiability
  - E.g., “will” -> “shall”

Course summary

Jingyue Li (Bill)

Exam

Vurderingsordning: Mappeevaluering

Termin	Statuskode	Vurdering	Vekting	Hjelpemidler	Dato	Tid	Eksamens-system
Vår	ORD	Arbeider	50/100				INSPERA
Vår	ORD	Hjemme-eksamen (1)	50/100		Utlevering 03.06.2022 09:00 Innlevering 03.06.2022 13:00		INSPERA
Sommer	UTS	Arbeider	50/100				INSPERA
Sommer	UTS	Hjemme-eksamen	50/100				INSPERA

Letter grade A-F

A: 89–100 points
B: 77–88 points
C: 65–76 points
D: 53–64 points
E: 41–52 points
F: 0–40 points

Evaluation and grading

- Exercises: 50%
- Exam: 50%
- You have to pass both!
  - You have to hand in **all** the exercises **and** get a passing grade to be allowed to take the exam
  - If you fail the exam, you will fail the course

Exercises	Exam	Final grade
Fail	Fail	Fail
Fail	Pass	Fail
Pass	Fail	Fail
Pass	Pass	Combine exercise and exam grades

Couse modules

- Requirement module
- Testing module
- Code review and refactoring
- Chosen topics
  - DevOps
  - Technical debt
  - Cost and effort estimation
  - Defect estimation
  - Software ecosystem
  - Software startup
- Exercises 1 - 3

Curriculum

- Journal and conference articles (uploaded to blackboard)
- Some chapters from books (accessible in NTNU library)

(1st/2nd edition)

Requirement module

Description	Students should be able to	To read
Requirement elicitation	• Apply goal oriented RE approach	• Ian Sommerville (2016), Software Engineering (10th ed.), ISBN 978- 0133943030, chapter 4 • Paper 1: Goal-oriented RE A guided tour
Requirement quality	• Identify quality issues of requirements and fix them • Define different function and non-functional requirements	• Paper 1: IEEE Recommended Practice for Software Requirements Specifications Std 830-1998 • Paper 2: Glinz, M. (2007). On Non-Functional Requirements. 15th IEEE International Requirements Engineering Conference (RE 2007), 21–26. • Paper 3: Ontology-Driven Guidance for Requirements Elicitation • Paper 4: Bidirectional requirement traceability

Testing module - 1

Description	Students should be able to	To read
Control flow and data flow testing	• Explain code, decision, path, and output coverages • Create test cases by using different data flow testing strategies • Explain how to use the test coverage information for different purposes	Software testing and quality assurance book, chapter 4 and 5
Domain and function testing	• Apply domain testing approach to generate test cases of single variable and multiple variables in combination • Explain risk-based testing	Software testing and quality assurance book, chapter 6 and 9

Testing module - 2

Description	Students should be able to	To read
Integration and system testing	• Explain different approaches for creating integration test cases and their pros and cons • Create different system test cases • Explain different categories of acceptance test cases • Explain different test prioritization approaches	Software testing and quality assurance book, chapter 7, 8, and 14
Regression testing	• Apply graph-walk safe regression test selection approach • Explain firewall regression test selection approach • Explain different regression test minimization and prioritization strategies	Paper 1: Regression testing minimization, selection, and prioritization: a survey

Testing module - 3

Description	Students should be able to	To read
Test in practice	• Explain concepts and strategies related to test planning and execution. • Create a test plan.	Software testing and quality assurance book, chapter 12 and 13

Code review and refactoring

Description	Students should be able to	To read
Code review and code refactoring	• Explain why code inspection and testing complement each other • Explain unsystematic vs. systematic reading techniques • Apply check-list based, defect-based, and perspective-based reading techniques • Explain the purpose and steps of code refactoring • Apply code refactoring methods to identify bad code smells in code (Python) and refactor them	Software reading techniques, chapter 3 and 4 Refactoring book: Improving the Design of Existing Code, chapter 3 (Bad smells in code) Paper 1: Modern code review a case study at google Paper 2: Expectations, Outcomes, and Challenges of Modern Code Review

Chosen topics - 1

Description	Students should be able to	To read
DevOps	• Explain the motivation for using DevOps • Explain the basic concepts of DevOps	• Slides only
Technical debt	• Explain the motivation and approaches to measure technical debt	• Slides only
Cost&Effort and defect estimation	• Explain importance and challenges in estimating cost and software defects • Describe different estimation techniques, i.e., expert judgement, estimation by analogy, Lines of code, and COCOMO • Describe machine learning techniques for software defect predictions	• Slides only

Chosen topics - 2

Description	Students should be able to	To read
Software ecosystem	• Explain definitions, key elements and examples of software ecosystems • Describe characteristics of software ecosystems: complexity, productivity, robustness, healthiness and coopetition	• Slides only
Software startup	• Explain definitions, key elements and examples of software startups • Describe concept of Technical Debt in Software startups • Explain logic of effectuation and causation • Describe concept and usages of Minimum Viable Product in Software startups	• Slides only

Structure of the exam

- Two case studies (10 points each, 20 points in total)
- Five open-ended questions (3 points each, 15 points in total)
- 15 close-ended (Multiple-choice) questions (1 point each, 15 points in total)

A case study example

- Company A is going to pay Company B for developing Autonomous Truck Platooning. Truck platooning (as shown in the above picture) is the linking of two or more trucks in convoy, using connectivity technology and automated driving support systems. These vehicles automatically maintain a set, close distance between each other when they are connected for certain parts of a journey, for instance, on motorways.

- The requirements Company B got from Company A are as follows.

We are going to develop Autonomous Truck Platooning. We want some numbers of trucks can drive autonomously, and other trucks are driven by humans. The autonomous trucks should drive at a default speed. We hope the truck platooning can drive along the route we set in GPS before the journey and follow the human drivers' command to change the route. We hope the autonomous trucks should have three chairs for the drivers. We want the truck to drive safely. So the trucks should have sensors to avoid collisions with obstacles because there are always other vehicles or obstacles in the motorway. When there are obstacles on the road, the trucks should stop. We also want the trucks to drive efficiently, which means that the trucks should minimize the distance between them and maximize the speed of the whole Platooning. The trucks should use sensors to detect the distances between trucks and the speed of the trucks.

A case study example (cont')

- Task: Identify quality issues in these requirements according to the requirements quality metrics: *ambiguity*, *inconsistency*, *forward referencing*, and *opacity*. If one requirement has several quality issues, list all of them. Then, try to fix the requirements quality issues of each requirement and write down the improved requirements.
- Example answers
- Ambiguity (1 point): So the trucks should have sensors to avoid collisions with obstacles because there are always other vehicles or obstacles in the motorway (Not explain obstacles).
  Your fix: Only need to make a short list of obstacles. It does not need to be a complete list.
- Inconsistency (1 point): E.g., We hope the truck platooning can drive along the route we set in GPS before the journey and follow the human drivers' command to change the route (Conflict on who can make the control decision).
  Your fix: Only need to eliminate the inconsistency.

An open-ended question example

Question: Explain what static backward slicing is and how to create backward slicing using data flow information.

Answers:

- The purpose of code slicing is to choose only a subset of code that is relevant to a particular variable at a certain point to minimize the code for more cost-efficient testing and debugging.

- Static backward slicing: A slice with respect to a variable *v* at a certain point *p* in the program is the set of statements that contributes to the value of the variable v at p.

- To establish static backward slicing, all Defs and All P-uses of a variable v before a certain point p are chosen.

Summary

- Use the slides as the table of content to read books and papers
- The content of the course, especially the chosen topics, varies each year. Not all questions from previous years are relevant to this year
- The case studies and open-ended questions in previous years' exams can be used for practice
- We will create a discussion channel in Blackboard for Q&A exam related questions

Where does code go when it is finished?

- A story about DevOps

About Me

- Platform Engineer at Vipps
  - Develop login-systems
- Very interested in software security
- **Twitter:** noratomas3
- **Linkedin:** Nora Tomas

NULL STRESS
VI TAR DET PÅ
vipps

About Me

- Studied Computer Science at Gløs
- Failed “a few” subjects in my time at NTNU
  - So don’t worry, you will be fine! :)

After this lecture I hope you will

- Know what a frontend and backend is
- Get a sense of what an API is
- Understand what deployment means and why it is important
- Know what DevOps means
- See (in the demo) what a DevOps engineer does
  - github.com/NoraTomas/DevOpsDemo

“Hello World”

BeginnersBook [~/PycharmProjects/BeginnersBook] - .../HelloWorld.py [Beg...

BeginnersBook 〉 HelloWorld.py 〉 HelloWorld

Pr...

HelloWorld.py

BeginnersBook

venv

HelloWorld.py

```
1  # This prints Hello World on the output screen
2  print('Hello World')
```

Run HelloWorld

```
/Users/chaitanyasingh/PycharmProjects/BeginnersBook/venv/bin/python /Us
Hello World

Process finished with exit code 0
```

1:47 n/a UTF-8

“Hello World”

- “Coding is not difficult” -> https://www.youtube.com/watch?v=hb7Q33ysCwI

“Hello World”

Time for a quick Kahoot!

Customer-Driven project - Localhost

Introducing Deployment

- “Software deployment is all of the activities that make a software system available for use”

Failed IT-projects - Forgotten Deployment

Forgetting deployment = forgetting to build the building

- Source: State of DevOps 2019 report
- Download here: https://cloud.google.com/devops/state-of-devops/

Agile Alignment Trap

- Companies get into the agile alignment trap

DevOps Engineer

- A lot of job postings for “DevOps Engineer”
- What does it really mean?

www.finn.no › Jobb › Ledige stillinger ▾ Translate this page

Ledige stillinger med søkeordet "devops" | FINN Jobb - FINN.no

Du finner 291 ledige stillinger med søkeordet "**devops**" på **FINN Jobb**. Søk blant alle typer **jobber** i hele **Norge**!

What is DevOps

To understand what DevOps engineer does...
Let us create and deploy an API!

Example: What is an API?

- API is a term used very loosely

Primary Secondary Success Danger Warning Info Light Dark Link

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>

<button type="button" class="btn btn-link">Link</button>
```

Example: What is an API?

- Most commonly, when people say API, they mean: HTTP endpoints (URLs) that you can use to retrieve information
- Example: https://openweathermap.org/current

API call

```
api.openweathermap.org/data/2.5/weather?q={city name}&appid=
{API key}
```

Let us make our own Weather API

- We use Java and Spring Boot
- What we will do
  - Show what a backend API looks like
  - Build a JAR file locally
  - Deploy the JAR file

Opening our API to the world

- Could do it from our own computer!
  - Need
    - Public IP
    - Open firewall in router
    - Make an “inbound connection” to port 8080
- Drawbacks from using our own computer
  - Your computer might become slow (with many connections)
  - A lot of other software taking space
  - Exposing your public IP to the world
  - You have to buy a new computer if you want more power
- A server is just a computer

SERVER :)

Let us deploy to EC2!

- EC2 is a virtual machine in AWS
- We can use that virtual machine - just like any other computer
- We just don’t own the hardware

We made a backend!

- Why is it a backend?
  - The Java-application is not supposed to run on users computers
  - End-users will not “download” this Jar-file and run it on their computers
  - We know what machine the Jar-file will run on, because we created it!
  - ***Front-end development:*** Making sure things work on different clients

Our Python application - was a frontend!

BeginnersBook [~/PycharmProjects/BeginnersBook] - .../HelloWorld.py [Beg...

BeginnersBook 〉 HelloWorld.py 〉 HelloWorld

Pr...

HelloWorld.py

BeginnersBook
venv
HelloWorld.py

```
1  # This prints Hello World on the output screen
2  print('Hello World')
```

Run HelloWorld

```
/Users/chaitanyasingh/PycharmProjects/BeginnersBook/venv/bin/python /Us
Hello World

Process finished with exit code 0
```

1:47 n/a UTF-8

Manual Deployment

- In the “old days” people did deployment like we do now
  - Developers would give operations a “manual” on what was required
- Many things can go wrong!
  - Operations installs the wrong version of Java on machines
  - Dependencies might not work well together
  - Operations can’t install what you need that easily (database software)
- Introducing Operations Engineer

Classic Failure - Dependencies

Before we build pipeline - quick recap!

- A frontend runs on the users computer
  - This is why frontend is more than just design, or HTML, or CSS
  - A console application (such as “Hello World” in Python) is also a type of frontend
- A backend runs on a computer that does not belong to the user. Usually it runs on a virtual machine, such as EC2 in AWS
- Backend and frontend communicate via the HTTP protocol
  - Also often called “An API”, but the term API is used very loosely and can mean almost anything
- A frontend and backend need to be deployed so that end users can interact with them
- The deployment process is very important and should not be left for last
- In our case, we deployed our backend by uploading the JAR-file to an EC2. We did this upload manually.
- Now, we want to automate the upload. This is where the pipeline comes in!

Pipelines - Automating Deployment

- A pipeline is a mechanism to move your “code” from a GitHub-repo into production
- Behind the scenes, a pipeline is just a computer that runs commands

Testing

- Testing is important because of automation
- Tests can be added to the pipeline
- Creating tests is like creating your first user

```
Run Test | Debug Test
class TestIncrementDecrement(object):
    Run Test | Debug Test
    def test_increment():
        assert increment(3) == 4

    Run Test | Debug Test
    def test_decrement():
        assert decrement(3) == 2
```

Thank you! :)

Control flow and data flow testing

Jingyue Li (Bill)

Outline

- ➢ Control flow testing and coverage
- Data flow testing and coverage
- Mutation testing and coverage

Industry standards define test needed

- IEEE 1012 – general software verification and validation
- IEC 61508 – general safety critical software
- ISO 26262 – automotive software
- Other domain specific standards, e.g.,
  - Healthcare
  - Oil & Gas
  - Railway

ISO 26262 – automotive software

The ASIL (Automotive System Integrity Level) – A, B, C, or D – is the outcome of the combination of three factors

- S – Severity. How dangerous is the event
- E – Probability. How likely is the event
- C – Controllability. How easy is it to control the event if it occurs

Finding the ASIL level

Severity	Probability	C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

ASIL chart*

ASIL chart

Functional category	Hazard	ASIL-A	ASIL-B	ASIL-C	ASIL-D
Driving	Sudden start				
	Abrupt acceleration				
	Loss of driving power				
Braking	Maximum 4 wheel braking				
	Loss of braking function				
Steering	Self steering				
	Steering lock				
	Loss of assistance				

* https://www.jnovel.co.jp/en/service/compiler/iso26262.html

Methods and measurements for software unit testing

Methods and measures	Sub-methods/Measures	ASIL			
		A	B	C	D
Functional tests	…	…			
Structural coverage	Statement coverage Decision coverage …	++ +	++ +	+ ++	+ ++
Resource usage measurement	…	+	+	+	++
Back-to-back test between simulation	…	+	+	+	++

Control flow coverage

- Statement coverage
- Decision coverage
- Path coverage

Statement coverage report

Coverage Report

http://www.cafeconlech

```
110 128            else if ( nav.isElement( first ) )
111                {
112 100                return nav.getElementQName( first );
113                }
114  28            else if ( nav.isAttribute( first ) )
115                {
116   0                    return nav.getAttributeQName( first );
117                }
118  28            else if ( nav.isProcessingInstruction( first ) )
119                {
120   0                    return nav.getProcessingInstructionTarget( first );
121                }
122  28            else if ( nav.isNamespace( first ) )
123                {
124   0                    return nav.getNamespacePrefix( first );
125                }
126  28            else if ( nav.isDocument( first ) )
127                {
128  28                return "";
129                }
130   0              else if ( nav.isComment( first ) )
131                {
132   0                    return "";
133                }
134   0              else if ( nav.isText( first ) )
135                {
136   0                    return "";
137                }
138                else {
139   0                    throw new FunctionCallException("The argument to the name
140                }
141            }
142
143   8        return "";
144
```

Done

Disabled

You can get test coverage report with [GitLab CI]

Coverage report

Coverage Report

http://www.cafeconlech

Packages

All
org.jaxen
org.jaxen.dom
org.jaxen.dom.html

All Packages

Classes

BaseXPath *(77%)*
Context *(93%)*
ContextSupport *(91*
DefaultNavigator *(3*
DocumentNavigator
DOMXPath *(100%)*
DocumentNavigator
HTMLXPath *(0%)*
NamespaceNode *(*
DocumentNavigator
Dom4jXPath *(100%*

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	205	69%	80%	2.811
org.jaxen	24	77%	73%	1.38
org.jaxen.dom	3	55%	60%	1.907
org.jaxen.dom.html	2	0%	0%	1.364
org.jaxen.dom4j	2	78%	85%	2.395
org.jaxen.expr	73	73%	84%	1.566
org.jaxen.expr.iter	14	98%	100%	1.029
org.jaxen.function	27	64%	76%	5.373
org.jaxen.function.ext	6	63%	72%	4.235
org.jaxen.function.xslt	1	86%	100%	2.5
org.jaxen.javabean	4	44%	72%	1.87
org.jaxen.jdom	3	62%	63%	2.897
org.jaxen.pattern	13	49%	52%	2.135
org.jaxen.saxpath	8	51%	81%	1.887
org.jaxen.saxpath.base	6	95%	100%	10.723
org.jaxen.saxpath.helpers	2	28%	83%	1.34
org.jaxen.util	15	41%	50%	2.432
org.jaxen.xom	2	71%	66%	1.783

Reports generated by Cobertura.

Done

Disabled

Test coverage at Google

- Per-project statement coverage
- 1.5 years worth of data across hundreds of projects

Probably by Apr. 2015

The median is 78%, the 75th percentile is 85% and 90th is percentile 90%.

Decision coverage

- Each one of the possible branch (true/false or switch-case) from each **decision** point is executed at least once

```
*   Read P
    Read Q

    If P+Q > 100 THEN
        Print "Large"
    ENDIF

    IF P > 50
       Print    " P Large"
    ENDIF
```

Statement coverage:
A, C, D, E, G, H

Decision coverage:
A, C, D, E, G, H
+
A, B, E, F, H

Branch coverage

- Count branches covered
- Slightly different from decision coverage

If 3 out of 4 branches of a switch statement are executed

- Branch coverage 75%
- Decision coverage 0% (a decision is considered covered only if all its branches are covered)

Path coverage

- All possible **paths** of the software should be tested

Decision coverage:
A, C, D, E, G, H
+
A, B, E, F, H

Path coverage:
A, C, D, E, G, H
+
A, B, E, F, H
+
A, B, E, G, H
+
A, C, D, E, F, H

White-box testing methods

- Usually need to generate control/data flow graph first using some tools, e.g., Java parser (Front End)*
- Then, use different test inputs to achieve different coverages
- Path coverage is more difficult to achieve
  - How many test cases will be sufficient?
    - McCabe's cyclomatic complexity can help
  - How to systematically create test cases?
    - Decision table can help

* http://www.semanticdesigns.com/Products/FrontEnds/JavaFrontEnd.html

McCabe's cyclomatic complexity

- From control flow graph of the program
- $v(G) = E - N + 2P$
- $v(G)$ = cyclomatic complexity
- E = the number of edges of the graph
- N = the number of nodes of the graph
- P = the number of connected components

Graph example

- $v(G) = E - N + 2P$
- E: edge, N:node, P: connected component

- E = 9
- N = 8
- P = 1

- V(G) = 9 - 8 + 2 x 1 = 3

Using v(G) to count number of paths

- The minimum number of paths through the code is v(G)

- As long as the code graph is a DAG – Directed Acyclic Graph – the maximum number of paths is 2**|{predicates}| (when all predicates have only two possible values)

- Thus, we have that

  V(G) ≤ number of paths ≤ 2**|{predicates}|

Simple case - 1

S1;
IF P1 THEN S2 ELSE S3
S4;

E = 4, N = 4, P = 1, E-N+2P = 2

- V(G) = 2,
- One predicate (P1)
- Two test cases can cover all paths

Simple case – 2

S1;
IF P1 THEN X := a/c ELSE S3;
S4;

- V(G) = 2,
- One predicate (P1)
- Two test cases will cover all paths but not all cases. What about the case c = 0?

Nested decisions

```
S1;
IF P1 THEN S2 ELSE
    S3;
    IF P2 THEN S4 ELSE S5
FI
S6;
```

v(G) = 3, while Max = 4.
Three test case will cover all paths.

NTNU

Problem – the loop

```
S1;
DO
    IF P1 THEN S2 ELSE S3;
    S4
OD UNTIL P2
S5;
```

Not a Directed Acyclic Graph
v(G) = 3 and Max
is 4 but there is an “infinite”
number of paths

Problem – the loop (cont')

- Loops are the great problem in white box testing
- It is common practice to test the system going through each loop
  - 0 times – loop code never executed
  - 1 time – loop code executed once
  - 5 times – loop code executed several times
  - 20 times – loop code executed "many" times

Use decision table to generate test cases for path coverage

- Make a table of all predicates
- Insert all combinations of True / False/Switch – 1 / 0/many – for each predicate
- Construct a test for each combination

- A general technique used to achieve full path coverage
- However, in many cases, it may lead to over-testing

Using a decision table

P1	P2	P3	Test description or reference
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Decision table example – binary

P1	P2	Test description or reference
0	0	S1, S3, S5, S6
0	1	S1, S3, S4, S6
1	0	S1, S2, S6
1	1	S1, S2, S6

Go to **www.menti.com** and use the code **5729 3680**

Exercise

Outline

- Control flow testing and coverage
- ➢ Data flow testing and coverage
- Mutation testing and coverage

Data flow testing

- Data flow testing uses the control flow graph to explore the unreasonable things that can happen to data (data flow anomalies)

- Data flow anomalies are detected based on the associations between definition, initialization, and use of variables, e.g.,
  - Variables are used without being initialized

Variable lifecycle

- Variables have a life cycle
  - Defined and initialized
    - A value is bound to the variable
    - E.g., x = …
  - Used
    - The value of the variable is referred
    - Predicate use (p-use), E.g., if (x>10)
    - Computational use (c-use). E.g., y = x + 1;
  - Killed (destroyed)

Variable lifecycle (cont')

```
{                       // begin outer block
    int x = 1;          // x is defined as an integer within this outer block
        ...;            // x can be accessed here
    {                   // begin inner block
     int y = 2;         // y is defined within this inner block
        ...;            // both x and y can be accessed here
     }                  // y is automatically destroyed at the end of this block
    ...;                // x can still be accessed, but y is gone
}                       // x is automatically destroyed
```


NTNU

Data flow anomalies and normal use

- **dd: define and then define again – suspicious**
- **dk: define and then kill – potential bug**
- **ku: kill and then used – serious defect**
- **kk: kill and then kill again – potential bug**
- du: define and then use – OK
- kd: kill and then redefine – OK
- ud: use and then redefine – OK
- uk: use and then kill – OK
- uu: use and then use – OK
- …

Why data-flow testing?

- To uncover possible bugs in data usage during the execution of the code
- Test cases are created to trace every definition to each of its use, and every use is traced to each of its definition
- Various coverage strategies are employed for the creating of the test cases

Data-flow test strategies

- All definitions (AD): every definition of every variable should be covered by at least one use (c-use or p-use) of that variable
- All computational uses (ACU): for every variable, there is a path from every definition to every c-use of that definition
- All predicate-uses (APU): for every variable, there is a path from every definition to every p-use of that definition
- All c-uses/some p-uses (ACU+P): for every variable and every definition of that variable, at least one path from the definition to every c-use should be included. If there are definitions of the variable with no c-use following it, then add p-use test cases to cover every definition

Data-flow test strategies (cont')

- All p-use/some c-use (APU+C): for every variable and every definition of that variable, at least one path from the definition to every p-use should be included. If there are definitions of the variable with no p-use following it, then add c-use test cases to cover every definition
- All uses (AU): for every variable, there is a path from every definition to every use of that definition, i.e., APU + ACU
- All du paths (ADUP): test cases cover every simple sub-path from each variable definition to every p-use and c-use of that variable
- Note that the "kill" usage is not included in any of the test strategies

All definitions

- All definitions (AD): every definition of every variable should be covered by at least one use (c-use/p-use) of that variable.

*

AD

- (1, 2, x)
- (3, 8, a)
- (7, 8, a)
- (6, 5, x)

Test cases for AD coverage

x = 0, a = any (1, 2, 4, 5, 6, 5, 7, 8)

x = 1, a = any (1, 2, 3, 4, 8)

*https://www.cs.ccu.edu.tw/~naiwei/cs5812/st5.pdf

All computational uses

- All computational uses (ACU): for every variable, there is a path from every definition to every c-use of that definition

ACU

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)
- (3, 8, a)
- (7, 8, a)

All predicate-uses

- All predicate-uses (APU): for every variable, there is a path from every definition to every p-use of that definition

APU

- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

All c-uses/some p-uses

- All c-uses/some p-uses (ACU+P): for every variable and every definition of that variable, at least one path from the definition to every c-use should be included. If there are definitions of the variable with no c-use following it, then add p-use test cases to cover every definition

ACU

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)
- (3, 8, a)
- (7, 8, a),

ACU + P

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)
- (3, 8, a)
- (7, 8, a)

All p-use/some c-use

- All p-use/some c-use (APU+C): for every variable and every definition of that variable, at least one path from the definition to every p-use should be included. If there are definitions of the variable with no p-use following it, then add c-use test cases to cover every definition

APU
- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

VS.

APU + C
- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

- (3, 8, a)
- (7, 8, a)

All uses

- All uses (AU): for every variable, there is a path from every definition to every use of that definition, i.e., APU + ACU

AU

- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

APU

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)

- (3, 8, a)
- (7, 8, a)

ACU

All du paths

- All du paths (ADUP): More demanding than All uses (AU). If there are multiple paths between a given definition and a use, they must all be included
- However, ADUP includes loop just once

- In AU, we include path (3, 8, a), meaning either (3, 4, 5, 7, 8) OR (3, 4, 8) OR (3, 4, 5, 6, 5, 7, 8) is ok
- In ADUP, since there are multiple paths from 3 to 8, we need to include (3, 4, 5, 7, 8) AND (3, 4, 8) AND 3, 4, 5, 6, 5, 7, 8)

Go to **www.menti.com** and use the code **5729 3680**

Exercise

Relationship between strategies

Why do we measure test coverage?

- One way of using statement/branch/path/dataflow coverage is to use it as a test acceptance criteria
  - Run test cases
  - Have we reached, e.g., 85% statement coverage
    - Yes – stop testing
    - No – coverage measure will help us directly identify untested code and indirectly help develop/select new test cases

Use test coverage to estimate defect coverage

- Defect coverage: the fraction of actual defects initially present that would be detected by a given test set

Enumerable ***i***

- Blocks/statements
- Branches
- C-uses
- P-uses

Logarithmic-exponential relationship between test coverage and defect coverage

Use coverage for program slicing

- Static backward slicing
  - A slice with respect to a variable *v* at a certain point *p* in the program is the set of statements that **contributes** to the value of the variable v at p
  - *S(v, p)* denotes the set of nodes in the control flow graph that contributes to the value of the variable v at point p

Static backward slicing

S(x, 1) = {1}

S(x, 2) = {1}

S(x, 3) = {1, 2}

S(x, 4) = {1, 2}

S(x, 5) = {1, 2, 4}

S(x, 6) = {1, 2, 4, 5, 6}

S(x, 7) = {1, 2, 4, 5, 6}

All Defs and All P-uses of a variable v before a certain point p are used for slicing*

* http://www.cs.cmu.edu/afs/cs/academic/class/17654-f01/www/refs/Binkley.pdf

Program slicing can help many activities

- Facilitate debugging (Finding and localizing errors)
- Facilitate programming understanding

S(x, 6) = {1, 2, 4, 5, 6}

Outline

- Control flow testing and coverage
- Data flow testing and coverage
- ➢ Mutation testing and coverage

Mutation testing and coverage

- The tester can also measure coverage of seeded faults and use it as an indicator to measure whether the test set is adequate
  - Seed faults in the code
  - Have we found all the seeded faults
    - Yes – test set is adequate
    - No – One or more seeded faults are not found. This tells us which locations and types of defects have not been covered by test cases sufficiently. Then we need to develop/select more test cases.

Fault seeding and estimation

- We assume
  $N_0 / N = S_0 / S$
- Thus
  $N_0 = N * S_0 / S$,

- N_0: number of faults in the code
- N: number of faults found using a specified test set

- S_0: number of seeded faults
- S: number of seeded faults found using a specified test set

Typical mutants

- Where and how to seed the faults
  - Save and seed faults identified during earlier project activities
  - Draw faults to seed from an experience database containing typical faults
    - Arithmetic operator replacement
    - Logical connector replacement
    - Relational operator replacement
    - Absolute value insertion, etc.
- More info. can be found at
  - http://www.uio.no/studier/emner/matnat/ifi/INF4290/v10/undervisningsmateriale/INF4290-Mutest.pdf
  - https://www.softwaretestinghelp.com/what-is-mutation-testing/
  - https://pitest.org/java_mutation_testing_systems/

Code to be tested

- This is what the code might look like:

  1) *Read Age*
  2) *If age>14*
  3) *Doctor= General Physician()*
  4) *End if*

Seed faults (mutants)

- Mutation type 1: Relational operator replacement

  Mutant #1:

  1) Read Age
  2) If age<14 'Changing the > with <'
  3) Doctor= General Physician()
  4) End if

  Mutant #2:

  1) Read Age
  2) If age=14 'Changing the > with ='
  3) Doctor= General Physician()
  4) End if

Mutation coverage

- If our test inputs are 14 and 15. Which of these mutants will be killed?

Test inputs	Expected result	Mutant 1 Changing > with <	Mutant 2 Changing > with =
14	GP not assigned	Succeeds - GP not assigned	Fails - GP assigned
15	GP assigned	Fails - GP not assigned	Fails - GP not assigned

- 14 finds a failure when runs again mutant 2, not effective against mutant 1
- 15 finds failures when runs again mutants 1 and 2

Summary

- We studied
  - Control flow test and related coverage measurement
  - Data flow test and related coverage measurement
  - Mutation test and coverage measurement

Next lecture

- Black-box unit testing

Domain and function testing

Methods and measures for functional software integration testing

Methods and measures	Criteria	ASIL			
		A	B	C	D
Requirements based test	Test case have to be derived from the requirements. All SW component requirements are covered	++	++	++	++
External interface test	Test cases can be determined with the help of equivalence classes and input partitioning of the external interface. This can be completed by boundary value analysis. Boundary values for data types as well as plausible ranges of values for interface variables have to be considered.	+	++	++	++
Fault injection test	…	+	+	++	++
Error guessing test	…	+	+	++	++

Outline

- ➢ Single variable
- Multiple variables
- Error guessing/Risk based
- Output coverage and testing

The challenge

The total number of possible combinations of X and Y is $2^{32} \times 2^{32}$

Combinatorial explosion!

- We need to systematically reduce the set of all possible values to few manageable subsets

Partitioning of domain

- Identifying domain first
  - Linear (variables whose values can be mapped onto a number line, e.g., integer variables)
  - Non-linear (e.g., variables with enumeration values)

- Ideally, partitioning of input & output domain should result in non-overlapping or disjoint partitions

Black-box domain testing approaches

- Identify input and output domains

- Generate test case to cover single variables in the domains and combination of the variables
  - Equivalence classes, boundary values
  - Decision tables, pairwise testing
  - Error guessing/risk-based

Partitioning of domain of single variable

- Variable types
  - Linear domain variable (e.g., integer)
  - Linear domain variable with multiple ranges
  - String variable
  - Enumerated variable
  - Multidimensional variable

Partitioning domain of single variable

- Step 1: Identify the variable
- Step 2: Determine the domain
  - All possible values of the variable
- Step 3: Identify risks
- Step 4: Partition the domain into equivalence classes based on the identified risks

Linear domain variable

- Software application example

  BankA issues Visa credit cards with credit limits in the range of $4000 to $40000. A customer is not to be approved for credit limits outside this range.

  A customer can apply for the card using an online application form in which one of the fields requires that the customer type in his/her desired credit limit.

Linear domain variable (cont')

- Identify the variable: "credit-limit"
- Determine the input domain: $4000 =< credit-limit <= $40000
- Identify risks
  - Failure to process credit limit requests between $4000 and $40000 correctly
  - Failure to disapprove credit limit requests less than $4000
  - Failure to disapprove credit limit requests greater than $40000
  - Mishandling of negative credit limit requests
- Partition the input domain into equivalence classes based on risks
  - $4000, $40000
  - $3999
  - $4001
  - -$4000

Equivalence class testing

- Rationale: complete testing but no redundancy
- Cover each partition at least once
- E.g., to test whether software can identify the equilateral triangle
  - After using (5, 5, 5) as test input
  - We do not expect to learn much from using (6, 6, 6), (10, 10, 10) as test inputs

Linear domain variable with multiple ranges

If Taxable Income Is Between:	The Tax Due Is:
0 - $9,225	10% of taxable income
$9,226 - $37,450	$922.50 + 15% of the amount over $9,225
$37,451 - $90,750	$5,156.25 + 25% of the amount over $37,450
$90,751 - $189,300	$18,481.25 + 28% of the amount over $90,750
$189,301 - $411,500	$46,075.25 + 33% of the amount over $189,300
$411,501 - $413,200	$119,401.25 + 35% of the amount over $411,500
$413,201 +	$119,996.25 + 39.6% of the amount over $413,200

Linear domain variable with multiple ranges (cont')

- Identify risks
  - Failure to calculate the tax correctly for each of the income sub-ranges
  - Mishandling of low and high boundaries of each of the sub-ranges
  - Mishandling of values just beneath and beyond low and high boundaries respectively for each of the sub-ranges

Equivalence class testing vs. boundary value testing

Figure 6.1 Traditional equivalence class test cases.

*

* Software Testing, A craftsman's approach, 4th Edition

Boundary value testing

- Rationale: Errors tend to occur near the extreme values of an input variable, e.g.,
  - Loop condition, may test for < when they should test ≤
- The idea is to use input values

  Min, min+, nom, max-, max

- Two variables (hold one in nom and vary the other one)

  $<X_{1nom}, X_{2min}>$ $<X_{1nom}\ X_{2min+}>$ $<X_{1nom}, X_{2nom}>$ $<X_{1nom}, X_{2max-}>$ $<X_{1nom}, X_{2max}>$

  $<X_{1min}, X_{2nom}>$ $< X_{1min+}\ X_{2nom}>$ $< X_{1max-}, X_{2nom}>$ $< X_{1max}, X_{2nom}>$

- Several variables (hold one in nom and vary the others)

Robust boundary value testing

Figure 6.1 Traditional equivalence class test cases.

*

* Software Testing, A craftsman's approach, 4th Edition

Special value testing

- Special value testing of the tax calculation system
  - Mishandling of non-numbers
  - Mishandling of negative incomes
  - Mishandling of smallest and largest value at the system level, and beyond
  - …
- Another special value testing example
  - E.g., test how the system react if *29 Feb.* as the input of a date
- Most intuitive and least uniform (ad hoc)

String variables

- For example, a string variable ‘s’ has to have a letter for the first character; the rest of them can be any printable graphic ASCII characters.

- Domain
  - First character

    (A-Z) 65 <= ASCII (first character) <= 90

    (a-z) 97 <= ASCII (first character) <= 122
  - Remaining characters

    32 <= printable graphic ASCII characters <= 126

String variables (cont')

- Identify risks
  - Failure to process strings correctly that have letters as their first character (B3, Zb)
  - Failure to process boundary characters (A~) (the ASCII code of ~ is 126)
  - Mishandling of string values that have nonletters for the first character ([A)
  - Mishandling of characters that belong to the extended ASCII set (Aβ)

Enumerated variable

- A variable which takes only a set of values/set of options

- Risks
  - Failure to present font effect correctly that has a Superscript/Subscript/… option selected
  - Mishandling of no option selected
  - Mishandling of multiple options selected

Multidimensional variables

- Multidimensional variable

  There is more than one dimension of analyzing such a variable

- Software application example

  To log in a system, the user must input the correct username and password. The username can have five to fifteen characters. Also, only digits and lowercase characters are allowed for the username.

- Username variable has two dimensions

  Length and String

Menti Exercise

Go to **www.menti.com** and use the code **6438 5825**

Outline

- Single variable
- ➢ Multiple variables
- Error guessing/Risk based
- Output coverage and testing

Multivariable application example (1)

- Boundary value testing <1, 0, 1> <1, 0, 2><1, 0, 50> <1, 0, 99> <1, 0, 100>
  <1, 1, 1> <1, 1, 2><1, 1, 50> <1, 1, 99> <1, 1, 100> ...
- Equivalence class testing <1, 0, 1> <1, 1, 0> <101, 0, 1> <101, 1, 0> ...

- Also need to analyze complex logical relationships of the input variables

Decision table-based testing (cont')

<4, 1, 2> <3, 3, 3> <2, 2, 3>

Table 7.2 Decision Table for Triangle Problem

c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F	F
c3: a = c?	—	T	T	F	F	T	T	F	F
c4: b = c?	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

*

* Software Testing, A craftsman's approach, 4th Edition

Why test variable combination?

- All variables will interact as they are part of one functional unit and they have to unite to achieve the duty that the functional unit is designated for
- Hence most of these variables influence each other in some, or the other way
- Testing variables in combination may reveal certain bugs that might not have been found by testing the variables in isolation

Multivariable application example (2)

- Online shopping system example
  - Parameters: Availability, Payment Method, Carrier, Delivery Schedule, Export Control

Availability	Payment	Carrier	Delivery Schedule	Export Control
Available	Credit	Mail	One Day	True
Not in Stock	Paypal	UPS	2-5 Working Days	False
Discontinued	Gift Voucher	Fedex	6-10 Working Days	
No Such Product			Over 10 Working Days	

4x3x3x4x2 = 288 combinations

The challenge of testing the online shopping system example

- Suppose there is a bug, and Credit does not work well with One Day delivery
  - Any combination of inputs that include Credit and a One Day delivery will expose that bug
  - Interaction between 2 variables
- Suppose Credit does not work well with a One Day delivery, but only with Fedex
  - Any combination of inputs that include Credit, a One Day delivery, and not with Fedex will expose that bug
  - Interaction between 3 variables
- Do we really need to test all 4x3x3x4x2 = 288 combinations?

Do we really need to test all combinations?

- The root cause analysis of many bugs shows they depend on the value of one variable (20%-68%)
- Most defects can be discovered in tests of the interactions between the values of two variables (65-97%)

Number of variables involved in triggering software faults*

Vars	Medical Devices	Browser	Server	NASA GSFC	Network Security
1	66	29	42	68	20
2	97	76	70	93	65
3	99	95	89	98	90
4	100	97	96	100	98
5		99	96		100
6		100	100		

*http://csrc.nist.gov/groups/SNS/acts/ftfi.htm

What is n-way test?

- Given any n variables (out of all the variables) of a system, every combination of values of these n variables is covered in at least one test

- For example, an application has 5 variables
  - 2-way test

    Every combination of values of any 2 variables (out of 5 variables) is covered in at least one test
  - 3-way test

    Every combination of values of any 3 variables (out of 5 variables) is covered in at least one test

What is n-way test (cont')?

- An application with 5 input variables A, B, C, D, E
  - A has values A1, A2
  - B has values B1, B2, B3,
  - C has values C1, C2
  - D has values D1, D2, D3, D4
  - E has values E1, E2
- Full combinations 2x3x2x4x2 = 96 test cases
- 2-way test (56 combinations)
  - Every combination of values of any 2 variables
  - AB, AC, AD, AE, BC, BD, BE, CD, CE, DE
  - AB (A1-B1, A1-B2, A1-B3, A2-B1, A2-B2, A2-B3)
  - AC (A1-C1, A1-C2, A2-C1, A2-C2)
  - …

All-Pairs combination testing

- 2-way testing
  - Every combination of values of 2 variables is covered in at least one test

- All-pair combination/pairwise testing is condensed 2-way testing
  - Many pairs can be tested at the same time in one combination

A1-B1, B1-C2, C2-D2 -> A1-B1-C2-D2

All-pair combination/pairwise testing approaches

- In Parameter Order
- Variation of In Parameter Order

In Parameter Order

- Initialization phase
- Horizontal growth phase
- Vertical growth phase

Example of In Parameter Order

- Variables
  - X (True, False)
  - Y (0, 5)
  - Z (P, Q, R)

In Parameter Order - Initialization

- X (True, False)
- Y (0, 5)

Find any two variables and generate test cases

- T= (True, 0)
  (True, 5)
  (False, 0)
  (False, 5)

In Parameter Order - Horizontal growth

- X (True, False)
- Y (0, 5)
- Z (P, Q, R)

- Add one more variable in each step to generate more combination
- Remove duplications

All 2-way test combinations

(True, 0)
(True, 5)
(False, 0)
(False, 5)

+

P
Q
R
P?Q?R?

vs.

~~(True, P)~~ ~~(True, Q)~~, (True, R)
~~(False, P)~~, (False, Q), ~~(False, R)~~
~~(0, P)~~, (0, Q), ~~(0, R)~~
~~(5, P)~~, ~~(5, Q)~~, (5, R)

In Parameter Order - Vertical growth

Merge short combinations to longer ones

A Variation of In Parameter Order

- Initialization phase
  - Order variables based on the number of their possible values
  - Generate test cases by full combinations of the first two variables
- Vertical growth phase
  - Add other variables one-by-one to generate a combination
  - Re-order certain variable values if needed
- Horizontal growth phase
  - If re-order cannot generate enough new combinations
  - Extend horizontally to add possibility of using re-ordering to generate new combinations

A Variation of In Parameter Order - Example

- A car ordering application *

- Variables
  - Order category (Buy, Sell)
  - Location (Oslo, Trondheim)
  - Car brand (BMW, Audi, Mercedes)
  - Registration numbers (Valid, Invalid)
  - Order type (E-Booking, In-store)
  - Order time (Working hours, Non-working hours)

*www.softwaretestinghelp.com/what-is-pairwise-testing/

Variation of In Parameter Order – Initialization (reorder variables)

- Order variables based on their number of values

- The results after ordering
  - **Car brand (BMW, Audi, Mercedes)**
  - Order category (Buy, Sell)
  - Location (Oslo, Trondheim)
  - Registration numbers (Valid, Invalid)
  - Order type (E-Booking, In-store)
  - Order time (Working hours, Non-working hours)

Vertical growth phase – iteration 1

Car Brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy				
BMW	Sell				
Audi	Buy				
Audi	Sell				
Mercedes	Buy				
Mercedes	Sell				

Vertical growth phase – iteration 2

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo			
BMW	Sell	Trondheim			
Audi	Buy	Oslo			
Audi	Sell	Trondheim			
Mercedes	Buy	Oslo			
Mercedes	Sell	Trondheim			

Vertical growth phase – iteration 2 (cont')

- After re-ordering

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo			
BMW	Sell	Trondheim			
Audi	Buy	Trondheim			
Audi	Sell	Oslo			
Mercedes	Buy	Oslo			
Mercedes	Sell	Trondheim			

Vertical growth phase – iteration 3

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid		
BMW	Sell	Trondheim	Invalid		
Audi	Buy	Trondheim	Valid		
Audi	Sell	Oslo	Invalid		
Mercedes	Buy	Oslo	Valid		
Mercedes	Sell	Trondheim	Invalid		

Vertical growth phase – iteration 3 (cont')

- After re-ordering

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid		
BMW	Sell	Trondheim	Invalid		
Audi	Buy	Trondheim	Valid		
Audi	Sell	Oslo	Invalid		
Mercedes	Buy	Oslo	Invalid		
Mercedes	Sell	Trondheim	Valid		

Vertical growth phase – iteration 4

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	
BMW	Sell	Trondheim	Invalid	E-booking	
Audi	Buy	Trondheim	Valid	E-booking	
Audi	Sell	Oslo	Invalid	In Store	
Mercedes	Buy	Oslo	Invalid	E-booking	
Mercedes	Sell	Trondheim	Valid	In Store	

Vertical growth phase – iteration 5

If re-order cannot generate enough new combinations

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	Working hours
BMW	Sell	Trondheim	Invalid	E-booking	Non-working hours
Audi	Buy	Trondheim	Valid	E-booking	Working hours
Audi	Sell	Oslo	Invalid	In Store	Non-working hours
Mercedes	Buy	Oslo	Invalid	E-booking	Working hours
Mercedes	Sell	Trondheim	Valid	In Store	Non-working hours

Horizontal growth

Extend horizontally to add possibility of using re-ordering to generate new combinations

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	Working hours
BMW	Sell	Trondheim	Invalid	E-booking	Non-working hours
	Buy				Non-working hours
Audi	Buy	Trondheim	Valid	E-booking	Working hours
Audi	Sell	Oslo	Invalid	In Store	Non-working hours
	Sell				Working hours
Mercedes	Buy	Oslo	Invalid	E-booking	Working hours
Mercedes	Sell	Trondheim	Valid	In Store	Non-working hours

All-pairs combination testing tools

- ACTS – ‘Advanced Combinatorial Testing System’, provided by NIST, an agency of the US Government

- VPTag free Pairwise Testing Tool

Outline

- Single variable
- Multiple variables
- ➢ Error guessing/Risk based
- Output coverage and testing

Error guessing / Risk-based

- Based on experience
  - Different types of faults
  - Probability of the faults
  - Consequence of the faults
  - …

Bach's risk-based testing*

- Bach's heuristics is based on his experience as a tester
- Two risk analysis approach
  - Inside-out: What can go wrong there?
  - Outside-in: What things are associate with this kind of risk?
- Based on this experience he has identified
  - A generic risk list – things that are important to test
  - A risk catalogue – things that often go wrong to a specific domain

*http://www.satisfice.com/articles/hrbt.pdf

Bach’s generic risk list

- Complex – large, intricate or convoluted
- New – no past history in this product
- Changed – anything that has been tampered with or “improved”
- Upstream dependency – a failure here will cascade through the system
- Downstream dependency – sensitive to failure in the rest of the system
- Critical – a failure here will cause serious damage

Bach's generic risk list (cont')

- Precise – must meet the requirements exactly
- Popular – will be used a lot
- Strategic – of special importance to the users or customers
- Third-party – developed outside the project
- Distributed – spread out over time or space but still required to work together
- Buggy – known to have a lot of problems
- Recent failure – has a recent history of failures

Bach’s risk catalogs example

- Wrong files installed
  - Temporary files not cleaned up
  - Old files not cleaned up after upgrade
  - Unneeded file is installed
  - Needed file not installed
  - Correct file installed in the wrong space
- Files clobbered
  - Old file replaces newer file
  - User data file clobbered during upgrade
- …

Outline

- Single variable
- Multiple variables
- Random testing
- Error guessing/Risk based
- ➢ Output coverage and testing

Output coverage

- All the coverage types that we have looked at so far have been related to input data
- It is also possible to define coverage based on output data
- Uncovered outputs help us define the new test cases

Output coverage (cont')

Output coverage (cont”)

- The main challenge with using output coverage measure is that output can be defined at several levels of details
  - An account summary
  - An account summary for a special type of customer
  - An account summary for a special event

Summary

- We studied
  - How to identify the domains of input & output variables
  - How to derive function test cases based on single and multiple domain variables
- Can be applied to
  - Unit test
  - Integration test
  - System test
  - Acceptance test

Integration, system, acceptance, and regression testing

V-model

Business requirements
Value realization test
Deployment test
Software requirements
Operation test
System test
We cover these tests
Regression test
Start - criteria
Workflow
Design
Integration test
External consistency
Unit design
Unit test
Internal consistency
Testing
Programmering

Integration testing

- Focus on testing interfaces between components
- Not as well understood as unit testing and system testing
- Usually poorly done

- Strategies
  - Decomposition-based
  - Call graph-based/interface matrix
  - Path-based

Decomposition-based integration

- Based on functional decomposition tree
- Need to know function dependencies between components
- Based on the order of integration testing, can be classified into four strategies
  - Top-down
  - Bottom-up
  - Sandwich
  - Big bang

Decomposition-based integration (cont')

- Top-down
  - Begin with main
  - Use "stubs" to simulate called functions
  - Replace stubs with real functions one by one

- Bottom-up
  - Begin with leaves
  - Use "drivers" to emulate functions call the leaves
  - Replace "drivers" with real function later

- Sandwich
- Big bang

Pros and cons of decomposition-based integration

- Pros
  - Incremental and intuitive
  - Easy fault isolation

- Cons
  - Need “stub” or “driver”

Call graph-based/interface matrix-based integration

- Based on the call graph/interface matrix of components
- Use actual components rather than “stubs” or “drivers”
- Two strategies
  - Pairwise integration
  - Neighborhoods integration

Pair-wise integration

*

Figure 13.9 Three pairs for pairwise integration.

* Software Testing, A craftsman's approach, 4th Edition

Interface matrix

	Sys 1	Sys 2	Sys 3	Sys 4
Sys 1	X	X		
Sys 2	X	X		
Sys 3			X	X
Sys 4			X	X

Neighborhoods integration

Figure 13.10 Three neighborhoods (of radius 1) for neighborhood integration.

Table 13.1 Neighborhoods of Radius 1 in Calendar Call Graph

Neighborhoods in Calendar Program Call Graph			
Node	*Unit Name*	*Predecessors*	*Successors*
1	Calendar (Main)	(None)	2, 3, 4, 5, 6, 7
2	getDate	1	8, 9
3	zodiac	1	9

* Software Testing, A craftsman's approach, 4th Edition

Pros and cons of call graph-based integration

- Pros
  - Do not need “stub” or “driver”
- Cons
  - Limited to local integration

Path-based integration

- Not focusing on testing interfaces among separately developed and tested unit
- Rather focus on interactions among these units

Path-based integration (cont’)

- Like enlarged unit testing
  - The node is not a statement
  - The node is a component/test unit
  - The edge is message transferred between components

Pros and cons of path-based integration

- Pros
  - Test more global and complex integrations
  - Closely coupled with actual system behavior

- Cons
  - Difficult fault isolation
  - Extra effort is need to identify message path

System tests categories

- Type of system tests
  - Functionality
  - Reliability
  - Usability
  - Performance
  - Robustness
  - Scalability
  - Stress
  - Load and stability
  - …

Robustness testing

- How sensitive is a system to erroneous inputs or changes in the operational environment?
  - Stupid/uncommon inputs (like testing special values in boundary testing)
  - Failures from other systems
  - Degraded node
  - Etc.

Scalability testing

- To identify how well the system can scale, i.e., the magnitude of demand that can be placed on the system while continuing to meet performance requirements

Stress testing

- It can ensure the system can perform acceptably under the worse-case condition

- The system is deliberately stressed by pushing it to and beyond its specific limits for a while

- For testing
  - Buffer allocation and memory carving
  - Network bandwidth

CULTURE

Olympics ticket system crashes

Domestic ticket sales in China are halted after 8 million hits and 200,000 orders per second overwhelm the booking system.

BY SUZANNE TINDAL | NOVEMBER 1, 2007 6:18 AM PDT

Load and stability testing

- Important for providing 24x7 services
- System and applications that run for months are likely to
  - Slow down
  - Encounter functionality problems
  - Silently failover
  - Crash altogether
- Ensure the system remains stable for a long period of time under full load

Scalability vs. Stress vs. Load and statbility testing

- Scalability

- Stress

- Load and stability

Using Apache JMeter to run load and performance testing

https://youtu.be/yNnbW2n9s8E

Acceptance testing

- ISTQB : **(user) acceptance testing**: Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

Typical forms of acceptance testing

User acceptance testing

- Functions are correct?
- Test basis
  - User/business requirements
  - System requirements
  - Use cases
  - Business processes
  - Risk analysis reports

Operational acceptance test

- Read to operate?
  - Backup facilities
  - Procedures for disaster recovery
  - Training or manual for end-users
  - Maintenance procedures and manual
  - Security procedures

Contract and regulation acceptance testing

- Test against contract

- Test against regulations
  - Governmental regulation
  - Legal standards
  - Safety standards

Alpha and beta testing

- Alpha
  - At developers' sites
  - By internal staff
  - Before it is released to external customers

- Beta (field testing)
  - At customers' sites
  - Before the system is released to other customers

Acceptance testing in agile

- The terms "functional test", "acceptance test" and "customer test" are used more or less interchangeably.

- Referring to user stories

Outline

- Testing approaches
  - Integration tests
  - System tests
  - Acceptance tests
- ➢ Regression tests

Regression testing

- Basic regression testing concepts
- Regression test case selection
- Regression minimization
- Regression test case prioritization

What is regression testing?

- Constitute the vast majority of testing effort in many software development projects

Two types of regression testing

- Corrective regression testing
  - No requirements change
  - Modified code behaves correctly
  - Unchanged code continues to behave correctly

- Progressive regression testing
  - Requirements change
  - Newly added or modified code
  - Unchanged code continues to behave correctly

Overview of first & regression test cases*

Test code that is unchanged and not impacted by the changed code, unnecessary to run in regression testing, but can be run if you like

Test changed code and code impacted by the changed code, must run in regression testing

* https://www.uio.no/studier/emner/matnat/ifi/INF4290/v11/undervisningsmateriale/INF4290-RegTest.pdf

Regression testing processes

- Test revalidation
  - Are test cases obsolete?
- (Regression) test selection
  - Verify changed and impacted code (corrective)
  - Verify new structure and requirements (progressive)
- Test minimization
  - Remove redundant test cases
- Test prioritization
  - Rank test cases and run them according to available resources

Test revalidation

- How can a test case be obsolete?
  - Their input/output relation is no longer correct due to changes in specification
  - They no longer test what they were designed to test due to modifications to the program
  - They are “structural” test cases that no longer contribute to structural coverage of the program

(Regression) test selection

- We now consider only corrective regression testing
- Strategies of (regression) test selection
  - Retest all
    - Can be costly, unless test execution is highly automated
  - Random selection
    - Better than no test
    - Hard to ensure coverage of the changed code
  - Selecting modification traversing tests
    - “Safe” regression test selection technique is preferred

Automated regression testing example

https://youtu.be/2jAy9cdblaE

Safe regression test selection

A technique that does not discard any test that will traverse a modified or impacted statement is known as "safe" regression test selection technique

- Assume program P has been tested by test set T against specification S
- Assume P is modified to P' to fix some bugs (i.e., specification S is not changed)

Question:

- What information is needed in order to select a safe regression test subset from T to verify P'?

General (regression) test selection process

- Establish traces between P and T
  - Execute P with test cases
  - Record program entities executed when running tests
- Compare P with P' to find differences
  - Identify program entities changed
- Select test cases from T that traverse the changed program and impacted entities

General (regression) test selection process (cont')

- Based on program entities traced and compared, the test selection methods can be classified into
  - Dynamic slicing based approach (statements)
  - Graph-walk approach (nodes in control flow graph/program dependence graph)
  - Firewall approach (OO classes, COTS components, etc.)
  - …

Graph-walk approach example

Code P

```
int  M (int x, int y){
  int z;
  if (x<y)
     z = f1 (x, y);
  else
     z = f2(x, y);
  return z;
}
```

```
int f1( int a, int b){
  if ((a+1) == b)
     return a*a;
  else
     return b*b; }
```

```
int f2( int a, int b){
  if (a == (b+1))
     return b*b;
  else
     return a*a; }
```

Code P’

If((a-1) == b)

Test set T of P

t1: <x = 1, y = 2>

t2 : <x = 1, y = 3>

t3 : <x = 3, y = 1>

Which test cases to be included in the safe subset to test P’?

Graph-walk approach example – step 1

- Establish trace between test case and CFG (control flow graph) nodes

Test set T

t1: <x = 1, y = 2>

t2 : <x = 1, y = 3>

t3 : <x = 3, y = 1>

Nodes traversed by tests

Functions	Nodes			
	1	2	3	4
M	t1, t2, t3	t1, t2	t3	t1, t2, t3
f1	t1, t2	t1	t2	-
f2	t3	None	t3	-

Graph-walk approach example – step 2

- Compare P and P' to find differences

If((a-1) == b)

Node 1 in f1

Graph-walk approach example – step 3

- Select test cases from T that traverse the changed CFG nodes

Test set T

t1: <x = 1; y = 2>

t2 : <x = 1, y = 3>

t3 : <x = 3, y = 1>

Set of tests that traverse a certain node

Functions	Nodes			
	1	2	3	4
M	t1, t2, t3	t1, t2	t3	t1,t2, t3
f1	t1, t2	t1	t2	-
f2	t3	None	t3	-

A technique that does not discard any test that will traverse a modified or impacted statement is known as “safe” regression test selection technique.

- Here, we did not cover the impacted statement
- Test oracles of the tests may need to be changed

General (regression) test selection process (cont')

- Based on program entities traced and compared, the test selection methods can be classified into
  - Dynamic slicing based approach (statements)
  - Graph-walk approach (nodes in control flow graph/program dependence graph)
  - Firewall approach (OO classes, COTS components, etc.)
  - …

Firewall approach

What is a firewall?

- A ***firewall*** separates the classes that depend on the class that is changed from the rest of the classes

Firewall approach (cont')

Why use a firewall?

- The ***firewall*** approach is based on the first-level dependencies of modified components

- The most severe effects are in those components that send messages to the changed component or receive messages from the changed component

Process for determining “Firewall” in OO systems

1. Identify classes changed

2. Inheritance? Then descendants of the changed classes

3. Classes send messages to or receive messages from the changed class

One empirical study of using firewall approach [1]

- System studied
  - Distributed component-based J2EE sys. owned by Swedbank
  - 1.2M line of code from 27 000 classes
  - 2 new releases of the system

- Steps 1: Coverage analysis and establishing trace
  - Instrumentation of Java byte code and run-time collection of coverage data during original test
  - Used a tool called Emma [2]

One empirical study of using firewall approach [1] (Cont')

Step 2: Find classes in firewall

- Identify changes – produced MD5 signatures for each compiled class file. If code changes, class signature will be different.
- Own tool

Step 3: Extracting dependency between classes

- Dependency finder [3]

Step 3: Select test cases that traverse classes in firewall

Another empirical study of using firewall approach [4]

- Studied an OO telecommunication software system at ABB

- Compared two firewall approaches
  - One level from changed component
  - More than one level from the changed component, depending on their logical dependencies

Another empirical study of using firewall approach [4] (Cont')

- Observation
  - Extended firewall approach finds more faults than classical firewall approach, but more costly to run

- Recommendations
  - Routine incremental changes (one level approach)
  - Major release (> one level approach)

Test minimization

- To reduce redundancies in the safe subset
- If every entity covered by t2 is a subset of entities covered by another test case t1, then we remove t2
- The entities could be
  - Statement, CFG nodes, functions, etc.

Test minimization example

- t1 covers the following statement
- t2 also covers the same statement
- Then, we can remove t2

(a<b)||(a<c)

Risk of test minimization

- Test minimization is **risky** and is not necessarily safe
  - It depends on the modifications (from P to P'), the faults, the entities used

(a<b)||(a<c)

t2 may test the (a<c) part and t1 may test the (a<b) part

If the t2 is removed because this statement is covered by t1, then, we will miss the opportunity to test (a<c)

Test prioritization

- Different from test minimization

- Test prioritization
  - Is not going to remove any test cases from the safe subset
  - Is to rank regression tests based on some criteria

- The goal is to reveal faults early

Test prioritization approaches

- Unlike test prioritization of the first test
  - **More information available**
  - Fault finding effectiveness
  - Test coverage
  - Cost of running the test

- Prioritization strategies [5]
  - Coverage based (high coverage)
  - Cost-aware based (low cost to run, high fault-finding probability)
  - …

Summary

- We studied the following testing approaches
  - Integration
  - System
  - Acceptance

- We studied regression test
  - Selection
  - Minimization
  - Prioritization

References

- [1] White L., Abdullah K. A firewall approach for the regression testing of object-oriented software. Proc. of the Software Quality Week, 1997.
- [2] http://emma.sourceforge.net/
- [3] Skoglund M. and Runeson P. A Case study of The Class Firewall Regression Test Selection Technique on Large Scale Distributed Software System. Proc. of empirical software engineering conf. 2015.
- [4] White L. Jaber K. Robinson B. and Rajich V. Extended firewall for regression testing: an experience report. Journal of software maintenance and evolution. Vol. 20. 2008
- [5] Yoo S., Harman M. Regression testing minimization, selection and prioritization: a survey, Software testing, verification, and reliability, 2007.

NTNU

System Testing Planning

Theory and Practice

Chapter 12

System Test Planning and Automation

AGENDA

1

Testing Planning

2

Testing Execution

Why do we have to plan tests?

- The purpose is to get ready and organized for test execution
- Without a plan it is highly unlikely that
  - the desired level of system testing is performed within the stipulated time and without overusing resources
  - Expected Costs and deadlines will be reached
- Planning for system testing IS part of the overall planning for a software project!
- It provides the framework, scope, resources, schedule, and budget for the system testing part of the project.

A System Test Plan

- Provides guidance for the executive management to support the test project
- Establishes the foundation of the system testing part of the overall software project
- Provides assurance of test coverage by creating a requirement traceability matrix
- Outlines an orderly schedule of events and test milestones that are tracked
- Specifies the personnel, financial, equipment, and facility resources required to support the system testing part of a software project

Planning the Test (Outline)

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Introduction

The *introduction* section of the system test plan includes:

- Test project name
- Revision history
- Terminology and definitions
- Name of the approvers and the date of approval
- References
- Summary of the rest of the test plan

The *feature description* section summarizes the system features that will be tested during the execution of this plan.

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Assumptions

- The *assumptions* section describes the areas for which test cases will not be designed
- Examples:
  - The necessary equipments to carry out scalability testing may not be available
  - It may not be possible to procure third-party equipments in time to conduct interoperability testing
  - It may not be possible to conduct compliance test for regulatory bodies and environment tests in the laboratory

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Approach

- Issues discovered by customers that were not caught during system testing in the past project
- Outstanding issues that need to be tested differently need to be discussed here
- Discuss test automation strategy
- Test cases from the database that can be re-used in this test plan

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Suite Structure

- Detail *test groups* and *subgroups* based on the test categories identified in the *test approach* section
- Test objectives are created for each test group and subgroups based on the system requirements and functional specification documents
- Identification of test objectives provides a clue to the total number of test cases needs to be developed
- A traceability matrix is generated to make an association between requirements and test objectives to provide the test coverage

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Environment

Multiple test environments are constructed in practice

Test activities can not interfere with development activities

To run security tests you need an environment that can be put down.

To run scalability tests one may need more resources than to run functionality tests

Test Environment

Test Bed Preparations

Get Information about the customer deployment architecture including hardware, software and their manufacturers

List of third-party products to be integrated with the SUT.

Third party test tools to be used to monitor, simulate, and /or generate real traffic.

Hardware equipment necessary to support special features specified in the requirements, for example backup, some network switches etc.

Analysis of the non-functional requirements for checking what is necessary to perform these tests

List of small but necessary networking, cables hubs, needed for the test.

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Execution Strategy

- It’s a game plan that addresses the following concerns:
  - How many times are the tests cases executed and when?
  - What does one do with the failed test cases?
  - Which test cases groups are blockers?
  - What happens when too many test cases fail?
  - In which order are the test cases executed?

Test Execution Strategy

- The processes of system test execution, defect detection, and fixing defects are intricately intertwined
  - Some test cases cannot be executed unless certain defects are detected and fixed
  - A programmer may introduce new defects while fixing one defect, which may not be successful
  - The development team releases a new build for system testing by working on a subset of the reported defects, rather than all the defects
  - It is a waste of resources to run the entire test set T on a build if too many test cases fail
  - When the system is more stable it may be wise to start prioritizing the tests for regression tests

An effective and efficient execution strategy must take into account the characteristics of the test execution, defect detection and defect removal!

A Multi-Cycle System Test Strategy

(a) Progress of system testing in terms of test cycles

(b) Desired increase in quality level

Characterization of Test Cycles

- Goals and Assumptions
- Test execution
- Revert and Extension criteria
- Actions
- Exit criteria

Characterization of Test Cycles

Goals and Assumptions

- **System test team sets its own goals to be achieved in each test cycle**
- **Goals are specified in terms of the number of test cases to pass in a cycle;**
- **Assumptions specify some "tolerance level for the test"**

Characterization of Test Cycles

Test Execution

- **Test Cases are executed in multiple test environments as specified in the test plan.**
- Prioritization of the test execution changes between test cycles.
- **Some good practices for prioritization of test cases**
  - **Test cases that exercise basic functionalities have higher priority than the rest in the first test cycle**
  - **Test cases that have failed in one test cycle have a higher priority in the following test cycle**
  - **Test cases in certain groups have higher priority than others**

Characterization of Test Cycles

Revert and Extension criteria

- **The conditions for prematurely terminating a test cycle and for extending a test cycle must be precisely stated**
- **It may not be useful to continue a test cycle if it is found that a software is of poor quality**
- **Often a test cycle is extended due to various reasons**
  - A need to re-execute all the test cases in a particular test group because a large fraction of the test cases within the group failed
  - A significantly large number of new test cases were added while test execution was in progress

Characterization of Test Cycles

Actions (Examples)

- Too many test cases may fail during a test cycle
  - **The development team needs to be alerted**
  - **The developers should take action in the form of root cause analysis (RCA)**
  - **Corrective actions are taken by updating the design specification, reviewing the code, and adding new test cases to the unit and integration test plan**
- The system test team has to design a large number of new test cases during a test cycle
  - **Tester have then to see why this happened.**
    - Mistmatches between what the tester expected and what was delivered?
    - Poor plannning of the tests ?
    - Different expectations of the tests objectives for that cycle?

Characterization of Test Cycles

Exit Criteria

- **An exit criterion specifies the termination of a test cycle**
  - Mere execution of all the test cases in a test suite does not mean that a cycle has completed maybe there is a need to add more “quality” criteria:
    - a percentage of passed/not passed
    - Number of environment that is acceptable
- **Example of exit criteria**
  - **95% of test cases passed and all the known defects are in CLOSED state**

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Effort Estimation

Cost of the test and the time to complete the test

- The number of test cases to be designed for the software project based on:
  - Test Group Category
  - Function Points
- Effort required to create a detailed test case
- Effort required to execute a test case and analyze the results of execution
- Effort needed to create the test environments
- Effort needed to report the defects, reproduce the failures and follow up with development team

Test Effort Estimation

TABLE 12.6 Test Effort Estimation for FR–ATM PVC Service Interworking

Test (Sub)Groups	Estimated Number of Test Cases	Person-Day to Create	Person-Day to Execute
Configuration	40	4	6
Monitoring	30	4	4
Traffic Management	3	2	3
Congestion	4	2	4
SIWF Translation Mode Mapping	12	4	4
Alarm	4	1	1
Interface	9	5	3
Robustness	44(4 + 40)	4	6
Performance	18	6	10
Stress	2	1.5	2
Load and stability	2	1.5	2
Regression	150(60 + 90)	0	15
Total	**318**	**35**	**60**

Scheduling and Test Milestones

ID	Task Name	Duration	Start	Finish
1	Development of the test plan	5 days	Jan 3	Jan 7
2	Review the test plan	0 days	Jan 7	Jan 7
3	Update the test plan	1 day	Jan 10	Jan 10
4	Test plan approved	0 days	Jan 10	Jan 10
5	Procurement of equipment	7 days	Jan 11	Jan 19
6	Set up the test environment	10 days	Jan 20	Feb 2
7	Creation of test cases in test factory	18 days	Jan 11	Feb 3
8	Review the test cases	0 days	Feb 3	Feb 3
9	Update the test cases in test factory	4 days	Feb 4	Feb 9
10	Test cases are in released state	0 days	Feb 9	Feb 9
11	Creation of test suite in test factory	1 day	Feb 10	Feb 10
12	Entrance criteria readiness meeting	1 day	Feb 11	Feb 11
13	Official release of S/W to system test	0 days	Feb 14	Feb 14
14	Start of first test cycle	0 days	Feb 14	Feb 14
15	First test cycle	15 days	Feb 14	Mar 4
16	End of first test cycle	0 days	Mar 4	Mar 4
17	Start of second test cycle	0 days	Mar 4	Mar 4
18	Second test cycle	15 days	Mar 7	Mar 25
19	End of second test cycle	0 days	Mar 25	Mar 25
20	Beta release criteria review meeting	0 days	Mar 25	Mar 25
21	Release the software to beta customer	1 days	Mar 28	Mar 28
22	Start of third test cycle	0 days	Mar 28	Mar 28
23	Third test cycle	5 days	Mar 29	Apr 4
24	End of third test cycle	0 days	Apr 4	Apr 4
25	Test report preparation	4 days	Apr 5	Apr 8
26	Release criteria review meeting	0 days	Apr 8	Apr 8
27	Release the software to the customer	1 day	Apr 13	Apr 13

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Menti?

AGENDA

1

Testing Planning

2

Testing Execution

NTNU

System Test Execution

Theory and Practice

Chapter 13

System Test Execution

Outline

- Lifecycle of a Defect
- Metrics for Monitoring Test execution
- Metrics for Monitoring Defect Reports
- Defect Analysis Techniques:
- Measuring Test Effectiveness
- System Test Report

Test Execution

- It is important to monitor the processes of test execution, tracking defects and measure test effectiveness.
  - There is a pressure to meet a tight schedule close to the delivery date;
  - There is a need to discover most of the defects before delivering the product;
  - It is essential to verify that defect fixes are working and have not resulted in new defects

Lifecycle of a Defect

- The life-cycle of a defect is usually simplistically represented by a state-transition diagram with five states:
  - NEW
    - A problem report with a severity level and priority is filled
  - ASSIGNED
    - The problem is assigned to an appropriate person
  - OPEN
    - The assigned person is actively working on the problem to resolve it
  - RESOLVED
    - The assigned person has resolved the problem and is waiting for the submitter to verify and close it.
  - CLOSED
    - The submitter has verified the resolution of the problem and accepted it.

Lifecycle of a Defect Jira

https://www.educba.com/jira-bug-life-cycle/

Defects Characteristics

Priority - A Measure of how soon the defect needs to be fixed (urgency)

Critical – must be resolved as soon as possible

High – must be resolved in a high priority

Medium – must be resolved but there is no pressure of time

Low – should be resolved when possible.

Severity - A measure of the extent of the detrimental effect of the defect on the operation of the product.

Critical : when one or more critical system functionalities are impaired and there is no workaround;

High: some functionalities of the system are impaired but there are workarounds;

Medium : no critical functionalities of the system is impaired.

Low: the problem is mostly cosmetic.

Defects Characteristics Jira

View Priorities

The table below shows the priorities used in this version of JIRA, in order from highest to lowest.

- Translate priorities

Name	Description	Icon	Color	Order	Operations
Highest	This problem will block progress.				Edit · Delete · Default
High	Serious problem that could block progress.				Edit · Delete · Default
Medium	Has the potential to affect progress.				Edit · Delete · Default
Low	Minor problem or easily worked around.				Edit · Delete · Default
Lowest	Trivial problem with little or no impact on progress.				Edit · Delete · Default

Add New Priority

Name *

Description

Icon URL * [select image]

(relative to the JIRA web application e.g /images/icons OR starting with http://)

Priority Color *

Add

https://www.cwiki.us/display/JIRA064/Defining+Priority+Field+Values

Defects Characteristics Jira

https://www.softwaretestinghelp.com/jira-bug-tracking/

Metrics for Tracking the Test Execution

- Test execution brings forth 3 different facets of software development:
  - Developers would like to know the degree to which the system meets the explicit as well as implicit requirements.
  - The delivery date cannot be precisely predicted due to the uncertainty in fixing problems.
  - The customer is excited to take the delivery of the product.

Metrics for Tracking the Test Execution

- It is very important to monitor certain metrics which truly represent the progress of system testing and reveal the quality level of the system!
  - Monitoring the test execution
  - Monitoring the defects
- This is very important for timely decisions about the test!

Metrics for Monitoring Test Execution

Test case Escapes (TCE)

- A significant increase in the number of NEW test cases during execution implies that deficiencies in the test design

Planned versus Actual Execution (PAE) Rate

- Compare the actual number of of test cases executed every week with the planned number of test cases

Execution Status of Test (EST) Cases

- Periodically monitor the number of test cases lying in different states
  - *Failed, Passed, Blocked, Invalid* and *Untested*

Monitoring Test Execution Jira

project IN (BM, AB) AND issuetype = Bug and assignee=currentuser() and priority in (blocker, critical) and status in ("To Do", open) Basic

1–9 of 9

T	Key	Summary	Assignee	Reporter	P	Status	Resolution	Created	Updated
	BM-132	Test	Swati Seela [Administrator]	Swati Seela [Administrator]		OPEN	Unresolved	17/Feb/16	03/Mar/16
	AB-38	Rest_validatAccount_creditcard_ ; credit card validation gives incorrect error messgae when Expiry date is prior than the present date.	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-37	Update functionality for SSN,accets input'0'with success,but does not update in DB	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-35	Timeout Error for server socket	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-29	Buy credit card accepts negative amount and it's adding in available limit in database	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-27	In CC card restful service "Add account" Credit limit ,available limit and cash limit accepts 0 and show "msg": "Scuccessfully saved credit card information".	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-26	In CC restful service "add account" CC expiry date should be greater than the current date but acccepts the date prior to the current date.	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-19	If we give non matching name and cedit card number i.e name of one person and registered credit card of other person the amount in the request get deposited in the account of name provided.	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15

https://www.softwaretestinghelp.com/jira-bug-tracking/

Metrics for Monitoring Defect Reports

- Function as Designed (FAD) Count
  - False positives
  - If the number of false positives are high, maybe the testers have inadequate understanding of the system.
- Irreproducible Defects (IRD) Count
  - If a defect cannot be reproduced, then the developers may not be able to gain useful insight into the cause of the failure.
  - If many defects are irreproducible, may mean that the system is unreliable for some reason.
- Defects Arrival Rate (DAR) Count
  - Measures who is finding the defects. For example, if the marketing and users are the ones that are mostly reporting bugs, it may mean that the test is ineffective.

Metrics for Monitoring Defect Reports

- Outstanding Defects (OD) Count
  - Defects that are always coming back. Or never resolved.
  - If defects are not resolved and only increasing, it may mean that the system is not increasing the quality level.
- Crash Defects (CD) Count
  - Defects causing a system to crash must be recognized as an important category of defects because a system crash is a serious event leading to complete unavailability of the system and possibly loss of data.
- Lead time to Resolution of Defects Count
  - If the defects that appear in the system are usually easy to fix and fast to fix it means that the system has better quality.

Orthogonal Defect Classification (ODC)

- A methodology for rapid capturing of the semantics of each software defect.
- In the NEW state, the submitter needs to fill out the following ODC attributes or fields:
  - *Activity:* This is the activity that was being performed at the time the defect was discovered
  - *Trigger:* The environment or condition that had to exist for the defect to surface
  - *Impact:* This refers to the effect, the defect would have on the customer, if the defect had escaped to the field.
- The owner needs to fill out the other ODC attributes or fields when the defect is moved to RESOLVED state.
- The ODC analysis can be combined with Pareto analysis to focus on error-prone parts of the software
  - 80% of the problems can be fixed with 20% of the effort

Orthogonal Defect Classification

- Target: The target represents the high-level identity, such as design, code, or documentation, of the entity that was fixed
- Defect type: The defect type represents the actual correction that was made
- Qualifier: The qualifier specifies whether the fix was made due to missing, incorrect, or extraneous code
- Source: The source indicates whether the defect was found in code developed in house, reused from a library, ported from one platform to another, or provided by a vendor
- Age: The history of the design or code that had the problem. The age specifies whether the defect was found in new, old (base), rewritten, or refixed code.
    - New: The defect is in a new function which was created by and for the current project.
    - Base: The defect is in a part of the product which has not been modified by the current project. The defect was not injected by the current project
    - Rewritten: The defect was introduced as a direct result of redesigning and/or rewriting of old function in an attempt to improve its design or quality.
    - Refixed: The defect was introduced by the solution provided to fix a previous defect.

Monitoring Test Execution Jira

https://www.softwaretestinghelp.com/jira-bug-tracking/

Defect Causal Analysis (DCA)

Used to raise the quality level of the products at a lower cost.

Focuses on understanding of cause-effect relationship of an error to

- Take actions to prevent similar errors from occurring in the future;
- Reduce the number of defects;
- Focus on systematic errors;

Defect Causal Analysis (DCA) Five Steps

Cause-effect diagram for defect causal analysis (DCA

Defect Causal Analysis (DCA)

- Examples of Recommended Actions can come from the DCA:
  - Training
    - People improving their knowledge about the product
  - Improvement in Communication
    - Communication between testers and developers can be of a big help in cases of too many false positives or duplicated defects reporting for example.
  - Using Tools
    - Recommend the use of tools to monitor memory for example
  - Process improvements
    - Actions to improve existing processes or defining new processes.

https://www.knowledgehut.com/blog/agile/root-cause-analysis-agile-teams

Defect Causal Analysis (DCA) in Agile

- Best Place for this are on retrospectives!

https://www.benlinders.com/2013/getting-to-the-root-causes-of-problems-in-a-retrospective/

System Test Report

A final summary report is created after the completion of all the tests

System Test Report

1. Introduction to the Test Project
2. Summary of Test Results
3. Performance Characteristics
4. Scaling Limitations
5. Stability Observations
6. Interoperability of the System
7. Hardware/Software Compatible Matrix
8. Compliance Requirement Status

Table 13.14: Structure of the final system test report.

System Test Report

Testflo example

https://deviniti.com/support/addon/server/testflo-83/latest/test-execution-report/#&gid=null&pid=2

Readiness Criteria Example

- All the test cases from the test suite should have been executed
- Test case results are updated with Passed, Failed, Blocked, or Invalid status
- The requirements are updated by moving each requirement from the Verification state to either the Closed or the Decline state, as discussed in Chapter 11
- The pass rate of test cases is very high, say, 98%
- No crash in the past two weeks of testing has been observed
- No known defect with critical or high severity exists in the product
- Not more than a certain number of known defects with medium and low levels of severity exist in the product
- All the resolved defects must be in the CLOSED state
- The user guides are in place
- Trouble shooting guide is available
- The test report is completed and approved

Measuring Test Effectiveness

- After a product is deployed measure the number of defects found by the customers that were not found by the testers and development team.
  - ESCAPED DEFECTS

Defect Removal Efficiency (DRE) metric defined as follows:

$$\mathrm{DRE} = \frac{\text{Number of Defects Found in Testing}}{\text{Number of Defects Found in Testing} + \text{Number of Detects Not Found}}$$

Measuring Test Effectiveness with Spoilage Metric

A new metric called spoilage is defined as

$$\text{Spoliage} = \frac{\sum(\text{Number of Defects} \times \text{Discovered Phage})}{\text{Total Number of Defects}}$$

Measuring Test Effectiveness with Spoilage Metric

Phase Injected	Phase Discovered								Weight	Total Defects	Spoilage = Weight/Total Defects
	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing			
Requirements	0	7	6	3	0	0	12	28	56	17	3.294117647
High-Level		0	8	8	3	8	30	6	63	22	2.863636364
Detailed Design			0	13	6	12	20	0	51	25	2.04
Coding				0	63	48	111	48	270	136	1.985294118
Summary	0	7	14	24	72	68	173	82	440	200	2.2

A spoilage value close to 1 is an indication of a more effective defect discovery process

How is this performed in Agile?

Discuss in groups how to do a test planning in testing management tool you know.

You can also use the example of TestFlo

https://deviniti.com/support/addon/server/testflo-84/latest/test-planning/

Further Material

How is this performed in Agile?

Certified Tester

Foundation Level Extension Syllabus
Agile Tester

Version 2014

International Software Testing Qualifications Board

ISTQB

Copyright Notice
This document may be copied in its entirety, or extracts made, if the source is acknowledged.

https://www.istqb.org/downloads/send/5-foundation-level-agile-tester/41-agile-tester-extension-syllabus.html

Further work

A complete quiz about Agile Testing from ISTB with 40 Questions

https://softwaretester.net/istqb_sample_exam_agile_tester_fl/

Code refactoring and review

Outline

- Code refactoring principles
  - Why refactoring
  - Bad smells in code
  - General refactoring principles and techniques
- Refactoring of Python code
- Code inspection and review
  - Testing vs. inspection
  - Reading techniques

Code refactoring principles

Readable and maintainable code is desired by industry

For software development (S), general programming knowledge (GP) and maintenance (MT) are the top two items.

GP: Clean and readable code with good documentation

MT: experience, especially with maintaining code

What competence do software companies want

Tor Stålhane, IDI, NTNU

Guttorm Sindre, IDI, NTNU

Extended version of conference paper "Hva vil programvareindustrien ha?", MNT-konferansen 2019

Stålhane, Deraas, Sindre and Abrahamsson:

Bad smells in code (code smells)

- Code smells are any violation of fundamental design principles that decrease the overall quality of the code.
- Not bugs or errors
- Can certainly add to the chance of bugs and failures down the line.

Code smells categories

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List

- Divergent Change
- Shotgun Surgery

- Feature Envy
- Data Clumps
- Primitive Obsession

- Switch Statements
- Lazy Element
- Speculative Generality
- Temporary Field

- Message Chains
- Middle Man

- Data Class
- Refused Bequest

Cod refactoring goals and properties

- Change the internal structure without changing external behavior

- Eliminate code smells for
  - Readability
  - Consistency
  - Maintainability

- Properties
  - Preserve correctness
  - One step at a time
  - Frequent testing

Code refactoring steps

- Designing solid tests for the section to be refactored
- Reviewing the code to identify bad smells of code
- Introducing refactoring and running tests (One step at a time)

Refactoring risks and countermeasures

- Refactoring is regarded as an overhead activity
- Introducing failures with refactoring
- Outdated comments and documents

- Balance cost and benefits
- Better to start from day one! Not at the end of the project/delivery
- Should have sufficient and efficient regression tests

Relevant books

Old edition, available online at NTNU lib.

New edition

Remove Duplicated Code – Pull up method (Don’t repeat yourself (DRY))

- You have methods with identical results on subclasses.

Remove Duplicated code - Substitute Algorithm

- You want to replace an algorithm with one that is clearer.

```
String foundPerson(String[] people){

  for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don"))
      { return "Don"; }

       if (people[i].equals ("John"))
       { return "John"; }

       if (people[i].equals ("Kent"))
       { return "Kent";  }
    }
    return "";
}
```

```
String foundPerson(String[] people){
     List candidates = Arrays.asList(new
String[] {"Don", "John", "Kent"});

  for (int i=0; i<people.length; i++)
       if (candidates.contains(people[i]))
              return people[i];

    return "";

}
```

Reduce size – shorten method/class

- **Long Method**
  - E.g., Extract method
- **Large Class**
  - E.g., Extract class, subclass, interface

Extract method example

If you have to spend effort looking at a fragment of code and figure out what it is doing, then you should extract it into a function/method and name it after "*what*."

```
void printOwing()
{
  printBanner(); //print details
  System.out.println ("name:
            " + _name);
  System.out.println ("amount
            " + getOutstanding());
}
```

```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}

void printDetails (double outstanding) {
  System.out.println ("name:" + _name);
  System.out.println ("amount:" +
outstanding);
}
```

Reduce size – shorten parameter list

- **Long Parameter List**
  - E.g., Introduce Parameter Object

Divergent Change

- **Code smell**
  - One module is often changed in different ways for different reasons.
  - Classes have more than distinct responsibilities that it has more than one reason to change
  - Violation of single responsibility design principle

- **Refactoring**
  - You identify everything that changes for a particular cause and put them all together

```java
public class Account {
    private int accountNumber;
    private double balance= 0;

    public Account(int accountNumber){
        this.accountNumber = accountNumber;
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void credit(double amount) {
        balance += amount;
    }

    public void debit(double amount) {
        balance -= amount;
    }

    public String toXml() {
        return "<account><id>" + Integer.toString(getAccountNumber()) + "</id>" +
                "<balance>" + Double.toString(getBalance()) + "</balance></account>";
    }
}
```

More than one responsibility, operation of a bank account + serializing the account

https://www.slideshare.net/nadeembtech/code-craftsmanship

```java
public class Account {
    private int accountNumber;
    private double balance= 0;

    public Account(int accountNumber){
        this.accountNumber = accountNumber;
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void credit(double amount) {
        balance += amount;
    }

    public void debit(double amount) {
        balance -= amount;
    }
}
```

```java
public class AccountXMLSerializer {

    public String toXml(Account account) {
        return "<account>" +
                "<id>" + Integer.toString(account.getAccountNumber()) + "</id>" +
                "<balance>" + Double.toString(account.getBalance()) + "</balance>" +
                "</account>";
    }
}
```

https://www.slideshare.net/nadeembtech/code-craftsmanship

Shotgun Surgery

- **Code smell**
  - A single change is made to multiple classes simultaneously
  - When changes are all over the place, they are hard to find, and it’s easy to miss important changes.

Graph from refactoring.guru

- **Refactoring**
  - You put all the changes into a single class
  - Once and only once (OAOO)

```java
package com.example.codesmell;

public class Account {

	private String type;
	private String accountNumber;
	private int amount;

	public Account(String type,String accountNumber,int amount)
	{
		this.amount=amount;
		this.type=type;
		this.accountNumber=accountNumber;
	}


	public void debit(int debit) throws Exception
	{
		if(amount <= 500)
		{
			throw new Exception("Mininum balance shuold be over 500");
		}

		amount = amount-debit;
		System.out.println("Now amount is" + amount);

	}

	public void transfer(Account from,Account to,int cerditAmount) throws Exception
	{
		if(from.amount <= 500)
		{
			throw new Exception("Mininum balance shuold be over 500");
		}

		to.amount = amount+cerditAmount;

	}

	public void sendWarningMessage()
	{
		if(amount <= 500)
		{
			System.out.println("amount should be over 500");
		}
	}

```

When we add another criterion in the validation logic: if the account type is personal and the balance is over 500

We have to make changes to all methods

https://dzone.com/articles/code-smell-shot-surgery

```java
package com.example.codesmell;

public class AcountRefactored {

        private String type;
        private String accountNumber;
        private int amount;



        public AcountRefactored(String type,String accountNumber,int amount)
        {
                this.amount=amount;
                this.type=type;
                this.accountNumber=accountNumber;
        }

        private boolean isAccountUnderflow()
        {
              return amount<=500;

        }


        public void debit(int debit) throws Exception
        {
                if(isAccountUnderflow())
                {
                        throw new Exception("Mininum balance shuold be over 500");
                }

                amount = amount-debit;
                System.out.println("Now amount is" + amount);

        }

        public void transfer(AcountRefactored from,AcountRefactored to,int cerditAmount) throws Exception
        {
                if(isAccountUnderflow())
                {
                        throw new Exception("Mininum balance shuold be over 500");
                }

                to.amount = amount+cerditAmount;

        }
```

Menti: code refactoring performed?

Go to **menti.com**, code **6344 0068**

```java
public class Account {
    double principal,rate;   int daysActive,accountType;

    public static final int STANDARD=0, BUDGET=1,
            PREMIUM=2, PREMIUM_PLUS=3;
}

...

public static double calculateFee(Account[] accounts)
{
    double totalFee= 0.0;
    Account account;
    for (int i=0;i<accounts.length;i++) {
        account=accounts[i];
        if (account.accountType== Account.PREMIUM||
                account.accountType== Account.PREMIUM_PLUS)
            totalFee+= .0125 * (       // 1.25% broker's fee
                    account.principal* Math.pow(account.rate,
                            (account.daysActive/365.25))
                            -account.principal);   // interest-principal
    }

    return totalFee;

}

```

```java
/** An individual account.  Also see CorporateAccount. */

public class Account {

    private double principal;
    /** The yearly, compounded rate (at 365.25 days per year). */

    private double rate; /** Days since last interest pay out */
    private int daysActive;
    private Type type;
}

/** The varieties of account our bank offers. */
public enum Type {STANDARD, BUDGET, PREMIUM, PREMIUM_PLUS}


/** Compute interest. **/
public double interest() {
    double years = daysActive / 365.25;
    double compoundInterest = principal * Math.pow(rate, years);
    return compoundInterest - principal;
}

/** Return true if this is a premium account. **/
public boolean isPremium() {
    return accountType == Type.PREMIUM ||
            accountType == Type.PREMIUM_PLUS;
}

/** The portion of the interest that goes to the broker. **/
public static final double BROKER_FEE_PERCENT = 0.0125;

/** Return the sum of the broker fees for all the given accounts. **/
public static double calculateFee(Account accounts[]) {
    double totalFee = 0.0;
    for (Account account : accounts) {
        if (account.isPremium()) {
            totalFee += BROKER_FEE_PERCENT * account.interest();
            }
        }
    return totalFee;
    }
}
```

http://www.ifi.uzh.ch/seal/teaching/courses/softwareWartung-FS16/CodeReviewExercise.pdf*

Increase cohesion

- **Feature envy**
  - A function in one module spends more time communicating with functions or data inside another module than it does within its own module.
  - Move function to give it a dream home

https://waog.wordpress.com/2014/08/25/code-smell-feature-envy/

Increase cohesion (Cont')

- **Data clumps**
  - Bunches of data often hang around together
  - Consolidate the data together, e.g., Introduce Parameter Object or Preserve Whole Object

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan =
plan.withinRange(daysTempRange());
```

Primitive Obsession

- Primitive fields are basic built-in building blocks of a language, e.g., int, string, or constants

- Primitive Obsession is when the code relies too much on primitives and when uses primitive types to represent an object in a domain

```php
class contactUs
{
    public function addressUsa()
    {
        $address = new Array();
        $address['streetNo'] = 2074;
        $address['streetName'] = 'JFK street';
        $address['zipCode'] = '507874';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function addressGermany()
    {
        $address = new Array();
        $address['streetNo'] = '25';
        $address['streetName'] = 'Frankfurter str.';
        $address['zipCode'] = '80256';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function hotLine(){
        return '+49 01687 000 000';
    }
}
```

The address is defined as an array. Every time we need the address we will have to hard code it

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

```php
class address{

    private $streetNo;
    private $streetName;
    private $zipCode;

    public function addressUsa()
    {
        $this->streetNo = 2074;
        $this->streetName = 'JFK street';
        $this->zipCode = '507874';

        return $this->streetName. ' '. $this->streetNo. ', '. $this->zipCode;
    }
    public function addressGermany()
    {
        $this->streetNo = 25;
        $this->streetName = 'Frankfurter str.';
        $this->zipCode = '80256';

        return $this->streetName. ' '. $this->streetNo. ', '. $this->zipCode;
    }
}
```

We create a new class called Address
Every time we need to add/edit an address we hit the Address class

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

Code smells categories

- **Duplicated Code**
- **Long Method**
- **Large Class**
- **Long Parameter List**

Decrease size

- **Divergent Change**
- **Shotgun Surgery**

Localize change

- **Feature Envy**
- **Data Clumps**
- **Primitive Obsession**

Increase cohesion

- **Switch Statements**
- **Lazy Element**
- **Speculative Generality**
- **Temporary Field**
- **Message Chains**
- **Middle Man**

Simplify logic within and between objects

- **Data Class**
- **Refused Bequest**

Proper OO practice

Simplify logic within object – Switch statement

- Replace Conditional with Polymorphism

```
double getSpeed() {
  switch (_type) {
   case EUROPEAN:
     return getBaseSpeed();
   case AFRICAN:
     return getBaseSpeed() - getLoadFactor() ;
    case NORWEGIAN_BLUE:
      return  getBaseSpeed(_voltage);
    }
   throw new RuntimeException (“error");
}
```

Simplify logic within object – Switch statement (cont')

- Replace Parameter with Explicit Methods

```
void setValue (String name, int value) {

  if (name.equals("height")) {
      _height = value;
      return;
  }

  if (name.equals("width")) {
      _width = value;
     return;
  }

   Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg) {

  _height = arg;

 }

void setWidth (int arg) {

   _width = arg;

}
```

Simplify logic within object – Lazy Element

- E.g., Lazy class: A class that isn't doing enough to pay for itself should be eliminated

```
class Person {
    get officeAreaCode ()
     { return this._telephoneNumber.areaCode;}
    get officeNumber ()
     {return this._telephoneNumber.number;}
}

Class TelephoneNumber {
    get areaCode () { return this._areaCode;}
    get number () { return this._number;}
}
```

```
Class Person {
    get officeAreaCode ()
{return this._officeAreaCode;}

    get officeNumber () {return
this._officeNumber;}

}
```


NTNU

Simplify logic within object - Speculative Generality

- When you have code that isn't actually needed today.
- Such code often exist to support future behavior, which may or may not be necessary in the future
- Remove unnecessary delegation, unused parameters, dead code

```
Public class Customer {
    private String name;
    private String address;

    Public Customer (String name, String add)
    { this.name = name;
      this.address = add;
     }

      String GetName () {
            return name;
        }
  }
```

Simplify logic within object – Temporary Field

- Class has a variable which is only used in some situation
- Trying to understand why a variable is there when it doesn’t seem to be used

```php
class User
{
    private $id;
    private $name;
    private $priviledges;
    private $content;
    private $contactDetails;

    public function __construct()
    {
        $this->contactDetails = new UserContactDetails();
    }
    function notify($message)
    {
        $messageBody = 'Dear ' . $this->name;
        $messageBody .= '( ' . $this->contactDetails->getStreetNumber() . ' ' .
            $this->contactDetails->getStreetName() . ').';
        $messageBody .= $message;
        $notificationService->sendSMS($this->contactDetails->getPhoneNubmer(), $messageBody);
    }
    function delete()
    {
        systemDelete($this->id);
    }

    function update() {
        systemUpdate($this->id, $this->priviledges, $this->content);
    }
```

$name and $contactDetails are only used in the notify() method.

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

Why not pass them as a method parameters

```
function notify($userName, $contactDetails, $message) {

    $messageBody = 'Dear ' . $userName;
    $messageBody .= '( ' . $contactDetails->getStreetNumber() . ' ' .
        $contactDetails->getStreetName() . ').';
    $messageBody .= $message;

    $notificationService->sendSMS($contactDetails->getPhoneNubmer(), $messageBody);
}
```

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

Simplify logic between objects

- **Message Chains**
- **Middle Man**

Employee->EmployeeConfig->Config

```php
Class Employee{
    public function getConfiguration(){
        $this->employeeConfig->getConfiguration();
    }
}

Class EmployeeConfig{
    public function getConfiguration(){
        $this->config->getConfiguration();
    }
}

Class Config{
    public function getConfiguration(){
        $this->loadConfiguration();
    }
}
```

Employee->Config

```php
Class Employee{
    public function getConfiguration(){
        $this->config->getConfiguration();
    }
}
```

Proper OO practice

- **Data Class**
  - Make a public field private and provide accessors

```
public String _name
```

```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

Proper OO practice (Cont')

- **Refused Bequest**
  - Only some of the inheritances are need, i.e., hierarchy is wrong
  - Inheritance makes no sense and the subclass really does have nothing in common with the superclass

https://refactoring.guru/smells/refused-bequest

Comments in code

- Good to have, can be simplified
- **Best if code is self-explanatory**
  - If you need a comment to explain what a block of code does, simply the code
  - If the method is already extracted but you still need a comment to explain what it does, rename the method
  - If you need to state some rules about the required state of the system, use Assertion

Python code refactoring

Python code smell* – code conventions

- Some are relevant to python code conventions** , e.g.,
  - Function names should comply with a naming convention
  - Class names should comply with a naming convention

- Tool support
  - Pylint (Checking structure of the code is compliant with PEP-8)

Pylint
Star your Python code!

* https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720

** https://www.python.org/dev/peps/pep-0008/

Pylint demo

https://www.youtube.com/watch?v=fFY5103p5-c

Python code smell*

- Remove duplications
  - Redundant pairs of parentheses should be removed
  - Using features of Python, e.g., Decorator**
- Reduce size
  - Functions, methods and lambdas should not have too many parameters
  - Lines should not be too long
- Speculative Generality, e.g.,
  - Nested blocks of code should not be left empty
  - Unused local variables should be removed

* https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720

**https://www.youtube.com/watch?v=n_Y-_7R2KsY

Python code smell* (cont’)

- Reduce complexity and simplify logics
  - Cognitive Complexity of functions should not be too high
  - Functions should not be too complex
  - Control flow statements "if", "for", "while", "try" and "with" should not be nested too deeply
  - Functions should not contain too many return statements

* https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720

Python code smell – tool support

E.g., PyCharm

Inspection and code review

What is inspection?

- Visual examination of software product

- Identify software anomalies
  - Errors
  - Code smells
  - Deviations from specifications
  - Deviations from standards
    - E.g., Java code conventions

    www.oracle.com/technetwork/java/codeconventions-150003.pdf

Why do we need inspection?

- Many software artifacts cannot be verified by running tests, e.g.,
  - Requirement specification
  - Design
  - Pseudocode
  - User manual

- Inspection reduces defect rates
- Inspection complements testing
- Inspection identifies code smells
- Inspection provides additional benefits

Inspection finds types of defects different from testing

Number of different types of defects detected by testing vs. inspection [1]

Some defect types	Testing	Inspection
Uninitialized variables	1	4
Illegal behavior, e.g., division by zero	49	20
Incorrectly formulated branch conditions	2	13
Missing branches, including both conditionals and their embedded statements	4	10

Inspection reduces defect rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

Testing vs. Inspection

	Testing	Inspection
Pros	• Can, at least partly, be automated • Can consistently repeat several actions • Fast and high volume	• Can be used on all types of documents, not just code • Can see the complete picture, not only a spot check • Can uses all information in the team • Can be innovative and inductive
Cons	• Is only a spot check • Can only be used for code • May need several stubs and drivers	• Is difficult to use on complex, dynamic situations • Unreliable (people can get tired) • Slow and low volume

Inspection provides additional benefits [2]

- Code and change understanding
- Knowledge transfer
- Increased team awareness
- Creation of alternative solutions to problems

Code review at Google [3]

- "All code that gets submitted needs to be reviewed by at least one other person, and either the code writer or the reviewer needs to have readability in that language. Most people use Mondrian to do code reviews, and obviously, we spend a good chunk of our time reviewing code."

  --Amanda Camp, Software Engineer, Google

Code reviews in OSS projects [4]

- Asynchronous and distributed review
  - Lightweight with tool support
  - Not limited to the existence of defects
  - Encourage discussing the best solution
- Frequent review
  - Most peer review happens within hours of completing a change
- Incremental review
  - Reviews should be of changes that are small, independent, and complete

The potential system benefit of accepting code that hasn't been discussed by a group of experts doesn't outweigh the risks. (Linus Torvalds)

Tools to support inspection

- Google’s Mondrian
- Facebook’s Phabricator
- GitHub
- ...

GitHub review tool

Software reading techniques

- Unstructured: Ad hoc reading
- Semi-structured: checklist-based reading
- Structured or systematic reading
  - Defect-based reading
  - Perspective-based reading

A sample (partial) checklist for requirement review [4]

1. Requirements specifications shall be testable.
2. Requirements specifications shall not conflict with other requirements specifications.
3. Conditional requirements specifications shall cover all cases.
4. Numerical values in requirements specifications shall include physical units if applicable.

A sample (partial) checklist for code review [4]

1. Have resources (e.g., memory, file descriptor, database connection) properly been freed?
2. Are shared variables protected/thread-safe?
3. Is logging implemented?
4. Are comments updated and consistent with the code?
5. Is data unnecessarily copied, saved, or reloaded?
6. Is the number of cores checked before spawning threads?

Checklist-based reading with active guidance

- Use a tailored checklist

- Guide reader how to use the checklist
  - Where to find?
  - To find what?
  - How to detect?

An example of a checklist with active guidance [4]

Where What How

Section	Feature	Checklist item
For each class	Inheritance	Q1. Is all inheritance required by the design implemented in the class?
		Q2. Is the inheritance appropriate?
	Constructor	Q3. Are all instance variables initialized with meaningful values?
		Q4. If a call to base class is required in the constructor, is it present?
For each method	Data referencing	Q5. Are all parameters used within a method?
		Q6. Are the correct class constants used?
		Q7. Are indices of data structures operating within the correct boundaries?

Best practice of checklists

- Should be periodically revised
- Should be concise and fit on one page
- Should not be too general
- Should not be used for conventions which are better checked or enforced with software tools

Unsystematic vs. Systematic reading

Defect-based reading

- Each reader is given specific responsibility to discover a particular class of defects

- Specific responsibility from defect taxonomy
  - Wrong or missing assignment
  - Wrong or missing data validation
  - Wrong timing or sequencing
  - Interface problems
  - Etc.

Perspective-based reading

- Each reader reviews software artifact from a particular perspective
- Then combine results from different perspectives
- Perspective examples
  - Novice user and expert user (GUI)
  - User, designer, tester (Requirements)

Template for perspective-based reading

- Introduction
  - Explain stakeholder’s interests
- Instruction
  - How to read and extract information
- Questions
  - For reviewers to answer by following the instruction

Novice use usability reading guideline [5]

- Introduction
  - You review the usability of the software to make PPT from the perspective of users who are new to use software to make slides.
- Instruction 1
  - Scan through menu, sub-menu, help info. in GUI
- Questions
  - Are there instructions or online help?
  - Are items in menu, sub-menu unambiguous in meaning?

Expert use usability reading guideline

- Introduction
  - You review the usability of the software to make PPT from the perspective of an expert who wants to use the software for making slides quickly.
- Instruction
  - Execute actions through finding and using shortcuts.
- Questions related to the instruction
  - Is it easy to find the frequently used shortcuts?
  - Is there any online help to guide the usage of shortcuts?

Benefits of perspective-based reading

- Focused
  - Particular coverage of part of the document
- Detailed
  - Instruction and questions
- Adaptable
  - To particular software artifacts
- Tailorable
  - To organizational and project setting

Effective factors affecting inspection [6]

- Individual expertise and training
  - Most important factor
- Amount of materials
  - Not to attempt to inspect too many materials in one cycle
  - E.g., less than 125 lines of code per hour
- Team size
  - 4 inspectors may not find significantly more defects than 2 inspectors
  - 4 or 2 usually find more defects than 1
- Process and tool support
  - With good tool support, inspection meeting may not be necessary

Summary

- We have studied
  - Why and how to do code refactoring
  - Why and how to perform code review

References

- [1] S.S. So et al. An empirical Evaluation of Six Methods to Detect Faults in Software, Software Testing, Verification, and Reliability, 12, 3, 2002, pp. 155-171.
- [2] A. Bacchelli and C. Bird: Expectations, outcomes, and challenges of modern code review. Proc. ICSE 2013.
- [3] CSE 403 course at University of Washington, available at https://courses.cs.washington.edu/courses/cse403/11sp/lectures/lecture15-reviews.pdf
- [4] Y. M. Zhu: Software reading techniques. Apress, 2016.
- [5] Z. Zhang, V. Basili, and B. Shneiderman: Perspective-based usability inspection: An empirical validation of efficacy. Journal of empirical software engineering, 4, 1999, pp. 43-69.
- [6] S. Kollanus and J. Koskinen: Survey of software inspection research, The Open Software Engineering Journal, 3, 2009, pp. 15-34.
- [7] D. Rombach et al. Impact of research on practice in the field of inspections, reviews, and walkthroughs, ACM SIGSOFT SE notes, 33, 6, 2008, pp. 26-35.

Software estimation – cost and quality

Mar 2022

Anh Nguyen Duc

Agenda

- The role of project estimation
- Traditional approaches for effort estimation
- Machine Learning in effort prediction
- Defect prediction models
- Within vs. Between project prediction
- Just-in-time defect prediction

What is a Project Estimate?

A declaration about needed

- cost and
- time
- quality
- human skill
- for delivering the project scope

How productive is this team/ developer in this project?

- What else?

Importance of project estimation

- ✓ Ressource allocation decisions
- ✓ Basis for the decision to start (or not to start)a project
- ✓ Foundation for project planning and set-up (business case)
- ✓ Foundation for project controlling
- ✓ If project time is a given,number of ressources can be determined
- ✓ Owner of an estimate is an indication about who is taking the project risk
- ✓ Decision and ressource allocation implications => Estimates are often part of political games
- ✓ Estimating is a core task of project management

Challenges for estimation

- ✓ Incomplete knowledge about:
  - ❖ *Project scope and changes*
  - ❖ *Prospective resources and staffing*
  - ❖ *Technical and organizational environment*
  - ❖ *Infrastructure*
  - ❖ *Feasibility of functional requirements*
- ✓ *Comparability of projects in case of new or changing technologies, staff, methodologies*
- ✓ *Learning curve problem*
- ✓ *Different expectations towards project manager*

Challenges for estimation

- ✓ Estimation is too low
  - ✓ *Scope and tasks(WBS) incomplete/unknown*
- ✓ Estimation is too high
  - ✓ *Political / human reasons*
  - ✓ *Learning curve*
- ✓ New technologies can make new parameters necessary

Challenges for estimation

Estimation uncertainty

Estimating Effort

Cost Estimation

- ✓ Effort costs (the dominant factor in most projects)
  - ✓ – salaries of engineers involved in the project
  - ✓ – costs of building, heating, lighting
  - ✓ – costs of networking and communications
  - ✓ – costs of shared facilities (e.g library, staff restaurant, etc.)
  - ✓ – costs of pensions, health insurance, etc.
- ✓ Other costs
  - ✓ – *Hardware and software costs*
  - ✓ – *Travel and training costs*

Cost Estimation

- ✓ Staff categories (based on experience, qualification and skills),for example:
  - ✓ teamlead, junior business analyst, senior business analyst, junior programmer, senior programmer, subject matter expert
- ✓ Cost rate: Cost per person per day
  - ✓ 2 alternatives for cost rate:
    - ✓ *Single cost ratef or all types(no differentiation necessary)*
    - ✓ *Assign different cost rates to different categories*
- ✓ Personnel cost = person days x cost rate

Estimating effort- Basic principles

Select an estimation model (or build one first)

Evaluate known information: project scope,resources, software process (for example documentation requirements),system components

Feed this information as parametric input data into the model
Model converts the input into an estimate about the effort

Basic of an Estimation model

Examples:

Data Input	Estimate
Size & Project Data	Effort & Schedule
System Model	Performance
Software Process	Cycle Time.

How to build an estimation model?

Calibrating the model

Top-Down and Bottom-Up Estimation

Two common approaches for estimations

Top-Down Approach

- Estimate effort for the whole project
- Breakdown to different project phases and work products

Bottom-Up Approach

- Start with effort estimates for tasks on the lowest possible level
- Aggregate the estimates until top activities are reached

Top-Down and Bottom-Up Estimation

Top-Down Approach

- Normally used in the planning phase when little information is available how to solve the problem
- Based on experiences from similar projects
- Not appropriate for project controlling (too high-level)
- Risk add-ons usual as result tends to be too low

Bottom-Up Approach

- Normally used after activities are broken down to task level and estimates for the tasks are available
- Result can be used for project controlling (detailed level)
- Smaller risk add-ons (tends to be too high)
- Often a mixed approach with recurring estimation cycles is used.

Estimation Techniques

- ✓ Expert judgement
- ✓ Estimation by analogy
- ✓ Parkinson's Law
- ✓ Pricing to win
- ✓ Lines of code
- ✓ Function point analysis
- ✓ Algorithmic cost modelling
- ✓ COCOMO

Expert judgement

= Guess from experienced people

- Mostly used top-down for the whole project, but also for some parts of a bottom-up approach
- Relatively cheap estimation method. Can be accurate if experts have direct experience of similar systems
- No better than the participants
- Result justification difficult
- Very inaccurate if there are no experts!

Estimation by analogy

(do you work on the same project before?)

- The cost of a project is computed by comparing the project to a similar project in the same application domain
- Accurate if project data available
- Impossible if no comparable project has been tackled. Needs systematically maintained cost database

Parkinson's Law

"work expands to fill the time allotted for its completion"

- The project costs whatever resources (people, money, time) are available
- No overspend
- System is usually unfinished

Pricing to win

The project costs whatever the customer has to spend on it

- You get the contract
- The probability that the customer gets the system he or she wants is small. Costs do not accurately reflect the work required

Line of Code (LOC)

Traditional way for estimating application size (FORTRAN and assembler -> line-oriented languages)

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

Line of Code (LOC)

Traditional way for estimating application size (FORTRAN and assembler -> line-oriented languages)

Advantage: Easy to do

Disadvantages:

- No standard definition for Line of Code (logical versus physical)
- Of no help given a written project scope or functional design
- You get what you measure: If the number of lines of code is the primary measure of productivity, programmers ignore opportunities of reuse
- Multi-language environments: Hard to compare mixed language projects with single language projects

The use of lines of code metrics for productivity should be regarded as professional malpractice (Caspers Jones).

Function Point Estimation

Based on FP metric for the size of a product

Based on the number of inputs (Inp), outputs (Out), inquiries (Inq), master files (Maf), interfaces (Inf)

Step 1: Classify each component of the product (Inp, Out, Inq, Maf, Inf) as simple, average average, or complex

- Assign the appropriate number of function points
- The sum of function pointers for each component gives UFP (unadjusted function points)

Function Point Estimation

- Step 2: Compute the technical complexity factor (TCF)
  - Assign a value from 0 (“not present”) to 5 (“strong influence throughout”) to each of 14 factors such as transaction rates, portability
  - Add the 14 numbers: This gives the total degree of influence (DI)
    - *$TCF = 0.65 + 0.01 \times DI$*
    - *The technical complexity factor (TCF) lies between 0.65 and 1.35*
- Step 3 The number of function function points (FP) is:
    - *$FP = UFP \times TCF$*

Function Point Estimation (example)

Function Point Estimation (example)

Mapping Functions to Transaction Types

Add Customer Change Customer Delete Customer Receive payment Deposit Item Retrieve Item Add Place Change Place Data Delete Place	External Inputs
Print Customer item list Print Bill Print Item List	External Outputs
Query Customer Query Customer's items Query Places Query Stored Items	External Inquiries

Function Point Estimation (example)

Calculate the Unadjusted Function Points

Function Type	Number		Weight Factors simple	average	complex		
External Input (EI)		x	3	4	6	=	
External Output (EO)		x	4	5	7	=	
External Queries (EQ)		x	3	4	6	=	
Internal Datasets (ILF)		x	7	10	15	=	
Interfaces (EIF)		x	5	7	10	=	
					Unadjusted Function Points (UFP)	=	

The unadjusted function points are adjusted with general system complexity (GSC) factors

Function Point Estimation (example)

After the GSC factors are determined, compute the

Value Added Factor (VAF): $\mathbf{VAF} = 0.65 + 0.01 * \sum_{i=1}^{14} GSC_i$ $\quad GSC_i = 0,1,\ldots,5$

Function Points (FP) = Unadjusted Function Points (UFP)* VAF

PF (Performance factor) = Number of function points that can be completed per day

Effort = FP / PF

The unadjusted function points are adjusted with general system complexity (GSC) factors

Function Point Estimation (challenges)

- Complete description of functions necessary
  - Often not the case in early project stages -> especially in iterative software processes
- Internal functions (algorithms) rather underestimated, as model is based on user oriented requirements and functions
- Only complexity of specification is estimated
  - Implementation is often more relevant for estimation
- High uncertainty in calculating function points:
  - Weight factors are usually deducted from past experiences (environment, used technology and tools may be out-of-date in the current project)
- Not suitable for project controlling.

Algorithmic cost modelling

- Most models before 2005 are basing on some form of parametric formulas
- After 2005, more advanced prediction and classification approaches

- 65-85: the search for right parametric forms
- 85-95: advances in size and complexity metrics
- 95-05: proliferation of software development styles
- 05-15: advances in prediction and classification models

Belady and Lehman (1972)

$$\text{Effort} = p + K^{c-d}$$

- *Effort*: total maintenance effort
- *p*: productive effort, including analysis, design, code testing
- *d*: degree of maintenance team familiarity with the software
- *c*: complexity caused by lack of structured design and document
- *K*: empirical constant, depends on the environment

The COCOMO model

- Effort (person month) is a function of size (LOC)
- Exists in three stages
  - Basic - Gives a 'ball-park' estimate based on product attributes Intermediate
  - Modifies basic estimate using project and process attributes
  - Advanced - Estimates project phases and parts separately

Effort = b *x* Size ^{c}

- Organic: small teams develops software in known environment (*b*=2.4, *c*=1.05)
- Embedded: inflexible and constrained environment (*b*=3.6, *c*=1.20)
- Semidetached: varying levels of team experience working on larger projects (*b*=3.0, *c*=1.12)

Effort = b x Size ^{c} x EAF

- Intermediate model
- b, c: calibrated factors
- EAF: effort adjustment factor

COCOMO suite of models (1981-2007)

Advantages

- Repeatable estimations
- Easy to modify input
- Easy to customize and refine formula

Disadvantages

- Subjective inputs
- Unable to deal with exceptional conditions
- Mainly designed for waterfall
- Needs historical data for calibration

Figure from Boehm et al. 2008

Machine Learning-based Estimating Effort

Types of ML techniques

Case-Based Reasoning (CBR)

Decision Trees (DT)

Bayesian Networks (BN)

Support Vector Regression (SVR)

Genetic Algorithms (GA)

Genetic Programming (GP)

Association Rules (AR)

Artificial Neural Networks (ANN)

J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, “Systematic literature review of machine learning based software development effort estimation models,” *Information and Software Technology*, vol. 54, no. 1, pp. 41–59, Jan. 2012

Case-Based Reasoning (CBR)

- Finding a similar past case, and reusing it in the new problem situation
- CBR is the most widely used method in the software estimation practice

Similarity measure:

- Manhattan distance
- Euclidean distance
- Minkowski distance
- grey relational coefficient
- Gaussian distance
- Mahalanobis distance

ID	Feature	Full name or explanation
1	TeamExp	Team experience in years
2	ManagerExp	Project manager's experience in years
3	YearEnd	Year of completion
4	Length	The length of project
5	Transactions	Number of transaction processed
6	Entities	Number of entities
7	PointsAdjust	Adjusted function points
8	PointsNonAjust	Unadjusted function points
9	Envergure	Complex measure derived from other factors
10	Effort	Measured in person-hours

Jorgensen M, Shepperd M (2007) A systematic review of software development cost estimation studies. IEEE Trans Softw Eng 33:33–53
E. Rashid, S. Patnaik, and V. Bhattacherjee, "Software Quality Estimation using Machine Learning: Case-based Reasoning Technique," *International Journal of Computer Applications*, vol. 58, no. 14, pp. 43–48, Nov. 2012.

Decision Trees (DT) / Classification and regression trees (CART)

- relatively easy to understand and are also very effective
- DT splits the original dataset to sub datasets which are more homogeneous.
- Recursive Binary Splitting: *all the features are considered and different split points are tried*

When to stop splitting?

Jorgensen M, Shepperd M (2007) A systematic review of software development cost estimation studies. IEEE Trans Softw Eng 33:33–53
E. Rashid, S. Patnaik, and V. Bhattacherjee, "Software Quality Estimation using Machine Learning: Case-based Reasoning Technique," *International Journal of Computer Applications*, vol. 58, no. 14, pp. 43–48, Nov. 2012.

Support Vector Machine (SVM)

- Builds a model that assigns new case to one category or the other (binary classifier)
- Map training cases to points in space to maximise the width of the gap between the two categories.
- A new case is then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

H_1 does not separate the classes.
H_2 does, but only with a small margin.
H_3 separates them with the maximal margin.

Artificial Neural Network (ANN)

- Input layers: size of the software in function points or Lines Of Code (LOC), and other effort drivers such as complexity of the software, database size, experience etc.
- Each node has its own linear regression model, compose of input and output variables

K. K. Gebretsadik and W. T. Sewunetie, “Designing Machine Learning Method for Software Project Effort Prediction,” *Computer Science and Engineering*, vol. 9, no. 1, pp. 6–11, 2019.

Deep Learning

- Model for Agile projects, predict efforts for the next Iteration
- Adopt Long Short Term Memory / Recurrent Neural Network
- Input: SP: story points, TD length: the number of words in the title and description of an issue, LOC: line of code

M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies, "A Deep Learning Model for Estimating Story Points," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 637–656, Jul. 2019

Defect Prediction

Importance of defect prediction

- Software defect: flaw or imperfection in software work or software process
- A defect is also referred as a fault or a bug
- Every software project has bugs, which we hope to reduce.
- Defect prediction is essential in the field of software quality and software reliability
- Focus on predicting those defects that affect project and product performance

Urgent Severe	Urgent Not Severe
Not Urgent Severe	Not Urgent Not Severe

Pareto rule (20-80 rule)

Most defects (80%) exist in few modules (20%)

Defect data is usually imbalanced

Early days prediction models - COQUALMO

- predicting number of residual defects/KSLOC (Thousands of Source Lines of Code) or defects/FP (Function Point) in a software product
- Base on COCOMO II model

Algorithmic approaches - Basic principles

Software defect prediction – mainly base on historical data

When to use: (1) use for planning (2) in-process monitoring of defect discovery numbers

What to predict?

- Number of defect
- Defect density
- Defect proneness

Software entity	Software attribute	Software metrics	References	# of Refs
Product	Size	LOC or number of use cases	S2, S37, S15, S21, S49, S32, S47, S35, S27, S13, S42, S34, S10, S19, S41, S16, S36,	17
		Size of artifact	S1, S13	2
		Code metrics: size metrics from NASA projects (Halstead size metrics)	S50, S33, S21, S28, S11, S12, S3, S9, S4, S25	10
		Requirement metrics: action, conditional, continuance, imperative, incomplete, option, risk level, source, weak phrase	S33, S28, S11, S12, S3, S9, S4, S20, S25	9
	Structure	McCabe Metrics (Complexity etc.)	S52, S21, S17, S47, S42, S44, S30, S39, S22,	12
		Halstead Metrics (total number of operators, operands etc.)	S7, S48, S19	
		Object-oriented Metrics (Complexity, Length, Coupling, Cohesion, Modularity, Reuse)	S51, S50, S29, S18, S14, S6, S16, S36, S19	9
		Design metrics from UML (e.g. Chidamber and Kemerer (1994))		
		Data flow complexity (DC), cyclomatic complexity (CC)	S8, S24, S43	3
		Requirements complexity (RC), Complexity of new functionality	S2, S37, S32, S35, S34, S48	6
		Program dependencies	S38	1
		Design metrics: edge count, node count, branch count, decision count, multiple condition count and condition count, densities, complexities	S20	1
		Architectural design metrics to quantify SDL (Specification and Description Language) blocks	S45	1
		Design, review or development effort measured in person hour	S1, S5, S23, S37, S31, S40, S35, S13, S43	9
		Creation effort, review effort	S26, S46	2
		Design review effectiveness (DRE)	S30, S7	2
		Review, inspection and walkthrough (RIW)	S2, S8, S32, S24, S34, S30, S7, S10, S48	9
		Total months of the project duration	S15, S31, S41	3
		Requirements stability (RS), requirement change request (RCR)	S2, S37, S17, S8, S32, S35, S27, S24, S34, S43, S30, S7, S10, S48	14
		Capability Maturity Model Integration (CMMI) Level	S2, S37, S8, S32, S40, S35, S24, S34, S30, S7,	10
		Number of detected defects from review	S1, S37, S15, S35, S13	5
		Requirement fault density (RFD), design defect density (DDD), fault days number (FDN), code defect density (CDD),	S15, S17, S8, S27, S24, S10, S48,	7
		Analysis, design, review quality	S37, S40, S35, S43	4
		Quality of documented test cases (QDT)	S35, S7, S41	3
		Defined process followed (DPF)	S2, S32, S34, S35, S37	5
		Number of stakeholders/members	S15, S49, S41	3
		Development language	S37, S15	2
		Configuration management	S37, S35, S41	3
		Project planning	S37, S35	2
		Scale of distributed communication	S37, S35	2
		Vendor management	S37, S35	2
		DBMS type, development solution, industry area	S15, S41	2
		Techno complexity	S26, S49, S46	3
		Urgency	S46	1
		Novelty to developer	S49	1
	Human characteristics	Staff experience	S2, S37, S32, S40, S35, S27, S34, S43, S7, S10	10
		Staff motivation	S37, S35	2
		Programmer capability	S37, S35, S30, S7	4
		Staff training quality	S37, S35	2
		Internal communication/interaction	S37, S35	2
		Productivity	S15	1
		Practitioners level	S26, S46	2
		Stakeholder involvement	S2, S32, S34	3
		People dependence	S41	1

R. Özakıncı and A. Tarhan, “Early software defect prediction: A systematic map and review,” *Journal of Systems and Software*, vol. 144, pp. 216–239, Oct. 2018, doi: 10.1016/j.jss.2018.06.025.

With-in vs. Cross-project prediction

due to the diversity in development processes, a defect prediction model is often not transferable and requires to be rebuilt when the target project changes

Critiques on defect prediction

- the unknown relationship between defects and failures
- problems with the “multivariate” statistical approach
- problems of using size and complexity metrics as sole “predictors” of defects
- problems in statistical methodology and data quality
- false claims about software decomposition and the “Goldilock’s Conjecture”

A typical ML model for defect prediction

History of ML/AI approaches in defect prediction

S. Omri and C. Sinz, “Deep Learning for Software Defect Prediction: A Survey,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, New York, NY, USA: Association for Computing Machinery, 2020, pp. 209–214. Accessed: Mar. 19, 2022. [Online]

Just-in-time defect prediction

Fixing Commit Message

Revert "Make VisibleRefFilter.Filter reuse the refs passed from JGit."

This reverts commit b032a529f83892dfbdfb375c47a90d89756dd8ab. This commit introduced an issue where tags were not replicated under certain circumstances.

Bug: Issue 2500
Bug: Issue 1748
Change-Id: I9c902b99c7f656c7002cf3eab9e525f22a22fb85

Defective Commit: b032a529f83892dfbdfb375c47a90d89756dd8ab

2 gerrit-server/src/main/java/com/google/gerrit/server/git/VisibleRefFilter.java

```
103 103   if (!deferredTags.isEmpty() && (!result.isEmpty() || filterTagsSeperately)) {
104 104     TagMatcher tags = tagCache.get(projectName).matcher(
105 105         tagCache,
106 106         db,
107     -       filterTagsSeperately ? filter(db.getAllRefs()).values() : result.values());
    107 +       filterTagsSeperately ? filter(refs).values() : result.values());
108 108     for (Ref tag : deferredTags) {
109 109       if (tags.isReachable(tag)) {
110 110         result.put(tag.getName(), tag);
111 111       }
112 112     }
113 113   }
```

Fixing Commit: 6db280663f836096c30a9626e7170f4a36d8cc1f

2 gerrit-server/src/main/java/com/google/gerrit/server/git/VisibleRefFilter.java

```
112 112   if (!deferredTags.isEmpty() && (!result.isEmpty() || filterTagsSeperately)) {
113 113     TagMatcher tags = tagCache.get(projectName).matcher(
114 114         tagCache,
115 115         db,
116     -       filterTagsSeperately ? filter(refs).values() : result.values());
    116 +       filterTagsSeperately ? filter(db.getAllRefs()).values() : result.values());
117 117     for (Ref tag : deferredTags) {
118 118       if (tags.isReachable(tag)) {
119 119         result.put(tag.getName(), tag);
120 120       }
121 121     }
122 122   }
```

Technical Debt Management in Visma

Mili Orucevic
Chief Software Quality Engineer

VISMA

Agenda

- Technical Debt in General
- How are we working with Technical Debt in Visma?
- Dive into some examples
  - Assessment
  - Dashboards
  - Indexes
  - Example of large Technical Debt
- Research: Why Technical Debt Matter?

VISMA

Mili Orucevic

Chief Software Quality Engineer

https://www.linkedin.com/in/milio

VISMA

Lehman’s Law of Software Evolution

Law of Continuing Change

“A system must be continually adapted or it becomes progressively less satisfactory”

Law of Increasing Complexity

“As a system evolves, its complexity increases unless work is done to maintain or reduce it.”

VISMA

Technical Debt definition

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The **danger occurs when the debt is not repaid.** Every minute spent on **not-quite-right code** counts as **interest on that debt**. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object- oriented or otherwise"

1992 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)

Ward Cunningham

"In software-intensive systems, technical debt consists of design or implementation constructs that are **expedient in the short term** but that set up a technical context that **can make a future change more costly or impossible**. Technical debt is a contingent liability whose impact is limited to internal system qualities—primarily, but not only, maintainability and evolvability."

Kruchten, Nord, Ozkaya (p.5)

Managing Technical Debt

Software systems are prone to the **build up of cruft** - deficiencies in internal quality that **make it harder than it would ideally be to modify and extend** the system further. Technical Debt is a metaphor, coined by Ward Cunningham, that frames how to think about dealing with this cruft, thinking of it like a financial debt. The extra effort that it takes to add new features is the interest paid on the debt.

Blogpost, 21.05.2019:
https://martinfowler.com/bliki/TechnicalDebt.html

Martin Fowler

VISMA

Technical Debt Landscape

←← Visible →→

←← Mostly Invisible →→

←← Visible →→

Business

New Features

Additional Functionality

Architecture
Architecture smells
Pattern Violations
Structural Complexity

Code
Code Complexity
Code Smells
Coding Style Violations
Low Internal Quality

Production Infrastructure
Build, Test and Deploy Issues

Defects

Low External Quality

End Users

Evolvability

Maintainability

Product Roadmap

Internal Team Work

End Product

VISMA

Cause, effect and management

VISMA

Visualizing Technical Debt

Possible symptoms of Technical Debt

- Decline in number of releases
- Increase in delivery lead time
- Less and less time for developing new features

- Might treat symptoms instead of real issues

VISMA

Measuring Technical Debt

Static Code Analysis

- 18626 days or 51 years of technical debt, for 1106 projects
  - ~16 days effort per project
- First 10 days of March, 51 days of effort of technical debt was introduced
  - 7 min per project per day

Other examples outside Visma

- Project: 15 year old code based
- Technical debt accumulated: 4000 years

Should not be the only way to measure technical debt

VISMA

How are we creating awareness and work with Technical Debt in Visma?

VISMA

Visma - a SaaS company

14 000
Engaged employees

With more than **6 500** developers

Since 2014 more than **140 companies** have joined Visma

1 135 000
Customers

We are where you are
Strong local presence with more than **150** locations

Running through Visma's systems in September '21:

10,3 million payslips
23,2 million invoices

What did we want to solve?

Raise the **awareness** of technical debt in teams

Create **visibility** of technical debt

Common **process** for dealing with technical debt

Keep a technical debt **register** for teams

Ensure that technical debt is **planned** and **repaid** where reasonable

VISMA

Technical Debt Maturity Assessment

Technical Debt Self-assessmen

TDM activity / TDM levels	TD Repayment	TD Communication	TD Prioritization
Organized (level 3)	☑ Continuous refactorin with monthly assigne of the development t	Continuous communication about TD issues in meeting with all stakeholders.	☑ Prioritization is done based on specific calculated numbers. Recommended to use Severity as the calculated number
Semi organized (level 2)	☑ Refactoring during normal development previously identified issues.	☑ Product/Service Owner communicate development issues to business stakeholders. Communication must describe customer value.	☑ Prioritization is done with basic low/medium/high scaling that is based on hunches and rough estimations.
Unorganized (level 1)	Refactoring only when the problem is impossible to avoid or no refactoring at	TD is not part of development discussions and often done only in coffee table discussions.	Prioritization is not conducted and decisions are done based on a hunch.
Responsibility for activity	Development team, softw architect(s)	Development team, software architect(s), team manager, product/service owner	Software architect(s), team manager, product/service owner

Technical Debt Maturity Model

Prevention

Primary responsible roles: Architects, developers, and anyone ese writing code

Unorganized (level 1)	The team has not agreed on coding standards and does not perform code reviews
Semi organized (level 2)	☐ Following the team coding standards and performing code reviews is optional or ad hoc
Organized (level 3)	☐ The team follows coding standards and performs code review for all non-trivial changes

Identification

Primary responsible roles: Architects, developers and anyone else that witnesses technical debt

Unorganized (level 1)	The team does not identify any technical debt during the development process, except when it happens accidentally
Semi organized (level 2)	☐ The team identifies technical debt during implementation and code reviews
Organized (level 3)	☐ The team identifies technical debt during implementation and code reviews ☐ The team identifies technical debt using static code analysis tools (e.g. ReSharper, SonarQube, Coverty) ☐ The team identifies technical debt as part of operational processes like incident reviews and problem management

Documentation

Primary responsible roles: Architects and developers

Unorganized (level 1)	The team does not document any technical debt, or documents it in a separate backlog
Organized (level 3)	☐ Technical debt is documented in the same backlog as everything else

Analysis

Primary responsible roles: Architects and developers

Unorganized (level 1)	Technical debt is not analyzed
Semi organized (level 2)	☐ Technical debt is analyzed and summarized using the Priority field in JIRA
Organized (level 3)	☐ Technical debt is analyzed and a severity score is calculated and documented in JIRA

Monitoring

Primary responsible roles: Product/Service Owner and Architects

Unorganized (level 1)	Technical debt is not monitored
Semi organized (level 2)	☐ Technical debt is measured over time in JIRA by looking at the number of technical debt issues
Organized (level 3)	☐ Different types of technical debt metrics is measured over time by looking at the sum of all severity scores and metrics in SonarQube

Communication

Primary responsible roles: Product/Service Owner, architects and developers

Unorganized (level 1)	The team rarely talks about technical debt internally (in the team) or externally (with stakeholders)
Semi organized (level 2)	☐ The team frequently talks about technical debt internally (in the team)
Organized (level 3)	☐ The team continuously communicates technical debt risks to stakeholders, focusing on the business and/or customer value of paying down the technical debt with the hig score. (Using the Architect Maturity index and a list of technical debt issues in JIRA sorted by the severity score are two ways of doing this.)

Planning

Primary responsible roles: Product/Service Owner and architects

Unorganized (level 1)	The team does not have a well-defined process for deciding what technical debt should be paid down next
Semi organized (level 2)	☐ The team decides on what technical debt should be paid down next based on the value of the Priority field in JIRA
Organized (level 3)	☐ The team decides on what technical debt should be paid down next based on the severity scores

Repayment

Primary responsible roles: Product/Service Owner

Unorganized (level 1)	The team only pays down technical debt when it becomes a blocking issue
Semi organized (level 2)	☐ The team performs sufficient refactoring as part of the normal backlog prioritization and development process
Organized (level 3)	☐ Some of the team's capacity is always reserved for paying down technical debt (20% or more is the industry standard and highly recommended)
Superorganized (level 4)	☐ The team has instituted a maximum cap on technical debt, and will dedicate 100% of their capacity to paying down technical debt whenever the agreed cap is exceeded

Inspired by research done at the Lappeenranta University of Technology by Jesse Yli-Huumo.

VISMA

Technical Debt Maturity Assessment

Latest version

Category	Requirements
Prevention	☐ The team follows coding standards and performs code review for all non-trivial changes ☐ The intention of implementing any workaround or corner-cutting is discussed within the team before any code is written; team decides together if the workaround will be implemented or more time will be allocated to avoid it › More Info
Identification	☐ The team identifies technical debt during implementation and code reviews ☐ The team identifies technical debt using static/dynamic code analysis tools (e.g. ReSharper, SonarQube, Coverity, Snyk) and adds it in the backlog. › More info ☐ The team identifies technical debt as part of operational processes like incident reviews and problem management ☐ A process to identify outdated dependencies is implemented (e.g Dependabot, dotnet-outdated)
Documentation	☐ Technical debt is documented in the same backlog as everything else and the NFR label is used ☐ The team documents tech debt about to be introduced (intentional) technical debt › More Info ☐ Our top 3 TD issues are logged and documented in a way that are understandable for stakeholders and decision makers as well. Business impact, if relevant, is part of the documentation. › More Info
Analysis	☐ Technical debt is analyzed and a severity score is calculated and documented in JIRA › Why Severity Scores? ☐ The most important technical debt for our product is reflected in the top 3 TD issues logged in Jira
Monitoring	☐ All types of technical debt is visible in Jira by looking at the sum of severity scores. Here are suggestions for TD JIRA Dashboard / TD Dashboard › Click here to expand...
Communication	☐ The team continuously communicates technical debt risks to stakeholders and decision makers, focusing on the business and/or customer value of paying down the technical debt with the highest severity score. (Using the ArchTech Maturity Index and a list of technical debt issues in JIRA sorted by the severity score are two ways of doing this.) › Why is it important to communicate and be transparent about he technical debt?
Planning	☐ The team includes technical debt decisions and planning into their periodic processes. › Intention and More Info ☐ The team decides on what technical debt should be paid down next based on the severity scores
Repayment	☐ Some of the team's capacity is always reserved for paying down technical debt (20% or more is the industry standard and highly recommended)

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Example assessment

Link to assessment

Dashboard - team A

Filter Results: TDMA Example

T			P	Risk Impact	Risk Likelihood	Risk Severity ↓
			=	3	3	9
			=	3	1	3
			=	1	1	1
			=	1	1	1
			=	1	1	1
			=	1	1	1
			≪	1	1	1
			=	1	1	1
			=	1	1	1
			=	1	1	1
			=	1	1	1
			=	1	1	1

1–12 of 12

VISMA

Dashboard - team B

Filter Results: TDMA Example 2

T	Key	Summary	P	Risk Impact	Risk Likelihood	Risk Severity ↓
			=			
			=			
			^			
			=			
			=			
			=	4	4	16
			=	4	4	16
			=	3	5	15
			=	4	3	12
			^	3	4	12
			=	3	3	9
			=	3	3	9
			=	3	3	9
			^	3	3	9
			=	3	3	9
			=	3	3	9
			=	4	2	8
			=	4	2	8
			=	2	3	6
			=	3	2	6
			=	2	3	6
			=	2	3	6
			=	2	3	6
			=	2	3	6
			=	2	3	6
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			v	3	1	3
			v	3	1	3
			v	2	1	2

Examples from SonarQube

QUALITY GATE STATUS

Passed

All conditions passed

MEASURES

New Code
Since November 10, 2020

Overall Code

QUALITY GATE STATUS

Failed

1 conditions fai

On New Code

1 N

MEASURES

New Code

Overall Code

QUALITY GATE STATUS

Failed

1 conditions failed

On New Code

1 New Critical Issues is greater than 0

MEASURES

New Code
Since 1.0.217-SNAPSHOT
Started 6 hours ago

Overall Code

11 Bugs

4 Vulnerabilities

0 Security Hotspots

13d Debt

340 Code Smells

Maintainability A

77.7% Coverage on 17k Lines to cover

994 Unit Tests

2.0% Duplications on 41k Lines

53 Duplicated Blocks

Conditions

Conditions on New Code

Conditions on New Code apply to all branches and to Pull Requests.

Metric	Operator	Value
Coverage	is less than	60.0%
Duplicated Lines (%)	is greater than	3.0%

Conditions on Overall Code

Conditions on Overall Code apply to branches only.

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	2.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Rating	is worse than	A
Unit Test Errors	is greater than	0
Unit Test Failures	is greater than	0

VISMA

Examples from SonarQube

'System.Exception' should not be thrown by user code.
Code Smell

'System.Exception' should not be thrown by user code.
Code Smell

Refactor this code to not nest more than 3 control flow statements.
Code Smell

'System.Exception' should not be thrown by user code.
Code Smell

Remove the unused local variable 'request'.
Code Smell

Fix this implementation of 'IDisposable' to conform to the dispose pattern.
Code Smell +2

```
67
68          }
69
70          private async Task<A[redacted]Name)
71          {
72              [redacted]
73              var request = new GetSecretValueRequest {SecretId = secretName};
```

Remove the unused local variable 'request'. Why is this an issue? last year L73
Code Smell Major Open [redacted] 5min effort Comment unused

Unused local variables should be removed

Unused local variables should be removed

Code Smell Minor unused Available Since Aug 14, 2015 SonarQube (C#) Constant/issue: 5min

If a local variable is declared but not used, it is dead code and should be removed. Doing so will improve maintainability because developers will not wonder what the variable is used for.

Noncompliant Code Example

```
public int NumberOfMinutes(int hours)
{
  int seconds = 0;   // seconds is never used
  return hours * 60;
}
```

General exceptions should never be thrown | Utility classes should not

VISMA

“Not our fault”

Example of a bigger technical debt

- Unintentional
- External factors
- Time

“Move from AngularJS to new version of Angular”

~ 1583 hours = 66 days

VISMA

Automatically track deviations

Visma Index

VISMA

Visma Index

VISMA

Visma Index

A&T - Technical Debt Maturity Assessment (TDMA)

1 . Technical Debt Maturity Assessment not done or report older than 12 months	Confluence Expiration date 3 months ago	1	1	1		
2 . Technical Debt Maturity Assessment not done or report older than 18 months	Confluence Expiration date in 3 months	0	450	0		
3 . Unresolved critical issues from the Tech Debt Assessment older than 30 days	Jira	0	200	0		
4 . Unresolved high issues from the Tech Debt Assessment older than 90 days	Jira	1	100	100		
5 . Unresolved medium issues from the Tech Debt Assessment older than 90 days	Jira	0	20	0		
6 . Unresolved low issues from the Tech Debt Assessment older than 90 days	Jira	9	1	9		
				110		

VISMA

Why Technical Debt Matter?

Lessons learned from own research

VISMA

Technical debt matters!

Products with **more technical debt** are developed by teams with **lower customer-centricity and innovation** scores.

Presumably, it's difficult to have time and energy for being innovative and customer-centric when you have work with and around old technology and practices.

VISMA

Technical debt matters!

The reduced innovation and customer-centricity in products with high technical debt drives down customer satisfaction and revenue growth.

1. Product NPS by technical debt (quartiles)

2. CBV Growth Rate (%) by technical debt (quartiles)

1. In general, products with less technical debt are more successful

- Higher product NPS
- Higher product growth rate

2. Product lines in the bottom 25% of technical debt have higher growth rate than expected. These product lines tend to have lower CBV, and could be smaller products that have accumulated technical debt through rapid growth.

VISMA

Entrepreneurial

Responsible

Dedicated

Inclusive

Make progress happen

VISMA

TDT 4242 Advanced Software Engineering

- Spring 2022

About Jingyue Li (Bill)

- Jingyue Li (Bill)
  - Master (Computer science) in China
  - Architect: IBM China Ltd.
    - Banking solutions
  - PhD and Post-Doc (Software engineering) at IDI, NTNU
  - Principal researcher: DNV Research & Innovation
    - Software engineering
    - Dependability of critical software systems
  - Teaching experience:
    - Software architecture, IT projects, software security and data privacy, advanced software engineering, empirical software engineering
    - Started teaching this course from 2017
  - Jingyue.li@ntnu.no

About Daniela

- Co-Responsible teacher: Daniela Cruzes
- Master (Computer science) in Brazil
- Dr. Ing (Software engineering) in Brazil
- Post-Doc in USA and at IDI
- Tester: CPqD.
- Billing solutions for Telecom.
- Software Security Officer: FARA.
- Transport Intelligent Systems
- Senior Research Scientist: SINTEF
- Professor of Software Engineering:
- NTNU
- Lead Security Researcher : VISMA

- Email: dcruzes@ntnu.no

About Anh Nguyen-Duc

- Occupation & Experience
  - Assoc. Prof. at NTNU, Assoc. Prof. At USN
  - Entrepreneurs (Muml, VVN, etc), Startup mentors (VVNOR, Klikkit, Mitanbud, Comartek AS, etc)
- Research interest:
  - Software startups & software ecosystem
  - Secure software development, Static analysis
  - AI for SE, SE for AI, AI ethics
- Education
  - Master (Software Engineering) in Sweden & Germany
  - PhD and Post-Doc (Software engineering) at IDI, NTNU
- Teaching experience:
  - Customer driven project (TDT4290), Software engineering (TDT4140), IT project (IT2901), Web development & HCI (WEB1000), Software process (SYS1000), Project management (PRO1000, MIS405), Green IT (MSM4103), Product Process and People (MS4102)
- Contact: anhn@ntnu.no or angu@usn.no

Outline

- ➢ Knowledge to learn to be a good software engineer
- What advanced SE knowledge to be covered in this course?
- How is this course organized?

Goals of the software methodology – Ensure value creation

- Ensure economic gain
- Ensure personal gain
- Ensure organizational (business) gain
- Ensure societal gain

- Phases of value achievement
  - Plan for value achievement. Business case and value realization plan
  - Perform change
  - Implement change and realize the value

Categories of software based on functions

- Application software (web, mobile, IoT, control SW)
- System software (OS, DB, …)
- Computer programming tools (compiler, MATLAB, etc.)
- …

Project types

- In-house development:
  - The same company makes requirements, develops, and uses the system
- Custom development:
  - One company is customer, and
    - Another company analyzes requirements and develops system
    - Another company analyzes requirements, yet another develops
  - May include outsourcing to subcontractors
- Mass-market software
  - Project to buy or rent package (SaaS)
  - Project to develop an adaptation of package (e.g., ERP-system)
- Reuse of resources in digital or software ecosystems

Typical software development lifecycle models*

Waterfall model

V-model

* https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/

Typical software development lifecycle models (cont')*

Evolutionary Prototyping Model

Spiral Model

* https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/

Typical software development lifecycle models (cont”)*

Agile Model

DevOps

* https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/

Tasks within development and maintenance of software systems

Software Engineering Body of Knowledge (SWEBOK)* covered

- Software requirements
- Software design
- Software construction
- Software testing
- Software maintenance
- Software configuration management
- Software engineering management
- Software engineering process

*https://www.computer.org/web/swebok

Software Engineering Body of Knowledge (SWEBOK) covered (Cont')

- Software engineering models and methods
- Software quality
- Software engineering professional practice
- Software engineering economics
- Computing foundations
- Mathematical foundations
- Engineering foundations

Related courses

- TDT4140 Software Engineering
- TDT4250 Advanced Software Design
- TDT4290 Customer-driven project

Outline

- Knowledge to learn to be a good software engineer
- ➢ What advanced SE knowledge to be covered in this course?
- ➢ How is this course organized?

High-level course modules

- High-quality requirements
- High-quality and effective testing
- High-quality and maintainable code
- Advanced topics
  - DevOps
  - Cost and defect estimation
  - Software startup
  - Software eco-systems

Tentative lecture plan

High-quality requirements

Week	Date	Lecture content	Speaker
3	17.01.22	Requirement engineering 1	Anh Nguyen Duc
4	24.01.22	Requirement engineering 2	Anh Nguyen Duc

Challenges in Requirements Engineering

- Importance of getting requirements right:
  1/3 budget to correct errors originates from requirements

Phase in which fixed	Relative Cost
Requirements	1
Design	3 – 6
Coding	10
Development Testing	15 – 40
Acceptance Testing	30 – 70
Operations	40 – 1000

Challenges in Requirements Engineering (Cont')

- Why projects are canceled:

Factors	Percentage of Responses
Incomplete Requirements	13.1%
Lack of User Involvement	12.4%
Lack of Resources	10.6%
Unrealistic Expectations	9.9%
Lack of Executive Support	9.3%
Changing Requirements	8.7%
Lack of Planning	8.1%
Didn't need it any longer	7.5%
Lack of IT Management	4.3%
Technology Illiteracy	9.9%

Requirements Engineering process

- Use cases
- User stories
- Goal-oriented RE approaches

Characteristics of good software requirements specification*

- Complete
- Unambiguous
- Consistent
- Correct
- Verifiable
- Traceable
- Ranked for importance and/or stability
- Modifiable

* IEEE Std 830-1998

Requirements matched by testing – the V-model

IEC 61508 – Test requirements

	Technique/Measure *	Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Probabilistic testing	C.5.1	---	R	R	**R**
2	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	**HR**
3	Data recording and analysis	C.5.2	HR	HR	HR	**HR**
4	Functional and black box testing	B.5.1 B.5.2 Table B.3	HR	HR	HR	**HR**
5	Performance testing	Table B.6	R	R	HR	**HR**
6	Model based testing	C.5.27	R	R	HR	**HR**
7	Interface testing	C.5.3	R	R	HR	**HR**
8	Test management and automation tools	C.4.7	R	HR	HR	**HR**
9	Forward traceability between the software design specification and the module and integration test specifications	C.2.11	R	R	HR	**HR**
10	**Formal verification**	**C.5.12**	**---**	**---**	**R**	**R**

Tentative lecture plan

High-quality and effective testing

Week	Date	Lecture content	Speaker
6	07.02.22	Control flow and data flow testing	Jingyue Li
7	14.02.22	Functional testing with Black Box	Jingyue Li
8	21.02.22	System and acceptance testing and Regression testing	Jingyue Li
9	28.02.22	Testing in Practice	Daniela Cruzes

Tentative lecture plan

High-quality and maintainable code

Week	Date	Lecture content	Speaker
10	07.03.22	Code review and code refactoring	Jingyue Li
11	14.03.22	Technical Debt process in Industry	Guest lecture from Visma

Code review, analysis, and refactoring

Refactoring Code

- Making code better, may not be faster
- Better structured, better built
- More readable, more understandable
- Easy to work with
- Easy to add new features
- Easy to spot and fix bugs
- Keeping code in control
- Improving existing code

Tentative lecture plan

Advanced topics

Week	Date	Lecture content	Speaker
5	31.01.22	DevOps and Git Lab	Nora Thomas from Vipps.no
12	21.03.22	Cost and defect estimation	Anh Nguyen Duc
13	28.03.22	Software startup Software eco-systems	Anh Nguyen Duc

DevOps

- DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.
- Integrated development and operations
- Supporting quick releases

Cost and defect estimation - Cost

- Why?
  - 55% total number of project runs out of budgets
  - Almost every software project experiences some sort of project delays
  - Estimation is needed for better planning and preventive actions
- Purpose of software estimation
- How
  - Expert judgement: Delphi method, focus group, planning poker, etc
  - Regression-based method: COCOMO models, Putnam model, and function points
  - Simulation-based
  - Machine learning based approach: Bayesian Network, Deep Learning, etc
- Where to apply:
  - Waterfall vs. Agile
  - Within-company vs. cross-company
  - Private vs open source dataset

Cost and defect estimation - Defect

- Defects vs. fault, error, breach, failure
- Software defect can be very costly!
- Defect prediction is a cost-effective to deal with defect
  - Informed static analysis
  - Bug triaging
  - Faulty vs. non-faulty classification
- Approaches
- Metrics:
  - Project-level metrics: project type, budget, duration
  - Team level metrics: engineer's competence
  - Code level metrics: LOC, CK metric suite, code smell, number of functions, number of interfaces, number of class, code complexity, etc
  - Product level metrics: number of component, cohesion, etc
  - Collaboration level metrics: people-to-people interaction, people-to-file interaction
- Advanced defect prediction methods
  - Regression-based approaches
  - Support Vector Machine
  - Machine Learning (supervised/ unsupervised)
  - Decision Trees
  - Naïve Bayes
  - Deep Learning

Software startup

- Airbnb, Uber, Spotify, Skype, Kahoot, AppSumo, Vipps, Mitandbud, etc are used to be startups
- Startup companies are unique:
  - Little or no operating history
  - Limited resources
  - Multiple influences
  - Dynamic technologies and markets
- A temporary organizational state with special way of working, engineering practices and activities can be done different than in established contexts

Software startup

- Understanding failure in startups
- Reasoning about software startup engineering
  - Problem-solution fit and Product-market fit
  - Startup way of working
  - Technical debt
  - Influences of multiple stakeholders
  - Competence and financial situations
- Components of software startups
  - Organization
  - Competence
  - Product
  - Way of working
  - Startup ecosystem

Software startup

- Processes, framework, practices for software startup engineering
  - Lean approach to developing product and businesses
  - Minimum Viable Product
  - Requirement Engineering
  - Minimum Viable User Experience
  - Design thinking
  - Customer journey
  - Growth hacking
- Continuous experimentation in startups
  - Goal-driven hypotheses
  - Build-measure-learn loops
  - Hypothesis driven engineering

Software ecosystem

- "*a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them*" Messerschmitt and Szyperski (2003)
- Key elements:
  - A central hub: an owner of the platform
  - A platform: Usually technological platform
  - Niche players: who generate value from the platform
- Example: app stores, open source projects, etc

Software ecosystem

- Scope of ecosystem

A. A hub between suppliers and customers

B. The hierachy of hubs

C. Dynamic relationships among hubs, suppliers and customers

Software ecosystem

- Scope of ecosystem
- Healthiness of ecosystem
- Collaboration vs. Competition
- Sustainability

Curriculum

- Journal and conference articles (will be uploaded to blackboard)
- Some chapters from books (accessible in NTNU library)

(1st/2nd edition)

Teaching goals

- Apply goal-oriented requirement engineering method and requirement boilerplate to identify and model requirements;
- Identify and correct typical requirements quality issues;
- Apply different testing and code review, analysis, and refactoring strategies;
- Explain the industrial state of the practice methods of verifying and validating high-assurance software-intensive system;

Teaching goals (Cont')

- Understand the state of the practice agile methods, such as DevOps, and apply corresponding tools;
- Explain the issues and solutions of distributed and collaborative software development and maintenance;
- Explain key issues and solutions to managing and operating large and complex software-intensive systems.

Evaluation and grading

- Exercises: 50%
  - Exercise 1: 10%
  - Exercise 2: 25%
  - Exercise 3: 15%

A: 89–100 points
B: 77–88 points
C: 65–76 points
D: 53–64 points
E: 41–52 points
F: 0–40 points

- Exam (3rd June): 50%
- You have to pass both!
  - You have to hand in **all** the exercises **and** get a passing grade to be allowed to take the exam
  - If you fail the exam, you will fail the course

Exercises	Exam	Final grade
Fail	Fail	Fail
Fail	Pass	Fail
Pass	Fail	Fail
Pass	Pass	Combine exercise and exam grades

Exercise 1

- The purpose is to let students practice
  - Software requirement elicitation methods
  - Evaluating software requirements quality

- You need to
  - Based on an existing medium-sized web application, collect and understand new requirements you get from a peer group
  - Document the new requirements in high quality

Exercise 2

- The purpose is to let students practice
  - DevOps tools
  - Extending existing code
  - Software testing methods
- We will ask you extending the existing system and test the existing code
- You need to
  - Apply DevOps tools
  - Develop new code
  - Run tests manually and write test scripts

Exercise 3

- The purpose is to let students practice
  - Code review
  - Code analysis
  - Code refactoring

- You need to
  - Do manual code review of the existing code
  - Refactor the code based on your review and analysis results

Exercise tasks and schedule

The exercise introduction lectures are digital via Zoom on Tuesday 08:15 AM - 10:00 AM.
The exercise introduction lecture schedule is in the following Table.

ID	Weeks	Exercises schedule		
		Introduction lecture	Start	Deliverable
1	3-7	18 Jan. 08:15 - 10:00	18 Jan.	• Final report of exercise 1. 14 February at 23:59 (Monday), Week 7
2	7-12	15 Feb. 08:15 - 10:00	15 Feb.	• Final report of exercise 2. 21 March at 23:59 (Monday), Week 12
3	12-14	22 March 08:15 - 10:00	22 March	• Final report of exercises 3. 4 April at 23:59 (Monday), Week 14

In other weeks without exercise introduction lectures, we will have TAs online via the same Zoom link to answer questions.

Exercise groups

- 1-2 students in each group
- To form a group
  - Send an e-mail with a list of the names and e-mails of the group members to tdt4242@idi.ntnu.no
- If you don't have a group
  - Send an e-mail to tdt4242@idi.ntnu.no
  - You will be assigned to a group
- Deadline: 1 Feb.

Deadline for exercises

All exercises will have a deadline for delivery.

This deadline may only be exceeded after agreement with the

- course responsible
- teaching assistant (email: tdt4242@idi.ntnu.no)

If no such agreement exists, we will deduct 20% points on the grade for any obligatory exercise for each week it is delayed.

Reference group

- A group of 2-3 students that have a special duty to provide feedback about the course during the semester
- We'll have 2-3 meetings where we can discuss content and form of lectures and assignments
- The group should be formed during the first 3 weeks, so please nominate yourself (or others)
- More information (In Norwegian) → http://www.ntnu.no/utdanningskvalitet/roller.html#Studentene

About you

I need to know a little more about you to adapt my teaching focuses and exercises!

Empirical research on Software Ecosystem (SeCo) and Software Startup

Anh Nguyen Duc

Universitetet i Sørøst-Norge

NTNU
Norwegian University of Science and Technology

Agenda

- Software Ecosystem (SECO)
- Software Startups

Software Ecosystem

Introduction

- Software ecosystems is an emerging trend within the software industry
- The shift from closed organizations and processes towards open structures
- Ecosystem metaphor is used to understand relationships among stakeholders

Biological ecosystem

Definitions of Software Ecosystem (SeCo)

- SeCo as an informal network of independent units that have a positive influence on the economic success of a software product and benefit from it (Kittlaus and Clough)
- SeCo as consisting of the set of software solutions that enable, support, and automate the activities and transactions by the actors in the associated social or business ecosystems and the organizations that provide these solutions (Bosch)
- a set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them (Jansen)
- Three shared concepts stand out in these definitions:
  - (1) actors, organizations and businesses,
  - (2) networks and social or business ecosystems, and
  - (3) software.

Menti.com

Examples of Software Ecosystems you know?

Foundation – two-sided market theory

Parker, Geoffrey; Van Alstyne, Marshall (2005). "Two-Sided Network Effects: A Theory of Information Product Design". *Management Science*. **51** (10): 1494–1504

Software Ecosystem (SECO)

- App stores: Apple, Google, Microsoft, etc

Software Ecosystem (SECO)

- Service providers on top of Shopify ecosystem

Other examples of Software Ecosystems

- **Standards** - XML, BPM, OSGi, J2EE, Corba, SEPA, etc.
- **Products** - OpenOffice, Microsoft Word, SAP BusinessOne, Grand Theft Auto, etc.
- **Hardware** - Playstation 3, HTC Diamond, PDAs, BMW 5 series, etc.
- **Platforms** - .Net, Facebook, Android, OS X, etc.

Types of SeCo

Example: The Apple iOS software ecosystem is based on a software platform and coordinated by a privately owned entity with a commercial extension market to which participants can submit extensions after making a payment .

- Core technology
  - software platform,
  - software service platform,
  - software standard
- Coordinated by
  - Community
  - Private organization
- Extension market
  - no extension market,
  - a list of extensions,
  - an extension market,
  - a commercial extension market,
  - multiple extension markets
- Accessibility
  - for free,
  - after a screening,
  - after making a payment

Characteristics of SeCos

- **Complexity -** In a platform ecosystem with numerous actors, complexity must be controlled somehow to reduce risk of gridlocks, unpredictable ripple effects and co-innovation problems
- **Productivity** - A network's ability to consistently transform technology and other raw materials of innovation into lower costs and new products. Simple to measure: return on invested capital.
- **Robustness** - Should be capable of surviving disruptions such as unforeseen technological change.
- **Niche creation** - the ecosystem's capacity to increase meaningful diversity through the creation of valuable new functions or niches.

Strategy as Ecology, Marco Iansiti and Roy Levien. Harvard Business Review, March 2004.

A healthy Software Ecosystem

- A stable and sustainable platform and mechanism to govern the platform
- Developers, software firms making benefits from collective innovation
- Consumers receive high quality services/ products, opportunities for value co-creation

Manikas, K., & Hansen, K. M. (2013). Software ecosystems – A systematic literature review. Journal of Systems and Software, 86(5), 1294–1306. https://doi.org/10.1016/j.jss.2012.12.026

A healthy Software Ecosystem

- Sustainability is the key!

- How to design and maintain a sustainable software ecosystem ?

A healthy Software Ecosystem

- Sustainability is the key!

- A viable business model
- A well-organized inter-organizational interaction model
- A paradigm supporting Collaboration
- A mechanism to allow Coopetition

Raza-Ullah, T., Bengtsson, M., & Kock, S. (2014). The coopetition paradox and tension in coopetition at multiple levels. Industrial Marketing Management, 43(2), 189–198. https://doi.org/10.1016/j.indmarman.2013.11.001

The 3C in SeCo

Coopetition in a nutshell

Netflix uses AWS cloud services while compte with Amazon Prime

https://fourweekmba.com/coopetition/

A model of coopetition

Eriksson, P. E. (2008). Procurement Effects on Coopetition in Client-Contractor Relationships. Journal of Construction Engineering and Management, 134(2), 103–111. https://doi.org/10.1061/(ASCE)0733-9364(2008)134:2(103)

Open coopetition

Coopetition +
Open Source

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

How do commercial companies maintain both collaboration and competition in an open source software ecosystem?

Our cases

Anh Nguyen Duc; Daniela Cruzes S.; Geir Hanssen K.; Pekka Abrahamsson; Terje Snarby (2017). "Coopetition of software firms in open source software ecosystems". ICSOB 2017

Wireshark ecosystem - social analysis

Wireshark ecosystem - conceptual layers

How do commercial firms manage coopetition with other firms in such context?

How do commercial firms manage coopetition with other firms in such context?

- Open-core sourcing policy
- Upstream first - get features integrated into open source projects before integrating them into firm’s product

A model of open coopetition in SECO

Anh Nguyen-Duc, Daniela S Cruzes, Snarby Terje, Pekka Abrahamsson (2019) Do software firms collaborate or compete? A model of coopetition in community-initiated OSS projects. E-informatyka Software Engineering

Software Startups

Software startups

Startup companies are unique:

- Little or no operating history
- Limited resources
- Multiple influences
- Dynamic technologies and markets
- Scalable business model

And more …

Software startups

Startup companies are unique:

- Software in value proposition
- Innovation focus
- Lack of resources
- High uncertainty
- Time-pressure
- Highly reactive
- Rapidly evolving
- Temporary organizational state
- Little/ no history of stable operations
- Product development and business development in parallel
- Seek for product-market fit
- Strong presence of entrepreneurial personalities

- **60%** of startups do not survive in the first five years
- **75%** of venture capital funded startups fail
- Little rigor and relevance exist in the studies about software startups
- Do not focus on investigating issues and challenges

Menti.com

Name a software startup that you know in Norway

Software startups

1. J. Melegati, A. Goldman, F. Kon, and X. Wang, “A model of requirements engineering in software startups,” *Information and Software Technology*, vol. 109, pp. 92–107, May 2019
2. V Berg, J Birkeland, A Nguyen-Duc, IO Pappas, L Jaccheri, Software startup engineering: A systematic mapping study, Journal of Systems and Software 144, 255-274, 2018

Software startup research

- Empirical
- Reveal ar
  reflect on
  phenomer

- **Technical debt:**
  - when development teams take actions to expedite the delivery of a piece of functionality or a project which later needs to be refactored
  - Intentional vs. unintentional
  - prioritizing speedy delivery over perfect code
- Technical debt as integral parts of software startups: the need to shorten time-to-market, by speeding up the development through low-precision engineering activities, is counterbalanced by the need to restructure the product before targeting further growth

1. C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, “Software Development in Startup Companies: The Greenfield Startup Model,” IEEE Transactions on Software Engineering, vol. 42, no. 6, pp. 585–604, Jun. 2016

- Technical debt:

1. C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, “Software Development in Startup Companies: The Greenfield Startup Model,” IEEE Transactions on Software Engineering, vol. 42, no. 6, pp. 585–604, Jun. 2016

Technical debt – four facet model in software startups

1. Cico et al. Technical Debt Approach and Consciousness in Startups Transitioning to Growth Phase - A Four Facet Model. Submitted to IEEE Access, 2022

Technical debt – fo

1. Cico et al. Technical Debt Approach and Consciousness in Sta
Access, 2022

- Domain knowledge, technical expertise, and teamwork are the key components in the early stages of a start-up. Shortages of any of these components can be compensated with specific practices
- Key difference and difficulty to practice software engineering in

Start-up life-cycle phases

INCEPTION Build and launch the first version of the product	STABILIZATION Develop the product further and prepare for scaling	GROWTH Aim to achieve desired market share and growth rate	MATURITY Transition into an established organisation

Engineering process areas

I TEAM

- G1 P1 Establish a team — 12 cases
- C1 Manage a large team — 14 cases
- C2 Lack of team expertise, engagement, coordination, and leadership — 14 cases
- C3 P2 Need for specialist skills — 33 cases
- P3 Outsource engineering work — 6 cases

MENTS
RING

- G2 P4 Validate own, invented ideas — 35 cases
- C4 P5 Establish a feedback loop Customer input drives product work — 32 cases
- Balance customer value with time to market — 6 cases
- Support business needs — 19 cases

1. J.
Sta
2. E.
En
3. C.
Gr

- **Hypothesis engineering** in software startups
  - There are always assumptions
  - Decisions are often driven by the instinct of startup founders
  - Lean startup – is this really used?

1. J. Melegati, H. Edison, and X. Wang, “XPro: a Model to Explain the Limited Adoption and Implementation of Experimentation in Software Startups,” IEEE Transactions on Software Engineering, pp. 1–1, 2020
2. E. Ries, *The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, 1 edition. New York: Currency, 2011.

- Cognitive map to visualize the hypotheses and their relationships

1. J. Melegati, H. Edison, and X. Wang, “XPro: a Model to Startups,” IEEE Transactions on Software Engineering, , ,
2. E. Ries, *The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, 1 edition. New York: Currency, 2011.

Nguyen Duc, A., Shah S. and Abrahamsson, P. (2016). Towards an early stage software startups evolution model. Euromicro SEAA

- People:
  - Startup Founders
  - Stakeholders in Startup Ecosystems
- Cynefine-based categories of startups' decisions:
- Simple decisions:
  - Setting up social media accounts for the startup
  - Focusing on acquiring funding
  - Creating a common team calendar

K.-K. Kemell, E. Ventilä, P. Kettunen, and T. Mikkonen, "Amidst Uncertainty -- or Not? Decision-Making in Early-Stage Software Startups," *arXiv:2102.06501 [cs]*, vol. 370, pp. 369–377, 2019, doi: .

Nguyen Duc, A., Shah S. and Abrahamsson, P. (2016). Towards an early stage software startups evolution model. Euromicro SEAA

- Cynefine-based categories of startups' decisions:
- Complicated decisions:
  - Practicing public speaking
  - Deciding on different pricing models
- Complex decisions:
  - Validating the most important features
  - Gathering a team

K.-K. Kemell, E. Ventilä, P. Kettunen, and T. Mikkonen, "Amidst Uncertainty -- or Not? Decision-Making in Early-Stage Software Startups," *arXiv:2102.06501 [cs]*, vol. 370, pp. 369–377, 2019, doi: .

- Two logics of making decisions: Effectuation and Causations
- Effectuation:

Sarasvathy, S.D.: Causation and eectuation: Toward a theoretical shift from economic inevitability to entrepreneurial contingency. Academy of management Review 26(2), 243-263 (2001

- Two logics of making decisions in startup projects:
  - Will they apply to decisions at technical and product-level as well?

A. Nguyen-Duc, K.-K. Kemell, and P. Abrahamsson, “The entrepreneurial logic of startup software development: A study of 40 software startups,” *Empir Software Eng*, vol. 26, no. 5, p. 91, Jul. 2021

- **What** - Minimum Viable Product
  - … as a prototype
  - … as a communication tool
  - …. as a learning mechanism
  - …. as documentation approach

V. Berg, J. Birkeland, A. Nguyen-Duc, I. O. Pappas, and L. Jaccheri, “Achieving agility and quality in product development – an empirical study of hardware startups,” *Journal of Systems and Software*, vol. 167, p. 110599, Sep. 2020

Nguyen-Duc, A., Wang, X., & Abrahamsson, P. (2017). What Influences the Speed of Prototyping? An Empirical Investigation of Twenty Software Startups. *XP Conference*

- **How -**

  Produ

  - …
  - …

Nguyen-Duc, A., V
Startups. *XP Confe*

Thank you! Questions?

Anh Nguyen-Duc

angu@usn.no