REQUIREMENTS

Expected: Explain different requirements elicitation and documentation approaches; Identify quality issues of requirements and fix them; Define different functional and non-functional requirements

Elicitation & Specification

Requirements Engineering: concerns the machine's effect on the surrounding world and the assumptions we make about it

Tasks: Elicitation – Analysis – Specification – Validation

Elicitation techniques: Interviews, workshops, focus groups, observations, questionnaires, system interface analysis, user interface analysis and document analysis

Specification approach:

CESAR Requirement Specification Languages (RSLs):

- Guided Natural Language: free text requirement format with assistance of prescribed words (simple or more formal). It retains the benefit of free text, doesn't introduce additional formal constraints and does not require a lot of expertise, yet it may be hard to avoid vague terms and words

- Boilerplate: statement-level template, semi-complete requirements parametrized to suit a particular context. It's a grammatical structure with placeholders

  The <system name> shall <function> <object> every <performance><units>

  - User stories: short simple description of a feature told from the perspective of the person who desires the new capability. Single iteration.
    As a <type of user> , I want <some goal> so that <reason>.

  - Use cases: written description of how users will perform tasks on the system. It outlines, from the user's POV, a system's behavior as it responds to a request. Easy to identify cross-cutting and reusable features

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.
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**

  These two boilerplate ways of stating requirements can complement each other.

+ Pattern based, SysML based, video based, Software Cost Reduction (SCR), Matlab Simulink, …

Requirements quality issues

- Complete: responses of the software to all realizable classes of input data in all realizable classes of situations are covered.
- Unambiguous: every requirement stated has only one interpretation

- Consistent: no subset of individual requirements described in is conflict
    - Specific characteristics of real-world objects
    - Logical or temporal conflict between two actions
    - Different terms for describing the same object
- Correct
    - Forward referencing: make use of problem world domain features not yet defined (decir una variable como si ya se hubera definido su valor anteriormente)
    - Opacity: rational or dependencies are invisible (no hay relación entre conceptos descritos como relacionados)
    - Noise: give no information on problem world features (conceptos no definidos en el dominio y que no tienen sentido)
- Verifiable: possible to define a method that determines whether the software meets a requirement. Tests are integrated parts of the requirements, they must be involved from the start.
    - Executing a test (give input, observe and check output)
    - Run experiments (input/output but involving users)
    - Inspect code and other artifacts (evaluation based on documents)

    Testability checklist

        - Modifying phrases: words like “as required” “shall”, etc, make requirement optional
        - Vague words: use words that express what the system must do (“manage, handle, approximately, usually …” are vague)
        - Pronouns with no reference.
        - Passive voice: must be in active voice
        - Negative requirements: only state what the system does, everything else is not done and not needed to specify so
        - Assumptions and comparisons: when comparing, specify to what
        - Indefinite pronouns: subject to interpretation (“All, Everybody, many, most”…)
        - Boundary values: ensure boundary values are defined properly
- Traceable: describe and follow the life of a requirement forward and backwards (origin, refinement and iteration, deployment…). Change analysis, coverage, root causes…)

Column by column explanation

Column Name	Description
Requirement Source	Origin of the requirement — e.g. business rule, stakeholder input, use case.
Product Requirements	ID and name of the requirement. These are the specific features the system must support.
HLD Section #	High-Level Design: abstract architecture of the system (modules, components).
LLD Section #	Low-Level Design: detailed internal logic or pseudocode describing ho to implement each function.
Code Unit	Source code files implementing the logic described in LLD.
UTS Case #	Unit Test Specification: tests verifying the behavior of individual components.
STS Case #	System Test Specification: tests verifying the complete system against requirements.
User Manual	Sections in the user documentation that relate to the requirement.

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

- Ranked for importance/stability: identifier to indicate importance or stability (rank by essential, conditional and optional)
- Modifiable: changes to the requirements can be made easily, completely and consistently while retaining structure and style. Minimize redundancy (boilerplate helps)

Non-functional requirements

NFR: attribute of or constraint on a system. Attributes (-ilities, -ities, -ness); Constraints (physical, legal, environmental…)

Guidelines

- ISO product quality model – defines a structured set of software quality attributes to organize NFRs into recognized categories. Tells you what kinds of quality you might need
- SMART – widely used guideline for writing objectives, useful for NFRs. Helps you write good requirements for each of those.

  Specific, Measurable, Attainable, Relevant, Time-sensitive

- Reliability
  - Turn abstract quality concepts in measurable metrics
    - Availability - % of time the system is operational
    - Recoverability – Mean Time To Recovery (MTTR)
    - Maturity – Mean Time To Failure (MTTF)
  - These metrics are empirically testable with proper setup. Lab test time can simulate a much longer period of use
    - Total Test Time (TTT) - 800
    - Usage Time (UT) – (TTT x 40/20) = 1600

Imagine you're testing **10 identical systems** (same software, same version), and you run them **all at the same time** in the lab for:

- **2 weeks × 40 hours/week = 80 hours per system**
- **10 systems × 80 hours = 800 total test hours (TTT)**

Now, let's say the system in real life is only used by **one customer**, for **20 hours per week**.

So:

- In real life, one system would reach **800 usage hours** after **40 weeks** (800 ÷ 20).
- But in the lab, you've already simulated those **800 hours of usage** in just **2 weeks**, by running 10 systems in parallel.

- The system has a requirement:
  MTTF (Mean Time To Failure) > 500 hours

Let's say **2 failures** occurred during the 800 TTT:

- Then, MTTF =

$$\frac{\text{Total Test Time (TTT)}}{\text{Number of failures}} = \frac{800}{2} = 400 \text{ hours}$$

→ This **does not satisfy** the requirement (400 < 500).

But if **only 1 failure** occurred:

- MTTF = 800 / 1 = **800 hours** → ✓ Requirement satisfied

Term	Value
TTT	800 hours
UT (effective)	1600 hours (equivalent usage)
MTTF Required	> 500 hours
Actual MTTF	800 ÷ #failures
Pass/Fail?	Depends on #failures

- Performance: response speed and amount of resources used
  - Time behavior
  - Resource utilization
  - Capability
- Usability: help users achieve specified goals with effectiveness, efficiency and satisfaction
  - User interface aesthetics
  - Easy to operate
  - Ease to learn
- MbO (Management by objectives) method: way to define qualitative requirements through step-by-step refinement, "what do you mean by…" until reached something testable
  - First step – get vague statements and push to clarify every time
  - Second step – turn statements into testable requirements and create training materials, test problems and usability questionnaires

NFRs require real user testing, some may conflict with each other and can shape the architecture, so take care of them from the beginning.

TESTING

Testing is a software quality assurance approach. Test cases are a simple pair of input, expected outcome.

- Black-box: looking at program from external POV based on specification, focus on if program produces correct output, impossible to write testcase for every set of inputs and outputs.
- White-box: looking at internal structure based on control/data flow, focus on reaching every branch and condition, doesn't address the specification or functionality coverage.

Control flow and Data flow testing (white box)

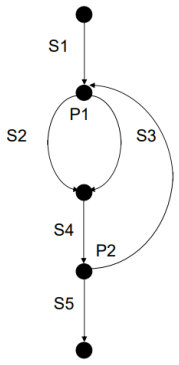
Expected: Explain the coverage criteria explained in the lectures; Create test cases by using different data flow testing strategies; Explain how to use the test coverage information for different purposes

Control flow testing

Structural testing strategy that uses program's control flow as model, pick enough test cases to assure every statement/decision/branch is executed at least once.

Method: generate control/data flow graph and then use different test inputs to achieve different coverages. Path coverage?

> McCabe's cyclomatic complexity: minimum number of paths v(G) = E- N + 2P. E=edges, N=nodes, P=connected components. Maximum number of paths 2^num of conditions. Problem: loops.




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

No DAG. v(G) = 3 and Max is 4 but there is an "infinite" number of paths

> Generate test cases for path coverage – decision table. Make table of all predicates and all combinations of True / False /Switch -1/ 0/many. Might lead to overtesting.

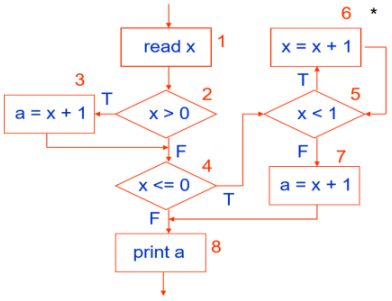
Why measure test coverage?

- Use it as test acceptance criteria, stop testing when reached a certain percentage of test coverage. And the coverage measure will help identify untested code to continue
- Estimate defect coverage (fraction of actual defects that would be detected by a given test set - some might not be observed in small programs

Data flow testing

Variables have a life cycle: Defined and initialized, used and killed. There are some anomalies of the order where you can do all these operations like define-kill, kill-use, kill-kill…

Data flow testing uses the data flow graph to explore the unreasonable things that can happen to data, ad various coverage strategies are employed for the creating of test cases.



Data flow graph is derived from control flow graph with the same set of nodes

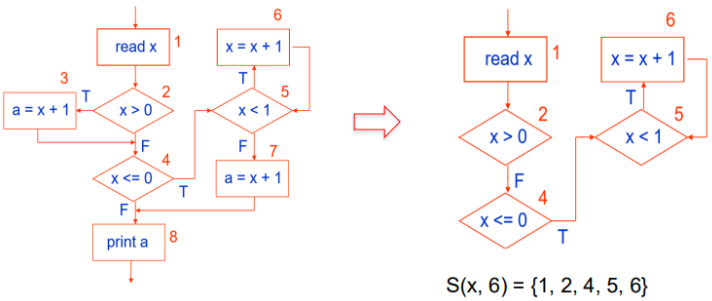
Node	Def	C-use	P-use
1	x		
2			x
3	a	x	
4			x
5			x
6	x	x	
7	a	x	
8		a	

- All definitions (AD): every definition of every variable should be covered by at least one use
- 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 definition, at least one path to c-use should be included, and if there is no c-use following the definition, add p-use test cases to cover those definitions.
- All p-use/ some c-use (ACU+C): for every variable and definition, at least one path to p-use should be included, and if there is no p-use following the definition, add c-use test cases to cover those definitions.
- All uses (AU): for every variable, there is a path from every definition to every use of that definition
- All du paths (ADUP): If there are multiple paths between a given definition and a use, they must all be included. However, ADUP includes loop just once

**c-uses are variables used in arithmetic expressions; p-uses are variables used in conditional expressions

Why measure test coverage?

- Number of bugs detected with data coverage are twice as high as branch coverage.
- Static backward slicing: slice (S(v,p))of a variable (v) at a certain time (p) is the set of statements that have contributed to the value of that variable. All defs and p-uses (yes - p-uses). This is to facilitate debugging and programming understanding



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

Function Testing (black box)

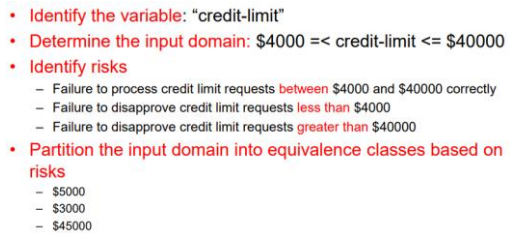
Expected: Apply domain testing approach to generate test cases of single variable and multiple variables in combination; Explain random testing; Explain risk-based testing

We need to reduce the set of all possible values to a few manageable subsets, so we need to do domain partitioning.

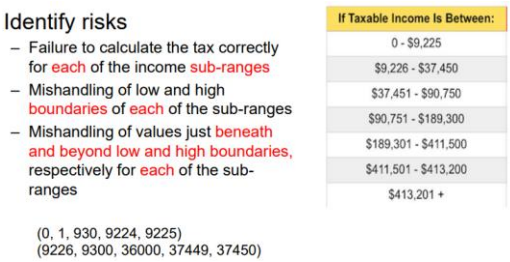
- Identify input and output domain first – linear (integer variables); non-linear (enum values)
- Identify domain partitions – ideally non-overlapping partitions
- Generate test cases to cover a single variable in the domains and combinations of the variables

Single variable

- Equivalence class testing: complete testing with no redundancy, cover each partition at least once. Identify variable, determine domain (all possible values), identify risks and partition the domain into equivalence classes. Variable types:
    - Linear domain variable



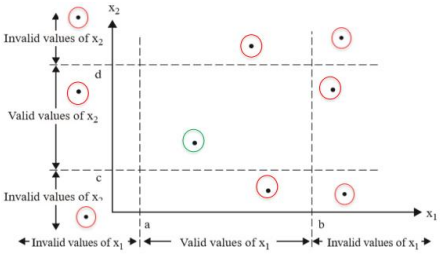
    - Linear domain variable with multiple ranges



    - String variable: mishandling string values, boundary values or extended ASCII set values
    - Enumerated variable: only takes a set of values/options. Mishandling no option, multiple option…
    - Multidimensional variable: more than one direction (example: length and string limit)
- Boundary value testing: check the extreme values of an input variable.
    - Normal: Min, min+, nom, max-, max
    - Robust: – Min-, Min, min+, nom, max-, max, max+
- Special value testing: mishandling non-numbers, negative, smallest and largest value at system level..

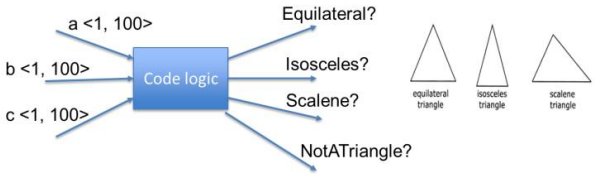
Multiple variables

- Equivalence class testing



- Boundary value testing: hold one in nom and vary the other/s

- Variable combination: all variables will interact as part of one functional unit, so the variables influence each other. Testing them will reveal certain bugs. But do we really need to test all combinations?



- By considering variables in isolation
  - Equivalence class testing <2, 2, 3> ...
  - Boundary value testing <1, 1, 100> <1, 1, 99> <1, 1, 101> ...

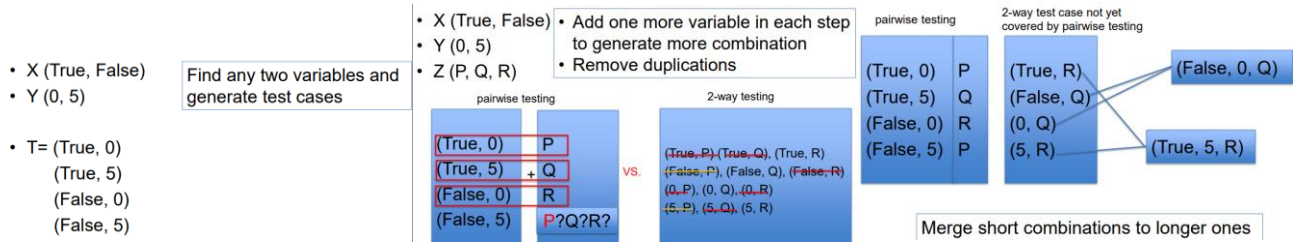
Also need to analyze the logical relationships of the input variables and test their combinations

  - N-way test: given any n variables (out of all variables) every combination of values of these n variables is covered in at least one test.

2-way test example

- 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 (66 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)

  - Pairwise testing: condensed 2-way testing, as many pairs can be tested at the same time in one combination
    - In parameter order (Initialization, Horizontal growth, Vertical growth)



    - Variation of in parameter order

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

- Output coverage testing (not necessarily multiple variables): possible to define coverage based on output data, and uncovered outputs can help us define the new test cases. We can do it by defining an equivalence class for outputs.

Random testing

Feed a random number from a generator to the system and check if behaves as expected. Useful for identifying security vulnerabilities (fuzz testing) as it covers possible attacker's inputs and effective to identify injection attacks; checking concurrency and crashes.

Risk based testing

Testing based on experience (types, probability and consequence of faults). Risk analysis approach:

- Inside-out: what can go wrong here?
- Outside-in: what things are associated with this kind of risk?

Bach identified:

- Generic risk list: important things to test. Complex, new, changed, upstream dependency, downstream dependency, critical, precise, popular, strategic, third-party, distributed, buggy, recent-failure
- Risk catalogue: things often go wrong (domain-specific). Wrong files installed, files clobbered…

Fault injection testing

Measure coverage of seeded faults as an indicator whether the test set is adequate. Seeds faults in the code and checks if test set has found all the seeded faults, and if it hasn't, where has not been covered properly. Draw faults to seed from an experience database with typical faults.

INTEGRATION, SYSTEM, ACCEPTANCE & REGRESSION TESTING

Expected: Explain different approaches for creating integration test cases and their pros and cons; Explain focuses of different types of system tests; Create different system test cases; Explain different categories of acceptance test cases; Explain different test prioritization approaches; Explain regression test selection approaches; Explain different regression test minimization and prioritization strategies

Integration testing

Testing interfaces between components, usually poorly done and not as understood.

Interface errors: construction, inadequate functionality, location, misunderstanding, changes in functionality, data structure alteration, inadequate error processing…

- Intra-system testing: low level integration testing to combine modules together
- Pairwise testing: two interconnected systems at a time
- Inter-system testing: high level testing interfacing independently tested systems

Strategies for intra- and inter- testing based on order

- Top-down: begin with main, replace stubs with real functions one by one
- Bottom-up: begin with leaves, replace drivers with real function later
- Sandwich: mix
  (Pros: incremental and intuitive, easy fault isolation; Cons: need "stub" or "driver")

- Incremental: test cycles, few more modules integrated each cycle based on graphs: neighborhoods integration, path-based integration
  (Pros: no need of “stub” or “driver”, test more global and complex integrations, close to actual system behavior; Cons: difficult fault isolation, extra effort needed to identify message path)
- Big bang: modules individually tested, then all put together to construct the entire system tested as a whole

System testing

- Functionality test: tests designed to verify each functionality, modules function individually.
- GUI test: look and feel, check accessibility, responsiveness, efficiency
- Interoperability test: test to verify the ability of the system to inter-operate with third party products – compatibility and backward compatibility (older versions)
- Robustness test: how sensitive a system is to erroneous inputs or changes in the operational environment (like special value testing)
- Performance test: tests designed to determine performance of the actual system
- Stress test: ensure system can perform acceptably under worse-case condition. Push it to and beyond its limits for a while
- Scalability test: identify how the system can scale, magnitude of demand that can be placed while meeting performance requirements.
- Load and stability testing: ensure system remains stable for a long period of time
- Security test: confidentiality, integrity, availability, nonrepudiation
- Regulatory test: check potential safety consequences and check it follows standards
- Safety assurance: focus on identifying and mitigating hazards

Acceptance testing

Confirm that the system meets the agreed upon acceptance criteria identify and resolve discrepancies.

- User acceptance testing: functions are correct based on user/business requirements, system requirements, use cases, risk analysis reports…
- Operational acceptance test: ready to operate based on backup facilities, procedures for disaster recovery, training or manual for users, security procedures…
- Contract and regulation acceptance test: test against contract and regultions (government, legal, safety standards)
- Alpha and beta testing: at developers sites by internal staff, at customers’ sits

Regression testing

Vast majority of testing effort in many software development projects.

Types

- Corrective regression testing: no requirements change, modified code behaves correctly, unchanged code behaves correctly
- Progressive regression testing: requirements change, newly added or modified code behaves correctly, unchanged code behaves correctly

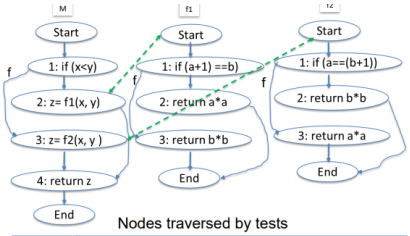
Testing processes:

- Test revalidation: are test cases obsolete? No longer correct due to changes in specification, no longer test due to modifications to the program, no longer contribute to structural coverage
- Regression test selection: retest all / random selection / Safe modification traversing test (does not discard any test that goes through modified statement)

*General selection process: establish traces between program and tests, compare new program with old, select test cases that traverse the changes, use selection methods

- Dynamic slicing-based approach
- Graph-walk approach

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	-

- 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	-

- Firewall approach: separates classes that depend on the changed class from the rest of the classes, first level dependencies of modified components

- Test minimization: reduce redundancies in the safe subset (if all entities of a test are covered by others, remove it). Risky.
- Test prioritization: not remove any test case, just rank them based on some criteria to reveal critical faults early (criteria can be fault finding effectiveness, test coverage and cost of running the test.

CODE SMELLS, REVIEW AND REFACTORING

Expected: Explain why code inspection and testing complement each other; Understand different types of code smell; Explain the purpose and steps of code refactoring; Apply code refactoring methods to identify bad code smells in code (Python) and refactor them.

Code smells

Code smell: characteristic of computer code that may indicate a deeper flaw

- Bloaters: code, methods and classes that have increased in size until they are hard to work with (accumulate over time as program evolves)
  - § Long Method
  - § Primitive Obsession
  - § Data Clumps
  - § Large Class
  - § Long Parameter List
- Object-Orientation Abusers: incomplete or incorrect application of OOP principles
  - § Alternative Classes with Different Interfaces
  - § Refused Bequest
  - § Temporary Field
  - § Switch Statements
- Change Preventers: when you have to change something you have to make changes in many other places too
  - § Divergent Change
  - § Parallel Inheritance Hierarchies
  - § Shotgun Surgery
- Dispensables: pointless and unneeded, cleaner code without them
  - § Comments
  - § Data Class
  - § Lazy Class
  - § Duplicate Code
  - § Dead Code
  - § Speculative Generality
- Couplers: excessive coupling between classes
  - § Feature Envy
  - § Incomplete Library Class
  - § Middle Man
  - § Inappropriate Intimacy
  - § Message Chains

Code review & refactoring

Code review: visual examination of software product to identify software anomalies: errors, code smells, deviations from specifications or standards… Similar terms:

- Code inspection: involves formal and structured review by trained inspector, with a checklist and focusing on detailed issues
- Peer review: type of code review where other developers check a colleague's code
- Code walkthrough: peer review at a meeting reviewing code line by line
- Code audit: formal process by outside expert focusing on both high-level and low-level issues

Many software artifacts can't be verified by running tests (design, requirement specification, pseudocode, user manual). Inspection reduces defect rates, complements testing, identifies code smells and provides other additional benefits.

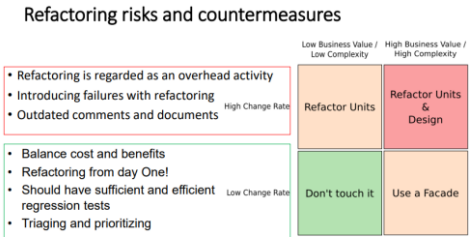
	Testing	Code Review
Pros	• Can, at least partly, be automated • Can consistently repeat several actions • Fast and high volume	• Can be used on all types of documents, not just code • Can see the complete picture, not only a spot check • Can uses all information in the team • Can be innovative and inductive
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

Refactoring: restructuring existing code to improve its readability, maintainability and extensibility without changing its external behavior. Not add new functionality.

- Maintain long-term quality of code
- Readable and maintainable code is desired by industry
- Avoid unnecessary technical debt
- Improve design of application
- Find bugs
- Make program run faster
- Support revolutionary development

Code refactoring methods:

- Rename classes/methods: improve clarity and intention of code
- Remove duplicated code: move repeated logic into a shared method or superclass
- Replace conditional with polymorphism: move conditionals into polymorphic classes so that when adding a new one you just have to create the class.
- Avoid primitive obsession: classes might be clearer and safer that basic types (String code= "12345" NOT; class Code { string value; Code(String value) {this.value = value;} Helps with validation, encapsulation and future flexibility.
- Reduce size of methods/classes: hard to read, test and maintain
- Simplify logic within objects: complicated conditions or loops confuse readers – (instead of if ((age > 18 && hasLicense && !isSuspended)) …; better if(isEligibleToDrive()…).
- Single Responsibility Design: each module/class/function should do one thing.
- Use Comments (only when necessary): clarify why is done, not what is done.



SUSTAINABILITY

Expected:

Explain importance and challenges in software sustainability
Describe different aspects of software sustainability

Sustainable development

Sustainable development: make development so that it meets the needs of the present without compromising the ability of future generations to meet their own needs. Dimensions:

- **Social**: Focusing on improving human well-being, equity, and community.
- **Environmental**: Preserving natural resources and minimizing ecological impacts.
- **Economic**: Supporting long-term economic growth without negatively affecting social and environmental sustainability.

Dimensions added to the general ones in software sustainability:

- Individual: maintaining human capital (health, education, skills, knowledge...)
- Technical: longevity of information, systems and infrastructure

Systems Thinking Approach

Systems thinking involves considering software's broader impact across all sustainability dimensions, where effects can be: Immediate (direct effects of production, operation and disposal of systems); Enabling (change enabled or induced by the system); Structural (structural changes caused by the ongoing use of the system)

Software sustainability

Software whose direct and indirect negative impacts on economy, society, human beings, and environment that result from development, deployment, and usage of the software are minimal and/or which has a positive effect on sustainable development

The **importance** lies in the fact that software systems are deeply embedded in our society, shaping everything from energy usage to human interaction. Poorly designed software can lead to increased energy consumption, reduced maintainability, and social exclusion.

The **challenges** include:

- Integrating sustainability goals into an already complex development lifecycle.
- Measuring indirect impacts (e.g. social or structural effects).
- Balancing short-term efficiency with long-term sustainability.
- Keeping systems adaptable while minimizing environmental resource usage.

Sustainability considerations can be integrated into every phase of the software development lifecycle:

- **Requirements Gathering and Analysis**: Introduce sustainability awareness frameworks and prioritize social sustainability.
- **Design**: Use architectural decision maps and sustainability quality models.
- **Implementation**: Measure the energy consumption of applications.
- **Testing/Quality Assurance**: Include impacted parties in software testing to ensure inclusivity and relevance.

- **Deployment**: Configure efficient physical or virtual servers to minimize resource usage.
- **Maintenance**: Assess the ecological debt of software during its lifecycle.

Viewpoints

- **Process viewpoint:** Responsible use of ecological, human, and financial resources across the lifecycle
- **Product Viewpoint:** Focus on runtime efficiency and minimization of the impact in its domain of use.

Sustainability as a non-functional requirement

Sustainability requirements define the properties or constraints of a software system to ensure it operates with a positive impact on the environment, society, and economy.

Example: Energy efficiency in a cloud computing service - This requirement could involve optimizing server infrastructure to reduce energy consumption without compromising performance.

Web Sustainability

WSG 1.0: Best Practices: Recommendations for creating, maintaining, and operating websites, applications, and digital services in an environmentally friendly and socially responsible manner.

Principles of Web Sustainability: clean, efficient, open, honest, regenerative and resilient.

Sustainable Design Practices: an approach to creating products with a focus on reducing environmental impact while maintaining functionality and usability.

Scalability

Expected: Explain the elements that influence software scalability (slides)

1. Architecture design
Decoupled components allow independent scaling of subsystems based on need.

Event-driven systems are highly scalable because they allow asynchronous, parallel processing and reduce load spikes

2. Shared State Management: Effective handling of shared resources like databases, caches and session stores is crucial. These components must also scale, or they can become single points of failure or performance limits.

3. Deployment Strategies for Smooth Scaling: Modern deployment methods support scalability and resilience: Rolling Updates, Canary Releases, Blue-Green Deployment, Shadow Deployment.

4. Use of Kubernetes: Kubernetes enables scalable infrastructure using containers and orchestration

5. Observability: to scale effectively you must monitor how the system behaves. Uses logs, metrics, traces and alerts following the USE (Utilization, Saturation, Errors) or RED (Rate, Errors, Duration) models. Four Golden Signals (Latency, traffic, errors and saturation)

6. Infrastructure for Autoscaling: Tools like Docker + Kind, Keda.sh... enable autoscaling based on real-time metrics

DevOps

Expected: Explain the challenges of manual deployment & Explain how GitHub Actions works

1. What is DevOps?

Software development methodology that integrates software development (Dev) and IT operations (Ops). The goal is to **increase collaboration**, **speed up delivery**, and **improve reliability** by combining people, processes, and tools.

Key ideas: encourages **automation** and **continuous feedback**, emphasizes being familiar with **the entire software lifecycle**, not just writing code and enables **faster and more reliable deployments**.

2. DevOps Lifecycle Stages

Planning: Define what needs to be built, changed, or tested. Level of formality depends on context (simple notes vs full requirement engineering).

Creating: developers code the solution in local environments using version control (e.g., Git) to track changes, quick feedback via local deployment.

Verifying: Includes **automated tests** (unit, stress tests) and **manual tests** (e.g., black-box testing). Must verify all features, especially new ones.

Packaging: use **Docker** to containerize each component (frontend, backend, gateway). And run multicontainer apps with internal networking and volume sharing with Docker Compose.

Releasing: Controlled deployment process. Example pseudocode: On push to production, log in to server, pull image, start and verify.

CI/CD (Continuous Integration/Deployment): Automate the pipeline: build → test → deploy. Ensures new code is tested and deployed quickly and reliably.

Configuring & Operating: running and managing the software in production. Involves scaling servers, managing resources, applying updates, load balancing during heavy traffic…

- **Networking: Reverse Proxy (s**its between users and the backend system and protects servers, provides caching, encryption, and basic load balancing), Load Balancer (distributes requests across multiple servers, increases fault tolerance and performance)
- **Orchestration:** automating and managing containerised apps (e.g., with Kubernetes). **Example**: Load balancer → frontend, backend, and database → scale automatically based on demand.
- **Infrastructure as Code (IaC):** defined programmatically instead of manually. Allows you to declare resources, version control infrastructure and recreate identical environments
- **Configuration Management**: ensuring systems remain in the desired state. Define *what* the system should look like, not *how* to achieve it.

Monitoring: collecting and visualizing data to ensure system health. **Pull vs Push models** for getting data from servers, tools gather logs, metrics, and traces, and dashboards and alerting rules help detect and act on problems early.

3. Challenges of Manual Deployment

Manual deployment involves logging into servers, copying files, restarting services, and configuring environments by hand. This leads to several problems:

- **Error-prone**: Small mistakes can break production.
- **Time-consuming**: Repetitive tasks waste developer time.
- **Inconsistent environments**: Differences across machines cause bugs

- **Difficult to reproduce**: No history of how the system was set up.
- **Poor scalability**: Not feasible when deploying frequently or across many servers.

4. How GitHub Actions Works

GitHub Actions is a CI/CD tool built into GitHub that automates workflows like building, testing, and deploying code.

- **Event-driven**: Workflows are triggered by events (e.g., push, pull_request, or schedule).
- **Workflow files**: Defined in YAML, stored in .github/workflows/ directory.
- **Jobs**: Each workflow consists of one or more jobs that run on virtual machines (runners).
- **Steps**: Each job contains steps, which are commands or actions.
- **Actions**: Reusable units of code (e.g., "checkout repo", "deploy to server").

Typical Workflow: Developer pushes code to main**,** GitHub Actions triggers the workflow**, i**t runs tests, builds the application, and if successful deploys the updated app automatically.

AI & Software Engineering

AI Software Engineering: training-based model development, non-deterministic, dynamic, highly data-dependent, continuous learning and evolution, performance evaluation on datasets, model explainability and require continuous monitoring, retraining and updates

Process

- **Problem definition**: understanding use case and objectives. Probabilistic behaviors make it hard to define deterministic requirements
- **Data Collection & Preprocessing**: gathering and refining data
- **Modeling Selection and Training:** choosing algorithms and training methods. Evaluation metrics like accuracy, precision, recall, F1-score…

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Use when: Classes are balanced (e.g. roughly equal number of "smelly" and "clean" methods).

$$\text{Precision} = \frac{TP}{TP + FP}$$

Use when: False positives are costly. (e.g., you don't want to flag clean code as smelly).

$$\text{Recall} = \frac{TP}{TP + FN}$$

Use when: False negatives are costly. (e.g., you don't want to miss any real code smells).

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Use when: You need a balance between precision and recall, especially in imbalanced datasets.

- Model evaluation
- Deployment: integrating AI models into application
- Monitoring & Maintenance: ensuring model performance over time

Model development and adaptation

- Pre-training and post-training: training model from scratch (random weights) / finetuning, training previously trained model (get weights from previous training)
- Dataset engineering: curating, generating and annotating data needed for training and adapting AI models
- Interference optimization: making models faster and cheaper with model compression, hardware optimization, efficient model architectures…
- Prompt engineering: getting models to express the desirable behaviors from the input alone, without changing the model weights.
    - Zero-shot/Few-shot: give examples of the task

- System/User prompt: system sets overall behavior, role or tone; and user has the actual input or question from the user
- Personas: give character or role the AI will adopt
- Chain of thoughts: make reasoning as example of the reasoning AI has to make

Testing AI systems

- Input Testing: analyse training data and input data at prediction time
- Model Testing: consider model in isolation
- Integration Testing: testing in software components, hardware components…
- System Testing: on complete integrated ML to evaluate compliance with requirements

Machine learning

Classical AI

Supervised learning – trains on labeled data (input-output pairs). Given (x1,y1),(x2,y2)… learn a function f(x) to predict y given x.

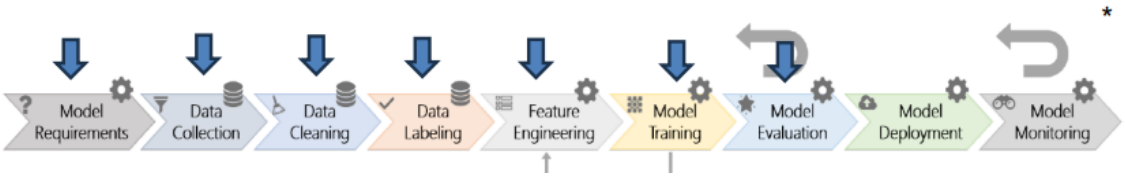
- Classification: y is categorical. Useful for detecting code smells.
- Regression: y is real-valued. Useful to estimate functional size for effort estimation

Unsupervised learning – learns patterns from unlabeled data

- Clustering: given x1, x2,… output hidden structure behind the x's. Useful for improving regression test case prioritization based on code coverage

Reinforcement learning – learns from interaction with the environment using rewards. Learn a policy for taking the optimal action in the observed states. Applied in sequential choice tasks, trial and error. Useful for testing (reward function favors actions that achieve high coverage and penalizes actions that activate marginal computations)

Workflow



Challenge on interference optimization

One challenge with foundation models is that they are often autoregressive (output variable depends linearly on its own previous values)

Making the correct model decision, when to build or buy (API cost vs engineering cost)

Tests in lab during model training/testing can be very different to results in operation

Generative AI for software engineering

Image generator – uses Generative Adversarial Network. A generator makes samples of data to fool discriminator, while it tries to distinguish real and fake. Generator learns from mistakes and creates better data. Useful for GUI designs

LLMs – prompts converted into tokens and the system analyzes what is likely to come next, based on the tokens in its own dataset

Code generation using Copilot

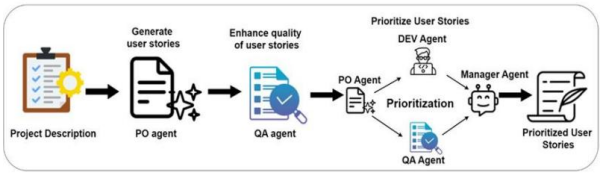
Applications for Software Engineering: requirements engineering, code generation and completion, test generation, minimization, output predication, code repair and refactoring…

Issues: hallucinations, non-deterministic, explainability, insecure code… A model is only as good as its data

Augmented, compound and agentic AI

Augmented (Retrieval-Augmented Generation – RAG): enhanced with external tools, systems or other knowledge sources to reason more effectively. Useful to summarize code using its comments and resulting in textual descriptions of the code; generate traceability between requirement and code.

Compound: combine multiple AI components working together in structured workflows (each component does part of the job and passes the result to the next)

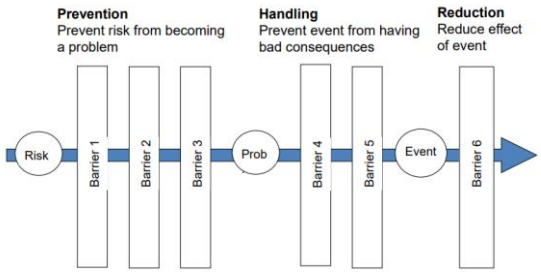


Agentic: like compound but not all components are LLMs, there should be an orchestrator and the information flows does not have to be predefined. Useful for game testing.

Safety

Expected: Explain how to define system integrity level and safety cases; Explain the advantages and challenge of using simulation-based testing.

Relationship between a system, its functions and the systems operating environment so that nobody gets hurt when using the equipment, even when it fails. Integrated part of the project and system, in agenda and with experts from day one.



Experience Harvesting: keep a database (hazard log) with experience related to accidents, near-accidents, unexpected behavior, malfunctions… and check handling coverage or reusability.

Software safety requirements: obtain safety requirements, decide how to achieve them and how to convince the client they are met. Get error consequences, probability and controllability -> Safety Integrity Level will decide how to approach requirements, design, development, testing and maintenance. Define SIL level (1-4) depending on those three error aspects, and then follow the techniques specified on the table and address the most important ones depending on the level.

Linguistic variables: agree upon definitions for variables like “a very slight probability”, “a high probability…”

	Technique/Measure *	Ref	SIL 1	SIL 2	SIL 3	SIL 4
1	Software module size limit	C.2.9	HR	HR	HR	HR
2	Software complexity control	C.5.13	R	R	HR	HR
3	Information hiding/encapsulation	C.2.8	R	HR	HR	HR
4	Parameter number limit / fixed number of subprogram parameters	C.2.9	R	R	R	R
5	One entry/one exit point in subroutines and functions	C.2.9	HR	HR	HR	HR
6	Fully defined interface	C.2.9	HR	HR	HR	HR

Agile and safety

In the Scrum process, you can add things to make it the SafeScrum process to ensure safety. You can also add Alongside Engineering, outside the realm of software development to ensure safety. Also add question 4 to agile development "Any safety related impact of the completed work?"

Hazard story: As a result <cause> <cause event> of which will lead to <accident event> [if <accident condition>]

Safety story: To keep <function>safe, the system must achieve or avoid <something>

Safety cases

Argument that a system is safe to use in a specified environment with claims (system is safe), arguments (because we have done A and B)* and proofs (A and B work and the evidence is in document D).

Identify system (what is inside and what is not), safety/use context (where are the arguments valid), assumptions (which have we made to arrive at our arguments).

Notations: prose (use short simple words, imperative mode, numbered steps…), structured prose (fewer words, text replaced by indentations, replace context with references to documents), or graphical notations

*Important to plan in advance, not consider evidence that is not used to support and argument, make them short and concise (else it looks like you are hiding something)

Main advices

- The language used throughout is simple – easy to read and easy to understand
- The goal – and possible sub-goals – is correct – this is really what you care about
- The arguments used are relevant and sufficient to meet the goal
- The poofs – evidences – are complete and believable and contain
    - Methods and tools (if any) used
    - The knowledge and experience of those who did the job
    - References to all relevant documentation

Simulation driven testing

To ensure that an autonomous system can be trusted we can use 'Simulation-driven' development. It helps to get feedback, make stress tests without safety concerns, scalabiliy, reproducibility…

But you can't only check that the system works, you also have to look for evidence the system does not work – get creative to think about scenarios where things can go wrong, also make real-world testing, use AI to discover failure scenarios, fuzz testing to discover weaknesses in the code, third party testing and asserts

- Adaptive stress testing: reinforcement learning techniques to find scenarios where the autonomy fails (discover difficult traffic patterns or environment conditions)
- Fuzz testing: provide a program with random inputs and monitor for crashes or errors

- Third party verification: use someone outside the development team to test the system independently, as internal teams might have confirmation bias (test what they think is important)
- Assert: check that something is always true, else the program crashes immediately (assert(x> 0)) to check preconditions, postconditions and invariants.

Limitation of Simulation testing: fidelity, additional research to create good metrics (sometimes it is feeling based), and it requires high computing and vertical and horizontal scaling.