Questions: 1.S-Process 2.S-Requirement 3.Software Architecture 4.Technical Debt and S-Maintenance 5.Code with Quality 6.Program Testing 7.Modern Code Review

Software Process

1.A *software engineering process model* is a set of (1) activities (2) techniques (3) deliverables (4) tools, so that software engineers and project managers follow (5) *best practices* to use the items (1)-(4) to manage software systems.

2.SE Process-Activities: (1) Collecting user requirements (2) Designing software (3) Coding/Implementation (4) Testing (5) Deploying the software at user sites (6) Maintain software (7) Configuration Management (8) Project Management.

3.Deployment diagram: Map software component to run on hardware device (or hardware virtualization). Write scripts or use a tool to publish it.

4.SE Process-Techniques, corresponding to SE Process-Activities: (1) Use cases/user stories/use case diagram/meeting (2) UML/ patterns/principles/tactics (3) Java/C++/ framework/platform (4)Unit test framework /debugger/… (5) Standalone software/plug-in/app/web services/… (6) Bug reporting/ software repository (7) Version control/ change management software (8) Work breakdown structure/scheduling algorithm.

5.SE Process-deliverables, corresponding to Techniques: (1) Requirement document (2) Software design document (3) Code listing and test script (4) Quality assurance (QA) report (5) Software at user site (6) Bug report & s/w release (7) Code change/patch/ change history report (8) Project schedule and status tracking.

6.SE Process-tools, corresponding to Techniques: (1) UML tool/ Requirement modeling tool/Word (2) UML tool/ SonarQube (3) Visual Studio/Eclipse (4) Junit/debugger/fuzzer… (5) Installer/plug-in framework (6) Bug reporting system (7) Apache subversion/git (8) MS Project/ Scrum tool.

7.SE Process-Best Practices: Different ordering of activities defines different types of software processes: (1) Waterfall model (2) V-shape waterfall model (3) Incremental software development model(4)Prototyping (5) Spiral model (6) Unified Process (UP) (7) Agile Methods (8) Hybrid of Agile Methods with Waterfall (Time ASC)

8.Hint: If an agile method requires fewer specific tools, activities, process steps, and intermediate products, then the agile method is more lightweight.

9.Extreme Programming vs Scrum: (1) XP has no PM (2) Scrum has PM (but a less tedious process than XP) & other practices/ activities/work products.

10.Waterfall Model: Completely produce the full set of *deliverables* of each activity before starting the next activity. Workflow: Requirements-Design-Coding/Implement-Testing-Deployment-Maintenance
Advantages: Simple workflow
Disadvantages: Need test against design, coding details, requirement at the same time.

Feature:(1)*Sequential*: Complete one phase nicely before the start of the next phase (2)*Detailed documentation*: Facilitate every one comprehensively understanding each phase (3)*Predictability in cost and timeline*: through upfront and high-quality planning (4)*Rigidity and Incompatibility with change*: Once confirmed, no change (5) *Late discovery of issues* (6) *Unsuitability of undefined projects*.

11.V-Shape Waterfall Model: + Stagewise Validation Goal.

12.Problems of "Waterfall Model" and "V-Shape Waterfall Model": (1) False sense of clear-cut phases of activities. (2) Nothing is done until they are all done.

13.Lesson Learnt: (1) Cost of fixing bugs at earlier stage is cheaper. (2) Implication of using Waterfall Model: we would rather have a gracefully degraded **sub**system which means a partial system, a system of lower quality, instead of no system at end.

14.***Software Process Improvement:*** (1) *Some stages (activities) can be done in parallel.* E.g.: Run technical development and user training in parallel. ~Development and QA within each parallel track. ~Each parallel track fits within a self-contained subsystem. ~Quicker for each parallel track to move forward, thereby earlier to rectify bugs->lower cost. (2) Backward Iterations, Feedback: From Design to Requirement ~Notice the direction of the arrows. ~Improve earlier stages whenever necessary. ~Avoid unnecessary rework or design bugs by working out the more accurate requires. (3) Model-Based Software Engineering Process (various modeling skills as the tecs).

15.Prototyping: (1) Prototyping = Code before design and requirement engineering. Incrementally obtain and validate the user/ system requirements before proper software design stage. (2) Prototype is a preliminary version of the final product: buggy with features partially implemented or only with mockup screens so that users can point out their requirements more easily rather than expressing them more abstractly.

16.Incremental Development: (1) Rather than using a parallel track to work on a smaller subsystem, another way is to deliver a smaller set of requirements and gradually enlarge the requirement set. (2) Divide the set of requirements into subsets. (3) Parallel tracks with each stage of each subsystem development are possible.

17.Spiral Model: (1) Contain characteristics of the waterfall, prototype, and incremental development, but is more systematic. (2) ***Basic idea:*** plan, do prototyping, and revise the previous plans until we know the item X well through a series of iterations. (3) **Spiral Software Life Cycle:** ~Step1: Determine objectives, alternatives, and constraints. ~Step2: Evaluate alternatives, identify risks, resolve risks. ~Step3: Develop, validate, verify next-level. ~Step4: Plan next level.

18.(Rational) Unified Process: Every type of activity runs in parallel (vertically) with other types of activities in the same iteration.

19.Ideas of Test-Driven Development: (1) Write automated test case. (2) The minimal functionality of the code that can pass all test cases) + applying coding principles (or code refactoring) to keep codebase better organized. (3) No obvious maintenance phase. (4) Each iteration must be short in duration in terms of time.

20.Agile Method: (1) *Iterative development*: Deliver works in small increments to users. (2) *Customer collaboration*: Evaluate and feedback on the values of requirements/ increments. (3) *Response to changes*: Continuously evaluate the requirements, plan, and deliver increments with user feedback. Change rapidly. (4) *Inadequate documentation*. (5) *Unpredictable delivery time and costs*. (6) *Dependence on client involvement*. E.g.: Extreme programming (XP) has five phases: (1) *Planning*: Meet with users, define user stories, and estimate story points. Plan for release through N iterations (for user stories delivery). (2) *Design*: Use Simple and consistent design sketch. (3) *Coding*: Apply XP practice. (4) *Testing*: Conduct automated unit tests and acceptance tests per user story.(5) *Feedback*: PM and users determine the values of the user stories delivered by the implementation. (6) *Emphases*: ~Generate feedback. ~Embrace changes. ~Keep customers engaged. ~Short iteration. ~Fix bugs early.
XP guidelines: (1) ***Planning***: programmers estimate efforts needed for implementing user stories and customer decides the scope and timing of release. (2) ***Small release***: monthly, or daily for small fixes. (3) ***Metaphor***: A shared story guides all developments by describing how the system works. (4) ***Simple design***: use simplest possible solution. (5) ***Testing***: use tests are implemented before the code. Customers write the functional tests. (6) ***Refactoring***: do refactoring frequently. (7) ***Pair programming***: two people write code at one computer. (8) ***Collective ownership***: anyone can change any part of the code at any time. (9) ***Continuous integration***: integrate the code to the project codebase as soon as it is ready. (10) ***On-side customer***: customer is available full-time. (11) ***Coding standards***: apply them. (12) ***Open workspace***: a large room with small cubicles preferred. (13) ***40-hour week***: No Overtime in two consecutive weeks. (14) ***Just rules***: Team has its own but changeable rules for all to follow.
Popular PM elements in Agile Methods: (1) *Estimate the story point for each user story*: Allocate extra time for research on a user story with high uncertainty. (2) *Pick a set of user stories into the next iteration in XP (equivalent to the next sprint in Scrum)*. (3) *Within a cycle (an XP iteration or a Scrum sprint), conduct (daily) standing meetings*: A super short meeting (15 mins in total), Every team member reports: ~what was done yesterday, ~what will be done today, ~what blockers are encountered. For this item, other teammates may help resolve it. (4) *Measure project velocity*. (5) *At the end of each cycle, conduct a review (a demo to users about the deliverable achieved in the cycle) and a self-reflection on the cycle.*
User story: (1) A user story is in the following format: As a <role>, I want an <action>, so that I can achieve a <goal>. (2) A user story usually has more information such as use cases, diagram sketch, user notes, data, reports from existing systems to clarify the context and scope of it.
Relative Story Point Estimation: (1) A story point is a value. (2) Pick one or more user stories as anchors. The whole team agreed on a story point for such user story to indicate the number of expected efforts to implement the user stories, considering uncertainty and implementation complexity.
Project Velocity: (1) A user story is implemented in a cycle if it passes the cycle review (by users and product owner). (2) An agile project is organized as a series of cycles (XP iterations or Scrum sprints). (3) Each cycle implements a set of user stories with assigned story points. (4) If a user story is too large to fit into a cycle, the user story should be broken down into multiple user stories. (5) The total story points delivered by the cycle are plotted.
Burndown Chart: Project velocity is the slope over a consecutive series of cycles.
Scheduled versus Actual: (1) When creating a release plan or a plan for a sprint, the project manager needs a schedule. (2) Adjust according to actual.
Scrum: (1) Is a kind of agile development process. (2) Arguably the most widely used development process nowadays. (3) driven by daily and periodic one-hour meetings: ~Go through the usage scenarios/stories to identify a slice of the highest priority backlog tasks to be completed by the next iteration (Sprint) agreed upon by the customer. ~Together with customers, discuss the goal of the current Sprint, prioritize functions to be completed, and divide into detailed tasks. ~Conduct periodic short planning and review meeting for tasks completed and not completed in time and revise the set of backlog tasks accordingly. (4) Customers may change their mind at any time: ~Accept that the requirements cannot be fully understood or defined, ~Focus on maximizing the team's ability to deliver quickly and respond to the selected backlog tasks. (5) A product backlog = user stories and development tasks due to the completion of some user stories (e.g., bugs found in a later cycle). (6) Priority the product backlog. (7) Suppose a cycle = two weeks (8) A high-priority subset of the product backlog is treated as the current sprint backlog: The size of the sprint backlog is determined by the total number of story points that the team can be delivered within a cycle based on the history. (9) Through the standing meeting, each team member picks an item from the sprint backlog. The target is to deliver the user story in the cycle. (Visualize as **task board**): ~The tasks needed to deliver the story when it is started are all placed into the "to-do" column of the task board. ~When a task is being worked on, it is moved to the "in-progress" column. ~When implementation of the task is completed, it is moved to the "to verify" column (which is usually merged with the-progress column) (10) At the end of a sprint, conduct a sprint review: Product owners/users verify whether the user story is delivered. If ok, then move the task to the "Done" column. (11) Also, conduct a sprint retrospective (reflect on what has been done right to make progress and what to be improved to reduce or avoid blockers to improve productivity).
Scrum Tools and Meeting: (1) *product backlog* is a list of features desired for a final product, the bugs to be removed, technical work to set up and maintain development environment and user site, and knowledge (e.g., learn to use a new framework) to acquire by the project. (2) *release burndown chart* tracks progress on a project. The chart itself is updated after each sprint. Teams can measure progress in any unit they choose. (3) *sprint backlog* is a list of tasks to complete during a sprint. It is updated once a day. (4) *task board* is a sheet that every member of the team can use and add to over the course of a sprint, and is a visual representation of every task and what phase of completion it's in. Usually, task boards include columns for stories, to-dos, work in process, things needing verification and items finished. Some teams also include burndown charts, notes and tests. ***Hang the board on a wall*** or digitalize it. (5) *Meeting:* The product owner shows up at the sprint planning meeting with the prioritized agile product backlog and describes the top items to the team. The team then determines which items they can complete during the coming sprint. The team then moves items from the product backlog to the sprint backlog. In doing so, they expand each Scrum product backlog item into one or more sprint backlog tasks.
Scrum Roles: (1) ***Scrum Master*** is the team's coach and helps Scrum practitioners achieve their highest level of performance: ~This role does not provide day-to-day direction to the team and does not assign tasks to individuals. ~In many projects, this role is assumed by the project manager. (2) ***Product owner*** is to prioritize the backlog during Scrum development. (3) ***Scrum development team*** as a whole.

21.Hybrid Method:(1)Agile PM + Waterfall.

(2) Flexibility and structure: ~Iterative to deliver increments rapidly. ~Adhere to (requirements) documentation. (3) Phased and iterative: ~Phased approach for well-defined components. ~Iterative approach for uncertain one.(4) Customer involvement and predictability: getting feedback early and often. (5) Applicable when: ~Diverse stakeholder needs, ~Varied project phases, ~Uncertain requirements, ~high-risk project with complex project structures.

Software Requirement

1.Introduction Example: "As a customer, I want to buy a PS5 from the website". This is a **functional** and **verifiable** requirement. We must know the non-functional part.

2.Requirements Engineering (RE): (1) RE is a process to **find out** and **structure** functional & non-functional **requirements** of the software to be built. (2) In many cases, a high proportion of all requirements are non-functional requirements. (3) is the first step in finding a solution for a data processing problem. (4) the results of requirements engineering is a requirements specification. (5) requirements specification is a: ~contract for the customer. ~starting point for design.

3.Many faces of non-functional requirement: (1) Be careful! (2) Different people in different domains may use different terminologies. (3) Example "words" from users to mean non-functional requirements: ~System property/characteristic/constraints, ~Quality attributes, non-behavioral requires, concerns, goals, extra-functional requires, quality requirements and system attributes. (4) Different application domains have different sets of major quality attributes. (5) Even in the same application domain, different types of applications have different sets of non-functional requirements. (6) Some quality attributes are undefined or vaguely known. (7) If a quality attribute **cannot** be measured or **cannot** be verified, we **cannot** know whether we cannot predict how well our software meets them: For such quality attributes, we can only rely on user evaluations on the software.
Non-functional requirements are as follows:

1 Accuracy	10 Installability	19 Reusability
2 Availability	11 Integrity	20 Safety
3 Communicativeness	12 Interoperability	21 Scalability
4 Compatibility	13 Maintainability	22 Security
5 Completeness	14 Performance	23 Standardizability
6 Confidentiality	15 Privacy	24 Traceability
7 Conformance	16 Portability	25 Usability
8 Dependability	17 Provability	26 Verifiability
9 Extensibility	18 Reliability	27 Viability

4.We need to Handle Multi-Level Concerns: (1) specific to the application domain: meet the general requirement of industry sector. (2) specific to type of software application: meet the general requirement for the type of software. (3) the specific to the application: meet the unique requirements of the current application. (4) In all levels, consider non-functional requirements. (5) If not, our application will not be used easily in the application environment.

5.Avoid Common Mistakes in RE: (1) **noise**: does not add relevant information. (2) **silence**: feature in problem not mentioned in specification. (3) **over-specification**: talk about the solution rather than the problem. (4) **contradictions**: inconsistent description. (5) **ambiguity**: unclear. (6) **forward references**: especially cumbersome in long documents. (7) **wishful thinking**: features that cannot realistically be realized.

6.Natural language specs are dangerous: We should *validate documented* requirements to avoid misunderstanding. It leads to the RE process.

7.There are 4 major steps in an RE process: (1) understanding the problem: **elicitation**. (2) describing the problem: **specification**. (3) agreeing upon the nature of the problem: **validation**. (4) agreeing upon the boundaries of the problem: **negotiation**. (5) This is an iterative process.

8.Conceptual modeling: (1) We **explicitly** model the scope of the requirements as a domain model. (2) Making a model explicit requires solving two problems: ~**Analysis**: to find and clear ambiguity: Because of the unspoken assumptions, different languages/ terminologies being spoken, incomplete codification of the problem domain. ~**Negotiation**: to resolve ambiguity due to stakeholders with different goals (e.g., worker against management), As an analyst, we must participate in shaping the domain model. (3) Have an explicit model reduces later surprises.

9.How we study the world around us: (1) people have a set of assumptions about a topic they study. (2) this set of assumptions concerns/effects: ~how knowledge gathered. ~how the world (they know) is organized. (3) this results in 2 dimensions: ~subjective-objective (w.r.t. knowledge) ~conflict-order (w.r.t. the world). (4) which result in 4 basic positions to requirements engineering that an analyst can take.

10.The Positions of Analyst in RE: (1) **functional** (objective + order): the analyst is the expert who empirically seeks the truth. (2) **social-relativism** (subjective + order): the analyst is a 'change agent'. RE is a learning process guided by the analyst. (3) **radical-structuralism** (objective+ conflict): there is a struggle between classes; the analyst chooses for either party. (4) **neo humanism** (subjective + conflict): the analyst is kind of a social therapist, bringing parties together.
10.1 Functional: (1) Conflicts are solved by management. (2) Management tells how things go. (3) Requirements are well-articulated, shared, objective. (4) Development is formal, rational. (5) Politics is irrelevant. (6) Reality is measurable, and same to everyone. (7) Design is a technical problem.
10.2 Social-Relativism: (1) There is not one truth, there may be different perceptions. (2) An information system is part of continually changing social environment. (3) System goals result from an organization shaping its own reality. (4) There is no objective criterion of good or bad. (5) The goal is to arrive at a consensus.
10.3 Radical-Structuralism: (1) A struggle between classes; the analyst chooses for either party. (2) There is a fundamental truth, there also is a fundamental battle between classes. (3) You may choose for either management (machines replace people, machines guide people's work, supervision, loss of jobs, less interesting work) or for the workers (enhance labor skills, make work more interesting and challenging).
10.4 Neo humanism (subjective + conflict): rather hypothetical; emancipation, remove barriers.

11.Requirements Elicitation: (1) Knowing the position we play in the RE process of a project, the next thing we do is to collect the requirements. (2) The activities in the requirements elicitation process can be classified into five types: ~Understanding the application domain, ~Identifying the sources of requirements, ~Analyzing the stakeholders, ~Selecting the techniques, approaches, and tools to use, ~Eliciting the requirements from stakeholders and other sources. (3) Since Elicitation is an iterative process, we cannot finish one type before starting another.
11.1 Understanding the application domain: (1) Start from understanding the application domain. (2) The current environment of the system needs to be thoroughly explored including the political, organizational, and social aspects related to the system, in addition to any constraints they may enforce upon the system and its development.
11.2 Identifying the sources of requirements: (1) Requirements spread across many sources and exist in different formats. (2) People: ~Stakeholders: the main source of requirements for the system, ~Users and subject matter experts: supply details and needs. (3) Things: ~E.g., Existing systems and operation processes, ~E.g., Existing documentation (e.g., manuals, forms, and reports), ~provide useful information about the organization, environment & rationales.
11.3 Analyzing the stakeholders: (1) Stakeholders are people who have interest in the system: ~Groups and individuals internal and external to the organization, ~Customers and direct users of the system. (2) Analyze and involve all the relevant stakeholders: ~The process of analyzing the stakeholders also often includes identifying key user representatives and product champions (who support the software). (3) Consult during requirements elicitation.
11.4 Selecting techniques, approaches, and tools to use: (1) Requirements elicitation is best performed using a variety of techniques (e.g., interviewing and form analysis): In projects, several elicitation techniques are employed during and at different software development life cycle stages.
11.5 Eliciting requirements from other sources and stakeholders: (1) After above four preparation steps, this step is the actual step to collect requirements. (2) Targets: ~Establish the scope of the system, ~Investigate in detail the needs and wants of the stakeholders, especially the users, ~Determine the future processes the system will perform with respect to the business operations, ~Examine how the system may support to satisfy the major objectives (and address the key problems) of the business.

12.Requirements Elicitation Techniques: ***interview***/Delphi technique/***brainstorming session***/***task*** ***analysis***/***scenario analysis***/***form analysis***/ethnography/analysis of natural language descriptions/questionnaire/***mind mapping***/synthesis from existing system/ domain analysis/Business Process Redesign (BPR)/Prototyping/***group storytelling***/***user stories***/***Focus group***/***Facilitated workshop***.
12.1 Interview: (1) Used in requirements elicitation. (2) Ask stakeholders questions about the current application's usage and usage of the application to be built. (3) Good at getting an overall understanding of stakeholders' needs & current application's problem from their views. (4) Inherently informal, and their effectiveness depends greatly on the quality of interaction between the participants. (5) Interviews provide an efficient way to collect large amounts of data quickly. (6) Three types of interviews: unstructured, structured & semi-structured (hybrid of the former two). (7) Unstructured: ~Limited control over the direction of discussions, ~Best applied for exploration when there is a limited understanding of the domain, ~Risky: Much detail in some areas, and not enough in others. (8) Structured: ~using a predetermined set of questions to gather specific information ~The success of structured interviews depends on knowing what the right questions are to ask, when should they be asked, and who should answer them. Templates that provide guidance on structured interviews for requirements elicitation, such as Volere cards, can be used. ~Questions can be open-ended first then close-ended (more specific).
12.2Brainstorming session: **Brainstorming** is a group-based activity by gathering a list of ideas iteratively toward a particular topic.
12.3 Task Analysis: (1) Task analysis analyzes how people perform their jobs: the things they do, the things they act on, and the things they need to know. (2) It employs a top-down approach where high-level tasks are decomposed into subtask and eventually detailed sequences until all actions and events are described: It determines the knowledge used or required to carry tasks out. (3) E.g., how a SGS staff completes the student enrolment task, in which typical and atypical situations need to be handled.
12.4 Scenario-Based Analysis: (1) Provide a user-oriented perspective on designing and developing an interactive system. (2) E.g., observe how the SGS staff handles the enrolments of a non-local student A and a local student B. Then, ask questions on cases not yet covered.
12.5 Form analysis: Figure out items that are certain or have variety, and the time (past, while filling the form, or future) that the information for the item is available.
12.6 Focus Group and Facilitated Workshop: (1)A **focus group** is a small, but diverse demographically, group of people whose reactions are studied in guided or open discussions: ~people are asked about their perceptions, opinions, beliefs, and attitudes towards, ~RE Analyst records vital points he or she is getting from the group. (2) A **facilitated workshop** involves cross-functional team members to study the topic from all perspectives and decide as the outcome of the workshop.
12.7 Mind mapping, group storytelling, user stories: (1) Mind map (2) Group storytelling: Co-construct a story, share a story, complete an unfinished story provided by a facilitator, zoom-in/out or summarize a story, roleplay a story, analyze a story. (3)User story: E.g., As a manager, I want to browse my existing quizzes so I can recall what I have in place and figure out if I can just reuse or update an existing quiz for the position I need now.

13.Elicitation Technique Selection: (1) **Interviewing**: particularly useful to gather initial background information when working on new projects in new domains. (2) **Collaborative Sessions** (**workshop, focus group, brainstorming**): Effective. (3) **Data Gathering from Existing Systems**: must do but not over-analyze. (4) **Agile** (**mind mapping, group storytelling, user stories**): popular nowadays. (5) **Questionnaires**: Not effective.

14.Specifying requirements, structuring a set of requirements: (1) While we collect the requirements, we should structure them: ~Hierarchical structure: Higher-level requirements are decomposed into lower-level requirements, ~Link requirements to specific stakeholders (e.g., management and end users each have their own set). (2) So, while we structure the requirements, we also analyze them and their relationships.

15.Validating Requirement-Direct versus indirect links: (1)Same requirements from multiple sources help us reason the validity of the requirements: ~Lesson 1: don't rely too much on indirect links (intermediaries, surrogate users). ~Lesson 2: The more links, the better (but up to a point).

16.Validating Requirement-Things to look at: (1) Inspection of the requirement specification with respect to correctness, completeness, consistency, accuracy, readability, and testability. (2) some aids at different stages of development: ~Early structured walkthroughs with customers. ~Prototypes of initial versions. ~Test plan or unit testing in Agile development. ~User acceptance testing when delivery.

17.Negotiating Requirements-Prioritizing Requirements: (1) Apart from negotiating requirements during requirement elicitation, we should also negotiate requirements with stakeholders when prioritizing them. (2) E.g., label a requirement item as "high", "medium", or "low" priority. (3) E.g., Classify it using the MoSCoW Method: ~**Must haves**: mandatory requirements, ~**Should haves**: important but not mandatory, ~**Could haves**: if time allows, ~**Won't haves**: not today(may be tomorrow)

18.Summary: (1) RE involves elicitation, specification, validation, and negotiation. (2) Elicitation has five iterative steps, four of which are preparation steps. We use multiple elicitation techniques. (3) We not only collect the requirements but also shape them: ~Analyzing requirements by structuring them, ~Negotiating with stakeholders in requirement elicitation and prioritization, ~Validating requirements from multiple sources.

Software Architecture

1.The Styles of Program Structure: (1) At a higher level, we refer to the software architecture (or system architecture), enterprise patterns, architectural patterns: The *big picture* of how a software application is organized. (2) At a lower level, we refer to coding idiom, coding principle, design patterns.

2.Software Architecture: (1) The basis is Modularity: ~style to decompose a module into sub-modules, ~The layering style in the OSI model in Computer Networks. (2) Our purpose is to use styles to address non-functional needs. (3) E.g., How to deal with "Fast start-up time" [non-function require.] when initializing software [functional req.]?: ~Separate the UI code from the program state initialization code: Use the Model-View-Controller pattern to decompose a given module into three lower-level module (Model, View, Controller), ~Initialize the part of the program state that affects the UI first: Divide the Model module into a UI-related sub-module, a UI-unrelated sub-module, and a submodule to maintain the model integrity, ~Initialize & transit a lower-quality, faster-initialization core to higher-quality, full core: The UI-related sub-module should support extensibility and compatibility between sub-models. Controller should interoperate between models without affecting the View module.

3.The Strategy We Observe: (1) Recursively decompose a module into *interacting* sub-modules. Assign **functional responsibility** to sub-modules to meet "picked" functional requirement: E.g., submodules for different models and model integrity. (2) Assign **non-functional quality attributes** to submodels: E.g., usability by View, extensibility by sub-models, interoperability by Controller. (3) Assign **non-functional constraints** on sub-modules: Interoperating different models by the Controller should not affect the View module. We also Observe: (1)In architecting, we **neither** work out **nor** decide following: (2) How the functional requirement will be implemented: don't bother about algorithms for software initialization or maintaining the model integrity. (3) How the non-functional requirement will be implemented: We only assign the quality attributes and non-functional constraints with explanation/ documentation to explain why they are necessary for the implementation to fulfill the non-functional requirement.

4. Definition of the Software Architecture is: (1) Architecture is **fundamental concepts or properties** of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution. Evolution = maintenance. (2) is *conceptual*. (3) about *fundamental* (important) things. (3) exists in some *context*.

5.Architect: (1)Responsible for all technical high-level design decisions for a system: ~High-level = the architectural level, ~All decisions made now or deferred to a later time, ~Lead the system to evolve in the right direction. (2) Know the business impact of the technical decision. (3) Embrace, anticipate and manage changes (and their risks). (4) As promotor and communicator of project vision among stakeholders (including developers).

6.Role of Software Architecture:
6.1 Pre-architecture life cycle: (1) Iteration mainly on **functional requirements**. (2) No balancing between functional and quality requirements.
6.2 Architecture in life cycle: (1)Iteration on both **functional & quality requirements**. (2) Balancing of functional and quality requirements.
6.3 Why is Architecture Important? (1) Ar. is vehicle for stakeholder communication. (2) Ar. manifests the earliest set of design decisions: ~Constraints on implementation, ~Dictates organizational structure, ~Inhibits or enable quality attributes.

7.Driver of Software Architecture:
7.1 Architectural Driver: (1) **Architectural driver** is a **design requirement** that will influence the software architects' **early design decisions**. (2) Note: ~Functional features (aka functionality) are software requirements, not sufficient to serve as the architectural drivers. ~Architecture deals with non-functional requirements (e.g., solutions with different quality to achieve the same functionality).
7.2 From **Traditional Payment System** to **FPS**: (1) **Keeping**: reliable, easy to use by merchants & customers. (2) **Improvement**: T+2 => real-time and always-on. (3) **Generalization**: single platform => cross-platform. (4) **Extension**: easy to add new features, single standard for QR code. (5) So, even if QR code is banned as a feature (thus, both traditional payment system and FPS has no difference in functional requirement), FPS needs to address other quality attributes.

8.Quality Attributes and Patterns/Tactics:
8.1 Functional Requirements: specify what the software system needs to do to satisfy fundamental reasons for system's existence.
8.2 Software Architecture & Quality: (1) The notion of **quality** is central in software architecting. (2) Some qualities are observable via execution: performance, security, availability, functionality, usability. (3) Some others are not observable via execution: modifiability, portability, reusability, integrability, testability. (4) Quality also affects different aspects of software projects.
8.3 Quality Attribute: (1) A quality attribute is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders.
8.4 Measurable/Testable: (1) Without know how far the current system is from a goal, designing a system to meet this goal is impossible: ~[non-measurable]As customer, I can see the prices of all [measurable] cryptocurrencies quickly [non-measurable]. ~[measurable] as customer, I can see latest and 4 historic prices of 20 cryptocurrencies I select in a second. (2) As an analyst, we should aim to get the measurable one.
8.5 QA Tradeoff: (1) ~Improving privacy => lowering usability ~Higher performance => lower interoperability ~Higher modularity => longer time-to-market ~High reliability => lower performance ~Quicker time-to-market and lower cost => lower stability. (2) We should be aware that addressing one quality attribute by a *system decomposition* should evaluate its impact on some other quality attributes (within the domain model and among the quality attributes of the requirements) Exemplified Design Quality: Feasibility/Reusability/Stability/Simplicity/ Consistency/Composability/Deploy-ability.
8.6 Operation Quality: *Capacity/Usability/ Perform/Scalability/Serviceability/Visibility.*
8.7 Failure-Related Quality: (1) Reliability: ~Mean time between failures (MTBF) ~Mean time to failure (MTTF) (2) Recover: MTTR: Mean time to recovery, mean time to repair, or mean time to respond. (3) (Functional) Availability: Can be calculated based on reliability and recoverability. E.g., Availability = MTTF/ (MTTF + MTTR).
8.8 Security-related Q.: (1) *Authentication*: to confirm user identity. (2) *Authorization*: to restrict access. (3) *Confidentiality*: to avoid unauthorized data access.(4) *Integrity*: to protect the system/data from tampering (5) *Defensibility*: to protect against attacks (6) *Privacy*: to secure confidential data.
8.9 Change-related Q.: (1) *Configurability* (2) *Customizability* (3) *Duration* (4)*Feature Evolution*. (4) *Compatibility* (5) *Portability* (6) *Interoperability* (7) *Integration*.
8.10 Project Health Quality: (1) Data persistence, checkpoint, and recovery. (2) Backup and disaster recovery plan. (3) Maintainability.
8.11 API Design Principles: (1) Explicit interfaces (2) Principle of least surprise (3) Small interfaces (4) Uniform access (5) Few interfaces (6) Clear interfaces

9.Attribute-Driven Design (ADD): ADD is an architectural design methodology to tackle quality attributes incrementally. (2) Choose a system module/connector to decompose (3) Refine this module: ~choose architectural drivers (quality is the driving force) ~choose patterns/tactics/styles that satisfy the drivers ~apply patterns/tactics/ styles and assign responsibilities (4) Repeat.
9.1Architectural Tactics:(1)ADD iteratively applies one or more architectural tactics to address a quality attribute issue. (2) Architectural tactics are concepts. They are usually used implicitly in a solution. (3) The most often mentioned tactics nowadays are Security tactics (defense approaches). (4) Including Availability and Modifiability, Performance and Security, Testability and Usability. (5) Apart from addressing quality attributes by components, we can also address the quality attributes by how we connect these components.

10.Uses of design decisions: (1) Identify key design decisions for a stakeholder. (2) Evaluate impacts. (3) Make the design decision explicit. Representation: While we make design decision, we need to manifest design concept as **explicit artifacts** to communicate with stakeholders.

11.Architecture presentations in practice: (1) PowerPoint slides for managers, users, consultants, etc. (2) UML diagrams for technician. As a result, these representations are both descriptive and prescriptive. The 4+1 view model in UML: (1) Logical view: Class diagram, sequence diagram, state diagrams. (2) Implementation/ Development view: Component diagram, package diagram. (3) Process view: Activity diagram. (4) Deployment/Physical view: Deployment diagram. (5) Scenario: Use case diagram.

12.Architectural Styles: (1) An architectural style is a description of **component and connector types** and a pattern of their runtime control and/or data transfer. (2) Some basic kinds of architecture style: Client-server/Publish-subscribe/Pipes and filters/Blackboard/Layered/Microkernel. Model-View-Controller: (1)used to improve user experience. (2) **model**: represent the core functionality and data. (3) **view**: displays a part of the model to the user. (4) **controller**: accept input & update models.

13.Summary: (1) Software architecture exists within a context (the quality attributes when delivering the functional requires in the deployment time). (2) We divide the system into modules and their connections by styles to address quality attributes (e.g., through ADD). (3)Many architectural styles: The successful ones are summarized as patterns. (4) An architectural description includes the stakeholders' concerns and the rationales of the architecture chosen. They are presented in form of views & viewpoints.

Technical Debt

1.Technical debt (TD): (1) A metaphor is that doing things in a "quick and dirty" way creates a technical debt. Like a financial debt, TD incurs interest payments, which come in the form of the extra effort that developers must dedicate in future development because of this quick and dirty design choice. (2) is a design or construction approach that is expedient in the short term but that creates a technical context in which **the same work will cost more to do later than it would cost to do now** (including increased cost over time). Process: TD incur, then pay interest, finally pay back. Counterexamples of TD: (1) Many kinds of delayed/incomplete work are not technical debt: feature backlog, deferred features, cut features. (2) In general, if "the same work will not cost more to do later than it would cost to do now", then it isn't a technical debt. When TD is present in the software, **only significantly effective way of reducing TD is to refactor the software**. Impacts of Technical Debts: (1) In some large organizations, **development time** dedicated to managing TD is **substantial**: an average of 25% of overall development. TD and its basic management: (1) TD have been classified by dimension: ~**Type** (e.g., design, testing, or documentation debt), ~**Intentionality**(or unintentionally), ~**Time** horizon (short or long term), ~**Degree of focus**. (2) (Popular) TD management idea: **Identify** a technical debt and **track** it through product backlog of the software project. **Discuss** them.

2.Identify Technical Debts: There are some well-known patterns in sw. development to increase technical debts: =>
Schedule Pressure: (1) When a team is held to a commitment that is unreasonable, they are bound to cut corners to meet managers' expectations: E.g., creeping scope of new features, change in team composition, late integration. (2) Solution: Use more flexible planning approach.
Duplication of Code: (1) Many reasons for code duplications: ~Lack of experience on part of the team members, ~Copy-and-paste programming practice, ~Conform to the poor design of existing software, ~Pressure to deliver, guess a schedule. (2) Solution: ~Use static analysis tool for assistance, ~Pair programming: spread the knowledge and improve competence of team members, ~Either repay debts now or track it if repay later: fix it now or add runtime exception to location of technical debts and fix it a few minutes later or add the debt (location, description, potential cost of not fixing it) to the project backlog.
Get it "right" the first time: (1) Opposite of duplication – create a lot of powerful design up-front: We have over-engineered & over-generalized our intended product. (2) Costs associated with *fixing keep growing*.

3.Manage Technical Debts: (1) If we can't avoid technical debt, we must manage it. (2) This means recognizing it, tracking it, making reasoned decisions about it, and preventing its worst consequences. Methods to Manage Technical Debts: (1) Tracking Technical Debts (Visually) (2) Repay Technical Debts (3) Discuss TD

Software Maintenance

1.Purposes of S/W Maintenance: *Purposes* of performing maintenance are to: (1) provide continuity of service, keep it operational: fix bugs, recover from failures, respond to environments. (2) support mandatory upgrade: ~respond to regulation changes, ~maintain competitive edge. (3) support user requests for improvement: customize, enhance functionality, improve performance. (4) facilitate future maintenance work: e.g., perform code restructuring, documentation update.

2.Characteristics of the Maintenance Phase: (1)[People] Staff expertise: Most essential during initial development and evolution (can be AI-assisted). (2)[Product] Software architecture: ~Most stable at the time of the first release version, ~Gradually degrades due to restructuring for change. (3)[Process] Software decay: ~Quality of design and code deteriorates, docs outdated, ~Ad hoc fixes avoid major changes but less maintainable, ~Requires increasingly more expertise to maintain. (4)[Cost] Economics: ~Heavy initial investment, gaining benefit up to a max; then trail off, forcing phaseout and closedown, ~To regain benefits, cycle repeats with next development.

3.Classification of Modification Requests: (1) **Correction**: modification to correct a discovered/potential pro. (2) **Enhancement**: modification to satisfy a new requirement. (3) Trend: efforts shift from Enhancement to Correction.

4.Four Type of Software Maintenance: (1) **Adaptive Maintenance**: Modify the sw. to ensure it continues to operate in a new or changed environment, such as when there are updates to operating system, hardware, or software dependencies. (2) **Perfective Maintenance**: ~Enhance the software by adding new features or requirements. It aims to improve the software's functionality and performance based on user feedback or new business needs. ~E.g., for meeting the requirements in a sprint backlog. (3) **Corrective Maintenance**: ~Fix bugs and errors that are discovered *after the software has been deployed*. ~Sometimes, to handle unexpected failures or critical issues that need immediate attention, which involves prompt fixing the software to restore its functionality and minimizing downtime. (4) **Preventive Maintenance**: Make changes to the software to prevent potential future problems. It aims to improve the software's maintainability and reliability by addressing issues before they become significant problems. (e.g., addressing technical debts)

5.Processes of the above four types are: (1) Adaptive Maintenance: **Identify changes in the system environment, analyze the impact on the software**, plan and design modifications, implement changes, test the updated software, and deploy the changes. (2) Perfective Maintenance: **Gather user feedback to identify areas of improve**, elicit new requirements, analyze and prioritize enhancements, design and implement new features or improvements, test the enhancements, and deploy them. (3) Corrective Maintenance: **Typical** bug fixing process. (4) Preventive Maintenance: Consider regular code reviews and **system audits**, **identify potential future issues**, improve maintainability, **test preventive measures**.

6.How to lower the S/W Maintenance Cost: (1) Apply CI/CD (DevOps). (2) Maintain the software maintenance regularly. (3) Thieve for good software development. (4) Use automation tools: e.g., auto-notification, auto-test. (5) Use a bug tracking system. (6) Use modern (reliable) frameworks. (7) Optimize software license. (8) Security update, version control, apply change management.

7.The ISO/IEEE Maintenance Activities: (1) Process Implement: ~Develop maintenance plans and procedures. ~Establish MR/PR procedures. ~Invoke the Configuration Management Process. (2) Problem and Modification Analysis: ~Analyze MR/PR to determine impact and priority. ~Replicate or verify the problem. ~Develop options for implement of modification. ~Document MR/PR, analysis results, implementation options. ~Obtain approval for the selected modification option. (3) Modification Implement: ~Conduct analysis, identify the units and docs to be modified. ~Invoke the Development Process (a mini-SDLC). (4) Maintenance Review/Accept: ~Conduct review to ensure integrity of modified system. ~Obtain approval for satisfactory completion of changes. (5) Migration: ~Identify modified software/data; Develop migration plan. ~Notify users; Provide training; Notify for completion. ~Post-hoc review: to assess impact of environment. ~Archive data (and old software). (6) Retirement: ~Develop retirement plan. ~Notify users; Implement parallel operation and training; Notify for completion. ~Archive data (and old software).

8.Stages of Maintenance: Within the maintenance phase, there are several stages: (1) **Initial development**: first release plus accompanying activities (training, etc.). (2) **Evolution**: major extension of the system's capabilities. (3) **Servicing**: ~in operation with only minor repairs or simple changes. ~work is stable, mature, well understood. (4) **Phaseout**: services discontinued, though system still in operation. (5) **Closedown**: system withdrawn; users directed to replacement if any.

Code with Quality

1.Coding: (1) In Agile method, developers sketch the software design, write codes incrementally, and focus on delivering the tasks we pay attention to: ~We unavoidably make mistakes during this "no-long-term-planning" (i.e., unsystematic) process. ~Our code is (un)intentionally coupled with the customized nature of these tasks. (2) The design and code will likely change as developers' understanding of the system to build is deepened. (3) The basic strategy is: Our code should be clean, minimal, and maintainable.

2.Basic Code Design Principles: There are a few basic design principles recommended to developers when they write code. We look at the core set of these design principle: (1)Abstract/Information Hiding/Coupling/ Encapsulation/Cohesion. (2) S - Single-responsibility principle/O – Open-closed principle/L – Liskov substitution principle/ I – Interface segregation principle/D – Dependency Inversion Principle.

3.Abstraction: (1) **Abstraction** is a fundamental concept in programming theory. It aims to avoid duplication of information and/or human effort. (2) Functional abstraction is to abstract the procedural steps in code. (3) If we abstract and group functions **critically related to** a data structure into a set, the set represents a **single functional purpose** of the data structure: ~Function with functional abstraction = operators on the data structure. ~functional signatures = interfaces. (4) **Data Abstraction** is to model the operators on a data structure as a set of related functions (called **interfaces**). In contrast, the data structure implementations are concealed by the interfaces.

4.Information Hiding and Encapsulation: (1) In mathematics, we have the concept of Projection. (2) Interfaces should be chosen to reveal **as little as possible about** its **internal workings**. (3) We should ask whether the level of details specified in the parameters of an operator in an interface is essential. (4) Encapsulation on X refers to **ensuring** outsiders cannot **gain knowledge** about certain information internal to X, X can be a class, a module, a file, a data record and also represent internal data, algorithms. (5) **Idea**: We achieve good encapsulation if we can describe the functional purpose using the least "working known-how" of the operators of a data structure and the functional signature does not specify more than what the functional purpose states. (6) **Abstraction**: separate interface from implementation with a purpose. (7) **Information hiding** remove details from interfaces of data structure (by interrelating these interfaces). (8)**Encapsulation**: restrict gaining knowledge about implementation details of the code to its client code.

5.Coupling and Cohesion: (1) **Coupling** is about syntactic dependency: our aim is to minimize coupling in the code listing. (2) **Cohesion** is about the reasons to place a set of entities together as a group or split them: ~*We cannot see dependency on cohesion from the code*, ~Our aim is: if two entities are related, they should be grouped together and/or used together. (3) Consequence: High cohesion and lower coupling together lead to higher quality: maintainability, reusability and lower complexity.
Six types of coupling: (1) Content (directly use a part of code) (2) Common (global variable) (3) External (interface protocol, file) (4) Control (control flow, workflow pipeline) (5) Stamp (a part of a data structure) (6) Data (the whole data structure)
7 types of cohesion: (1) Coincidental: grouped without a reason (2) Logical: grouped because "we" treat them as one category (3) Temporal: A part of each entity in the group is processed similarly. (4) Procedural: A group of functions, when composed, represent different sequences of control of abstraction (5) Communicational: grouped because of operating on the same data (6) Sequential: input of one depends on the output of another (7) Functional: to perform different aspects of a well-defined task. From (1) to (7) is worse to better.

6.SOLID in Object-Orientation: (1) A, IH and Encapsulation are fundamental. (2) There are also many other more pragmatic guidelines to help developers to organize their code better. One of the most popular ones is categorized as SOLID.
Summary-Fundamental concepts: (1) **SRP**: one purpose for making change to a class. (2) **OCP**: extending behavior without changing the source code. (3) **LSP**: a subclass can do whatever a superclass does logically. (4) **ISP**: do not force a client code to implement unwanted methods. (5) **DIP**: a higher level entity does not depend on a lower lever entity.

7.Code Smells [also known as anti-patterns]: (1) Many possible metrics:~Inefficient code in general, ~Code when input parameters in outliner range will result in performance bloat, ~Risky code, difficult to make modification, ~Violation of design principle. (2) **Code smell** is a violation of a design principle and (potentially) harms the code. (3) **Code Review** is a good source for the team to manage the code. (4) Some **static analysis** tools can help detect code smells. (5) Code smell is a kind of TD.

8.Refactoring: (1)To improve code structure and organization (2) Triggering points: code review and in code maintenance. (3) Every good IDE (e.g., Eclipse, Visual Studio) has a menu of popular refactoring to assist developers to manage their code. (4) Result: perform manually, improve code readability and maintainability, worry about regression bugs and build breaks.

9.Summary: (1) Maintaining code structure is important. (2) Accumulating problems in code will lead to have more code changes, more latent bugs, and less comprehensible code "in the future": Manage the amount of code smells in code. (3) Refactor the code to improve readability and maintainability, either manually or with tool assistance, if developers are confident about validation of changes and able to handle code changes across different branches in version control system: Note that removing all of them is never an option.

Program Testing

1.What is Program Testing: Conduct validation on a program to find bugs or demonstrate the program working under certain scenarios (acceptance test).
General Process Step: (Step 1) Defining Test Objectives=> (Step 2)Test Design=> (S3)Test Execution=>(S4)Test Analysis.
Step 1: Define the overall **goal** for designing and executing a test and the type of tests to be conducted.
Step 2: (1) **Goal**: To get, clarify, and concretize user requirements by examples. (2) **Implement** test code (or define test profiles for fuzzing).
Step 3: (1) Execute the test cases written in the Test Design step. (2) E.g., ~Run the Junit test scripts ~Run in Continuous Integration.
Step 4: (1) **Identify failures and anomalies** from the test results. (2) If we want to automate Step 2 or Step 3, we shall automate this step as well.
Continuous Integration: (1) Use a single shared repository (CI server) to host and integrate the main branch of code (baseline) frequently. (2) Developers' practices: ~Before pulling into the CI server, conduct the unit test locally. ~Make commits every day (to synchronize the common baseline). ~Enhance the test scripts to validate every bug-fixing commit. (3) Every commit in the CI server triggers an automated build in an integration environment: ~We need a build script and unit test scripts. ~The build should include self-testing (run tests by itself). (4) Test in a clone of the production environment (known as staging) after validating the committed code at the unit test level. (5) Continuous Delivery (CD) extend CI with auto deployment.
In a Larger Context: (1) Continuous delivery/deployment (CD): is a process to deploy software into production iteratively. (2) Key characteristics: ~Software is updated in *small increments* that are *independently deployable*. ~Fully deploy them immediately after development and testing are completed. ~The decision to deploy is largely left up to the developers. ~Developers responsible for development, deployment, and testing can be notified immediately when system failure in production occurs. (3) Key practice: ~Implement automated unit- and sub-system tests. ~Must use automated testing tools to enable **early and frequent** testing. ~Incorporate code review. ~Configure auto-deployment tool. ~Deploy updates by stage.
Some Characteristics of Effective Testing in Uncertain User Need: In Agile development,

requires are often not defined in detail, and without strict processes to elicit them, developers need an understanding of the business and experience in domain model: (1)**Require. testing**(via system testing): demonstrate the (vague and incomplete) requirement rather than finding bugs in the design and implementation. (2) **Test coverage**: knowing all possible usage scenarios is infeasible. Paying *more attention to test features with larger user bases in the next release of the program*. Judge by developers on whether the feature has been *sufficiently tested for the users to make effective use of the feature*. (3) **Test automation**: DO NOT replace manual testing. However, it is a good way to show users that the produced code is testable to ***educate users*** to raise checkable requires and support feature implementation approval in a testable way.

Pair Programming: (1) Pair programming is a practice used in agile development. (2) Developers X and Y write the same piece of code simultaneously. (3) ≠ X writes the code, and Y reviews the code after code completion. (4) ≠ X independently completes part A, and Y independently completes part B, and then the two parts are integrated. (5) = Y reviews the code being written **while** X writes each line; X and Y may switch their roles during the process.

Note: it's almost impossible to guess out the return value logic by developers themselves. We call such a bug an **omission fault**.

How to Spot Omission Faults: (1) Conduct Code Review (2) Follow the guidelines and checklist of the company (3) Always ask users or client developers of the functions whether something should be returned apart from the text outputs (4) Do not aim to validate the written code only. Also, check against user intention or requirements (5) Reference the code serving similar purposes if available (6) Use automated testing tools and static analysis tools to guard against generic errors (e.g., security vulnerability) (7) Ask Large Language Model.

2.Fuzzing: Automated method to generate test input, execution tests and expose security vulnerability (crashes).

(1) Fuzzing is the process of finding vulnerabilities code by repeatedly testing code with modified inputs. It's becoming standard in the commercial software development process.

Monkey: (1) The Monkey tool emulates user interaction with an app. It generates and injects gestures commands or system events based on predefined event frequency distribution into app's event input stream. (2) The tool watches for exceptional conditions, e.g., app crashing or throwing an exception. (3) The tool can vary the "throttle", i.e., the time delays between events; by default, there is no delay between events (throttle=0) (4) **Manual testing has better code coverage than Monkey**. (5) **Setting different time delays is insignificant to improve code coverage**. (6) Instead of command-line monkey, one can write code to interact with an API version of monkey and thus produce one's own (simple) fuzz testing tool.

American Fuzzy Lop (AFL): (1) **Feedback-based**: run, get info from previous runs in guiding the selection of the next input. (2) **Mutation-based**: change inputs to become new inputs. (3) **Grey-box**: need both input seeds and code coverage.

The basic strategy of AFL: (1) Start with a small set of input sample files as a queue of inputs: The test cases for user stories (in Scrum) can be these seed files. (2) Mutate "randomly" an input file from the queue. (3) If the mutated file reaches new or simpler path(s) (in log2 sense), add it to the queue.

Fuzzing finds crashes or ANR (App Not Responding), so-called code vulnerabilities. In most cases, developers start debugging based on the failing test cases.

If an input is long (e.g., crash a computer game by fuzzing), the debugging will be tedious and slow. More desirable to find a smaller input to trigger the same bug before debugging.

3.Scientific Debugging:

Distinguishing Failures from Bugs: (1) A *failure* is an *incorrect output* of a program. (2) A ***bug (or fault)*** is *an incorrect implementation (coding)* that causes the program to produce failures.

How to Debug – Three steps:(1) Locate Bug (2) Repair Program (3) Retest Program.

A model of program execution: (1) A program *P* is a set of ***statements*** and a set of ***variables***: ~Each variable keeps a value. ~Each statement may read a value from one or more variables and/or write a value to a variable. ~Outputting value from a program means to write a value to a (system-specific) variable. (2) A ***program state*** is a set of key-value tuples, each containing a variable *v* as the key and the value of the variable *v*. (3) An execution trace is sequence of executed instructions to represent the execution of a test case: ~Note that the same program statement may occur in the same trace multiple times (and so we have first occurrence, second occurrence, etc.) ~Thus, by walking through the statements in an execution trace σ from start to end and one by one, the program state of the execution trace σ is updated while we walk through the trace. Failure and Bug: (1) Failure: ~A ***failure*** of a program P is an incorrect program output of an execution trace σ of the program P. ~We mark the statements that produce the failure on the trace σ. (2) Run: ~A ***failing run*** refers to an execution trace that is marked with failure(s). ~A ***passed run*** refers to an execution trace that is not marked with my failure. (3) ~A ***bug*** (also known as a ***fault***) is a *mistake* appearing in a program *P*. ~E.g., In an execution trace σ, a bug is marked as an incorrect single statement or an incorrect non-empty subsequence of statement in it. Thus, the positions of a bug can be marked in the execution trace σ. Note that some execution traces do not contain any bug.

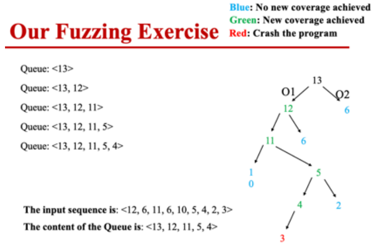
Propagation from Bug to Failure: (1) To observe a failure from a run, a bug should ***propagate*** its ***error*** along the trace from a buggy statement (say at position σ[b]) to the statement outputting a failure (say at σ[f]). (2) The buggy statement σ[*b*] transforms a correct program state before executing this buggy statement into an ***infected*** program state E_b. (3) If there is any difference between the expected program state and an infected program state, the infected program state incurs an ***error.*** (4) From the position of this infected program state E_b, the subsequent instructions (at positions σ[*b+1*], σ[*b+2*], …., σ[*f*]) in the trace σ will further transform E_b into a sequence of infected program states E_1, E_2, …., E_f . When the statement at the position σ[*f*] is executed, the corresponding error in the infected program state E_f will be observed as a failure of the execution trace σ.

The Procedure of Scientific Method: (1) **Observe** some aspect of *the thing*. (2) Formulate a ***hypothesis*** consistent with the observation in step 1. (3) Use the hypothesis in step 2 to make ***new predictions***. (4) Tests the predictions by **new experiments**. (5) **Repeat** 2 to 4 to refine the hypothesis in step 2. Notice the order of steps 2 and 3 ahead of step 4. Do not swap them.

Reminder: (1) The scientific method is **tedious to apply**, but it always works: So, if we have exhausted our own ideas to debug our programs but still failed, try it. (2) A key merit of the scientific method is that we not only locate the bug, but also produce a code fix (i.e., a patch of the code to correct the program!)

Delta Debugging: Input Reduction or Code Reduction. Input File Reduction: (1) In the process, ensure we can still crash the web browser by printing the page. (2) Remove single character one at a time: 40k tests. (3) More sensible strategy: ~First: cut away large parts. ~Then: iteratively cut out the other parts bit by bit.

Reduction **Strategy**: Try by reducing firstly by half (and if not work) then randomly.



The input sequence is: <12, 6, 11, 6, 10, 5, 4, 2, 3>
The content of the Queue is: <13, 12, 11, 5, 4>

Fuzzing: (1) can generate system-level test cases as they generate inputs accepted by UI or main() as arguments. (2) could not help developers when the integration and system testing have not been performed.

Concurrency Testing: (1) Programs can be multi-threaded and/or multi-process architecture. (2) These programs are nondeterministic: Running the same test case multiple times may give different outputs. (3) Strategy 1: Run the same test case many times without controlling how program threads are coordinated in each test execution: Ineffective, if you have no other choices, just using it is better than none. (4) Strategy 2: Run the same test cases multiple times with the use of a controller to control when threads execute their program statements relative to one another. (5) Concurrency testing: Can check individual program executions faster, can suppress false positives, limited in ability, false negative: Thread sanitizer.

Static Analysis: (1) Static analysis is to inspect the code without running it. (2) Many modern IDEs have incorporated static analysis to spot potential concurrency issues. (3) Static analysis is a routine task in large projects. (4) Static analysis: Can check the whole codebase, applicable to check Library/Framework, slow, non-scalable, many false positives. (5) The mathematics are quite extensive.

Automated Test Case Generator: (1) The target is to produce some useful test cases (but without proper check on test outcomes). (2) Idea in Evosuite: ~Given a set of Junit test methods, each of which contains a sequence of statements. ~Like fuzzing, apply a series of mutation operators to modify some statements (e.g., exchange statements across test methods, delete some statements, add some statement, change parameter value) to generate a new set of Junit test methods. ~No idea on whether a test method detects a program failure.

Test Oracle Problem: (1) Recalling from EvoSuite's and fuzzing's limitations that they could not point out functional failures of the program being written: Sometimes, we simply cannot get it: how do you know a Deep Learning engine produces a correct classification? (2) We need to formulate either a *correctness criterion* or *an error condition* for test cases to know whether the program actually passes these test cases. (3) This is called the **test oracle problem**.

Generic Test Oracle [automated]: There are some errors that we know them being errors. These errors are general and applicable to all types of situations and programs (E.g., crash, deadlock): They are widely used in automated testing tools.

Pattern-based Test Oracle [automated]: Sometimes, some code patterns in the code tend to be erroneous. They should be removed from the code if possible. They are formulated heuristically. ~Many static analysis tools use them. ~E.g., FindBugs for Java, pylint for Python, safety for checking requirement.txt for Python.

Reference Test Oracle [automated]: For programs with clear specialized purposes, one knows the expected test results and test cases. For instance, to test for browser compatibility, we can run the predefine benchmark test suite available on the web.

Regression Test Oracle [automated]: For programs with versioning, many test cases will be applicable to multiple program versions without revision. In these cases, the test oracle or output of a program version on a test case can be used to check output of another program version on same test case.

Program-Specific Test Oracle [manual]: (1) Each program has its own purpose and functionality. There is no universally applicable test oracle to verify functional outputs fitting every program. (2) In a vast majority of cases, developers write their own Assertion Statements (assertEqual() in JUnit) or validate the outputs manually. (3) If developers have some knowledge about the application domain of a program, they can use their "common sense" to spot failures from testing.

Test Oracle across Multiple Executions: (1) Suppose developers have some knowledge about the application domain of a program. (2) They will know something could be wrong by comparing multiple test results and the expectations on the relations on test results. (3) It is known as **Metamorphic Testing** [automated]. (4) Is experimentally used in testing Autonomous Driving System. (5) Is used in startup.

Idea of Metamorphic Testing: (1) Consider a deterministic program P that finds a shortest path from a directed graph G1 from node x to node y. (2) Test case P(G1, x, y) outputs a sequence L1 of nodes in G1 to go from x to y. (3) **Test case part:** By deleting some nodes (and their connecting edges) in G1 where each of them is not in L1, we construct another graph G2. (4) **Test case part:** Test case P(G2, x, y) outputs L2. (5) **Oracle part:** Expectation: L1 = L2. If violated, at least one of the two test cases is a failed test case.

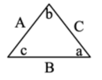
Metamorphic Relation: (1) An expected relation over multiple test inputs and test outputs to be held. (2) Such an expected relation is called a **metamorphic relation:** ~E.g., P(G1, x, y) = L1∧P(G2, x, y)= L2∧ L1 = L2. ~The test case {G1, x, y} is called a **source test case.** ~The test case {G2, x, y} is called a **follow-up test case.** (3) Often, the "Common Sense in the application domain." (4) Metamorphic testing can be applied beyond Equivalent Inputs (and thus Equivalent Outputs). (5) Developers who know application domains can formulate their own relations for their programs.

Simple Metamorphic Relations Could Be Effective to Detect Bugs: (1) A compiler P (e.g., the objective-c compiler) to generate the binary code B1 from a source code S1. (2) Suppose *t* is a test case of S1. (3) Compile S1, run B1(t), and then remove some statements in S1 that *t* does not pass through them to construct a new program S2, compile S2 to generate the binary code B2: Expectation: B1(t) and B2(t) produce the same output. (4) Thousands of bugs have

been detected from the *gcc* compiler.
Fuzzing + Metamorphic Testing: (1) An effective use of both techniques. (2) Fuzzing to generate one test case as a source test case for metamorphic test to follow up.
Fuzzing + Automated Test Oracle: (1) E.g., ContractFuzzer for testing Ethereum smart contracts.

Appendix: Possible Solution for Metamorphic Relations

- sin(x) = sin(180-x)
- sin(x) = -sin(x+180)
- sin(-x) = -sin(x)
- sin(x) < sin(x+y) if 0 < x < 90-y and y > 0
- |sin(x)*sin(x)| <= |sin(x)|



sin(a)/A = sin(b)/B = sin(c)/C

Modern Code Review

1.What is Code Review: (1) Performing human inspection on the source code by developers other than the author of the code. (2) a well-established software engineering practice. (3) aim at **maintaining and promoting** source **code quality**, as well as **sustaining** the development **community** by means of *knowledge transfer* of design and implementation solutions applied by others. (4) Apart from finding bugs, code review has **many purposes**: code improvement, alternative solutions, knowledge transfer, team building, and so on.

2.Different Levels of Formality: (1) Code walkthrough: ~informal meeting where the programmer leads the review team through his/her code, and the reviewers try to identify faults. ~More effective if the programmer and reviewer are not the same person. ~Sometimes, integrate with Pair Programming (2)Code Inspection: ~Formal meeting. ~Effective to find bugs in code. (3) Modern Code Review.

3.Code Walkthrough: (1) Similar to testing in effectiveness for finding bugs. (2) But code walkthrough requires more human effort: less effective. (3) In recent years, developers have become aware of technical debts (non-bug issues), where testing is ineffective in addressing them, E.g., readability, reusability, maintainability, etc. (4) The remote collaboration raised from the needs of "work from home" makes code walkthrough by visual impaired developers join code walkthroughs remotely.

4.Code Inspection: (1) Is a highly structured and formal process of review: ~Sometimes called ***static testing*** in company. ~Effective to find bugs and spot process improvements. ~Decreasingly popular. (2) Things to check: ~Efficiency of algorithm, ~Logics and correctness of code. ~Comments and consistency with code. ~Develop your own checklist. (3) Main Ideas: ~**Pre-meeting**: Authors educate reviewers, and reviewers read/question the code from a testing angle. ~**In-meeting**: Walk through the code (by code author) with reviewers and check against checklists. Answer questions and resolve ambiguity. Each reviewer has a predefined role assigned pre-meeting. ~**Post-meeting**: Authors follow up and rectify the code. The moderator verifies the required corrective actions taken properly. (4) Characteristics: Formal, synchronous, check code, tool support is secondary, heavy-weighted.

5.Modern Code Review (MCR): (1) MCR is a lightweight and tool-based code review of code change (not whole piece of code!): is the norm for many software development projects. (2) Characteristics of MCR: Informal; tool-based logistic; asynchronous; and focused on reviewing code changes; lighter weight than code inspection. (3) Main Ideas: ~Author A makes a patch P on a code block C to address some problems and sends the patch P via e-channel (e.g., email, WhatsApp, WeChat, or GitHub tool) to a set of reviewers R. ~Each reviewer R evaluates P, deems P good, requests change on P, or rejects P. ~Author A commits P; post-commit reviewer by other Rs possible. (4) Authors and reviewers exchange ideas, find bugs, and discuss alternative solutions to better design the structure of a submitted code change.

6.Characteristics of Modern Code Review: (1) Informal: Ad hoc group of reviewers without predefined roles, and the review process does not follow a formal procedure. (2) Tool-based logistics: ~Email, WhatsApp, and WeChat are popular. ~Open-source tool: GitHub, Gerrit (for Git), Review Board, Phabricator. (3) Asynchronous: Authors and reviewers need not engage in the same task simultaneously (but not extensively delayed) (4) Focus the **review** on the **patch** (code change, the delta).

7.MCR Best Practices: (1) peer review follows a lightweight, flexible process. (2) Reviews happen early (before a change is committed), in time, and frequently. (3) Reviewers have a prior knowledge of the context and the code to complete reviews quicker and provide valuable feedback to the authors. (4) Change sizes are small. (5) Small number of reviewers (1 to 5 reviewers) (6) Read informal posts and comments on the code and patch. (7) Review is not just to find bugs at group-level: problem solving (solution development), code readability and maintainability, following the norm (of the belonging company), accidence prevention, gate-keeping. (8) Reviewer candidates: Code owner/Developers who made previous changes on the code/ Developers who develop the code before (e.g., in producing the code through pair programming)/Experienced reviewers/ Developers who develop similar features of the changed fragments in some other code.

8.MCR Issues: (1) Review Quality & Code Size: ~**Reviews on code with higher complexity get fewer discussions and feedback**. ~**Code with fewer review feedback will encounter more post-release bugs**. (2) Distance: ~Causing delay in the review process or leading to misunderstanding. ~Aspect1: Geographical issue: Large physic distance between authors and reviewers. ~Aspect 2: Organization issue: Reviewers from different teams or taking different roles. (3) Social Interaction: ~Tone: comments with negative tones are less useful. ~Power: ask another person to change their comments. (4) Context: Misunderstanding due to mismatched expectations on the code change (e.g., nice to have vs. urgent fix, full solution vs. high level sketch).

Midterm solution

1.You goal is to explain how an agile process differs from the current process.
(1) The organization of development phases in the software development lifecycle.
~The current process is sequential; An agile process involves backward iterations.
~No deliverables are produced to the client before the user acceptance test in the current process. An agile process delivers a partial system by releasees and incrementally improves it.
(2) The involvement of customers.
~Current process: Customers have no knowledge of any intermediate task unless developers call for a meeting for that task.
~Agile process: Customers may be involved in backlog selection and prioritization and validate the implementation of a backlog as soon as it is ready for validation.
(3) Prioritization of tasks to perform.
~The project manager prioritizes and distributes the tasks in the current process.
~In an agile process, customers join the meetings and discuss the next upcoming target, affecting the prioritization of tasks.
(4) The duration period of a subsystem.
~In the current process, all subsystems are integrated followed by system testing.
~In an agile process, a subsystem is marked done if it is integrated into the current release of system after passing the system testing of the current release.
(5) Programmers' control on the tasks they perform.
~Current process: Programmers control the implementation of the tasks they are assigned to implement.
~Agile process: Programmers are self-motivated to pick tasks for implementation according to their abilities.

2.(a) Give one example of a measurable non-functional requirement and describe what changes to your example so that it becomes non-measurable. Explain your change.

- Example of measurable non-functional requirements. The marking scheme is given as follows:
  - the example is a non-functional requirement. (1′)
  - the example is measurable. (2′)
- Change made and explanation
  - the example becomes non-measurable. (1′)
  - explanation (keyword: quantitative). (1′)

- The system ... should be **scalable (non-functional requirement)** to serve **1000 (measurable)** ... sessions concurrently so that each session can complete its purchase within **1 minute (measurable)**.
- Change: The system ... should be **scalable (non-functional requirement)** to serve **many (non-measurable)** … sessions concurrently so that each session can complete its purchase **quickly (non-measurable)**.
- Explanation: The **quantitative** description is eliminated.

(b) Requirement elicitation has five basic types of activities. How do you use them to improve requirement elicitation activities described in the current software processes.
(1) Understanding the application domain: Before the series of meetings with the client:
~The project manager should seek the project background information, then;
~The project manager and system analyst should first study and understand the application domain.
(2) Identifying the source of requirements: During the meeting series, if there are any existing documents/materials about the project, the team should:
~Identify the source of these requirements,
~And collect their related requirements.
(3) Analyzing the stakeholders: If new sources of requirements are identified:
~The team should identify stakeholders, and;
~Meet with newly identified stakeholders, if possible, in the meeting series.
(4) Selecting the techniques, approaches, and tools to use: Apart from listening the requirements from the client, the team should use multiple elicitation techniques:
~For example: prototyping, storytelling, form analysis, focused group.
(5) Eliciting the requirements from stakeholders and other sources: To increase the reliability that a correct requirement and its related requirements are elicited:
~The team should aim to have the same requirements supported by multiple, preferably direct, sources/links.
~A confirmation of the requirement such as through paraphrasing the requirements heard, should be included in the meeting.

3.The following is a list of requirements:
(a) Each shop should be assigned with a unique beep sequence.
(b) Each shop should install a single speaker to emit its beep sequence repeatedly when the speaker turns on.
(c) A beep can be either short or long like the morse code. The length of the beep sequence for a shop should not be longer than 32 beeps, and the sound emitted by the speaker should be in the range suitable for the population of visually impaired persons.
(d) The app can hear a beep sequence from a speaker with an accuracy of 95% or higher from the microphone of a low-end mobile phone.
(e) The app can use the beeps heard to identify a shop with an accuracy of 95% or higher.
(f) Within 3 seconds after the app speaks out the shop name, if a user uses a figure to knock on the phone display three times with 1.5 seconds, the app will mark the user to enter the shop.
(g) The app will cache the mapping between the beep sequence heard in item (e) and the marked shop in item (f) for one month to speed up the next round of processing of the shop.
Question (a): Extract **four** quality attributes of the app from the requirement and paraphrase them into your own words. For each quality attribute, you should also state the suitability of Model, View, and Controller (MVC) architecture.

Extracting Quality Attributes and MVC Applicability (5 points for each example):
- Identify a quality attribute that is relevant to the application requirements. (1′)
- Precisely paraphrase the quality attribute, demonstrating understanding of its meaning. (2′)
- Provide an explanation of the applicability of the quality attribute in the MVC architecture, demonstrating understanding of the architecture. (2′)

- Quality Attribute: In requirement (d), the app should hear a beep sequence from a speaker with an accuracy of 95% or higher (Identification of quality attribute of the app), where the "accuracy of 95% or higher" is a quality requirement for the app to collect data from the speaker (Description of quality attribute).
- Suitability of MVC architecture: The beep sequence should be received by a microphone, which is a user interface to collect data (Description), and it is suitable for View in the MVC architecture (Suitability of the detailed part of the MVC architecture).

Question (b): The target user population are visually impaired persons. The person is difficult to know if the enhanced app is not responding or crashed. Sketch a design plan to design the app to address this issue.

- **Problem and Solution:** Because the target user is visually impaired and difficult to know if the enhanced app is not responding or crashed (specific problems), the app should help them to monitor and detect such faults once they occur, and then we could inform them accordingly (Design purpose). I select "heartbeat" for fault detection (General Solution).
- **Solution Design:** We could involve a "heartbeat" component in the app to exchange messages periodically between the system monitor and the app (Message Exchange Mechanism). It will stop exchanging messages once it does not respond or a crashed fault occurs in the enhanced app (Detection and Information Communication). Therefore, the system monitor could detect such faults and try to inform the problem to the user .

4.(b) A saying is that accumulating technical debts will slowly kill a project and even a company. Many people say that technical debts are unavoidable. Many people also say that a project does not resolve all technical debts before the application releases. Propose a cost-effective strategy to handle technical debts in a software project.
~Mention that using hybrid approach or cost-effective strategy to deal with technical debts.
~Talk about reasonable and flexible project schedule.
~Track the technical debts in the project backlog.
~Refactoring the project based on backlog from time to time.
(c) Give one example of technical debt in your course project, which should belong to Documentation, Intentionally, Short Term, and Unfocused.
~Reference answer: we only have enough time to focus on code implementations but don't have enough time to write any documentation or annotations for our code.