Intro

Software is intrinsically: complex, intangible, malleable, scalable and evolutionary.

Agile Team Organisation

What is Agile? Small teams, frequent structured meetings, small deliveries, customer engagement. Maintenance techniques like continual testing, analysis, refactoring. Change of project objectives and priorities is constant. Frequent review of software process.
Agile Methods: TDD, BDD, Planning Poker, Scrum, Kanban, CI/CD.
Risk of Agile: Lack of customer engagement, poor code quality, poor team coordination, stakeholder conflict.
Roles in Scrum: Product Owner, Scrum Master, Team Manager, Quality Assurance manager, Chief Architect, Developer.
Managing work in Scrum: Work is organisted into releases and sprints. Begin first sprint with a project launch meeting. Begin a sprint with a planning meeting, end with review meeting and retrospective. Stand-ups take place throughout the sprint to review progress.
Project launch meeting determines major features to deliver, identify a MVP, develop initial user stories, establish cost estimates and priorities.
Release planning meeting long projects have multiple releases, identify high level features, create a roadmap of milestones.
Sprint Planning Meetings: decide on a main goal for the sprint: features, bugs, refactors. Select tasks from the backlog.
Stand-ups are used for each member to reflect on progress, blockers, and next steps. Max 10 minutes.
Managing delays can be done by getting an extension or reducing feature set.
Review the project in the sprint review meeting and the process in the retrospective. Deliver and demonstrate new version to customer. summarise completed work, identify new feature set.

Change Management

Control items are any artifacts that a software engineer might directly edit.
Centralised version control system has a single centralised repository somewhere on a server. Each dev has a local copy. Changes are pushed to the central repo.
Distributed version control system has a repository on each machine. Each repo can be linked to other repos.
Commits are saved with a hash for uniquely identifying the changes.
Branching is used for maintaining multiple development lines. **Trunk-based development** invloves commiting to one main branch, useful for reducing merge conflicts, requires developers to make smaller changes, and provides clear visibility of project process. **Feature Branching** is more convenient for managing code reviews. **Staging branches** are used after making a commit to the main branch, some testing is done here then a push is made to the deployment branch. Manage branches by deleting them and squash commiting.

Customer Management

First customer meeting: decide on rules of engagement, clarify overall goals, identify stakeholders, determine appropriate IP ownership, identify any significant risks, decide on goals for first sprint.
Intellectual property and licensing is a complex issue. Both parties may contribute background IP.
Customer meetings: set an agenda, use time boxing, have roles.

Requirements Management

Cannot isolate the requirements from the implementation.
Actors: categories of users, motiviation.
Non-functional requirements: can be expressed as user stories.
Functional requirements: what the system does.
User stories: a short description of a feature from the user's perspective. Used to document the requirements for a software system.

Software Process Improvement

Process improvement frameworks: **ISO 9001**, **Six Sigma**, **CMMI**.
Goal is to arrive at the root cause when discussing challenges in ret-rospectives. These meetings, unfortunately, are infrequent by nature and issues early in the sprint may be hard to recall. It is important to vary the retrospective structure.
How this is done: gather data, analyse data, identify root causes, implement changes.

Code Reviews

Purpose is to detect defects, identify refactoring opportunities, develop a shared understanding of the codebase.
Merge requests should be small: 300 lines and/or require 30 minutes to review. Should choose one of: corrective, adaptive, preventative, perfective. A code review should adhere to architectural patterns and re-use existing code.

Build, Release and Dependency Management

A software project should have a **build configuration file**. This specifies targets (resolve dependencies, compile code, test binary), mappings (relationship between source and generated artifacts), tasks (actions to satisfy mappings, e.g. execute a specific compiler).
Types of dependencies: environmental, application
Types of releases: Core executable, Tailored executable, Optional extensions, Sources, Documentation (compositions). Bleeding edge/snapshot, Beta test release, Production release (schedule intent).
Types of APIs: Private, Public Published (APIs that can be externally access but not explicitly documented as being part of the public API).
Specifying dependencies: project almost always has transitive dependencies, do not rely on them.
Semantic versioning: major.minor.incremental[-tag]
Deprecated feature: Left in for compatibility, but intended to be removed in future releases.
Migration plans and scripts: adapting existing code to use new published APIs.

Continuous Integration

Integration hell: Spending more time re-integrating features, than in creating the features.
Continuous Integration Practices: change management, quality assurance, deployment.
Broken build: the highest-priority for a team; other operations must momentarily cease. **Build times**: should be less than 10 minutes.
Staging platform: Used to test software before being released to users. Limitations include lack of realism, too many simultaneous users, network endpoints and data sets inaccessible outside of production. Multiple staging environments may be needed when several components, each of which intended for use on a different platform, exist.

Static Analysis, Readability and Design Quality

Static vs Dynamic analysis: The former is applied on program artefacts at rest, while the latter is conducted during execution.
Fan-in (afferent coupling) complexity: Number of inbound references to a class from other classes. Identifies the number of classes that will need to be modified if the subjected class is changed.
Fan-out (efferent coupling) complexity: Number of outbound references for a given class. Determines the frequency that the class in question will need to change.
Inheritance depth and width: Deep inheritance suggests an over-abstraction of the class.

Behaviour-driven development

Given: To set up the test case fixture, initialize the necessary components and dependencies required for the test. This may include creating mock objects, setting up the database state, or configuring the environment to ensure that the test runs in isolation.
When: During the test case execution, invoke the method or function under test with the specified inputs. Ensure that all necessary preconditions are met and that the system is in the correct state to perform the operation.
Then: After executing the test case, assert the expected outcomes. This includes checking the state of the system, verifying that the correct outputs are produced, and ensuring that any side effects (such as changes to the database or external systems) are as expected. Use assertions to confirm that the actual results match the expected results.

Evaluating Test Suites

The more **effective** a test is, the less **efficient** it becomes.
Mutation testing: works by representing the introduction of defects into a system as combinations of small-scale code mutations of the target system's code.
Mutant operations: conditional operators with their boundary counterpart.
Killed mutants: successfully detected by the test suite. **Survivor mutants**: successfully pass all tests and are undetected. **Undetermined mutants**: programs that do not halt.

System Scale Testing

Reliability testing: PFD (probability of failure on access), meantime to failure (time for system to fail from initiation, or time be-tween failures; good metric when repair is expensive), down-time (useful when system is high-demand).
Hostile system environment: A system environment may be considered hostile if there is a rationalization for why threats/attackers may wish to exploit it.
Fuzz testing: Providing unusual inputs. **Penetration testing**: An attacking team attempts to gain unauthorized access with the expected tools of a hacker.
Heterogenous systems: The greater the variation in organisational culture the harder it is to develop a consistent testing programme. Despite agreed standards, variations inevitably occur.
Social-technical systems: incorporate both computer software and hardware, the computer system's users, and the surrounding organ-isational and cultural practices.
Systems of systems: represents multiple heterogeneous semi-autonomous systems that cooperate or are coordinated to produce emergent effects.
Feedback from the wild: Crash reporting, A/B testing.

Software Architecture

Software component: refers to a software bundle of self contained state and behaviours with well defined interfaces. Some components require functions provided by other components.
Objects versus Components: Components are specializations of the object-class type. Components are long-living entities, deployed for the full life-time of a software system. Component middle-ware allows components to distribute be-tween different component environments and different hard-ware. Cannot interact with component implementation directly likeyou can with objects. Each of the interfaces provided by a component may be re-alised by a different object within the component.
Why not componentize every object?: Communication costs increase from mediating (middleware) component interaction. Development costs: documentation of component interfaces need to be maintained.
Types of components: general purpose, application specific.
Design by contract: benefits of using the interface as offered by the providing com-ponent, obligations imposed on the component that uses the interface
Leaky abstractions: whenever two component implementations (aprovider and a requirer of an interface) are wired together, theirfuture is influenced by assumptions on how theinterface will beutilizedandrealized.
Architectural patterns: Model View Control, Client-Server, Peer-to-Peer, Message-oriented architecture, Pipe and Filter, Plugin architecture.
Thin-client architecture: Purist approach to client-server. Clientscontain minimal logic.
Fat-client architecture: Clients perform more logic. Reduces com-munication with server by caching information.
Peer-to-peer: Resolves issue of resource scalibility. Every peer isa client and a server. All logic moved to clients (goes further thanfat-client).
Information processing patterns: Message-oriented architecture, pipe and filter.
Message-oriented architectural pattern: provides a basis for asyn-chronous communication. Communication occurs as discrete mes-sages passing through a message bus, which re-routes to the appro-priate client based on routing policy. **Message driven**: Computation in a component is the resultof message reception from another component. **Message broker**: Deciding which component receives the mes-sage
Pipe and Filter Architecture: Each filter provides and implements the same interface, called the pipe. The filtered wired into an assembly form a pipeline. Data source component provides input and requires the pipe interface. Data sink component provides the pipe interface on the rightto accept the system's output on the right. To allow re-orderable filters each filter must provide and requirethe same interface.
Plugin architecture maintains a flexible mechanism for extension. Plugins stored in plugin registry. A Plugin provides an interface to the core application. Loader component instantiates and configures the component for use by the main application, using the registry supplied specification. Inner platform effect.

Software Refactoring

Refactor when implementing new functionality, correcting defects, code reviews, trying to understand a software artefact.
Code smells: cloning, complex structures, long parameter lists, excessive comments

Software Licensing

A software license can cover ownership, distribution rights, usage rights, liability, etc.
Copyright: The legal right to control the reproduction of a creative work for a specific time; varies between jurisdictions **Warranty**: The length of time for which certain functionality can be expected/resolved. **Liability**: Where the responsibility lies.

Startup Growth Engineering

Basic compounding growth loop: Users attract other to consider product *if and only if* some of them use the product. **Direct-invitational loop**: Users invite their colleagues *if and only if* some invitees sign-up.
Traditional growth engineering: Marketing (acquire users) and product engineering (build features). **Compounding growth**: The rate of growth is proportional to the number of users. **Churn**: The percentage of users lost after successive time periods. **Optimizing growth cycle**: Quantify each stage of the growth cycle and see which stages can be improved. Gauge effects and continue.