# SOFT2412 Final Exam Review (Weeks 1–12)

## Week 1: Introduction to Software Development Processes

**Software Engineering vs. Programming:** Software engineering is an **engineering discipline** covering all aspects of software production, from specification through maintenance[1]. It involves systematic methodologies, tools, and people management beyond just writing code. A **software process** defines a lifecycle of activities (specification, design/implementation, testing/validation, evolution) and their order[2][3].

**Common Software Process Models:** Several **Software Development Life Cycle (SDLC)** models exist, each describing activities and their sequence[4]. Key models include:

- **Waterfall Model:** A linear, phase-driven model with sequential stages – **Requirements → Design → Implementation → Testing → Maintenance**[5][6]. Each phase must complete before the next begins. *Advantages:* simple to understand, clear milestones[7]. *Disadvantages:* inflexible to change; problems detected late may require costly backtracking[8]. Waterfall suits projects with well-understood, stable requirements (e.g. certain large engineering projects) but struggles with rapid change[9][10].

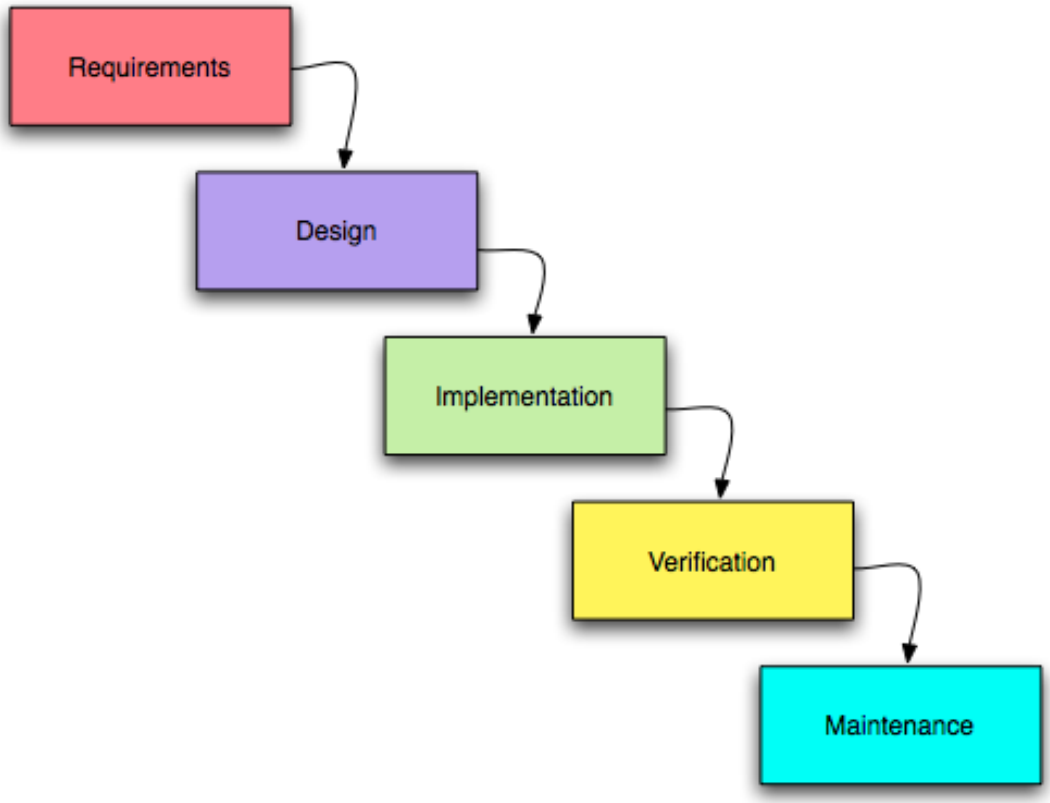


*Figure: The classic Waterfall model – a sequence of phases (Requirements, Design, Implementation, Verification/Testing, Maintenance) where each feeds into the next. Changes are expensive once later phases are reached, making this model suitable only when requirements are stable.*[11][12]

- **V-Model:** An extension of Waterfall emphasizing testing. For each development phase there is a corresponding test phase, forming a "V" shape (e.g. requirements phase matches acceptance testing, design matches system testing, etc.)[13][14]. *Key idea:* Plan tests in parallel with development phases, enabling early defect detection[15] (see **Figure: V-Model** below).

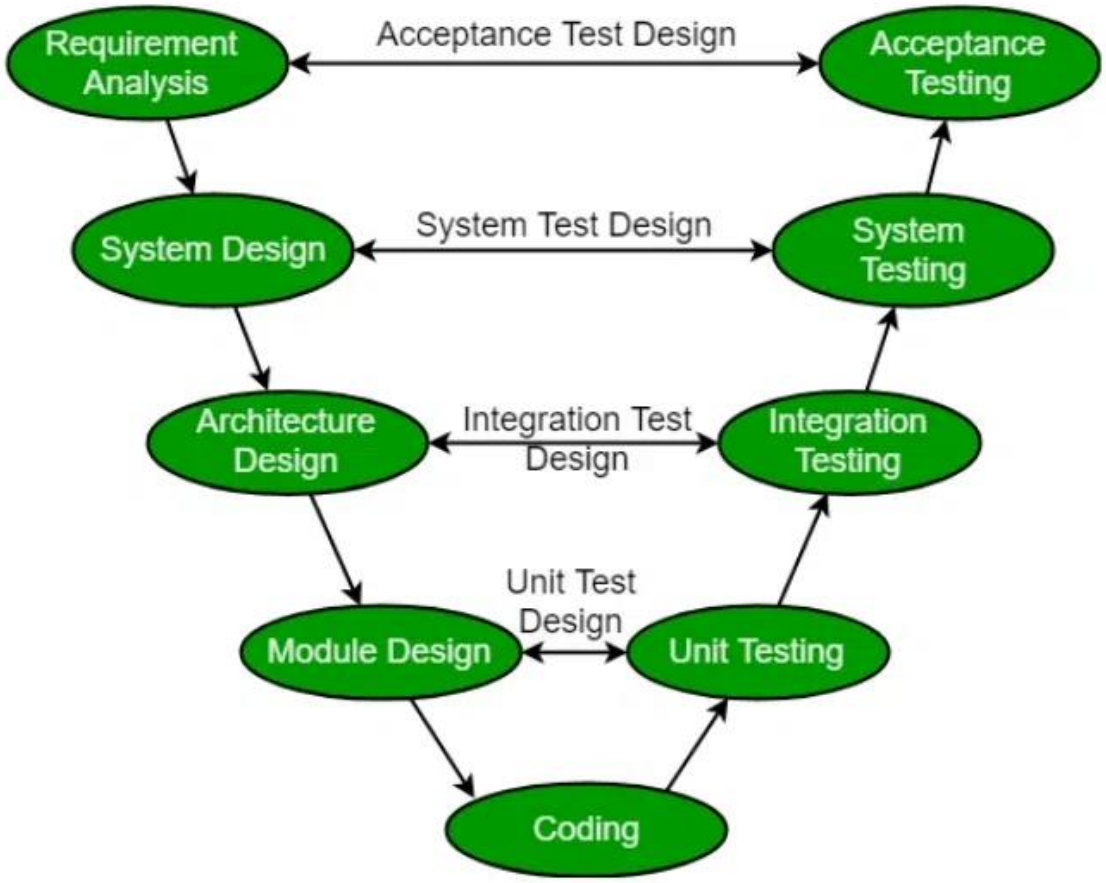


*Figure: V-Model – a development process where the left side represents development phases (requirements, system design, architecture, module design, coding) and the right side represents corresponding test phases (unit, integration, system, acceptance testing). Each development stage has an associated test plan, ensuring verification and validation steps mirror development.*[16][17]

- **Spiral Model:** An **iterative, risk-driven** process. Development passes through repeated loops (spirals), each containing planning, risk analysis, engineering (development & testing), and evaluation phases[18][19]. This model focuses on early identification and mitigation of risks and is suited for large, high-risk projects. Each iteration produces a prototype or increment, and requirements can be refined with each loop[20][21].

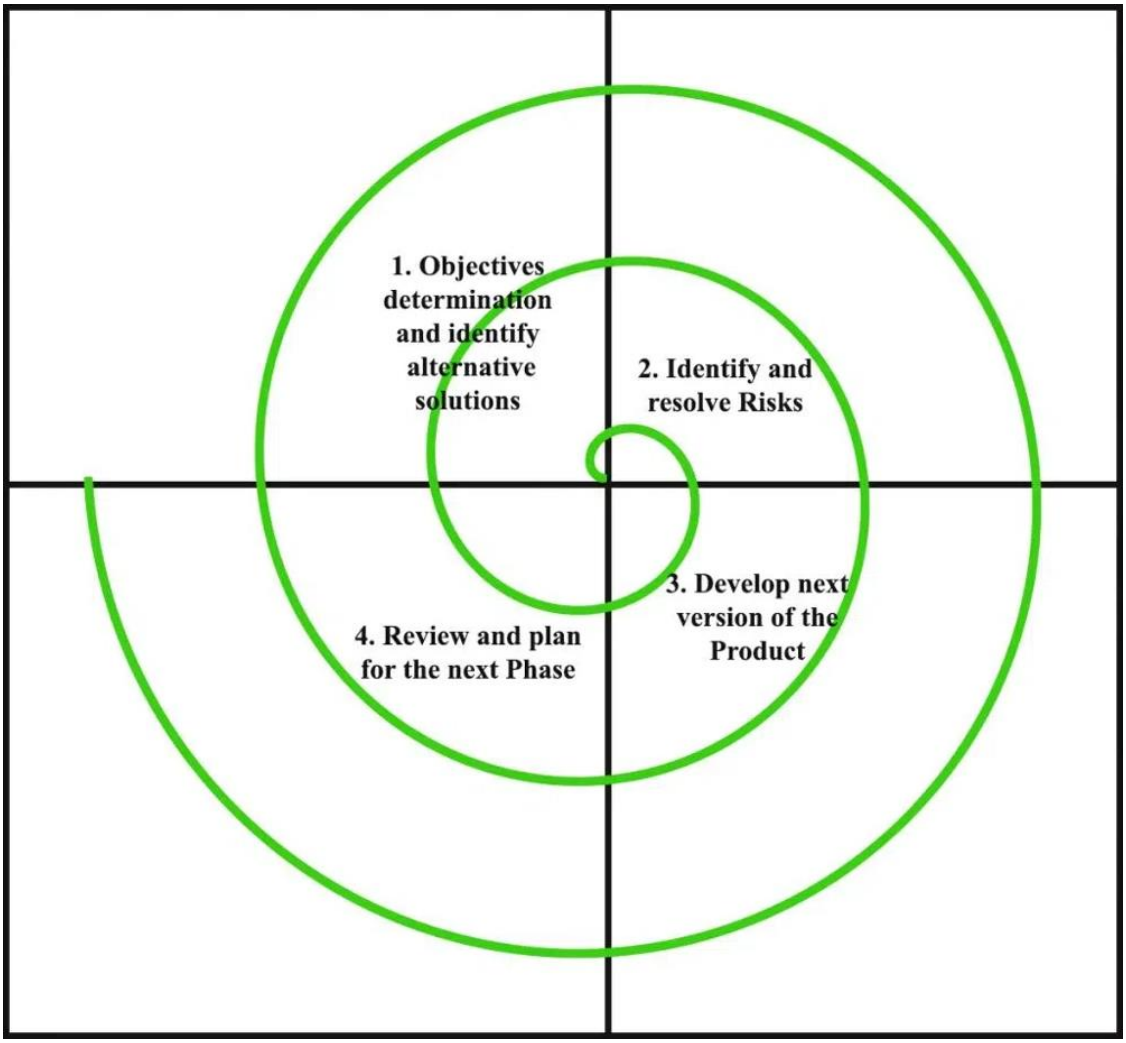


*Figure: Spiral model – development is organized in loops addressing 1) Objectives and alternative solutions, 2) Risk analysis and resolution, 3) Development and verification, 4) Review and planning for the next cycle[20][22]. The project repeatedly goes through these stages, mitigating risks and refining requirements with each iteration.*[20][23]

- **Agile Model:** A family of "light-weight" **iterative and incremental** methods focused on rapid delivery of working software and responsiveness to change[24]. Rather than a heavy upfront plan, agile processes embrace evolving requirements and continuous stakeholder feedback. Common agile approaches (Scrum, Extreme Programming, etc.) will be detailed in Weeks 8–10. In general, *plan-driven vs. agile:* Plan-driven (waterfall-like) processes plan everything in advance and resist change, whereas agile plans continuously and welcomes requirement changes[25]. Most real-world processes blend both approaches depending on context[26].

**Agile Manifesto (2001):** Articulates core values favoring *individuals and interactions, working software, customer collaboration,* and *responding to change* over corresponding traditional emphases (processes, documentation, contracts, plans). Twelve guiding principles include satisfying the customer through early and continuous delivery,

welcoming changing requirements even late in development, delivering working software frequently, daily cooperation between business people and developers, and continuous improvement[27][28]. These principles underpin agile methods' focus on flexibility and customer value.

**Other Models:** *Incremental development* (deliver software in small functional pieces), *Rational Unified Process (RUP)* (a heavy-weight iterative framework combining elements of other models[29]), etc., also exist. There is **no "one size fits all"** – different projects may require different process models or hybrids. Key is understanding the project's needs (risk level, requirement volatility, team size, etc.) to choose an appropriate process.

**Key Exam Tips:** Focus on **strengths/weaknesses of each model** (e.g. Waterfall vs Agile), definitions of activities (requirements, design, validation), and the **Agile Manifesto values/principles**. For example, be able to explain why **Waterfall** struggles with change or how **Spiral** handles risk. Understanding plan-driven vs. agile characteristics is important[30].

## Week 2: Tools and Technologies for Controlling Artifacts (Version Control)

Software artifacts (code, documents, etc.) are valuable work products that **must be managed and versioned**. **Version Control Systems (VCS)** track changes to files over time so that specific versions can be recalled and changes by multiple people merged[31][32].

**VCS Types:**
- **Local VCS:** Simple databases (on one computer) to track file revisions. Rarely used now due to lack of collaboration support.
- **Centralized VCS (CVCS):** A single central repository that clients check out from. Examples: CVS, SVN. Multiple users can work concurrently; commits merge changes (optimistic concurrency)[33][34]. *Pros:* Everyone sees a single latest version, simple administration. *Cons:* Central server is a single point of failure; requires network access; branching is heavy-weight (often copying entire project)[35]. Merging conflicts can be cumbersome[36].
- **Distributed VCS (DVCS):** Each user has a full copy of the repository (including history). Examples: **Git**, Mercurial. *Pros:* Fast local operations, no single failure point, easy branching and merging (lightweight branches)[37][38], allows various workflows (e.g. decentralized collaboration via multiple remotes).

**Git Fundamentals:** Git is a widely used DVCS. It treats data as snapshots of the project over time (snapshot-based storage) rather than file diffs[39]. Key properties: - **Local operations:** Almost every operation (commits, diffs, logs) is done locally without a network, making Git very fast[40]. - **Integrity:** Every file and commit is checksummed (SHA-1 hash) – data is **immutable and referenced by hash**, making it difficult to lose commits[41]. - **Three states: Modified** (file changed, not yet committed), **Staged** (file marked to include in next commit), **Committed** (change saved in the repository)[42]. Git's structure includes

a **Working Directory**, **Staging Area (Index)**, and **Repository** (`.git` folder)[43][44]. You edit files in the working directory, stage desired changes, then commit to repository. - **Workflow basics:** Edit files -> `git add` (stage changes) -> `git commit` (record snapshot). You can view status with `git status` and history with `git log`[45][46]. *Tracked vs Untracked:* Files Git knows about vs new files not yet under version control[45].

**Branching and Merging in Git:** Branches in Git are **movable pointers** to commits[47][48]. The default branch is usually **master** (or `main`). Creating a new branch (e.g. `git branch featureX`) is instantaneous and just creates a new pointer to the current commit[48]. Switching branches (`git checkout branchName`) moves HEAD to that branch, changing the working copy to match that branch's last commit[49][50]. Development often follows a branching strategy: for example, create a branch for a new feature or bugfix so as not to destabilize the main line. Git encourages frequent branching due to its low cost.

- **Merging:** To bring changes from one branch into another, Git provides merge operations. If the branches evolved independently, Git tries an automatic **three-way merge**, using the common ancestor and each branch's latest commit to combine changes. If two branches changed the **same lines**, a **merge conflict** occurs. Git marks conflict areas in the affected files with special **conflict markers** (e.g. `<<<<<< HEAD` in files) indicating two versions[51]. The developer must **manually resolve conflicts** by editing the file to the correct content and then marking as resolved (`git add` the file) before committing the merge[51][52].

- **Fast-forward merge:** If a branch has progressed and the target branch has no new commits, merging simply moves the pointer forward (no divergent history)[53]. If history diverged, a merge commit is created to join them.

**Example – Branching Workflow:** A common scenario is using branches for features and hotfixes. Suppose you have a `master` branch for production. When developing a new feature "ISS-53":

```
$ git checkout -b iss53    # create and switch to feature branch
# ...make commits on iss53 branch...
$ git checkout master      # switch back to main branch for an urgent fix
$ git checkout -b hotfix   # create hotfix branch from master
# ...fix issue and commit on hotfix...
$ git checkout master
$ git merge hotfix         # merge hotfix into master (fast-forward if no div
ergence)
$ git branch -d hotfix     # delete hotfix branch after merging
$ git checkout iss53
# ... continue working on feature, then later ...
$ git merge master         # merge latest master (hotfix) into feature if nee
ded
$ git checkout master
$ git merge iss53          # merge finished feature back to master
```

In this example, the team could continue feature work while quickly addressing a production bug on a separate branch[54][55]. After verifying the hotfix, it was merged back. When feature development finished, it was merged into master. **Conflict Handling:** If both the feature and hotfix modified the same code, the merge would raise conflicts to be resolved[56][57].

**Distributed Collaboration:** With Git, teams often use remote repositories (e.g. on GitHub). Key commands: `git **fetch**` (download new commits from remote), `git **merge**` or `git pull` (which does fetch+merge) to integrate them, and `git **push**` to upload local commits to a remote server[58][59]. Branches can have tracking relationships (e.g. local `main` tracking `origin/main`). Collaboration models include *centralized workflow* (everyone pushes to one shared repo) and more complex flows like *Integration Manager* or *Dictator-Lieutenant* (see Week 3) where contributions flow through multiple levels[60][61].

**Key Exam Focus:** Understand the **Git commit lifecycle (modify → stage → commit)**[42], how **branching/merging** works, and how to handle a **merge conflict**[51][52]. Be able to explain differences between centralized and distributed VCS (e.g. SVN vs Git)[37][38]. Also know basic Git terminology: HEAD, fast-forward, rebase (in L3 context, rebase means integrate changes by replaying commits instead of merge, often used to avoid merge commits – e.g. `git pull --rebase` uses rebase[62]). Also recall best practices like committing often, writing good commit messages, and **resolving conflicts by editing then staging files** (Git won't consider a conflict fixed until you add the file after editing).

## Week 3: Managing Collaborative Development Artifacts (Advanced Git & Collaboration)

Building on version control basics, this week covers **collaborative workflows and repository hosting.**

**Remote Repositories & Branches:** In Git, *remote repositories* allow teams to collaborate. A remote (e.g. `origin`) is just another copy of the repo on a server. **Remote-tracking branches** (like `origin/master`) reflect the state of remote branches at last fetch/pull[63][64]. Key commands: - `git fetch <remote>`: download new commits from remote (does not merge)[65]. - `git pull <remote> <branch>`: fetch and then merge (or rebase) into your current branch[59][66]. By default `git pull` merges; `git pull --rebase` integrates changes by rebasing[66]. - `git push <remote> <branch>`: upload your new commits on a branch to the server[67]. If others pushed conflicting changes, push will be rejected until you merge/rebase their work.

**Collaboration Workflow Models:** Different projects use different Git workflows:
- **Centralized Workflow:** All developers have write access to a single central repo. Developers push to the main branch; conflicts are resolved as they arise. Simple but doesn't scale well to very large teams.
- **Integration-Manager Workflow:** Common in open-source. Developers **fork** the main repository (their own public copy), push changes to their fork, then submit **pull requests** or

merge requests to an integrator. The integration manager (project maintainer) reviews and merges changes into the official repo. This limits direct pushes to the official repo and adds a review gate.
- **Dictator-and-Lieutenants Workflow:** Used by very large projects (e.g. the Linux kernel). There is a "benevolent dictator" (project lead) and trusted lieutenants. Developers work on topic branches, frequently **rebase** their work on the latest `master` (to keep history linear)[60][61]. They submit their changes to a lieutenant responsible for a subsystem. Lieutenants merge changes into their own branch and pass upstream. The project dictator then merges lieutenants' branches into the main repository[61][68]. This hierarchical model helps manage extremely large contributions. **Rebasing** in this context avoids too many merge commits – developers apply their commits on top of the latest master so history looks as if changes were developed sequentially.

**Handling Multiple Contributors:** With many contributors, conflict frequency increases. Good practices include: - **Frequent integration:** Don't let branches drift too far apart – integrate mainline often to reduce big bang merges. - **Consistent workflow guidelines:** e.g. require code review via pull requests, enforce all tests passing before merge (often via CI – see Week 6). - **Commit access control:** Large projects may restrict who can push to certain branches (e.g. only maintainers can push to `main`). Others must contribute via forks and pull requests, which are then merged.

**Commit Guidelines:** Teams often adopt standards for commits: - Write **descriptive commit messages** (what & why of changes).
- Make **logical commits** – each commit should address one issue/feature. - Avoid commits with unrelated changes. - Use code review (another pair checks commits before merge). - Check for common mistakes like whitespace errors (Git's `--check` flag can detect trailing whitespace before commit)[69].

**Repository Hosting & Team Management:** Many projects host on platforms like **GitHub, GitLab, or Bitbucket** which provide additional collaboration tools: - **Pull Requests/Merge Requests:** Proposed changes where team members can review code, discuss, and approve before merging. - **Issue Tracking:** Link issues or user stories to code changes for traceability. - **Continuous Integration hooks:** Automatically build and test on each push (see Week 6). - **Organizations and Teams:** On GitHub, an organization can own repositories and manage members' access rights (Owners vs Members)[70][71]. You can create teams with specific repository permissions (e.g. a "Frontend Team" with access to front-end repo only)[72][73]. Fine-grained permissions and audit logs help manage large projects securely[74][75].

**Continuous Collaboration Issues:** Be aware of issues like the *"throw it over the wall"* mentality – developers, testers, ops working in isolation. Agile emphasizes breaking down these silos (developers and business working together daily, etc.)[76][77]. Modern DevOps culture (see Week 6) also stresses collaboration between dev and ops rather than finger-pointing ("it works on my machine, ops problem"). Good version control practices + team culture mitigates these issues.

**Key Exam Points:** Explain **distributed vs centralized workflows** and know terms like **origin, fetch, pull, push**. Understand how an **open-source contribution** works (fork & pull request model) versus an **enterprise with direct pushes**. Possibly be ready to describe the **Dictator-Lieutenant model** (Linux)[60] or why teams might enforce rebasing (to keep history linear). Also mention how **GitHub facilitates collaboration** (issues, pull requests, teams) and ensures codebase integrity via reviews and CI. Lastly, recall any **commit best practices** and **common pitfalls** (e.g. avoiding merge conflicts by frequent syncs, using feature branches, writing clear commit messages).

## Week 4: System Build Automation

As projects grow, manually compiling and deploying becomes error-prone. **Build Automation** ensures that building the system (compiling code, linking, packaging into executables or deployable units) is repeatable and reliable with minimal human effort. It is a key part of **Software Configuration Management (SCM)**, which also includes version management, change management, and release management[78][79].

**Why Build Automation:**
- **Consistency:** Automated builds produce consistent results (no "it works on my machine" issues).
- **Efficiency:** One command can compile dozens of files, run tests, package artifacts, etc. Saves developer time and reduces mistakes.
- **Continuous Integration:** Build automation is essential for CI (Week 6) – every commit can trigger an automated build/test.
- **Complexity management:** Large projects may have multiple modules, dependencies, and configurations. Build tools handle these systematically.

**Make and Makefiles: Make** is one of the oldest build tools (for C/C++ and others). It uses a **Makefile** to define **targets**, **dependencies**, and **commands**. For example:

```
# Simple Makefile example
hello: hello.o util.o
    gcc -o hello hello.o util.o

hello.o: hello.c
    gcc -c hello.c

util.o: util.c
    gcc -c util.c

clean:
    rm -f *.o hello
```

In this example, "hello" (executable) depends on object files; Make will rebuild `hello` if any dependency changed. Make's fundamental is **timestamp-based**: it rebuilds targets whose prerequisites have newer timestamps. Key features: macros (variables), pattern rules, and

special targets (like `.PHONY` for targets that are not real files, e.g. “clean”)[80][81]. Modern best practices for Make: - Always provide an `all` target (default build) and a `clean` target to cleanup artifacts[80]. - Use `.PHONY` for targets like `clean` that don’t correspond to files[81]. - Use variables for compilers/flags (e.g. `CC = gcc`). - Test with `make -n` (dry run) to see what would happen without actually executing commands[82].

Make is powerful but can become complex for large projects, especially with cross-directory builds, custom rules, etc. It also doesn’t manage dependencies beyond what you script (e.g., fetching libraries).

**Apache Ant:** Ant is a Java-based build tool that uses **XML** files (`build.xml`) to describe build processes. Unlike Make’s implicit rules and dependency on file timestamps, Ant is more procedural. You define **targets** and **tasks** in XML. Example snippet:

```
<project name="Hello" default="jar">
  <target name="compile">
    <mkdir dir="classes"/>
    <javac srcdir="src" destdir="classes"/>
  </target>
  <target name="jar" depends="compile">
    <jar destfile="hello.jar">
      <fileset dir="classes" includes="**/*.class"/>
      <manifest><attribute name="Main-Class" value="HelloProgram"/></manifest>
    </jar>
  </target>
</project>
```

Here, Ant compiles Java source to `classes/` directory, then packages a JAR. Ant doesn’t force a particular project structure or conventions – it’s *highly flexible*[83][84]. However, this flexibility is a double-edged sword: - *Pros:* You can script any sequence of steps; not limited to file timestamps. Good for complex tasks.
- *Cons:* **Verbose XML**, requires specifying many details (even for simple builds), and since there’s no standard structure, each project’s Ant file might be very different. Developers can essentially “roll their own” build process, which can be inconsistent[85][86]. Maintaining large Ant scripts can become cumbersome.

**Apache Maven:** Maven (for Java and other JVM languages) takes a different approach: **“convention over configuration.”** Instead of writing detailed instructions, Maven defines a **standard project structure and lifecycle** and uses a **Project Object Model (POM)** file (`pom.xml`) to configure the build. Maven conventions, for example, assume source code is in `src/main/java`, tests in `src/test/java`, outputs go to `target/`, etc. If you follow the conventions, you need little configuration. Maven’s key features: - **Lifecycle phases:** e.g., `compile`, `test`, `package`, `install`, `deploy`. Running `mvn package` will automatically compile sources, run tests, then package the code (e.g., into a JAR)[87]. - **Dependency Management:** Maven can automatically download project dependencies from a central repository (like Maven Central). You declare dependencies in the POM, including groupId,

artifactId, version, and Maven fetches them (and transitive dependencies) for you[88][89]. This eliminates the need to manually include library jars. - **Plugins:** Maven's functionality is extended via plugins (for compiling Java, running tests, etc.). Many frameworks have Maven plugins. - **POM Example:** In a POM, you specify coordinates and dependencies. For instance, adding JUnit dependency:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId><artifactId>junit</artifactId>
        <version>4.13.2</version><scope>test</scope>
    </dependency>
</dependencies>
```

Maven will include JUnit on the test classpath and run tests with it.

*Pros:* Standardization – any Maven project has a similar structure and build commands. Less verbose if you accept conventions. *Cons:* The rigid structure may not fit all projects, and XML can still be verbose. Also, Maven's dependency management can lead to "dependency hell" if not managed (though better than manually tracking jars). The summary notes that while Maven uses conventions and has a central repo, it can be **verbose** and **rigid** (you must adapt to Maven's way)[90][91].

**Gradle:** A newer build tool that combines the flexibility of Ant with the conventions of Maven. Gradle uses a **Groovy (or Kotlin) DSL** instead of XML. It's highly customizable and supports incremental builds. Not explicitly detailed in lecture slides until summary: Gradle builds are organized into tasks with dependencies (forming a DAG – Directed Acyclic Graph)[92][93]. Gradle embraces convention (especially if using **Gradle Wrapper** and standard plugins for Java, etc.) but is extensible via code. It's popular for Android and modern Java projects.

**SCM Concepts in Build:** This week likely introduced broader **configuration management concepts**: - **Codelines and Baselines:** A **codeline** is a sequence of versions of source code (like a branch). A **baseline** is a stable snapshot (a labeled/released version) that serves as a reference point[94][95]. - **Semantic Versioning:** Convention of version numbers *MAJOR.MINOR.PATCH* (Major changes break compatibility, Minor add features but backwards compatible, Patch are bug fixes)[96]. - **Release Management:** The process of preparing software for release (packaging, release notes, version tagging) and tracking versions deployed to customers[79]. Build automation supports release management by making it easy to produce release builds reproducibly.

**Key Exam Areas:** Be prepared to explain differences between **Make, Ant, Maven, Gradle** – particularly *how they approach builds* (Make = declarative with targets and file timestamps; Ant = procedural XML scripts; Maven = declarative with strong conventions and dependency management; Gradle = programmable scripts with a mix of convention and flexibility)[97][98]. Know why **build automation** is crucial (ties to CI) and possibly identify a snippet of a build file. For example, understand what a simple Makefile rule means or what

the significance of a Maven `pom.xml` is (don't need to memorize XML, but know it's for listing dependencies and project metadata). Also recall SCM terms like baseline, and versioning schemes.

## Week 5: Software Verification and Testing

Testing is the primary method of **software verification** (ensuring the software meets requirements and is defect-free). This week covers testing principles, levels, techniques, and test automation.

**Why Testing Matters:** Real-world failures (e.g., the Therac-25 radiation machine overdose, Ariane 5 rocket explosion, Nissan airbag recalls) show that software bugs can cause huge financial loss or even endanger lives[99]. Testing improves quality assurance by catching defects before deployment. Even though *"exhaustive testing is impossible"*, well-planned testing significantly reduces risk[100][101].

**Testing Fundamentals:** Testing checks if software meets specified requirements and finds defects. *Who tests?* Developers (unit tests), dedicated QA/testers, and end-users (acceptance/beta testing). Independent testers often find different issues than the original coder[101]. *When to test?* In waterfall, testing is a distinct late phase; in agile, testing is continuous (each iteration includes testing, and automated tests run frequently)[102].

**Testing Levels (Scope):**
- **Unit Testing:** Verify individual components (functions, classes) in isolation. Usually done by developers.
- **Integration Testing:** Verify interactions between integrated units/modules (e.g., testing API calls between two subsystems).
- **System Testing:** Test the complete integrated system against requirements (ensuring the system works as a whole).
- **Acceptance Testing:** Validate software against user needs and requirements – often performed with customer involvement (e.g. UAT – User Acceptance Testing).
- **Regression Testing:** Re-running test suites after changes to ensure no new bugs (regressions) are introduced. This is an ongoing level that can apply to unit/integration/system tests whenever code changes.

These levels form a typical testing pyramid (many unit tests at base, fewer system tests at top). Also note **alpha/beta testing** occur at system/acceptance level (alpha – internal, beta – external users).

**Testing Techniques:**
- **Black-Box Testing:** Designing tests based on specifications **without seeing the code**. Focus on input-output behavior (functional requirements). E.g., for a function defined by a spec, create tests for each requirement, including edge cases. *Equivalence partitioning* and *boundary value analysis* are classic black-box techniques: partition input domain into classes that should be treated similarly, and test at boundaries of these classes[103].
- **White-Box Testing:** Designing tests with knowledge of internal code structure. E.g.,

writing tests to exercise all branches of a decision, loops, or specific paths (also called structural testing)[104]. *Coverage criteria* (like ensuring every line or branch is executed by some test) guide white-box testing.
- **Test-Driven Development (TDD):** An **agile testing practice** where tests are written *before* the code. The cycle: *Red – write a small test and see it fail; Green – write just enough code to pass test; Refactor – improve code while keeping tests green*. TDD yields a comprehensive suite of unit tests and can drive design (ensuring code is testable and well-factored)[104].

**Test Case Design:** Instead of random testing, testers systematically design cases:
- **Equivalence Classes:** Partition inputs into groups where the program should behave similarly[103]. Test one representative from each class.
- **Boundary Values:** Inputs at the edges of equivalence classes often reveal off-by-one errors (e.g., if valid input is 1–100, test 0, 1, 100, 101).
- **Error Guessing/Guidelines:** Leverage experience or common bug patterns (e.g., empty input, null values, large sizes) to choose test cases[105].

**Test Automation Goals:** In modern development, tests are often automated (especially unit and integration tests). Good automated tests are intended to: - Improve quality by catching bugs early (every code change triggers tests). - Serve as **executable specifications** – tests document what the code should do (readable, describe behavior). - **Prevent regressions:** When a bug is found, a test is added to ensure it stays fixed. Running the full test suite frequently prevents old bugs from reappearing[106]. - Localize defects – if tests are fine-grained, a failing test pinpoints the problematic component. - Be easy to run (e.g. one command to run all tests) and ideally fast, so developers run them often. - Minimize maintenance cost – tests should be reliable (not *flaky*) and only fail when there's a real issue, otherwise teams lose trust in them[106].

**Unit Testing with JUnit (Example):** JUnit is a popular framework for Java unit tests[107]. Key features: - **Annotations:** e.g. `@Test` to mark a test method, `@Before`/`@After` for setup/teardown methods (run before/after each test), `@BeforeClass`/`@AfterClass` for class-level setup (run once).
- **Assertions:** Methods like `assertEquals(expected, actual)`, `assertTrue(condition)` to check outcomes. A test passes if all assertions pass without throwing, fails if any assertion fails (throws an AssertionError).
- **Test suites:** You can group tests into suites to run together.
- **Parameterized tests:** Run the same test logic with different inputs.
- **Integration:** JUnit results integrate with IDEs and build tools; for example, Maven's `surefire` plugin will run tests during `mvn test` phase and report results. CI servers also parse JUnit XML reports.

JUnit example structure:

```
public class MathUtilsTest {
    @Before public void setup() { /* setup code */ }
    @Test public void testAdd() {
```

```
        int result = MathUtils.add(2, 3);
        assertEquals(5, result);
    }
    @Test(expected = IllegalArgumentException.class)
    public void testDivideByZero() {
        MathUtils.divide(1, 0); // should throw exception
    }
    // ...
    @After public void teardown() { /* cleanup code */ }
}
```

This shows typical usage: multiple `@Test` methods, checking results and expected exceptions.

**Testing Patterns and Smells:** Over time, practitioners identified **test design patterns** (good practices) and **test smells** (common problems in test code): - *Test Organization Patterns:* e.g., one test class per production class or per feature; use **test fixtures** (`setUp`) to initialize common objects for tests; use **descriptive test names** and clear assertions (so failures pinpoint cause)[108][109]. - *Test Double Patterns:* Often real dependencies are replaced with **test doubles** to isolate the unit under test. Types of test doubles: - **Dummy:** passed around but not used (just to fill parameter lists). - **Stub:** returns canned responses for certain inputs (used to isolate behavior, e.g. a stubbed database always returns the same data for testing). - **Mock:** a fake object that not only stubs behavior but also **verifies** that certain interactions happened (e.g., confirm a method was called with specific arguments). Typically created with a mocking framework. - **Fake:** a lightweight implementation of a component that works but is simpler (e.g. an in-memory database fake vs real database). - **Spy:** like a stub that records information about how it was used, so tests can assert on that later.
- *Test Smells:* Indications of potential problems in tests[109][110]: - **Fragile Tests:** Tests that fail after minor code changes that don't actually break functionality – often due to over-specification or reliance on details that change (like GUI element positions, or internal code structure). Fragile tests might break if method implementation changes even if output is same[111][112]. Categories include *Interface sensitivity* (UI or API changes break tests), *Behavior sensitivity* (small spec changes break many tests), *Data sensitivity* (tests rely on specific external data state), *Context sensitivity* (dependent on environment, like date/time or external services)[113][114]. - **Assertion Roulette:** Multiple assertions in a test with no messages, so when a test fails it's unclear which assertion failed. The failure output might not indicate what was being checked[115]. - **Flaky Tests:** Tests that sometimes pass, sometimes fail without code changes (nondeterministic). Causes: concurrency issues, tests depending on timing, external resources, or shared state ("test run wars" where tests interfere with each other)[116][117]. Flaky tests undermine confidence in the test suite. - **High Maintenance Tests:** Tests that need frequent updates when code changes, possibly because they test internal implementation too closely or aren't robust to requirement changes[118]. If tests are hard to maintain, there's a risk management will abandon testing. - **Developers Not Writing Tests:** A project smell where the team isn't contributing tests due to schedule pressure, lack of skill, or discouragement

from management[119]. This often surfaces in retrospectives when quality issues arise. It can be addressed by fostering a testing culture and addressing barriers (like making test writing easier, pairing newbies with experienced testers, etc.).

**Code Coverage:** A metric to evaluate how much of the code is exercised by tests. Coverage criteria include: - **Statement Coverage:** % of statements executed by tests[120]. - **Branch Coverage:** % of branches (if/else outcomes) taken[121]. - **Condition Coverage:** % of boolean sub-expressions evaluated both true and false[122]. - **Loop Coverage:** Each loop run 0 times, 1 time, >1 times at least once[123]. High coverage (like 80%+ statements) is generally good, but coverage alone doesn't guarantee bugs are caught – it's possible to have high coverage with poor assertions. However, safety-critical software often mandates near 100% coverage of certain types (e.g. NASA or aviation software standards)[124]. Tools such as **JaCoCo/EclEmma** for Java measure coverage and highlight untested code (e.g., in Eclipse IDE, green/yellow/red highlighting lines for fully/partially/not covered)[125][126]. Coverage tools integrate with CI to ensure coverage does not drop.

**Testing in Agile:** Agile methods stress **test automation** and practices like **TDD** or **Continuous Testing**. The idea of writing tests early (even before code, in TDD) ensures testability and that you have a safety net of regression tests. Agile teams often use **CI (Continuous Integration)** to run tests on every commit (see Week 6) and maintain a *"green"* build (tests passing).

**Key Exam Topics:** Be ready to distinguish **black-box vs white-box** testing[127], name the **levels of testing** (unit/integration/system/etc.)[128], and explain one or two **test design techniques** (like equivalence classes and boundary values[103]). Know what **TDD** is and its red/green/refactor cycle. Understand what is meant by **test fixtures, mocks/stubs**, and be able to cite examples of **test smells** like flaky or fragile tests[129][115]. Also, basic knowledge of **coverage criteria** (e.g. "What does 100% branch coverage mean?") and a tool like JaCoCo could be useful[120][121]. Finally, know the importance of **automated testing in CI** and goals of test automation[106] (e.g. preventing regressions, documentation, quick feedback).

## Week 6: Continuous Integration & Continuous Delivery (CI/CD)

Modern software teams use **CI/CD pipelines** to frequently integrate and deliver changes reliably. **Continuous Integration (CI)** is the practice of merging all developers' working copies to a shared mainline **at least daily** with automated builds and tests to catch issues early[130]. **Continuous Delivery (CD)** extends CI through automated deployment up to staging or production environments so that software could be released at any time (release is a business decision)[131]. **Continuous Deployment** goes one step further – every change that passes all tests is automatically deployed to production without human intervention[132].

**DevOps Culture:** CI/CD is part of the broader **DevOps** movement, which emphasizes collaboration between Development and Operations, automation of processes, and quick feedback loops. DevOps goals include **shortening the software lifecycle** (faster releases),

improving reliability, and sharing responsibility for quality across dev, QA, and ops[133][134]. Key DevOps principles relevant here: - **Collaboration:** Break silos (developers, testers, ops work together, share ownership)[135]. - **Automation:** Everything from builds to testing to infrastructure provisioning is automated to eliminate manual errors and speed up delivery[134][136]. - **Infrastructure as Code (IaC):** Manage servers and environments with code scripts (e.g. Docker, Terraform) so they are versioned and reproducible[137]. - **Monitoring & Feedback:** After deployment, monitor systems and user feedback to quickly detect issues and inform the next iterations[138].

**Continuous Integration Practices:**
- **Frequent Commits:** Developers commit small incremental changes frequently (at least once a day)[139]. This reduces integration conflicts because changes are bite-sized. Integration frequency being high makes merges easier and if a bug is introduced, it's in a small change set. - **Automated Build & Test:** A CI server (e.g. Jenkins, GitLab CI, GitHub Actions) monitors the repository. When changes are pushed, the server automatically compiles the code and runs the test suite (unit tests, possibly integration tests)[140][141]. The goal is to get feedback within minutes – typical guidance is CI builds should run in under ~10 minutes if possible[142][143]. Longer-running tests (like extensive integration or UI tests) might be run less frequently or in parallel pipelines so as not to slow down the quick feedback. - **Fail Fast and Fix:** If the build or tests fail, CI marks the build as broken. A key team rule is **never leave the build broken** – if something fails, developers stop and fix it immediately or revert the offending change[144][145]. This prevents compounding errors and ensures a **deployable mainline** at all times. - **Pre-merge Checks:** Some workflows require that changes pass CI tests (in a branch or via a pull request) *before* merging to main. This prevents breaking main altogether. - **Artifact Storage:** CI often archives build artifacts (compiled binaries, Docker images, etc.) and test results (reports, logs). This helps in debugging failures and also preparing deliverables for CD.

**Deployment Pipeline (Continuous Delivery):** A deployment pipeline is an automated path that code takes from commit to deployment[141][146]. Stages typically include: 1. **Commit stage (CI build):** Compile code, run unit tests (fast feedback).
2. **Automated acceptance tests:** More extensive tests (integration tests, UI tests, API tests) possibly in a staging environment. These might be a second stage triggered if commit stage passes.
3. **Deploy to staging/UAT:** The build is deployed to a staging environment closely resembling production. Perform user acceptance tests or exploratory testing. Possibly also perform performance tests or security scans here.
4. **Production deployment:** In Continuous Delivery, this step is manual (a human can push the "deploy to prod" button when ready). In Continuous **Deployment**, this step is automatic – if all prior stages pass, the pipeline deploys to production immediately.

Each stage acts as a quality gate. Only builds that pass all tests and checks flow to the next stage. The pipeline ensures that **software is always in a releasable state**[147][148] – you could decide at any time to release the latest successful pipeline build.

**Key CI/CD Concepts:**
- **Never go home on a broken build:** A cultural rule emphasizing responsibility – don't leave a failing build for others; fix it or rollback.
- **Small batches:** Integrating small changes often (reduces risk). If a pipeline fails, it's easier to find the bug among a few commits.
- **Deployment Automation:** Scripting the deployment (infrastructure, configuration, installation steps) so it's repeatable. This could involve containerization (Docker images built in CI) or infrastructure-as-code to spin up servers. Automated deploys eliminate manual errors and make deployments routine[149][150]. - **Rollbacks:** The pipeline should allow easy rollback if a deployment causes issues (e.g. keep previous version artifacts, or use blue-green deployment strategies). While not explicitly in lecture, it's part of CD best practices.

**CI/CD Tools – Jenkins Example: Jenkins** is a popular open-source automation server for CI/CD. It supports "Pipeline as Code" – Jenkinsfile where you script the pipeline steps (declarative or scripted)[151][152]. Jenkins can handle: - **Multiple stages**: e.g. Build, Test, Deploy steps defined in code. - **Parallel execution**: run tests in parallel to speed up pipeline. - **Integration with tools**: e.g. Docker (build images), JUnit (parse test results), artifact repositories, notifications (Slack/email on failures)[153][154]. - **Plugins ecosystem:** Jenkins has hundreds of plugins to integrate with version control, cloud platforms, static analyzers, etc. - The lecture mentions special steps like retrying flaky steps, timeouts, and post-build cleanup in Jenkins pipelines[155]. That implies knowledge of writing robust pipelines that can handle occasional infra issues. - **History:** Jenkins (2011) succeeded tools like CruiseControl (2001) and even earlier systems like Tinderbox[156][157]. It's noted for introducing pipeline-as-code and broad plugin support[158][159].

**Real-World Impact:** High-performing teams deploy very frequently. Examples given: - Amazon: deployment every 11.6 seconds on average[160]. - Facebook: 2 deployments per day (as of 2013)[160]. - Google: 30,000+ commits per day integrated with a massive automated testing infrastructure[161]. These illustrate the extreme end of CI/CD capabilities.

**Metrics:** With CI/CD, teams often measure: - **Build duration** (want it low). - **Deployment frequency** (how often you release). - **Change lead time** (from commit to production). - **Change failure rate** (what percentage of releases cause incidents). - **MTTR** (mean time to recover from failure).
These DevOps metrics correlate with software delivery performance (from the "Accelerate" research, not explicitly in lecture but relevant to CI/CD goals).

**Key Exam Areas:** Be able to **define CI and CD** and differentiate Continuous Delivery vs Continuous Deployment[130][132]. Know the **main benefits of CI** (early bug detection, integration risk reduction) and **practices** (frequent commits, automated tests, immediate fix on fail)[140][144]. Possibly expect a question on what a **deployment pipeline** is or stages in a pipeline[140][146]. Also recall some specifics about **Jenkins** or CI tools – e.g.,

what is a Jenkins pipeline, or how CI tools have evolved (CruiseControl to Jenkins)[156]. Understanding **DevOps culture** points (collaboration, automation, monitoring) could be relevant to an essay question[133][134]. Finally, linking CI/CD to previous topics: build automation (Week 4) is a prerequisite for CI; automated tests (Week 5) are essential for CI; version control (Weeks 2–3) enables CI triggers. So think of CI/CD as integrating those tools to achieve rapid, reliable delivery.

## Week 7: Team Dynamics and Collaboration

Software development is a **team effort**, and understanding team dynamics is crucial. **Team dynamics** refers to the psychological forces influencing how a team operates and performs[162]. Good dynamics (trust, healthy conflict, cooperation) can boost performance, while poor dynamics (infighting, silo mentality) can derail projects[163][164].

**Teamwork in Agile:** Agile methodologies put heavy emphasis on individuals and interactions (Agile Manifesto value #1) and cross-functional, self-organizing teams. For example, Scrum teams are meant to be self-organizing (decide how to do work) and cross-functional (all skills needed are within the team). Agile principle: "Business people and developers must work together daily throughout the project"[76] underlines collaboration.

**Common Team Problems:** A classic problem is *"throw it over the wall"* – developers, testers, ops, etc., working in isolation and handing off work without ongoing collaboration[77]. This leads to siloed perspectives and blame games. Agile aims to eliminate this by including all roles in one team with shared goals. Another issue can be individuals focusing only on "their" tasks and using only agile practices that affect them (the "Agile Elephant" analogy: each blindfolded person feels a part of the elephant and thinks they understand it)[165][166]. For instance, if developers only care about automated builds and testers only care about the task board, the team might miss the big picture of agile values (like responding to change or customer collaboration). **Whole-team mindset** is needed: everyone is responsible for quality, delivery, and improvement, not just their silo.

**Team Dynamics Definition:** "Team dynamics are the unconscious, psychological forces that influence the direction of a team's behavior and performance"[162]. Factors include personalities, work styles, skills, organizational culture, and background differences[167]. These can manifest in how teams communicate, handle conflicts, and make decisions.

**Tuckman's Stages of Team Development:** Classic model by Bruce Tuckman describing stages teams go through: 1. **Forming:** Team meets and forms initial relationships. *Characteristics:* high reliance on leader for guidance, roles not clear, people polite.
2. **Storming:** As team starts working, conflicts arise over approaches or personalities. *Characteristics:* power struggles, challenge of ideas, clarity of purpose increases but plenty of uncertainties in relationships[168][169]. This stage needs conflict resolution and patience.
3. **Norming:** Team establishes norms and harmonious working relationships. *Characteristics:* roles and relationships clear, team begins to trust each other,

commitment to team goals forms[170][171]. Team starts to "gel".
4. **Performing:** Team is now a well-oiled machine. *Characteristics:* high trust, autonomous, focuses on achieving goals, handles conflicts constructively, makes decisions collaboratively with little supervision[172][173].
*(5. Adjourning:* for completeness, when team disbands after project, but as note says "not applicable to us" if we focus on ongoing teams[174].)

Recognizing these stages can help leaders know when to intervene (e.g. in Storming, facilitate conflict resolution; in Forming, provide clear direction). Agile teams aim to reach Performing quickly, but it takes time and perhaps facilitation from Scrum Master or team lead.

**Productive vs Unproductive Conflict:** *Productive conflict* means team members feel safe to disagree and debate ideas, leading to better solutions (diverse perspectives)[175][164]. *Unproductive conflict* is personal, hostile, or constant without resolution – it hurts morale and progress. Team dynamics should be managed to encourage healthy debate but prevent destructive fights. Techniques include setting team working agreements, conflict resolution training, or having retrospectives to air grievances constructively.

**Identifying and Resolving Team Issues:** If a team is underperforming or unhappy, possible causes are myriad (mismatched personalities, unclear roles, external stress, poor leadership). The lecture suggests investigating root causes through **confidential interviews or open retrospectives**[176][177]. Once causes are known, improve dynamics via: - Changes in team composition or role definitions if needed. - Team-building exercises or workshops (to improve trust, communication styles)[178][179]. - Addressing work environment issues (maybe co-locate the team, change office layout to increase communication)[178]. - Training on cultural differences if a distributed/multicultural team. - **Agile retrospectives** each sprint provide a forum to discuss what's working/not working in team collaboration and to implement improvements continuously.

**Agile Teams and Skills Mix:** Agile myth is that only "superstar" developers can succeed. In reality, a mix of skills and experience levels can work well if managed. It's suggested that every team should have at least one strong **lead/mentor** ("master") and then up to 4–5 less experienced developers per experienced one[180]. This mix allows knowledge sharing and growth while maintaining quality (the experienced members ensure good practices). The quote highlights that with a good skill mix and agile discipline, teams can succeed repeatedly[181].

**Team Culture (Dev vs Ops vs Test):** Some scenarios illustrate bad culture: e.g., a developer saying "It works in dev, ops must have messed up production" or a tester saying "It passed UAT, any failure in prod must be a config issue"[182]. These show lack of shared responsibility – each role deflects issues to others. DevOps culture (mentioned in Week 6) tries to eliminate this by making it a *team* responsibility for the software working in production. Solutions: cross-functional teams (developers, testers, ops in one team), blameless post-mortems (focus on fixing system, not blaming individuals), and fostering empathy across roles.

**Key Exam Points:** Be prepared to answer what **team dynamics** means and factors affecting it[162][167]. Recall **Tuckman's stages** (Forming, Storming, Norming, Performing – know characteristics of each)[183][169]. Perhaps know examples of good vs bad team dynamics or conflict. They may ask how agile addresses team collaboration (e.g. **self-organizing teams, daily stand-ups, retrospectives** help team dynamics). Know that **"throw it over the wall"** is bad and agile encourages cross-functional teamwork to avoid that[77]. Another focus could be the importance of **communication, trust, and conflict resolution** in teams. Possibly mention how a Scrum Master or team lead can help improve team dynamics (facilitating communication, removing impediments like interpersonal issues). Also, referencing that agile principle: "The best architectures and designs emerge from self-organizing teams"[76] suggests empowered teams perform better. So linking these concepts in an answer about team effectiveness would be wise.

## Week 8: Agile Methods – Scrum (Part 1)

**Scrum Overview:** Scrum is a popular **agile framework** for managing and executing complex projects. It structures development into fixed-length iterations called **Sprints** (typically 2–4 weeks) that produce a potentially shippable product increment[184][185]. Scrum defines specific **roles**, **events**, and **artifacts** to facilitate agile principles (iterative development, continuous feedback, self-organization).

**Scrum Roles:**
- **Product Owner (PO):** Represents the stakeholder/customer interests. Owns the **Product Backlog** (the prioritized list of work) and ensures the team is building the most valuable features first. They define user stories, set priorities, and accept completed work. They balance scope vs. value and are the "voice of the customer".
- **Development Team:** Cross-functional group (3–9 people typically) that builds the product. *Self-organizing*: they decide how to accomplish work each Sprint[186]. *Cross-functional*: collectively they have all skills needed (analysis, programming, testing, etc.)[187]. No sub-teams or specialized roles inside the Dev Team – everyone contributes to completing Sprint goals. They are accountable as a whole for the increment. Scrum avoids titles within the Dev Team to encourage shared ownership[188]. - **Scrum Master (SM):** Servant-leader and coach for the team and organization. Ensures Scrum is understood and enacted correctly[189]. Responsibilities: remove impediments blocking the team, facilitate Scrum events, coach the team in self-organization, shield the team from external disruptions, and help improve processes. The Scrum Master also serves the Product Owner by helping with backlog grooming and ensuring the PO and team communicate effectively[190][191]. They are not a traditional project manager (no authority over team in terms of what work to do), but rather a process coach and facilitator.

**Scrum Events (Ceremonies):** Scrum prescribes a set of meetings to provide regular cadence and feedback: - **Sprint:** The core of Scrum is the Sprint – a timeboxed iteration (max one month, commonly 2 weeks) in which a "Done", usable increment is developed[184][192]. Sprints occur back-to-back with no gaps. Scope may be clarified or re-negotiated during Sprint, but **no outside additions**: once a Sprint starts, requirements

are frozen for that Sprint (if new work emerges it goes to backlog for future Sprint)[193]. This protects the team's focus. Each Sprint should produce something of value (potentially shippable). - **Sprint Planning:** Kicks off the Sprint. Timebox: max 8 hours for a one-month Sprint (proportionally less for shorter Sprints, e.g. ~4 hours for 2-week sprint)[194][195]. Attendees: Product Owner, Scrum Master, Development Team. *Part 1:* **What** – PO presents the top Product Backlog items (stories) and their business value; team selects which items to commit to based on priority and their capacity[196][197]. They craft a **Sprint Goal** – a short statement of the Sprint's objective. *Part 2:* **How** – For each selected Product Backlog item (story), the Dev Team discusses and breaks it into **tasks** (often tracked on a Sprint Backlog or task board)[198][199]. The team may estimate tasks and ensure they haven't overcommitted. Output: **Sprint Backlog** (the selected stories + the plan to deliver them)[196][200]. The team forecasts what they can deliver; PO must trust the team's judgment of capacity. - **Daily Scrum (Stand-up):** 15-minute daily meeting for Dev Team (Scrum Master and PO can attend as observers or participants if team wants). Same time and place each day[201][202]. Purpose: synchronize and surface issues. Each member typically answers: **Yesterday** what did I do toward the Sprint Goal? **Today** what will I do? **Any blockers?**[203]. It is not a status report to a manager but a coordination meeting for the team. If problems are raised, solutions are discussed *after* the stand-up (often team members will have follow-up meetings for problem-solving). The Scrum Master ensures the daily scrum happens and stays within timebox, but the team is responsible for running it. Importantly, this meeting makes impediments visible daily so they can be addressed promptly[201][204]. - **Sprint Review:** Held at the end of the Sprint (timeboxed ~2–4 hours for monthly sprint). The team **demonstrates the working product increment** to stakeholders (could include PO, users, management – anyone interested)[205][206]. Only completed ("Done") items are shown – partially done work is not demonstrated as "done"[207][206]. The Product Owner formally accepts backlog items that meet the **Definition of Done**. Stakeholders give feedback – this is a chance to inspect the product and adapt the backlog (e.g. new ideas, changed requirements might be added to backlog)[208][209]. The Sprint Review is more informal than a stage-gate; it's a working session to adapt the product roadmap based on the increment. Outcome: a revised Product Backlog with potential new items or changed priorities for future Sprints[209][210]. - **Sprint Retrospective:** After the review (typically same day or next), the team (Dev Team, Scrum Master, optionally Product Owner) reflects on the **process** and **team dynamics**. Timeboxed ~1.5 hours for a 2-week sprint (3 hours for month sprint). Purpose: inspect how the Sprint went in terms of people, processes, tools, collaboration, and **identify improvements**[211][212]. Typical questions: *What went well? What could be improved? What will we commit to improve in the next Sprint?*[213][214]. The team might discuss things like "We had trouble with branch merging – let's try all integrating twice a week" or "Our testing lagged – maybe pair a tester earlier on each story." The Scrum Master ensures the retrospective is positive and productive (and reminds everyone of the Prime Directive: assume everyone did the best they could given the circumstances[215]). At least one concrete improvement is added to the next Sprint's backlog (often as a non-functional improvement item)[216][217].

**Scrum Artifacts:**
- **Product Backlog:** The single, ordered list of everything desired in the product (features, enhancements, bug fixes, technical work, etc.)[218][219]. Managed by the Product Owner, it is **dynamic** – constantly refined and re-prioritized as new information emerges[219][220]. Each item typically is written as a **User Story** (a brief description of a feature from end-user perspective) and has an estimate (often in **Story Points** – see Week 10) and a priority. The backlog is ordered by *value*, *risk*, *priority*, and/or *dependencies* – highest value items that are ready go to the top[219][221]. It's never "complete" until the product is done; it evolves. **Key exam points:** The Product Backlog is the *source of requirements* and only the PO can accept changes to it, though the Dev Team and stakeholders contribute ideas[222][219]. - **Sprint Backlog:** The list of Product Backlog items (stories) selected for the current Sprint **plus** the plan (tasks) to implement them[223]. It's essentially a subset of the Product Backlog that the team commits to for the Sprint, typically visualized on a **task board** (with columns like To Do / In Progress / Done)[224]. The Dev Team owns the Sprint Backlog; during the Sprint they update it daily (e.g., moving tasks on the board, adding new tasks as they discover work, updating remaining time estimates)[225][226]. It provides **visibility**: anyone can see what the team is working on. The Sprint Backlog can include at least one improvement action from last retrospective (to continuously improve)[225][226]. If some tasks are finished early, the team may collaborate with PO to pull in another small backlog item if possible (but only if it doesn't jeopardize the Sprint Goal)[227][228]. - **Increment:** The potentially shippable product increment at Sprint's end – i.e. the sum of all completed Product Backlog items in the Sprint **integrated with previous increments**. Each Increment must meet the **Definition of Done (DoD)** – a team's agreement on what it means for work to be complete (e.g., "coded, tested (unit + integration), documented, and deployed to staging")[229][230]. If something isn't done per this DoD, it cannot be counted in the increment. Over multiple Sprints, increments cumulate to the full product. The Scrum Team delivers *Increments* iteratively, getting feedback at reviews. The **Definition of Done** is crucial – it ensures quality is not deferred (for example, not saying "we'll write tests later" – if tests are part of DoD, the feature isn't done until tests are written and passing)[231][232].

**Scrum Process Flow (Summary):** At the start of Sprint, Sprint Planning selects items -> During Sprint, Daily Scrums + development happen -> End of Sprint deliver an increment -> Sprint Review (demo increment) -> Retrospective (process improvements) -> next Sprint. The cycle repeats, delivering incremental value.

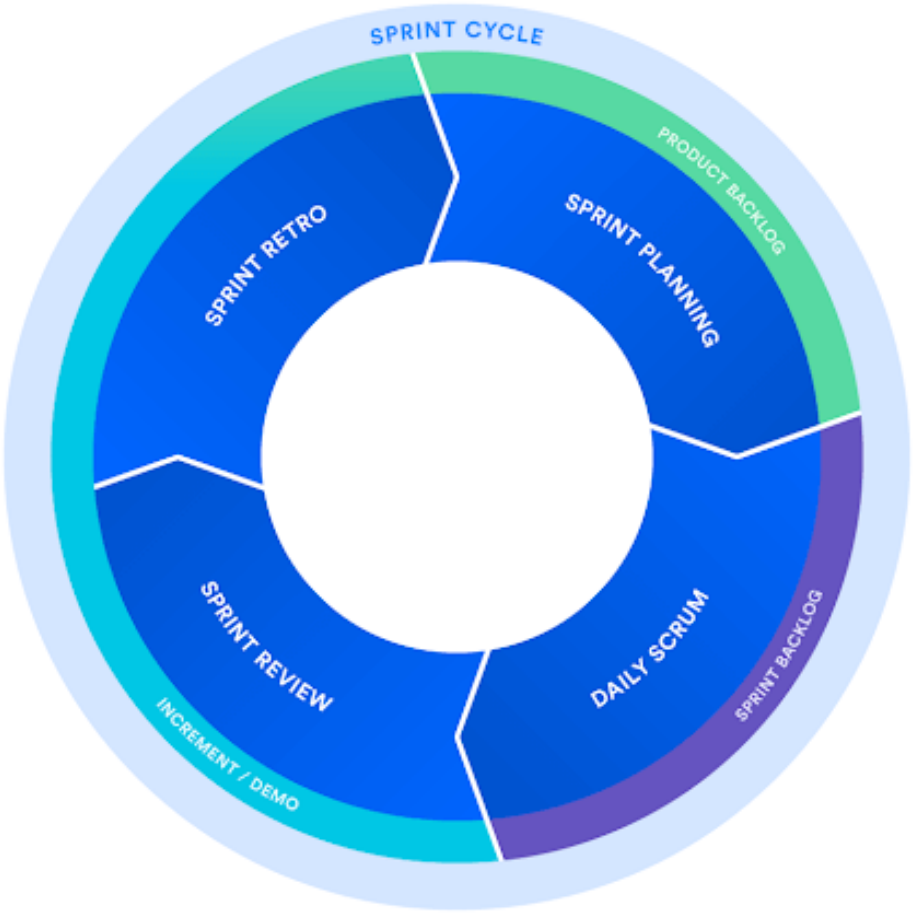


*Figure: Scrum Sprint Cycle – each Sprint begins with Sprint Planning (using the Product Backlog to choose Sprint Backlog items), includes daily Scrums during execution, and ends with a Sprint Review (increment/demo) and Sprint Retrospective. The cycle then repeats, leveraging feedback and continuous improvement.*[194][201]

**Key Scrum Concepts to Highlight:**
- **Self-Organizing Teams:** They decide how to get work done – no micromanagement. This often appears as a question of how Scrum teams differ from traditional teams. Key point: there is **no project manager** assigning tasks in Scrum; the team volunteers for tasks and manages itself.
- **Timeboxing:** All events are time-constrained to keep them efficient (e.g., daily scrum 15 min).
- **Sprint Goal & Flexibility:** The Sprint Goal provides focus. If some scope needs adjustment mid-sprint, team can negotiate with PO *but* goal shouldn't change. If a major change is needed (e.g., requirement change that makes Sprint Goal obsolete), Scrum says

consider terminating the sprint and replanning.
- **Burndown Chart:** Scrum often uses a **Sprint Burndown Chart** to track remaining work during the Sprint. It's a daily updated graph of remaining tasks or effort vs. days. A steady burn-down indicates progress; a flat line indicates no progress (possible impediment). (This is sometimes considered an artifact for monitoring progress.) The **"Burndown" chart** is explicitly mentioned in lectures as a progress monitoring tool[232][233]. It tracks remaining effort and is updated daily by the Scrum Master or team. *Velocity* (see Week 10) is measured across Sprints to help forecast future capacity[234][235].

**Exam Tip:** Know **Scrum roles, events, artifacts** thoroughly. For example, a typical question might be: *"What happens in a Sprint Review versus a Retrospective?"* – you'd say review is product-focused (demo and adapt product backlog)[206][208], retrospective is process-focused (how to improve teamwork)[213][214]. Or *"Who is responsible for X in Scrum?"* – e.g., Product Owner owns backlog prioritization[218][219], Scrum Master owns process and removing impediments[189][191], Dev Team owns estimates and delivery. Understand **Definition of Done** and its importance[229][236]. Also important: **Scrum values** (commitment, courage, focus, openness, respect) which weren't explicitly listed above but are part of Scrum Guide – however, focus likely on mechanics given lecture content. Since the user explicitly mentioned **Scrum, Story Point, Burndown Chart, Git Rebase** as key terms: we've explained Scrum events and artifacts (story points & rebase come in Week 10 and Week 3 respectively). Here ensure you can describe a **Burndown Chart** usage: it's a daily chart showing remaining work vs time, updated by Scrum Master, ensuring transparency of progress[232][233].

## Week 9: Agile Methods – Scrum (Part 2: Agile Requirements & Execution)

In this week, focus shifts to **Agile requirements handling** and practical Scrum techniques during a Sprint. Key topics include **User Stories, Task Boards, and Agile modeling (lo-fi UI, storyboards).**

**User Stories:** Agile teams capture requirements as **User Stories** – short, simple descriptions of a feature told from an end-user perspective. A common template: *"As a [user role], I want [goal] so that [reason]."* For example: *"As a movie fan, I want to add a movie to the database so that I can share it with others."*[237][238]. The lecture example given: Feature "Add a movie" was written in that form[237].

User stories emphasize *who* wants it, *what* they want, and *why* – keeping the value in focus. They are usually small pieces of functionality that can be built within one Sprint. Larger features are split into multiple stories or into an **Epic** (a big story that spans sprints, later broken down)[239][240].

**Definition of Done for User Stories:** The team should agree when a story is considered "done." This often involves writing **Acceptance Criteria** – conditions of satisfaction for the story. For the movie example, acceptance criteria might be: - User can input all required fields (title, rating, etc.) and save. - After saving, the new movie appears in the listing. - If any required field is missing, an error message is shown (just hypothetical).

These criteria ensure developers and PO agree on what needs to happen for the story to be accepted[241][242]. Acceptance criteria can be written in **Given-When-Then** format (a form of BDD – Behavior-Driven Development scenarios): e.g., *GIVEN I am on the "Add Movie" page, WHEN I fill in all fields and click Save, THEN the movie is added to the list*.[243][244]. This format was shown in lecture as scenario format for test cases (given/when/then is common for acceptance tests)[245][246].

**Breaking Down Stories into Tasks:** During Sprint Planning Part 2, the team breaks each selected user story into **tasks**[247][248]. For example, tasks for "Add movie" might include: create UI form, client-side validation, server-side API endpoint, database schema change, write unit tests, etc. Each task is typically a day or less of work, and team members sign up to do them. This breakdown is often captured on the **Task Board** (or in an agile tool like Jira). As tasks progress, they move from "To Do" -> "In Progress" -> "Done" columns[249][250]. The board provides at-a-glance status. A point noted: team members should update the board frequently, and they can add new tasks if needed to complete a story (not all tasks are known upfront)[251][252].

**Sprint Execution & Task Board Usage:** During the Sprint, each developer picks tasks, moves them to in-progress, and eventually to done when finished[253][254]. If work is blocked or a story is bigger than thought, they communicate in Daily Scrum. The board may also include a **burndown chart** or remaining story points indicator to track progress (discussed in Week 8 and 10). Team members might swarm together on one story if another is finished (Scrum encourages teamwork, not just parallel silo work).

**Unfinished Stories:** If a story isn't completed by end of Sprint (not meeting DoD), it cannot be counted as done or shown in review. It usually goes back to Product Backlog (possibly to be re-estimated and reprioritized). Scrum emphasizes either done or not done – no partial credit. Teams aim to avoid this by right-sizing stories and collaborating.

**Agile Modeling: Lo-Fi UI & Storyboards:** The lecture covers creating quick design artifacts:
- **Low-Fidelity (Lo-Fi) UI Sketches:** Simple hand-drawn screens or wireframes to illustrate user interfaces and flows[255][256]. They are cheap, quick to make, and help clarify requirements with stakeholders early. Lo-fi prototypes (paper sketches) involve minimal effort but can capture the essence of user interactions and get feedback (especially useful to engage non-technical stakeholders in discussion)[257][258].
- **High-Fidelity (Hi-Fi) UI Prototypes:** More detailed, closer to actual design (could be done in software). These are used when needed for documentation or to nail down specifics, but they take more time[259][260]. Agile favors starting from lo-fi to iterate quickly, and only moving to hi-fi when the basic design is validated. - **Scenarios and Storyboards:** A **Scenario** is a narrative describing how a user accomplishes a task in the system (step-by-step from user's perspective)[261]. A **Storyboard** is a visual sequence of panels (like comic strips) showing a user's interactions step by step[262][263]. These help in understanding the context and flow, especially for UI/UX or complex sequences. For each user story, you could have one or more scenarios illustrating different usage contexts. The lecture shows scenario format: *Scenario: [brief description]; GIVEN [context] WHEN [event]*

*THEN [outcome]* (with possibly multiple GIVEN/WHEN/THEN lines for complex scenarios)[245][264]. This format is essentially the Gherkin syntax used in BDD (which ties into acceptance criteria and automated acceptance tests). - The example given: *Feature: User can manually add a movie; Scenario: Add a movie; GIVEN I am on home page, WHEN I follow "Add new movie", THEN I see a form*... etc.[265].

**Story Refinement & SMART Criteria:** Though not explicitly in snippet, agile teams ensure stories are *"INVEST"* (Independent, Negotiable, Valuable, Estimable, Small, Testable) or *SMART*. The content mentions *SMART user stories and 5 Whys*[266] – likely referring to making sure stories are Specific, Measurable, Achievable, Relevant, Time-bound (or similar interpretations). The "Five Whys" exercise example in snippet shows questioning the real business need behind a feature repeatedly until the true value is revealed[267][268]. In that example, "Why add Facebook link? -> to enjoy show more -> sell more tickets -> theater makes more money -> not go out of business -> I keep my job." It uncovers the *real motivation* (keeping business afloat) which might help decide if feature is truly needed. The point is to ensure features deliver real business value, not just "cool" additions with unclear ROI.

**Sprint Planning Revisited:** The lecture revisits Sprint Planning to emphasize value-based selection (PO brings prioritized list) and team estimating effort to commit properly[269][270]. The PO must prepare by **backlog grooming** (ensuring top items are clear and estimated). If velocity is known, they select roughly that many points. Part 2 output is tasks on a Task Board (which we covered)[249].

**During the Sprint:** Communication is key. The **Daily Scrum** is already covered (Week 8). The team should communicate any added tasks or changes. Scrum expects transparency – if a team member finds they can't finish a story, they alert and team swarms or scope might be re-negotiated early (but mid-sprint scope change is discouraged unless absolutely necessary).

**Definition of Done (DoD) and Quality:** Teams often include quality tasks in their DoD (e.g., "code reviewed" or "unit tests written"). In agile, testing is not a separate phase – testers (if any) are part of team and test stories within the sprint. Possibly the mention "set-up a better build server" or "improve design principles" as retrospective actions[216] shows that even improving tools and practices can be backlog items.

**Key Exam Topics:** This week gets into nitty-gritty: - **User Story format and purpose**[237]. Be able to write one and explain components (role, goal, benefit). Possibly know what an **Epic** is vs a story[239]. - **Acceptance Criteria & Definition of Done:** Why they're important (ensures clarity on when a story is truly complete)[241]. - **Task breakdown and Sprint execution:** How a team uses a Task Board (Kanban board style) to track work in a Sprint[249][254]. - **Agile modeling techniques:** Lo-fi prototyping, storyboards, scenario writing (Given-When-Then) – useful to discuss how agile teams handle requirements and design in a lightweight way before/during development[245][264]. - Possibly **"What are test doubles or fakes"** might appear but more likely they stick to scrum context. If anything from Week 5, maybe just to mention that agile teams use TDD or unit tests as part

of story completion (ensuring quality). - Understand **why agile favors incremental refinement:** e.g., starting with simple sketches rather than full specifications (this ties to responding to change and involving user feedback earlier).

Also, since **Story Points** are a big topic (they come in Week 10, but maybe introduced as estimation units by week 9): be aware that the team estimates story sizes in Story Points (an abstract unit relative to a baseline story). Week 10 goes deeper on that, but a quick note: story points measure *complexity/effort*, not directly days, and allow comparing stories. The velocity (points done per sprint) helps planning (see next week).

## Week 10: Agile Methods – Scrum (Part 3: Estimation and Planning)

This week dives into **Agile estimation (Story Points, Planning Poker)** and how to use **velocity** for release planning. Agile avoids rigid, upfront estimates; instead it relies on relative estimation and continuous re-planning.

**Story Points:** A Story Point is a unit of measure for the **relative size of a user story** – encompassing complexity, effort, and risk. Teams assign point values (often from a Fibonacci-like sequence: 1, 2, 3, 5, 8, 13, ...) to stories. Important: *Story points are relative*, not tied to a specific time. For example, a 2-point story is about twice the effort of a 1-point story, and a 5-point story is a bit more than two 2-point stories, etc. Story points separate **effort estimation from duration estimation**[271] – the team estimates effort in points without immediately worrying about how long in calendar time, which can reduce anchoring to hours and accounts for different productivity factors.

**Calibrating Story Points:** Teams typically establish a baseline story (e.g., "the simplest story we can think of = 1 point")[272][273]. Then they estimate others relative to that. Alternatively, sometimes they classify a small story as 2 points (so nothing is 1, to give room for smaller than baseline), but the idea is consistency. Two approaches mentioned: - Small expected story = 1 point; medium story maybe ~5; larger ones accordingly[272][273]. - Or pick a few reference stories of 1, 3, 5 to guide others.

**Why Story Points (benefits):**[274][275] - They are **simple** and teams control the scale (they're basically an abstraction). - Developers often find it easier to compare sizes than to guess exact hours, so it "gets team talking" about the work in a meaningful way[274][276]. - Avoids **precision fallacy** of hour estimates – when using days/hours, people might wrongly assign more certainty. Points highlight it's an estimate of size, not a promise of time. - Points encourage **team estimation** – everyone contributes perspective, leading to shared understanding of work (like "this story touches database and UI, so maybe more points"). - They help with **commitment**: if the team agrees on points and then sees their historical velocity, they can commit to a realistic amount of work. Also, developers are less intimidated by point estimates vs hours, reducing fear of being "wrong" – which fosters openness.

**Ideal Days vs Story Points:** *Ideal days* means "how many days of work if one person worked on it with no interruptions." *Elapsed days* is real calendar time (with distractions,

multi-tasking, etc.)[277][278]. Often, especially novice teams, think in terms of ideal days. However: - Ideal days are easier to explain externally ("this story is about 2 ideal days of work")[279]. - But each person's ideal day is different (experience, context switching, etc.) – so story points avoid that by normalizing differences ("my 1 point is your 1 point, regardless of skill, because it's relative")[280][281]. - Ideal days might be easier when starting, but story points are usually faster after the team calibrates[280][282]. - Also, velocity using ideal days is directly translatable to time, which some prefer for forecasting, but it may give a false sense of precision (and managers might treat it as commitment of actual days).

In practice, some teams do estimate in ideal days implicitly, then convert to points – but exam focus likely on the conceptual differences.

**Planning Poker (Estimation Technique):** A consensus-based estimation technique: - Each estimator (developer) has a set of cards with numbers (points). - For a given user story, team discusses briefly, then each member privately selects a card representing their estimate and reveals simultaneously[283][284]. - If everyone agrees, that's the estimate. If not, discuss differences: usually the high and low estimators explain their reasoning, then re-vote[285][286]. - Repeat until consensus or narrow agreement is reached. Planning Poker combines expert opinion, analogy (comparing to known stories), and disaggregation (if a story is too big, break it down) – and it's gamified/fun[287][288]. It helps avoid anchoring (because everyone shows at once, no one influences others by speaking first)[289][285]. Mentioned source: Mike Cohn popularized it[290].

**Story Points vs Ideal Days Recap:** The lecture lists reasons: - Story points promote cross-functional behavior (everyone estimates, not just one specialty)[280][282]. - They are pure size measure, independent of who works on it (if Bob is slower, velocity accounts for that rather than different point values). - Estimating in points is often faster once team gets it, and avoids the "my ideal day != your ideal day" problem[280][281]. - However, ideal days are easier for outsiders to grasp and easier when team first starts (since we all think in days naturally)[279]. Some teams use both: estimate in points but roughly map them to ideal days internally for sanity check (e.g., "1 point ~ 1 ideal day"). The exam might expect knowledge that points are preferred in agile, but ideal days have pros/cons.

**Velocity:** The rate at which the team delivers value, measured in story points per Sprint[291][292]. For example, if in Sprint 1 team completed stories worth 20 points total, velocity = 20. Over a few sprints, velocity stabilizes (e.g., team consistently doing ~20-25 points). Velocity is used for **forecasting**: - **Capacity Planning:** If velocity is ~20, team shouldn't plan 40 points next sprint – they'll likely over-commit. It helps decide how many backlog items to take into Sprint (commitment). - **Release Planning:** If product backlog has 100 points of work left and velocity is 10 points/sprint, roughly 10 sprints remain[293][294]. Or if a deadline is fixed (say 5 sprints left in project), multiply velocity by 5 to see how many points likely done by then; compare to backlog size to see if scope needs trimming.

**Estimating Project Duration:** Sum of all backlog story points / velocity = number of sprints needed[293][294]. E.g., backlog 100 SP, velocity 11 SP/2-week Sprint => ~9.1 Sprints, so about 10 Sprints (20 weeks)[295][296]. The lecture explicitly did such an example[297][298].

**Re-estimation and Refinement:** As team learns, they might re-estimate some backlog items if understanding changes (e.g. a story initially thought 8 points might be split into two 5s or downgraded to 3 after research)[299][300]. But generally, avoid constant re-estimation just because velocity was off – velocity itself will adjust for under/over estimates[301][302]. Rule: *re-estimate if relative size understanding changed, not just because of performance issues*. If progress is slower than expected, don't inflate points of remaining stories; instead velocity accounts for it and schedule can be recalculated. Only re-estimate if a story's scope changed or initial estimate clearly off relative to others[301][302].

**Tracking Progress with Velocity:** - **Burn-down Chart:** (already touched in Week 8) – within Sprint, tracks remaining work down to 0 ideally by end[232][233]. - **Burn-up Chart:** Could be used across Sprints to show cumulative progress vs total scope (especially as scope can change). Not sure if covered, but likely not deeply. - **Cumulative Flow Diagram:** Possibly not covered explicitly, but it's another agile tracking graph (likely not on exam if not mentioned).

**Resourcing and Velocity:** If team composition changes, velocity changes. Also one team's points not comparable to another's (points are relative to each team's scale). Good to mention if asked.

**Exercise Example:** They might have given the class an exercise: e.g., sum all story points to see how many Sprints given a velocity (like the example given, 100 SP, velocity 11 SP/2w -> ~9 iterations ≈ 20 weeks)[303][304]. Or they might ask to compute velocity given data (e.g., Sprint 1 did X points, Sprint 2 Y, so average velocity?).

**Planning Poker specifics:** The origin (Grenning 2002, Cohn's book 2006) was referenced[290], but exam likely more on process: how does it avoid bias (simultaneous reveal)[289], incorporate expert opinion (everyone speaks), etc.

**Continuous Improvement of Estimates:** As team delivers, velocity may improve or change. Agile allows updating forecasts each Sprint with new velocity data. If velocity is inconsistent, often use the average of last few Sprints or a range (e.g. "velocity 20 ± 5").

**When Not to Use Story Points:** Possibly mention – points are great for new development but maybe less for support work or research tasks (some teams then use time or have separate capacity). But probably not needed in exam unless tricky question.

**Key Exam Points:** Expect questions like: - *Explain what a Story Point is and why agile teams use them instead of days.*[274][280] - *What is Planning Poker and how does it work?*[283][289] - *Define velocity and how it's used in planning releases.*[291][293] - *Given a backlog size and velocity, calculate approximate iterations needed.* (Basic math). -

Possibly *pros/cons of story points vs ideal days*.[280][279] - *When should you re-estimate stories?* (Answer: only when understanding of relative size has changed significantly, not just because you underperformed – let velocity handle performance issues)[301][302].

Also know the difference between **Continuous Delivery vs Deployment** (from Week 6, but might tie in release planning context – however, likely in CI/CD question).

Link to **Burndown Chart & Velocity:** The sprint burndown shows progress within sprint; velocity is measured after sprint. Also mention **"Don't overcommit – use velocity as a guide."**

Finally, **Planning in Agile vs Waterfall:** Agile planning is incremental – rather than a full Gantt chart upfront, agile does high-level estimates with velocity and then continuously reprioritizes and re-estimates each Sprint. A possible exam scenario: *"If management asks for an exact delivery date for all features, how does Scrum handle that?"* – Answer: Scrum provides an **estimate** based on velocity and backlog size, but emphasizes it's adaptable. As project proceeds, refine timeline continuously. If necessary, adjust scope or resources to hit fixed date (the iron triangle – agile usually fixes time & resources, and flexes scope).

## Week 12: Ethics, Intellectual Property, and Open-Source Software

(This was Week 12 since Week 11 is skipped as per instructions.) This topic addresses professional responsibilities of software engineers, intellectual property rights, and using/opening software source code.

**Software Engineering Ethics:** Professionals must adhere to ethical principles beyond just coding. Major professional bodies like **ACM**, **IEEE Computer Society**, and **ACS (Australian Computer Society)** have codes of ethics or conduct: - The **ACM Code of Ethics (2018)** outlines **General Ethical Principles**: e.g., *public good*, *no harm*, *honesty*, *fairness*, *respect for privacy*, *confidentiality*[305]. It also details **Professional Responsibilities**: e.g., strive for high quality, maintain competence, know and follow laws, respect intellectual property, accept and provide appropriate review, give comprehensive evaluations of systems including risks, perform only in your area of competence, etc.[306][307]. And **Leadership Principles** for those in leadership roles: promote ethical approach, ensure sustainable development, consider social impacts of systems, etc.[308][309]. - The **IEEE-CS/ACM Software Engineering Code of Ethics (SE Code)** (from late 90s) has 8 Principles: *Public, Client & Employer, Product, Judgment, Management, Profession, Colleagues, Self*[310][311]. For example, Software Engineers shall act in public interest, in best interest of client/employer consistent with public interest, ensure products meet highest standards, maintain integrity and independence in judgment, promote ethical management, advance the profession, be fair to colleagues, and pursue lifelong learning[312][313]. - **ACS Code of Professional Conduct** lists core values: Primacy of Public Interest, Enhancement of Quality of Life, Honesty, Competence, Professional Development, Professionalism[314][315]. Similar spirit: put public interest first, ensure your work benefits society, be honest about your skills and work, continuously improve your skills, etc.

**Ethical Scenarios:** The lecture scenario: A consultant finds a way to finish a project in 2 months for $1M instead of contracted 1 year $6M. Boss says hide this to keep revenue[316][317]. This pits honesty and public interest vs employer interest. Ethically, the engineer should consider the client's interest and the honesty principle. The code says to act in best interests of client and public – here, continuing unnecessarily for profit violates those. It's a dilemma: if they reveal the cheaper solution, the company might lose revenue or staff may be let go; if they hide it, client overpays and resources wasted. The code would lean toward disclosing the more efficient solution to the client (public and client interest over personal/business interest). Expect to possibly discuss conflict of interest, whistleblowing, or following the code in such dilemmas.

Another angle: **Software failures and liability.** Cases like Therac-25 (radiation machine overdose due to software error causing patient deaths) highlight ethical responsibility. If you're aware of a flaw, you must act (cannot just hide it). **Who is responsible** if software causes harm? Programmers? Testers? Company? The lecture asks "who is ethically responsible for damage caused by faults?"[318][319]. Ultimately, licensed professional engineers (in fields like civil, or if software is under engineering licensure) carry responsibility to protect public safety. In software, it's often collective: the organization delivering the software is responsible, but individual engineers should uphold standards to prevent harm. The cited text from an engineering association suggests minimizing risk of failure is a duty to public interest[320][321].

**Privacy and Data Ethics:** Likely touched on: ensure user data privacy, guard confidentiality. E.g., not divulging sensitive data or violating privacy laws (like GDPR). The ACM principles explicitly mention privacy and confidentiality[305][322].

**Intellectual Property (IP):** Four main types relevant to software: - **Copyright:** Legal right of the creator of original works (literature, art, software code) to control copying, distribution, derivative works. In software, code is automatically copyrighted by the author. Copyright typically lasts many years (life of author + 70 years, or for works for hire, 95 years from publication in US). Copyright covers expression, not ideas/algorithms. Software licenses operate as copyright licenses (owner grants certain permissions). *Key point:* If you write code, you (or your employer if done at work) own the copyright unless you explicitly license it or assign it. - **Patents:** Protect inventions (including software algorithms or techniques in some jurisdictions) by granting exclusive rights for ~20 years, in exchange for publishing the invention. Software patents exist (especially in US) but are controversial (many countries exclude pure software or require a technical effect). Patents require novelty and non-obviousness. If you patent a software algorithm, others cannot use that algorithm implementation without permission (or they risk infringement suits). *Exam note:* Patent vs copyright: copyright protects code (expression), patent protects ideas/functional concepts. Also mention *patent grants are needed in open source licenses* – e.g. Apache 2.0, GPLv3 include explicit patent usage rights from contributors[323][324]. - **Trade Secrets:** Information (like source code, algorithms, customer lists) kept secret by a company to maintain advantage. Protected by confidentiality, NDAs – no registration required, but once secret is out, protection is lost. Example: Coca-Cola formula. For software, if code

isn't released and only distributed in binary, it can be a trade secret. Engineers must not steal trade secrets from employers or share them improperly. - **Trademarks:** Protect brand names, logos, etc. Less directly software code, but relevant for naming software or companies. For example, "Java" is a trademark of Oracle; you can't name your product "Java Something" without permission. Open source projects sometimes rename if trademark issues (e.g. Debian had to rename Firefox to Iceweasel at one point due to trademark).

**Open-Source Software (OSS):** Software where source code is made available under a license permitting use, modification, and distribution. OSS fosters collaboration and re-use. However, using open-source in projects requires understanding its license terms to comply.

**Open Source Licenses:** There are various licenses with different conditions. Two broad categories: - **Permissive Licenses:** e.g. MIT, Apache, BSD. These allow you to use the code in proprietary projects with minimal requirements. - **MIT License:** Very short and permissive. Essentially says *"do whatever you want with the software; include my copyright and license notice; no liability or warranty from me"*[325][326]. It does not require sharing modifications. Many libraries use MIT. - **Apache License 2.0:** Permissive but with protections. Allows use/distribution/modification, but you must include original copyright & license notices, and if you distribute binaries you must reproduce the license. Also, it has an explicit **patent grant** – contributors grant users rights to any patents covering the software[327][328]. It also says if you sue someone for patent infringement on this software, you lose your license (patent retaliation clause). Apache 2.0 allows distributing modified code under different terms (you can even make it closed source) as long as you keep notices and state changes, etc.[327][329]. - **BSD License (2-clause, 3-clause):** Similar to MIT: minimal requirements (keep copyright notice) and disclaimer of liability. 3-clause adds non-endorsement clause (don't use the author's name to endorse derived products without permission). - **Mozilla Public License (MPL) 2.0:** A weak copyleft – file-level copyleft. If you modify an MPL-licensed file, you must open source that file under MPL when distributing[113][330]. But you can use MPL code in a larger project and only files derived from MPL ones need to be MPL; you can combine with proprietary files. So it's between permissive and strong copyleft. - **Copyleft (Strong) Licenses:** e.g. GNU GPL (General Public License). These require that if you distribute modified versions or anything that *derives* from the GPL code, you must release the source under the **same license (GPL)**, thus "copyleft" ensures software (and derivative works) remain free/open[324][331]. - **GPL v3:** If you distribute a program that contains GPL'd code or is derived from it, the entire program must be licensed as GPL v3 and accompanied by source code[324][331]. It also has patent grants like Apache (contributors grant patent rights)[332][333]. It prohibits adding restrictions – you can't take a GPL program and make it proprietary. GPL is viral: linking (for software libraries, if statically linked or certain dynamic linking) can trigger the requirement that the combined work is GPL. (This is a complex area – but as a highlight, using a GPL library in your software can force your software to be GPL if distributed.) - **LGPL (Lesser GPL):** A weaker copyleft intended for libraries. It allows linking to the library in proprietary programs, as long as any modifications to the library itself are released under

LGPL/GPL[334][335]. So you can use LGPL libraries without open-sourcing your whole app, but if you modify the LGPL library or directly include its code in yours, those changes are LGPL. The excerpt shows LGPL requires source of modifications under LGPL or GPL[334][335]. - **AGPL (Affero GPL):** Like GPLv3 but closes the "SaaS loophole" – if you use the software on a server (and users interact with it over network), AGPL requires you to provide source to those users on request[336][323]. It ensures freedom even in web services usage.

- **"The Unlicense" and public domain:** The Unlicense attempts to dedicate code to public domain (no copyright). It basically says do whatever without attribution, and often includes explicit waiver of copyright[337][338]. It's the most permissive (though legally, public domain dedication may vary by jurisdiction). Example given: youtube-dl project uses Unlicense[338][339].

**Using Open Source in Projects:** Engineers need to comply with licenses. For instance, if you use GPL code in your application, you may be obligated to open source your whole app (which might be unacceptable in commercial context). So companies often use more permissive licensed components (MIT, Apache) to avoid contamination. Some will release modifications to open source they use out of good citizenship or obligation (depending on license).

**Contributing to Open Source:** If you contribute to an OSS project, you typically agree that your contributions are licensed under the project's license (some projects ask for explicit Contributor License Agreements). You must also respect the license when, say, including someone else's GPL code in your project – you can't just copy-paste code without following license terms (like including their copyright notice, licensing your code appropriately, etc.).

**Ethical Use of Code:** Plagiarism vs reuse – ethically, using open source is encouraged but you must attribute and comply. Taking code from StackOverflow or GitHub without checking license can be risky.

**Additional Ethics Topics:** - **Privacy:** If your software handles user data, you must follow privacy laws and ethical guidelines (e.g., don't collect more than needed, protect user data, be transparent). - **AI Ethics** (not sure if covered): bias, etc. - **Professionalism:** Don't falsify your work, don't over-promise on capabilities, continuous learning (the code of ethics point). - **Cybersecurity ethics:** e.g., responsibly disclosing vulnerabilities vs selling exploits.

**Key Exam Points:** They could ask: - Summarize key principles from a code of ethics (like IEEE or ACM)[310][311]. Possibly scenario-based questions: "What would you do in situation X per code of ethics?" (like the project scenario – answer using principles like public interest, honesty). - Differentiate copyright, patent, trade secret, trademark as applied to software. - Explain differences between open source licenses (MIT vs GPL for example): *MIT allows proprietary use with just attribution[325]; GPL requires open-sourcing derivative code* 【*45†L19-L27}*. Or *Apache vs GPL:* both grant patent rights[324][328], but

GPL is copyleft vs Apache permissive. - Possibly ask what license you'd choose for a project if you wanted it used widely (permissive like Apache or MIT), vs if you wanted to ensure improvements are shared (GPL). - Perhaps define copyleft or why GPL is called viral. - For open source usage: *"Can you use a GPLv3 library in a commercial closed-source application?"* – basically no, not without opening your code (unless it's LGPL or some carve-out). - They might mention **AGPL** if focusing on difference for network services[336]. - Also the importance of **including license notice** when using open source (common compliance step). - Ethical issues: *"Is it ethical to keep a software bug secret?"* (No, if it can harm – duty to report or fix for user safety). - Possibly mention famous cases: Therac-25, the Volkswagen emissions software scandal (where software cheated tests – unethical programming), etc., if time permits.

Be ready to reference the **ACM/IEEE principles** such as public interest, client & employer, product, etc., and how they guide behavior[310][311]. And emphasize that as a software professional, one must **speak up** when something is wrong (e.g. safety issue) – ties to whistleblowing obligations for public safety (though whistleblowing can conflict with loyalty to employer; codes typically favor public welfare in such conflict).

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [18] [19] [24] [25] [26] [27] [28] [29] [30] L1_ Introduction to Software Development Processes.pdf

file://file_0000000033dc72099b15fed2520bb078

[16] [17] SDLC V-Model - Software Engineering - GeeksforGeeks

https://www.geeksforgeeks.org/software-engineering/software-engineering-sdlc-v-model/

[20] [21] [22] [23] What is Spiral Model in Software Engineering? - GeeksforGeeks

https://www.geeksforgeeks.org/software-engineering/software-engineering-spiral-model/

[31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [340] L2_ Tools and Technologies for Controlling Artifacts.pdf

file://file_0000000005d072098fb6844444b9ee76

[58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] L3_ Tools and for Managing Collaborative Development Artifacts.pdf

file://file_0000000093bc72099c2a056d5f4b2db3

[76] [77] [162] [163] [164] [165] [166] [167] [168] [169] [170] [171] [172] [173] [174] [175] [176] [177] [178] [179] [180] [181] [182] [183] L7_ Team Dynamics.pdf

file://file_00000000b5e47209a63fd7a8448bda72

[78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [96] [97] [98] L4_ System Build Automation.pdf

file://file_000000004da07209b2ae8bb976050bea

[99] [100] [101] [102] [103] [104] [105] [106] [107] [108] [109] [110] [120] [121] [122] [123] [124] [125] [126] [127] [128] L5_ Software Verification.pdf

file://file_00000000322c7209a6465073a70927b4

[111] [112] [113] [114] [115] [116] [117] [118] [119] [129] [305] [306] [307] [308] [309] [310] [311] [312] [313] [314] [315] [316] [317] [318] [319] [320] [321] [322] [323] [324] [325] [326] [327] [328] [329] [330] [331] [332] [333] [334] [335] [336] [337] [338] [339] L12_ Ethics, Intellectual Property, and Open-Source Software.pdf

file://file_0000000076807209b3c34c5be5aad882

[130] [131] [132] [133] [134] [135] [136] [137] [138] [139] [140] [141] [142] [143] [144] [145] [146] [147] [148] [149] [150] [151] [152] [153] [154] [155] [156] [157] [158] [159] [160] [161] L6_ CI_CD.pdf

file://file_00000000def8720992d272ff76664b49

[184] [185] [186] [187] [188] [189] [190] [191] [192] [193] [194] [195] [196] [197] [198] [199] [200] [201] [202] [203] [204] [205] [206] [207] [208] [209] [210] [211] [212] [213] [214] [215] [216] [217] [218] [219] [220] [221] [222] [223] [224] [225] [226] [227] [228] [229] [230] [231] [232] [233] [234] [235] [236] L8_ Agile Methods Scrum.pdf

file://file_00000000bef87209881810c07ac8c118

[237] [238] [239] [240] [241] [242] [243] [244] [245] [246] [247] [248] [249] [250] [251] [252] [253] [254] [255] [256] [257] [258] [259] [260] [261] [262] [263] [264] [265] [266] [267] [268] [269] [270] L9_ Agile Methods Scrum (2).pdf

file://file_000000006f607209803a00cc680982a7

[271] [272] [273] [274] [275] [276] [277] [278] [279] [280] [281] [282] [283] [284] [285] [286] [287] [288] [289] [290] [291] [292] [293] [294] [295] [296] [297] [298] [299] [300] [301] [302] [303] [304] L10_ Agile Methods Scrum (3).pdf

file://file_0000000009f0720982eae6efe7091641