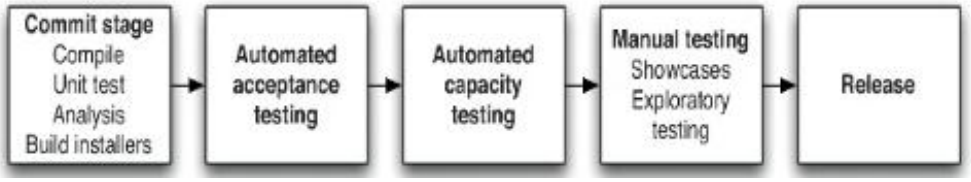
Study Point Assignment (10 SP): Continuous Integration

Objective

You must implement a **commit CI** for a repo that runs: lint, build, unit tests, and produces a primary artifact plus an SBOM.

The commit stage begins with a change to the state of the project—that is, a commit to the version control system. It ends with either a report of failure or, if successful, a collection of binary artifacts and deployable assemblies to be used in subsequent test and release stages, as well as reports on the state of the application. Ideally, a commit stage should take less than five minutes to run, and certainly no more than ten.

The commit stage represents the entrance into the deployment pipeline: It is the point at which a new release candidate is created (Jez Humble, Continuous Delivery 2010)**:**



The commit stage is the gatekeeper of your deployment pipeline. Every code change triggers it, and its job is simple but vital: Compile code, run fast tests, generate artifacts for later stages and analyze code health.

Your Tasks

You must:

- Implement a protype with some interesting business logic that can be unit tested.
- Build commit pipeline in GitHub Actions.
- Run build + tests + static analysis + test coverage.
- Make sure to make some violations that break things and watch CI respond (Checkstyle violation, failing unit test, low coverage, SpotBug issues)
- Produce JAR *or* Docker image and push to GitHub Container Registry (remember you don't need to use Java)
- Upload artifact + SBOM.
- Enforce branch protection
- Demonstrate green/red checks in PR (Make PRs with violation to block the merge, fix things and see merge allowed (= branch protection)

Hand-in

Tuesday 16/9 Mail to tm@ek.dk: Team names, GitHub link

Example

You can use this example repo as inspiration Tine-m/java-ci-template: Starter repo for Commit Stage CI assignment (Java + Maven + GitHub Actions).

Explanations about the repo:

In the **GitHub Actions workflow**, before building the Docker image, we have a step:

```
- name: Build, test, quality gates
  run: mvn -B -ntp verify
```

That runs:

- compilation
- unit tests (JUnit)
- JaCoCo coverage
- Checkstyle
- SpotBugs
- SBOM

Where to find results

- Build artifact (JAR): target/*.jar
- Coverage report: target/site/jacoco/index.html
- Checkstyle report: target/checkstyle-result.xml
- SpotBugs report: target/spotbugsXml.xml
- SBOM: target/bom.json
- Open the index.html file from JaCoCo in a browser to see coverage results.
- Docker images are stored in **GitHub Container Registry (GHCR)** as a **Package**.

How to find Docker image in the GitHub UI

- Go to your GitHub **profile page** (top-right → your avatar → *Your profile*).
- On your profile, look at the tabs: **Overview | Repositories | Projects | Packages | Stars**.
- Click **Packages** → you'll see a list of all containers/packages you own.

At the **commit stage** in the CI pipeline, we produce and store **artifacts** so that **downstream steps** (release, deployment, auditing) can use them **without rebuilding**.

The separation of concerns

- **Workflow step (mvn verify)** → ensures code is correct (tests, quality gates).
- **Dockerfile step (mvn package -DskipTests)** → just packages the already-verified code into a runnable JAR.

That's a common pattern: **tests in CI → skip tests in Docker image builds**.

Two types of triggers in the workflow

1. On pull requests (PRs)

- Triggered when someone opens/updates a PR into main.
- The workflow runs **all the checks**:
  - Maven build + tests
  - Quality gates (Checkstyle, SpotBugs, etc.)
  - Docker build (docker build)

⚡ **But:** the Docker image is **not pushed** to GitHub Container Registry.

- Reason: PR code isn't merged yet → it might still be broken or insecure.
- Purpose: this step only proves the **Dockerfile is valid** and that the app *can* be containerized.

👉 Outcome: "Does the Docker build succeed?" ✅ / ❌
👉 No permanent artifact is published.

On pushes to main

- Triggered when code is merged (PR approved and merged) or pushed directly to main.
- The workflow again runs the checks **and** builds the Docker image.
- ⚡ **Here it pushes the image** to GHCR (ghcr.io/...).

👉 Outcome: "This commit is trusted, so let's publish a Docker image for it."
👉 Artifact = permanent, pullable image in the container registry.

Why this distinction?

- **PR builds** = early feedback.
  - Fast check: will this PR break the build/tests?
  - Does the Dockerfile still work?
  - But don't release/publish anything yet.
- **Main builds** = publish stage.

  - Only trusted, reviewed, merged code gets released.
  - The Docker image goes to GHCR so others can run/deploy it.

Branch protection: what it does

When you protect a branch (usually main), you can tell GitHub:

- ✅ **Require PR reviews** before merging.
- ✅ **Require status checks (CI)** to pass before merging.
- ✅ Optionally: require up-to-date with main, block force pushes, enforce linear history, etc.

Result

Now, when someone opens a **PR into main**:

- GitHub will show a **checklist** on the PR:
  - "✔ build-test passed"
  - "✔ docker-image passed"
- The **Merge button stays greyed out** until **all required checks are green**.

If tests fail → ❌ PR blocked.
If review is missing → ⚠️ PR blocked.
Only when everything passes → ✅ Merge button enabled.

Tests that belong in "commit tests"

- **Unit tests:** fast, hermetic, parallelizable; enforce a time budget (e.g., < 5–10 min).
- **Component/contract tests (lightweight):** only if they can run quickly in-process or with **ephemeral infrastructure** (Testcontainers).
- **Coverage thresholds with nuance:** require coverage *trend* not just a fixed %, and exclude generated code.