Scrum Document

Scrum Document

Team Members

- Tester: Abou El-ezz Salma
- Developer: Sijamhodzic Ishak
- Product Owner: Sauer Michael
- Requirements Engineer: Baysal Hülya
- Scrum Master: Spranz Maximilian

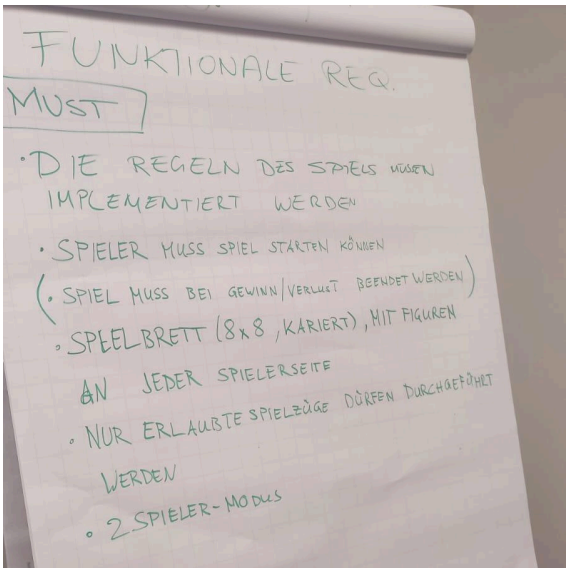
Collaboration/Communication

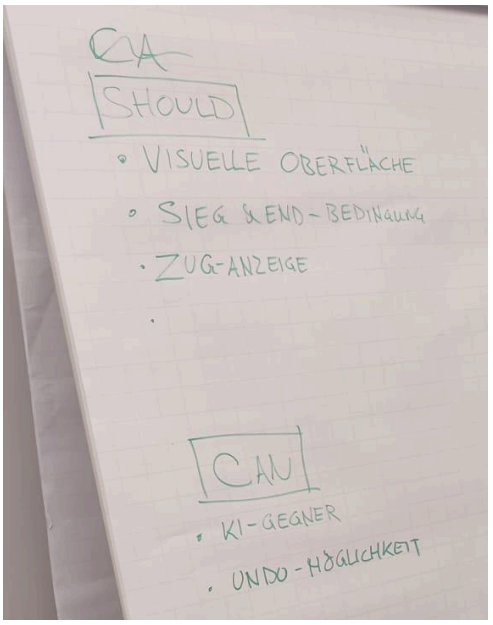
The team was organized through different platforms with the focus of enhancing decentralised working.

- Communication: Whatsapp Groupchat
- Backlog/Burndown: Atlassian Jira, Atlassian Trello
- Repository: Github Repository (https://github.com/ms-se-2025/hcw_colab2025 https://github.com/tm26-se-checkers/collab_hcw2025)
- IDE: IntelliJ

Epic Planning

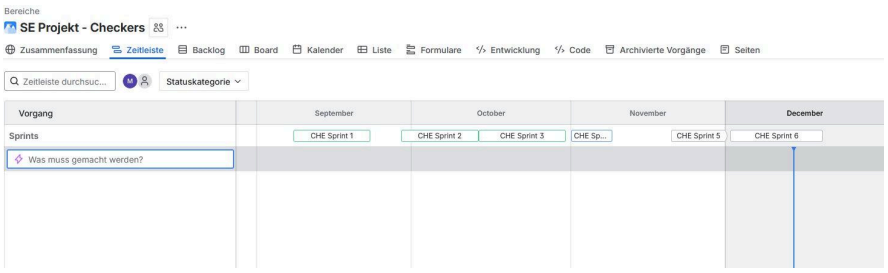
As Epic, the commonly known game Checkers, which was assigned to the group during the first course of the Software Engineering lecture, was defined.
The team began by setting up the repository, the communication channel, as well as a Trello board, which was later replaced by a Jira backlog.
After that, the scope of the Epic was discussed and broken down in functional and non-functional requirements, according to the MoSCoW method.





The team agreed to a 6 sprint development cycle, which was organised by prioritising the “must”-functional requirements, followed by “should” and “can” requirements.
After that, non-functional requirements were planned to be developed.

In the end, as all members were under high (external) pressure, certain requirements were developed, according to its initial estimated effort and not along the defined list as depicted above. As a result, sprint 6, which contained mainly non-functional requirements was cancelled as voted by the team.



Working phase

In the first sprints, the team was working with a private Github repository, which turned out to cause some issues for the members working collaboratively, as read/write access was restricted to the administrators only, of which there could only be one. In addition, adding the repository to the IDE for testing and development needed some workarounds, which was later resolved by moving the repository to an organisation account.
Throughout development, updates, and changes have been managed internally via the Whatsapp groupchat, which was later adapted in the merge requests.

The Software Engineering lectures have been the main meeting point to conduct standups and work through the story points of each sprint. If challenges arose, the group helped each other to resolve these circumstances, i.e. difficulties working with the IDE.

Testing was done via unit testing throughout the development process, which, in the end, created a high performance game for players to enjoy. By the end of sprint 4, refactoring and code maintenance began to take shape in the form of less objects being used and more comments throughout the codebase, in order to create an easier to maintain code.

The process of documenting the project was done by the developer and requirements engineer with the help of the product owner.

Certain items, like computer players or a GUI, were dropped from the scope, as the team was under high external pressure.

A retrospective will be held, once the project has been presented in the final lecture.

Final thoughts

The team, as assembled, proved to be a well functioning organisation, where a very helpful way of collaboration was common throughout the process. The team is satisfied with the outcome presented.

Retrospectifically, the scrum master should have been more involved in team communication and managing the backlog more closely with the product owner, as the dropped story points could have been implemented, if time management had been done more carefully.
In addition, more (off campus) standups could have helped to achieve the full scope as defined in the beginning.

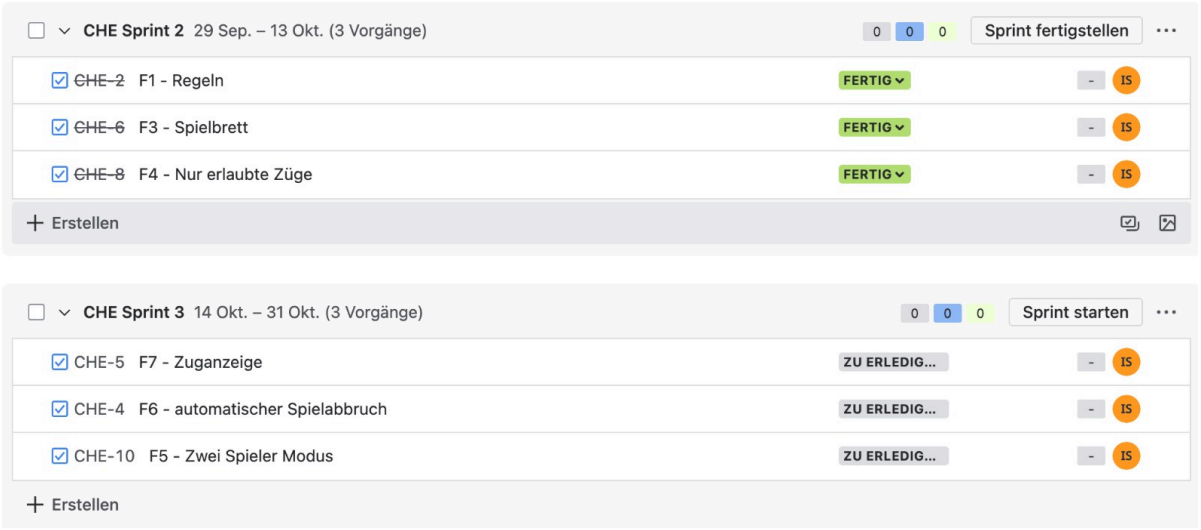
Sprint Overview

Sprint 1:

Administrative Tasks

- GitHub Repo
- Organisation
- Requirements Engineering
- Deadline definitions
- Roles setup

Sprint 2 and Sprint 3:
Implementation of these features



Sprint 4:
Implemented (with tests)

F2 - Spiel beenden können
F9 - Neues Spiel starten können
F8 - Help
F12 - Rückgängig

Sprint 5:

GUI Feature moved to backlog after alignment with team and decision

Implementing features and feedback from SQT Lecture:

- Added comments to Piece and Board classes
- Included Log4J for easier debugging and following of the app's behaviour

Design Document

Design Document

1) Introduction / Project Information

The “Checkers” project is a text-based game for two players.

The goal is to implement a complete game that correctly follows Checkers rules, including:

- Piece movement and captures
- Multi-capture chains
- Promotion to king
- Game-over detection

This document provides an overview of the system architecture, components, interfaces, UI design, design patterns, and error handling, making the implementation fully understandable.

2) System Overview / Scope

System Overview:

- Console-based two-player game
- 8×8 board displayed using Unicode symbols for pieces
- Input via commands: *<square> <l|r> [b]*
- Game logic: move validation, captures, multi-captures, king promotion
- Game-over detection: no legal moves or all pieces of one color captured

Scope / Limitations:

- No AI for single-player mode
- No undo or saving game state
- Focus on robust game logic and clear separation between logic and UI

Example Scenario:

- Player enters *a3 r* → piece on a3 moves right → game checks validity → board updates position

3) Architecture and Design

Layered Architecture

1. **Model** ***(com.checkers.model)***
    - Classes: *Board, Piece, Color, IllegalMove*
    - Responsible for storing piece and board data, validating moves, and handling errors
2. **Logic** ***(com.checkers.logic)***
    - Class: *GameService*
    - Manages game logic: move validation, multi-captures, player turn management, and game-over detection
3. **UI** ***(com.checkers.ui)***
    - Class: *ConsoleUI*
    - Reads user input, displays the board, and shows error messages

Interaction Flow (Example):

- User enters a move via *ConsoleUI*
- *GameService* validates the move
- *Board* updates piece positions and checks for promotions or multi-captures
- Result is sent back to *ConsoleUI* for display

Class Diagram

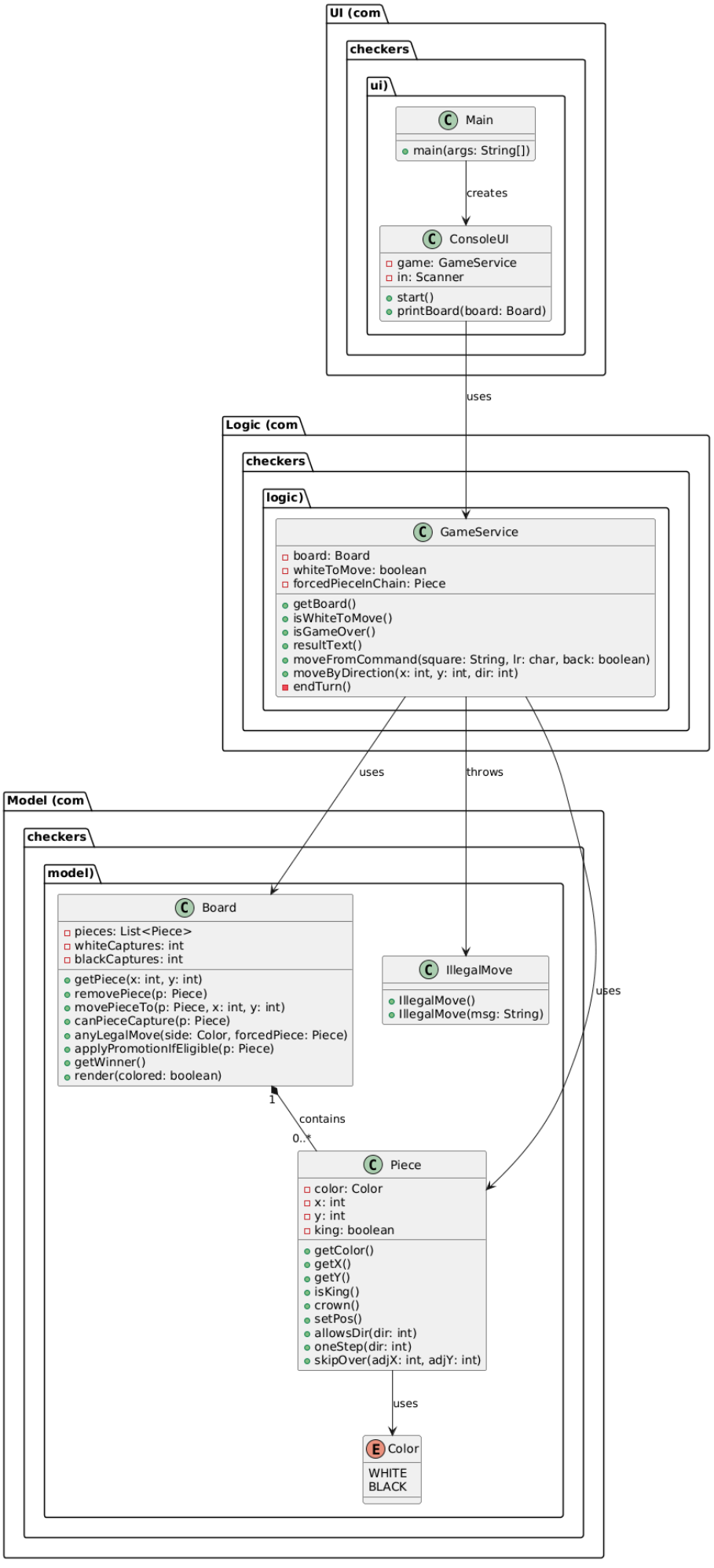
The class diagram illustrates the layered architecture of the Checkers game.

- The **Model layer** contains the game entities *(Piece, Board, Color, IllegalMove)* responsible for storing the game state and validating moves.
- The **Logic layer** *(GameService)* manages game rules, player turns, move validation, and multi-capture chains.
- The **UI layer** *(ConsoleUI)* handles user input and displays the board state.

The diagram also shows key relationships:

- Aggregation: Board contains multiple Piece objects
- Usage: ConsoleUI uses GameService
- Exception handling: IllegalMove thrown by Board and GameService

This visualization helps readers understand both the structure and responsibilities of each system component.


UI (com
checkers
ui)
Main
main(args: String[])
creates
ConsoleUI
game: GameService
in: Scanner
start()
printBoard(board: Board)
uses
Logic (com
checkers
logic)
GameService
board: Board
whiteToMove: boolean
forcedPieceInChain: Piece
getBoard()
isWhiteToMove()
isGameOver()
resultText()
moveFromCommand(square: String, lr: char, back: boolean)
moveByDirection(x: int, y: int, dir: int)
endTurn()
uses
throws
Model (com
checkers
model)
Board
pieces: List<Piece>
whiteCaptures: int
blackCaptures: int
getPiece(x: int, y: int)
removePiece(p: Piece)
movePieceTo(p: Piece, x: int, y: int)
canPieceCapture(p: Piece)
anyLegalMove(side: Color, forcedPiece: Piece)
applyPromotionIfEligible(p: Piece)
getWinner()
render(colored: boolean)
IllegalMove
IllegalMove()
IllegalMove(msg: String)
uses
1
contains
0..*
Piece
color: Color
x: int
y: int
king: boolean
getColor()
getX()
getY()
isKing()
crown()
setPos()
allowsDir(dir: int)
oneStep(dir: int)
skipOver(adjX: int, adjY: int)
uses
Color
WHITE
BLACK

Design Patterns / Decisions

Memento

- **Intent:** Capture and restore object state without exposing internals.
- **Where in your code:** *GameService* maintains an internal stack of *HistoryEntry* objects (each containing a deep copy of *Board* and whose turn it is).
- **Why:** Enables undo functionality for moves or capture chains without exposing the board's internals, keeping encapsulation intact.

Facade

- **Intent:** Provide a simple interface to a complex subsystem.
- **Where:** *GameService* exposes methods like *moveFromCommand(), undo(), restartGame(), endGame(), isGameOver(), resultText().* It hides internal details such as turn management, captures, promotion, and move validation.
- **Why:** Keeps *ConsoleUI* thin, testable, and free from rule orchestration logic.

Strategy

- **Intent:** Define a family of algorithms and make them interchangeable.
- **Where:** *Piece* delegates movement rules to a *MovementPolicy* interface:
    - *ManMovementPolicy* → forward-only movement
    - *KingMovementPolicy* → forward & backward movement

      Piece.crown() swaps the strategy when a piece becomes a king.
- **Why:** Separates what a piece is from how it moves, allowing future changes to movement rules or new piece types without changing core piece logic.

4) Detailed Component and Interface Design

Class	Responsibility	Key Methods / Examples
Piece	Represents a game piece	*getColor(), getX(), getY(), isKing(), crown(), setPos(), allowsDir()*
Color	Enum for player colors	*WHITE, BLACK*
IllegalMove	Exception for invalid moves	*Constructors*
Board	Stores pieces, validates moves and promotions	*getPiece(), removePiece(), movePieceTo(), canPieceCapture(), anyLegalMove(), render()*
GameService	Controls game logic	*moveFromCommand(), moveByDirection(), isGameOver(), resultText()*
ConsoleUI	User interface	*start(), printBoard()*

Example Interaction:

- Player selects piece on *c3* → *GameService* checks move left → *Board* updates position → *ConsoleUI* displays new board

5) User Interface Design

- Input via console: *<square> <l|r> [b]*
- Board displayed using Unicode symbols (⚪ / ⚫ for regular pieces, ♕ / ♛ for kings)
- Colored output using Jansi
- Immediate error messages for invalid moves

Example Commands:

- *a3 r* -> White piece moves from a3 to the right
- *d6 l b* -> Black king moves left backward

6) Error Handling and Dependencies

- **Error Handling:**
    - Invalid moves throw *IllegalMove* exceptions
    - Examples: moving to an empty square incorrectly, moving in the wrong direction, jumping over own piece, attempting a capture without a landing square
    - *ConsoleUI* catches exceptions and prints clear messages
- **Dependencies:**
    - Jansi: colored console output
    - Apache Commons Lang3: input validation (*StringUtils*)
    - Java standard libraries: *Scanner, Collections*
    - Logging: Log4j 2 (optional, INFO for end/restart/promotion, DEBUG for moves)

OOP Principles - Examples

Encapsulation (state hidden behind methods)

- **`Piece`**: private fields (`color`, `x`, `y`, `king`) with getters, `setPos()`, `crown()`.
- **`Board`**: internal `List<Piece>` not exposed; access only via `getPiece()`, `movePieceTo()`, `removePiece()`, `applyPromotionIfEligible()`.
- **`GameService`**: hides rule orchestration and history; UI can't mutate internals directly.

Abstraction (interfaces / contracts)

- **`MovementPolicy`** (interface): defines *what* movement means (`allowsDir`, `delta`) without committing to *how*.
- **`IllegalMove`**: domain-specific type abstracts "rule violation" from printing/logging concerns.
- **`Color`** enum abstracts player side.

Polymorphism (same call, different behavior)

- **Strategy implementations**: `ManMovementPolicy` vs `KingMovementPolicy` both implement `MovementPolicy`.
  `Piece` calls `allowsDir()/delta()`—behavior varies by concrete policy (man vs king).
- **`Piece.crown()`** switches the strategy at runtime → the same `Piece` starts behaving like a king.

Inheritance (type hierarchy)

- Lightweight but present: both policies implement the same interface; `IllegalMove` extends `Exception`.
  (You deliberately avoid deep inheritance for domain objects—good modern practice.)

Composition / "has-a"

- **`Board` has many `Piece`s** (aggregation).
- **`GameService` has a `Board` and a history `Deque<HistoryEntry>`** (Memento implementation).
- **`ConsoleUI` has a `GameService`**.

Information hiding / Layering (good OOP design)

- 3 layers: `ui → logic → model`.
  UI only talks to `GameService` (Facade), never to `Piece`/`Board` directly (except printing).
  Model has no knowledge of UI or Console.

Responsibility separation (SRP)

- **`Piece`**: what a piece is + how it moves (via strategy).
- **`Board`**: grid + piece placement and local rules (promotion/capture availability).
- **`GameService`**: turn rules, validation, multi-capture flow, undo/end/restart.
- **`ConsoleUI`**: input/output only.

Configuration Management Manual

Configuration Management Manual

1) Purpose

Define how we control source, deps, builds, tests, and releases so builds are repeatable and changes are traceable.

2) Repo & Layout (Git)

Everything is versioned (NF110).

```
src/
  main/java/com/checkers/{ui,logic,model}/
  main/resources/ (checkstyle.xml, log4j2.xml)
  test/java/com/checkers/{logic,model,testutil}/
docs/
pom.xml
```

3) Branching & Flow

- Default: `main` (always releasable).
- Work: `feature/<name>`, `fix/<name>`.
- Flow: branch → small commits → PR → review → **CI green** → merge.
- Commit msgs reference FR/NF when relevant.

4) Versioning

- SemVer-ish: `MAJOR.MINOR.PATCH`.
- Dev: `-SNAPSHOT` in `pom.xml`; release tags `vX.Y.Z`.

5) Build & Quality Gates

- Tooling: **Maven**, JDK **21**.

Single command (local & CI):

```
mvn clean verify
```

- Runs: tests (JUnit 5) + **Checkstyle** + **PMD** + **SpotBugs** (NF150).
- Reports: `target/site/{checkstyle.html,pmd.html}`, `target/spotbugsXml.xml`.

6) Dependencies & Plugins

- External libs (NF130):
  - `org.fusesource.jansi:jansi` (colored console)
  - `org.apache.commons:commons-lang3` (string utils)
  - `org.apache.logging.log4j:log4j-{api,core}` (logging)
- Plugins: surefire, checkstyle, pmd, spotbugs.
- Upgrades via PR; CI must pass.

7) Environments

- Local: JDK 21, Maven 3.9+.
- CI: same `mvn clean verify`.
- (Optional) Jansi warning: add `--enable-native-access=ALL-UNNAMED`.

8) Config Files

- `src/main/resources/checkstyle.xml` – coding rules.
- `src/main/resources/log4j2.xml` – console logging (INFO default, DEBUG dev).

9) Testing

- Unit tests only; mirrors `main/` packages.
- Helpers in `com.checkers.testutil.TestHelpers`.

Run single test:

```
 mvn
-Dtest=com.checkers.logic.GameServiceRulesTest#multiple_capture_chai
n test
```

-

10) CI/CD (light)

- Trigger: push/PR to `main` and feature branches.
- Steps: checkout → JDK 21 → `mvn clean verify` → publish reports.
- Release (manual): bump version, tag `vX.Y.Z`, create release notes.

11) Release Checklist

1. Remove `-SNAPSHOT`, set `X.Y.Z` in `pom.xml`.
2. `mvn clean verify` (green).
3. Tag `vX.Y.Z` and push.
4. Release notes (features, fixes, checks).
5. Post-release: bump to next `-SNAPSHOT`.

12) Coding Standards (Verification) — NF150

- Enforced by Checkstyle/PMD/SpotBugs in `mvn verify`.
- Build fails on violations (except any temporarily quarantined PMD rule; documented in README).
- Proof: show `pom.xml` plugin config + generated reports.

13) Documentation & Traceability

- Docs in `docs/` (sprints, test design, this manual).
- PRs reference user stories/acceptance criteria; commit history is the audit trail.

14) Quick Commands

```
git checkout -b feature/some-change
mvn clean verify
git tag v0.2.0 && git push origin v0.2.0
```

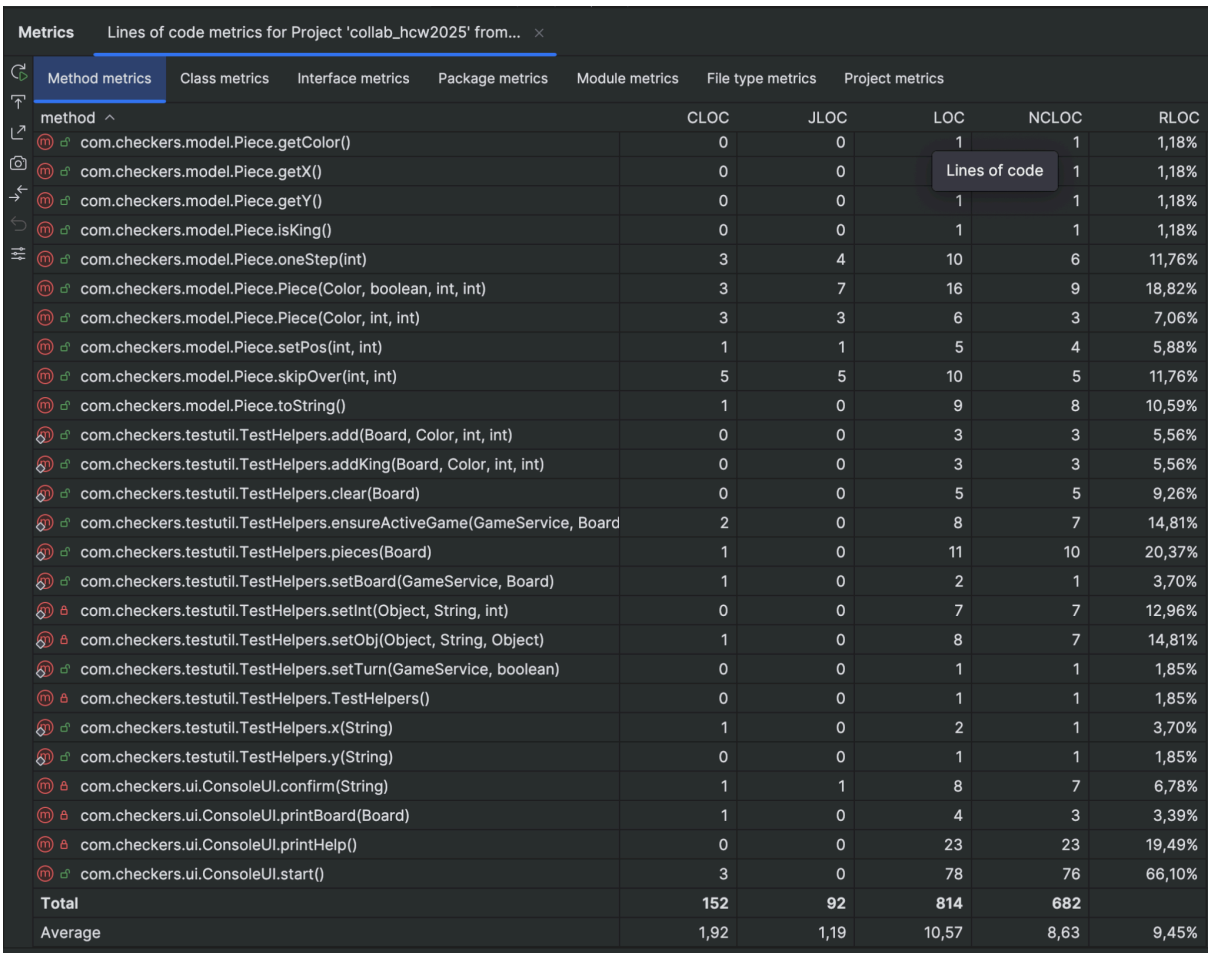
15) Known Deviation (if applicable)

- PMD 7 StackOverflow bug: project may pin PMD 6.x or run PMD non-blocking; Checkstyle + SpotBugs remain strict.

KPIs

Lines Of Code Analysis

Method Analysis



method	CLOC	JLOC	LOC	NCLOC	RLOC
com.checkers.model.Piece.getColor()	0	0	1	1	1,18%
com.checkers.model.Piece.getX()	0	0	1	1	1,18%
com.checkers.model.Piece.getY()	0	0	1	1	1,18%
com.checkers.model.Piece.isKing()	0	0	1	1	1,18%
com.checkers.model.Piece.oneStep(int)	3	4	10	6	11,76%
com.checkers.model.Piece.Piece(Color, boolean, int, int)	3	7	16	9	18,82%
com.checkers.model.Piece.Piece(Color, int, int)	3	3	6	3	7,06%
com.checkers.model.Piece.setPos(int, int)	1	1	5	4	5,88%
com.checkers.model.Piece.skipOver(int, int)	5	5	10	5	11,76%
com.checkers.model.Piece.toString()	1	0	9	8	10,59%
com.checkers.testutil.TestHelpers.add(Board, Color, int, int)	0	0	3	3	5,56%
com.checkers.testutil.TestHelpers.addKing(Board, Color, int, int)	0	0	3	3	5,56%
com.checkers.testutil.TestHelpers.clear(Board)	0	0	5	5	9,26%
com.checkers.testutil.TestHelpers.ensureActiveGame(GameService, Board	2	0	8	7	14,81%
com.checkers.testutil.TestHelpers.pieces(Board)	1	0	11	10	20,37%
com.checkers.testutil.TestHelpers.setBoard(GameService, Board)	1	0	2	1	3,70%
com.checkers.testutil.TestHelpers.setInt(Object, String, int)	0	0	7	7	12,96%
com.checkers.testutil.TestHelpers.setObj(Object, String, Object)	1	0	8	7	14,81%
com.checkers.testutil.TestHelpers.setTurn(GameService, boolean)	0	0	1	1	1,85%
com.checkers.testutil.TestHelpers.TestHelpers()	0	0	1	1	1,85%
com.checkers.testutil.TestHelpers.x(String)	1	0	2	1	3,70%
com.checkers.testutil.TestHelpers.y(String)	0	0	1	1	1,85%
com.checkers.ui.ConsoleUI.confirm(String)	1	1	8	7	6,78%
com.checkers.ui.ConsoleUI.printBoard(Board)	1	0	4	3	3,39%
com.checkers.ui.ConsoleUI.printHelp()	0	0	23	23	19,49%
com.checkers.ui.ConsoleUI.start()	3	0	78	76	66,10%
Total	**152**	**92**	**814**	**682**	
Average	1,92	1,19	10,57	8,63	9,45%

Average actual Lines of Code per method: 10.57
→ good result, methods do one thing and one thing only

Class Metrics

See below

Metrics | Lines of code metrics for Project 'collab_hcw2025' from...

Method metrics | Class metrics | Interface metrics | Package metrics | Module metrics | File type metrics | Project metrics

class	CLOC	JLOC	LOC
com.checkers.AppTest	0	0	2
com.checkers.logic.GameService	45	27	212
com.checkers.logic.GameService.HistoryEntry	1	1	6
com.checkers.logic.GameServiceRulesTest	26	0	98
com.checkers.logic.GameServiceUndoEndRestar	19	7	82
com.checkers.Main	0	0	13
com.checkers.model.Board	63	59	193
com.checkers.model.BoardInitTest	6	0	32
com.checkers.model.Color	0	0	1
com.checkers.model.IllegalMove	0	0	5
com.checkers.model.KingMovementPolicy	5	1	17
com.checkers.model.ManMovementPolicy	6	1	19
com.checkers.model.Piece	32	31	85
com.checkers.testutil.TestHelpers	6	0	54
com.checkers.ui.ConsoleUI	5	1	118
Total	**214**	**128**	**937**
Average	14,27	8,53	62,47

Dependencies

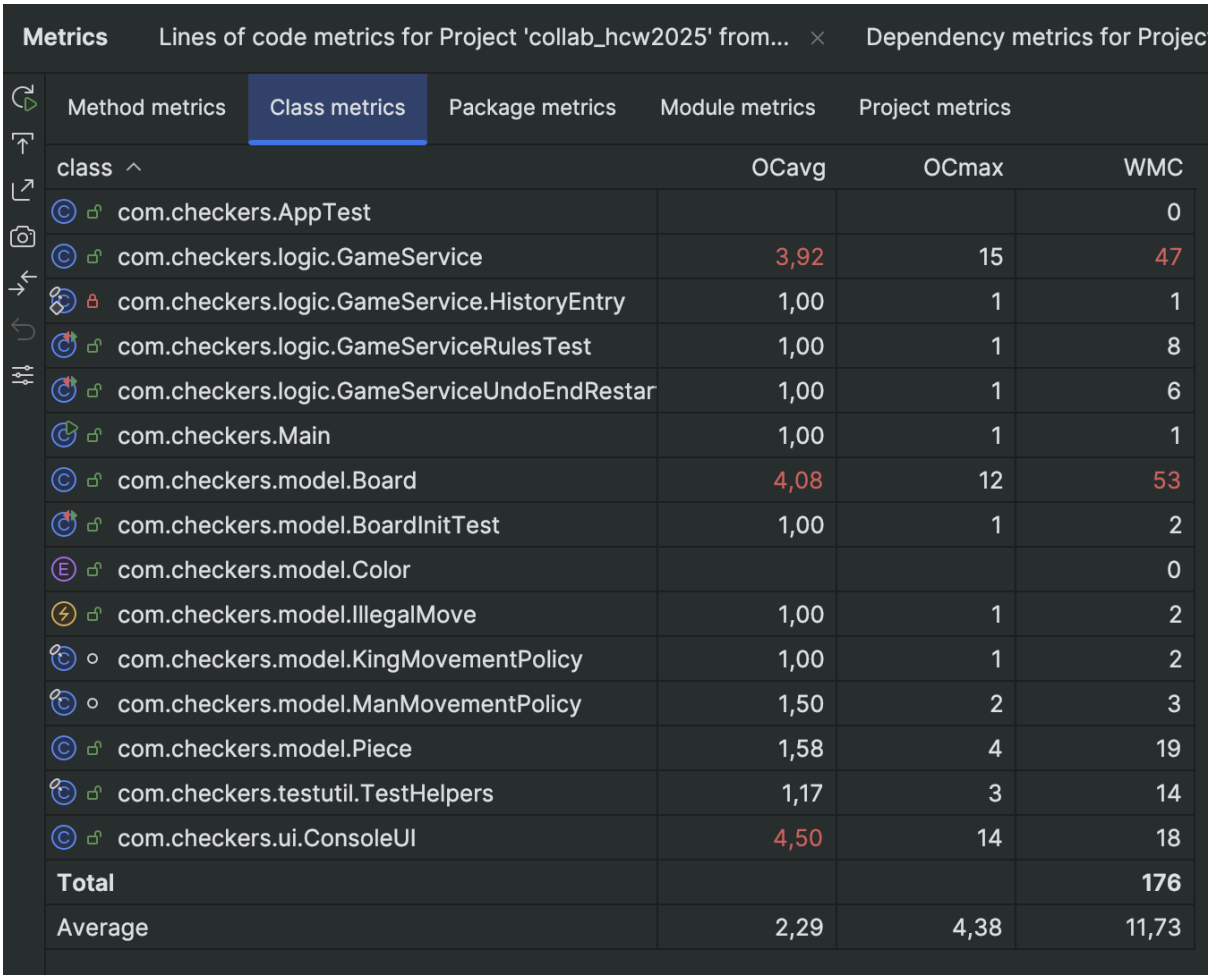
Class Metrics

Metrics | Lines of code metrics for Project 'collab_hcw2025' from... | Dependency metrics for Project 'collab_hcw2025' from ...

Class metrics | Interface metrics | Package metrics

class	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
com.checkers.AppTest	0	0	0	0	0	0	0
com.checkers.logic.GameService	0	5	9	4	6	2	3
com.checkers.logic.GameService.HistoryEntry	0	1	7	1	7	1	1
com.checkers.logic.GameServiceRulesTest	0	5	11	0	0	3	0
com.checkers.logic.GameServiceUndoEndRestar	0	4	11	0	0	3	0
com.checkers.Main	0	1	11	0	0	1	0
com.checkers.model.Board	1	3	6	8	9	1	4
com.checkers.model.BoardInitTest	0	4	11	0	0	2	0
com.checkers.model.Color	0	0	0	10	13	0	3
com.checkers.model.IllegalMove	0	0	0	2	7	0	2
com.checkers.model.KingMovementPolicy	0	2	2	1	10	1	1
com.checkers.model.ManMovementPolicy	0	2	2	1	10	1	1
com.checkers.model.Piece	1	5	6	5	9	1	3
com.checkers.testutil.TestHelpers	0	4	10	3	3	2	2
com.checkers.ui.ConsoleUI	0	4	10	1	1	2	1
Total							
Average	0,13	2,67	6,40	2,40	5,00	1,33	1,40

Complexity



Metrics | Lines of code metrics for Project 'collab_hcw2025' from... | Dependency metrics for Projec

Method metrics | Class metrics | Package metrics | Module metrics | Project metrics

class	OCavg	OCmax	WMC
com.checkers.AppTest			0
com.checkers.logic.GameService	3,92	15	47
com.checkers.logic.GameService.HistoryEntry	1,00	1	1
com.checkers.logic.GameServiceRulesTest	1,00	1	8
com.checkers.logic.GameServiceUndoEndRestar	1,00	1	6
com.checkers.Main	1,00	1	1
com.checkers.model.Board	4,08	12	53
com.checkers.model.BoardInitTest	1,00	1	2
com.checkers.model.Color			0
com.checkers.model.IllegalMove	1,00	1	2
com.checkers.model.KingMovementPolicy	1,00	1	2
com.checkers.model.ManMovementPolicy	1,50	2	3
com.checkers.model.Piece	1,58	4	19
com.checkers.testutil.TestHelpers	1,17	3	14
com.checkers.ui.ConsoleUI	4,50	14	18
Total			**176**
Average	2,29	4,38	11,73

Files Übersicht

Source (main)

com.checkers.ui

- **ConsoleUI**
  - Reads console input; prints board.
  - Routes commands to `GameService` (`help`, `undo`, `end`, `restart`, moves).
  - Uses Log4j for trace/info logs.

com.checkers.logic

- **GameService** *(Facade + Memento)*
  - Validates & executes moves, captures, chain captures, promotion.
  - Manages turn, undo history, end/restart.
  - Single API the UI calls.

com.checkers.model

- **Board**
  - 8×8 state; piece lookup/move/remove; promotion & capture checks; winner calc; console render.
  - Deep copy used for undo snapshots.
- **Piece** *(Strategy)*
  - Color, position, king flag; delegates movement rules to policy; `crown()` switches policy.
- **MovementPolicy / ManMovementPolicy / KingMovementPolicy**
  - Direction rules for men vs kings.

- **Color** (enum), **IllegalMove** (domain exception).

Tests

- **testutil/TestHelpers**: fixtures (clear/add pieces, set turn, map “c3”→(x,y)).
- **model/BoardInitTest**: start layout + render smoke.
- **logic/GameServiceRulesTest**: valid/invalid moves, captures, chain, promotion, king backward, “no global mandatory capture”.
- **logic/GameServiceUndoEndRestartTest**: undo, end, restart behavior.

Config / Build

- `pom.xml`: Java 21, JUnit 5; deps = Jansi, Apache Commons Lang, Log4j2; plugins = Surefire, Checkstyle, PMD, SpotBugs.
- `src/main/resources/checkstyle.xml`: style rules (imports, names, braces/whitespace, line length, file/class).
- **How to verify**: `mvn verify` → runs tests + static analysis; reports in `target/site/`.

Design Patterns (what, why, where)

- **Facade** – provide a simple unified interface to a subsystem (GoF).
    - *Why:* decouple UI from rules, simpler API.
    - *Where:* `GameService`.
- **Memento** – capture & restore object state without exposing internals (GoF).
    - *Why:* safe undo while keeping `Board` encapsulated.
    - *Where:* `GameService` history (deep-copied `Board` + turn).

- **Strategy** – swap algorithms at runtime via an interface (GoF).
    - *Why:* cleanly separate movement rules for men vs kings.
    - *Where:* `MovementPolicy` with `ManMovementPolicy` / `KingMovementPolicy`, used by `Piece`.

Non-functional requirements — proof checklist

- **NF100 Scrum used**
    - *Proof:* sprint docs/boards (e.g., `docs/sprint-1.md`, `docs/sprint-2.md`), user stories & acceptance criteria you listed; commit history grouped by sprint.
    - *How to show:* link to sprint notes/backlog + PRs per sprint.
- **NF110 Git for all files**
    - *Proof:* project is a Git repo; all source/tests/config checked in.
    - *How to show:* `.git/` exists; `git log` shows history; PRs in remote.
- **NF120 3-layer architecture**
    - *Proof:* package split `ui` → `logic` → `model` (one-way deps).
    - *How to show:* tree screenshot + “UI imports logic; logic imports model; model imports nothing upward”.
- **NF130 ≥2 external Java libraries**
    - *Proof:* `pom.xml` dependencies:
        - `org.fusesource.jansi:jansi` (colored console),
        - `org.apache.commons:commons-lang3` (input helpers),
        - `org.apache.logging.log4j:log4j-api/core` (logging).

    - *How to show:* open `pom.xml`; show usage in `ConsoleUI`/`Board`.

- **NF140 Design patterns**
    - *Proof:* Facade (`GameService`), Memento (undo snapshots), Strategy (`MovementPolicy` in `Piece`).
    - *How to show:* point to classes and brief code snippets (constructor of `HistoryEntry`, `Piece.crown()` switching policy, UI only touching `GameService`).

- **NF150 Coding standards verifiable**
    - *Proof:* Checkstyle/PMD/SpotBugs configured in `pom.xml`; rules in `checkstyle.xml`.

Test Case Design

-> in einem File im Code enthalten

Objectives
- Verify functional requirements of the Checkers game (movement, capture, promotion, board init).
- Guard against regressions when new features are added (end/restart/undo/help).
- Keep tests fast and deterministic (unit-level focus).

Scope (what we test)
- **Model:** `Board`, `Piece`, movement helpers and capture detection.
- **Logic:** `GameService` (turn handling, move validation, chain-capture, promotion, undo/end/restart).
- **UI:** `ConsoleUI` is exercised manually (smoke) because parsing/printing is thin and already covered indirectly through `GameService`.

Test Levels & Tags
- **Unit tests** (`@Tag("unit")`): `model` and `logic` packages (default).
- **Integration tests** (`@Tag("integration")`): not needed currently; add later if you wire DB.
- **E2E / UI tests**: N/A for console UI.

Test Techniques
- **Equivalence classes**
  - Valid vs invalid moves (diagonal vs horizontal; correct color to move vs wrong color).
  - Men vs king movement.
  - Capture possible vs not possible; single vs multi-capture.
- **Boundary values**
  - Moves on edges/corners (a1, h8).
  - Promotion on back ranks (y = 1 or 8).
- **Stateful sequences**
  - Multi-capture chains.
  - Undo across multiple turns.
  - End/restart mid-game.

Test Data / Fixtures
- Use **`TestHelpers`** to:
  - clear the board, add specific pieces, set turn, wire a custom board into `GameService`.
  - call `ensureActiveGame` to keep one piece per side so the game is not “already over”.

Naming & Structure
- One test class per unit:
  - `model/BoardInitTest` – initial board and rendering smoke.
  - `logic/GameServiceRulesTest` – movement, capture, promotion, chain capture.
  - `logic/GameServiceUndoEndRestartTest` – undo, end game, restart.
- Test names read as behavior statements (Given/When/Then in comments).
- Use `@DisplayName` for human-readable test names.

Coverage Matrix (excerpt)
| FR / NF | Area | Test(s) |

|---|---|---|
| FR: Board init (8×8, 12 each, distinguishable) | `Board` |
`BoardInitTest.starts_with_12_white_and_12_black_on_correct_rows`,
`board_render_has_axes_and_grid` |
| FR: Rules (diagonal movement, capture, promotion, kings) | `GameService` |
`white_c3_to_d4_is_valid`, `black_forward_uses_plain_lr`,
`promotion_to_king_on_back_rank`, `king_can_move_backward`, `multiple_capture_chain` |
| FR: Only allowed moves (reject invalid) | `GameService` |
`invalid_horizontal_move_rejected` |
| Sprint 2: Undo / End / Restart | `GameService` | `undo_*`, `endGame_*`, `restartGame_*` |
| NF: No mandatory capture globally; chain capture continues | `GameService` |
`may_ignore_available_capture_when_not_in_chain`,
`must_continue_chain_with_same_piece` |

How to run
```bash
# all tests + static analysis
mvn verify

# only unit tests
mvn -q -Dgroups=unit test    # (if you enable groups mapping to tags)
```

ND140 - Design Patterns

1) Memento (implemented)

Intent: capture and restore object state without exposing internals.
Where: `GameService` uses an internal `HistoryEntry` stack (board deep copy + whose turn) to implement **undo**.
Why: clean, reliable undo of single moves or capture chains without leaking board internals.

2) Facade (implemented)

Intent: provide a simple interface to a complex subsystem.
Where: `GameService` exposes simple methods (`moveFromCommand`, `undo`, `restartGame`, `endGame`, `isGameOver`, `resultText`) and hides rule orchestration, turn handling, captures, promotion, etc.
Why: keeps `ConsoleUI` thin and testable.

3) Strategy (implemented)

Intent: define a family of algorithms and make them interchangeable.
Where (new): `Piece` delegates movement rules to a `MovementPolicy`:

- `ManMovementPolicy` (forward-only)
- `KingMovementPolicy` (both directions)
  `Piece.crown()` swaps the strategy.
  Why: clearly separates *what* a piece is from *how* it may move; simplifies future variants or rule changes.

NF150 — Coding standards

Standards

General

  - Java 21 (use `maven.compiler.release=21`).
  - Packages: `com.checkers.{ui|logic|model|testutil}`.
  - No `System.out` in `logic`/`model`; use logging. `ConsoleUI` may print.

- **Naming & style**
  - Classes `UpperCamelCase`, methods/fields `lowerCamelCase`, constants `UPPER_SNAKE_CASE`.
  - One top-level public class per file; no wildcard imports; line length ≤ 120.
- **Javadoc**
  - Public classes and public methods must have Javadoc.
- **Exceptions**
  - Use domain exception `IllegalMove` for rule violations; never swallow exceptions; include helpful messages.
- **Logging**
  - Use **Log4j 2**; `INFO` for significant events (end/restart/promotion), `DEBUG` for move detail.
- **Tests**
  - JUnit 5; structure test packages mirroring `main` packages (`logic`, `model`, `testutil`).
- **Immutability & visibility**
  - Keep state private; expose minimal getters; prefer final where feasible.
- **Dependencies**
  - External libs: Jansi (UI color), Apache Commons Lang (UI input), Log4j 2 (logging).