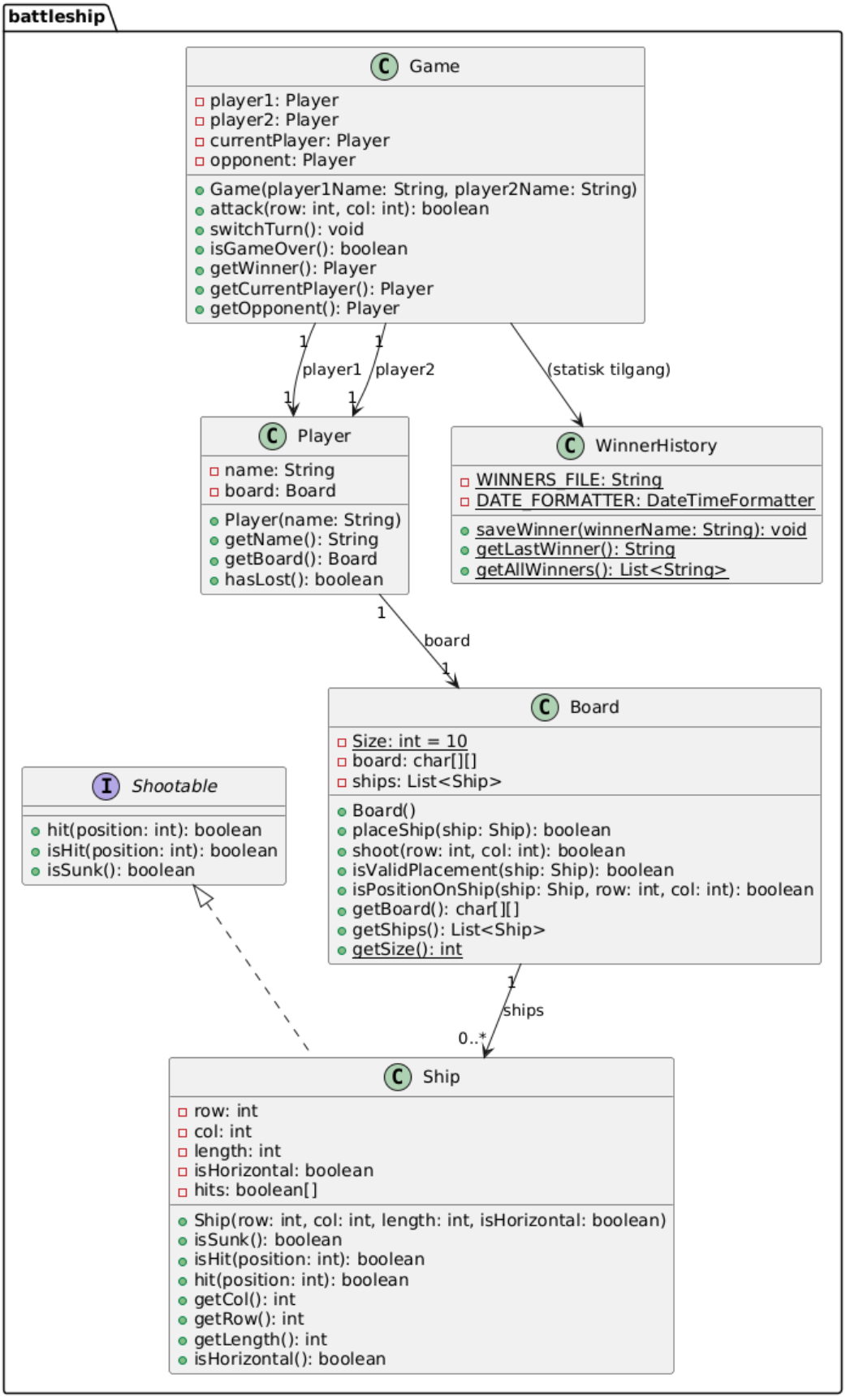
1. Beskrivelse av appen (150-200 ord)

Battleship er en digital versjon av det klassiske brettspillet Battleship, utviklet i Java med JavaFX for det grafiske grensesnittet. Applikasjonen lar to spillere konkurrere mot hverandre ved å plassere skip på et 10x10 rutenett og deretter skyte mot motstanderens brett for å senke skipene deres. Spillet starter med en startmeny der spillere skriver inn navnene sine, etterfulgt av en fase der hver spiller plasserer fem skip av varierende lengde på sitt brett. Deretter veksler spillerne på å skyte mot motstanderens brett, der treff markeres med "X" og bom med "M". Spillet avsluttes når en spillers skip er senket, og vinnerens navn lagres i en fil (winners.txt) for historikk. Tidligere vinnere vises i startmenyen. Applikasjonen er bygget med objektorienterte prinsipper, inkludert innkapsling i klasser som Ship, Board, og Game, og delegering for å håndtere spilllogikk. Grensesnittet (Shootable) brukes for å definere atferd for skip, og unntakshåndtering sikrer robust filhåndtering. Enhetstester i JUnit 5 verifiserer kjernefunksjonalitet som skipplassering, skyting, og vinnerlagring.


battleship
Game
- player1: Player
- player2: Player
- currentPlayer: Player
- opponent: Player
+ Game(player1Name: String, player2Name: String)
+ attack(row: int, col: int): boolean
+ switchTurn(): void
+ isGameOver(): boolean
+ getWinner(): Player
+ getCurrentPlayer(): Player
+ getOpponent(): Player
1
player1
1
1
player2
1
(statisk tilgang)
Player
- name: String
- board: Board
+ Player(name: String)
+ getName(): String
+ getBoard(): Board
+ hasLost(): boolean
WinnerHistory
- WINNERS_FILE: String
- DATE_FORMATTER: DateTimeFormatter
+ saveWinner(winnerName: String): void
+ getLastWinner(): String
+ getAllWinners(): List<String>
1
board
1
Board
- Size: int = 10
- board: char[][]
- ships: List<Ship>
+ Board()
+ placeShip(ship: Ship): boolean
+ shoot(row: int, col: int): boolean
+ isValidPlacement(ship: Ship): boolean
+ isPositionOnShip(ship: Ship, row: int, col: int): boolean
+ getBoard(): char[][]
+ getShips(): List<Ship>
+ getSize(): int
Shootable
+ hit(position: int): boolean
+ isHit(position: int): boolean
+ isSunk(): boolean
1
ships
0..*
Ship
- row: int
- col: int
- length: int
- isHorizontal: boolean
- hits: boolean[]
+ Ship(row: int, col: int, length: int, isHorizontal: boolean)
+ isSunk(): boolean
+ isHit(position: int): boolean
+ hit(position: int): boolean
+ getCol(): int
+ getRow(): int
+ getLength(): int
+ isHorizontal(): boolean

Spørsmål

1. Hvilke deler av pensum i emnet dekkes i prosjektet, og på hvilken måte?

Battleship-prosjektet dekker flere sentrale deler av pensum i objektorientert programmering. For det første har vi brukt klasser og instanser i stor grad, som i Ship, Board, Player, og Game, der vi definerer felt, metoder, og konstruktører. For eksempel har Ship private felt som row, col, og hits, og vi bruker this i konstruktøren for å tilordne verdier til disse feltene. Innkapsling er implementert ved å gjøre alle felt private og tilby offentlige metoder som getRow() og isSunk() for tilgang, noe som beskytter objektets tilstand og definerer et klart grensesnitt.

Vi har også brukt grensesnitt med Shootable, som Ship implementerer. Dette grensesnittet definerer metoder som hit() og isSunk(), og viser hvordan vi skiller grensesnitt og implementasjon, noe som fremmer abstraksjon og polymorfisme. Delegering er brukt i Game, der attack() delegerer til Board.shoot(), som igjen delegerer til Ship.hit(). Dette fordeler ansvar og reduserer kobling mellom klasser.

Unntakshåndtering er implementert i WinnerHistory, der vi håndterer IOException (en checked exception) ved å fange den i try-catch-blokker og returnere standardverdier som "Ingen vinnere ennå". Vi har også brukt Collection-rammeverket med List<Ship> i Board for å lagre skip, og List<String> i WinnerHistory for å hente vinnere fra fil. Generics (<Ship>, <String>) sikrer type-sikkerhet og eliminerer behovet for casting.

Synlighetsmodifikatorer som public, private, og protected er brukt for å kontrollere tilgang, og static og final er brukt i WinnerHistory for konstanter som WINNERS_FILE. Filhåndtering er implementert med java.nio.file.Files for å lese og skrive til winners.txt, som er en form for tegnstrømhåndtering. Til slutt har vi brukt enhetstesting med JUnit 5 for å teste objektoppførsel i klasser som Ship og Board, inkludert edge-cases som ugyldige posisjoner og tomme filer.

2. Dersom deler av pensum ikke er dekket i prosjektet deres, hvordan kunne dere brukt disse delene av pensum i appen?

Selv om prosjektet dekker mange deler av pensum, er det noen konsepter vi ikke har brukt, men som kunne vært nyttige. For det første har vi ikke brukt arv eller abstrakte klasser. Vi

kunne for eksempel ha laget en abstrakt klasse GameEntity som Ship arver fra, med abstrakte metoder som hit(). Dette ville tillatt oss å legge til andre typer enheter (f.eks. miner) som også kan skytes på, ved å arve fra GameEntity og bruke super for å kalle felles funksjonalitet. Instanceof og casting kunne vært brukt i Board.shoot() for å håndtere ulike typer Shootable-objekter dynamisk.

Vi har heller ikke brukt funksjonelle grensesnitt eller Stream-teknikken. For eksempel kunne vi ha brukt Stream i WinnerHistory.getAllWinners() for å filtrere vinnere basert på dato, som Files.readAllLines(path).stream().filter(line -> line.contains("Player1")).collect(Collectors.toList()). Funksjonelle grensesnitt som Predicate<String> kunne vært brukt for å definere filtreringslogikk. Comparable eller Comparator kunne vært brukt for å sortere vinnerlisten etter dato, ved å parse tidsstempelet i hver linje.

Observatør-observert-mønsteret er heller ikke brukt, men kunne vært nyttig for å oppdatere grensesnittet automatisk. For eksempel kunne Game vært en observatør som lytter til endringer i Board (f.eks. når et skip senkes), og oppdatert GameBoardController deretter. Optional kunne vært brukt i Game.getWinner() for å returnere Optional<Player> i stedet for null, noe som ville gjort koden mer robust mot NullPointerException.

3. Hvordan forholder koden deres seg til Model-View-Controller-prinsippet?

Battleship-applikasjonen følger delvis Model-View-Controller (MVC)-prinsippet, men det er rom for forbedring. Model er representert av klassene Game, Player, Board, Ship, og WinnerHistory, som håndterer spillogikk og data. Disse klassene er uavhengige av grensesnittet og fokuserer på spilltilstand og regler, som å plassere skip (Board.placeShip) og registrere treff (Ship.hit). View er implementert i JavaFX FXML-filene (AppBat.fxml og GameBoard.fxml), som definerer det grafiske grensesnittet, som rutenett og etiketter. Controller er BattleshipController og GameBoardController, som binder modellen til visningen ved å håndtere brukerinteraksjoner, som klikk på rutenettet (handleCellClick).

Imidlertid har vi plassert for mye logikk i GameBoardController, som oppdatering av grensesnittet (updateBoards). Dette bryter med MVC-prinsippet, da controlleren bør være en tynn kobling som delegerer til modellen, ikke håndtere visningslogikk direkte. En bedre tilnærming ville vært å flytte updateBoards-logikken til en separat View-klasse som

håndterer rendring, mens GameBoardController kun kaller metoder på Game og sender data til visningen. I tillegg mangler vi en klar separasjon mellom modell og visning – GameBoardController har direkte tilgang til modellens data (game.getCurrentPlayer().getBoard()), noe som skaper tettere kobling enn ønskelig. Vi kunne ha brukt observatør-observert-mønsteret for å la modellen varsle controlleren om endringer, som ville gjort koden mer i tråd med MVC.

4. Hvordan har dere gått frem når dere skulle teste appen deres, og hvorfor har dere valgt de testene dere har? Har dere testet alle deler av koden? Hvis ikke, hvordan har dere prioritert hvilke deler som testes og ikke?

Vi har brukt JUnit 5 for å skrive enhetstester for å verifisere objektoppførsel i modellklassene WinnerHistory, Ship, Board, og Game. Vi startet med å identifisere kritiske metoder som påvirker spillogikken direkte, som Ship.hit(), Board.shoot(), og Game.switchTurn(). Vi skrev totalt 7 tester, inkludert 1 for WinnerHistory (for fillagring og lesing), 3 for Ship, 2 for Board, og 1 for Game. Testene fokuserer på både typiske tilfeller og edge-cases. For eksempel, i WinnerHistoryTest tester vi saveWinner og getLastWinner ved å slette filen først og deretter lagre og hente en vinner. I ShipTest tester vi hit() med ugyldige posisjoner, isSunk() for å sikre at et skip senkes korrekt, og isHit() for å verifisere at individuelle segmenter registreres riktig. I BoardTest verifiserer vi både plassering av skip og skyting, inkludert at brettet oppdateres med 'X' og 'M'.

Vi valgte disse testene fordi de dekker kjernefunksjonalitet som er avgjørende for spillets korrekte oppførsel: skipplassering, skyting, vinnerbestemmelse, og lagring av vinnere. Vi prioriterte å teste metoder med kompleks logikk, som Board.shoot(), som håndterer treff, bom, og ugyldige posisjoner, fremfor enkle get/set-metoder. Vi testet ikke alle deler av koden, som BattleshipController og GameBoardController, fordi kravene spesifiserte at vi ikke skulle teste controller-klassene, og fordi de inneholder minimal logikk etter at vi flyttet mesteparten til modellen.

Prioriteringen av testene var basert på hvilke deler som har størst innvirkning på spillets funksjonalitet og hvor feil ville vært mest kritiske. For eksempel, en feil i Ship.isSunk() kunne ført til at spillet ikke avsluttes korrekt, så dette var en høy prioritet. Vi testet ikke mindre kritiske metoder som Board.initializeBoard(), da denne metoden er enkel og mindre utsatt for feil. Testene har hjulpet oss med å avdekke og fikse feil, som å sikre at Board.shoot() håndterer allerede truffede posisjoner korrekt, og har gitt oss trygghet i at kjernelogikken fungerer som forventet.