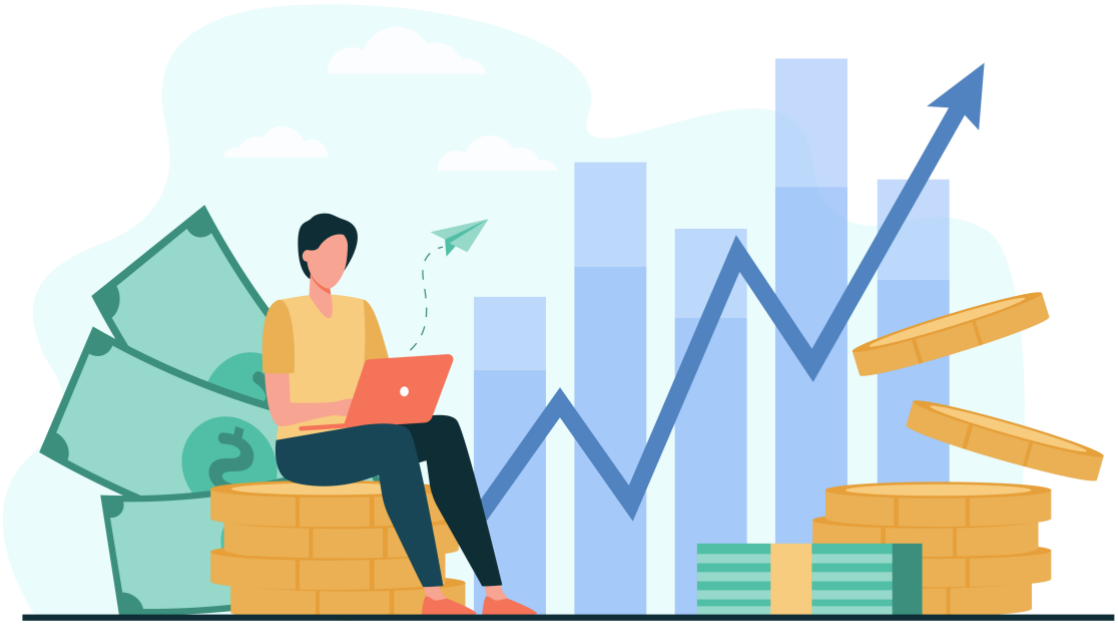


Mappeoppgave: Millions

IDATx2003 Programmering 2, våren 2026

Revisjonshistorikk

Versjon	Dato	Endring
1.0	26.01.2026	Mappe del 1

Innhold

Figurliste

Innledning

I mappeoppgaven skal dere utvikle et aksjespill. Når spillet er ferdig skal man kunne:

- starte et nytt spill med en gitt startkapital, med aksjedata lest fra fil
- hente ut aksjeinformasjon og statistikk
- kjøpe og selge andeler
- se alle transaksjoner som er gjennomført (kjøp og salg)
- få informasjon om vinnere og tapere på markedet
- avansere til ny uke, som medfører endring i priser
- hente ut sin nettoverdi og status
- selge unna hele sin beholdning og avslutte programmet

En mer detaljert liste over funksjonelle krav presenteres i del 3.

Mappens tre deler

Mappen består av et prosjekt i tre deler (teller 70%) og en rapport (teller 30%). Delene presenteres etappevis gjennom semesteret:

- I del 1 er fokuset på grunnleggende klasser og logikk
- I del 2 skal koden utvides med blant annet filhåndtering og statistikk
- I del 3 skal dere implementere et grafisk brukergrensesnitt, anvende designmønster, samt utvide løsningen basert på egne ideer

Hver del bygger på foregående del. Når ny del publiseres vil dere få muntlig tilbakemelding på arbeidet som er gjort så langt, med mulighet til å justere og forbedre løsningen. Dere vil få i alt tre slike tilbakemeldinger. Den fullstendige mappen leveres i Inspera til slutt.

Før dere begynner

Følgende krav og betingelser gjelder for alle oppgavene:

Enhetstesting

Dere må lage enhetstester (både positive og negative) for den delen av koden som er forretningskritisk, altså for den koden som er viktigst for å oppfylle sentrale krav. Feil her vil kunne få store negative konsekvenser for programmet.

Unntakshåndtering

Uønskede hendelser og tilstander som forstyrrer normal flyt skal håndteres på en god måte. Håndteringen skal være rimelig og balansert (ikke for mye, ikke for lite) med det formål å gjøre koden mer robust. Vi gjør oppmerksom på at unntakshåndtering ikke er eksplisitt angitt i oppgavene under. Dere må selv avgjøre når og hvordan unntak skal implementeres.

Kodekvalitet

Det forventes høy kodekvalitet i prosjektet. Bruk verktøy som CheckStyle (med Google sine regler) og SonarLint. Begge verktøy finnes som plugins til IntelliJ og VS Code.

Versjonskontroll

Prosjektet skal legges under versjonskontroll. Krav til versjonskontroll er ytterligere beskrevet i del 1 – Oppsett av prosjektet.

Prosjektrapport

Det skal skrives en rapport for prosjektet. I rapporten skal dere forklare hvordan løsningen er bygget opp, hvilke valg som er tatt underveis, hvordan dere har anvendt designprinsipper, reflektere rundt bruk av KI-verktøy osv. Rapporten skal være på 2500-3000 ord. I tillegg kommer figurer og eventuelt små kodesnutter for å vise hvordan utvalgte problemer er løst. Det er lurt å begynne på rapporten tidlig. Dokumentmal er tilgjengelig i Blackboard.

Bruk av KI-verktøy

Det er tillatt å anvende KI-verktøy som støtte for å løse mappen, så lenge

- verktøyene først og fremst benyttes som sparringspartner/rådgiver og oppslagsverk
- eventuell kode som foreslås gjennomgås kritisk/bearbeides før bruk
- dere kommenterer i koden der dere har fått hjelp av slike verktøy
- bruken dokumenteres i rapporten (hvilke verktøy ble benyttet til hva, gjerne med eksempler på prompts og svar)

Del 1: Grunnleggende klasser og logikk

I denne første delen skal dere opprette et nytt prosjekt og fokusere på grunnleggende klasser og logikk. Når dere har jobbet dere gjennom oppgavene skal resultatet vises til en lærings-assistent. Det er ikke et krav at alle oppgavene skal være fullstendig løst før fristen, men de må som minimum være påbegynt for å få godkjent. Solid og jevn innsats vil gi et godt grunnlag for konstruktive tilbakemeldinger. Vi gjør oppmerksom på at dere også skal levere det som er gjort i Blackboard.

Det kan være lurt å lese gjennom hele dokumentet før dere begynner på oppgavene.

Oppsett av prosjektet

Opprett et tomt Maven-prosjekt og gi prosjektet en fornuftig groupId og artefactId. Prosjektet må følge kravene til konfigurasjon og versjoner som er angitt i BB. Når dere svarer på kodeoppgavene under skal filer lagres iht standard Maven-oppsett: enhetstester legges i katalogen “test/java”, eventuelle ressursfiler (bilder, konfigurasjon osv) legges i “main/resources”, mens resten av koden hører hjemme i katalogen “main/java”. Det skal være mulig å bygge, teste og pakke med Maven fra kommandolinja. Det betyr bl.a. at kommandoen «mvn clean package» skal kjøre uten feil.

Prosjektet skal være underlagt versjonskontroll og være koblet mot et sentralt repositorium:

- legg først koden under lokal versjonskontroll
- opprett så et nytt sentralt repo (tomt prosjekt) med samme navn på GitLab/GitHub (studenter på campus Ålesund og campus Gjøvik skal bruke GitHub Classroom)
- til slutt kobler dere lokalt repositorium mot sentralt repositorium

Dere kan nå samarbeide og pushe mot en felles kodebase. Sjekk inn (commit) jevnlig. Husk at hver commit-melding skal beskrive endringene som er gjort på en kort og konsis måte. Hvis dere sjekker inn på slutten av en oppgave, men senere trenger å gjøre endringer i koden og sjekke inn på nytt, så er det selvfølgelig helt greit. Det er også viktig å merke seg at .git-katalogen skal være med når dere leverer besvarelsen i BB (.git-katatlogen er en "usynlig" mappe i rot-katalogen til prosjektet deres som inneholder all versjonshistorikk).

Modellen vår

Før vi begynner å kode kan det være greit med en kort beskrivelse av de grunnleggende klassene i spillet.

En spiller (Player) kan kjøpe og selge andeler (Share) i en aksje (Stock). Kjøp (Purhase) og salg (Sale) foregår på en børs (Exchange), og representerer finansielle transaksjoner (Transaction). En transaksjon har verdier og kostnader som beregnes av en dedikert kalkulator (TransactionCalculator). Vi opererer med ulike kalkulatorer for kjøp (PurchaseCalculator) og salg (SaleCalculator). Andeler oppbevares i en portefølje (Portfolio), og transaksjonshistorikk lagres i et arkiv (TransactionArchive).

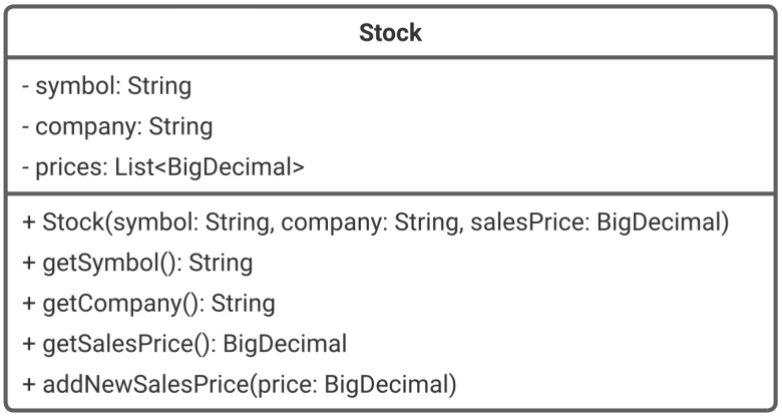
Det er lov å gjøre endringer i modellen dersom det fører til en mer robust arkitektur med løsere koblinger og høyere kodekvalitet. Eventuelle endringer må dokumenteres og begrunnes i prosjektrapporten.

Stock og Share

I dagligtale brukes gjerne begrepene Stock og Share om hverandre. Rent teknisk er det imidlertid vanlig å skille mellom andelen man eier og selve aksjen, og det er denne forståelsen vi legger til grunn for modellen vår.

Stock

Klassen representerer en aksje i et selskap. Den har et unikt symbol, f.eks. «AAPL» for Apple Inc. Aksjer selges på børs (Exchange) for en salgspris som oppdateres ukentlig.

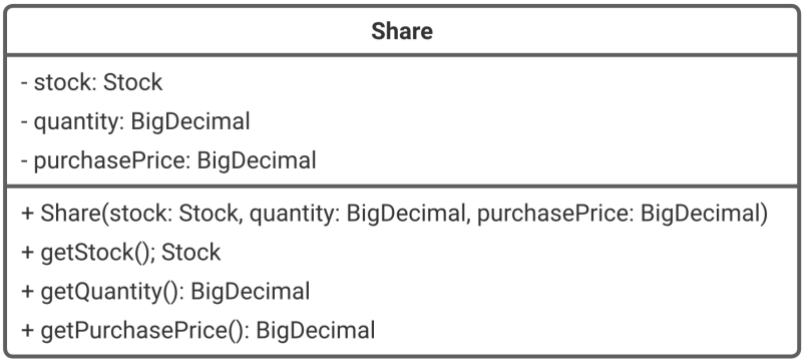


Figur 1: Klassen Stock

Metoden getSalesPrice() skal returnere gjeldende salgspris, som er den siste prisen som ble lagt til i prices.

Share

Et kjøp resulterer i en andel ved klassen Share. Andelen har informasjon om hva slags aksje som ble kjøpt, hvor mye som ble kjøpt (kvantitet), og kjøpspris.

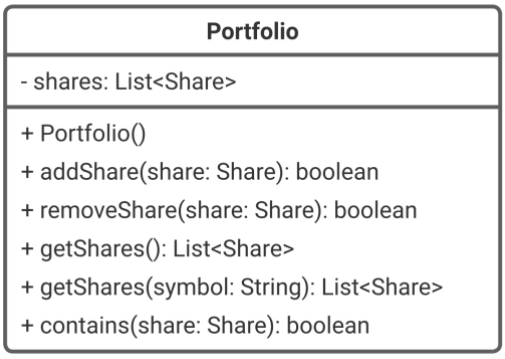


Figur 2: Klassen Share

Legg merke til at vi bruker BigDecimal og ikke float for å representerer desimaltall. Datatypen float har utfordringer knyttet til nøyaktighet[1]. BigDecimal er en del av Java APIet og den anbefalte løsningen når vi må regne med penger.

Portfolio

Vi lagrer andelene til en spiller i en portefølje. Portfolio-klassen er relativt enkel; den har en liste for andeler og metoder for å legge til, fjerne, hente ut, samt sjekke om en andel finnes i porteføljen:



Figur 3: Portfolio-klassen

Transaksjoner

Når vi kjøper og selger aksjer gjennomfører vi en finansiell transaksjon. En transaksjon er knyttet til en andel og en gitt uke. Det er verdier og kostnader knyttet til transaksjonene:

- Bruttoverdi: verdi før avgifter
- Kurtasje (commission): en avgift som betales til megleren
- Skatt (tax): en avgift som betales til staten
- Totalverdi: verdi etter avgifter

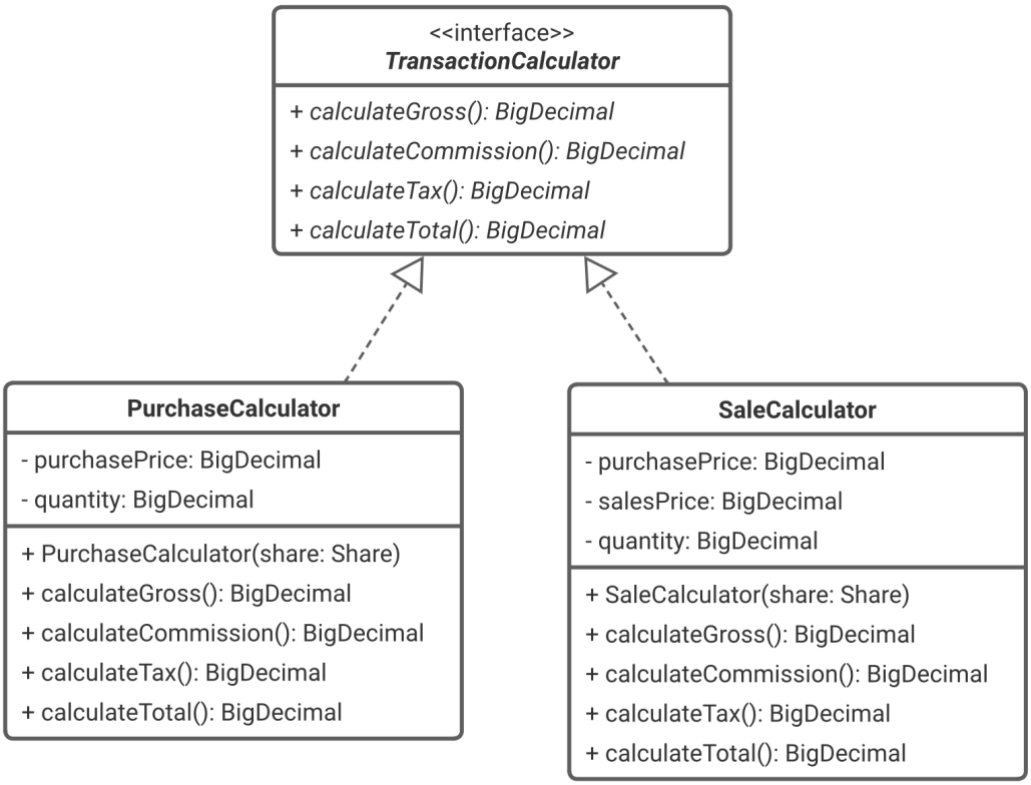
Vi velger å skille ut koden som utfører beregninger av verdier og kostnader i egne klasser.

TransactionCalculator, PurchaseCalculator og SaleCalculator

Alle transaksjoner har egenskapene som beskrevet over, men beregningene for kjøp og salg gjøres forskjellig. Vi definerer derfor de nødvendige beregningsmetodene i et felles grensesnitt (TransactionCalculator), og implementerer de i henholdsvis PurchaseCalculator og SaleCalculator.

Begge klassene har en konstruktør som tar inn en andel. PurchaseCalculator baserer sine beregninger på andelens kjøpspris og kvantitet, mens SaleCalculator i tillegg henter ut salgspris.

[1] Se https://stackoverflow.com/questions/3730019/why-not-use-double-or-float-to-represent-currency



Figur 4: Kalkulatorklasser

For PurchaseCalculator gjelder følgende:

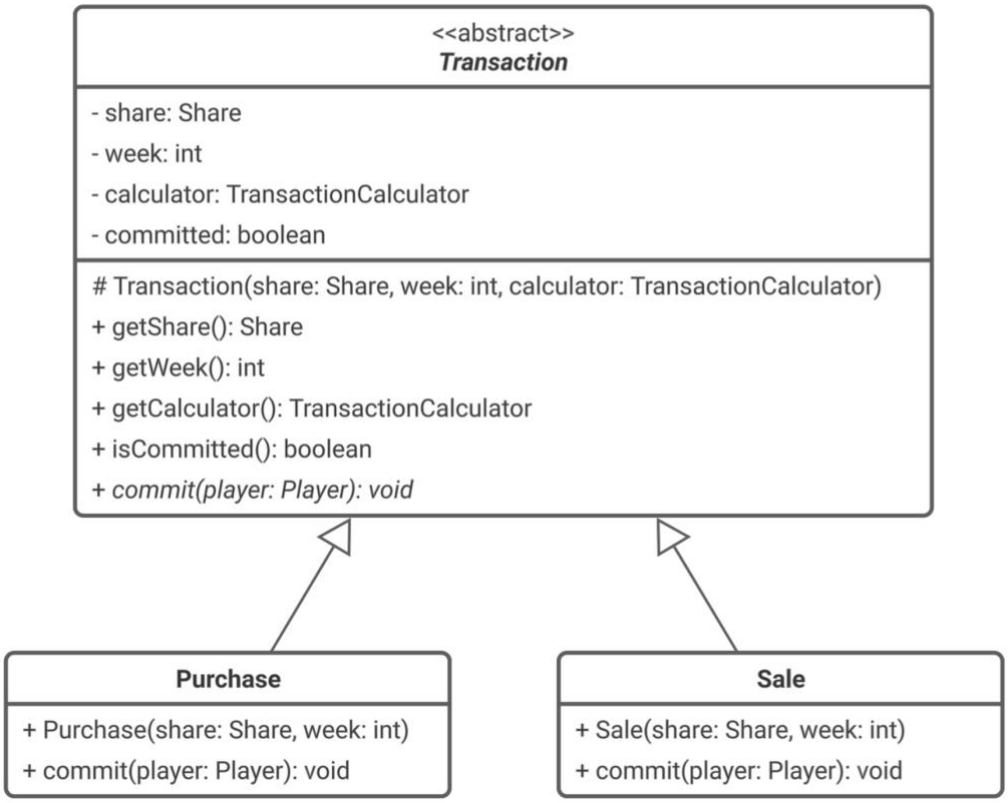
- calculateGross(): bruttoverdi, altså kjøpspris × kvantitet
- calculateCommission(): en avgift på 0.5% av bruttoverdi
- calculateTax(): ingen skatt ved kjøp
- calculateTotal(): total kostnad, dvs bruttoverdi + avgift + skatt

For SaleCalculator blir det litt annerledes:

- calculateGross(): bruttoverdi, altså salgspris × kvantitet
- calculateCommission(): en avgift på 1% av bruttoverdi
- calculateTax(): 30% skatt på gevinst
    - Gevinsten kan beregnes som bruttoverdi – avgift – kjøpskostnader
    - Vi kan forenkle litt og si at kjøpskostnader er kjøpspris × kvantitet
- calculateTotal(): total salgsverdi, altså bruttoverdi – avgift – skatt

Transaction, Purchase og Sale

Kjøp og salg har det til felles at de er knyttet til en andel og en uke, og at de bruker en kalkulator for beregninger. Men det er også vesentlige forskjeller. De bruker logisk nok ulike kalkulatorer, og selve gjennomføringen er forskjellig. Da er det naturlig å samle det som er felles i en abstrakt superklasse som vi kaller Transaction. Vi delegerer gjennomføringen til en abstrakt commit-metode, som implementeres i subklassene Purchase og Sale:

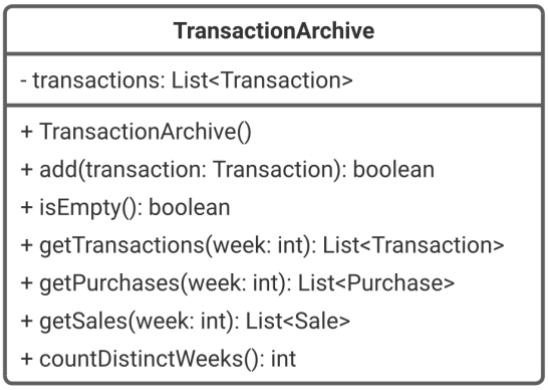


Figur 5: Transaksjonsklasser

Commit-metodene avhenger av Player-klassen, så vi må vente litt med å implementere disse. Enn så lenge kan vi opplyse om at alle transaksjoner skal betraktes som unike; den samme transaksjonen kan ikke gjennomføres flere ganger. Derfor har vi et flagg i super-klassen kalt committed, som settes til true første gangen commit kalles.

TransactionArchive

Alle fullførte transaksjoner skal oppbevares i et arkiv. Klassen TransactionArchive har følgende egenskaper:

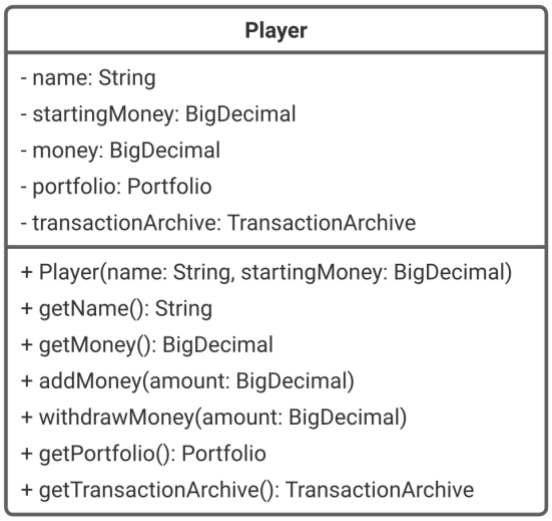


Figur 6: Klassen TransactionArchive

Den siste metoden – countDistinctWeeks() – skal telle antall uker med faktisk handel, og blir nyttig i del 2 av mappen.

Player

Player-klassen ser slik ut:



Figur 7: Player-klassen

En ny spiller har et navn og startkapital. Merk at vi opererer med to attributter for penger; startingMoney og money. Beløpene vil være like ved oppstart, men det er money som skal representere den til enhver tid gjeldende pengebeholdningen. Vi trenger imidlertid å vite hva vi startet med når vi senere skal beregne status i spillet, derfor lagrer vi også startkapitalen. Konstruktøren må også instansiere en ny portefølje og et tomt transaksjonsarkiv.

Nå som Player-klassen er implementert, kan vi gyve løs på commit-metodene i Purchase og Sale.

Purchase sin commit-metode fullfører et kjøp ved å:

1. trekke totalkostnadene fra spillerens pengebeholdning
2. legge til andelen i spillerens portefølje, og
3. lagre transaksjonen i spillerens transaksjonsarkiv

Flagget committed settes til true hvis alt går bra. Det skal ikke være mulig å fullføre et kjøp hvis spilleren ikke har tilstrekkelig med penger, eller hvis transaksjonen er gjennomført tidligere.

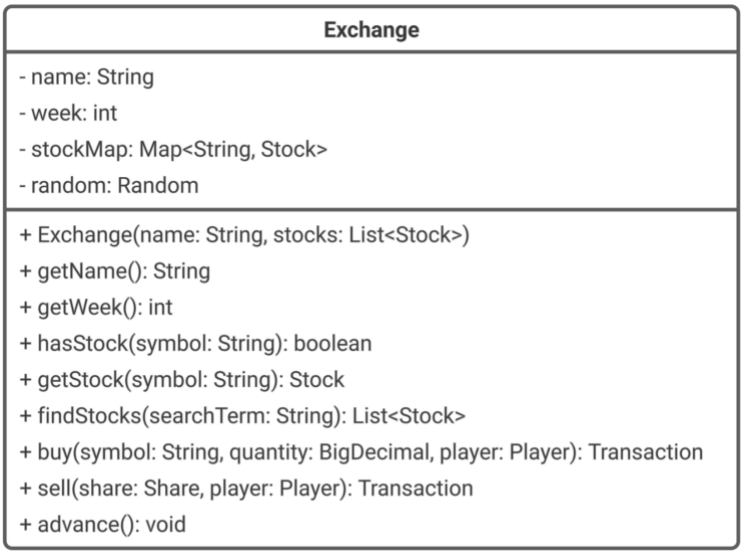
Sale sin commit-metode fullfører et salg ved å:

1. legge til total-verdien til spillerens pengebeholdning
2. fjerne andelen fra spillerens portefølje, og
3. lagre transaksjonen i spillerens transaksjonsarkiv

Flagget committed settes til true hvis alt går bra. Det skal ikke være mulig å fullføre et salg hvis spilleren ikke eier andelen som skal selges, eller hvis transaksjonen er gjennomført tidligere.

Exchange

Exchange er den siste klassen i modellen vår, og representerer ganske enkelt en børs hvor man kan handle aksjer:



Figur 8: Exchange-klassen

Klassen har en konstruktør som tar inn et navn og en liste med aksjene som kan handles på børsen. Internt lagres aksjene i et Map, hvor «symbol» brukes som nøkkel. Ukenummer settes til 1.

Det skal være mulig å kjøpe og selge aksjer som er notert på børsen via buy- og sell-metodene. En handel fullføres som en transaksjon.

Metoden advance() brukes for å gå videre til neste handelsuke, ved å inkrementere ukenummeret og oppdatere prisene. Oppdateringen skjer ved å gå gjennom hver eneste aksje og enten øke eller senke prisen. Ny pris må være tilfeldig for hver aksje, og endringen bør ikke være for stor. Tilfeldighet kan oppnås ved å bruke Random-klassen[2] i Java APIet.

Det skal være mulig å hente ut informasjon om aksjer:

- getStock returnerer ganske enkelt aksjen for et gitt symbol
- findStocks returnerer en liste med alle aksjer som innholder et søkeord. Vi må sjekke både symbolet og selskapsnavnet. Et søk på «Go» vil f.eks. kunne returnere både «**GO**OGL – Alphabet Inc» og «WFC – Wells Far**go**».

Klassen har også metoder for å hente ut navn og ukenummer.

[2] https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Random.html

Klient som kjører et spill (frivillig)

I utgangspunktet skal enhetstestene være nok for å verifisere at modellen fungerer som forventet. Men hvis dere ønsker kan dere også lage en enkel klient som gjør det mulig å kjøre et spill fra kommandolinja.

Viktige sjekkpunkter

Når dere løser mappeprosjektet bør dere dobbeltsjekke følgende[3]:

- Maven:
  - Er prosjektet et Maven-prosjekt med fornuftige prosjekt-verdier og gyldig katalogstruktur?
  - Følges kravene til prosjektkonfigurasjon?
  - Kan man kjøre Maven-kommandoer for å bygge, teste og pakke uten at det feiler?
  - Er det mulig å kjøre disse kommandoene fra terminalen med mvn?
- Versjonskontroll med git:
  - Er prosjektet underlagt versjonskontroll med sentralt repo?
  - Har prosjektet en .gitignore-fil som filtrerer bort irrelevante kataloger og filer?
  - Sjekkes det inn jevnlig?
  - Beskriver commit-meldingene endringene på en kort og konsis måte?
  - Benyttes grener («branches») på en hensiktsmessig måte?
  - Benyttes tags for å merke versjoner?
- Enhetstester:
  - Har enhetstestene beskrivende navn som dokumenterer hva testene gjør?
  - Følger de mønsteret Arrange-Act-Assert?
  - Tas det hensyn til både positive og negative tilfeller?
  - Er testdekningen fornuftig?
  - Benyttes livssyklus-metoder der det er fornuftig (@BeforeEach osv)?
- Er klasser og grensesnitt implementert iht oppgavebeskrivelsen?
- Kodekvalitet:
  - Er koden godt dokumentert iht JavaDoc-standard?
  - Er koden robust (unntakshåndtering, validering mm)?
  - Har variabler, metoder og klasser beskrivende navn?
  - Er klassene gruppert i en logisk pakkestruktur?
  - Har koden god struktur, med løse koblinger og høy kohesjon?
  - Benyttes arv og grensesnitt slik at løsningen er fleksibel og lett å utvide/endre?
  - Benyttes funksjonell programmering (lambda og streams) der det er naturlig?
  - Benyttes linter (f.eks. SonarLint) og tilsvarende verktøy (f.eks. CheckStyle) for kvalitetssjekk?
- Rapport:
  - Har rapporten en fornuftig struktur og et godt språk?
  - Er de formelle kravene oppfylt (innholdsfortegnelse, figur- og tabelliste, sidetall, korrekt referansebruk og referanseliste, vedlegg, makslengde osv)?
  - Omtaler rapporten temaene/kapitlene som er listet opp i malen, med solide beskrivelser, begrunnelser og refleksjoner?

[3] Merk: sjekklista vil oppdateres når del 2 og 3 publiseres.

Illustrasjoner

Alle illustrasjoner og figurer er tilrettelagt av fagstaben i IDATx2003, med unntak av bildet på forsiden, som er utarbeidet av pch.vector/Freepik.