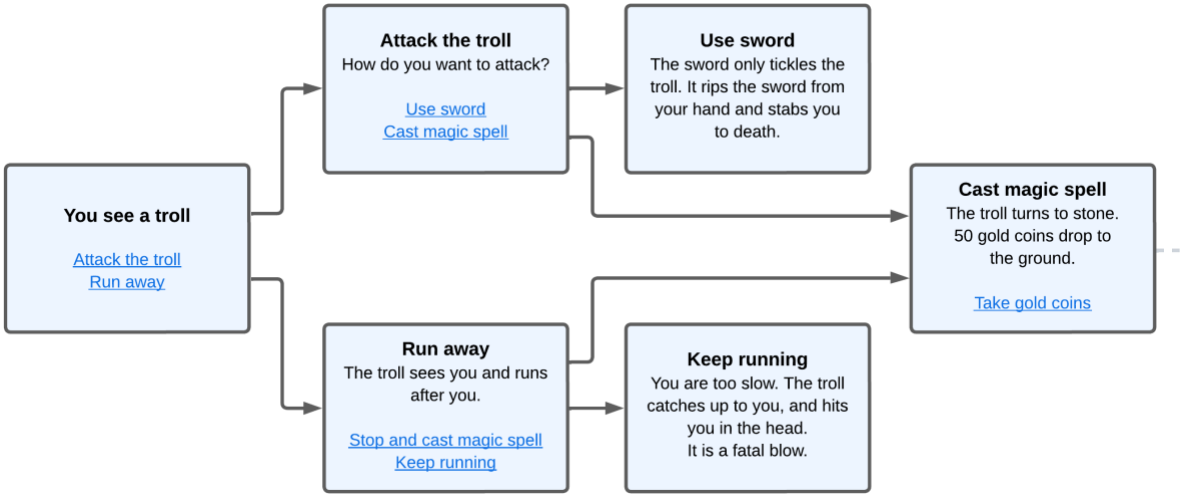
Paths - Del 1



Dere skal utvikle et program kalt Paths, som er en spillmotor for valgbasert og interaktiv historiefortelling. I denne første delen skal dere fokusere på grunnleggende klasser og logikk. I del 2 og 3 skal dere utvide med filhåndtering, et grafisk brukergrensesnitt, samt annen funksjonalitet. Det fullstendige programmet skal leveres i Inspera til slutt.

Når dere har jobbet dere gjennom oppgavene skal resultatet vises til en læringsassistent. 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.

Før dere begynner

Følgende krav og betingelser gjelder for alle oppgavene:

Enhetstesting

Dere må lage enhetstester for den delen av koden som er forretningskritisk, altså for den koden som er viktigst for å oppfylle sentrale krav. Feil her vil 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.

Versjonskontroll

Prosjektet skal legges under versjonskontroll. Krav til versjonskontroll er ytterligere beskrevet i oppgave 1.

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 osv. Rapporten skal være på maks 2500 ord. I tillegg kommer figurer og eventuelt små kodesnutter for å vise hvordan utvalgte problemer er løst. Det er lurt å begynne på rapporten allerede nå. Dokumentmal finner dere på BB.

Oppgave 1: Maven og git

Opprett et tomt Maven-prosjekt og gi prosjektet en fornuftig groupId og artefactId. Prosjektet må følge kravene til konfigurasjon og versjoner som er angitt på 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.

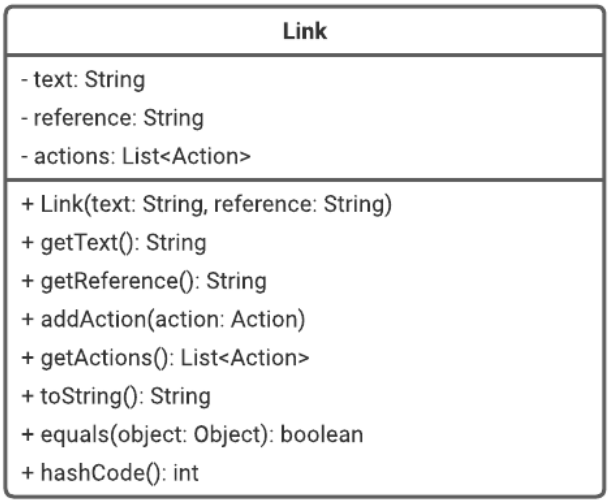
Prosjektet skal være underlagt versjonskontroll og være koblet mot et sentralt repo:

- Legg først koden under lokal versjonskontroll.
- Opprett så et nytt sentralt repo (tomt prosjekt) med samme navn på GitLab (studenter på campus Ålesund skal bruke GitHub Classroom).
- Til slutt kobler dere lokalt repo mot sentralt repo.

Dere kan nå samarbeide og pushe mot en felles kodebase. For hver av oppgavene under skal dere gjøre minst én innsjekk (commit). 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).

Oppgave 2: Link

En link gjør det mulig å gå fra en passasje til en annen. Linker binder sammen de ulike delene av en historie.



Figur 1: Link-klassen

Diagrammet viser at Link-klassen har tre attributter:

- text: en beskrivende tekst som indikerer et valg eller en handling i en historie. Teksten er den delen av linken som vil være synlig for spilleren.
- reference: en streng som entydig identifiserer en passasje (en del av en historie). I praksis vil dette være tittelen til passasjen man ønsker å referere til.
- actions: En liste med spesielle objekter som gjør det mulig å påvirke egenskapene til en spiller. Listen med tilhørende metoder kan først legges etter at dere har løst oppgave 7.

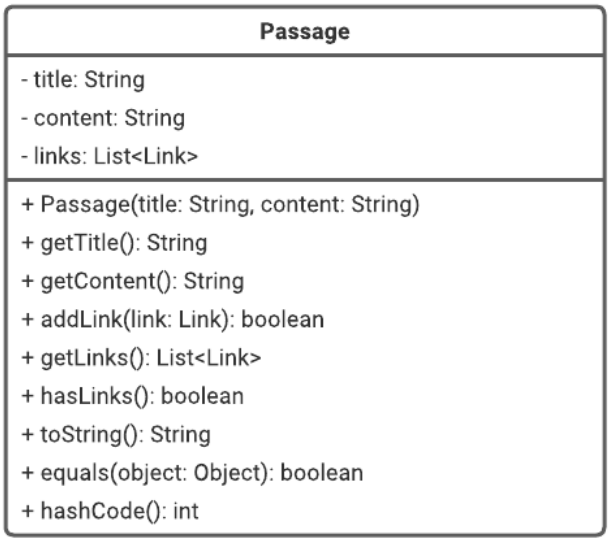
Klassen skal også ha en konstruktør, metoder for å hente ut og manipulere tilstand, samt toString() som skal returnere en fornuftig tekstlig representasjon av linken.

> 🤔 **Hvorfor er ikke reference en referanse til et Passage-objekt?**
>
> Ved å referere til en passasje indirekte gjennom en streng kan vi opprette linken før Passage-objektet er klart. Dette gjør det blant annet enklere å implementere filhåndtering, som vi skal se på i del 2 av prosjektet.

Vi minner om at versjonskontroll, enhetstester og unntakshåndtering også må være på plass (gjelder som tidligere nevnt for alle oppgavene).

Oppgave 3: Passage

En passasje er en mindre del av en historie, et avsnitt om du vil. Det er mulig å gå fra en passasje til en annen via en link.



Figur 2: Passage-klassen

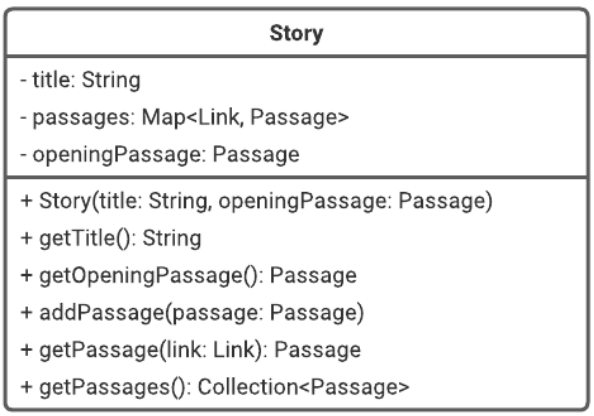
Passage-klassen har tre attributter:

- title: en overordnet beskrivelse som også fungerer som en identifikator.
- content: tekstlig innhold som typisk representerer et avsnitt eller del av en dialog.
- links: linker som kobler denne passasjen mot andre passasjer. En passasje med to eller flere linker gjør historien ikke-lineær.

Klassen skal også ha en konstruktør og metoder som angitt i klassediagrammet over.

Oppgave 4: Story

Nå som Link og Passage er på plass kan Story-klassen kodes. En historie er et interaktivt, ikke-lineært narrativ som består av en samling passasjer.



Figur 3: Story-klassen

Story har tre attributter:

- title: historiens tittel.
- passages: en Map som inneholder historiens passasjer. Nøkkelen til hver passasje er en link.
- openingPassage: den første passasjen i historien. Objektet skal også legges til i passages.

Metoden addPassage skal legge til en passasje i passages. Da trenger vi også et Link-objekt. Dette løser vi ved å opprette en ny link basert på passasjens tittel. Tittelen kan fungere både som tekst og referanse.

Husk også å implementere konstruktøren og de resterende metodene fra klassediagrammet.

Oppgave 5: Player

Player-klassen representerer en spiller med ulike egenskaper som kan påvirkes i en historie.

Figur 4: Player-klassen

Spilleren har et navn, helse (kan aldri bli mindre enn 0), poeng, gull og en liste med ting (inventory). Klassen har en rimelig standard konstruktør, og attributtene kan hentes ut og manipuleres via egne metoder.

Oppgave 6: Game

Game er en fasade for et Paths-spill. Klassen kobler en spiller mot en historie, og har hendige metoder for å starte og manøvrere i spillet.

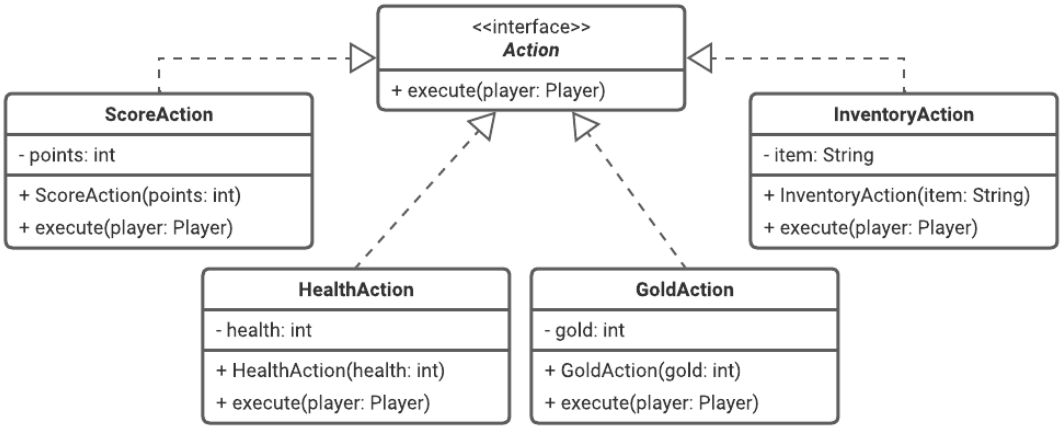
Figur 5: Game-klassen

Klassen har tre attributter. De to første er selvforklarende. Det siste (goals) refererer til en liste med spesielle objekter som angir ønskede resultater i et spill. Attributtet goals og metoden getGoals() kan først legges til etter at oppgave 8 er løst.

Metoden begin() skal ganske enkelt returnere den første passasjen i historien for dette spillet. Metoden go(link: Link) skal returnere passasjen som matcher den angitte linken. Utover dette har klassen en konstruktør og aksessor-metoder for attributtene.

Oppgave 7: Actions

En Action er en handling som representerer en fremtidig endring i tilstanden til en spiller. Dette inkluderer endringer i spillerens poengsum, helse, gullbeholdning eller inventar.



Figur 6: Grensesnittet Action med implementerende klasser

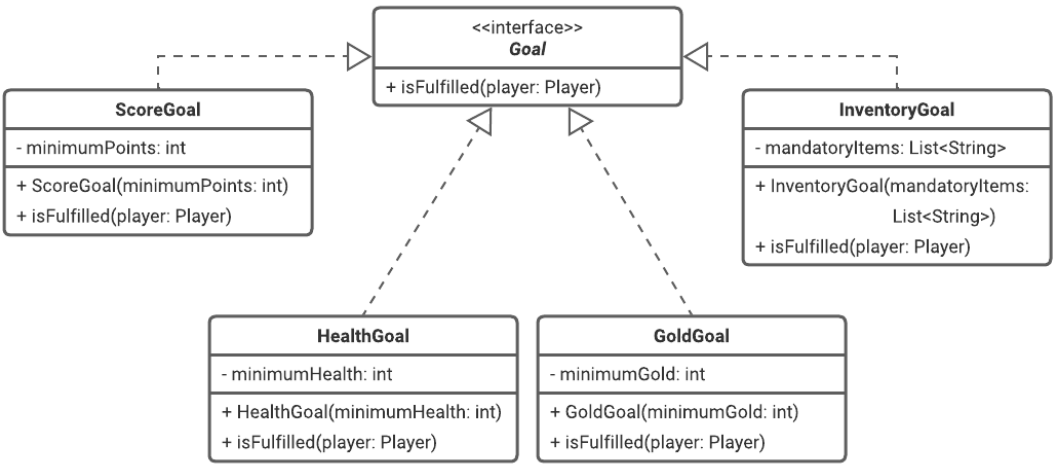
Klassediagrammet viser et grensesnitt og fire implementerende klasser:

- Grensesnittet Action har metoden execute(player: Player), som skal endre tilstanden til spilleren når den kalles.
- Klassen ScoreAction endrer spillerens poengsum.
- Klassen HealthAction endrer spillerens helse.
- Klassen GoldAction endrer spillerens gullbeholdning.
- Klassen InventoryAction legger til en ting i spillerens inventar.

Når grensesnitt og klasser er implementert må dere huske å oppdatere Link-klassen slik at den inkluderer actions.

Oppgave 8: Goals

Et Goal representerer en målverdi eller et ønsket resultat knyttet til spillerens tilstand. Mens actions endrer tilstanden til spilleren underveis, gjør goals det mulig å sjekke om spilleren har oppnådd forventet resultat.



Figur 7: Grensesnittet Goal med implementerende klasser

Klassediagrammet viser et grensesnitt og fire implementasjoner:

- Grensesnittet Goal har metoden isFulfilled(player: Player), som skal returnere true hvis målet er oppnådd eller false hvis det ikke er oppnådd.
- Klassen ScoreGoal representerer en forventet minimum poengsum.
- Klassen HealthAction representerer en forventet minimum helseverdi.
- Klassen GoldGoal representerer en forventet minimum gullbeholdning.
- Klassen InventorGoal representerer et forventet inventar med ting.

Når grensesnitt og klasser er implementert må dere huske å oppdatere Game-klassen slik at den inkluderer goals.

Oppgave 9 (frivillig): Klient som kjører et spill

I utgangspunktet skal enhetstestene være nok for å verifisere at programmet fungerer som forventet. Men hvis dere ønsker kan dere også lage en klient som oppretter en historie og kjører et spill fra kommandolinja.

Viktige sjekkpunkter

Når du løser oppgaven bør du dobbeltsjekke følgende:

- Maven:
    - Er prosjektet et Maven-prosjekt med fornuftige prosjekt-verdier og gyldig katalogstruktur?
    - Kan man kjøre Maven-kommandoer for å bygge, teste, pakke og installere uten at det feiler?
    - Er det mulig å bygge, teste og pakke fra terminalen med mvn?
- Versjonskontroll med git:
    - Er prosjektet underlagt versjonskontroll med sentralt repo?
    - Finnes det minst én commit per kodeoppgave?
    - Beskriver commit-meldingene endringene på en kort og konsis måte?
- Enhetstester:
    - Har enhetstestene beskrivende navn som dokumenterer hva testene gjør?
    - Følger de mønstret Arrange-Act-Assert?
    - Tas det hensyn til både positive og negative tilfeller?
    - Er testdekningen god nok?
- Er klassene Link, Passage og Story implementert iht oppgavebeskrivelsen?
- Er klassene Player og Game implementert iht oppgavebeskrivelsen?
- Er grensesnittet Action med tilhørende klasser kodet iht oppgavebeskrivelsen?
- Er grensesnittet Goals med tilhørende klasser kodet 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?