NTNU
Kunnskap for en bedre verden

# Wargames - Del 1

*Du skal utvikle et program kalt Wargames, som simulerer et slag mellom to arméer i en krig. I denne første delen skal du fokusere på koden som omhandler enheter og simuleringslogikk. I del 2 skal du utvide programmet med bl.a. filhåndtering og et grafisk brukergrensesnitt. I del 3 skal du anvende designmønstre samt legge til mer funksjonalitet. Det fullstendige programmet skal leveres i mappen din til slutt.*

*Når du har løst alle oppgavene under leverer du på Blackboard (der finner du også en mer detaljert beskrivelse av hvordan du skal levere). Du vil få tilbakemelding på besvarelsen din i etterkant og kan gjøre forbedringer i koden helt frem til endelig mappeinnlevering.*

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

## Før du begynner

Følgende krav og betingelser gjelder for alle oppgavene:

**Enhetstesting**
Du må lage solide 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.

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

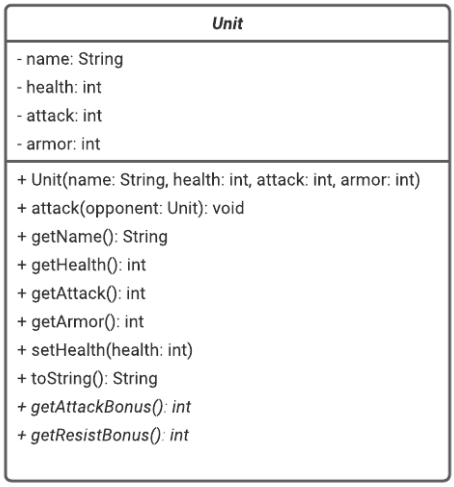
## Oppgave 1: Maven og git

Opprett et tomt Maven-prosjekt og gi prosjektet en fornuftig groupId og artefactId. Prosjektet skal følge JDK v11 eller høyere. Når du 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 kjøre med Maven uten feil.

Legg så prosjektet under lokal versjonskontroll. For hver av oppgavene under skal du gjøre minst én innsjekk (commit). Husk at hver commit-melding skal beskrive endringene som er gjort på en kort og konsis måte. Hvis du 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 du leverer besvarelsen i BB (.git-katatlogen er en "usynlig" mappe i rot-katalogen til prosjektet ditt som inneholder all versjonshistorikk).

# Oppgave 2: Superklassen Unit

En armé vil bestå av ulike typer enheter med varierende forsvars- og angrepsevner. De ulike typene vil ha noen fellestrekk som det er naturlig å samle i en egen superklasse. Det første du skal gjøres er derfor å implementere den abstrakte klassen Unit:




*Figur 1: Den abstrakte klassen Unit*

Diagrammet viser at en enhet har fire attributter:

- name: et kort beskrivende navn, f.eks. "Swordsman" eller "Archer"
- health: en verdi som angir helsa til enheten. Verdien reduseres når enheten blir angrepet, og kan aldri være lavere enn 0.
- attack: en angrepsverdi som representerer enhetens våpen
- armor: en forsvarsverdi som beskytter under et angrep

En enhet kan angripe en motstander med metoden attack(opponent). Et angrep påfører motstanderen skade etter følgende formel:

$$health_{opponent} = health_{opponent} - (attack + attackBonus)_{this} + (armor + resistBonus)_{opponent}$$

Verdier knyttet til enheten som blir angrepet har subteksten *opponent*, mens *this* henviser til verdier for enheten som utfører angrepet. Resultatet fra formelen over vil være helsa til *opponent* etter selve angrepet.

En enhet har også aksessor-metoder for alle private attributter/felt, en mutator-metode for å endre helseverdien, samt toString() som returnerer en fornuftig tekstlig representasjon av enheten.

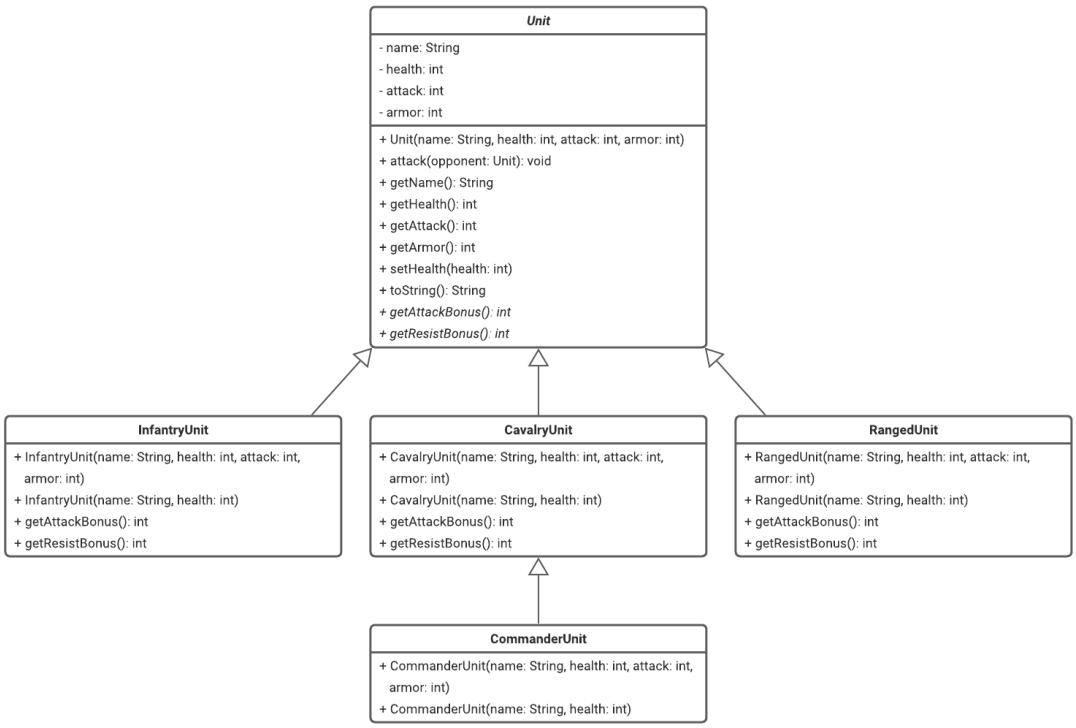
Klassen har dessuten to abstrakte metoder som brukes i formelen over:

- **getAttackBonus()**: bonus ved angrep. Legges til angrepsverdien når man angriper en annen enhet.
- **getResistBonus()**: bonus ved forsvar. Legges til forsvarsverdien til enheten som blir angrepet.

Husk å sjekk inn ny kode i lokal versjonskontroll.

## Oppgave 3: InfantryUnit, CavalryUnit, RangedUnit og CommanderUnit

I denne oppgaven skal du implementere fire spesialiserte typer enheter. InfantryUnit, CavalryUnit og RangedUnit arver alle fra Unit og implementerer de abstrakte metodene getAttackBonus() og getResistBonus(), mens klassen CommanderUnit arver direkte fra CavalryUnit:




*Figur 2: Spesialiserte typer enheter*

**InfantryUnit:**

- Den forenklede konstruktøren InfantryUnit(name, health) skal opprette en ny instans med attack=15 og armor=10.
- Enheten har en styrke i nærkamp (melee). Metoden getAttackBonus() skal returnere en verdi som representerer denne fordelen (f.eks. 2).
- Metoden getResistBonus() skal returnere en verdi som representerer en liten forsvarsbonus (f.eks. 1).

**RangedUnit:**

- Den forenklede konstruktøren RangedUnit(name, health) skal opprette en ny instans med attack=15 og armor=8.
- Enheten har en fordel fordi den kan angripe fra avstand (range). Metoden getAttackBonus() skal returnere en verdi som representerer denne fordelen (f.eks. 3).
- Enhetens forsvarsevne er basert på avstanden til fienden. Metoden getResistBonus() skal derfor returnere en betydelig verdi første gangen enheten blir angrepet (f.eks. 6). Verdien skal reduseres neste gang enheten blir angrepet (til f.eks. 4), fordi vi da antar at fienden er nærmere. Fra og med det tredje angrepet skal det returneres en standardverdi (f.eks. 2).

**CavalryUnit:**

- Den forenklede konstruktøren CavalryUnit(name, health) skal opprette en ny instans med attack=20 og armor=12.
- Enheten har en styrke første gang den angriper (charge) og i nærkamp (melee). Metoden getAttackBonus() skal returnere en verdi som representerer disse fordelene (f.eks. 4+2 ved første angrep, deretter 2).
- Enheten har en liten fordel når den blir angrepet i nærkamp. Metoden getResistBonus() skal returnere en verdi som representerer denne fordelen (f.eks. 1).

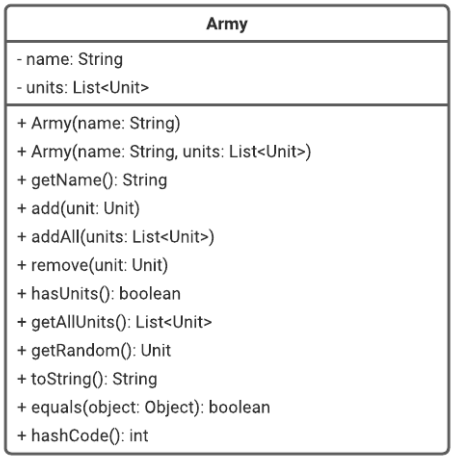
**CommanderUnit:**

- En CommanderUnit er en mer kapabel versjon av CavalryUnit. Den forenklede konstruktøren CommanderUnit(name, health) skal derfor opprette en ny instans med attack=25 og armor=15.
- Enheten har samme bonusberegninger som CavalryUnit.

Når du har implementert de fire entitets-klassene må du huske å lage enhetstester som verifiserer at koden fungerer. Husk også å sjekke inn endringene i lokal versjonskontroll.

# Oppgave 4: Army-klassen

En armé er en samling med enheter som kan angripe andre enheter i et slag. Klassen ser slik ut:




*Figur 3: Army-klassen*

Diagrammet viser at en armé har to attributter: et navn og en liste med enheter. Klassen har aksessor-metoder for å hente ut disse. Army-klassen har dessuten følgende metoder:

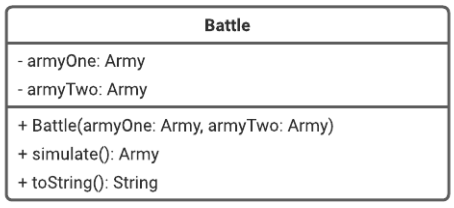
- add(unit): legger til en enhet i listen med enheter
- addAll(units): legger til alle enheter fra input-listen units til listen med enheter
- remove(unit): fjerner en enhet fra listen med enheter
- hasUnits(): returnerer true hvis listen med enheter har elementer, false hvis listen er tom
- getRandom(): returnerer en tilfeldig enhet fra listen over enheter. Tips: java.util.Random kan brukes for å generere en tilfeldig indeks.

Klassen har også standardmetoder som toString(), equals(object) og hashCode().

Og naturligvis: husk å skrive enhetstester og sjekke inn endringer i lokal versjonskontroll.

# Oppgave 5: Simulering av et slag med klassen Battle

Så må vi lage en klasse som simulerer et slag mellom to arméer:




*Figur 4: Simuleringsklassen Battle*

UML-diagrammet over skal være mer eller mindre selvforklarende. Sentralt i klassen står metoden simulate(). Simuleringen er relativt enkel: en tilfeldig enhet fra en armé angripe en tilfeldig enhet fra den andre arméen. Hvis en enhet har helse lik 0 fjernes den fra arméen. Denne dansen gjentas inntil en av arméene er utslettet. Til slutt returneres seierherren (armyOne eller armyTwo).

Også her må du huske å implementere enhetstester. Til slutt sjekker du inn i lokal versjonskontroll.

## Oppgave 6 (frivillig): Klient som kjører en simulering

I utgangspunktet skal enhetstestene fra oppgave 5 være nok for å teste simuleringen. Men hvis du vil kan du i tillegg lage en klient som oppretter arméer med enheter og kjører simuleringen fra kommandolinja.

## Eksempel på arméer og enheter

Dere står fritt til å navngi arméene og enhetene selv, men her får dere et enkelt forslag inspirert av spillet Warcraft:

**Human army:**

- 500x InfantryUnit med navn Footman og helse 100
- 100x CavalryUnit med navn Knight og helse 100
- 200x RangedUnit med navn Archer og helse 100
- 1x CommanderUnit med navn Mountain King og helse 180

**Orcish horde:**

- 500x InfantryUnit med navn Grunt og helse 100
- 100x CavalryUnit med navn Raider og helse 100
- 200x RangedUnit med navn Spearman og helse 100
- 1x CommanderUnit med navn Gul´dan og helse 180

# 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, installere, teste og kjøre uten at det feiler?
- Versjonskontroll med git:
  - Er prosjektet underlagt versjonskontroll med lokalt 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 superklassen Unit implementert iht oppgavebeskrivelsen?
- Er de spesialiserte enhetsklassene implementert iht oppgavebeskrivelsen?
- Er Army-klassen implementert iht oppgavebeskrivelsen?
- Er klassen Battle implementert iht oppgavebeskrivelsen og fungerer simuleringen som forventet?
- 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?