NTNU

Oblig 3

IDATA2001 Programmering 2
Institutt for IKT og realfag (IIR), NTNU

Arv og polymorfi II – abstrakte klasser og interface

I denne oppgaven skal vi jobbe videre med BonusMember-prosjektet fra Oblig 2. Vi skal forbedre løsningen ved å:

- Innføre et enkelt menybasert brukergrensesnitt (basert på RealEstate-prosjektet i IDATA1001 i høst)
- Legge til noen Unit-test klasser for å sikre at medlemsklassene **SilverMemebr**, **GoldMember** og **BasicMember** fungerer som de skal, **og** er robuste mot **feil bruk** (negativ testing)
- Innføre *abstrakt klasse*
- Innføre bruk av *Interface*
- Anvende **versjonskontroll** med **branches** og **merge**

Du vil også få et par tips om nyttige **plugins** til IntelliJ som kan hjelpe deg til å kode bedre og sørge for at du har god **kodestil**.

Oppgave 1 - versjonskontroll

Før du starter på selve kodingen, skal du sette prosjektet ditt under versjonskontroll. Dersom du allerede har gjort det i Oblig 2, kan du hoppe over de første stegene.

Trinn 1 – initialisere Git

Har du allerede satt prosjektet under versjonskontroll, kan du hoppe til Trinn 2.
Dersom du ikke har installert Git på din maskin, må du gjøre det nå. Gå til https://git-scm.com/downloads og last ned for ditt operativsystem, og installer.
Det kan også være greit å laste ned et grafisk brukergrensesnitt på toppen av Git, som f.eks. SourceTree (https://www.sourcetreeapp.com/).
For å **initialisere** Git for ditt prosjekt, kan du enten gjøre dette fra **kommandovindu/terminalvindu** eller via **SourceTree**.
Åpne kommandovindu/terminalvinduet. Gå til mappen der Java-prosjektet ditt ligger (rot-mappen). Så dersom du har laget en mappe som heter «.../Bruker/arne/BonusMembers», så gå til denne mappen. Skriv så:

```
git init
```

For å sjekke at alt er OK og at git ble initalisert, kan du skrive

```
git status
```

Git vil nå fortelle deg at du ikke har noen commits ennå, og du får en liste over filer som ikke er satt under versjonskontroll.

Legg nå filene dine til versjonskontroll (det er kun **kildekodefiler** og **prosjektfiler** som skal legges til, ikke **class-filer** og lignende) med følgende kommando:

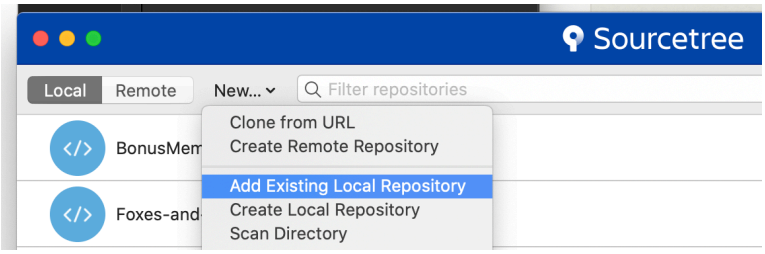
```
git add .
```

Sjekk med *git status* at filene nå er registrert for versjonskontroll (skal stå «new file:» foran hver av filene). Du er nå klar til å ***sjekke inn*** (committ) din første versjon:

```
git commit -m "Intial commit"
```

Gratulerer, du har sjekket inn din første versjon av koden din.

Dersom du nå ønsker å fortsette med et GUI-basert verktøy over Git (som f.eks. SourceTree), så kan du det. I SourceTree gjør du da som følger:



Bildet viser SourceTree på Mac. På Windows (og Linux) kan det se helt annerledes ut, men valget heter det samme; «Add Exsisting Local Repository». Velg dette, og du blir bedt om å peke på mappen som prosjektet ditt ligger i. Igjen er det «rot-mappen» til prosjektet ditt du skal velge.

Trinn 2 – Lag en gren (branch)

Når vi jobber med versjonskontroll, bruker vi **grener** (branches) for å ha kontroll på koden vår. Som minimum anbefales det at vi bruker 2 grener:

- **master** – Når du setter opp Git første gang, opprettes denne grenen som default. Denne grenen skal helst kun inneholde versjoner av koden din som er utgitt (releaset) til kunde altså ***formelle*** releases som har et versjonsnummer type v1.0.3 osv.
- **develop** – Grenen du utvikler langs på daglig basis. Når du er klar til å release applikajsonen din til kunde, ***fletter (merge)*** du denne grenen inn i master-grenen, og ***merker (tag)*** master med versjonen til applikasjonen din (v1.0 el.l.).

Opprett en ny gren kallt **develop** med følgende kommando:

```
git branch develop
```

For å se alle grenene (branches) du har, kan du skrive:

```
git branch
```

Den grenen du står på er merket med en '*' foran.

For å skifte til **develop**-grenen (det vi kaller å ***sjekke ut*** en gren):

```
git checkout develop
```

Skriv *git branch* igjen, og verifiser at du nå står på develop-grenen (som vil si at din *arbeidsmappe (engelsk: working folder*) nå er oppdatert med kode fra develop-grenen.

Videre i oppgaven skal du nå *sjekke inn (commit)* koden din for hver endring du gjør (etter å ha lagt til en metode, en klasse osv.). Hvor ofte du bør sjekke inn finnes det ikke noen fasit på, men koden du sjekker inn ***bør/må som minimum kompilere*** før du sjekker inn.

Oppgave 2 – Unit-test

Før vi begynner å gjøre endringer i koden, er det lurt å sørge for at eksisterende kode er tilstrekkelig testet.

Lag **positive** og **negative** tester for klassene **BasicMember**, **GoldMember**, og **SilverMember**. Å teste super-klassen **BonusMember** direkte gir lite mening. Denne blir testet indirekte via de tre subklassene.

Test spesielt logikken knyttet til registrering av poeng.

Oppgave 3 – Innfør brukergrensesnitt med meny

Basert på det meny-baserte grensesnittet du lagde i høst (RealEstate-prosjektet f.eks.), lag en ny klasse som skal representere brukergrensesnittet i løsningen, og implementer en meny med følgende valg:

1. Add bonus member
2. List all members
3. Upgrade qualified members
4. Register bonus points
5. Quit

Når du implementerer denne menyen vil du garantert få behov for både å flytte kode fra eksisterenede klasser (som f.eks. MemberArchive), og opprette nye metoder.

Tips 1: For å opprette et LocalDate-objekt basert på dag, måned og år (som du leser inn fra bruker), kan du bruke metoden ***of(int year, int month, int dayOfMonth)*** som finnes i klassen **LocalDate**. Denne metoden er **static** og returnerer et **objekt** av klassen LocalDate. Eksempel:

```
LocalDate date = LocalDate.of( 2020, 2, 5 ); //5/2-2020
```

Tips 2: Ved utskrift av alle medlemmene i arkivet, kan det være lurt å lage egne metoder or å skrive ut de ulike typene meldemmer, som feks **displayBasicMember()**, **displayGoldMember()** osv. Her er det da naturlig å bruke **instanceof** og **casting** for å velge riktig display-metode.

Tips 3: Lag en hjelpemetode som fyller medlemsarkivet med 5 medlemmer ved oppstart av applikasjonen.

Oppgave 4 – merge og release

Nå har du utvidet appliaksjonen din såpass mye at det er nok ny funksjonalitet til å lage en ny release til kunden. Vi lar dette bli v1.1. Gjør da som følger:

1. Kompiler, kjør alle Unit-testene, og test manuelt og verifiser at applikasjonen din er **release-ready**.
2. Sørg for at alle endringer er sjekket inn (committed) til repositoriet (i develop-grenen).
3. Flett så **develop**-grenen inn i **master-grenen** ved først å flytte deg til master (sjekke ut master):

```
git checkout master
```

4. Flett så **develop** grenen inn i **master**. Det vil si det du *egentlig* gjør nå er å flette develop sammen med koden du nå har i arbeidskatalogen din (working folder). Dersom det oppstår **konflikter** vil du måtte rette opp konfliktene, og så sjekke inn. Dersom ingen konflikter oppstår, trenger du ikke å sjekke inn noe etter merg.

```
git merge develop
```

5. Til slutt skal vi **merke** (**tag**) denne versjonen av koden din med «v1.1»:

```
git tag -a v1.1 -m "Release 1.1 with menu"
```

6. Før vi returnerer tilbake til **develop**-grenen vår og fortsetter utviklingen

```
git checkout develop
```

Oppgave 5 – refactoring og abstrakt klasse

a) Studer super-klassen **BonusMember**. Er det noe i denne klassen som **ikke** er felles for **samtlige** subklasser?
Gjør nødvendige endringer/forbedringer kompiler og sjekk inn koden i Git-repositoriet ditt.

b) Klassen BonusMember er en klasse det ikke vil være naturlig å lage instanser (objekter) av. Den er innført for å være en ***superklasse*** for de faktiske medlemstypene (basic, silver og gold). Hva bør vi gjøre med klassen for å gjøre det umulig å lage objekter av klassen?
Gjør endringen, kompiler og sjekk inn koden i Git-repositoriet ditt.

c) En vesentlig forskjell mellom klassene **BasicMember**, **SilverMember** og **GoldMember** er måten registrering av poeng gjøres. For BasicMember registreres poengene direkte (500 poeng blir 500

poeng), for SilverMember og GoldMember blir poeng som skal registreres *skalert opp* med en *faktor* ved registrering. Metoden **registerPoints()** i klassen **BonusMember** er derfor både ulik for alle medlemstyper, og det fines strengt tatt ingen fornuftig **default** implementasjon av metoden. Metoden bør derfor gjøres **abstrakt**.
Gjør nødvendige endringer/forbedringer kompiler og sjekk inn koden i Git-repositoriet ditt.

Oppgave 6 – refactoring og Interface

Har du lurt på hva som **egentlig** skjer i en for-each løkke, og som gjør at vi kan bruke enhver collection klasse (ArrayList, HashSet osv) direkte i en for-each løkke?

```
ArrayList<Person> persons = new ArrayList<>();

for (Person p : persons)
{
     System.out.println("Name: " + p.getName());
}
```

Det er fordi for-each-løkka bruker **Iteratoren** til ArrayList. Men hvordan vet for-each-løkka at ArrayList har en Iterator? Fordi ArrayList **implementerer interfacet *Iterable*** ;-) (https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html). Så dersom du lar **MemberArchive**-klassen implementere interfacet **Iterable**, og du re-definerer (override) metoden **iterator()** til å returnere iteratorene til samlingen du har valgt å bruke internt i MemberArchive, så kan du i brukergrensesnitt-klassen skrive:

```
MemberArchive memberArch;

// …… code ……

for (BonusMember bm : memberArch)
{
     System.out.println("Points: " + bm.getPoints());
}
```

Gjør nødvendige endringer/forbedringer kompiler og sjekk inn koden i Git-repositoriet ditt.

Oppgave 7 – Sortering

Legg til et valg i menyen som skriver ut listen over bonusmedlemmer **sortert** etter bonusboeng.
Her kan du bruke **sorted()**-metoden i **Stream**. Denne metoden vil sortere elementene i en stream i hehold til **"natural order"**. Dersom elementene f.eks. er av String, betyr "natural order" i alfabetisk rekkefølge.
Dersom elementene er av en slik type at det er vanskelig å definere en "natural order" uten videre, må vi ta i bruk **Comparable**-interfacet.

Vår klasse **BonusMember** er et eksempel på et slikt element. Dersom vi ønsker at den naturlige rekkefølgen å sortere medlemmene på er etter bonuspoeng, må vi la BonusMember implementere interfacet **Comparable<BonusMember>**. Dette interfacet definerer 3 metoder som da må re-defineres (overrides):

- **equals()** – skal returnerer *true* dersom dette objektet er likt et annet objekt (equals to) og *false* om ikke likt.
- **hashCode() –** som skal returnere en tilsynelatende unik tallkode (en hash) basert på verdiene til alle feltene i objektet.
- **compareTo(BonusMember otherMember)** – som sammenligner dette objektet med *otherMember* i forhold til om dette objektet skal sorteres før *otherMember* eller etter. Dersom dette objektet er *mindre enn* otherMember-objektet, returneres -1 (et negativt tall). Er de like, returneres 0, og er dette objektet *større enn* otherMember-objektet, returneres 1.

Implementer **Comparable**-interfacet i din kode og re-definer de tre metodene. Her er et eksempel: https://www.concretepage.com/java/jdk-8/java-8-stream-sorted-example.

Implementer menyvalget "List members by bonus points" i koden din. Det kan da være lurt å lage en metode i MemberArchive-klassen som f.eks. har følgende signatur:

```
public List<BonusMember> getArchive();
```

Da kan du bruke denne som følger (i brukergrensesnitt-klassen din):

```
MemberArchive memberArch;

// …… code ……

memberArchive.getArchive()
     .stream()
     .sort()
     .forEach(……skriv ut detaljer om medlem ......);
```

Her er noen lenker til som kan være nyttige:

https://howtodoinjava.com/java8/stream-sorting/
https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#compare-T-T-

Når du er ferdig, sjekker du inn koden i Git-repositoriet ditt, og lager en ny release v1.2.

Bonus oppgave – Lamda

På samme måte som at ArrayList og HashSet osv har en metode **forEach()** som kan ta et lamda-uttrykk som vil bli utført på hvert element i samlingen, så kan også du innføre en **forEach()**-metode i **MemberArchive**-klassen din. Dersom du har brukt en HashMap internt i MemberArchive for å holde på medlemmene, vil en forEach-metode kunne se slik ut:

```
public void forEach(Consumer<? super BonusMember> action)
{
  this.members.values().forEach(action);
}
```

Har du brukt en ArrayList<BonusMember>, vil forEach() se slik ut:

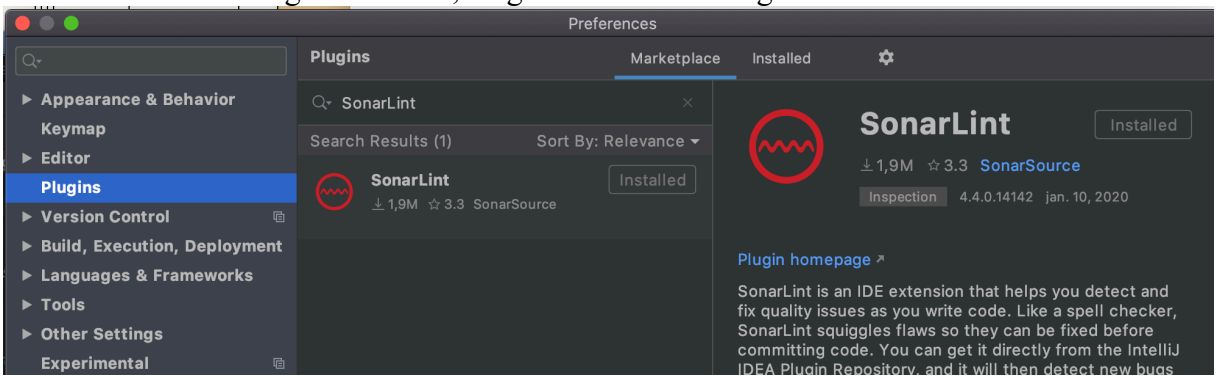
```
public void forEach(Consumer<? super BonusMember> action)
{
  this.members.forEach(action);
}
```

Implementer **forEach()**-metoden i din kode, skriv om koden som skriver ut alle medlemmene til å bruke lambda-uttrykk.

Kodekvalitet – plugins til IntelliJ

SonarLint

SonarLint (https://www.sonarlint.org/) er en plugin til IntelliJ som sjekker koden din for sikkerhetsbrudd, «smelly code» osv og bidrar til å øke kvaliteten i koden din. Integrert i IDEen vil du få god oversikt over hvilke regler du har brutt, og for hver regel får du en detaljert beskrivelse av **hvorfor** denne regele finnes, hvorfor den er viktig, samt eksempel på hvordan du **ikke** bør kode, samt forslag til hvordan du **heller** bør kode.
Gå til Preferences/Plugins i IntelliJ, velg «Market Place» og søk etter SonarLint:

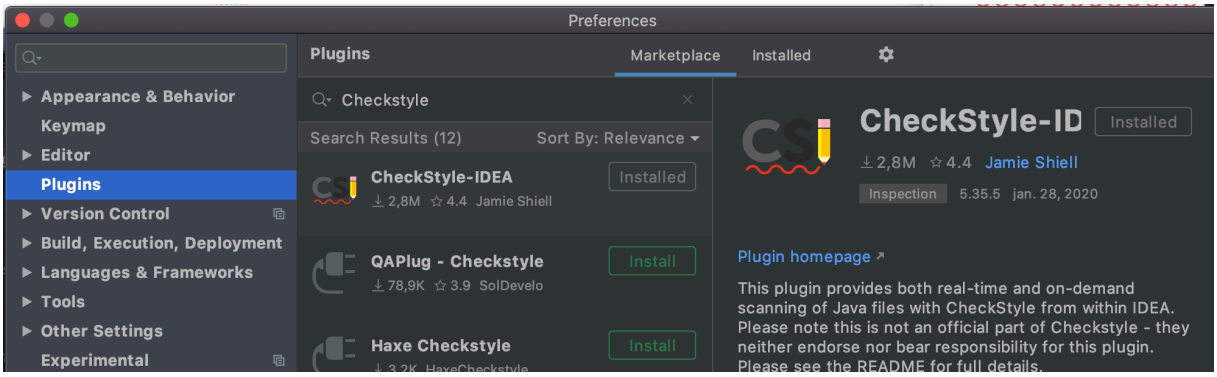


CheckStyle

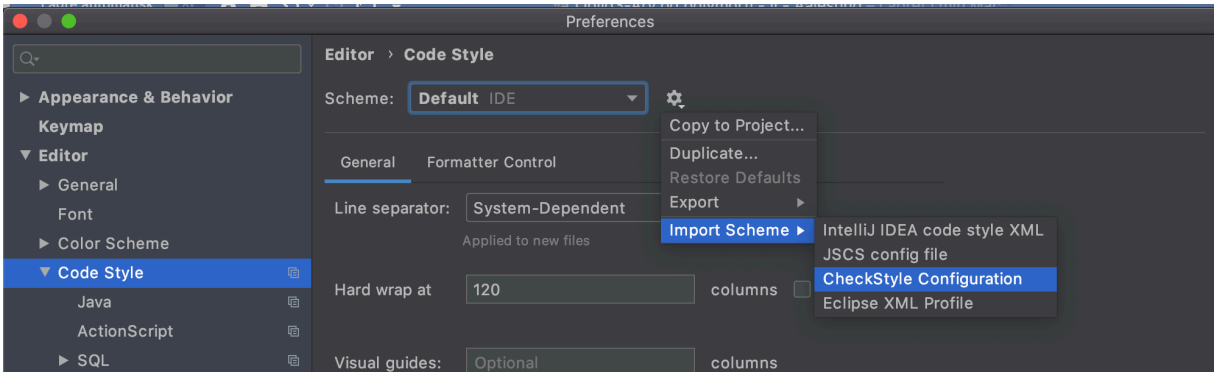
CheckStyle er et verktøy som kontrollerer at du har fulgt **en kodestandard**. Her snakker vi helt ned på detaljnivå som hvor stort et innrykk er (2 eller 4 karakterer), hvor '{'-paranteser skal stå, at det er luft mellom operander og operatorer (ikke a+b men a + b), at **samtlige** metoder som er publi har javaDoc, **og** at JavaDoc er fullstendig skrevet (med @param, @return osv).

Selve kodestandarden er konfigurert i en XML-fil, som du selv kan sette opp.

CheckStyle-plugin i IntelliJ kommer default med 2 kodestiler: Google og Sun (nå Oracle), så da kan du få testet om din kode ville ha passert kodestilsjekken til Google ;-)



NB! Husk at IntelliJ sin editor i utgangspunktet er satt opp til å følge en gitt kodestil. Det er derfor lurt å *kalibrere* IntelliJ slik at den innebygde kodestilen i IntelliJ er i synk med f.eks. Google checkstyle.



Du må da ha tilgang til checkstyle-filen som definerer reglene. Du finner både sun og Google versjonene her:

https://github.com/checkstyle/checkstyle/tree/master/src/main/resources
I tillegg er de lagt ut i BlackBoard under Ressurser.

Med disse to verktøyene installert, er du meget godt rustet til å skrive god kode med høy kvalitet og som er skrevet i henhold til en etablert kodestil. En enkel måte å sikre noen poeng ekstra til eksamen ☺