

Et større programeksempel

Hvordan løse et reelt problem
med en objektorientert fremgangsmåte

Plan for forelesingen

- Beskrive en større problemstilling
- Planlegge programmet
- Skrive koden, én klasse om gangen
- Teste og kjøre programmet
- En kommentar rundt personvern
- Underveis: rette opp feil, lure på hvor vi er og hvor vi skal, svare på spørsmål og kommentarer fra dere ...
 - Kort sagt: mye rart kan hende når man programmerer live!

En større problemstilling

- Mange sykdommer er delvis arvelige
- Dette skyldes at gener finnes i ulike varianter (som vi arver fra våre foreldre), hvor noen gir økt risiko for bestemte sykdommer
- Vi skal i dag skrive et program som sjekker tusenvis av gen-varianter (DNA-endringer) for en pasient mot hundrevis av potensielle sykdommer

Problemstillingen

Problemstillingen

- Ditt DNA kan ses på som en tre milliard lang streng

Ditt DNA: AGCTAT

Problemstillingen

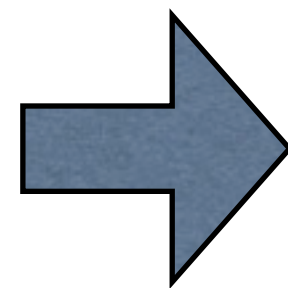
- Ditt DNA kan ses på som en tre milliard lang streng
 - På noen indekser i strengen har du en annen bokstav enn den vanlige (la oss kalle det en mutasjon)

Vanlig: ACCGAT
Ditt DNA: AGCTAT
Indeks: 012345

Problemstillingen

- Ditt DNA kan ses på som en tre milliard lang streng
 - På noen indekser i strengen har du en annen bokstav enn den vanlige (la oss kalle det en mutasjon)
 - Vi bryr oss ikke om selve strengen eller bokstavene, kun indeksene (posisjonene) hvor du har en endring

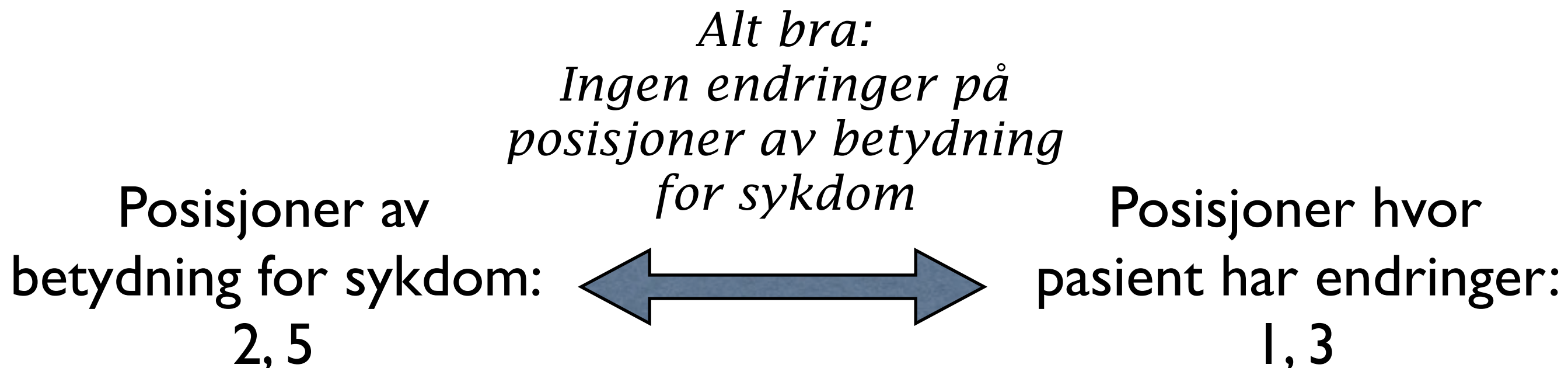
Vanlig: ACCGAT
Ditt DNA: ACCTAT
Indeks: 012345



Posisjoner hvor
du har endringer:
1, 3

Problemstillingen (forts.)

- Noen deler av DNA har betydning for sykdom
- Også for sykdommer bryr vi oss kun om posisjoner (indekser av strengen som er menneskets DNA)
 - For hver sykdom har vi en liste av posisjoner hvor avvik er av betydning
 - Vi ønsker å se om en pasient har endringer på posisjoner assosiert med en bestemt sykdom



Programmet vi skal skrive

"Vi skal sjekke tusenvis av DNA-endringer for en pasient mot hundrevis av potensielle sykdommer"

- Håndtere posisjoner av betydning for en sykdom:
 - Lager en klasse: **Sykdom**
- Håndtere hundrevis av ulike sykdommer:
 - Lager en klasse: **SykdomsKatalog**
 - *(som inneholder mange Sykdom-objekter)*
- Håndtere posisjoner hvor en pasient har DNA-endringer:
 - Lager en klasse: **Pasient**
 - *(en del likheter med klassen sykdom, siden begge håndterer posisjoner)*

Begynne fra toppen eller fra bunnen

- Fra toppen: beskrive hele programmet på overordnet nivå (planlegge alle klasser)
 - Bryr oss bare om grensesnitt - klasser og metodenavn
 - Fordel av å se helheten og se at ulike deler av programmet fungerer sammen
- Fra bunnen: gjør oss ferdig med én klasse av gangen
 - Bryr oss bare om en liten bit av det hele om gangen
 - Fordel av å slippe å ta alt innover seg med en gang. Ser at en bit virker og kan legges bort før man tenker på neste bit.
 - (Vi begynner fra bunnen i dag - kunne godt valgt fra toppen)

Test-drevet utvikling

- Et program bør testes på flere nivå
 - Enhetstest: sjekke at hver enkelt klasse virker etter planen
 - Integrasjonstest: sjekke at flere klasser virker sammen
 - Applikasjonstest: at programmet virker som forespeilet
- Det intuitive er kanskje å skrive programmet før testen
 - Men det motsatte også mulig og kalles test-drevet utvikling
 - Man skriver da enhetstester før selve klassene
- Vi skriver test av en klasse Sykdom før selve klassen

Grensesnittet til klassen Sykdom

"Håndtere posisjoner av betydning for en sykdom"

- Må kunne registrere en posisjon av betydning
 - `def leggTilPosisjon(self, posisjon):`
- Må kunne sjekke om en gitt posisjon er registrert
 - `def erAssosiert(self, posisjon):`
- Praktisk å kunne sette navn ved oppretting av objekt
 - `def __init__(self, navn):`

En test av klassen Sykdom

- Vi lager et objekt av klasse Sykdom
- Vi registrerer et par posisjoner (f.eks. 10 og 20)
- Vi sjekker etterpå at posisjonene 10 og 20 finnes i objektet, mens posisjonen 15 ikke finnes
- Selve testen skriver vi i en fil "*test_sykdom.py*"
- Vi forsøker å kjøre testen - får selvfølgelig en feil (klassen Sykdom finnes ikke)
- Først etter at testen feilet begynner vi å skrive Sykdom

Representasjon for klassen Sykdom

- Må ta vare på registrerte posisjoner:
 - *self._posisjoner*
- Må kunne lagre et navn på sykdommen:
 - *self._navn*

Klassen Sykdom i UML

Sykdom

- navn: str

- posisjoner: list

+ __init__(navn:str)

+ leggTilPosisjon(posisjon:int)

+ erAssosiert(posisjon:int): bool

+ hentNavn(): str

Tilbake til testen

- Hvordan og hvorfor skrive tester er et stort tema i seg selv - vi nøyer oss her med å skrape overflaten
- Det finnes solide rammeverk for å skrive enhetstester (*bl.a. unittest*) - vi nøyer oss her med en enkel tilnærming
- Når vi tror vi har implementert Sykdom riktig går vi tilbake til testen
 - Når testen kjører uten å feile er vi i mål!

Et litt kritisk blikk på utviklingen frem til nå

- Jeg lovet i starten av vi skulle skrive kode for et formål (å løse et reelt problem)
 - Klassen Sykdom kan jo virke ganske så meningsløs - man legger inn posisjoner og kan deretter sjekke om de finnes..
 - Viktig å klype seg i armen underveis - kode skal være nyttig!
 - Foreløpig svar: trust me - dette vil vise seg nyttig (vi kommer tilbake til hvordan)

Neste steg: mange sykdommer

"[skal sjekke mot] hundrevis av potensielle sykdommer"

- Lager en klasse: *SykdomsKatalog*
- Her er det kanskje lettest å tenke på ansvarsområde:
 - Skal holde orden på mange objekter av klasse Sykdom
 - Skal kunne lese inn data fra fil

Data fra fil:

```
5, pest  
12, pest  
13, kolera  
14, kolera  
20, pest
```

Grensesnittet til klassen SykdomsKatalog

- Opprette objekt og lese inn sykdomsdata fra fil
 - `def __init__(self, filnavn):`
- Sjekke om en gitt posisjon er assosiert med en eller annen sykdom i samlingen
 - `def sjekkMutasjon(self, posisjon):`
 - *(merk at samme formål som "erAssosiert" i klassen Sykdom)*

Representasjon for klassen SykdomsKatalog

- Holde orden på mange sykdommer
 - Hver posisjon lest fra fil skal lagres i et objekt for sykdommen den tilhører
 - Ut fra sykdomsnavn må man kunne finne frem til objekt for sykdommen
- *_sykdommer* = {}

Data fra fil:

```
5, pest  
12, pest  
13, kolera  
14, kolera  
20, pest
```

Private metoder

- Noen ganger er det fint å legge funksjonalitet i en metode som kun er ment for bruk i egen klasse
 - Man kan da lage en privat metode (navn starter med _)
- Vi vil lese sykdomsdata fra fil ved oppretting av objekt
 - Kan skrive koden for lesing fra fil direkte i konstruktøren
 - Kan likevel være ryddigere å ha egen metode for dette
 - Lager privat metode og kaller denne fra konstruktøren
 - *def _lesFil(self, filnavn):*

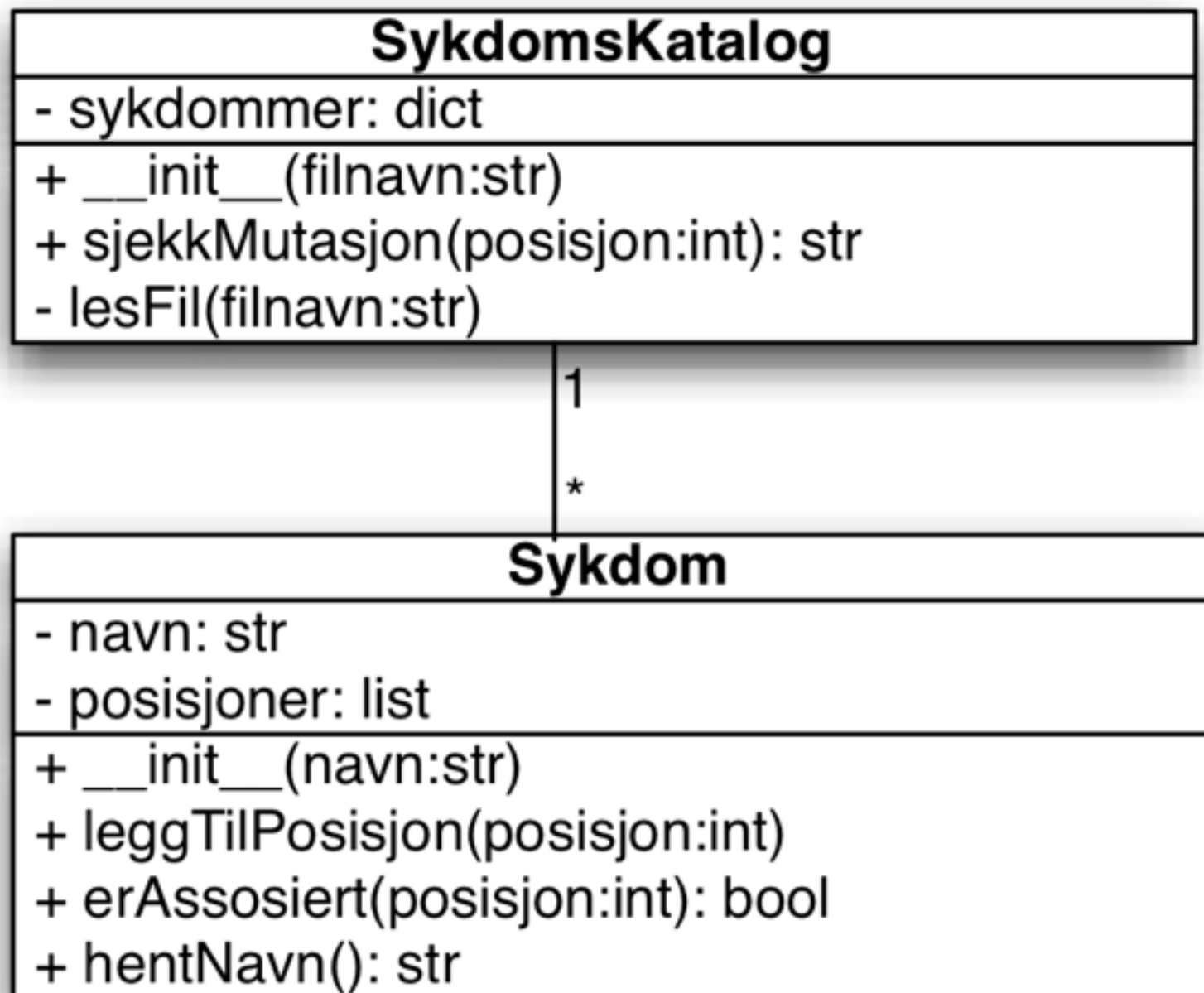
Lese sykdomsdata fra fil

- Innlesingen fra fil er kanskje det mest innfløkte i hele programmet
 - Selve lesingen er vanlig lesing av tekst-linjer
 - Det man leser inn må imidlertid sendes til separate objekter per sykdom (men med posisjoner tilhørende samme sykdom samlet)
- Fremgangsmåte for innlesing
 - Les linje med posisjon og tilhørende sykdom
 - Finn objekt for sykdommen, eller lag nytt
 - Legg lokasjonen inn i objektet

Data fra fil:

```
5, pest  
12, pest  
13, kolera  
14, kolera  
20, pest
```

Klassen SykdomsKatalog i UML



Enhetstest av Sykdomskatalog

- Lager denne gangen testen etterpå, men ellers på tilsvarende måte som for Sykdom
- Oppretter objekt, noe som også leser inn data fra fil:
 - `s = new Sykdomskatalog("sykdommer_test.csv")`
- Sjekker om posisjon samsvarer med innhold i fil:
 - `assert s.sjekkMutasjonForsteVersjon(14) == "kolera"`

Og så var det pasienten..

"sjekke tusenvis av DNA-endringer for en pasient"

- Blir ganske likt én enkelt sykdom
 - Skal stort sett bare holde en liste av posisjoner lest fra fil
- Vi begynner denne gangen utenfra (fra toppen)
 - Skriver en hoved-modul for analyse og ser hvilket grensesnitt det vil være praktisk å ha tilgjengelig for Pasient

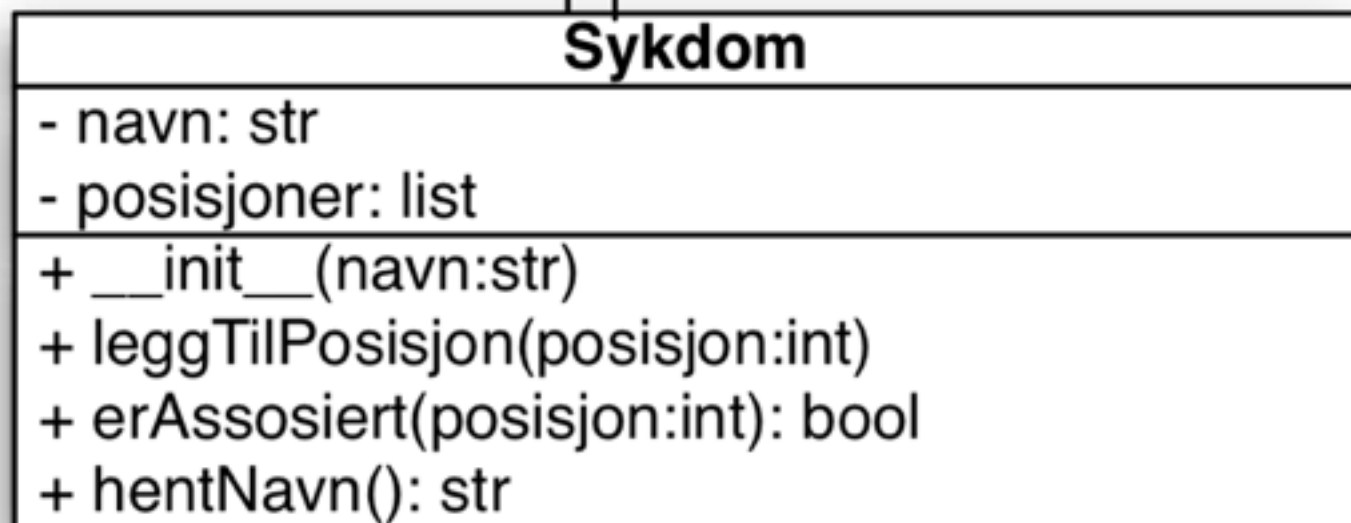
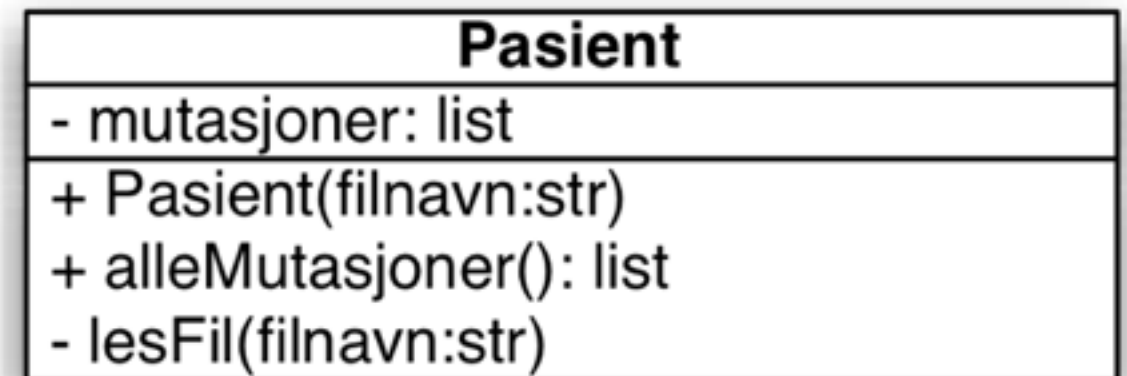
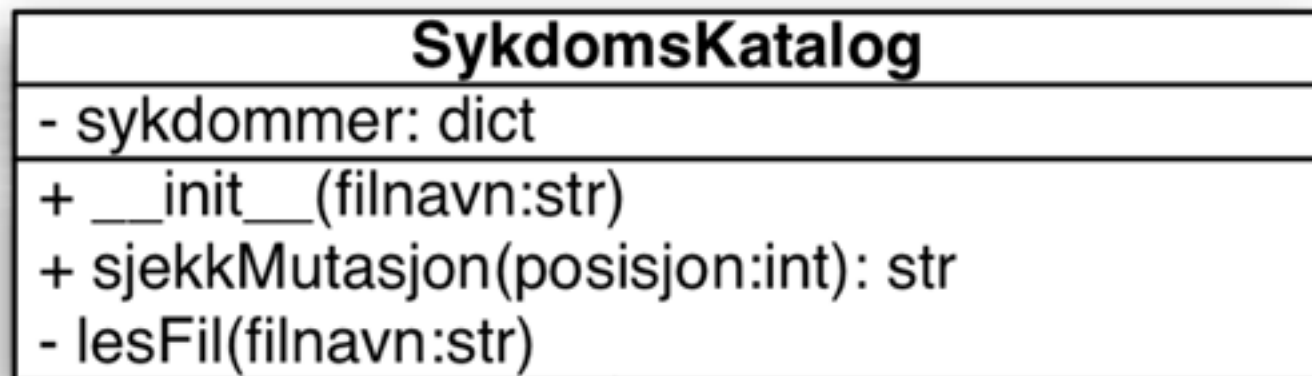
Hoved-modul for analyse

- Lage SykdomsKatalog-objekt med data fra fil
 - `katalog = SykdomsKatalog("syk.txt")`
- Lage Pasient-objekt med data fra fil
 - `pasient = Pasient("mutasjoner.txt")`
- Iterere gjennom en pasients mutasjoner (posisjoner)
 - `for posisjon in pasient.alleMutasjoner():`
- Sjekke hver mutasjon
 - `print(katalog.sjekkMutasjon(posisjon))`

Dermed følger grensesnittet for Pasient

- I `analyse.py`: `Pasient("mutasjoner.txt")`
 - *def __init__(self, filnavn):*
- I `analyse.py`: `for posisjon in pasient.alleMutasjoner():`
 - *def alleMutasjoner(self):*
 - (og for å kunne loope må altså metoden returnere en liste)

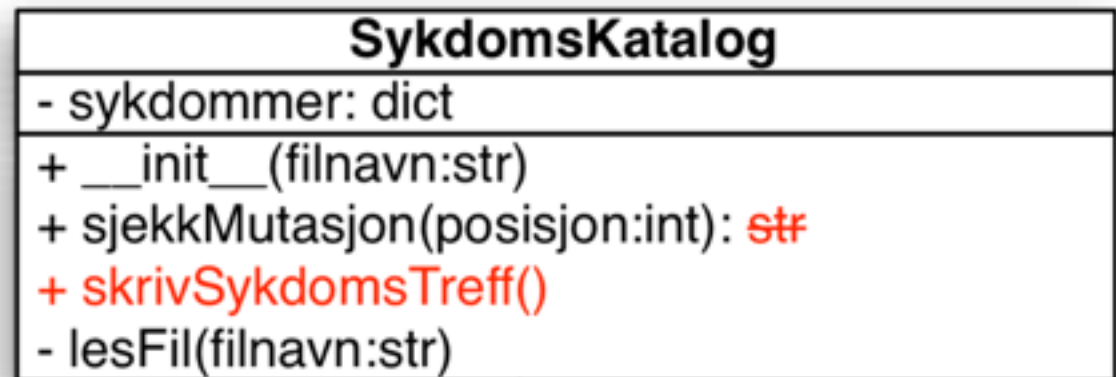
Alle tre klassene i UML



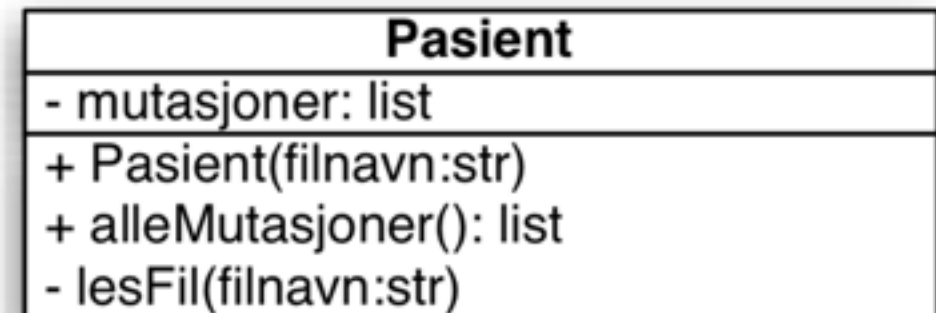
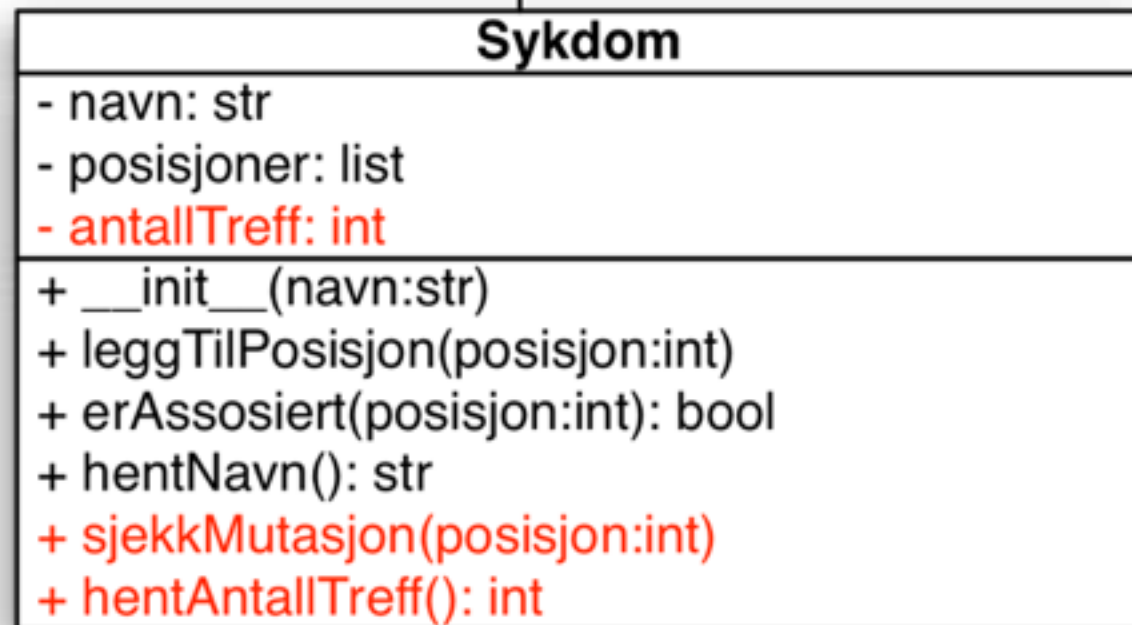
En liten justering av programmet

- I forrige versjon av Analyse skrev vi treff underveis:
 - *print(katalog.sjekkMutasjon(posisjon))*
- Vi vil heller samle opp og skrive antall treff for hver sykdom til slutt:
 - Sykdom må ha instansvariabel *antallTreff* som inkrementeres i en metode *sjekkMutasjon*
 - SykdomsKatalog trenger nå ikke returnere noe fra metoden *sjekkMutasjon*, men trenger ny metode *skrivSykdomsTreff*
 - Pasient kan være som før
 - *Analyse.py* kaller *katalog.skrivSykdomsTreff()* etter løkka

De justerte klassene i UML



1
*



Kjøre programmet

- Vi kan nå kjøre `analyse.py` med test-data
 - Sier at både pest og kolera kan være aktuelt
- Men det hadde jo vært mer meningsfullt med ekte data
 - Alt vi trenger å gjøre er å skaffe ekte datafiler

Ekte data

- Studier på tusenvis av ulike sykdommer er samlet og lett tilgjengelig for nedlasting
 - <https://www.ebi.ac.uk/gwas/>
- Mange personer ("pasienter") har delt sine personlige data (mutasjoner) gjennom et prosjekt på Harvard:
 - <http://www.personalgenomes.org/>
- Vi henter data for tusenvis av sykdommer, samt personlige data (mutasjoner) for Justin L. Smith
 - Justin burde kanskje oppholde seg mest innendørs
 - Merk: dette er ekte data, men ikke ment som klinisk analyse

En kommentar rundt personvern

- Posisjonene hvor en person har endringer er i utgangspunktet sensitive data
 - For å undersøke endringer hos en pasient på et norsk sykehus må man få godkjenning og følge strenge rutiner
- Dataene vi har brukt i dag er ekte, fra et prosjekt hvor personer frivillig har offentliggjort sine gendata
 - I utgangspunktet høres dette veldig raust og fint ut
 - Men hva synes dere om at vi kunne site her og analysere hvilke sykdommer Justin L. Smith kan ha risiko for?
(merk at formålet vårt var å lære programmering og diskutere personvern, så analysen er ikke fin-innstillt..)
 - Hva om forsikringselskap eller arbeidsgiver gjør det samme?

Konklusjon

- Objektorientert programmering lar oss pakke inn kompleksiteten i mange små deler
 - Store objektorienterte programmer har ofte mange klasser og metoder, men hver metode er vanligvis liten og enkel
 - Med en objektorientert fremgangsmåte kan man ofte løse store problemer ett steg av gangen
- Mulighetene for å koble informasjon gjør det viktig å tenke gjennom hva man deler av personlige data
- Det dere har lært i faget gjør dere i stand til å løse viktige problemer!