UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Design og Testbarhet

IN1010 2022 Eyvind W. Axelsen

Tillegg av Siri Moe Jensen til 25.4.2023 (grønn bakgrunn)

```
public class GodEttermiddag {
   public static void main(String[] args) {
      System.out.println("Hei på deg!");
      System.out.println("Du bør kunne lese dette - sitter du for langt bak?");
   }
}
```

Hvem er jeg?

Eyvind W. Axelsen
(eyvinda@ifi.uio.no)

Førsteamanuensis II ved Ifi, UiO

- Det vil si, jeg er deltidsansatt
- Programmering og Software Engineering (PSE)-gruppa (10. etasje)
- Underviser *IN3040 – Programmeringsspråk* til høsten

CTO/IT-direktør ved Fürst Medisinsk Laboratorium

- Norges største medisinske laboratorium, > 450 ansatte
- > 15 000 pasientprøver per dag, > 200 000 analyser per dag
- Lager "ordentlige" programmer daglig i språk som C#, TypeScript og SQL

Mål for denne forelesningen

Se viktigheten av et godt design

Forstå hva et testbart design er

Kunne skrive enkle programmer på en testbar måte

Dessuten (i lysark med grønn bakgrunn)

- Vise koblinger til andre deler av IN1010 (og andre emner)
- Flere design-relaterte begreper og eksempler

➢ Den originale forelesningen til Eyvind (uke 12; 2022) ligger med lysark og opptak sammen med denne (litt utvidete) presentasjonen på semestersiden uke 13; 2023

ca ukenr (kalender)	IN1030 Systemer, krav og konsekvenser	IN1010 v-23 Objektorientert programmering	IN2000 Software Engineering med prosjektarbeid
		Uke 3: Forkurs: VSCode, git, linux	
3			Kotlin – bygger på Java fra IN1010
4		Uke 1-2: Fra Python til Java	
5			git
6		Uke 6-7: Arv og interface	
7			Android Studio -MVVM (~MVP og MVC)
8			
9			
10		Uke 9: Oppstart prosjektarbeid oblig 4	
11	Smidige praksiser og teamarbeid		
12			
13			
		Uke 12: GUI 2 m/ MVC	
15			
16	OO design, UML klasse og sekvensdiagrammer, patterns	Uke 13: Design og testbarhet	
17			
18		Uke 15: Patterns	
19			
20			

Hva er design?

Oppgaver på ulike nivåer for å lage et IT-system eller en hel IT-arkitektur

IN1000
IN1010

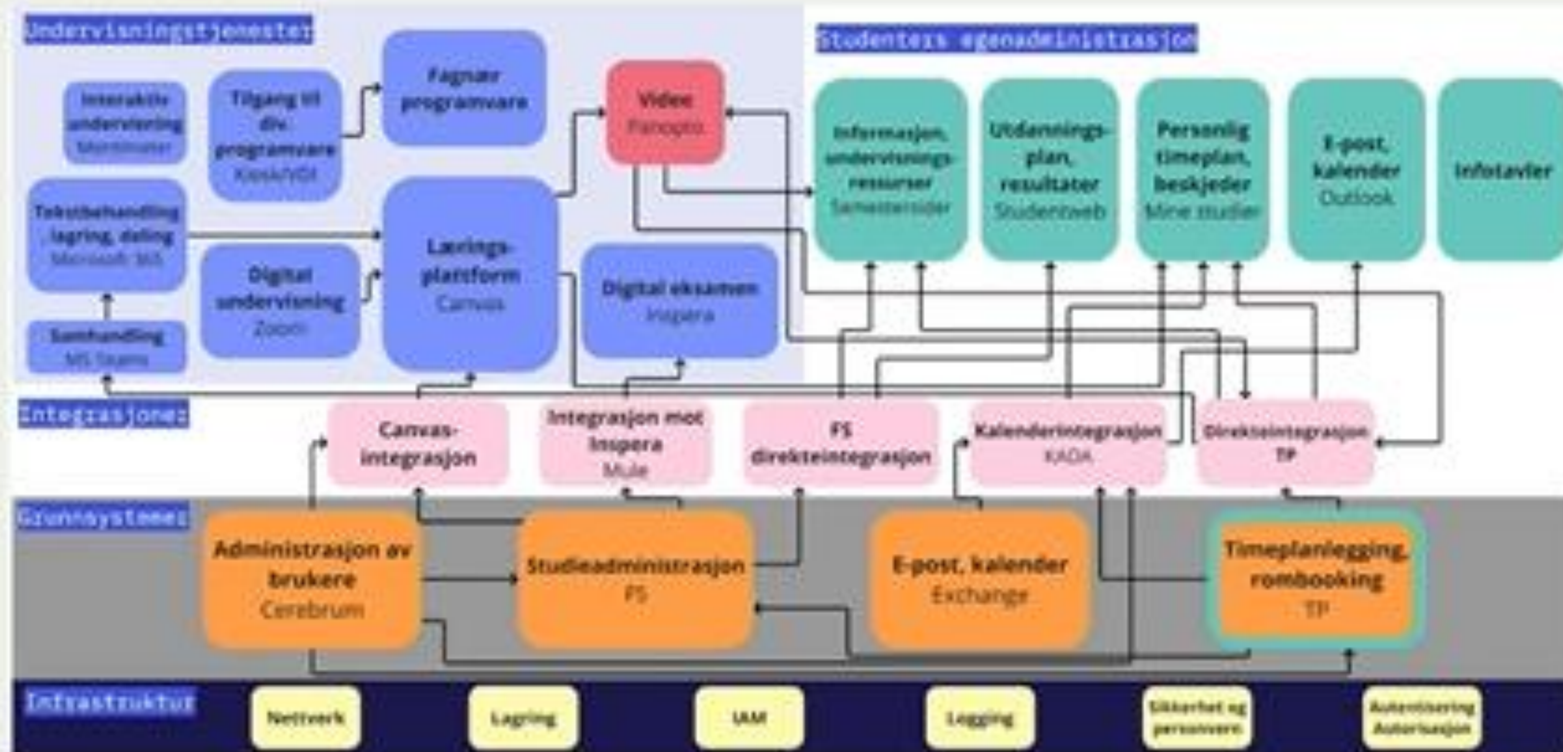
- Skrive en funksjon der parametere, returverdi og semantikk er oppgitt
- Skriv en klasse med et definert grensesnitt
- Definere grensesnittet til en klasse
- Identifisere fornuftige klasser for et program
- Bestemme arbeidsdeling og relasjoner mellom klassene
- :
- Identifisere samvirke og interaksjon for komponenter i et IT-system
- :
- :
- Bygge/ videreutvikle/ sette rammer for IT arkitektur i en større organisasjon
- (Enterprise / Virksomhet-arkitektur)

programmere

designe

arkitektur-
beslutninger

Kompleksitet ....



Hva ønsker vi av et design?

- Intuitivt/ **leservennlig**. Oversiktlig struktur, **konsistens** i hvordan klasser er delt opp, navngitt og forholder seg til hverandre. Gjenkjennbar modell.

- Gjenbrukbar kode
- Utvidbart, endringsvennlig program

- Testbart! Kommer tilbake til dette.

Design i IN1010

Ofte ferdig oppgitt i oppgaver, men endel hensyn og fremgangsmåter/ hjelpemidler/ prinsipper nevnes i forelesninger – f.eks:

- Innkapsling
  Veldefinerte grensesnitt, usynlig innmat (implementasjon)
  Klassen kontrollerer dataene sine. Reduserer problemer i ett program og ved endringer over tid ved gjenbruk (biblioteker)

- Arv eller komposisjon?
  Du har en klasse som bare delvis dekker et nytt behov. Lage en subklasse (utvidelse - **arv**) eller en ny klasse som refererer til et objekt av den eksisterende klassen (**komposisjon**)?

- Coupling og cohesion (se forrige uke + IN1030)
  Vi ønsker minst mulig avhengigheter mellom ulike klasser (lav kobling) MEN/ OG
  hver klassen **én** oppgave (høy cohesion/ samsvar/ sammenheng)

- Programmeringsmønstere som f eks MVC (egen forelesning, se også forelesning om GUI II)

Objektorientert design i Big Java: Kapittel 12

Dette kapittelet er ikke pensum i IN1010 – men gir en god oversikt over sentrale begreper, og tips om fremgangsmåter for programutvikling

12.1 Identifisere klasser og ansvarsområder. Cohesion ("samsvar").

12.2 Relasjoner mellom klasser. UML. Coupling (kobling) og avhengigheter.
NB: *Special Topic 12.3* kan være nyttig å lese først!!

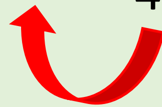
12.3 Eksempel. Javadoc.

12.4 Packages. Klasser som hører sammen. Lage, importere, stier.

Fra IN1000 uke 7 (introduksjon oop)

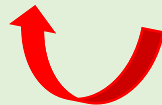
Fremgangsmåte: OOP

1. Identifiser aktuelle klasser (hva er sentrale «ting»/ begreper programmet skal behandle?)
2. Design klassens grensesnitt (hvilke tjenester ønsker jeg fra objekter av klassen?)
3. Design klassens datarepresentasjon (hvordan skal objektene representere dataene sine?)
4. Implementer (fyll ut) metodene (skriv klassen ferdig)



Refactoring/ refaktorering/ omstrukturering: Uunngåelig!

5. Lag et testprogram som oppretter et eller flere objekter og kaller på metodene i objektenes grensesnitt



Gjelder både på nivået vist her fra IN1000, og i større / mer komplekse systemer. Begynn et sted, og vær forberedt på endring

Kjente feil i programmer

Eller: hvorfor bry seg om design og testbarhet?

Therac 25, 1985-87

- En maskin som ga strålebehandling til pasienter
- En aritmetisk overflyt-feil gjorde at stråler med svært høy energi ble sendt direkte mot pasienter

Hvor stort er et heltall?

Hva er 2 147 483 647 + 1?
= -2 147 483 648?

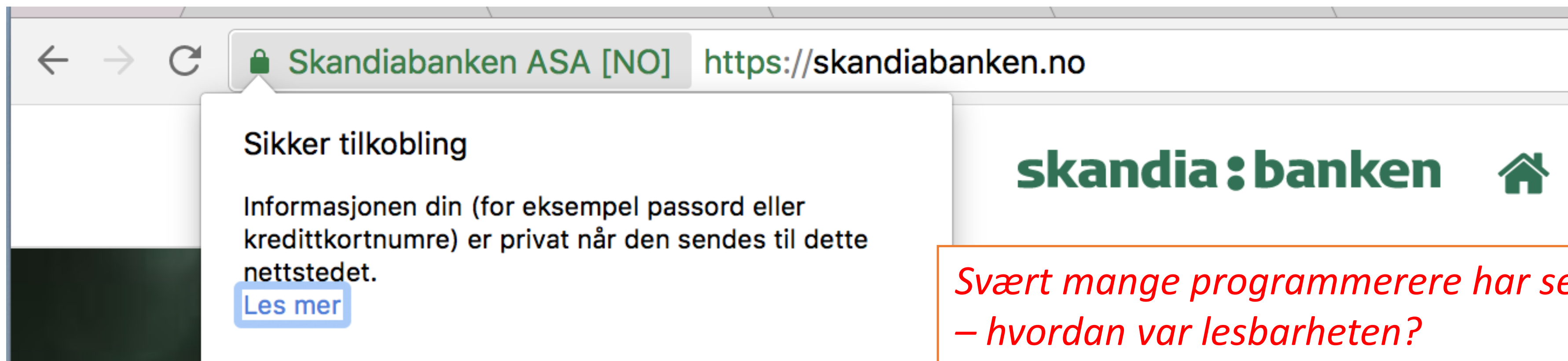
Hvilke ting er det viktig å teste for?
Dette henger også sammen med omgivelsene, som f.eks. valg av programmeringsspråk.

Kjente feil i programmer

Eller: hvorfor bry seg om design og testbarhet?

Heartbleed bug, 2014

- Feil i den svært populære OpenSSL-programvaren (PKI)
- *Én* stakkars programmerer glemte å validere lengden på forespørselen før svaret ble sendt
- Kunne føre til at *hva enn som var i minnet* ble sendt tilbake til en ondsinnet klient, f.eks passord, sertifikater, etc.



Svært mange programmerere har sett på koden uten å se feilen – hvordan var lesbarheten?

Hvor vanlig er det med feil i programmer?

Hva tror du? Hvor mange feil har et program i snitt?

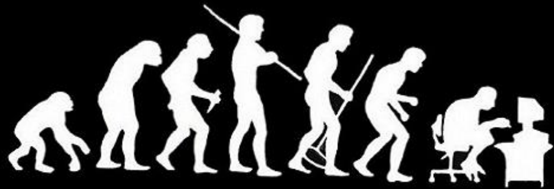
Hvor vanlig er det med feil i programmer?

En større studie av over 10 000 000 000(!) kodelinjer fant at det var ca 0,6 – 0,7 feil per 1000 linjer kode.

Åpen kildekode er, i gjennomsnitt, bedre enn kommersiell (lukket) kildekode.

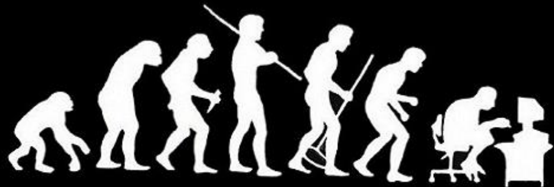
Kan god design og testbarhet gjøre at det blir færre feil i programmene våre?

[http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf]



Hva er programmering?

Ta 2 minutter og diskuter dette med personen ved siden av deg



Hva er programmering?

Programmering er en prosess hvor man ut fra en beskrivelse av et *beregnbart* problem utvikler et *eksekverbart* **program**.

(Som fortrinnsvis løser problemet.)

Hmm...

Hva er programmering?

Kristen Nygård og Ole-Johan Dahl

«To program is to understand»

- K. Nygård

Kildekode skal typisk leses mye oftere enn den skrives

"Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write."

— Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

- Godt design er viktig!
    - Men, hva er det?
    - Vi ser på noen hovedprinsipper

S O L I D

5 designprinsipper

S
O
L
I
D

- Single responsibility principle
  En klasse bør bare ha ett ansvarsområde, og dette bør være veldefinert

  cohesion

- Open/closed principle
  Komponenter bør være åpne for utvidelser, men lukket for modifikasjoner. Eksempel: klasser kontrollerer hva subklasser kan endre (private, protected, final, etc).

- Liskov substitution principle
  Objekter i et program bør kunne byttes ut med instanser av subtyper uten at dette endrer programmets korrekthet.

- Interface segregation principle
  Mange små grensesnitt er bedre enn store, monolittiske "super-grensesnitt"

- Dependency inversion principle
  Komponenter bør avhenge av abstraksjoner, og ikke konkrete implementasjoner

S O L I D

- Single responsibility principle
  En klasse bør bare ha ett ansvarsområde, og dette bør være veldefinert
- Open/closed principle
  Komponenter bør være åpne for utvidelser, men lukket for modifikasjoner. Eksempel: klasser kontrollerer hva subklasser kan endre (private, protected, final, etc).
- Liskov substitution principle
  Objekter i et program bør kunne byttes ut med instanser av subtyper uten at dette endrer programmets korrekthet.
- Interface segregation principle
  Mange små grensesnitt er bedre enn store, monolittiske "super-grensesnitt"
- Dependency inversion principle
  Komponenter bør avhenge av abstraksjoner, og ikke konkrete implementasjoner

Designprinsipp:
Single responsibility – ett ansvarsområde

- En klasse bør ha ett veldefinert ansvarsområde
- Når endringer må gjøres i et program, så bør det bare være én grunn til å endre en gitt klasse
- Prinsippet er ment å sikre robusthet og forståelighet

- Tett knyttet til prinsippet om «separation of concerns» → ikke bland alt sammen i en diger smørje!
  - Hvert «concern» er et område av applikasjonen, som bør ha en selvstendig implementasjon

Har denne klassen ett veldefinert ansvarsområde?

```
public class Ansatt {

    String fornavn;
    String etternavn;
    double lønn;
    String stilling;
    double stillingsProsent;
    Ansatt sjef;

    public Ansatt(String fornavn, String etternavn, double lønn, String stilling,
                  double stillingsProsent, Ansatt sjef) throws IllegalStateException {

        this.fornavn = fornavn;
        this.etternavn = etternavn;
        this.lønn = lønn;
        this.stilling = stilling;
        this.sjef = sjef;

        if(stillingsProsent > 100)
            throw new IllegalStateException("Kan ikke jobbe mer enn 100 %!");

        this.stillingsProsent = stillingsProsent;
    }

    public void justerLønnMed(double beløp) throws IllegalStateException {
        lønn = lønn + beløp;

        if(lønn > sjef.lønn)
            throw new IllegalStateException("Ingen kan tjene mer enn sjefen!");
    }
}
```

```java
public class Ansatt {

    String fornavn;
    String etternavn;
    double lønn;
    String stilling;
    double stillingsProsent;
    Ansatt sjef;

    public Ansatt(String fornavn, String etternavn, double lønn, String stilling,
                  double stillingsProsent, Ansatt sjef) throws IllegalStateException {

        this.fornavn = fornavn;
        this.etternavn = etternavn;
        this.lønn = lønn;
        this.stilling = stilling;
        this.sjef = sjef;

        if(stillingsProsent > 100)
            throw new IllegalStateException("Kan ikke jobbe mer enn 100 %!");

        this.stillingsProsent = stillingsProsent;
    }

    public void justerLønnMed(double beløp) throws IllegalStateException {
        lønn = lønn + beløp;

        if(lønn > sjef.lønn)
            throw new IllegalStateException("Ingen kan tjene mer enn sjefen!");
    }
}
```

Har denne klassen ett veldefinert ansvarsområde?

Sammenblanding av ansatt- og stillingsdata, og regler for stillinger

Har denne klassen ett veldefinert ansvarsområde?

```java
public class Ansatt {

    String fornavn;
    String etternavn;
    double lønn;
    String stilling;
    double stillingsProsent;
    Ansatt sjef;

    public Ansatt(String fornavn, String etternavn, double lønn, String stilling,
                double stillingsProsent, Ansatt sjef) throws IllegalStateException {

        this.fornavn = fornavn;
        this.etternavn = etternavn;
        this.lønn = lønn;
        this.stilling = stilling;
        this.sjef = sjef;

        if(stillingsProsent > 100)
            throw new IllegalStateException("Kan ikke jobbe mer enn 100 %!");

        this.stillingsProsent = stillingsProsent;
    }

    public void justerLønnMed(double beløp) throws IllegalStateException {
        lønn = lønn + beløp;

        if(lønn > sjef.lønn)
            throw new IllegalStateException("Ingen kan tjene mer enn sjefen!");
    }
}
```

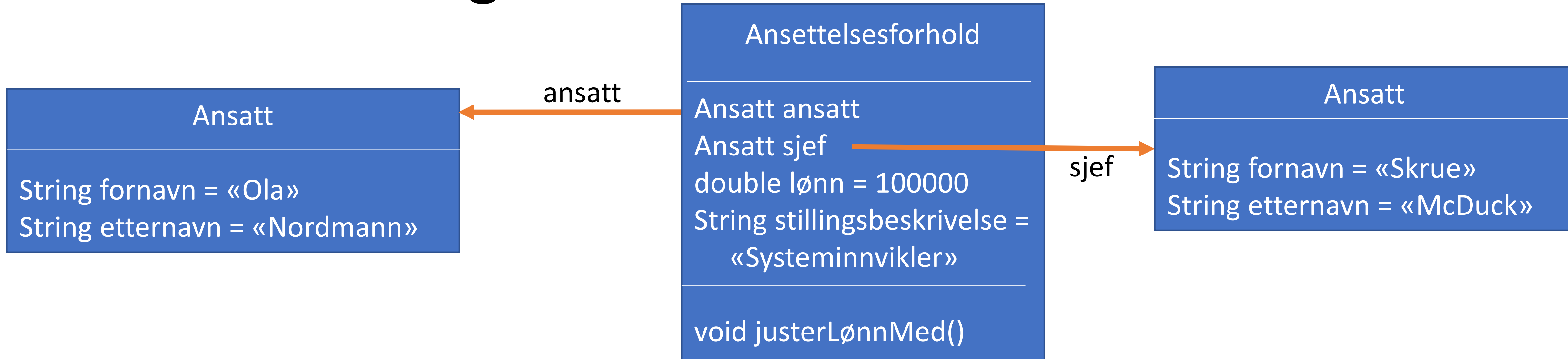
Sammenblanding med organisasjonskart og hierarki

Har denne klassen ett veldefinert ansvarsområde?

```java
public class Ansatt {

    String fornavn;
    String etternavn;
    double lønn;
    String stilling;
    double stillingsProsent;
    Ansatt sjef;

    public Ansatt(String fornavn, String etternavn, double lønn, String stilling,
                double stillingsProsent, Ansatt sjef) throws IllegalStateException {

        this.fornavn = fornavn;
        this.etternavn = etternavn;
        this.lønn = lønn;
        this.stilling = stilling;
        this.sjef = sjef;

        if(stillingsProsent > 100)
            throw new IllegalStateException("Kan ikke jobbe mer enn 100 %!");

        this.stillingsProsent = stillingsProsent;
    }

    public void justerLønnMed(double beløp) throws IllegalStateEx
        lønn = lønn + beløp;

        if(lønn > sjef.lønn)
            throw new IllegalStateException("Ingen kan tjene mer enn sjefen!");
    }
}
```

Sammenblanding med forretningslogikk

Et bedre design?



```java
class Ansettelsesforhold {
   String stilling;
   double stillingsProsent, lønn;
   Ansatt ansatt, sjef;

   public Ansettelsesforhold(Ansatt ansatt, double lønn, String stilling, double
         stillingsProsent, Ansatt sjef) throws IllegalStateException {
      this.ansatt = ansatt;
      this.lønn = lønn;
      this.stilling = stilling;
      this.stillingsProsent = stillingsProsent;
      this.sjef = sjef;

      if (stillingsProsent > 100)
         throw new IllegalStateException("Kan ikke jobbe mer enn 100 %!");

      this.stillingsProsent = stillingsProsent;
   }

   public void økLønnMed(double beløp) throws IllegalStateException {
      lønn = lønn + beløp;
      if (lønn > sjef.lønn)
         throw new IllegalStateException("Ingen kan tjene mer enn sjefen!");
   }
}
```

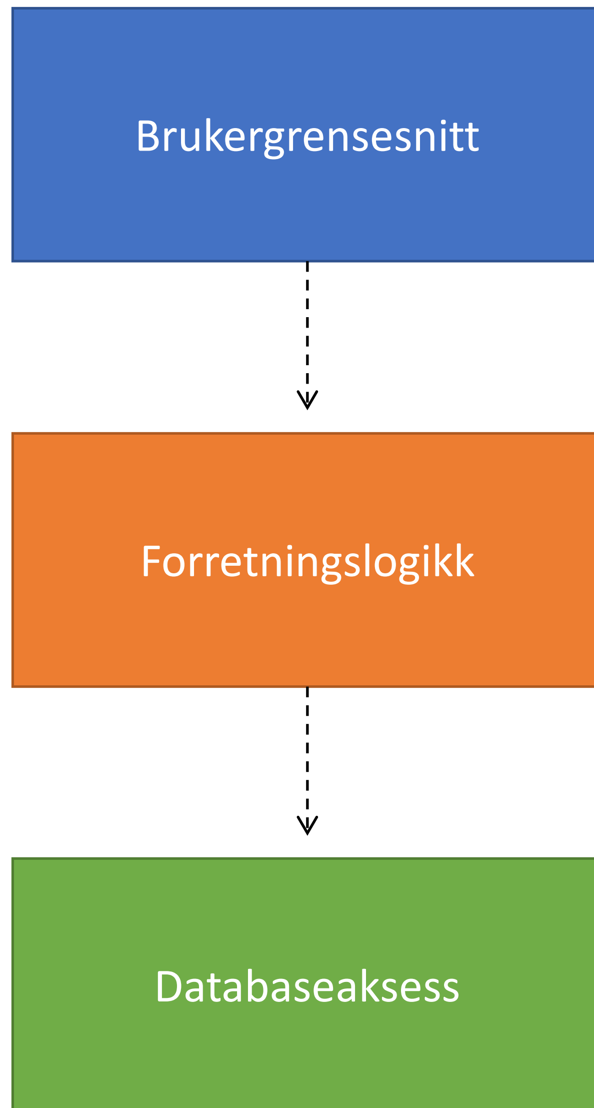
Designprinsipp:

Dependency inversion – omvendte avhengigheter

- Moduler på et høyt nivå bør ikke avhenge av moduler på lavere nivåer

- Abstraksjoner bør ikke avhenge av implementasjoner, men implementasjonene avhenger i stedet av abstraksjonene

- Nivået som trenger abstraksjonene eier dem



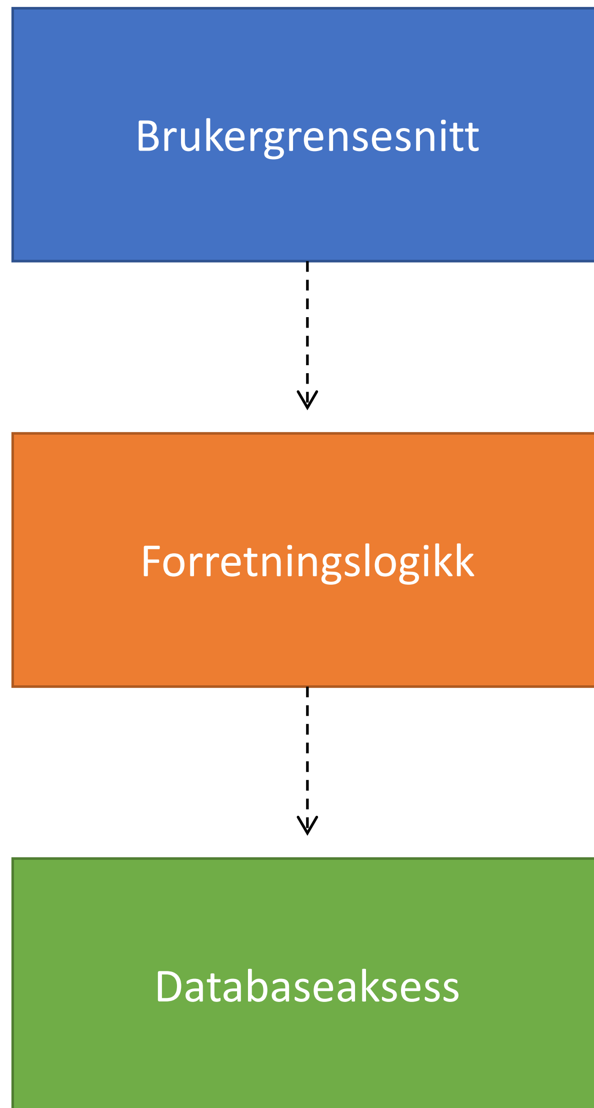
«Tradisjonell» design:

- Brukergrensesnittet er avhengig av forretningslogikk, som igjen er avhengig av et databasenært lag.
- Hver boks kan representere et sett av klasser

Hva er problemet med dette?

- Tette knytninger
- Vanskelig å teste komponenter i isolasjon

Avhengighet

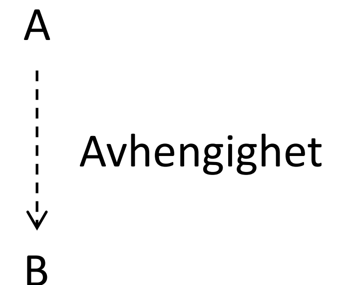


Avhengighet, sier du? Hva betyr det?

```
class A {
    B b = new B();
}

class B {
    /* ... */
}
```

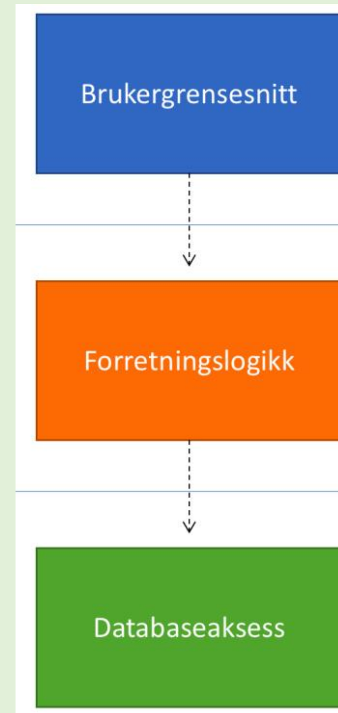
A har en avhengighet til B

(merk at denne er skjult, ikke synlig i A's grensesnitt)



Avhengighet

Litt mer om"avhengighet"



Konkret: Kjenner til, refererer til, kaller metoder i (se også Big Java 12.3)

- Blå klasse(r) kan ikke kjøres uten orange klasse(r)
- Orange klasser kan ikke kjøre uten grønne klasser

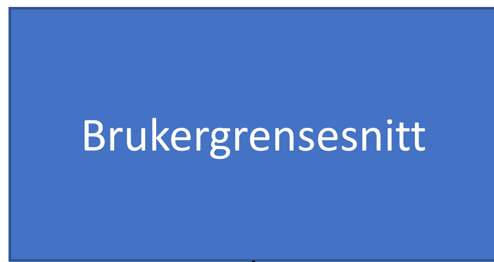
Konsekvenser

- Om grønn klasse erstattes av en gul klasse må alle referanser (og antakelig metodekall) til objekter i den grønne klassen endres i den orange klassen – selv om den nye gule klassen skulle ha akkurat samme funksjonalitet
- Om man ønsker å teste den orange klassen må man bruke den grønne – eller endre frem og tilbake i orange klasse før og etter test => ikke trygt

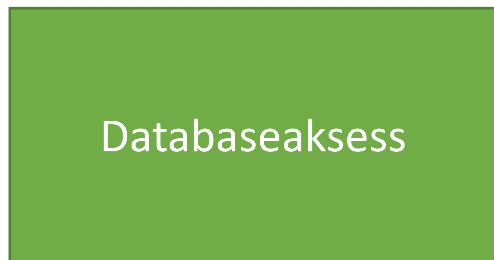
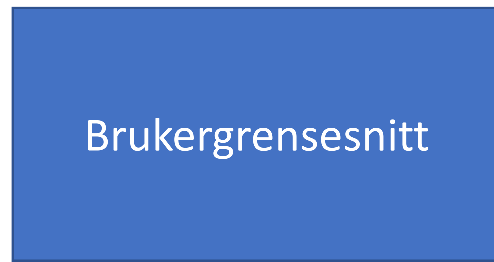
Et interface (grensesnitt) er (2):

- Et interface ligner en abstrakt klasse
- **Alle** metodene i et interface er abstrakte og polymorfe
- En interface inneholder **ingen** variabler eller annen datastruktur (men litt annet som vi ikke bruker i IN1010)
- En klasse som arver egenskapene til et interface må selv putte inn kode i alle de abstrakte metodene (og deklarere passende variabler som disse metodene bruker for å gjøre jobben sin).
- En klasse kan arve egenskapene til mange grensesnitt (men bare en klasse)

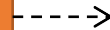
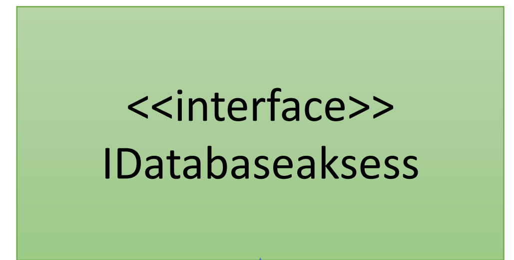
- Å arve (en samling metoder) = å spille en rolle
- Husk, med interface arves bare metodesignaturer
  - Implementasjon arves ikke

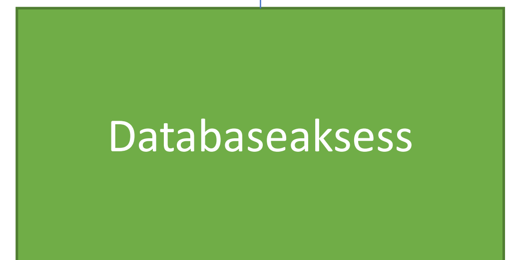
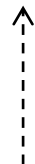

Brukergrensesnitt

Forretningslogikk


Databaseaksess

Brukergrensesnitt

<<interface>>
IForretningslogikk
Hvert lag eier abstraksjonene det er avhengig av

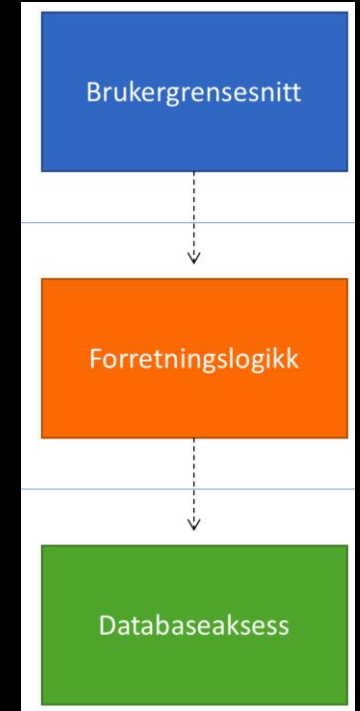
Forretningslogikk


<<interface>>
IDatabaseaksess


Databaseaksess

Avhengighet

Implementasjon

Uten DI

```
class Brukergrensesnitt {
   Forretningslogikk forretningslogikk = new Forretningslogikk();
   // ...
}

class Forretningslogikk {
   Databaseaksess dataaksess = new Databaseaksess();
   // ...
}

class Databaseaksess {
   // ...
}
```

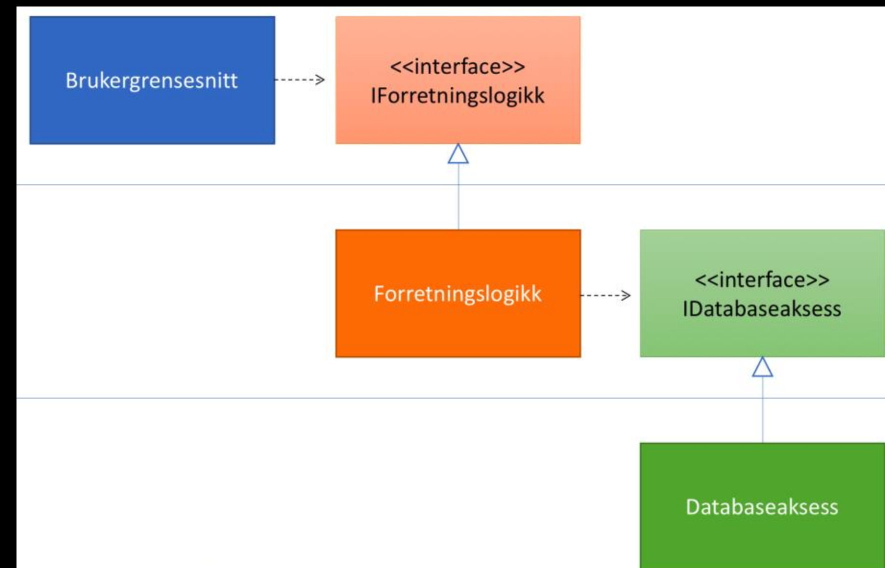


Med DI

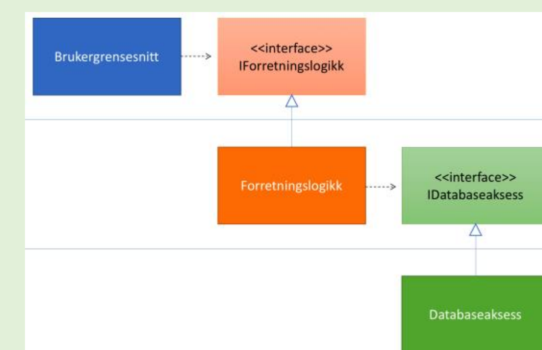
```
interface IForretningslogikk { /* ... */ }

class Brukergrensesnitt {
   public Brukergrensesnitt(IForretningslogikk forretningslogikk) { /* ... */ }
   // ...
}


interface IDatabaseaksess { /* ... */ }

class Forretningslogikk implements IForretningslogikk {
   public Forretningslogikk(IDatabaseaksess databaseaksess) { /* .. */ }
   // ...
}


class Databaseaksess implements IDatabaseaksess {
   // ...
}
```

Eksplisitt avhengighet til grensesnitt

Kan *byttes ut* f.eks. under testing



Hva er gjort?



- Instansvariabelens type i "blå klasse" er ikke lenger en bestemt klasse, men et interface – som kan implementeres av flere klasser
- Det fremgår av konstruktøren (det vil si i grensesnittet til blå klasse) at den krever et objekt som implementerer dette interfacet.

Det betyr

- Ingen endringer i blå klasse om orange klasse f.eks. erstattes av testkode som implementerer samme interface – blå klasse kan (enhets)testes uten at koden endres før og etter
- Bruken av interfacet er dokumentert av blå klasses grensesnitt - man trenger ikke gå inn i klassen og se på datarepresentasjonen for å oppdage det

Testbarhet – å legge til rette for programmatisk testing

Hvis man skal teste et program, så er første bud at kriteriene det testes for er klare

- Kriteriene for et riktig program må være
  - Utvetydige
  - Kvantitative/målbare
  - Verifiserbare i praksis

- Et testbart program bør være modularisert på en slik måte at hver komponent lar seg teste for seg
  - Enhetstesting – hver komponent testes i isolasjon
    - Ikke så enkelt som det kanskje høres ut som – dette må kodes inn eksplisitt!
  - Integrasjonstesting – samspillet mellom enkeltkomponenter testes

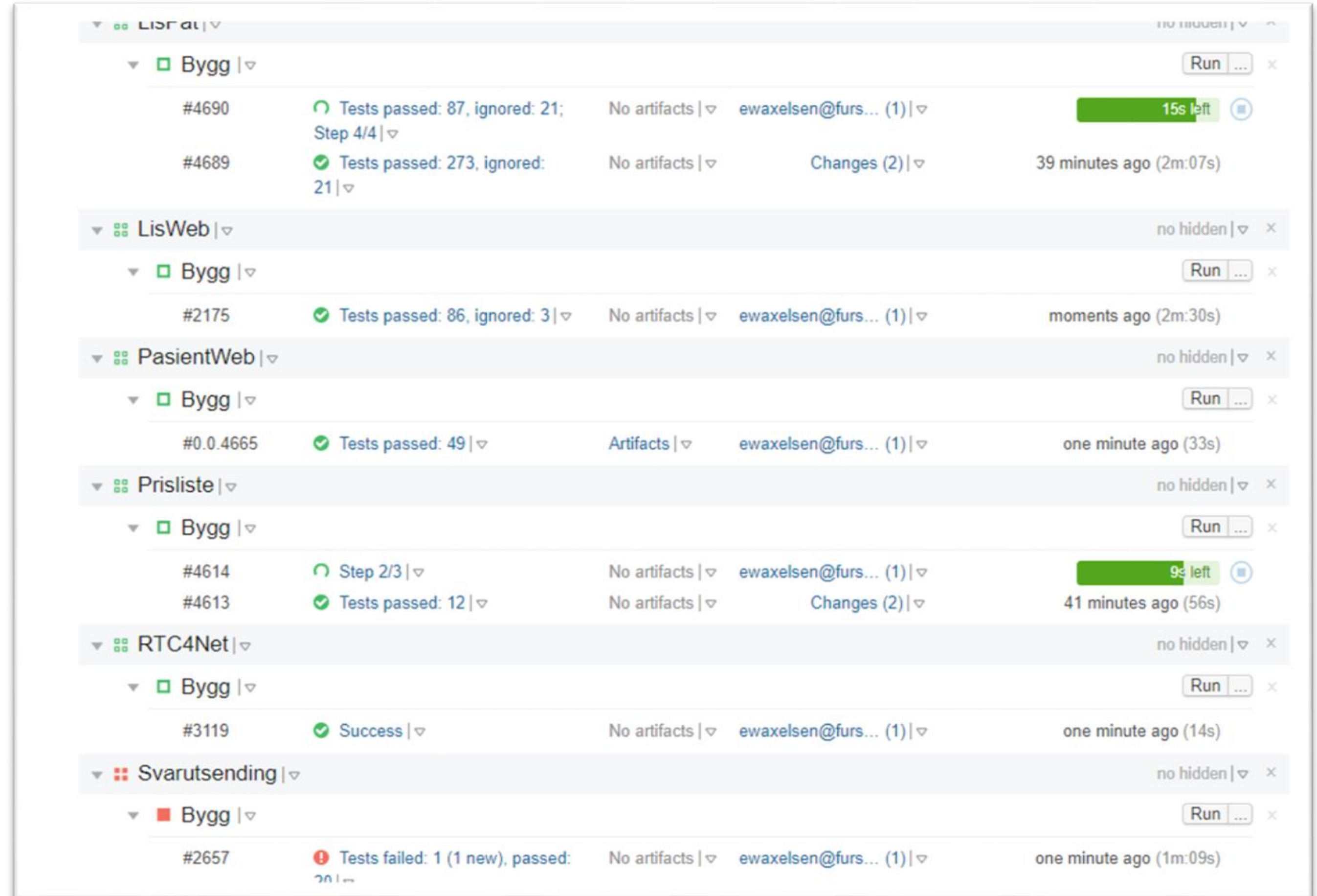
Har du skrevet en test før?

Hvordan tester du dine programmer?

- Snakk med personen ved siden av i 2 minutter

CI – Continuous Integration («kontinuerlig integrasjon»)

- Alle utviklere på et prosjekt sjekker inn/«commiter» kode kontinuerlig, f.eks. til git
- Hele prosjektet bygges automatisk ved hver innsjekk
- Tester kjøres automatisk ved hvert bygg for å forhindre regresjoner
- Regresjon: kode som tidligere fungerte har blitt ødelagt



Et eksempel: Studentlisteprinter

Krav til programmet:

- Skal definere et konsept «Student», med fornavn, etternavn og brukernavn
- Studentene skal skrives ut som en liste på papir, sortert etter brukernavn
- Formen på utskriften skal være slik:
  «Ola Olsen (olao)»
- Programmet skal være testbart

class Student
String fornavn String etternavn String brukernavn
String getFornavn() String getEtternavn() String getBrukernavn()

```java
class Program {
   public static void main(String[] args) throws PrintException, IOException {
      List<Student> studenter = new ArrayList<Student>();
      studenter.add(new Student("Dag", "Langmyhr", "dag"));
      studenter.add(new Student("Stein", "Gjessing", "steing"));
      studenter.add(new Student("Eyvind", "Axelsen", "eyvinda"));

      StudentPrinter studprint = new StudentPrinter();
      studprint.print(studenter);
} }

class StudentPrinter {
   public void print(List<Student> studenter) throws PrintException, IOException {

      for (Student student : studenter) {
         String data = student.getFornavn() + student.getEtternavn() + student.getBrukernavn();

         try (InputStream is = new ByteArrayInputStream(data.getBytes())) {
            DocFlavor flavor = DocFlavor.INPUT_STREAM.AUTOSENSE;
            PrintService service = PrintServiceLookup.lookupDefaultPrintService();
            // Create the print job
            DocPrintJob job = service.createPrintJob();
            Doc doc = new SimpleDoc(is, flavor, null);
            // Monitor print job events; for the implementation of PrintJobWatcher,
            PrintJobWatcher pjDone = new PrintJobWatcher(job);
            // Print it
            job.print(doc, null);
            // Wait for the print job to be done
            pjDone.waitForDone();
} } }
```

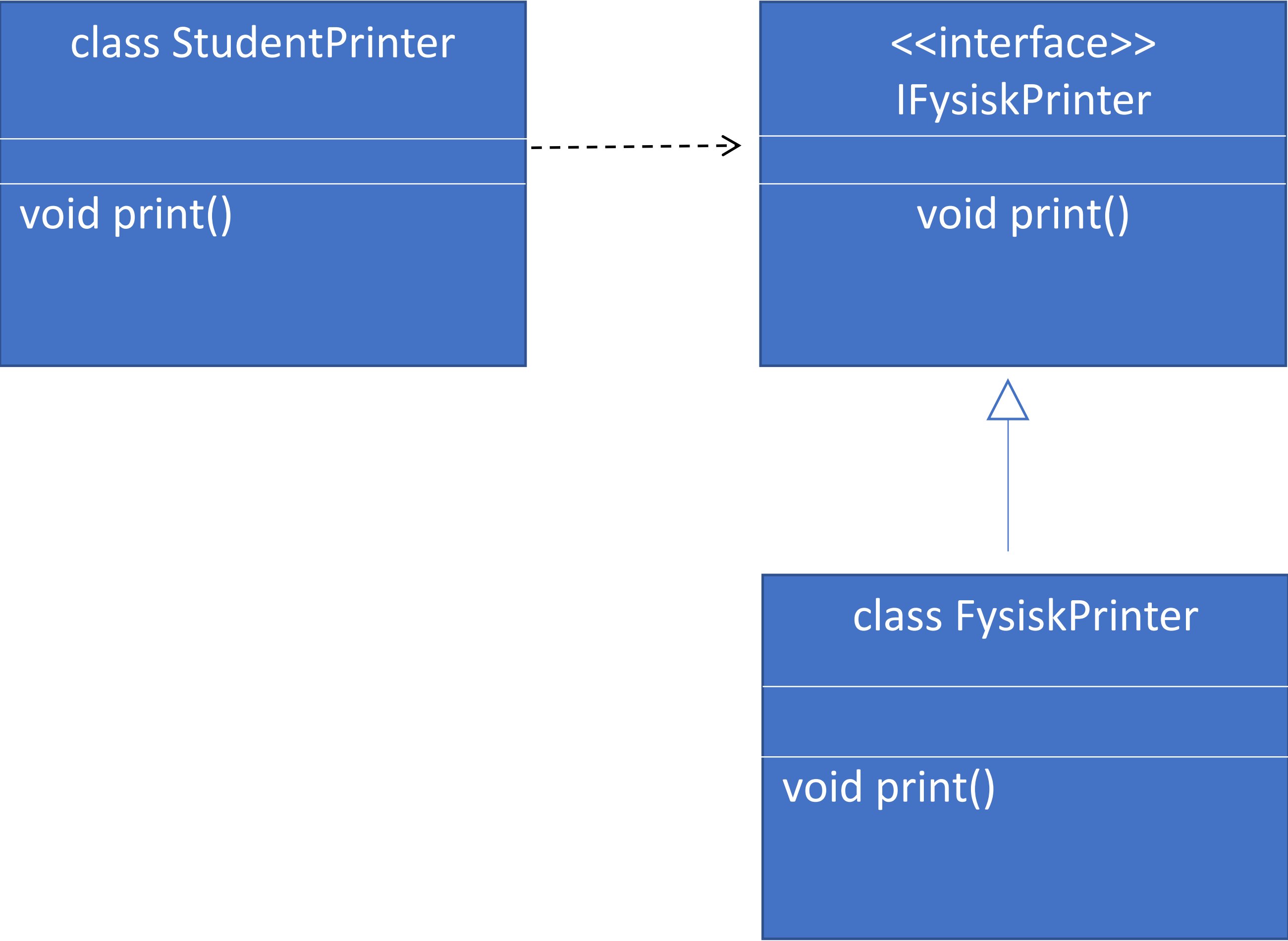
```java
class Program {
   public static void main(String[] args) throws PrintException, IOException {
      List<Student> studenter = new ArrayList<Student>();
      studenter.add(new Student("Dag", "Langmyhr", "dag"));
      studenter.add(new Student("Stein", "Gjessing", "steing"));
      studenter.add(new Student("Eyvind", "Axelsen", "eyvinda"));

      StudentPrinter studprint = new StudentPrinter();
      studprint.print(studenter);
} }

class StudentPrinter {
   public void print(List<Student> studenter) throws PrintException, IOException {

      for (Student student : studenter) {
         String data = student.getFornavn() + student.getEtternavn() + student.getBrukernavn();

         try (InputStream is = new ByteArrayInputStream(data.getBytes())) {
            DocFlavor flavor = DocFlavor.INPUT_STREAM.AUTOSENSE;
            PrintService service = PrintServiceLookup.lookupDefaultPrintService();
            // Create the print job
            DocPrintJob job = service.createPrintJob();
            Doc doc = new SimpleDoc(is, flavor, null);
            // Monitor print job eve
            PrintJobWatcher pjDone =
            // Print it
            job.print(doc, null);
            // Wait for the print jo
            pjDone.waitForDone();
} } }
```

- Skal definere et konsept «Student», med fornavn, etternavn og brukernavn
- Studentene skal skrives ut som en liste på papir, sortert etter brukernavn
- Formen på utskriften skal være slik:
  «Ola Olsen (olao)»
- Programmet skal være testbart
- Er det noe her du kunne tenke deg å endre på?

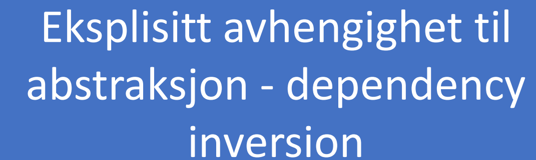
Observasjoner

- StudentPrinter-klassen gjør (minst) to ting:
  - Henter ut data som skal printes fra Student-objekter
  - Gjør fysisk utskrift
  - Vi prøver oss på å skille disse fra hverandre!
    - Single responsibility principle

- Det er dumt at det fyker papir ut på printeren for hver gang man kjører en (automatisert) test
  - Vi trekker ut fysisk print som et eget grensesnitt
  - Benytter dependency inversion for å sette opp avhengighetene


class StudentPrinter
void print()
<<interface>>
IFysiskPrinter
void print()
class FysiskPrinter
void print()

```java
interface IFysiskPrinter {
    public void print(String data) throws PrintException, IOException ;
}

class StudentPrinter {
    IFysiskPrinter fysiskPrinter;

    public StudentPrinter(IFysiskPrinter fysiskPrinter) {
        this.fysiskPrinter = fysiskPrinter;
    }

    public void print(List<Student> studenter) throws PrintException, IOException {

        // sortere på brukernavn:
        studenter.sort((s1, s2) -> s1.getBrukernavn().compareTo(s2.getBrukernavn()));

        // løp gjennom alle studenter, og skriv ut
        for (Student student : studenter) {
            String data = student.getFornavn() + student.getEtternavn() +
                    student.getBrukernavn();
            fysiskPrinter.print(data);
        }
    }
}
```



3 A-er: Arrange, Act, Assert

Oppsett for enhetstester

- Arrange: gjør alt oppsett for testen
- Act: utfør selve handlingen som skal testes
- Assert: sjekk at vi fikk forventet resultat

```java
class StudentPrinterTester {

    public void print_KaltMed3Studenter_SkriverUtStudenterIRiktigRekkefølge() throws Exception {

        // arrange
        DummyPrinter dp = new DummyPrinter();
        StudentPrinter eut = new StudentPrinter(dp);

        List<Student> studenter = new ArrayList<Student>();
        studenter.add(new Student("Dag", "Langmyhr", "dag"));
        studenter.add(new Student("Stein", "Gjessing", "steing"));
        studenter.add(new Student("Eyvind", "Axelsen", "eyvinda"));

        // act
        eut.print(studenter);

        // assert
        if(!dp.utskrift.get(0).equals("Dag Langmyhr (dag)")) throw new RuntimeException("Feil");
        if(!dp.utskrift.get(1).equals("Eyvind Axelsen (eyvinda)")) throw new RuntimeException("Feil")
        if(!dp.utskrift.get(2).equals("Stein Gjessing (steing)")) throw new RuntimeException("Feil");
    }
}
```

```java
class DummyPrinter implements IFysiskPrinter {
    public List<String> utskrift = new ArrayList<String>();

    public void print(String data) { utskrift.add(data); }
}
```

```java
class StudentPrinterTester {

    public void print_KaltMed3Studenter_SkriverUtStudenterIRiktigRekkefølge() throws Exception {

        // arrange
        DummyPrinter dp = new DummyPrinter();
        StudentPrinter eut = new StudentPrinter(dp);

        List<Student> studenter = new ArrayList<Student>();
        studenter.add(new Student("Dag", "Langmyhr", "dag"));
        studenter.add(new Student("Stein", "Gjessing", "steing"));
        studenter.add(new Student("Eyvind", "Axelsen", "eyvinda"));

        // act
        eut.print(studenter);

        // assert
        if(!dp.utskrift.get(0).equals("Dag Langmyhr (dag)")) throw new RuntimeException("Feil");
        if(!dp.utskrift.get(1).equals("Eyvind Axelsen (eyvinda)")) throw new RuntimeException("Feil")
        if(!dp.utskrift.get(2).equals("Stein Gjessing (steing)")) throw new RuntimeException("Feil");
    }
}
```

DI lar oss bytte ut den fysiske printeren med en dummy testversjon

Assert-delen sjekker at utskriften faktisk blir som spesifisert

```java
class DummyPrinter implements IFysiskPrinter {
    public List<String> utskrift = new ArrayList<String>();

    public void print(String data) { utskrift.add(data); }
}
```

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
```

```java
class StudentPrinterTester {

  @Test void print_KaltMed3Studenter_SkriverUtStudenterIRiktigRekkefølge() {

    // arrange
    DummyPrinter dp = new DummyPrinter();
    StudentPrinter eut = new StudentPrinter(dp);

    List<Student> studenter = new ArrayList<Student>();
    studenter.add(new Student("Dag", "Langmyhr", "dag"));
    studenter.add(new Student("Stein", "Gjessing", "steing"));
    studenter.add(new Student("Eyvind", "Axelsen", "eyvinda"));

    // act
    eut.print(studenter);

    // assert
    assertEquals("Dag Langmyhr (dag)", dp.utskrift.get(0));
    assertEquals("Stein Gjessing (steing)", dp.utskrift.get(1));
    assertEquals("Eyvind Axelsen (eyvinda)", dp.utskrift.get(2));
  }
}
```

```java
class DummyPrinter implements IFysiskPrinter {
    public List<String> utskrift = new ArrayList<String>();

    public void print(String data) { utskrift.add(data); }
}
```

```java
class StudentPrinterTester {

    @Test void print_KaltMed3Studenter_SkriverUtStudenterIRiktigF

        // arrange
        DummyPrinter dp = new DummyPrinter();
        StudentPrinter eut = new StudentPrinter(dp);

        List<Student> studenter = new ArrayList<Student>();
        studenter.add(new Student("Dag", "Langmyhr", "dag"));
        studenter.add(new Student("Stein", "Gjessing", "steing"));
        studenter.add(new Student("Eyvind", "Axelsen", "eyvinda"))

        // act
        eut.print(studenter);

        // assert
        assertEquals("Dag Langmyhr (dag)", dp.utskrift.get(0));
        assertEquals("Stein Gjessing (steing)", dp.utskrift.get(1)
        assertEquals("Eyvind Axelsen (eyvinda)", dp.utskrift.get(2
}
}
```

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage	
All Packages	221	84%	2970/3513	81%	859/1060
junit.extensions	6	82%	52/63	87%	7/8
junit.framework	17	76%	399/525	90%	139/154
junit.runner	3	49%	77/155	41%	23/56
junit.textui	2	76%	99/130	76%	23/30
org.junit	14	85%	196/230	75%	68/90
org.junit.experimental	2	91%	21/23	83%	5/6
org.junit.experimental.categories	5	100%	67/67	100%	44/44
org.junit.experimental.max	8	85%	92/108	86%	26/30
org.junit.experimental.results	6	92%	37/40	87%	7/8
org.junit.experimental.runners	1	100%	2/2	N/A	N/A
org.junit.experimental.theories	14	96%	119/123	88%	37/42
org.junit.experimental.theories.internal	5	88%	98/111	92%	39/42
org.junit.experimental.theories.suppliers	2	100%	7/7	100%	2/2
org.junit.internal	11	94%	149/157	94%	53/56
org.junit.internal.builders	8	98%	57/58	92%	13/14
org.junit.internal.matchers	4	75%	40/53	0%	0/18
org.junit.internal.requests	3	96%	27/28	100%	2/2
org.junit.internal.runners	18	73%	306/415	63%	82/130
org.junit.internal.runners.model	3	100%	26/26	100%	4/4
org.junit.internal.runners.rules	1	100%	35/35	100%	20/20
org.junit.internal.runners.statements	7	97%	92/94	100%	14/14
org.junit.matchers	1	9%	1/11	N/A	N/A
org.junit.rules	20	89%	203/226	96%	31/32
org.junit.runner	12	93%	150/161	88%	30/34
org.junit.runner.manipulation	9	85%	36/42	77%	14/18
org.junit.runner.notification	12	100%	98/98	100%	8/8
org.junit.runners	16	98%	321/327	96%	95/98
org.junit.runners.model	11	82%	163/198	73%	73/100

Report generated by Cobertura 1.9.4.1 on 12/22/12 2:25 PM.

JUnit Test Results

Statistics | Output

55 tests passed, 10 tests caused an error.

- com.commercehub.perspective.impl.housemodel.test.HousePerspectiveTest FAILED
  - testLoadPerspectivesStore passed (0.281 s)
  - testLoadPerspectivizedHouseWithPerspInsideRoom caused an ERROR (0.031 s)
  - testLoadPerspectivizedHouseWithPerspInsideCircuit passed (0.016 s)
  - testGetPropertyPaths caused an ERROR (0.0 s)
  - testGetPerspectiveIterator caused an ERROR (0.0 s)
  - testGetSubRealization caused an ERROR (0.0 s)
  - testGetSubRealizationIterator caused an ERROR (0.0 s)
  - testEditPerspective caused an ERROR (0.0 s)
  - testRemovePerspective caused an ERROR (0.0 s)
  - testGetContextualRelationship caused an ERROR (0.0 s)
  - testGetCircuitWithPerspective caused an ERROR (0.0 s)
  - testGetPerspective caused an ERROR (0.0 s)
- com.commercehub.perspective.impl.pojo.test.MetadataImplTest passed

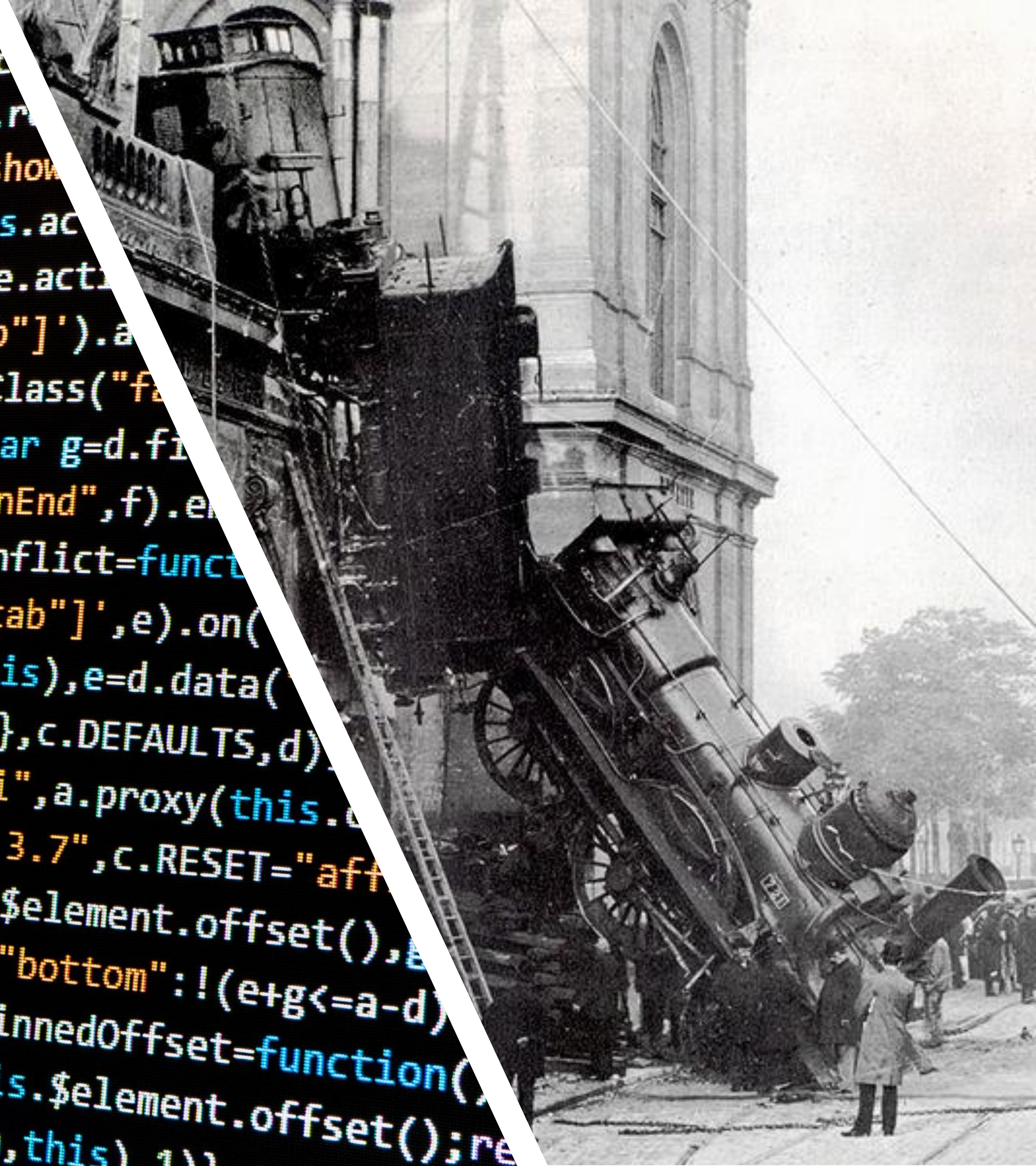
JUnit Test Results | Output

Vil du lære mer om testing av programvare?

- Sjekk emnet **IN3240 – Testing av programvare** (finnes også i Mastervariant, IN4240). Bruker bl.a. ny norsk bok "Plettfri kode. Hvordan forhindre feil i programvare?"

Oppsummert

- Å programmere handler om å forstå
- Kode leses mer enn den skrives – god lesbarhet er ekstremt viktig
- Bruk av SOLID-prinsippene kan fremme god lesbarhet og design
  - Single responsibility og Dependency inversion har vi sett på i dag
- Testbarhet oppnås ved
  - å ha klart definerte, enkle, ansvarsområder for klasser
  - å gjøre avhengighetene eksplisitte og utbyttbare
- Ved enhetstesting kan man bruke arrange, act, assert-mønsteret
  - Om man ønsker finnes det mange verktøy for å automatisere testkjøringen
  - Hjemmelekse: vil testen vi skrev gå bra, eller feiler den?



SOLID
Enhets-
testing
Tydelig
kravspec
Godt
design