

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Design og Testbarhet

IN1010 vår 2024 – Eyvind W. Axelsen

```java
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?");
  }
}
```

Mål for denne forelesningen

Se viktigheten av et godt design

Forstå hva et testbart design er

Kunne skrive enkle programmer på en testbar måte

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.

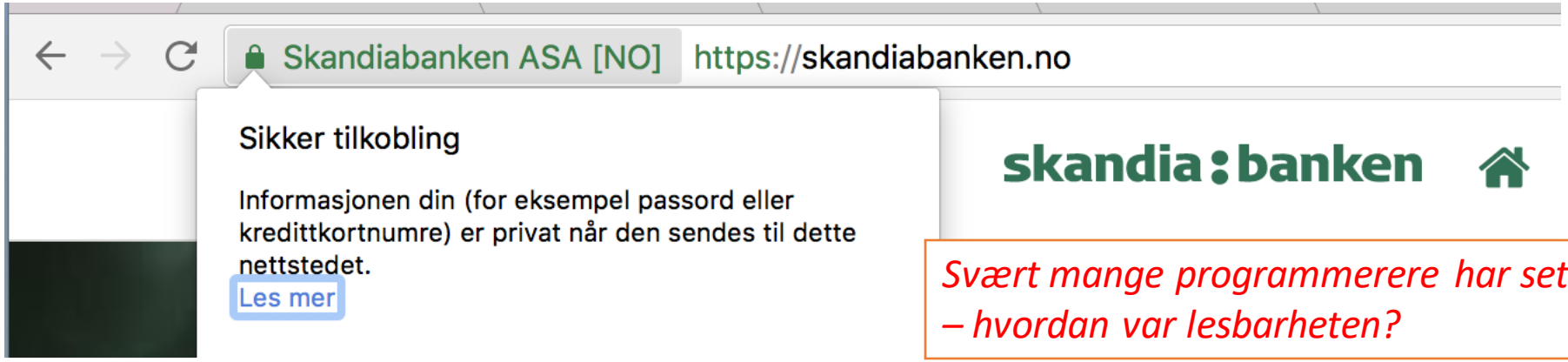
Kjente feil i programmer

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?


-00:00:17

JUNE 15

Tesla Autopilot tops the list of most crashes on driver-assist features in new NHTSA report

Fred Lambert - Jun. 15th 2022 7:53 am PT @FredericLambert





Hvor vanlig er det med feil i programmer?

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



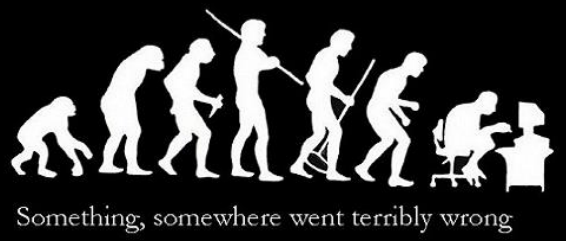
Hvor vanlig er det med feil i programmer?

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]

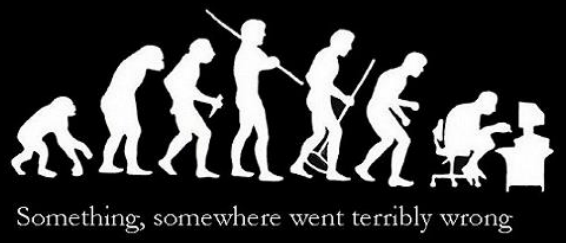


Hva er programmering?

Tenk et par minutter,
diskuter gjerne 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
- 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!");
   }
}
```

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!");
   }
}
```

Sammenblanding av ansatt- og stillingsdata, og regler for stillinger

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!");
    }
}
```

Sammenblanding med organisasjonskart og hierarki

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 %!");

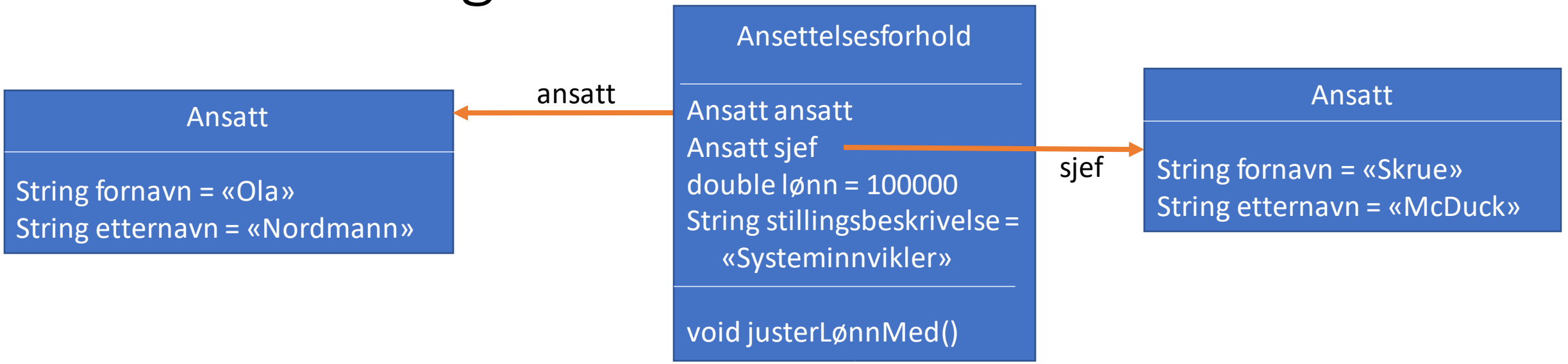
        this.stillingsProsent = stillingsProsent;
    }

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

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

Sammenblanding med forretningslogikk

Et bedre design?



```
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



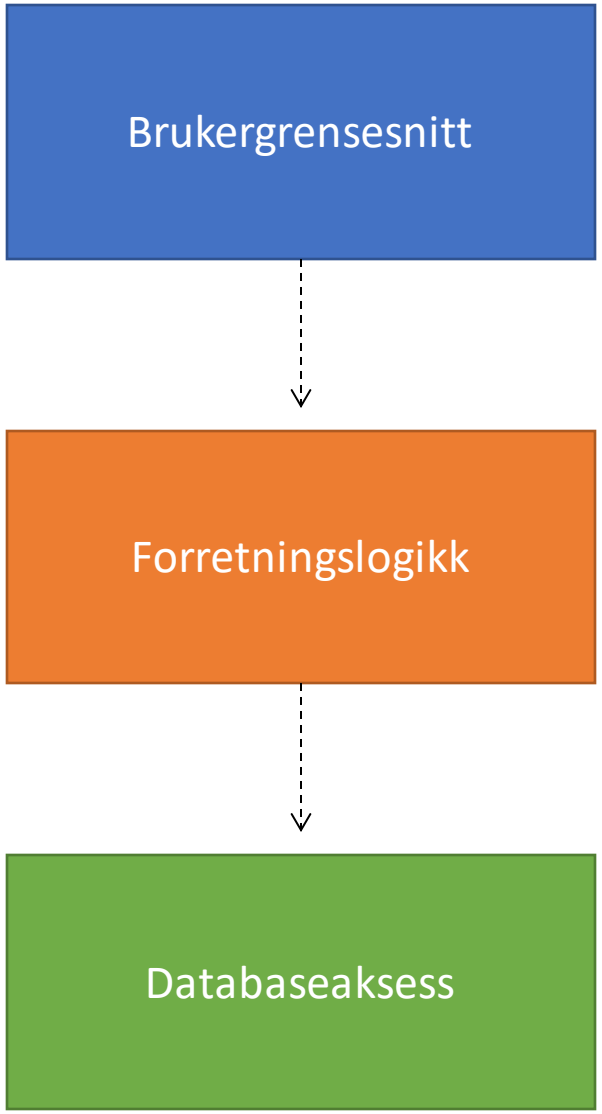
UNIVERSITETET I OSLO

Et interface (grensesnitt) er (2):

Institutt for informatikk

- 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

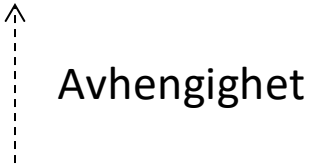


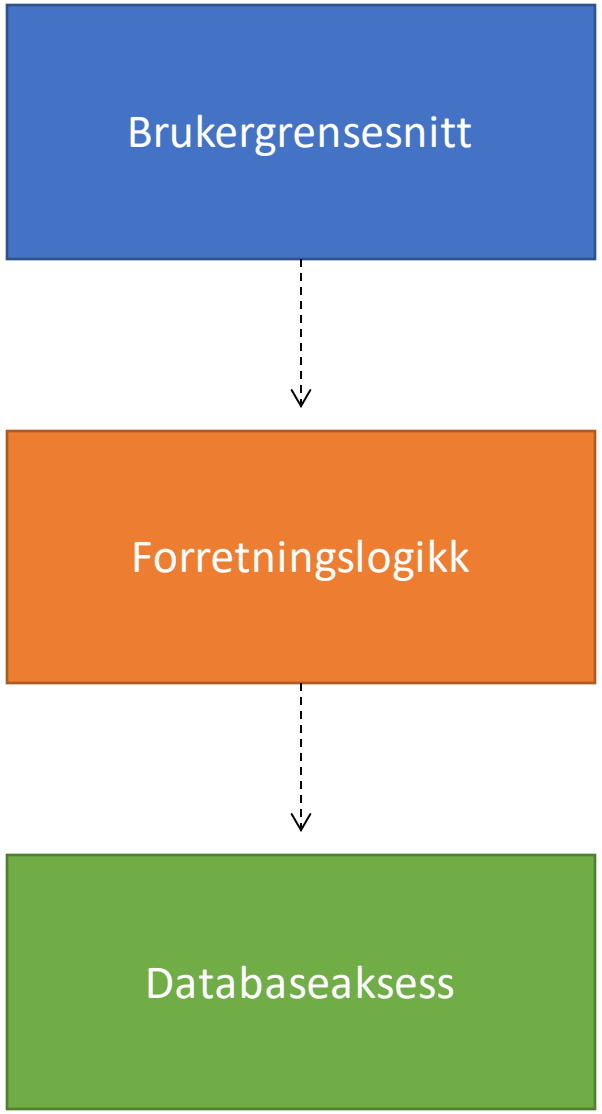
«Tradisjonell» design:

- 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



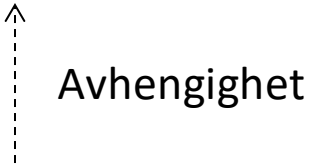


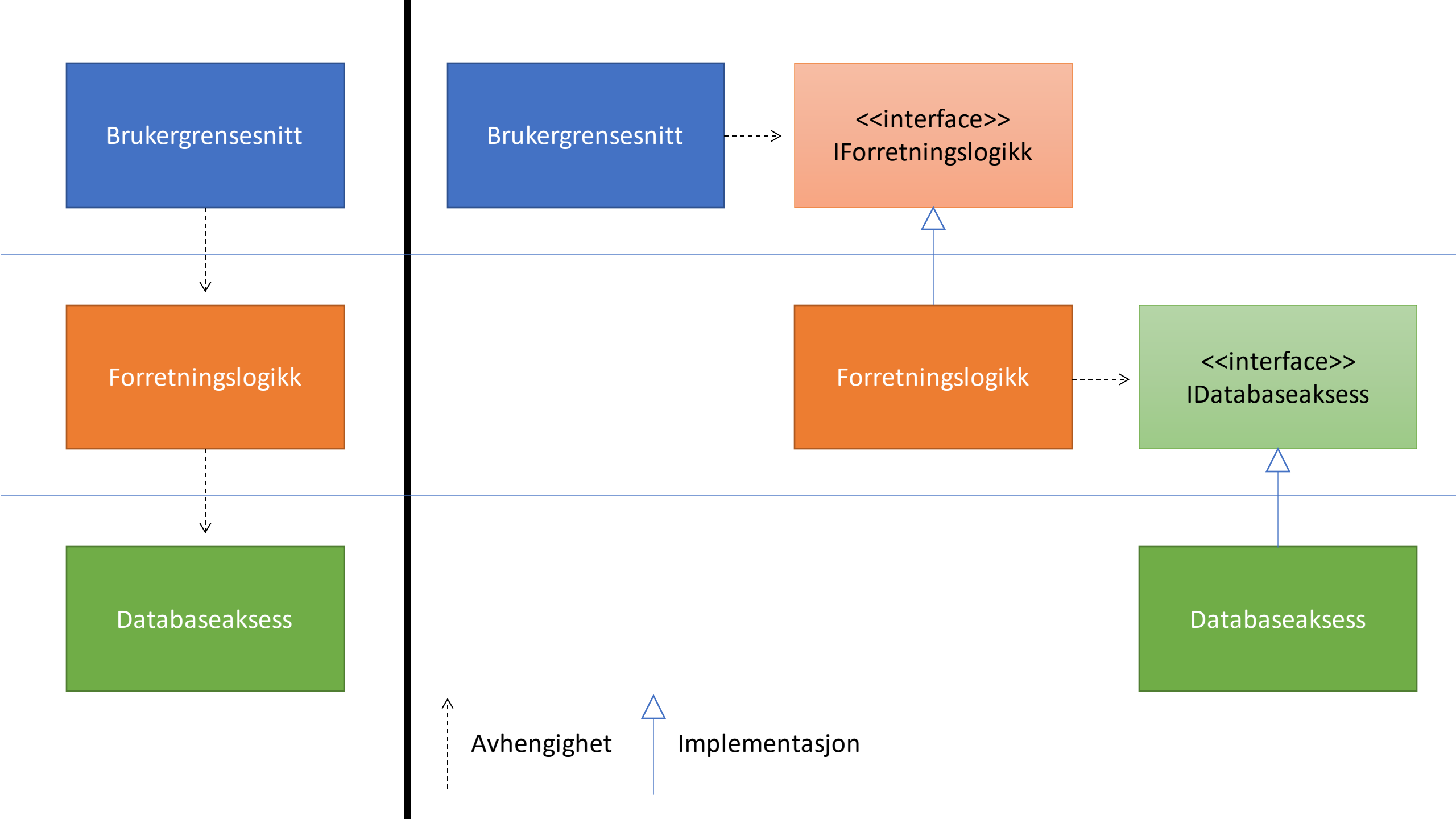
Avhengighet, sier du? Hva betyr det?

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

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

A har en avhengighet til B



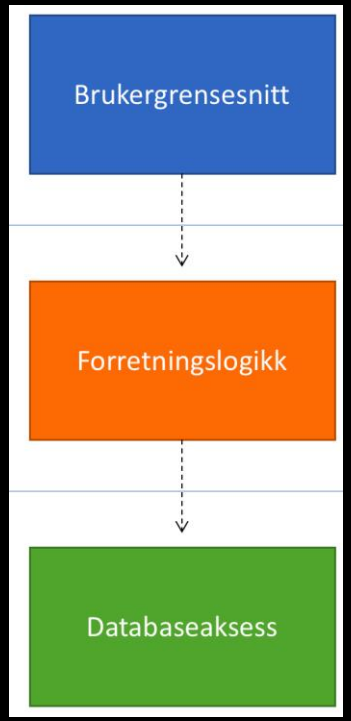

Brukergrensesnitt
Forretningslogikk
Databaseaksess
Brukergrensesnitt
<<interface>>
IForretningslogikk
Forretningslogikk
<<interface>>
IDatabaseaksess
Databaseaksess
Avhengighet
Implementasjon

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 { /* ... */ }

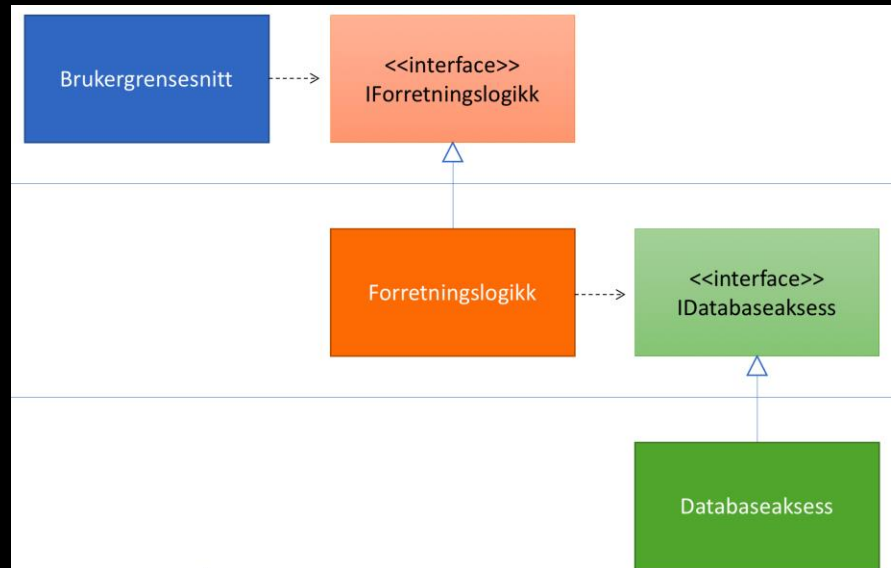
class Forretningslogikk implements IForretningslogikk {
  public Forretningslogikk(IDatabaseaksess databaseaksess) { /* .. */ }
  // ...
}



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

Eksplisitt avhengighet til grensesnitt

Kan *byttes ut* f.eks. under testing



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

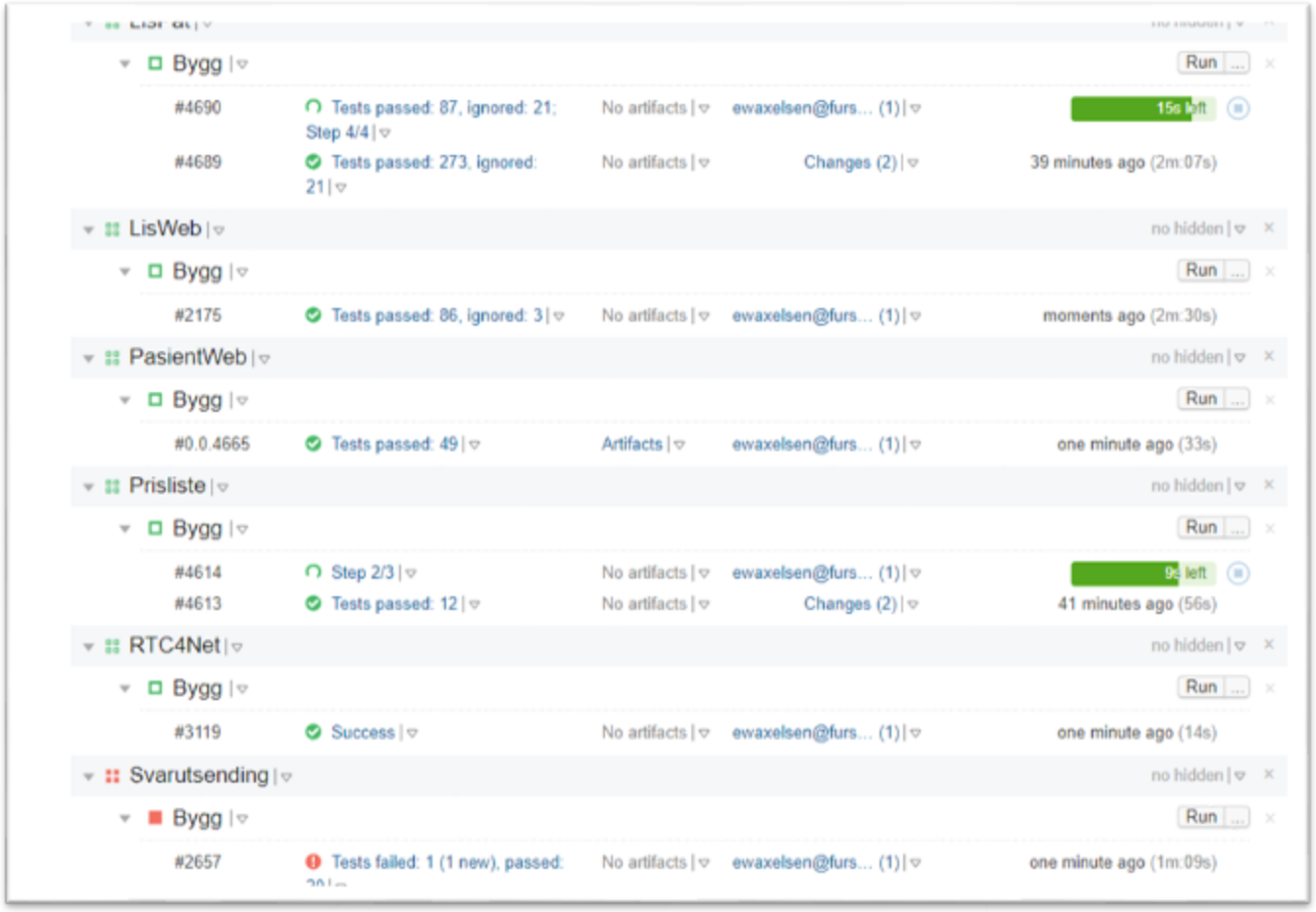
- Et testbart program bør være modularisert på en slik måte at hver komponent lar seg teste for seg
  - Enhetstesting – hver komponent testes i isolasjon
    - Ikke så enkelt som det kanskje høres ut som – dette må kodes inn eksplisitt!
  - Integrasjonstesting – samspillet mellom enkeltkomponenter testes

Har du skrevet en test før?

Hvordan tester du dine programmer?

CI – Continuous Integration («kontinuerlig integrasjon»)

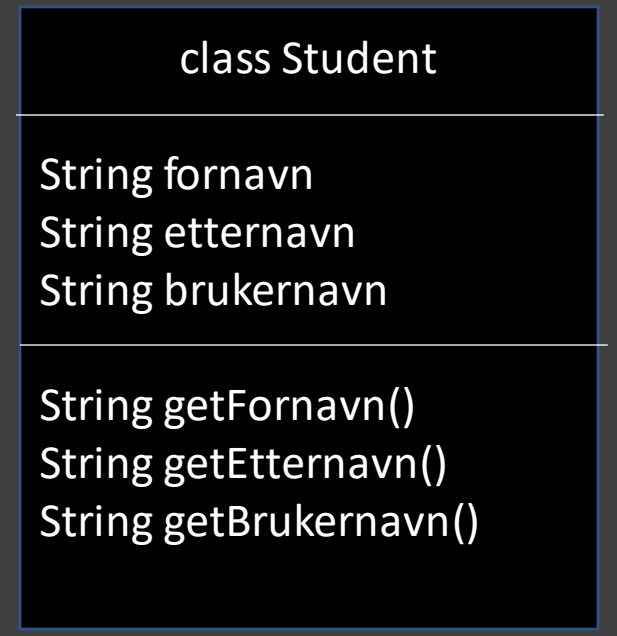
- Alle utviklere på et prosjekt sjekker inn/«commiter» kode kontinuerlig, f.eks. til git
- Hele prosjektet bygges automatisk ved hver innsjekk
- Tester kjøres automatisk ved hvert bygg for å forhindre regresjoner
- Regresjon: kode som tidligere fungerte har blitt ødelagt



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 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();
} } }
```

```
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 implementa
        PrintJobWatcher pjDone = new PrintJobWatch
        // Print it
        job.print(doc, null);
        // Wait for the print job to be done
        pjDone.waitForDone();
} } }
```

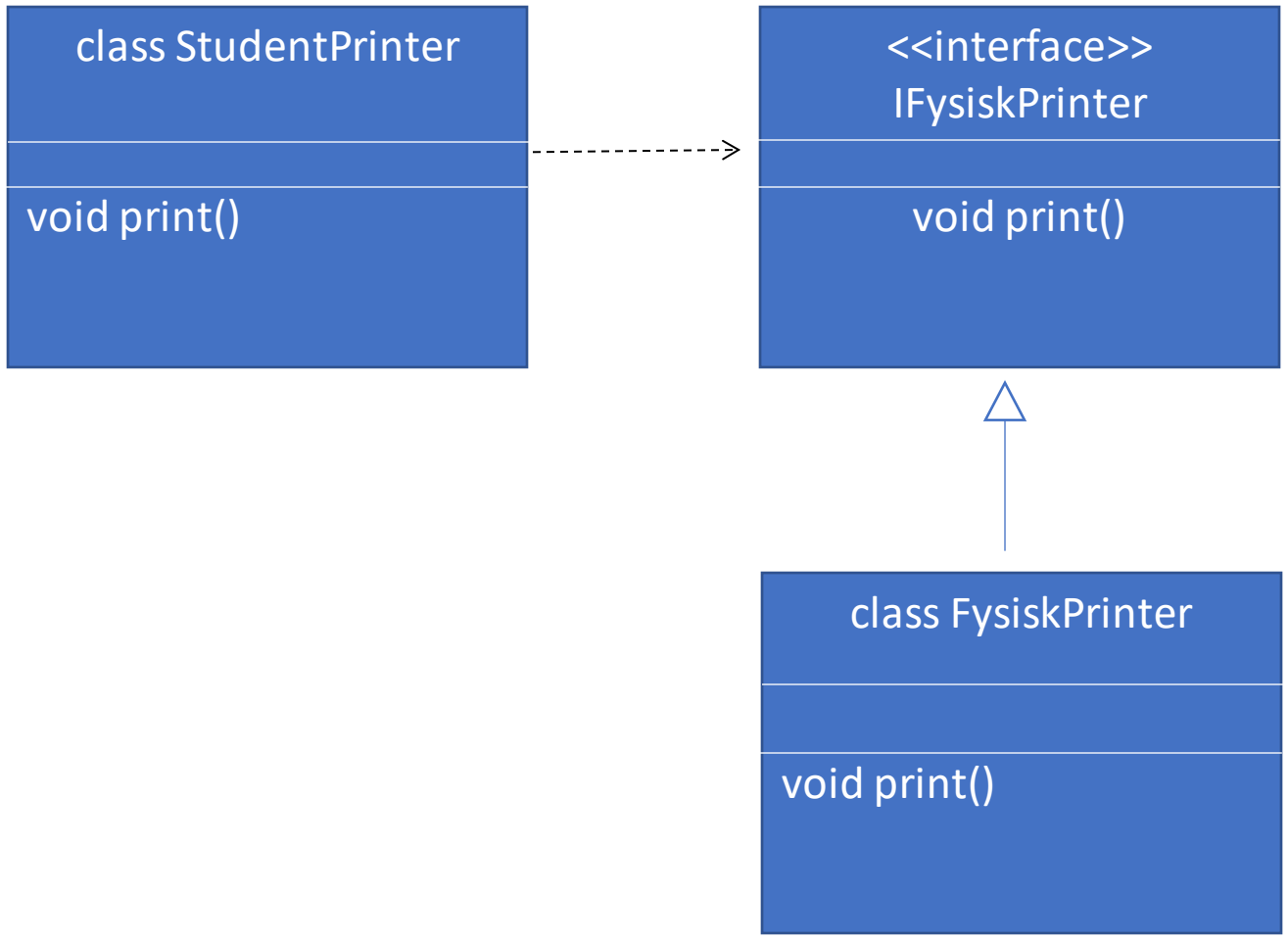
- Skal definere et konsept «Student», med fornavn, etternavn og brukernavn
- Studentene skal skrives ut som en liste på papir, sortert etter brukernavn
- Formen på utskriften skal være slik:
  «Ola Olsen (olao)»
- Programmet skal være testbart
- Er det noe her du kunne tenke deg å endre på?

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()

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

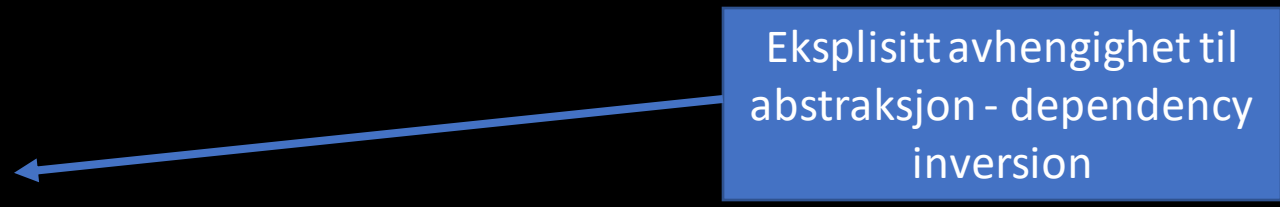
class StudentPrinter {
  IFysiskPrinter fysiskPrinter;

  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 {

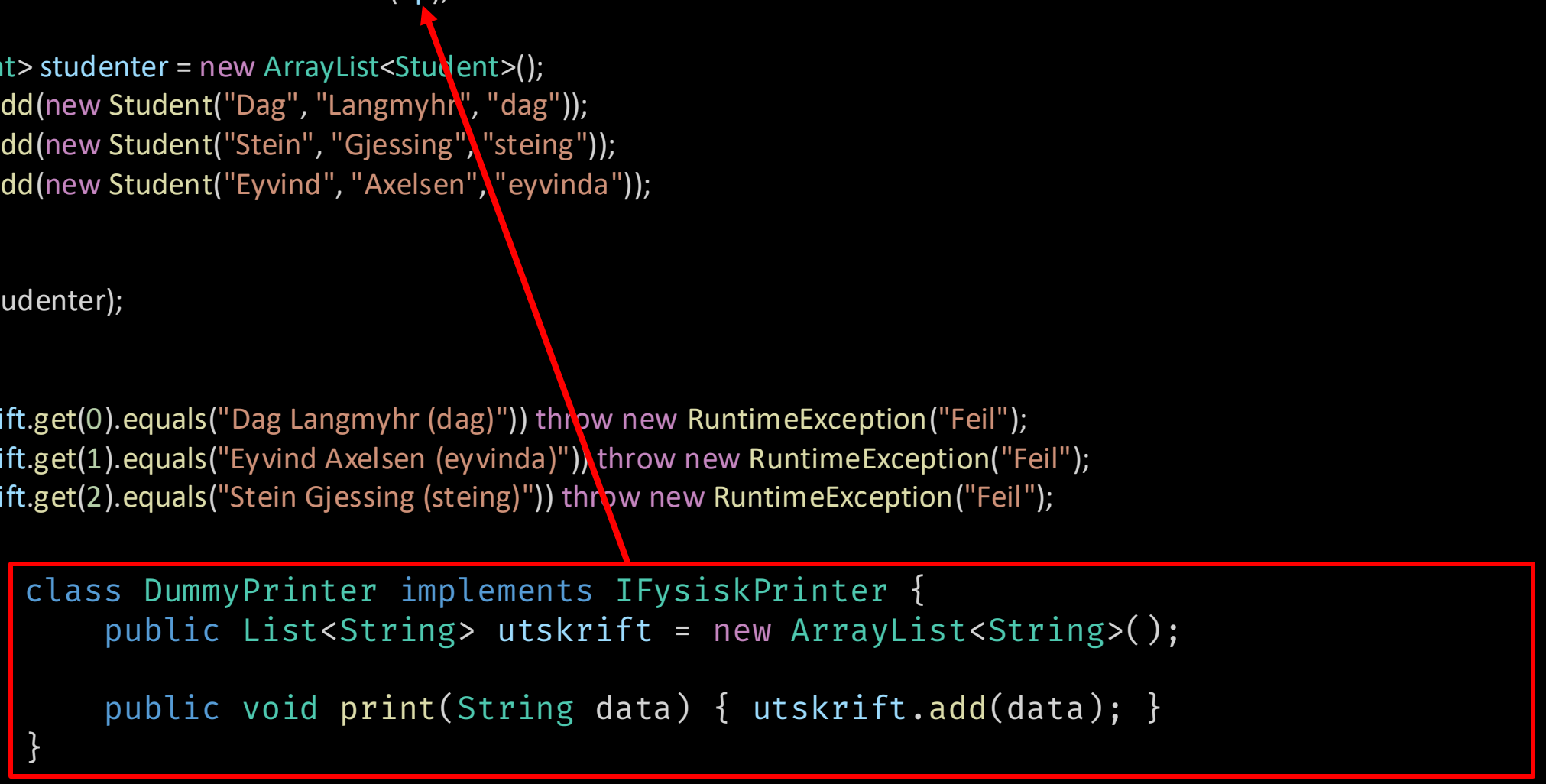
  public void print_KaltMed3Studenter_SkriverUtStudenterIRiktigRekkefølge() throws Exception {

    // 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); }
}
```

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

```
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));
}
}
```

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

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

```
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));
}
}
```

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.

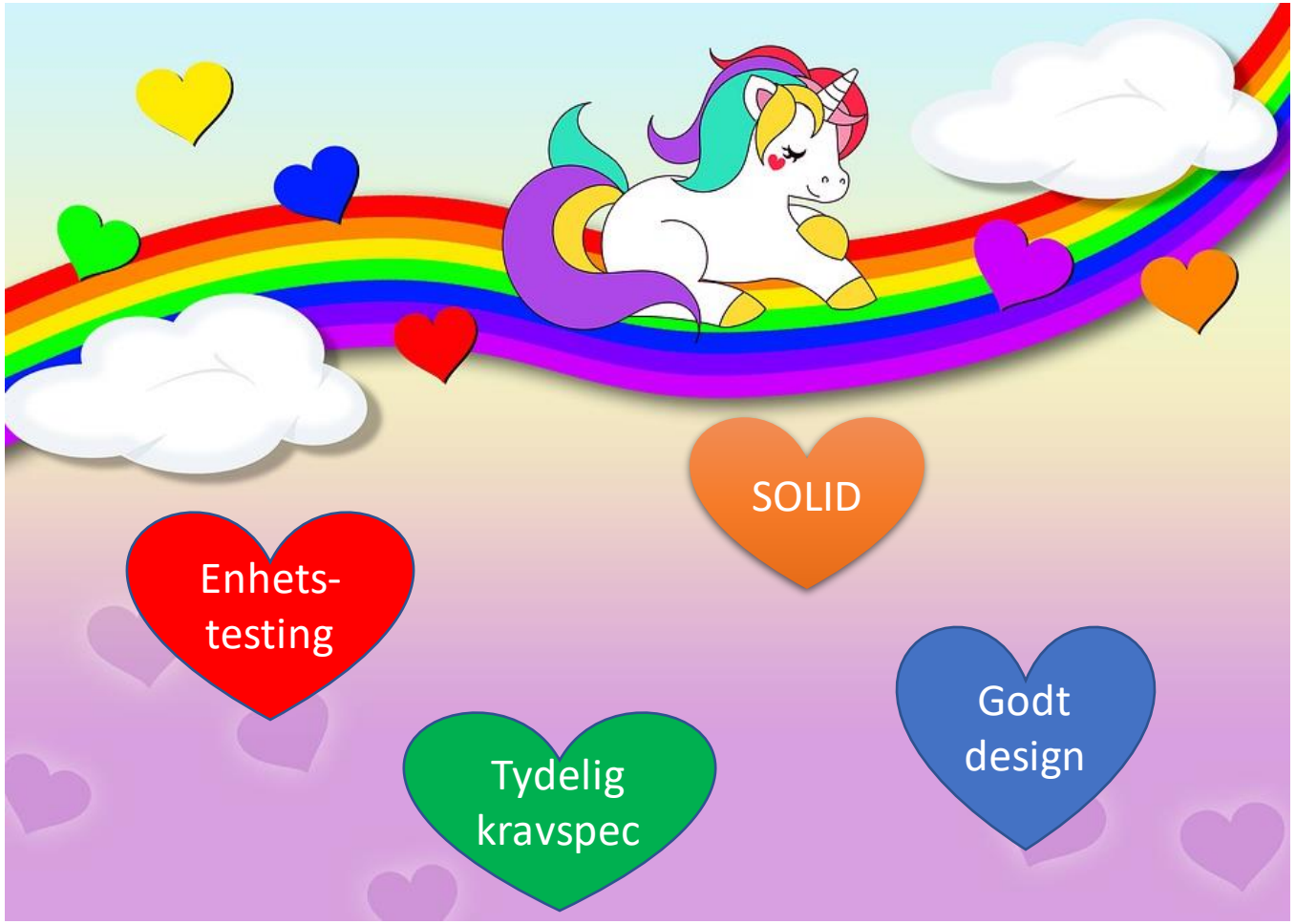
- com.commercehub.perspective.impl.housemodel.test.HousePerspectiveTest FAILED
  - testLoadPerspectivesStore passed (0.281 s)
  - testLoadPerspectivizedHouseWithPerspInsideRoom caused an ERROR (0.031 s)
  - testLoadPerspectivizedHouseWithPerspInsideCircuit passed (0.016 s)
  - testGetPropertyPaths caused 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

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
Godt
design
Tydelig
kravspec