

Identitet och ekvivalens



Värdesemantik eller pekarsemantik?

```
class Pair { Object fst; Object snd; }
```

```
void foo(Pair o) {  
    o.snd = 4711;  
}
```

```
public static void main(String[] args) {  
    Pair p = new Pair();  
    p.fst = 42;  
    foo(p);  
    System.out.println(p.snd);  
}
```

4711

or

null

Identiska par?

```
class Pair {  
    Object fst; Object snd;  
    public Pair(Object a, Object b) { fst = a; snd = b; }  
}  
  
public static void main(String[] args) {  
    Pair p1 = new Pair(42, 4711);  
    Pair p2 = new Pair(42, 4711);  
    System.out.println(p1 == p2);  
}
```

true

or

false

Ekvivalenta par?

```
class Pair {  
    Object fst; Object snd;  
    public Pair(Object a, Object b) { fst = a; snd = b; }  
}  
  
public static void main(String[] args) {  
    Pair p1 = new Pair(42, 4711);  
    Pair p2 = new Pair(42, 4711);  
    System.out.println(p1.equals(p2));  
}
```

true

or

false

Likhet mellan par?

```
Pair p1 = new Pair(...);  
Pair p2 = new Pair(...);
```

```
// Definition 1
```

```
p1 == p2
```

```
// Definition 2
```

```
p1.fst == p2.fst && p1.snd == p2.snd
```

```
// Definition 3
```

```
p1.fst.equals(p2.fst) && p1.snd.equals(p2.snd)
```

```
// Definition 4
```

```
p1.equals(p2)
```

Standarddefinitionen av equals i Object

```
public class Object {  
    public boolean equals(Object o) {  
        return this == o;  
    }  
}
```

Definitionen av ekvivalenta par

```
public class Pair {  
    private Object fst;  
    private Object snd;  
  
    public boolean equals(Pair o) {  
        return this.fst.equals(o.fst) &&  
            this.snd.equals(o.snd);  
    }  
}
```

**Kontravarians för
argument!!!**



Klassiskt fel: överlagring, inte overriding av equals

```
public class Pair {  
    private Object fst;  
    private Object snd;  
  
    public boolean equals(Pair o) {  
        return this.fst.equals(o.fst) &&  
            this.snd.equals(o.snd);  
    }  
}
```



Overloading!

```
Object p1 = new Pair(1, 2);  
Object p2 = new Pair(1, 2);  
System.out.println(p1.equals(p2));
```



Anrop till Object:s equals()!

false

Definitionen av ekvivalenta par

```
public class Pair {  
    private Object fst;  
    private Object snd;  
  
    public boolean equals(Object o) {  
        if (o instanceof Pair) {  
            return this.equals((Pair) o);  
        } else {  
            return super.equals(o);  
        }  
    }  
  
    public boolean equals(Pair o) {  
        return this.fst.equals(o.fst) &&  
            this.snd.equals(o.snd);  
    }  
}
```




Overriding!



Overloading!

Definitionen av ordnade par

```
public class Pair<T> implements Comparable<Pair> {  
    private T fst;  
    private T snd;  
  
    public int compareTo(Pair o) {  
        ...  
    }  
}
```



Vi vet inte nog för att jämföra innehållen!

Definitionen av ordnade par

```
public class Pair<T extends Comparable> implements Comparable<Pair> {  
    private T fst;  
    private T snd;  
  
    public int compareTo(Pair o) {  
        ... this.fst.compareTo(o.fst)  
    }  
}
```



Imprecis typ!

Tillåter oss att jämföra vilka typer av par som helst!

Definitionen av ordnade par

```
public class Pair<T> extends Comparable<T>>
    implements Comparable<Pair<T>>> {
    private T fst;
    private T snd;

    public int compareTo(Pair<T> o) {
        ... this.fst.compareTo(o.fst)
    }
}
```

Sammanfattning

Subklasser ger inte subtypes automatiskt — vi måste aktivt arbeta för detta

Liskovs substitutionsprincip guidar oss, och motiverar varför subtyper är A Good Thing™

Experimentera med detta tills du förstår varför och hur!

Java använder undantagshantering för felhantering

Finally används också för resurshantering orelaterat till undantag

Utdelad kod denna föreläsning: Undantagshantering.java — experimentera!

Identitet och inkapsling är superviktiga begrepp

Klassiskt fel: **public boolean** equals(T o) { ... } där $T \neq \text{Object}$

Java-idiom: $a.\text{equals}(b) \Rightarrow a.\text{hashCode}() == b.\text{hashCode}()$

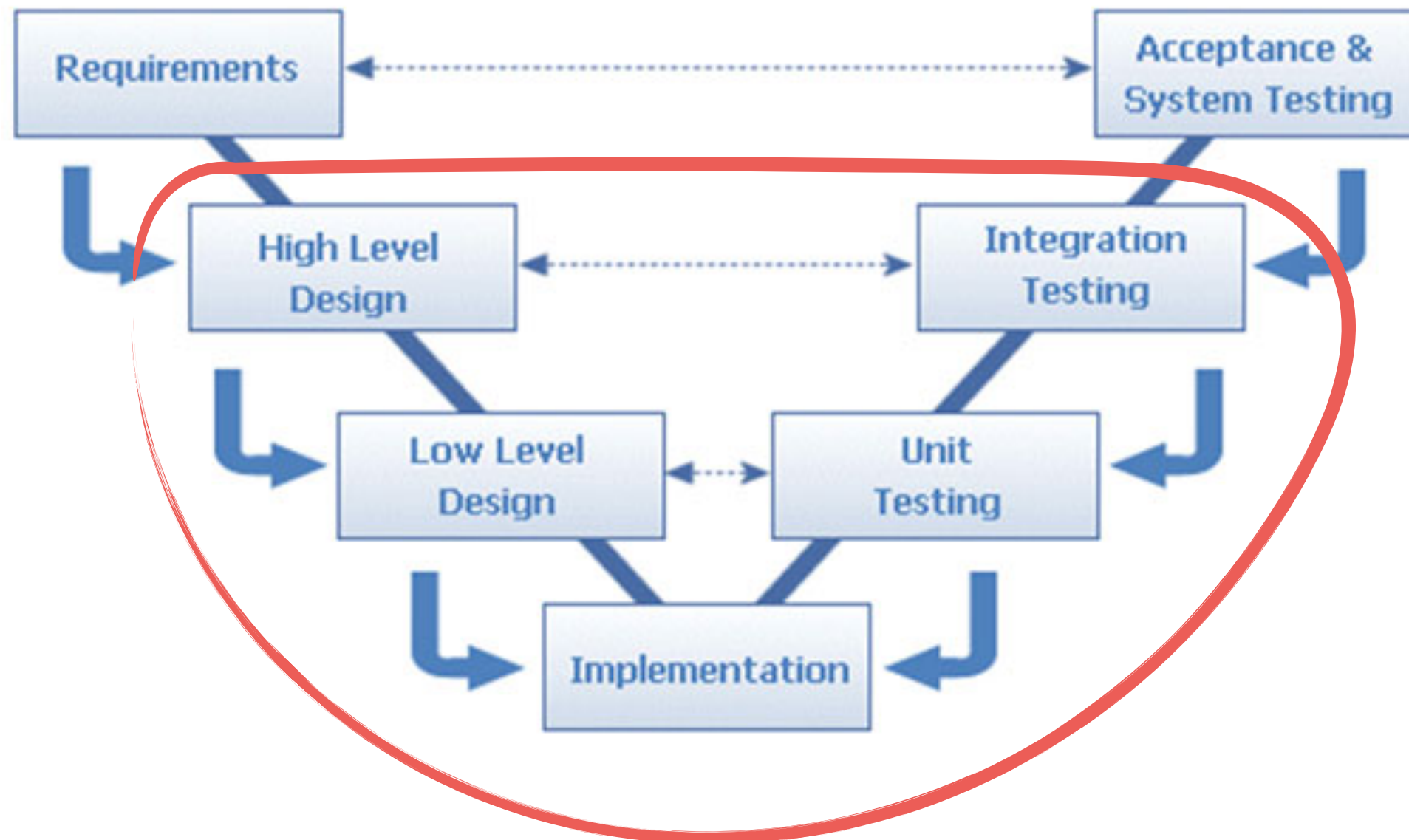
Föreläsning 21

Tobias Wrigstad

Testning



Många olika sorters testning



Tester (recap)

- Tester måste vara **automatiserade** — make test (eller motsvarande) är idealiskt
- Det måste vara **tydligt var fel uppstår**, som provocerats fram av tester
- **Målet med tester är inte att de skall passera**, så optimera för att hitta och laga fel

(White box vs. Black box)-testning

- Black box — testa mjukvaran utifrån kraven

Varje funktion är en svart låda

- White box — testa mjukvaran utifrån källkoden

Här är t.ex. code coverage viktigt (möjligt) — välj input som testar alla vägar

Se föregående föreläsning om coverage/path/logic/state machine

- Denna uppdelning inte längre väldigt relevant, ofta tillämpas en hybrid

Regressionstestning

- Inte ovanligt att defekter återintroduceras under ett programs livstid

Utvecklare som inte är i sync

Dålig integration mellan olika moduler

...

- Varje gång vi gjort en förändring vill vi kontrollera att vi inte har fått en **regression**, dvs. att något som brukade fungera nyss inte längre fungerar

Regressionstestning kan vara t.ex. enhetstester, men kan också vara t.ex. integrationstester och systemtester

Integrationstestning

- Test av att moduler fungerar i samverkan
- Utförs först när vi vet att de enskilda modulerna fungerar enskilt (enhetstester!)
- Exempel på strategier

Big bang: sätt samman allt och kör med trovärdiga data

Bottom-up: testa lägsta byggsten, bygg på tills hela programmet är testat

Riskdriven: börja med det som är mest kritiskt

- Kontinuerlig integration

Se till att alltid ha ett fungerande system (känns detta igen?)

Vanligt t.ex. att ha ett CI-system kopplat till GitHub som körs vid commit



TobiasWrigstad commented 7 days ago



@EliasC This is exactly what the spec says, so *great!*



Support for immutable variables ...

✓ de336ff

Add more commits by pushing to the ~~features/va1~~ branch on **EliasC/encore**.



All checks have passed

1 successful check

[Show all checks](#)



This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request



You can also [open this in GitHub Desktop](#) or view [command line instructions](#).



Write

Preview

AA

B

i

“

<>

🔗

☰

1/2/3

✓

↩

@

★

Leave a comment

Attach files by dragging & dropping or [selecting them](#).



Styling with Markdown is supported

Close pull request

Comment

Labels

ready for review

Milestone

No milestone

Assignees

No one—assign yourself

2 participants



Notifications

🔊 Unsubscribe

You're receiving notifications because you were mentioned

🔒 Lock conversation

Hur man fixar en bugg

- Buggrapport kommer in / vi hittar ett problem i programmet på något sätt

- **Steg 1:**

Skriv ett testfall som provocerar fram buggen (=inte passerar)

- **Steg 2:**

Fixa felet

- **Steg 3:**

Visa att testet passerar

Varför steg 1 och 3?

Testdriven utveckling [Test-Driven Development]

- All utveckling drivs av testerna
- Skriv tester först och skriv endast kod när ett test inte passerar
- Svårt att applicera, speciellt i början
- Tricket: ”start small”

Enkla tester

Utöka koden långsamt och endast som en följd av att testerna kräver det

Refaktorera emellanåt

JUnit

- Fungerar ungefär som CUnit, men med mindre *boilerplate*
- Använder sig av att ett Java-program har förmåga att inspektera sig självt (**reflection**)
- Samma typer av asserts
- Förstår i viss utsträckning identitet/ekvivalens-problematik
- Kräver installation av JUnit på din dator, och att du kompilerar mot JUnit

CLASSPATH

- En kolonseparerad (a:b:etc) lista av sökvägar där javac/java letar efter klasser
- Javas standard-API inkluderas automagiskt
- Övriga klasser måste läggas till till **CLASSPATH** manuellt

Antingen som [växel](#) till kompilatorn

```
java -cp /path/to/junit.jar:/path/to/my/stuff SomeFile.java
```

Eller [miljövariabel](#)

```
$ export CLASSPATH=/path/to/junit.jar:/path/to/my/stuff  
$ javac SomeFile.java
```


Paketstrukturer i Java

- **package** foo; längst upp i filen
- Skapar ett paket som måste ligga i katalogen foo vid kompilering
- Använd -d . till javac för att få .class-filerna i rätt katalog automatiskt
- Importeras i två steg:

import foo.SomeClass; i annan .java-fil

katalogen där katalogen foo ligger måste finnas i **CLASSPATH** vid *kompilering*

- Java tillämpar dynamisk länkning, dvs.

katalogen där katalogen foo ligger måste finnas i **CLASSPATH** vid *körning* också

- För underlättat distribution kan man skapa en **JAR**-fil som är som en .zip-fil

Kom igång med JUnit

Installera

Exempel med JUnit 4.12 (senaste är ≥ 5.0 — principerna är samma)

Skall *förhoppningsvis* finnas installerat på datorerna i datorsalarna

Finns **jar**-filer länkat från kurswebben (vid slides — junit.zip, packa upp!)

```
export JUNIT=junit4-12.jar:hamcrest...
```

Kompilera mot JUnit

```
javac -cp $JUNIT TestSuite.java TestRunner.java
```

Kör JUnit-testerna

```
javac -cp $JUNIT TestRunner TestSuite
```

En Test Suite i JUnit

- Markera varje test med @Test
- Använd assert:s, precis som i CUnit

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestSuite {
    @Test
    public void testAdd() {
        String str = "JUnit is working fine";
        assertEquals("JUnit is working fine", str);
    }
}
```

Asserts i JUnit

Kontrollerar att två värden är lika

void assertEquals(**T** expected, **T** actual)

Kontrollerar att ett uttryck returnerar false

void assertFalse(**boolean** condition)

Kontrollerar att ett uttryck inte returnerar null

void assertNotNull(Object object)

Kontrollerar att ett uttryck returnerar null

void assertNull(Object object)

Kontrollerar att ett uttryck returnerar true

void assertTrue(**boolean** condition)

Signalerar att ett test inte passerar

void fail()

Annoteringar i JUnit

@Test

Anger att metoden i fråga är ett test

@Before / @After

Denna metod skall köras före / efter varje test

@BeforeClass / @AfterClass

Denna statiska metod körs före något / efter alla test i klassen körts

@Ignore

Ignorera denna metod (t.ex. kör inte detta test)

Tänkvärt 1/2

- Att **beakta testbarhet vid utvecklingen** styr koden bort från vissa mönster

T.ex. funktioner som initierar en datastruktur i ett enda svep

Vi vill kunna testa små bitar åt gången (isolerade från resten av systemet)

Nackdel: nu kan vi se objektet i ett felaktigt tillstånd

- Undvik globalt tillstånd (ex. variabler som är **static**) i testkod

Data sparas mellan test

Koden kan ibland bli något mer komplex utan globalt tillstånd

Tänkvärt 2/2

- Arbeta efter principen att varje kodenheter endast skall ansvara för ett åtagande

Konsekvens: fler kodenheter (funktioner, klasser, moduler, etc.)

- Minimera beroenden

Annars blir testerna väldigt komplexa

Tips 1/2

- Det är extremt viktigt att välja bra namn (på allt!)
- Undvik ”fancy koding” (som vanligt) — men var smart
- Få rader ökar läsbarheten, för få rader minskar den
- Många små funktioner som går att kombinera är en design som underlättar återanvändning och underhållsbarhet – och därför också testning!
- Testning går att tänka på som återanvändning
- Använd assert:s, speciellt för sådant som aldrig skall hända (omöjliga situationer)
- Undvik **null**
- Initiera alltid variabler även om du vet att de kommer att tilldelas före de används

Tips 2/2

- Ta bort redundant eller oanvänd kod – mindre = mer läsbart
- Undvik tilldelningar i booleska uttryck och i argumentposition (och argument!)
- Kod med för många hopp (**break**, **return**, etc.) är svår att följa
- Pröva alltid det mest sannolika fallet först (det är det som nästa är intresserad av!)
- Pröva alltid gränsvärden och index mot storlekar
- Titta alltid på returvärdena från funktioner som du utgår från lyckas (t.ex. malloc)
- Dokumentera allt användande av malloc inuti en funktion som leder till data som returneras (annars blir minneshanteringen knepig)
- Lämna alltid tillbaka resurser på ett förtjänstfullt sätt