Programmering Fortsättning

Föreläsning 2

Joachim von Hacht

Innehåll

1 Testning

Testdriven utveckling[1] (en strategi)

> "Program testing can be used to show the presence of bugs, but never to show their absence!" // Edsger Dijkstra

Två exempel (finns en mängd olika tester (varianter) förutom dessa).

[1] Egentligen skall testerna skrivas innan koden men så gör inte vi.

- Enhetstestning (unittesting). Test de minsta delarna i ett program (klasser, paket, moduler). I vårt fall klasser.

- Regressionstestning (regression testing). Alla tidigare tester sparas. Vi ändringar (refactoring) måste ev. ny test + alla tidigare bli godkända.

1.1 Enhets- och regressions-testning

- Det är oerhört mycket rationellare att ha testerna nedskrivna och exekverbara än att försöka komma ihåg, skriva kommentarer eller testa för hand. Att testa går mycket smidigt på detta sätt.

- Testerna är tänkta att vara automatiska, ingen mänsklig interaktion skall förekomma (testen lyckas eller misslyckas).

- Testerna sparas. Vi modifiering (refactoring) av en klass skall den gamla testen (troligen) fortfarande lyckas (detta blir automatiskt regressionstestning). Så att inte ändringar introducerar nya fel i tidigare testad kod.

- Man kan se testerna som en en verifikation av en specifikation, se senare semiformella resonemang.

1.1.1 Teknik

Vi använder ramverket JUnit 4.

Vi skiljer alltid på testkod och applikationskod. Dett görs genom att använda två olika "source folders" i Eclipse (src för applikationskod och test för testkod). I testfolderna skapar vi samma paketstruktur som i src. Innebär att vi kommer att kunna testa paket-privata klasser.

Normalt testas publika metoder (ej set/get). Om man har behov av att testa privata metoder använder man specialla hjälpklasser (inte ändra klassen vi testar!). Att man behöver testa privata metoder kan tyda på dålig design, finns det en annan klass inuti den vi testar?

Om metoderna under test returnerar något kan vi direkt m.h.a. JUnit's Assert-klass avgöra om testen lyckades. Om metoden är void måste vi ha tillgång till någon metod som kan avläsa eventuell tillståndsförändring. Ofta får man lägga till en get-method som endast används vid testning (ingår inte i något gränssnitt, se vidare gränssnitt).

- Typiska att testa är extremfall; Tom lista, tomma strängen, full lista, ...första, sista, ...

- Se upp med bra namn på testmetoder (använd lååååånga beskrivande). Exempel: testIfLastNodeContainsReferenceToHead(...)

1.2 Code coverage

Om testerna skall ha något värde måste vi veta att en stor del av koden (all) har exekverates och därmed testats, t.ex. olika grenar i if-satser, körs loopen?, o.s.v.) s.k. code coverage[2]. Det gäller att utforma testerna så att detta uppfylls!

2 Språk

2.1 Repetition: Uttryck och sats

- Uttryck (expression) representerar ett värde t.ex. Math.sin(alpha) + Math.PI/2

- Sats (statement) Ett imperativ (uppmaning). Gör det! En instruktion! In-

[2]Triviala metoder get/set testas aldrig.

get värde produceras (tomma satsen = ; (bara semikolon)).

- Exempel:

```
// Wrong, if-statement has no value
int value = if( x > 0 ) { y } else { -y };

// If-expression has a value
// (it's an expression)
int value = (x > 0) ? y : -y;

// Assignment is an expression
// (what's the value of an assignment?)
int x, y, z;
x = y = z = 1;
```

Observation Alla uttryck i Java har en typ. Satser har ingen typ.

2.2 Repetition: Deklaration och synlighetsområde

(JLS 6.1) "A declaration introduces an entity into a program and includes an identifier (§3.8) that can be used in a name to refer to this entity."

- I Big Java används begreppet definition?? Kan inte hitta detta i JLS. Korrekt term är deklaration.

En deklaration har ett visst synlighetsområde (scope), det område i koden där man kan använda identifieraren (namnet) (JLS 6.3). En deklaration kan skugga (shadow) en annan t.ex[3].

```
class Test {
  static int x = 1; // x here...
  public static void main(
                String[] args){
    int x = 0;     //..and here
    // Which x?
    System.out.print("x=" + x)
    System.out.println("Test.x="
                    + Test.x);
  }
}
```

- I Java gäller ungefär block-scope (mellan { och }) för deklarationer, finns många fall (JLS 6.3);
  - "The scope of a declaration of a member m declared in or inherited by a class type C is the entire body of C, including any nested type declarations."
  - "The scope of a parameter of a method (§8.4.1) or constructor (§8.8.1) is the entire body of the method or constructor."
  - "The scope of a local variable declaration in a block (§14.4.2) is the rest of the block in which the declaration appears..."

2.3 Typsäkra enums

(JLS 8.9) Enums (enumerations, enumerated types) används då man behöver ett litet antal värden av någon typ t.ex. veckodagar, färger, m.m. Tidigare (eller i andra språk) använde man t.ex. heltal till detta. T.ex.

```
public final static int RED = 0;
public final static int GREEN = 1;
public final static int BLUE = 2;

int myColor = RED;
```

Här kan kompilatorn inte kontrollera om vi råkar tilldela myColor värdet 99 (en ogiltig färg!). För att lösa detta har man infört typsäkra enums. En enum är ett specialfall av en vanlig klass. Skrivs (simpelt ex);

[3] Se även JLS index shadowing.

```
public enum Day {
   MONDAY, THUESDAY, WEDNESDAY,...
}
```

- En enum-klass innebär att det automatiskt skapas exakt så många instanser som värden vi räknar upp (alltså 7 st för dagar). Varje instans är ett värde.
- Kan inte göra new Day() (konstruktor privat).
- Värdena ovan: Day.MONDAY, Day.THUESDAY,...o.s.v. i kod (inte typkomatibla med något annat...);

```
Day someDay = Day.MONDAY;
```

- Man kan loopa igenom alla värden enl. följande (se vidare traversering av behållare...);

```
// Static method call
for (Day d : Day.values()) {
  // Writes MONDAY,..
  System.out.println(d);
}
```

- Vanligt med omvandling mellan strängar och enums. Görs enligt;

```
// String must 'match' enum
// value exactly
Day d = Day.valueof("MONDAY");

String day = Day.MONDAY.toString();
```

- En hel del specialfall: Enums får inte vara abstrakta och är implicit final, se arv senare...(se JLS)

KOD lab 1, LanguageName

2.4 Metodsignatur

(JLS 8.4.2) Metodsignatur = signatur (signature) = metodnamn + parameterlista (antal parametrar, ordningen på dessa och typ för respektive). *Inget annat ingår* (speciellt inte returtypen).

- Två metoder med samma signatur inom samma synlighetsområde (t.ex. i samma klass) är aldrig tillåtet (compile time error). Kan aldrig ha t.ex

```
// Same signature, bad
class A {
  public int doIt( String s ){...}
  public String doIt( String s ){...}
}
```

3 Gränssnitt

3.1 Klass och typ

Det är operationerna (metodsignaturerna) som är det karakteristiska för en typ. Däremot är inte implementeringen. Man kan tänka sig två olika klasser med exakt samma operationer men med olika implementationer. Detta leder till att man istället för att introducera en typ m.h.a. en klass (en implementation) introducerar den m.h.a. en specification (ett slags kontrakt). Helt enkelt; Vad kan man göra med objekt av denna typ? Detta kallas för ett gränssnitt (en klass introducerar en typ men en typ behöver inte introduceras med en klass, ...kan använda ett gränsnitt i stället).

3.2 Gränssnitt

Ett gränssnitt (interface); introducerar en referenstyp (kan alltså deklarera variabler, arrayer av typen).

(JLS 9) Genom att använda varibler med gränssnitttyper kan vi minska beroendena

mellan klasser/moduler. Vi kommer inte att känna till de konkreta implementationerna (dessa kan t.ex. bytas ut utan att resten av programmet behöver ändras).

- Ett gränssnitt är konkret; en "klass" med enbart abstrakta metoder d.v.s de saknar metodkropp (bara signatur och returtyp med ';' sist).
  - Anges speciellt med reserverade ordet interface istf class.
- En gränssnittsdeklaration;

```
public interface INet {
  public abstract
       void send( Message m );
  :
}
```

Scenario*:* Antag att vi deklarerat ett gränssnitt, IA [4] med metoderna public abstract void a() och public abstract void b(). Vi deklarerar en variabel av typen;

```
IA something = ...
```

Vi vet då att something är något som kan hantera (kompilatorn godkänner);

```
something.a();
something.b():
```

Men var är koden som exekveras, gränssnittet har ingen kod (alla metoder abstrakta)? D.v.s denna del av applikationen vet mindre om resten av applikationen.

Observation Programmering mot gränssnitt är en central teknik för att minska beroenden och öka modifierbarheten.

[4] Vi inleder namn på gränssnitt med ett stort I (gammal IBM standard).

3.2.1 Teknik

- En gränssnitt per fil, gränssnittsnamn och filnamn måste stämma (som klasser).
- Alla metoder implicit public abstract (vi anger public ändå). Metoderna är "instance methodes"[5] d.v.s. i samband med anropet måste det finnas ett objekt.
- Kan (vanligen) inte instansiera gränssnitt[6];

```
// Error
IA a = new IA();
```

- Ett gränssnitt är inte en ett objekt[7].
- Kan innehålla annat än metoder men vi anger endast metoder (best practices).

3.3 Implementering av gränssnitt

För att specificera vilken implementation som uppfyller kontraktet låter man någon klass *implementera* gränssnittet skrivs;

```
// Class (must) fulfill contract for IA
// Compiler checks
public class MyClass implements IA {
  :
}
```

BILD (UML)

Alla objekt av typ MyClass kan nu (skall kunna) hantera det som specifierats i gränssnittet.

[5] Alltså inte static, se static.
[6] Finns undantag, se inre anonyma klasser
[7] En klass- eller en arrayinstans är ett objekt.

3.3.1 Teknik

- implements innebär att vi *overrides (överrider, omdeklarerar, överskuggar? Finns inget bra...!)* de abstrakta metoderna i gränssnittet genom att i den implementerande klassen ange metoder med *samma signatur och returtyp men dessutom lägger till en metodkropp*[8].
  - Kopiera från interface till klass, lägg till metodkropp.
  - Vi anger alltid (annotationen) @Override vid implementationen av metoderna från gränssnittet (kompilator kontrollerar, minskar risk för fel)[9].
- En klass som implementerar ett gränssnitt kan alltid typomvandlas från klasstypen till gränssnittstypen (säkert, sker automatiskt) de är typkompatible.

```
//A implements IA
IA ia = A();   //Ok
```

- Tvärtom bryter mot typsäkerheten;

```
IA ia = ...
A a = (A) ia; //Dangerous!
```

  iface.casting

  Skall inte behöva förekomma, troligtvis designfel.

- Måste ange public för metoder i implementerande klass annars minskas synligheten, ej tillåtet (default för klasser är paketsynlighet).
- Olika klasser kan implementera samma gränssnitt, de blir då utbytbara (även under körning). Se vidare polymorfism nedan.
- En klass kan implementera flera olika gränssnitt. Objekt kan då uppträde i flera olika "roller" (se upp, "Single responsibility principle" ev bättre med flera klasser/objekt).
- Undantag: "A method declaration must not have a throws clause that conflicts (§8.4.6) with that of any method that it overrides or hides; otherwise, a compile-time error occurs."//JLS 8.4.8.3. Återkommer vid undantagshantering.
- Viktigt att inse att inget "händer" med parametern när vi deklarera den som ett gränssnitt (objektet förändras inte på något sätt av deklarationen, om så var fallet skulle inte polymorfism fungera, se nedan).

4 Polymorfism

(Polymorphism, mångformighet) Svårfångat begrepp[10]. Här handlar det om att avgöra vilken kod som kommer att exekveras (vilken metod). Se även arv...

4.1 Override

- Vid override innebär polymorfism att metod väljs utifrån signatur, returvärde[11] och **objektets** typ (runtime type, dynamic type) **inte** refer-

[8] Saknas någon metod eller metodkropp blir klassen abstrakt, finns dessutom villkor för returtyp återkommer...

[9] Ännu viktigare vid arv, återkommer...

[10] Verkar inte finnas någon riktig enighet om begreppet.

[11] Eller subtyp, se kovarians.

ensens typ (statiska typen)[12]. Exempel; Variabeln (referensen) är deklarerad som ett gränssnitt men koden väljs från något objekt av typen (klassen) som implementerar gränssnittet.

- Exekveringsmiljön avgör under körning vilken kod (metodkropp) som skall exekveras[13]. Kallas *sen bindning* (late, virtual, dynamic binding)[14].

- Kan *inte* bestämma metod vid kompilering (vet inte vilket objektet är)[15].

```
public void foo(ITest t) {
   t.a();  //No code in ITest!
}
```

- Alla instansmetoder i Java är polymorfa d.v.s detta är det normala vid exekvering av publika metoder (om inte metoden är final, se arv)[16]. Exempel;

```
String t = "ab";
Object o = t + "c";
String r = "abc";
//equals polymorph
//(True,uses runtime type)
boolean b = o.equals(r);
```

- Underförstått att metoderna på något sätt handlar om samma sak.

[12]Kallas "single dispatch". Finns även double dispatch där man även tar hänsyn till parametrarnas runtime-typer.

[13]Tar tid och minne, språk med polymorfism är alltid långsammare än språk utan.

[14]Kallas även dispatching, kräver att alla objekt har en referens till en funktionstabell (dispatchtable).

[15]Noggrannare: Vid kompileringen skapas en beskrivning (descriptor) av vad som skall köras vid exekvering. Under exekvering används beskrivningen för att hitta metoden (koden).

[16]private metoder är implicit final.

KOD iface.multipleimpl, iface.multipleiface,

4.2 Overload

- (JLS 8.4.9) Vid overload innebär polymorfism att metod väljs utifrån aktuella parametrar (antal, ordning och typ). Metoderna i typen (klassen) har **samma namn** men olika signatur, d.v.s. olika parameterlistor.

- Vid kompilering fixerar signaturen. *Tidig bindning* (early, static binding). Eventuellt typomvandling för att hitta denna. Runtime söker man efter en metod med matchande signatur.

4.2.1 Overloaded operators

Innebär att operatorn beter sig på olika sätt beroende på operandernas typ (helt olika funktionalitet). I Java finns bara +-operatorn[17]. Addition för numeriska typer eller konkatenering (sammanslagning) av sträng och annan typ (en operand måste vara av typen string). Om en klass implementera toString() metoden anropas denna annars skrivs klassnamnet och ett s.k. hashvärde ut (unikt id för objektet).

KOD overload.*

4.3 Override kontra overload

- Override; metoder i olika interface/klasser med samma signatur och returtyp (se upp med undantag). Klasserna har ett implementsförhållande (generellt; ett arvsförhållande återkommer...).

[17]I t.ex. C++ kan man själv skapa overlagrade operatorer.

- Overload; metoder i samma klass[18] med olika signaturer (returtyp och undantag räknas alltså ej).

4.4 Design

Overriding skall användas då;

- Vi vill ha ett beteende baserat på typ av objekt. T.ex. olika saker händer för olika objekt som implementera samma interface (kan ersätta if-satser!).

Overloading skall användas då;

- Samma generella funktionalitet gäller för samtliga metoder (samma sak skall göras). Exempel; max av två saker, t.ex. int, long, ...(elegantare att kunna ha samma namn på samma funktionalitet)
- Samma metod kan ta olika antal parametrar (den med flest parametrar är basmetoden övriga har förbestämda (default) värden för vissa parametrar.
  - Overloading av konstruktorer mycket vanligt (kan anropa en konstruktor från en annan. Anropas this(...))

5 Lite om Java miljön

(JVMS 4.1) För att t.ex. förstå overriding m.m. kan det underlätta att känna till lite "runt om" språket t.ex. JVM:en. Så här ser strukturen för en class-fil ut (den kompilerade Java koden);

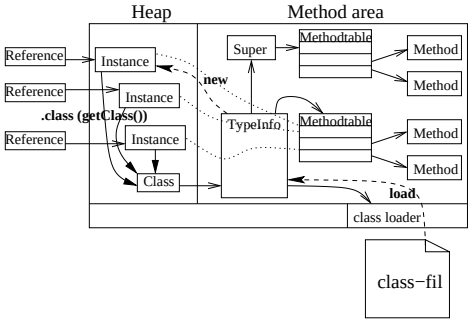
```
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool[
           constant_pool_count-1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[
           attributes_count];
}
```

Exekvering se Figur 1(innan vi startar har vi kompilerat koden till class-filer);

1. Då vi startar JVM:en (java) ger vi en klass som argument (java edu.gu...Main). Klassen måste innehålla en main-metod.
2. JVM söker rätt på class-filen (m.h.a. classpath), läser in filen (bytes) till den s.k. method-area:n (sköts av en speciell del av JVM:en, class loader:n)[19]. En struktur i metodarean håller all information från class-filen. I sammband med laddningen skapas en instans av java.lang.Class, ett objekt vi kan använda för att läsa av information om strukturen i metodarean. Varje gång JVM:en under exekvering stöter på en klass som den inte kan hitta i method arean upprepas detta (hittas inte class-filen får vi ClassNotFoundException).
3. Vid metodanrop kan JVM:en utifrån anropet och typinformationen leta rätt på den kod som skall köras.

[18] Skall inte finnas i olika interface/klasser, fel design, se arv...

[19] Att det är en Java-klassfile kan man se på de inledande byten "magic"(i hex) CAFEBABE.



Figur 1: JVM:en

5.1 Argument till Program och JVM

Man kan skicka argument till programmet då det startar sker enligt;

```
// Starting JVM running class Main
// (with main-metod). Arg is 'hello'
java edu.gu....Main hello

// In program
public static void main(String[] args){
  // Print 'hello'
  System.out.prinltn(args[0]);
```

Argument till JVMen skrivs t.ex. (-Dsystemegenskap=värde, se upp med mellanslag);

```
java -Djava.security.policy=
security.policy
-Djava.rmi.server.hostname=
127.0.0.1  edu.gu.....Main
```

6 Runtime type information (RTTI)

Alltså sådan typinformation som finns kvar efter kompileringen (under körning).

6.1 instanceof

Den inbyggda operatorrn "instanceof" kan användas för att undersöka typkompabilitet. Ger ett booleskt värde.

> "At run time, the result of the instanceof operator is true if the value of the RelationalExpression is not null and the reference could be cast (§15.16) to the ReferenceType without raising a ClassCastException. Otherwise the result is false.."//JLS 15.20.2

Exampel;

```
if( o instanceof A ){
    A a = (A) o;// Safe casting
}
```

- Undvik instanceof, se nedan.

KOD _instanceof

6.2 .class och getClass()

Ger tillgång till objektets exakta runtime typ i form av Class-objekt. Alltså; Ger ett object. Class är en generisk typ, man anger vilken klass Class-objektet representerar. Används;

```
// Using an object of type MyClass
Class<MyClass> c = obj.getClass();

// Using a named type
Class<MyClass> c = MyClass.class;
```

KOD classloading, _class

6.3 Polymorfism och RTTI

Man skall undvika RTTI, ersätt med polymorfism. Exempel;

```
// Typecompability checked at
// runtime
// Tedious, hard to change.
// Have a tendecy to crop up
// all over the application
// (duplicate code)
if( o instanceof A ){
   ((A)o).doIt();
}else if( o instanceof B ){
   ((B)o).doOther();
}else...
```

Det ovan kallas bl.a. switch-code smell. Vi riskerar att behöva ändra i (potetiellt många) switch då nya klasser tillkommer (eller gamla försvinner). Bättre, låt B ha en metod doIt som gör det doOther skulle ha gjort[20];

```
// Much simpler
o.doIt()
```

7 Designmönster

Ett designmönster (Design Pattern) är en återanvändbar lösning på ett återkommande design problem (ej språkberoende).

> "Design Patterns: Elements of Reusable Object-Oriented Software (ISBN 0-201-63361-2) is a software engineering book describing recurring solutions to common problems in software design. The book's authors are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides with a foreword by Grady Booch. They are often referred to as the GoF, or Gang of Four."//Wikipedia

Mönstren kategoriseras efter: Creational (skapa objekt), Structural (skapa en löst kopplad, modifierbar structur), m.fl

7.1 Subsystem

Ett subsystem är en en modul (paket) som utåt har ett funktionellet sammanhållet gränssnitt t.ex. en nätverksmodul, en databasmodul,... (en funktionell enhet).

7.2 Facade mönstret

Ett designmönster (structural). Låt ett subsystem gentemot resten av applikationen representeras som en gränssnittstyp (använd variabler av denna typ). En enda *synliga* klass i modulen implementerar gränssnittet, resterande klasser döljs.

- Gränssnittet till subsystemet ligger aldrig i samma paket som klassen som implementerar gränssnittet (gränssnitt används mellan moduler. Intern i modulen räknar man med att allt är mer eller minder beroende).

Observation Genom att använda Facade minskar vi beroenden. Facade gör att övriga delar av applikationen bara känner till gränssnittet implementationen kan bytas ut/ändras.

BILD UML

[20] Effective C++, by Scott Meyers :
"Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself.

7.3 Factory-method

(creational) För att få tag i objekt av "den enda synliga" -klassen ovan kan man använda ett designmönster som kallas factory method[21]. Lägg märke till paket och att metoden returnerar gränssnittstypen.

```
package mysubsys;
public interface IMy {
  ...
}

package mysubsys.impl
public class MyImpl imlements IMy{
   // Factory method (a class method)
   public static IMy getInstance(){
      return new MyImpl();
   }
   // Private constructor here
}
```

KOD facade, facade.impl

- Factory method innebär att vi slipper new i koden, vi blir inte beroende av den exakta implementation (getInstance kan t.ex returnera en subklass, återkommer vid arv).

Andra fördelar;

- Antalet objekt kan kontrolleras.
- Effektivitet, kanske inte nödvändigt att skapa nytt objekt.

7.4 En objektorienterad applikation

Nedan visas på ett ungefär hur en OO-applikation kan konstrueras.

BILD

[21] En statisk metod (klassmetod) återkommer.

8 The open closed principle (OCP)

Säger att vi skall förändra eller bygga ut applikationen genom att lägga till kod, inte ändra i gammal d.v.s. open för ny kod men closed för förändring av gammal. Kan ske m.h.a. ny implementioner av ett gränssnitt (se också arv).

9 Sammanfattning

- Programmering mot gränsnitt minskar beroenden.
- Polymorfism gör att vi inte behöver hålla reda på vilken kod som exekveras, rätt kod väljs automatiskt. De former av Vi använder av polymorfism vi använder är; override och overload.
- Overriding (väljs runtime) och overloading (väljs compile time) är väldigt olika!
- Polymorfism kan ersätta selektion.
- Ett subsystem implementeras alltid m.h.a. Facade mönstret och ofta med Factory method.
- OPC säger att vi skall förändra genom att lägga till.