Programmering Fortsättning

Föreläsning 4

Joachim von Hacht

Innehåll

1 Begränsa tillståndsrymden

(En strategi) Idé: Försök minska tillståndsrymden alternativ att tillståndet inte kan ändras eller varför inte helt tillståndslöst!?

1.1 Icke-muterbara objekt

En klass som ges ett väldefinierat starttillstånd som sedan inte kan ändras kallas för icke-muterbart objekt (ett konstanta objekt, immutable object).

OBS! Att vi syftar på *logiskt* icke-muterbart. Omgivande objekt upplever objektet som oföränderligt men internt kan det förändras (sällsynta fall, stor försiktighet)[1].

Observation Icke-muterbara objekt minskar komplexiteten vi behöver inte bekymmra oss om tillståndet (bara att initiera).

- Icke muterade objekt är lätta och säkra att använda (även trådsäkra, se senare...), använd om möjligt (best practises).
- Ett problem är konstruktorn, ett icke-muterbart objekt är muterbart under

[1] Tänkbara fall: Debugging, cache:ing.

tiden kontruktorn exekverar (med något undantag). Kan ställa till problem vid trådade program, se senare, men även vid andra tillfällen[2].

- Dessutom tillkommer problem vid arv...
- Ett icke-muterbart objekt motsvarar ett värde[3]. Ett problem är att man ev. måste skapa många värden[4] eller nya värden (hela tiden). Kan leda till effektivitetsproblem. Å andra sidan kanske man kan återanvända samma värde (riskfritt att t.ex. dela på värdet).

1.1.1 Teknik

Konkret i Java skapar man en klass som saknar muterande metoder, alla attribut är privata och final och sätts i konstruktorn (ingen representationsexponering förekommer).

KOD immutable.*

1.1.2 Exempel: String och StringBuffer

Standardklasser för stränghantering: Den ena muterande en andra icke-muterbar.

- Icke-muterbar[5]: String, operationer på typen innebär att nya objekt skapas och data kopieras (ineffektivet).
- Muterbar: StringBuffer används då vi "arbetar" med en sträng (lägga till, ta bort, m.m.). Omvandling StringBuffer till String sker m.h.a. toString().
- Se upp med +-operatorn för strängar, operatorn "skalar inte" (don't scale) d.v.s. den blir för innefektiv (i tid) om man har för många, långa strängar.

BILD

1.1.3 StringBuilder

Finns en ännu något snabbare men inte trådsäker (återkommer) klass StringBuilder, se trådar.

KOD string

1.2 Begränsat muterbara objekt

Problemet med att det ibland krävs en parameterlös konstruktor dyker upp igen. Kan tänka sig möjlighetern att förändra tillstånd *en* gång t.ex. en init metod. Begränsat muterbara objekt minskar komplexiteten men är trassligare att hantera.

KOD immutable.semi

1.3 Tillståndslösa objekt

Tillståndslösa objekt representerar ren funktionalitet utan sidoeffekter, kallas ibland för "pure" classes. En klass som saknar attribut är tillståndlös.

Observation Tillståndslösa objekt minskar komplexiteten.

Ingen större idé att skapa tillståndslösa objekt använd static.

2 Designmönster

Se kursidor för utförligare beskrivningar.

[2] Se t.ex. Immutable objects in Java , C. Haack et alt.
[3] Enum fungerar precis på detta sätt.
[4] new är en av de mest tidskrävande operationerna i Java.
[5] Dessutom är klassen final innebär att inga subklasser kan skapas, återkommer...

2.1 Singleton

Singleton innebär att vi skapar exakt ett objekt av en klass.

- Minskar tillståndsrymden vi har ju bara ett objekt av klassen.
- Knepigt att få rätt. Rekommenderat är att använda en enum.

KOD singleton

2.2 Factory

Ett Factory är en klass som används för att producera objekt vilka alla implementerar samma gränssnitt (eller subtyp återkommer...). Ofta använder man rent statiska icke-muterbara klasser (allt är static). Några fördelar[6];

- Antalet objekt ska kontrolleras (återanvändas, effektivitet).
- Kan returnara vad som helst som implementerar ett gränssnitt.
- Flexibilitet: Det går inte att ha flera konstruktorer med samma signatur, konstruktorerna heter ju likadant. Med två fabriksmetoder kan man lösa detta (troligen ganska ovanligt).

KOD factory

2.2.1 Teknik

- För att förhindra instansiering görs konstruktorn privat och klassen final (kan inte subklassas återkommer..).

[6]Kommmer troligen efter hand att ersättas av s.k. dependency injection.

- För att användaren lätt skall kunna specificera vilket sorts implementation man vill ha skapar man en enum.

```
public final class MyFactory{

  public enum Items {
     SOME_ITEM,
     OTHER_ITEM
  }
  private static final Map<Items, IA>...
  //No instance possible!
  private MyFactory(){
    :
  }
    :
  public static IA getItem(Items i){
    :
  }
}
```

BILD

KOD factory

2.3 Observer

Fungerar som en prenumeration. Ett observer-objekt registrerar sig hos ett annat objekt (observable). Vid tillståndsförändringen anropa observable observer och meddelar vad som hänt (skicka ev. data). kallas för en push-design, observable trycker ut förändringar till observers. OBS! Att man kan ha flera observers. Samtliga dessa kommer att kontaktas samtidigt, därmed får man en konsistent bild av tillståndet för observable.

- Leder till låg koppling mellan objekt som behöver signalera mellan varann (events) eftersom allt bygger på gränssnitt.

BILD UML

KOD Observer

3 Introduktion till Arv

Grundläggande om arv (inheritance) detaljer kommer...

- Att B ärver (extends, derives, refines) A innebär att i varje *objekt* av typ B ingår ett delobjekt av typ A.
  - Alla B är A men inte tvärt om d.v.s. B är en delmängd av A (A $\supset$B). Kan kännas lite bakfram, B är ju en "större" klass än A (innehåller lika mycket eller mer än A).
  - Kallas implementationsarv.
- A kallas superklass (super class, parent, base) till B och B subklass (sub class, child, derived) till A.
- Arv skapar en partiell ordning mellan super och subklasser, skrivs (ofta) A :> B (A super, B subtyp). Arvsrelationen är reflexiv ($\forall$A; A :> A) och transitiv (A :> B $\wedge$ B :> C => A :> C).

BILD UML

3.1 Arv i Java

3.1.1 Implementationsarv och gränssnittsarv

Implementationsarv (extends) innebär alltså att kod delas mellan super och subklass. En finess men inte något absolut krav för OO. Anges i Java med extends (oturligt nog...).

```
class B extends A {
  :
}
```

Gränssnittsarv (implements) ingen kod delas. Grundläggande princip i OO!

```
class A implements IA {
  :
}
```

- Kombinationer av enkelt implementationsarv och grässnittsarv är tänkbar.

```
// Reuse code from A
// Fulfill contrac IB and IC
class B extends A
          implements IB, IC  {
  :
}
```

Observation Implementationsarv skapar starka beroenden mellan super och subklasser, kod delas.

Gränssnittsarv däremot reducerar beroenden (som vi sett tidigare).

3.1.2 Teknik

- Java tillåter bara enkelt implementations arv, gränssnittsarv kan man som bekant ha multipelt.
- En ny accesspecification; protected = bara subklasser och klasser i *samma paket* kommer åt fält/metod. Undvik! Sammanfattning synlighet;

	Klass	Paket	Subklass	Övriga
public	Ja	Ja	Ja	Ja
protected	Ja	Ja	Ja	Nej
none	Ja	Ja	Nej	Nej
private	Ja	Nej	Nej	Nej

- Om A:> B; A's (fält, metoder) kan användas i B såvida de inte är private.
- Om A:> B och B ligger i annat paket en det vi befonner oss i; Alla A's fält/metoder kan anropas/tilldelas på ett objekt av typ B (såvida de inte är private eller protected).
- Man bör vara mycket försiktig med protected, undvik, låt även subklasser använda get/set (d.v.s. använd private) .

KOD inherit.access(._protected)

3.2 Typomvandlingar

- En subtyp kan alltid omvandlas till sin supertyp (ev. i flera led), görs implict vid behov (subtypen innehåller ju en supertyp-del). Upcasting, widening.
- Tvärtom är osäkert och måste ske med explicit typomvandling, undvik. Jämför gränssnitttyper och implementationer. Downcasting, narrowing.

KOD inherit.casting

3.3 Klassen Object

- Alla klasser ärver implicit klassen objekt d.v.s. Gäller även arrayer (men inte gränssnitt).
  - $\forall$T, T refenstyp, T $\neq$ interface; Object :> T $\wedge$ Object :> T[]
- Förklaringen till att typen Object kan hålla en referens till vilken klass som helst (allt kan typomvandlas till Object).
- Alla klasser i Java, utom Object, har en superklass.
- Object deklarerar ett antal mycket generella metoder, t.ex. toString, equals, m.fl. återkommer, se kanonisk form.

KOD inherit.object

3.4 Super

Vi konstaterar bara helt kort att saker i subklassen kan dölja saker i basklassen (t.ex. m.h.a. override metod, som gränsnitt).

- Om en subklass vill komma åt något dolt i superklassen anger man super före fältet eller metoden. Exempel;

```
class B extends A {
 // Override A's
 public void doIt(){
    :
  }
  :
  // Call B's
  doIt();
  // Call A's
  super.doIt();
  :
}
```

KOD inherit.super

3.5 Initering

Det vi tidigare har sagt gäller men dessutom... När en subklass skapas anropas alltid, implicit, åtminstoner en superklass konstruktor (Object). Sker i omvänd ordning d.v.s. om A :> B :> C så;

- Då C instansieras: C's kontruktor anropar B's som anropar A's. När denna är färdig körs B's färdigt och därefter C's.
- Explict anrop av superklasskonstruktor görs super(...). *Måste göras först i kontruktorn.* Kan behövas då konstruktorn tar argument.
- Se upp med default-kontruktor! Om konstruktorn är overloaded i superklassen finns ingen implicit default-kontruktor (subklassen kan då inte ha default-konstruktor).
- Konstruktor skall aldrig anropa en overridden metod. Metoden kan köras innan objektet är färdigkonstruerat.

KOD inherit.init

3.6 Ärva gränssnitt

- Gränssnitt kan ärva gränssnitt, extends (inte implements).
- Lägger till nya metoder till "baskontraktet".

```
public interface IA {
  // A contract with theese methods
}

public interface IB extends IA {
  // Extended contract
}
```

3.7 Design

- Vi använder arv för att undvika kodduplicering (smell).
- Kod gemensam för subklasser flyttas uppåt. kan även flytta delar av gemensam kod, se senare designmönster Template.

4 Sammanfattning

- Fler strategier för att minska komplexitet;
  - Begränsa antalet objekt (Singleton).
  - Begränsa tillståndet: Icke muterande eller begränsat muterande objekt. Tillståndslösa objekt.
- Gränssnittsarv är problemfritt.
- Implementationsarv öker komplexiteten, skapar en stark koppling mellan klasser. Används i princip bara för kodåtervinning (refactoring common code)!