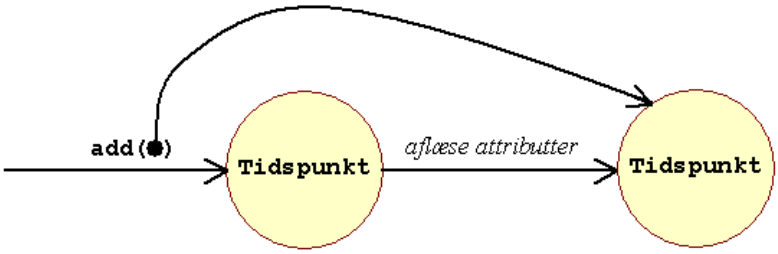
Komposition

Sidst ændret: 07/19/2018 14:11:39
Opgaver

Dette kapitel bygger direkte videre på kapitlet "Klasser", idet der bygges videre på eksemplet med klassen **`Tidspunkt`**. *Der gennemgåes to eksempler på hvordan man kan opbygge små objektsystemer med objekter der til sammen udvider de muligheder klienten har at arbejde med.*

Kortvarig eksistens

Hidtil har vi enten kun lavet ét objekt, eller hvis vi har lavet flere, har de kun kortvarigt haft med hinanden at gøre - f.eks. i forbindelse med metoder som **`add`** og **`sub`**. Lad os se et øjebliksbillede af objekterne under et kald af **`add`**:



Figur 1: *Midlertidigt objektsystem ved kald*

Vi venter mens objektsystemet arbejder

Her har vi et lille objektsystem med to objekter. Først kommunikerer vi selv med det venstre objekt, idet vi sender det en reference til det højre objekt. Dernæst kommunikerer det venstre objekt med det højre objekt og til slut returneres der tilbage til os fra det første objekt. Det at der er to objekter, og at de en overgang kommunikerer med hinanden, bringer liv til objektsystemet. Et øjeblik lever det for sig selv, mens vi venter på, at det løser en opgave for os.

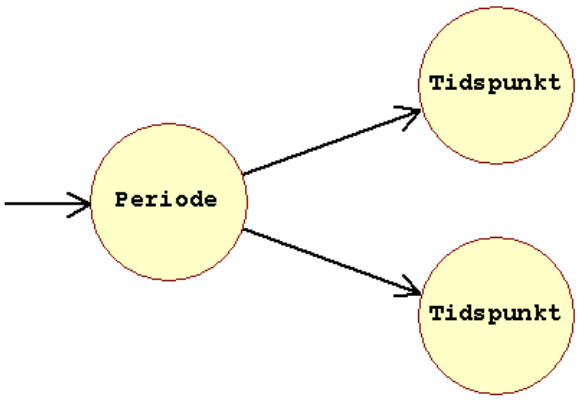
Det er det første objektsystem vi har haft berøring med, og det eksisterer kun et kort øjeblik. Lad os lave et objektsystem der ikke blot er midlertidigt, men lever videre selvom vi for en tid måtte undlade at anvende det.

1. `class Periode`

Start- og slut-tidspunkter

Vi vil lave en klasse **`Periode`**, der repræsenterer en tidsperiode - et tidsinterval. Et sådant interval vil være givet ved et starttidspunkt og et sluttidspunkt. Eftersom vi allerede har en klasse **`Tidspunkt`**, ud fra hvilken vi kan lave objekter der repræsenterer tidspunkter, vil det være oplagt at genbruge den. Vi vil derfor have to instanser af **`Tidspunkt`**: En til at repræsentere starttidspunktet og en anden til sluttidspunktet.

Datakernen i **`Periode`**-objektet bliver kun disse to andre objekter. Lad os se hvordan objektsystemet kommer til at se ud:



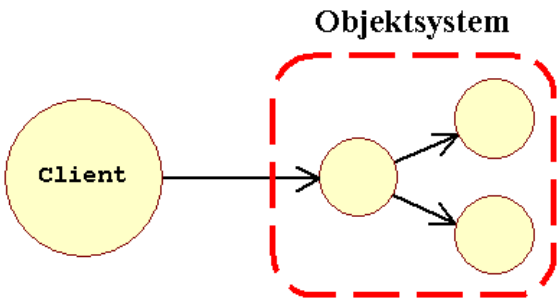
Figur 2: *Objektsystem for* **`Periode`**

Når vi anvender dette objektsystem er det kun **`Periode`**-objektet vi vil kommunikere med. Arbejdet med de to instanser af **`Tidspunkt`** vil blive udført af **`Periode`**-objektet og vi vil ikke mærke noget til det.

1.1 Client

Bruger-rolle

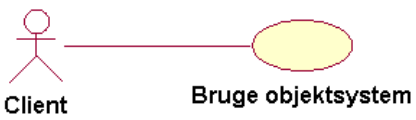
Vi har flere gange sagt at "vi" bruger objektsystemet. Et objektsystem bruges af "nogen", og man kalder denne "bruger" for **klienten** (eng.: client). Man taler om **klient-rollen** og hvad klienten **ser** eller ikke ser:



Figur 3:
Klient med objektsystem

I vores eksempel ser klienten ikke de to **`Tidspunkt`**-objekter, de er skjult bag **`Periode`**-objektet. **`Periode`**-objektet indkapsler de to **`Tidspunkt`**-objekter, så klienten er afskåret fra adgang til dem.

I objektorienteret systemudvikling bruger man use cases til at beskrive funktionelle krav til systemet. Ovenstående figur, med klient og objektsystem, minder om en use case. Vi kunne lave en generel use case ud fra figuren, med følgende udseende:



Figur 4:
Use case version af Figur 3

Selvom det er besnærende at foretage denne sammenligning, er der en forskel. Figur 3 beskriver en objektarkitektur - use casen beskriver en funktionel sammenhæng. Det kan godt være klienten optræder som aktør i en eller flere use cases, hvor den kalder en metode på det forreste objekt i objektsystemet, men det er ikke det figur 3 beskriver - den beskriver kun kendskabet objekterne imellem.

1.2 Information hiding

Man kan sammenligne indkapsling med en osteklokke. En mus, der jo er den klassiske aktør i denne sammenhæng, vil ikke kunne få adgang til osten. Den vil søge et sted, hvor den kan komme ind under osteklokken, men forgæves. Osten er indkapslet og beskytte, og musen vil kun være i stand til at påvirke osten ved at skubbe til selve osteklokken; hvilket må formodes at være utilstrækkeligt, når man tager størrelse af en mus i betragtning. I vores osteklokke (til venstre) mangler der en ost, og en mus vil sikkert ikke vise osteklokken den store interesse. Det er tydeligt for den, hvad der er under osteklokken, og dens adfærd vil afspejle kendskabet til den manglende ost.

I forbindelse med osteklokken har vi indkapsling, men vi har ikke skjult kendskab til hvad der befinder sig inde bag det beskyttende glas. Man kan sammenligne et objekt med en osteklokke. Selve osteklokken er objektet, og dens indhold er datakernen. Datakernen er beskyttet, men kendskabet til datakernens opbygning er her blotlagt - er vi interesseret i det? Lad os vende tilbage til vores **`Periode`**-eksempel.

Eksistens er skjult

Ikke alene indkapsler **`Periode`**-objektet de to **`Tidspunkt`**-objekter, den skjuler også deres eksistens. I sin anvendelse af **`Periode`**-objektet har klienten ingen brug for kendskab til de to **`Tidspunkt`**-objekter. Dette, at klienten ikke ved af de findes, kalder man **information hiding**. Informationen om at de to objekter eksisterer bag **`Periode`**-objektet er skjult. Er det en fordel?

1.3 Kobling mellem objekter

Information hiding er godt!

Information hiding er en klar fordel. Hvis klienten bruger en viden om at de to **`Tidspunkt`**-objekter findes, ville det skabe en **kobling**. En kobling er en afhægighed mellem objekter. Der er tale om en afhængighed mellem et objekt og noget udenfor objektet, noget eksternt.

Der er grundlæggende to måde at lave koblinger mellem objekter på.

1.3.1 Kobling i form af requests

Signaturen skaber kobling

En form for kobling er hvis et objekt sender en request til et andet objekt. Koblingen ligger i hvordan requesten skal formuleres og hvordan returneringen fra den skal forstås. Hvordan requesten skal formuleres betyder i praksis hvad metoden hedder og hvilke parametre den skal have. Forståelsen af det der returneres ligger rent teknisk i returtypen, men også i betydningen af hvad den returnerer.

Ethvert program har mange koblinger, men koblinger skal holdes på et minimum. Hvorfor?

Koblinger laver lækager

Det skyldes at koblinger laver lækager. Det er lækager, hvor igennem ændringer og fejl kan sive ud til resten af objektsystemet. Hvis vi f.eks. vil ændre en metodes navn eller de formelle parametre, vil disse ændringer afstedkomme tilsvarende ændringer alle steder i programmet hvor der foretages kald af den pågældende metode - det kan være mange! Jo flere steder metoden kaldes, jo større arbejde er det, og jo flere chancer er der for at lave fejl.

1.3.2 Kobling i form af hensyn

I eksemplet med de to **`Tidspunkt`**-objekter er det ikke kobling i form af requests der kan skabe problemer. klienten har ingen reference til **`Tidspunkt`**-objekterne og kan derfor ikke kommunikere med dem.

Indrette sig

Derimod kan der opstå en kobling i form af hensyn til at de to objekter findes bag **`Periode`**-objektet. I vores simple eksempel er det måske vanskeligt at forestille sig hvilke hensyn klienten skulle tage til de to objekter, men det kunne f.eks. være, at man designede klienten så den tog hensyn til evt. effektivitetsproblemer, begrænsninger eller andre egenskaber ved dem. På den måde bliver præmisserne for vores anvendelse af **`Periode`**-objektet afhængige af at der ikke sker ændringer i forbindelse med **`Tidspunkt`**-objekterne, som ændrer disse forudsætninger. Sker det, vil vores anvendelse blive uhensigtsmæssig.

1.4 Associering

Der er et par begreber mere som vil hjælpe os til at beskrive det vi laver.

Kendskab

Det første er **associering**. Et objekt er associeret et andet objekt hvis det er kendt af det. I vores eksempel er de to **`Tidspunkt`**-objekter associeret **`Periode`**-objektet. Periode-objektet kender de to instanser af **`Tidspunkt`**, men de kender ikke noget til **`Periode`**-objektet. Der er derfor tale om om **envejs-associering**.

1.4.1 Aggregering

Består af = ejer

Et **aggregat** (eng. aggregate, dk: samling) er noget der er opbygget af andre dele. Når vi taler om aggregering, mener vi at et objekt, "består af" andre objekter. Når vi i komposition **realiserer aggregering**, gør vi det ved at lade nogle objekter være "ejet" af andre objekter - et objekt "ejer" de objekter det "består af".

Variation i betydning

Hvis man studerer den objektorienterede literatur vil man finde mindre variationer i definitionen af aggregering. De strengeste mener at objektet, der er aggregeret et andet objekt, skal instantieres, ejes, bruges og destrueres af kun ét objekt - nemlig det det er aggregeret[1].

Ejer objektet for en tid

Vi vil dog bruge en lidt mindre streng definition; hvilket de fleste også gør. Vores definition vil være, at det aggregerede objekt i *en periode* ejes af det objekt det er aggregeret, og at objektet i den periode kontrolleres af ejeren, der kan tillade andre objekter at anvende det. Man kan sammenligne det med at eje en bil. Den har muligvis tilhørt en anden ejer før vi købte den, men nu er den vores. Det er os der kan bruge den uden at spørge andre om lov, men vi kan også låne den ud, hvis det passer os. Vi kan også vælge at sælge den til en ny ejer. Eksemplet med en bil har dog en mangel: Når vi låner et aggregeret objekt ud, kan vi stadig bruge det, og vi kan låne det ud til flere samtidig.

Objekt-ejer bestemmer

Objektet vil være aggregeret os, men det kan være associeret vilkårligt mange andre objekter. Vi er objekt-ejer - det er os der bestemmer!

1.5 Konstruktorerne

Efter nu at have beskrevet designet af vores objektsystem vil vi gå over til implementationen.

Klassen **`Periode`** skal i første omgang have en eller flere konstruktorer.

Default-, set- og copy-konstruktor? Ja, lad os lave alle tre!

1.5.1 Set-konstruktor

Set-konstruktoren tager datakernen som parameter. Det vil i vores tilfælde være to instanser af **`Tidspunkt`**:

Source 1:

```
public class Periode {
```

Set-konstruktor der overtager objekter

```
  private Tidspunkt start, slut;

  public Periode( Tidspunkt start, Tidspunkt slut ) {
    this.start = start;
    this.slut = slut;
  }

  ...
}
```

Der er en ting vi skal overveje. Når man sender disse to tidspunkter til den nye instans af **`Periode`**, har man så givet afkald på dem? Som nævnt vil de to tidspunkter som et **`Periode`**-objekt bruger, være aggregeret det. Det betyder at **`Periode`**-objektet kommer til at eje objekterne og vi derfor må give afkald på dem.

Vil vi give afkald på dem?

Det er det der sker ved anvendelse af denne set-konstruktor. Den sætter blot referencerne til at referere til de to objekter, så regner den med at det er dens. Konstruktoren besværliggør på denne måde instantieringen af et **`Periode`**-objekt; hvis vi ikke ønsker at give afkald på objekterne. I givet fald må vi lave to kopier som vi sender til konstruktoren så den kan få dem for sig selv.

Vi vil derfor ændre set-konstruktoren så den selv foretager denne kopiering af objekterne, og dermed er bekvem at anvende uanset om man selv vil beholde dem eller ej.

Source 2:
Set-konstruktor der kopierer objekter

```
public Periode( Tidspunkt start, Tidspunkt slut ) {
  this.start = new Tidspunkt( start );
  this.slut = new Tidspunkt( slut );
}
```

1.5.2 Copy-konstruktor

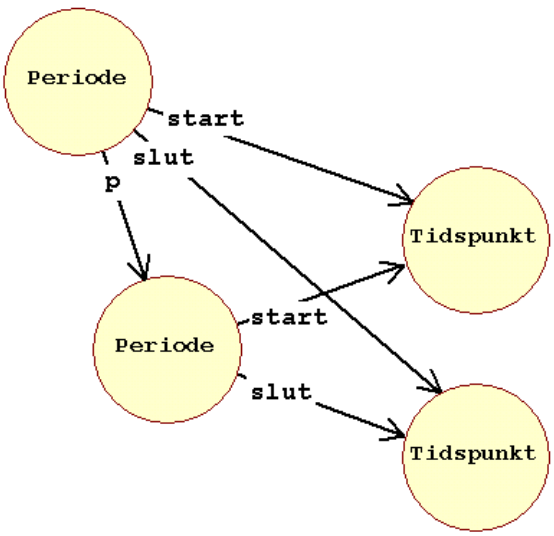
Implementationen af copy-konstruktoren anvender set-konstruktoren:

Source 3:
Copy-konstruktor

```
public Periode( Periode other ) {
  this( other.start, other.slut );
}
```

Man bemærker at copy-konstruktoren ikke behøver lave kopier af de to **`Tidspunkt`**'er, da der foretages en kopiering i set-konstruktoren.

Hvis der ikke skete en kopiering af de to **`Tidspunkt`**'er, ville vi få følgende situation; hvor to **`Periode`**'r var fælles om to **`Tidspunkt`**'er.



Figur 5:
***`Periode`**'r der deles om de samme **`Tidspunkt`**'er*

(i figuren skal **`p`** være **`other`**).

1.5.3 Default-konstruktor

Hvilken periode skal være default? Vi vil vælge en periode der strækker sig fra 0:00:00 til 0:00:00, og dermed over nul tid. Igen anvender vi set-konstruktoren:

Source 4:
Default-konstruktor

```
public Periode() {
  this( new Tidspunkt(), new Tidspunkt() );
}
```

1.6 Metoder

1.6.1 Get-agtige metoder

Returnerer noget ud fra kernen

Hvad "get" er der ved en **get-agtig metode** [*FKJ*]? En get-agtig metode er ikke en get-metode, men den kunne have været det, hvis datakernen var implementeret anderledes. Umiddelbart lyder det som om alle metoder der returnerer noget, kan betegnes som get-agtige, da man vel altid kan forestille sin en form for datakerne; hvor resultatet kan hentes direkte. Om en metode er get-agtig er i sidste ende et skøn - en vurdering af hvor "agtig" den er. Jo mere nærliggende det er, at datakernen kunne være implementeret på en måde, der ville gøre metoden til en get-metode, jo mere get-agtig er den. Betegnelsen har ingen implementationsmæssige konsekvenser, og tjener kun det formål, at den kan bruges til at katagorisere metoder. Ved at betegne en metode som get-agtig sender vi det signal at "den gør noget rimelig enkelt ud fra datakernen og returner resultatet".

For klassen **`Periode`** kunne sådan en metode være **`varighed`**. Idéen med **`varighed`** er, at man kan få oplyst hvor lang tid **`Periode`**'n strækker sig over. Metoden returnerer en instans af **`Tidspunkt`**, der indeholder tidsintervallets længde:

Source 5:
Periodens varighed

```
public Tidspunkt varighed() {
  Tidspunkt v = new Tidspunkt( slut );
  v.sub( start );

  return v;
}
```

Her trækker vi **`start`** fra **`slut`** og returnerer resultatet.

Uformelt kan man se det get-agtige i, at der beregnes noget rimelig simpelt ud fra datakernen og resultatet returneres.

Hvordan skulle datakernen se ud, hvis det skulle have været en rigtig get-metode?

Alternativ kerne

Vi opbevarer perioden som et start- og et sluttidspunkt, men vi kunne også gøre det med et starttidspunkt og en varighed. Sluttidspunktet ville i så fald være starttidspunktet plus varigheden. Med en sådan implementation, vil det være get-metoden for **`slut`**, der blev get-agtig. Det vil ske fordi den nu skal foretage en beregning for at nå frem til sluttidspunktet:

Source 6:
*get-agtig **`slut`**-metode*

```
public class Periode {
  private Tidspunkt start, varighed;

  public Tidspunkt slut() {
    Tidspunkt res = new Tidspunkt( start );
    res.add( varighed );

    return res;
  }

  ...
}
```

Vi skal også have en **`toString`**:

Source 7:
`toString`

```
public String toString() {
  return "[" + start + "-" + slut + "]";
}
```

Klistre-plus kalder selv `toString`

Da vi først undlod at kalde **`toString`** på objekter, var det i forbindelse med **`println`**. **`println`** er som bekendt overloaded, så den selv kalder **`toString`** når man beder den om at "udskrive et objekt". Klistre-plus har samme funktionalitet, den kalder også selv **`toString`**; hvilket vi her har udnyttet i forbindelse med de to tidspunkter ovenfor.

1.6.2 Ændringer

Ændringer af intervallet betyder ændring af start- og/eller slut-tidspunkt.

Ligesom vi i klassen **`Tidspunkt`** implementerede tidsændringer som addition eller substraktion med et andet tidspunkt, således vil vi lave metoder til ændring af intervalgrænserne, der flytter disse på ligenende vis. Lad os først lave en metode **`startBefore`**, der rykker starttidspunktet tilbage, så det starter tidligere:

Source 8: **`startBefore`**

```
public boolean startBefore( Tidspunkt delta ) {
  if ( delta.compareTo( start ) <= 0 ) {
    start.sub( delta );
    return true;
  } else
    return false;
}
```

Ugyldigt interval passerer midnat

Som man ser, foretages der en kontrol. Det der checkes er om der stadig er tale om et gyldigt tidsinterval efter ændringen. Vi kontrollerer at **`delta`** ikke er så stor at **`start`** vil passere tilbage hen over "midnat"; hvilket vi i denne sammenhæng ikke ønsker. Vi vælger kun at udføre ændringen hvis dette ikke sker og returnerer boolsk om intervallet blev ændret.

Dernæst skal man blot supplere **`startBefore`** med **`startLater`**, **`endBefore`** og **`endLater`**, der er fuldstændig tilsvarende.

1.6.3 Sammenligninger

Først sammenligning for lighed:

Source 9: **`equals`**

```
public boolean equals( Periode other ) {
  return start.equals( other.start ) && slut.equals( other.slut );
}
```

En simpel metode der baserer sig på **`Tidspunkt`**'s **`equals`**-metode.

Mht. til sammenligning ved ulighed skal vi beslutte os for hvad vi vil forstå ved at et interval er større end et andet.

Sammenhæng med `equals`

Der skal naturligvis være konsistens med **`equals`**-metoden, dvs. er intervallerne ikke ens, *skal* det ene være større end det andet. Vi vil vælge den ordning, der bygger på hvilket interval der kommer først. Det interval der starter først er mindre end det interval der starter senere. Såfremt de starter samtidig er det sluttidspunktet der er afgørende.

Med denne ordning, bliver metoden:

Source 10: **`compareTo`**

```
public int compareTo( Periode other ) {
  int res = start.compareTo( other.start );

  if ( res == 0 )
    return slut.compareTo( other.slut );
  else
    return res;
}
```

Det er en enkel implementation, idet den bygger på det arbejde vi lavede da vi implementerede den tilsvarende metode i klassen **`Tidspunkt`**.

Løs opgave 7

Den form for sammenligning vi laver mellem to **`Periode`**'r er på sin vis lidt kompliceret, men den understreger hvordan **`Periode`**-objektet løser opgaven ved at arbejde med **`Tidspunkt`**-objekterne, og det er det, der er det centrale her! Man kan forestille sig en mere simpel form for sammenligning, der alene bygger på varigheden af en **`Periode`**. I opgave 7 arbejdes der med en sådan sammenligning - en opgave man bør løse.

1.6.4 Mængde-relaterede metoder

Da vi har et interval af tidspunkter, giver det os mulighed for at behandle tidspunkter ud fra en mængde-betragtning. En **`Periode`** er en mængde af tidspunkter, et kontinuert interval.

Lad os lave en metode der returnerer fællesmængden af to **`Perioder`**.

Bruge `null` som tom mængde

Vi skal ikke fordybe os i selve matematikken, men man kan relativ enkelt se, at *hvis* der er en fællesmængde, så går den fra det seneste af starttidspunkterne til det tidligste af sluttidspunkterne. Hvis der ikke er nogen fællesmængde vil vi lade metoden returnere **`null`**, da ingen instans af **`Periode`** kan fortolkes som repræsentant for den tomme mængde:

Source 11:

Fælles-mængde

```
public Periode fællesMængde( Periode other ) {
  Tidspunkt resStart, resSlut;

  // fællesmængde med tom mængde er altid tom mængde
  if ( other == null )
    return null;

  if ( start.compareTo( other.start ) < 0 )
    resStart = new Tidspunkt( other.start );
  else
    resStart = new Tidspunkt( start );

  if ( slut.compareTo( other.slut ) < 0 )
    resSlut = new Tidspunkt( slut );
  else
    resSlut = new Tidspunkt( other.slut );

  if ( gyldigtInterval( resStart, resSlut ) )
    return new Periode( resStart, resSlut );
  else
    return null;
}
```

Hvor vi har anvendt følgende service-metode **`gyldigtInterval`**. Metoden returnerer boolsk om intervallet er gyldigt.

Source 12:
Service-metode der checker interval

```
private boolean gyldigtInterval( Tidspunkt t1, Tidspunkt t2 ) {
  return ( t1.compareTo( t2 ) <= 0 );
}
```

Med yderligere et par service-metoder bliver **`fællesMængde`** endnu lettere at læse:

Source 13:
Med service-metoder

```
public Periode fællesMængde( Periode other ) {
  Tidspunkt resStart, resSlut;

  resStart = new Tidspunkt( sidst( start, other.start ) );
  resSlut = new Tidspunkt( først( slut, other.slut ) );

  if ( gyldigtInterval( resStart, resSlut ) )
    return new Periode( resStart, resSlut );
  else
    return null;
}
```

Vi vil lade det være en øvelse at implementere service-metoderne **`først`** og **`sidst`**.

Kun sammenhængende intervaller

Det er ikke muligt tilsvarende at lave foreningsmængder og differensmængder, da disse ikke nødvendigvis vil være sammenhængende intervaller, og vi kun er i stand til at repræsenteres sammenhængende intervaller med instanser af **`Periode`**.

1.7 Testanvendelse

Source 14:
Testanvendelse af **`Periode`**

```
public class Main {

  public static void main( String[] argv ) {

    Periode periode1 = new Periode();
    System.out.println( "periode1: " + periode1 );

    Periode periode2 = new Periode( new Tidspunkt( 18, 30, 0 ), new Tidspunkt( 21, 0, 0 ) );
    System.out.println( "periode2: " + periode2 );

    Periode periode3 = new Periode( periode2 );
    System.out.println( "periode3: " + periode3 );

    System.out.println( "Varighed af periode3: " + periode3.varighed() );

    periode2.startBefore( new Tidspunkt( 0, 30, 0 ) );
    System.out.println( "periode2: " + periode2 );

    System.out.println( "periode2 lig periode3: " + periode2.equals( periode3 ) );

    System.out.println( "periode2 compare to periode3: " +
                        periode2.compareTo( periode3 ) );

    Periode periode4 = new Periode( new Tidspunkt( 16, 0, 0 ), new Tidspunkt( 19, 0, 0 ) );
    System.out.println( "periode4: " + periode4 );

    System.out.println( "Fællesmængde mellem periode2 og periode4: " +
                        periode2.fællesMængde( periode4 ) );
```

```
  }
}
```

```
periode1: [[0:00:00]-[0:00:00]]
periode2: [[18:30:00]-[21:00:00]]
periode3: [[18:30:00]-[21:00:00]]
Varighed af periode3: [2:30:00]
periode2: [[18:00:00]-[21:00:00]]
periode2 lig periode3: false
periode2 compare to periode3: -1
periode4: [[16:00:00]-[19:00:00]]
Fællesmængde mellem periode2 og periode4: [[18:00:00]-[19:00:00]]
```

1.8 Objektorienteret vurdering

Inden vi går videre med et andet eksempel, der også anvender **`Tidspunkt`**-klassen, vil vi kort opsummere og beskrive **`Periode`**'s objektorienterede egenskaber/kvaliteter.

`Periode` har to instanser af **`Tidspunkt`** aggregeret, eller i mere abstrakte termer: en **`Periode`** *består* af to **`Tidspunkt`**'er.

`Periode` ikke alene indkapsler de to instanser af **`Tidspunkt`**, den skjuler også deres eksistens overfor klienten.

Implementationen af **`Periode`** arbejder kun med instanser af **`Tidspunkt`**, der arbejdes *aldrig* med timer, minutter eller sekunder.

2. `class Dato`

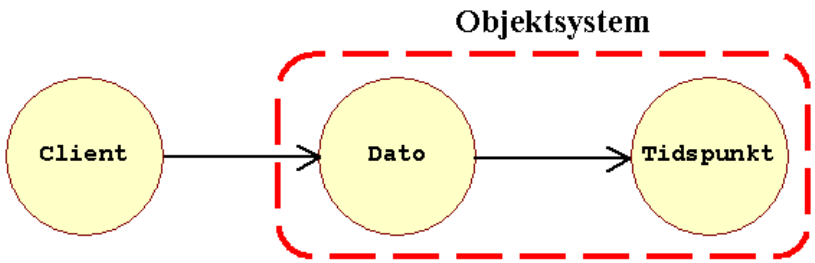
Dato med tidspunkt

Klassen **`Tidspunkt`** er begrænset til et klokkeslæt, men man kunne måske ønske en klasse der repræsenterede en hel **dato**. Vi vil derfor lave en klasse **`Dato`** til dette formål. Vi vil forstå en dato som en angivelse af ikke alene år, dag og måned men også af timer, minutter og sekunder - altså en udvidet forståelse i forhold til den gængse.

Tidspunkt aggregeret

Designet opdeler repræsentations-opgaven i to dele. **`Dato`**-objektet skal have et tidspunkt aggregeret, og selv tage sig af den del af datoen som tidspunktet ikke kan klare, nemlig år, dag og måned.

Objektsystemet kommer derfor til at indeholde to objekter, her sammen med den altid nærværende klient.



Figur 6: *Objektsystem for dato*

Klasse-skelettet med instansvariable bliver:

```
public class Dato {
  private Tidspunkt tidspunkt;
  private int totalDage;

  ...
}
```

Source 16: *Instans-variable*

Vi vil, som for klassen **Tidspunkt**, arbejde med ét samlet tal - i dette tilfælde det totale antal dage. Ud over dette, indeholder datakernen også en reference til **Tidspunkt**-objektet.

Vi fokuserer ikke på komplikationer

I forbindelse **totalDage** er der nogle komplikationer omkring det at vi ikke i det daglige angiver månederne med tal fra 0-11 (dvs. januar som 0, og december som 11), og tilsvarende problemer omkring dage - for ikke at nævne år, hvor der jo ikke findes noget der hedder år 0 (vi lader dog som at dette er tilfældet i vores implementation). Det betyder at

der enkelte steder i vores implementation bliver trukket én fra eller lagt én til månederne og dagene. Vi vil forbigå det ubemærket når det sker, og overlade det til den interesse som læseren eventuelt selv måtte have for disse detaljer.

2.1 Konstruktorer

Vi vil lade default-konstruktoren initialisere datoen til midnat d. 1/1 år 0:

Source 17:
Default-konstruktor

```
public Dato() {
  tidspunkt = new Tidspunkt();
  totalDage = 0;
}
```

Copy-konstruktoren er også enkel, idet den for tidspunktets vedkommende hviler på **`Tidspunkt`**'s copy-konstruktor:

Source 18:
Copy-konstruktor

```
public Dato( Dato other ) {
  tidspunkt = new Tidspunkt( other.tidspunkt );
  this.totalDage = other.totalDage;
}
```

Set-konstruktoren er et større problem, for hvad skal den gøre? Datakernen er sammensat af to dele! En set-konstruktor vil efter definitionen skulle have følgende udforming:

Source 19:
Set-konstruktor

```
public Dato( int totalDage, Tidspunkt tidspunkt ) {
  this.tidspunkt = new Tidspunkt( tidspunkt );
  this.totalDage = totalDage;
}
```

Type-mæssigt blandet flok

Men det synes oplagt at vi ikke ønsker at udsætte den der ønsker at instantiere en Dato for en sådan række af parametre. Specielt **totalDage** er ikke noget vi ønsker man udenfor klassen skal forholde sig til.

Følgende konstruktor er mere ensartet i abstraktionsniveauet mht. parametrene, men til gengæld er den meget konkret, og håndterer ikke tidspunktet som en instans af **Tidspunkt**:

Source 20:
Med primitive parametre

```
public Dato( int år, int måneder, int dage, int timer, int minutter, int sekunder ) {
  this.tidspunkt = new Tidspunkt( timer, minutter, sekunder );
  totalDage = getTotalDage( år, måneder, dage );
}
```

Selv om abstraktionsniveauet er lavt, med angivelse af disse seks parametre, er det nok en sådan parameterliste man vil finde praktisk når man skal instantiere en **Dato**, og det bliver derfor vores bud på en set-agtig konstruktor.

2.2 Metode

2.2.1 get-/set-metoder

Vi skal have en række get-/set-metoder, specielt fordi de danner grundlag for omregningen mellem år, måned og dag, og **totalDage**. Disse metoder ligner meget de tilsvarende fra klassen **Tidspunkt**.

Først skal vi have en række konstanter, der skal bruges i vores beregninger:

Source 21:
Nyttige konstanter

```
private static final int MÅNED = 30;
private static final int ÅR = 12 * MÅNED;
```

Dernæst skal vi have en metode, der kan regne fra år, måned og dag, til **totalDage**:

Source 22:
Omregning til totalDage

```
private int getTotalDage( int år, int måneder, int dage ) {
  return år * ÅR + ( måneder - 1 ) * MÅNED + ( dage - 1 );
}
```

Som det ses af **private**, er det en service-metode vi kun vil bruge internt i klassen.

Og endelig tre get-metoder for år, måned og dag:

Source 23:
De tre get-metoder

```
public int getÅr() {
  return totalDage / ÅR;
}

public int getMåneder() {
  return ( totalDage % ÅR ) / MÅNED + 1;
}

public int getDage() {
  return totalDage % MÅNED + 1;
}
```

Eftersom **`toString`**-metoden baserer sig på get-metoderne vil vi medtage den her under samme afsnit::

Source 24:
`toString`

```
public String toString() {
  return "[" + getDage() + "/" + getMåneder() + "-" + getÅr() + " " + tidspunkt + "]";
}
```

Vi skal også have en række set-metoder. Først den metode der kan sætte alle tre på én gang:

Source 25:
Samlet set-metode

```
private void set( int år, int måneder, int dage ) {
  totalDage = getTotalDage( år, måneder, dage );
}
```

Som det (igen) ses af **private**, er det en service-metode vi kun vil bruge internt i klassen.

Dernæst skal vi have de tre set-metoder til år, måned og dag:

Source 26:
De tre get-metoder

```
public void setÅr( int år ) {
  set( år, getMåneder(), getDage() );
}

public void setMåneder( int måneder ) {
  set( getÅr(), måneder, getDage() );
}

public void setDage( int dage ) {
  set( getÅr(), getMåneder(), dage );
}
```

Som det ses er disse get-/set-metoder meget "lig" de metoder vi lavede i forbindelse med **Tidspunkt**-klassen.

2.2.2 Ændringer

`Tidspunkt` er designet til at være lidt for selvstændig

I forbindelse med ændring af datoer kunne det være nyttigt at have to metoder analogt til **`add`** og **`sub`**, som vi kender dem fra **Tidspunkt**-klassen. På den måde ville vi kunne ændre datoen relativt. I den forbindelse får vi brug for at regne med det aggregerede tidspunkt. Selve klokkeslættet der befinder sig i det aggregerede objekt er mindst betydende i forhold til år, dag og måned som vi selv implementerer. Det betyder at vi ved addition og substraktion får behov for at kunne regne videre med en mente i form af dage, efter vi har enten adderet eller substraheret tidspunkterne. **`Tidspunkt`** er ikke designet med mulighed for at levere en mente. Den er lavet så den kan klare sig selv. Det betyder at **`add`** og **`sub`** ikke kan hjælpe os. Hvad gør vi så?

Dårlig idé at ændre `add` og `sub`

En mulighed var at ændre **`add`** og **`sub`** i **Tidspunkt** så de passer til vores behov. Det er en meget dårlig idé!

Vi har allerede anvendt instanser af **`Tidspunkt`** i forbindelse med klassen **`Periode`**, og man kunne forestille sig, at den på nuværende tidspunkt havde været anvendt i mange andre sammenhænge. Den kobling der f.eks. er mellem **`Periode`** og **`Tidspunkt`** betyder, at en ændring af hvad **`add`** og **`sub`** returnerer, ville sprede sig til **`Periode`**. Hvad gør vi i stedet?

En bedre løsning er at udvide **`Tidspunkt`**'s interface med to specielle metoder der returnerer den mente vi ønsker. På den måde vil det ikke kræve ændringer andre steder.

Source 27:
Ekstra metoder til mentes-regning

```
public class Tidspunkt {

  ...
```

```
  private static final int DØGN = 24 * TIME;

  ...

  public int menteAdd( Tidspunkt other ) {
    int res = this.totalSekunder + other.totalSekunder;

    totalSekunder = res % DØGN;

    return res / DØGN;
  }

  public int menteSub( Tidspunkt other ) {
    int res = this.totalSekunder - other.totalSekunder;

    if ( res >= 0 ) {
      totalSekunder = res;
      return 0;
    } else {
      totalSekunder = res + DØGN;
      return 1;
    }
  }

  ...
}
```

Vi vil ikke fordybe os i hvordan disse to metoder er implementeret. **menteSub** kunne laves lidt kortere, men vi har prioritet at den skulle være mere læsevenlig.

Ved at anvende disse to nye metoder kan vi implementere de to metoder til ændring af datoen:

Source 28:
Metoder til ændring af dato

```
public class Dato {

  ...

  public void add( Dato other ) {
    int mente = tidspunkt.menteAdd( other.tidspunkt );
    this.totalDage += other.totalDage + mente;
  }

  public void sub( Dato other ) {
    int mente = tidspunkt.menteSub( other.tidspunkt );
    this.totalDage -= other.totalDage + mente;
  }
}
```

Generelt design er mere holdbart

I forbindelse med de to metoder til ændring, fik vi problemer med **`Tidspunkt`**. Det er ikke altid muligt at forudse alle fremtidige anvendelser når man designer en klasse, men bestræber man sig for at gøre metoderne så generelle som muligt vil de være mest holdbare. Vores udgangspunkt for at designe **`Tidspunkt`** var netop meget generelt. Det viste sig senere at det krævede nogle specialiserede metoder at implementere ændring i **`Dato`**, og netop derfor var **`Tidspunkt`** ikke forberedt.

2.2.3 Sammenligninger

Igen skal vi bruge en **`equals`**-metode til test af lighed:

Source 29:
Lighed

```
public boolean equals( Dato other ) {
  return this.totalDage == other.totalDage &&
         tidspunkt.equals( other.tidspunkt );
}
```

Metoden overlader delvist opgaven til **`Tidspunkt`**-objektet og klarer selv sin egen del.

Ved implementationen af **`compareTo`** foretages der først en sammenligning af dato-delen, da det er den mest betydende. Hvis der er tale om samme dag overlades opgaven til **`Tidspunkt`**-objektet.

Source 30:
Ulighed

```
public int compareTo( Dato other ) {
  int sign = signum( this.totalDage - other.totalDage );

  if ( sign == 0 )
    return tidspunkt.compareTo( other.tidspunkt );
  else
    return sign;
}
```

Vi har her anvendt den samme **signum**-metode som i **Tidspunkt**-klassen, og den skal derfor kopieres over i **Dato**-klassen. Vi skal senere se hvordan man kan fjerne denne form for koderedundans i et senere kapitel (i forbindelse med

nedarvning).

2.3 Testanvendelse

Source 32:
Testanvendelse af **`Dato`**

```
public class TestDato {

  public static void main( String[] argv ) {
    Dato dato1 = new Dato();
    System.out.println( "dato1: " + dato1 );

    Dato dato2 = new Dato( dato1 );
    System.out.println( "dato2: " + dato2 );

    System.out.println( "dato1 lig dato2: " + dato1.equals( dato2 ) );

    Dato dato3 = new Dato( 1999, 10, 3, new Tidspunkt( 23, 43, 04 ) );
    Dato dato4 = new Dato(    1,  3, 8, new Tidspunkt(  2,  2,  5 ) );
    System.out.println( "dato3: " + dato3 );
    System.out.println( "dato4: " + dato4 );
    dato4.add( dato3 );
    System.out.println( "dato4 += dato3: " + dato4 );

    System.out.println( "dato3 compare to dato4: " + dato3.compareTo( dato4 ) );

    dato4.sub( dato3 );
    System.out.println( "dato4 -= dato3: " + dato4 );
  }
}
```

```
dato1: [1/1-0 [0:00:00]]
dato2: [1/1-0 [0:00:00]]
dato1 lig dato2: true
dato3: [3/10-1999 [23:43:04]]
dato4: [8/3-1 [2:02:05]]
dato4 += dato3: [11/12-2000 [1:45:09]]
dato3 compare to dato4: -1
dato4 -= dato3: [8/3-1 [2:02:05]]
```

3. Indirection

Noget via noget andet

Når vi arbejder med komposition bruger vi indirection. Indirection er, når man arbejder med noget via noget andet. Man kunne oversætte det til "indirektethed", men mit sprogøje vil nok foretrække den engelske betegnelse i mange år endnu.

Reference

Den simpleste form for indirection er en reference. Det er vha. en reference vi arbejder med et objekt. F.eks. sender vi en request til et objekt *via* en reference. Fordelen ved en reference er at den i løbet af programudførelsen kan referere til forskellige objekter. Det gør referencen **dynamisk**. Referencer er grundlæggende i objektorienteret indirection[2], og mange andre objektorienterede former for indirection bygger på referencer.

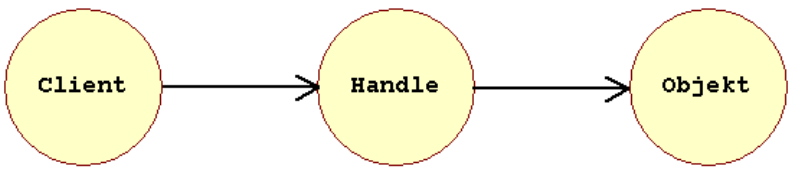
Et ekstra lag

I de to eksempler med **`Periode`** og **`Dato`** placerede vi et objekt mellem **`Tidspunkt`**-objekteterne og klienten. I stedet for at arbejde direkte med tidspunkterne, havde vi objektet til at hjælpe os. Det at sætte et objekt imellem, skaber et ekstra "lag" af indirection. Denne form for indirection vil vi se nærmere på.

3.1 Handle pattern[3]

Den mest enkle form er et **Handle pattern**. Et handle er et objekt, som har den simple opgave at "holde" et andet objekt for klienten:

Figur 7:
Objektsystem med Handle pattern



Handle formidler kommuni-kationen

Enhver request sendt til **handle** bliver sendt videre til objektet. Klienten ved, at handle ikke er det "rigtige" objekt, men sender requests til handle i den forvisning, at handle formidler kommunikationen med objektet (hvis klienten ikke vidste det, ville man kalde det en **proxy** - se evt. Proxy Pattern (dette kapitel er ikke skrevet endnu!))

Klient ikke

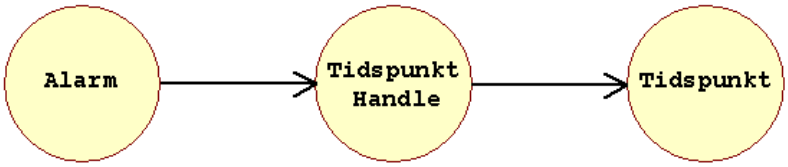
Hvilke fordele er der ved dette pattern? Det giver os først og fremmest en dynamisk fordel vedrørende udskiftning af objektet. Med det ekstra lag af indirection behøver klienten ikke at blive involveret i en sådan udskiftning. Kun handle

involveret i udskiftning

behøver at kende til skiftet, da den har den direkte reference til objektet. Lad os se et eksempel.

I figuren nedenfor er klienten en alarm. Alarmen er tidsindstillet og har derfor et behov for at vide hvornår den skal lade alarmen lyde. Denne "husk et tidspunkt"-opgave lægges ud til et **`Tidspunkt`**-objekt. Ved at spørge **`Tidspunkt`**-objektet kan alarmen undersøge om tidspunktet er inde.

Figur 8:
Alarm med handle til tidspunkt



Lad os se hvordan et sådant handle kunne være implementeret.

Source 33:
Handle-klassen

```
public class TidspunktHandle {
  private Tidspunkt objekt;

  public TidspunktHandle( Tidspunkt objekt ) {
    setObjekt( objekt );
  }

  public void setObjekt( Tidspunkt objekt ) {
    this.objekt = objekt;
  }

  public int compareTo( Tidspunkt other ) {
    return objekt.compareTo( other );
  }

  ...
}
```

`TidspunktHandle` har en set-konstruktor og en set-metode, men ellers er alle andre metoder de samme som **`Tidspunkt`**'s; hvor disse dog i handle er implementeret med simple kald videre til objektet. Dette er vist for **`compareTo`**, som klienten kalder når den vil checke om det er tid.

Source 34:
Klient-klassen

```
public class Alarm {
  private TidspunktHandle tidspunkt;

  public Alarm( TidspunktHandle tidspunkt ) {
    this.tidspunkt = tidspunkt;
  }

  ...

    if ( tidspunkt.compareTo( nu ) <= 0 )
      // alarm!!!

  ...
}
```

Variablen **`nu`** er en reference til et **`Tidspunkt`**-objekt der indeholder det nuværende tidspunkt.

Vi kunne anvende disse klasser ved følgende:

Source 35:
Testanvendelse

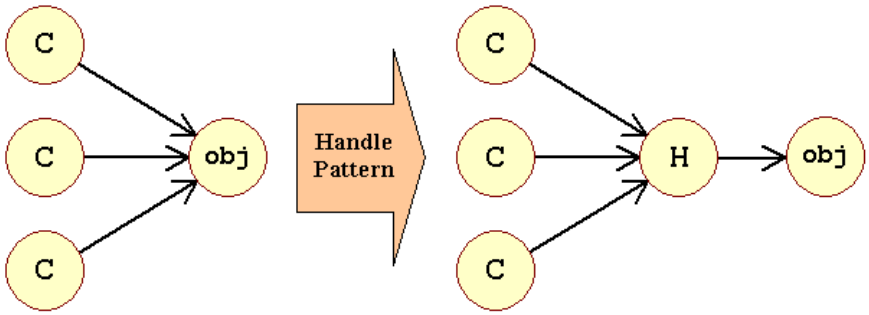
```
public class Main {

  public static void main( String[] argv ) {
    Tidspunkt alarmTid     = new Tidspunkt( 8, 55, 0 );
    TidspunktHandle handle = new TidspunktHandle( alarmTid );
    Alarm alarmen          = new Alarm( handle );

    ...

    // Udskiftning af alarm-tidspunktet uden at alarmen
    // er involveret

    handle.setObjekt( new Tidspunkt( 9, 50, 0 ) );

    ...
  }
}
```

"Skifte-kode" isoleret i handle

Eksemplet er meget enkelt og man ser ikke nogen tydelig gevinst ved at holde alarmen undenfor skiftet. Generelt kan det dog være en fordel, hvis "en anden" ofte har brug for at udskifte objektet. Ved at holde klienten udenfor, behøver man ikke komplicere den med "skifte-kode", men kan i stedet isolere denne kode i handle.

Et godt eksempel på den fordel man kan få af at anvende Handle Pattern, er tilstedeværelsen af flere klienter der anvender det samme objekt via et handle:

Figur 9:
Alternative design-muligheder



I stedet for den direkte forbindelse fra klienter til objekt, vil et design med et handle, lette opgaven med at meddele udskiftning af objektet. Hvis klienterne har direkte adgang til objektet skal de alle have besked om udskiftningen, men med et mellemliggende handle, skal kun dét have besked.

3.2 Adapter pattern

Adapter tager sig af alle forskelligheder

Koblingen er blevet svagere, men den kan gøres meget svagere, hvis vi i stedet anvender det der kaldes **Adapter pattern**. Den tilbageværende kobling ligger i, at metoderne skal hedde det samme. Hvis vi i stedet tillader at metoderne i handle ikke behøver have de samme navne som i objektet, vil en ændring af disse ikke sprede sig til klienten, men ikke komme længere end til handle. En adapter skal fungere som tolk mellem klienten og objektet. Det er adapterens opgave at udligne alle forskelligheder og oversætte modtagne requests, til requests til objektet.

Hvis vi udskifter **`TidspunktHandle`** med en adapter kunne den undlade at tilbyde samtlige metoder, og den kunne lave en metode der mere rettede sig mod alarmens behov:

Source 36:
Adapter-klasse

```
public class TidspunktAdapter {
  private Tidspunkt objekt;

  public TidspunktAdapter( Tidspunkt objekt ) {
    setObjekt( objekt );
  }

  public void setObjekt( Tidspunkt objekt ) {
    this.objekt = objekt;
  }

  public boolean tidenErInde( Tidspunkt nu ) {
    return ( objekt.compareTo( nu ) <= 0 );
  }
}
```

Alarmen ville nemmere kun anvende denne adapter.

Source 37:
Anvendelse af specialiseret metode

```
public class Alarm {
  private TidspunktAdapter tidspunkt;

  public Alarm( TidspunktAdapter tidspunkt ) {
    this.tidspunkt = tidspunkt;
  }

  ...

    if ( tidspunkt.tidenErInde( nu ) )
      // alarm!!!

  ...
}
```

Koblings-problemer isoleret i adapteren

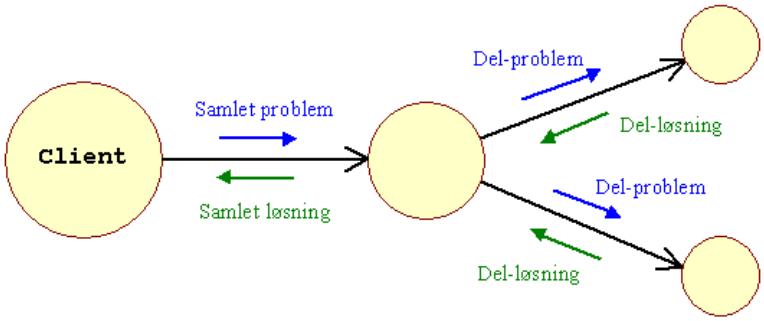
Koblingen mellem klienten og tidspunktet er nu væk, og den er nu delt ud på en kobling mellem klient/adapter og adapter/objekt. Her har vi gjort et godt bytte. Koblingen mellem klient/adapter får vi sandsynligvis aldrig problemer med, da det alene er adapterens opgave at tilpasse sig klienten. Tilsvarende er det adapterens opgave at tilpasse sig objektet, så klient og objekt skulle nu være forskånede for koblings-problemer. Koblings-problemerne er blevet isoleret i adapteren!

3.3 Delegering

Nedbrydning i del-problemer - samling af delløsninger

Der er en grundlæggende samarbejdsform, der karakteriserer både Handle Pattern og Adapter Pattern. Det er **delegering** (også kaldet Delegation Pattern). Delegering betyder, at et objekt besvarer en request fra klienten i samarbejde med andre objekter. Requesten har form af et problem der skal løses, og objektet løser det ved helt eller delvist at sende det videre til et eller flere andre objekter. Objektets opgave er dermed reduceret til at kombinere de delløsninger den modtager fra de andre objekter, så den til slut kan returnere den samlede løsning til klienten.

Figur 10:
Delegering ved opdeling i del-problemer



(Dette er måske den vigtigste figur i hele DocJava!)

I Handle Pattern er delegeringen meget simpel, da hele problemet delegeres videre og handle derfor blot skal returnere løsningen til klienten. I Adapter Pattern er det mere interessant.

Adapteren tolker

Adapteren er mere avanceret. Den skal løse et problem ved at omsætte det til passende kald af et andet objekt. På den måde delegerer den problemet videre, men den kan også få behov for at "oversætte" del-løsninger til den form som klienten ønsker. Adapteren kan derfor få behov for at tolke begge veje, både ved kald og ved returnering.

---

fodnoter

1 Denne strenge opfattelse af aggregering stammer fra programmeringssprog, hvor man kan **indlejre** objekter i hinanden. I Java bruger vi altid en reference-variabel til at håndtere et objekt, men i andre sprog er det muligt at erklære variable, der direkte *er* objekter. Med sådanne objekter er det en tvivlsom praksis at dele dem med andre, da det åbner det objekt de er aggregeret, og på den måde bryder indkapslingen.

Ejerskabet er af denne grund mere ensidigt og permanent for indlejrede objekter. Man bruger i den forbindelse ofte betegnelsen "en del af" som synonym for aggregering.

2 **Funktionel indirection** er et eksempel på indirection, der ikke er objektorienteret. Funktionel indirection er når metoder (funktioner) kalder hinanden. Når en metode kalder en anden metode, kunne den i stedet selv gøre dét den kaldte metode udfører. I så fald ville der ikke være nogen indirection - metoden gør det selv. Når den i stedet vælger at kalde en anden metode, bliver kaldet en indirection fordi der nu sker "noget via noget andet", nemlig at koden i den anden metode bliver udført via kaldet.

I objektorienteret indirection er der også tale om metodekald, men via referencer. Metodekaldene i funktionel indirection er statiske, da det altid vil være den samme metode der bliver kald. Med referencer, vil signaturen også være den samme, men det kan være forskellige metoder alt efter hvilket objekt der er i den anden ende af referencen (Det vil vi studere i forbindelse med **polymorfi**).

Get-/set-metoder er et andet eksempel på funktionel indirection. Alternativet er at gøre instans-variable public så de kan tilgås direkte via en reference (der er i begge tilfælde også tale om objektorienteret indirektion). I stedet sætter vi get-/set-metoder imellem og skaber et nyt lag af indirection, idet vi nu har adgang via noget andet, nemlig get-/set-metoderne.

3 Her, og i det følgende, vil vi bruge betegnelsen **pattern** (dk.: **mønster**) uden at berøre hvad et pattern præcist er. Det eneste der er nødvendigt at vide om patterns på dette sted er, at et pattern beskriver fællestræk ved en gruppe af forskellige løsninger på problemer/opgaver. Det kan f.eks. være en karakteristik af hvordan objekter samarbejder om at løse en opgave, uden at man går i detaljer med hvilken opgave de løser, men i stedet fokuserer på hvordan de arbejder sammen om at løse den.

Repetitionsspørgsmål

1 Hvorfor er objektsystemerne der opstår i forbindelse med `add` og `sub`, i klassen `Tidspunkt`, kun midlertidige?

2 Hvad er klientens rolle?

3 Hvad er information hiding?

4 Hvad er koblinger mellem objekter?

5 Hvilke slags koblinger findes der?

6 Hvorfor er koblinger et problem?

7 Hvad er associering mellem objekter?

8. Hvad er aggregering?

9. Hvorfra stammer en strengere opfattelse af aggregering, end den vi bruger?

10. Hvad skal man tage stilling til når man sender objekter til en set-konstruktor?

11. Hvad er get-agtige metoder?

12. Hvad sker der når man anvender en reference i forbindelse med klistre-plus?

13. Hvorfor er fællesmængden den eneste mængdeoperation, det er muligt at realisere med **`Periode`**?

14. I forbindelse med implementationen af metoderne til ændring af datoer, er det en dårlig idé at ændre **`add`** og **`sub`** i **`Tidspunkt`**-objektet - Hvorfor?

15. Hvad er fordelen ved at lave metoder generelle?

16. Hvad er indirection?

17. Hvilken rolle spiller referencer i indirection?

18. Hvad er Handle pattern?

19. Hvad er fordelen ved Handle pattern?

20. Hvad er Adapter pattern?

21. Hvad er fordelen ved Adapter pattern?

22. Hvad er delegering?

Svar på repetitionsspørgsmål

1. Fordi de kun eksisterer under kaldet

2. Klienten har opgaven/problemet

3. At informations *eksistens* er skjult

4. Afhængigheder mellem objekter

5. Koblinger i form at metode-kald og i form af hensyn

6. Fordi de er lækager hvor igennem ændringer og fejl kan sprede sig

7. Kendskab. Det vil i praksis sige: referencer

8. Ejerskab

9. Fra sprog hvor man kan indlejre objekter i hinanden

10. Om man derved giver afkald på det, og om man i givet fald vil det

11. Metoder der gør noget rimeligt simpelt ud fra kernen og returnerer resultatet. Rent teknisk er det metoder der kunne have været get-metoder, hvis kernen havde været implementeret anderledes

12. Klistre-plus kalder selv **`toString`**, analogt til **`println`**

13. Fordi **`Periode`** kun kan repræsentere ét sammenhængende interval

14. Fordi der vil kræve ændringer alle steder metoderne kaldes

15. At de kan anvendes i flere sammenhænge, og derfor bliver mere holdbare

16. At man gør noget via noget andet

17. Man foretager kald vi en reference til et andet objekt

18 I Handel pattern sætter man et objekt imellem som kun har til formål at sende requests direkte videre til det andet objekt

19 At man kan udskrifte det andet objekt uden at involvere det første

20 I Adapter pattern sætter man et objekt imellem som skal fungere som tolk mellem de to objekter

21 At man får isoleret forskellighederne i adapteren, og dermed også får placeret de væsentligste koblingsproblemer i adapteren

22 At man sender en opgave, helt eller delvist, videre til et andet objekt