Det är bra om man kan se när ett mönster används i programkod?

Hur signalerar man att kod följer mönstret Template Method?

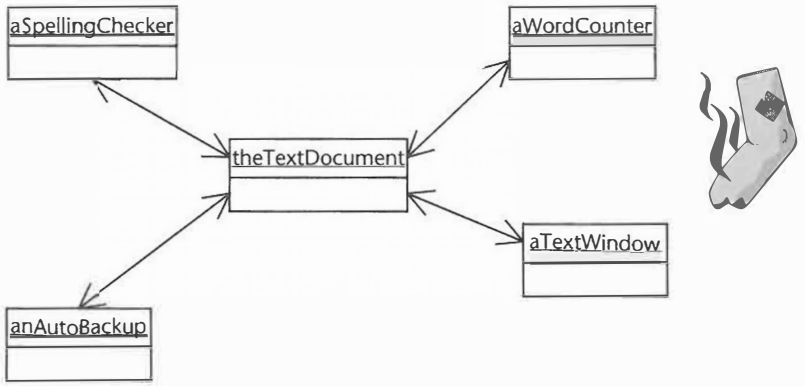
5.8 Att enkelrikta beroende: *Observer*

Kärt barn har många namn. Originalidén till detta mönster kommer från programmeringsspråket Smalltalk, där man i dess bibliotek ville skilja på en modell och dess visualisering, *Model/View*. Denna idé har sedan förvaltats vidare och givits många namn som i praktiken uttrycker samma sak: när ett objekttillstånd uppdateras kan det finnas ett antal andra objekt som måste få reda på detta för att i sin tur göra något. GoF valde att kalla det *Observer*, ett objekt som behöver kunna observera ett annat objekts tillståndsändringar. Det kan gälla en visualisering men generaliseras lätt till andra behov.

Text-dokument med observatörer

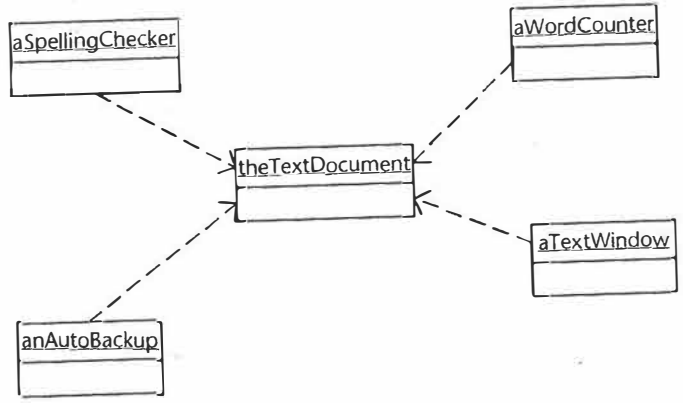
Låt oss ta ett exempel med en textredigerare. Denna innehåller själva textdokumentobjektet. Till detta objekt är knutet ett antal objekt som på olika vis är beroende av texten: ett fönster som visar texten, ett autobackuphanterare, en ordräknare, en stavningskontrollant, etc.

Om nu textobjektet självt skulle hålla reda på alla sina observatörer, deras behov, funktion och gränssnitt blir det snabbt rörigt eftersom observatörerna naturligtvis också måste känna till textobjektet för att kunna göra sitt jobb. Det blir också svårt att lägga till nya observatörer eftersom textobjektet då måste anpassas för den nya observatören. Vi har ett dubbelriktat ett-till-många-och-tillbaka-till-ett beroende:



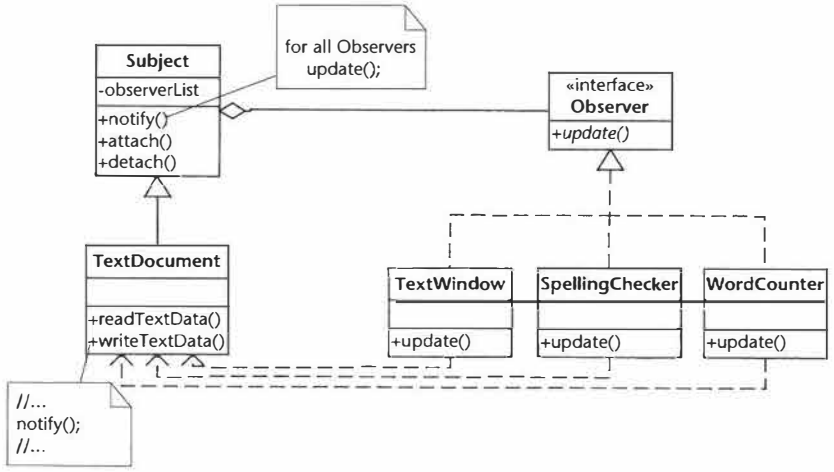
Figur 5.13 Onödiga dubbelriktade beroenden

Låt oss göra om detta till ett enkelriktat beroende. Textobjektet skall inte behöva känna till vilka olika sorters observatörer som observerar. Det blir likt en tidning som inte behöver veta exakt vilka sorts prenumeranter den har, därav ett av alternativnamnen på mönstret: *publish/subscribe:*



Figur 5.14 Beroendena enkelriktade

Liksom tidningen ser sina prenumeranter utan att veta vilken sort de är, måste nu textobjektet hålla reda på alla observatörerna, dock utan att exakt behöva veta vilken sort de är. Eftersom detta alltid blir samma jobb kan vi samla det i en bibliotekssuperklass till textobjektet. Denna klass håller reda på vilka som för tillfället observerar och kan dessutom sköta distributionen av info till observatörerna: något har hänt med textobjektet, se figur 5.15.

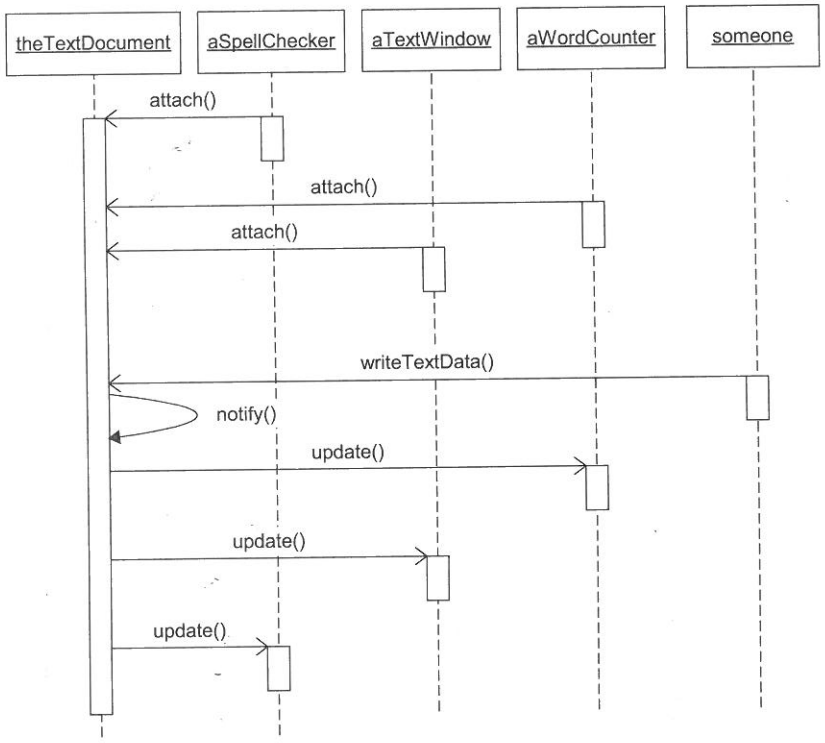


Figur 5.15 Klassdiagram för dokumentobservatörer

Det innebär att vi befriat textobjektet från ansvaret att hålla reda på vilka som observer det och i och med detta förvandlat dubbelriktat beroende till enkelriktat. Textobjektet kan nu inte längre skicka specialinformation till varje observatör, men det offrar vi för beroendeminskningens skull. Nu kan vi dock lägga till nya observatörer utan att påverka vare sig textobjektet eller andra observatörer. Textobjektet gör *notify* på sig själv och låter *Subject*-delen ta hand om distributionen, se figur 5.16.

Detta mönster har många alternativnamn.

Går det att hitta goda argument för dem alla: Subject/Observer, Model/View, Publish/Subscribe, Observable/Observer, Document/View respektive Dependents?



Figur 5.16 Sekvensdiagram för dokumentobservatörer

Tillämpning

Observer är ett beroende-enkelriktar-mönster. Ett eller flera objekt är beroende av ett annat objekt (kallat *Subject*) och behöver få reda på ändringar i dess tillstånd. I stället för att subjektet känner till och meddelar alla objekten samlar vi informationen om dessa i en superklass till subjektet.

Observer

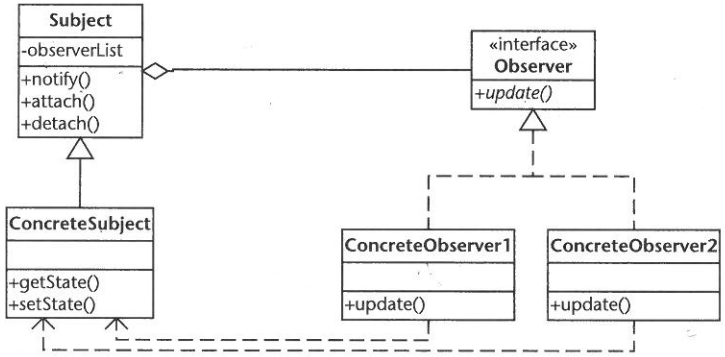
Andra namn – översättning

Dependents, model/view, publish/subscribe, document/view – observatör

Syfte

Att definiera ett sätt för ett objekt att, oberoende av kännedom om andra objekt, kunna distribuera meddelanden till dessa. *Observer* enkelriktar ett ett-till-många beroende.

Struktur



Användning

Alla tillfällen när ett objekt (*ConcreteSubject*) har andra (*ConcreteObserver*) som är beroende av objektet. Man vill och behöver inte specialkoda för att direkt ta hand om de beroende objekten.

Liknande mönster

Mediator beskriver kommunikation mellan många som är beroende av varandra, *Observer* bara ett envägsberoende.

För- och nackdelar

Den största fördelen att en ny *Observer*-klass kan läggas till utan att vare sig subjektet eller andra observatörer påverkas alls. Kopplingen från subjekt till observatörer är endast abstrakt.

Eftersom en observatör endast implementerar ett *interface*, kan en observatör registrera sig hos fler subjekt.

Om subjektet tillstånd ofta ändrar sig kan man få prestandeproblem om det finns många observatörer som samtidigt vill ta reda på om och vad de behöver uppdatera sig med. Eftersom de inte heller känner till varandra är det risk att en observatörs uppdatering av subjektet leder till många ytterligare okontrollerade observatörsoperationer.

Varianter

I stället för att *notify*- och *update*-metoderna saknar parametrar kan *ConcreteSubject* skicka med information om vad som hände. Då kan varje observatör genast avgöra om det är något att bry sig om eller strunta i, och slippa onödigt hämtande av statusinfo från *ConcreteSubject*.

Man kan också tänka sig att en observatör vid registreringen kan prenumerera endast på vissa typer av händelser, för att undvika onödiga anrop av *update*.

Java har implementerat *Observer* i sitt standardbibliotek, men kallar *Subject* för det passande namnet *Observable*. Där har också *update*-metoden två parametrar: en referens till *Observable*-objektet skickas alltid med samt ett godtyckligt info-*Object* som via *notify*-metoden säger något om vad som hänt. (Java har också döpt om *notify* till *notifyObservers*.)

C#

event **och** *delegate*

I C# har man byggt in mönstret i språket med en egen mekanism. Vi tar en kort variant av exemplet med *TextDocument* ovan.

Först måste vi deklarera en händelse, *event*, som representerar det som hänt, i vårt fall en ändring i dokumentets text, samt den kod som genererar denna händelse vid ändring:


```
public class TextDocument
{
  //...
  public event EventHandler TextChange;
  //...
  public void OnTextChange(EventArgs e)
  {
                                                    // motsvarar notify
    if (TextChange != null) TextChange(this, e);    // generera händelsen
  }
}
```

I *Observer*-klasserna definieras *delegate*-funktionerna och i konstruktorn kopplas de att lyssna till *TextChange* med operatorn +=.

```
public class SpellingChecker
{
  //...
  void TextDocument_TextChange(object sender, EventArgs e)
  {
    // gör vad som skall göras vid ändrad text i dokumentet
  }
  //...
  public SpellingChecker()
  {
    //...
    theTextDocument.TextChange +=
              new EventHandler(TextDocument_TextChange);
    //...
  }
}
```

Vid varje anrop till *OnTextChange* kommer nu samtliga lyssnare anslutna med operatorn += att får sina *delegate*-funktioner anropade. Både *EventHandler* och *EventArgs* är definierade i .NET-paketet *System*.

På samma sätt som i Java skickas alltid avsändarreferens tillsammans med ett »tips« om vad som hänt, det senare i *EventArgs*-parametern som naturligtvis kan subklassas med nya varianter.

C#