Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 8 – DP Observer och Template Method

Designmönster med C++

Syfte:	Introduktion av två ’behavioural’ designmönster: DP Observer och DP Template Method
Att läsa:	Design Patterns Explained kap 18 sid 315 – 329 och kap 19 sid 331 – 343

Kapitel 18 – ”The Observer Pattern”

I GoF’s ’Design Patterns’ delas de 23 designmönstren in i tre kategorier:

- ’Creational’ – skapar/instansierar objekt, t.ex. Abstract Factory och Singleton.

- ’Structural’ – kombinerar klasser och interface för att skapa ny funktionalitet. Detta görs flexibelt så att sammansättningarna kan ändras under runtime, t.ex. Decorator, Adapter och Bridge.

- ’Behavioural’ – hanterar algoritmer och fördelning av ansvar mellan olika objekt. Dessa designmönster beskriver även kommunikationsmönster mellan inblandade objekt. Underlättar variationer av beteende. Exempel är Strategy och Command, andra exempel är Observer och Template Method som behandlas i denna lektion.

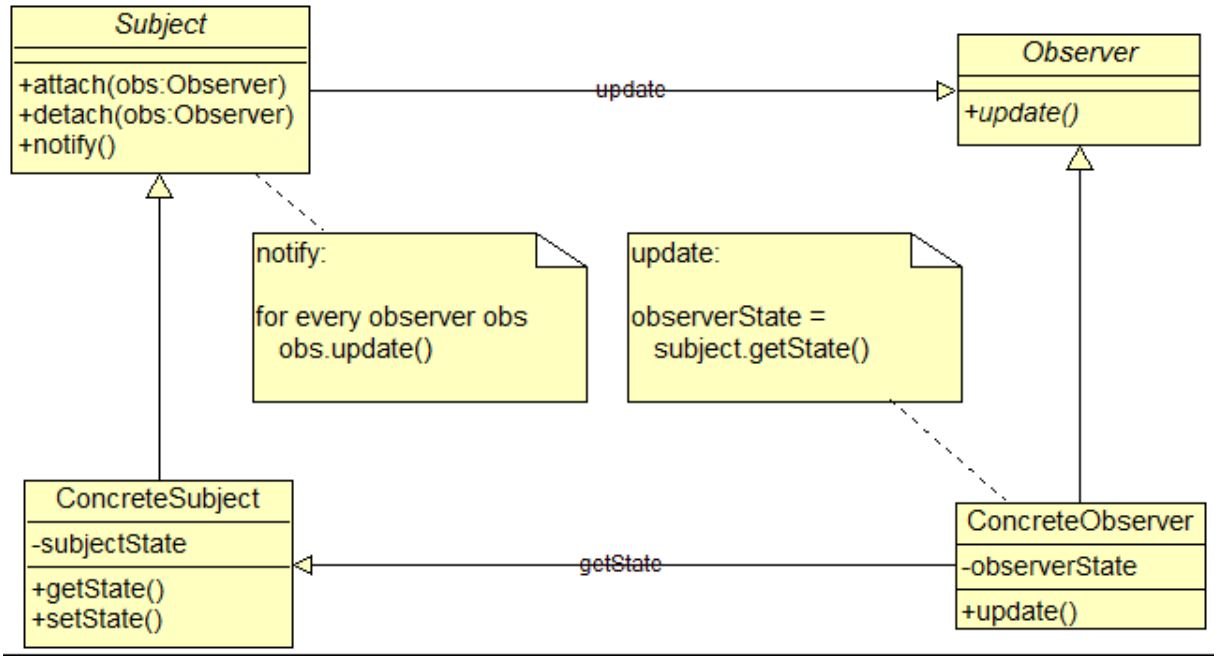
Kursboken argumenterar för en fjärde kategori, ’decoupling patterns’, för DP vars främsta syfte är att minska kopplingen (och därigenom beroendet) mellan objekt för att öka skalbarhet och flexibilitet. Det skulle i så fall närmast bli en undergrupp till ’Behavioural patterns’. Ett exempel på ett DP som skulle hamna i denna kategori är Observer.

DP Observer

Syfte:
” *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.* ”
(GoF)

I många sammanhang har man en uppsättning objekt (observers) som ska uppdateras (av ett ’subject’) till följd av någon händelse. Antalet och typ av objekt som ska uppdateras kan variera dynamiskt. DP Observer hanterar detta.

Generell struktur:



Pilarna med fylld spets i klassdiagrammet symboliserar ’call’, dvs metodanrop.

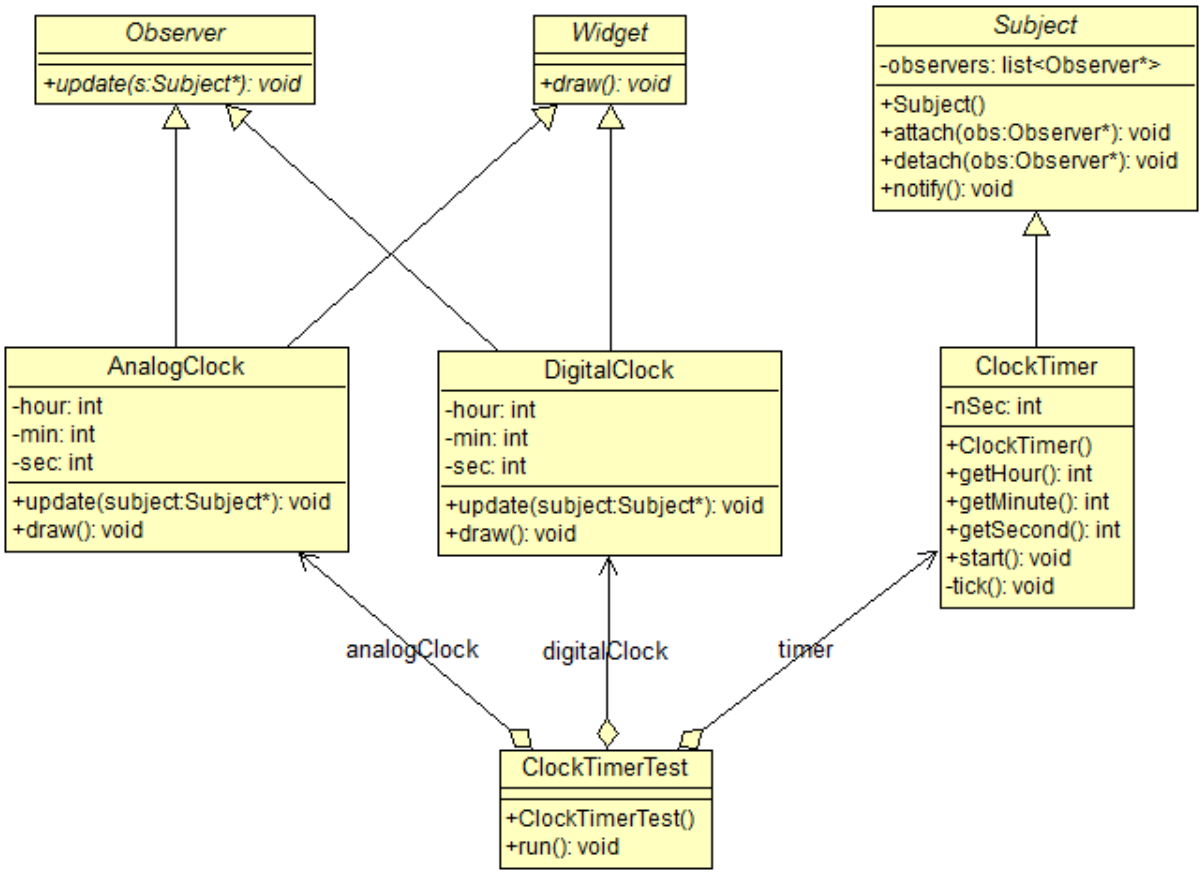
Exempel ClockTimer (exClockTimer)

- En ClockTimer (subject) startas och uppdateras varje sekund.Visningen av förfluten tid ska ske på olika displayer (Observers), en med analog och en annan med digital visning. De ska dock alltid visa samma tid. Nya displayer ska kunna anslutas/tas bort dynamiskt.

- Subject implementerar metoderna attach och detach för att lägga till/ta bort Observers. Metoden notify används i ClockTimer för att trigga en uppdatering av anslutna Observers. ClockTimer är en konkret Subject-klass. Den innehåller metoder som Observers använder för att fråga aktuell tid.

- Alla Observers implementerar metoden update som uppdaterar tillståndet genom att fråga ClockTimer. Som argument till update skickar Clocktimer sin this-pekare som Observers använder för att fråga efter det nya tillståndet. AnalogClock och DigitalClock är två konkreta Observer-klasser.

Strukturen visas i följande klassdiagram.

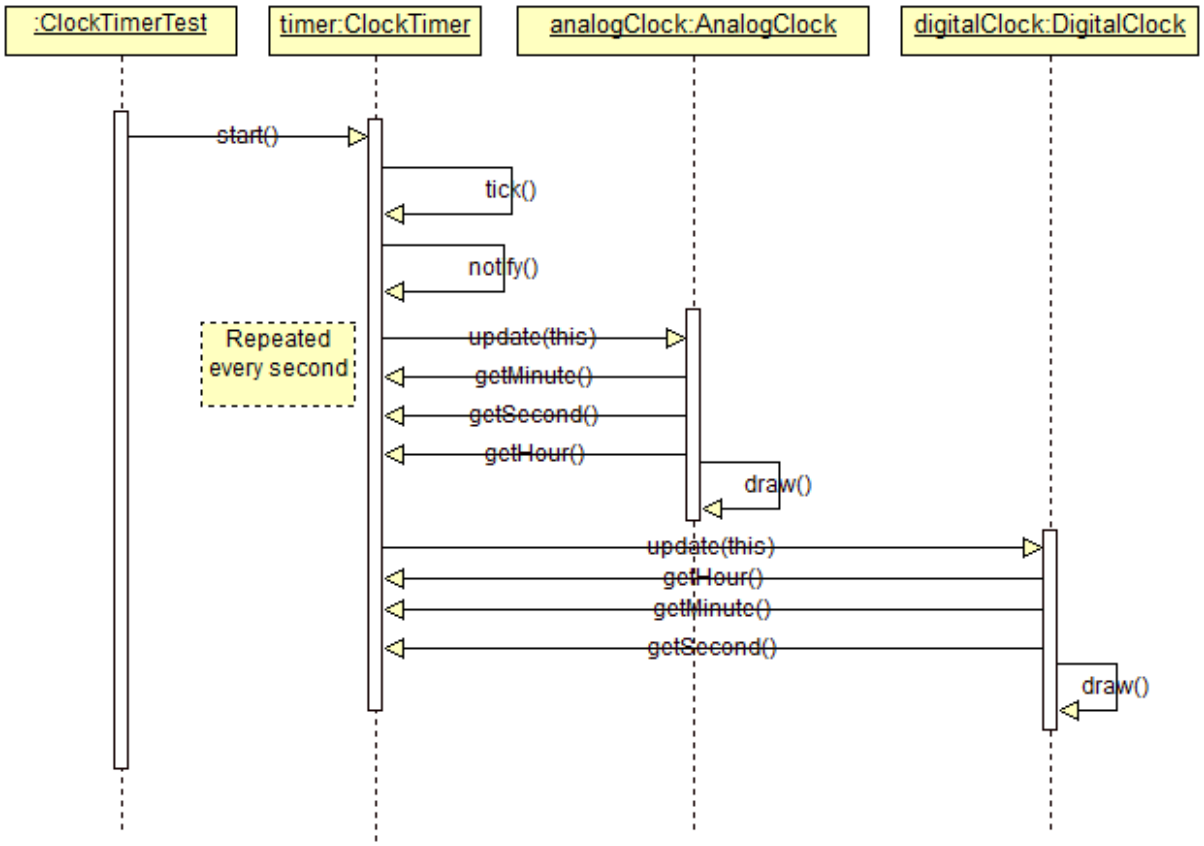


Protokollet mellan Subject och Observer kan kallas *Observer-pull*:

- Subject meddelar sina Observers att en uppdatering av tillståndet har skett genom meddelandet `update(this)`.

- Varje Observer hämtar information om det nya tillståndet genom att använda interfacet i ClockTimer: getHour( ) osv. för att sedan rita om sitt aktuella fönster genom draw( ) som implementerar Widget::draw( ).

Interaktionerna kan beskrivas med följande sekvensdiagram.



Observer-pull-modellen kräver att Subject implementerar alla metoder som krävs för att en Observer ska kunna uppdatera sig genom att fråga om det nya tillståndet.

En annan modell är *Subject-push* där Subject direkt uppdaterar sina aktuella Observers via utökade update-metoder.

- Fördelar
  - Enklare protokoll, kommunikationsmönstret förenklas.

- Nackdelar
  - Kräver ett större interface hos Observer
  - Subject måste ha större kännedom om Observers interface för att kunna uppdatera selektivt, notify måste implementeras i de konkreta Subject-klasserna.

Exempel ChatServer (exChatServer)

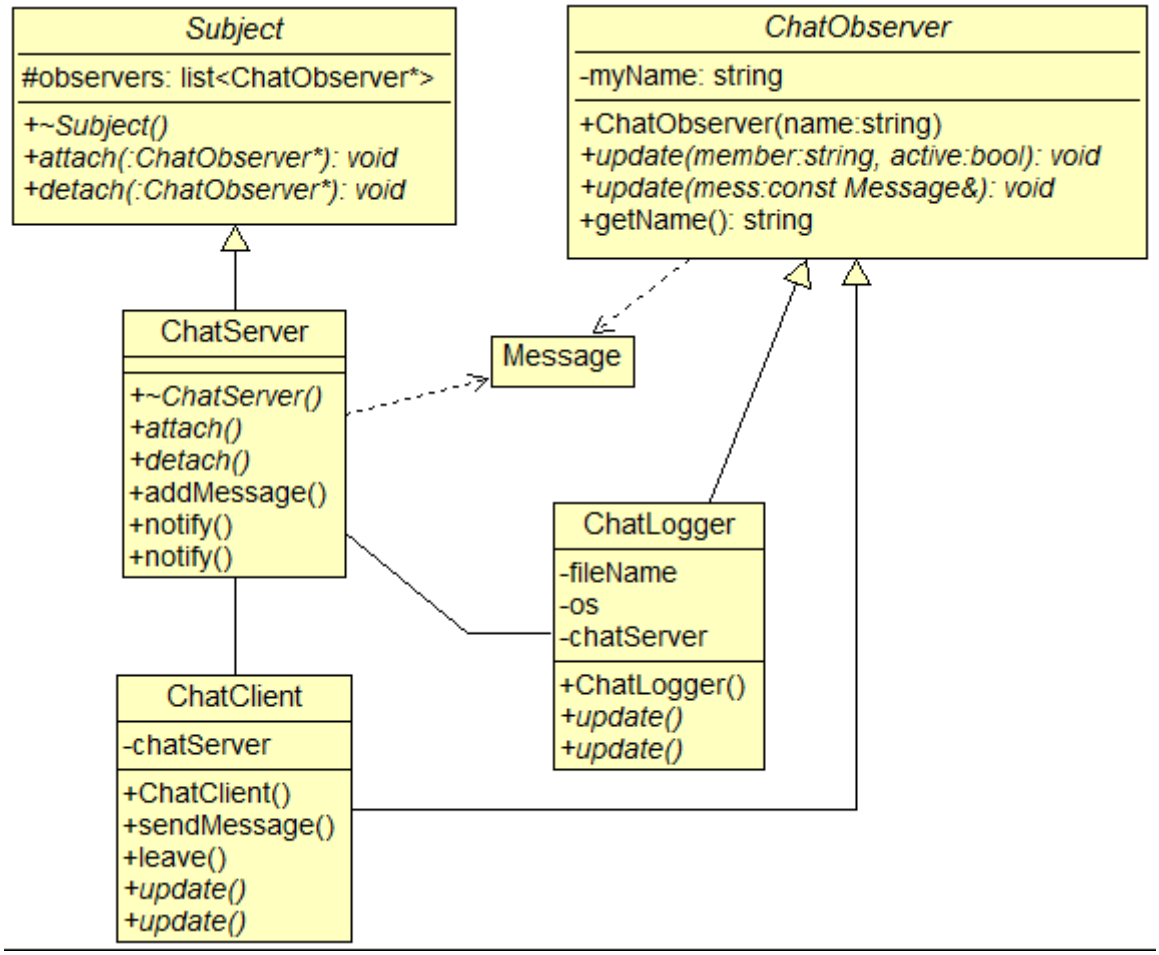
- ChatServer (Subject) ska hantera en många-till-många-konversation genom att ta emot ett meddelande från en klient och distribuera till andra

anslutna klienter.

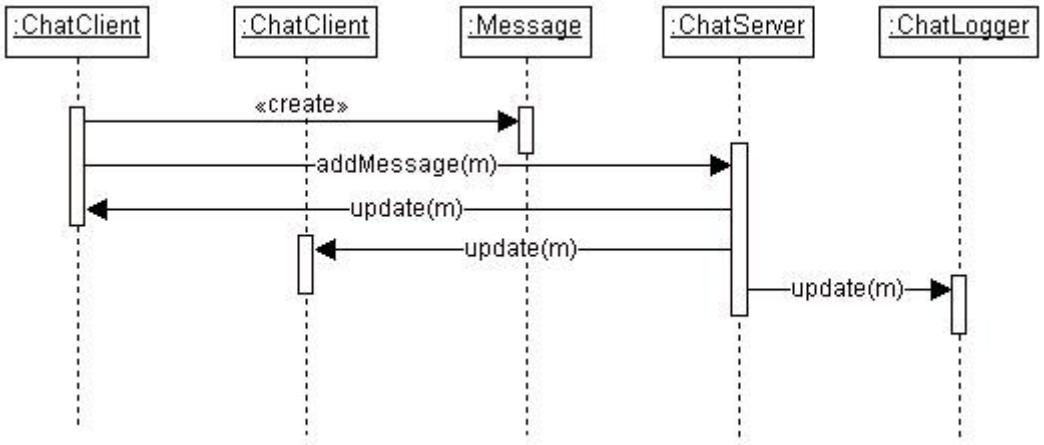
- Varje ansluten klient representeras av ett ChatClient-objekt som implementerar interfacet ChatObserver.

- Uppdatering av klienter sker genom subject-push vilket kräver ett utökat interface för update-operationer. Uppdatering sker direkt via någon av de två notify-operationerna.

- Alla meddelanden skrivs till en loggfil genom en instans av klassen ChatLogger som också är en ChatObserver.



Interaktionen när en ChatClient skickar ett meddelande visas i följande sekvensdiagram.

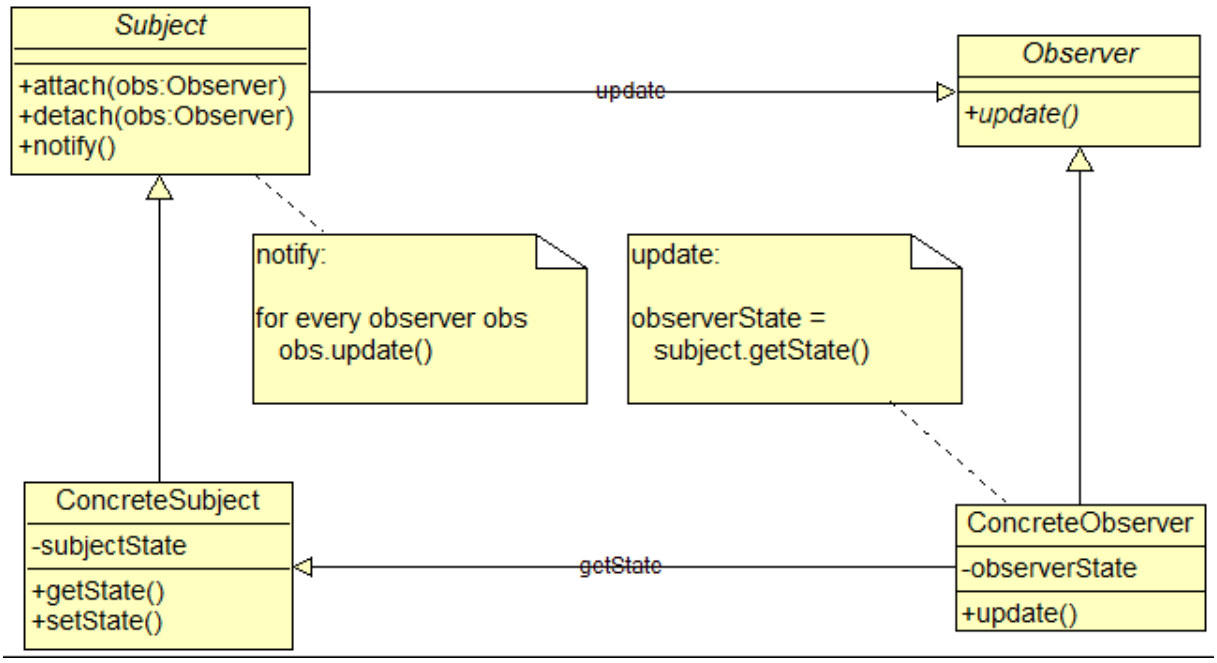


Kommunikationsprotokollet förenklas på bekostnad av ett utökat interface för update-operationer.

Sammanfattning

DP **Observer** (’Publisher/Subscriber’)

- Struktur



- Syfte
  - Definera ett En-till-Många-beroende mellan objekt så att när ett objekt ändrar tillstånd så kommer alla beroende objekt att uppdateras automatiskt.

- Problem
  - Du ska notifiera en varierande uppsättning objekt att en händelse har inträffat.

- Lösning
  - Observers överlåter till ett centralt objekt, Subject, att övervaka relevanta händelser.

- Deltagare
  - En konkret Subject känner till sina Observers eftersom de har registrerat sig. Subject måste notifiera sina Observers när en relevant händelse har inträffat. Observers är i regel ansvariga för att själv hämta information från Subject när de är notifierade (Observer-pull). Subject kan ibland direkt uppdatera sina Observers (Subject-push).

- Konsekvenser
  - Om olika Observers är intresserade av olika händelser kan Subject notifiera Observers i onödan. Subject kan göras ansvariga för att filtrera vilka Observers som ska notifieras genom att använda ett Strategy DP för att testa om en viss Observer ska notifieras. En Observer ger då Subject ett lämpligt Strategy-objekt vid registreringen.

- Implementation
  - Observers registrerar sig hos ett Subject-objekt som i sin tur notifierar alla Observers vid en viss händelse. Händelsen kan vara extern eller så kan Subject själv ta initiativ till en notifiering. Observers ansvarar i regel för att hämta relevant information från Subject. Om uppsättningen Observers är heterogen kan Adapter DP användas för att implementera Observer-interfacet för alla Observers. Strategy DP kan användas för att låta Subject avgöra vilka Observers som ska notifieras för en viss händelse.

- Använd Observer DP
  - när en abstraktion har två aspekter, den ena beroende av den andra. Genom inkapsling i separata objekt kan du variera dem och återanvända dem oberoende av varandra.
  - när en förändring i ett objekt kräver förändring av andra objekt och du inte kan veta exakt vilka objekt som ska uppdateras.
  - när ett objekt ska kunna notifiera andra objekt utan att göra antaganden om vilka dessa objekt är, dvs när du vill undvika stark

koppling (tight coupling) mellan ett objekt och andra beroende objekt.

I Observer DP kan man hitta följande principer för objektorientering:

- Ett objekt tar ansvar för sig själv
  - Olika Observers hämtar själva den information de behöver från sitt Subject och hanterar den på lämpligt sätt.
- Abstrakta klasser
  - Klassen Observer representerar konceptet för objekt som behöver notifieras om en viss händelse. Den ger ett gemensamt interface för Subject att utföra notifieringen-
- Polymorfisk inkapsling
  - Subject vet inte vilken Observer den kommunicerar med. Nya konkreta Observer-typer kan läggas till utan att Subject behöver ändras.

Kapitel 19 – 'The Template Method Pattern"

DP Template Method

Syfte:
" *Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.* " (GoF)

Exempel – En databasfråga (exQueryTemplate)

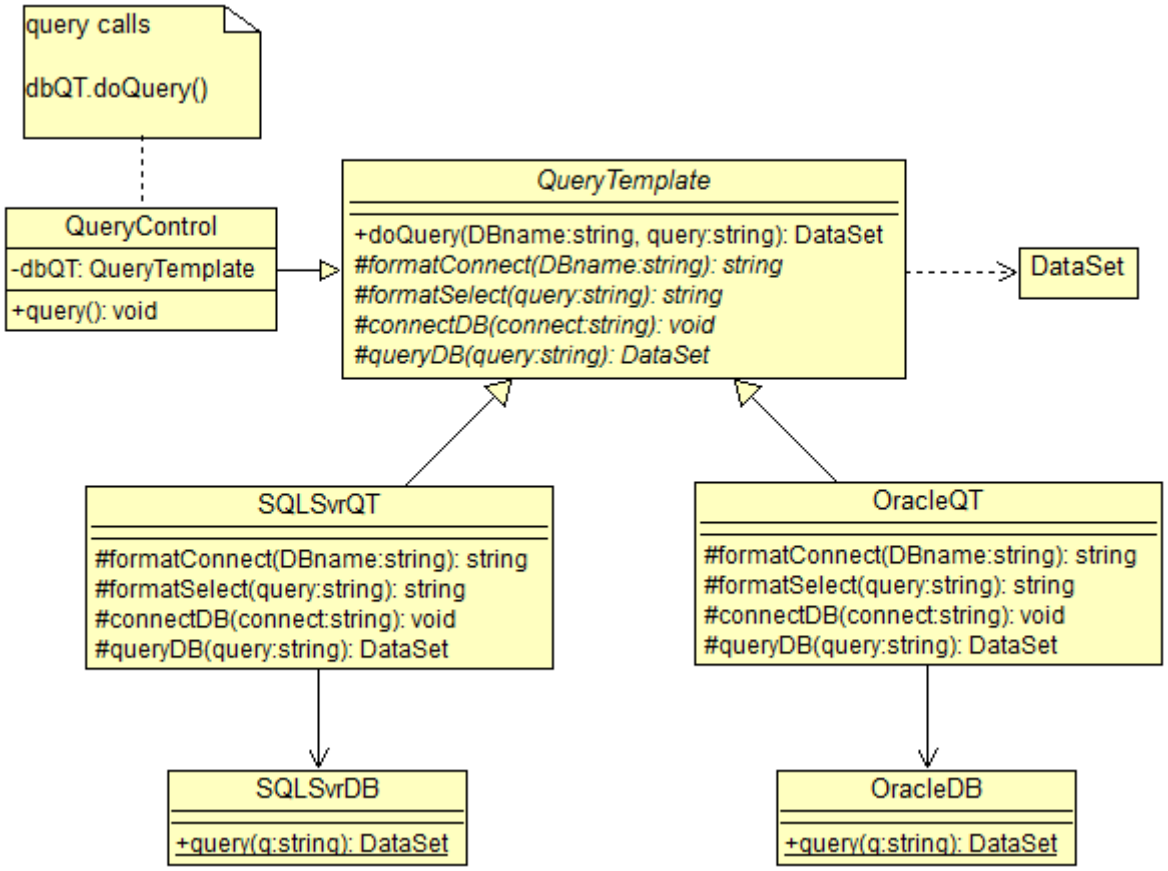
Att ställa en fråga till en databas innefattar stegen
1. Ansluta till och öppna databasen
2. Formulera och ställ frågan
3. Ta emot ett returnerat dataset

För olika databaser t.ex. Oracle och SQL Server kan syntaxen för de olika stegen skilja sig åt. De utför i princip utför samma sak i varje steg (samma semantik) medan syntaxen varierar. Det handlar alltså om *samma konceptuella process*. Med Template Method kan vi kapsla in de gemensamma konceptuella stegen i en abstrakt klass medan deriverade klasser kapslar in det som varierar i implementationen av stegen.

Abstrakta klassen QueryTemplate får innehålla en publik metod doQuery(DBname, query) som hanterar en fråga.

doQuery *delegerar de konceptuella stegen* till andra metoder
- formatera en sträng för att ansluta till DB ➔ formatConnect
- ansluta och öppna DB ➔ connectDB
- formatera en sträng för frågan ➔ formatSelect
- ställa frågan till DB och hämta ett dataset ➔ queryDB

Eftersom formatConnect, connectDB, formatSelect och queryDB kapslar in variationerna mellan olika databaser måste dessa omdefinieras i konkreta subklasser till QueryTemplate. Eftersom de endast ska kunna anropas av doQuery får de accessnivån protected.



Klassen QueryControl har en referens (av typen QueryTemplate) till någon konkret xxxQT-klass.

- Anropet till doQuery( ) innebär att de steg som definieras av de övriga skyddade metoderna i QueryTemplate genomförs av implementationerna i rätt klass.
- Ännu en databas kan läggas till genom att derivera en ny sub-klass från QueryTemplate och implementera formatConnect, connectDB formatSelect och queryDB på lämpligt sätt.

’Hook’-operationer

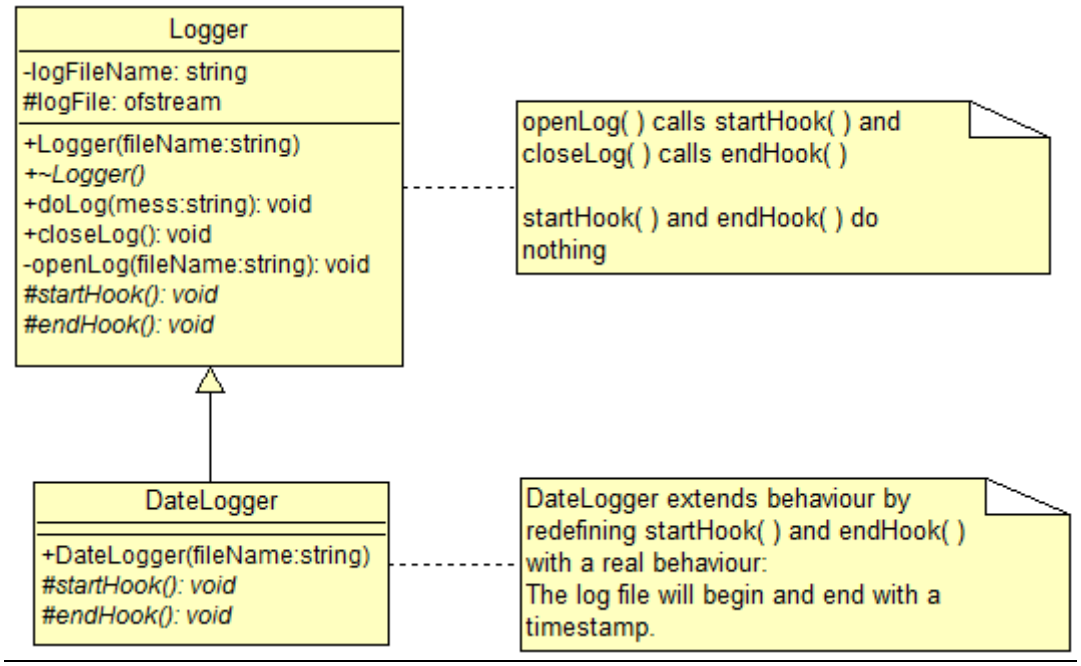
Det är viktigt att skilja på

- abstrakta operationer som **måste definieras** i sub-klasser
- operationer som **kan omdefinieras** i sub-klasserna
- operationer som **kan utvidgas** (extend) i sub-klasserna.

Template Method kan användas för att skapa *hook-operationer* där föräldraklassen har kontroll över hur sub-klasser kan utvidga beteendet hos en operation.

Exempel – Logger (exLogger)

Klassen Logger skriver text till en loggfil.



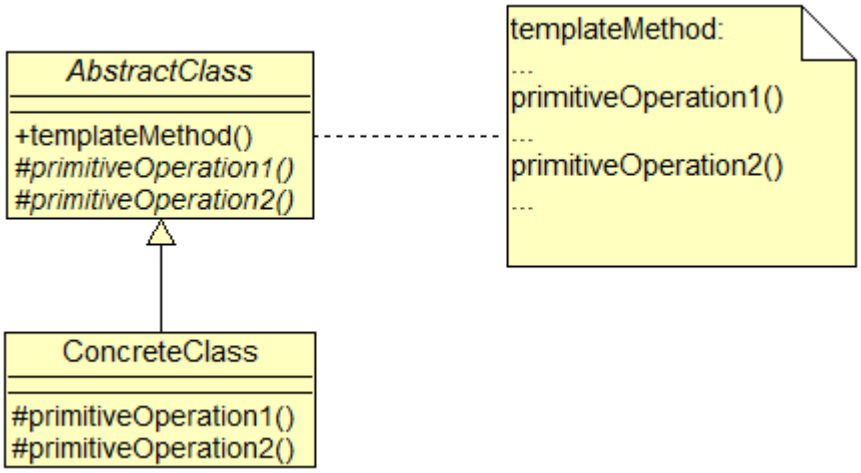
- doLog( ) anropar openLog( ) en gång vid första anropet. Från openLog( ) anropas startHook( ).
- closeLog( ) anropar endHook( ) innan filen stängs.

Dessa metoder är implementerade som tomma metoder i Logger. Detta ger sub-klasser möjligheten att omdefiniera metoderna för att utvidga beteendet i samband med loggfilens öppning och stängning. Detta genomförs i den deriverade klassen DateLogger där startHook( ) och endHook( ) skriver tidsstämplar till log-filen.

Vi får alltså ett utvidgat beteende men var/när detta kan ske är bestämt av basklassen Logger, alltså en tillämpning av DP Template Method.

Sammanfattning
DP **Template Method**

- Struktur



- Syfte
  - Definiera strukturen av en algoritm i en operation men delegera ett antal steg i implementationen till sub-klasser. Template Method låter sub-klasser omdefiniera vissa steg i en algoritm utan att förändra algoritmens struktur.

- Problem
  - En algoritm har på en abstrakt nivå ett antal steg som ska genomföras men de olika stegen har olika konkreta implementationer beroende på sammanhanget.

- Lösning
  - Template Method tillåter att implementationen av de olika stegen kan variera utan att algoritmens grundläggande struktur ändras

- Deltagare

  - *AbstractClass*
    Definierar abstrakta grundläggande metoder som konkreta sub-klasser definierar för att implementera de olika stegen i en algoritm. Implementerar en 'mall-metod' som definierar skelettet till algoritmen. Mall-metoden anropar de grundläggande metoderna och eventuellt andra metoder.

  - *ConcreteClass*
    Implementerar de grundläggande metoderna för att utföra de sub-klass-specifika stegen i algoritmen.

- Konsekvenser
  - Exempel på Hollywood-principen: "Don't call us, we'll call you": föräldraklassen anropar metoder som implementerats i sub-klasser och inte tvärtom.

  - Template method är en bra teknik för återanvändning av kod, t.ex. för att åstadkomma likartat beteende hos klasser i ett klassbibliotek. En template method binder ihop algoritmsteg som omdefinierats i sub-klasser och garanterar att alla steg finns med.

  Från en Template method anropas bl.a.
  - konkreta operationer antingen i ConcreteClass eller i en klient-klass

  - konkreta AbstractClass operationer (som är generellt användbara i sub-klasserna)

  - factory methods (se kommande lektioner)

  - hook-operationer som ger ett default-beteende som sub-klasser kan utvidga vid behov. En hook-operation utför default ingenting.

- Implementation
  - Skapa en abstrakt klass som implementerar en algoritm genom att använda abstrakta operationer. Dessa operationer måste implementeras i sub-klasser för att utföra varje steg i algoritmen. Om stegen varierar oberoende av varandra kan de implementeras med Strategy DP.

- Använd Template Method
  - för att implementera skelettet till en algoritm och överlåta till sub-klasser att implementera det beteende som kan variera

  - när ett gemensamt beteende hos sub-klasser ska lokaliseras till ett objekt för att undvika duplicering av kod enligt 'Once-and-only-once-rule’.

  - för att kontrollera utvidgningar i sub-klasser genom att införa hook-
  - operationer på speciella ställen i en template method.