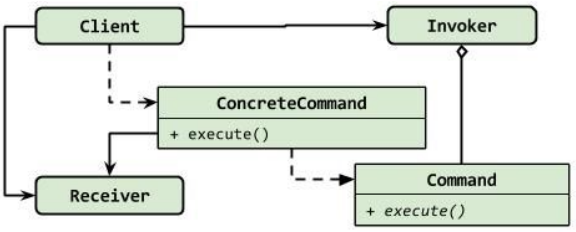
COMMAND *behavioral*

"Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations"



I många situationer kan man dra fördel av en lös koppling mellan klienten som utfärdar en begäran (request), **command issuer**, och objektet som vet hur anropet skall utföras, **command receiver**. Detta kan åstadkommas genom att paketera anropet i ett objekt (**Command Objekt**). Ett sådant objekt kan skickas som parameter, sparas, köas, loggas osv.

SYFTE
Att kapsla in ett kommando i ett objekt så att klienter kan parametriseras med olika kommandon. Kommandon kan då loggas, köas och schemaläggas. Dessutom erhålls stöd för **undo/redo** av kommandon.

PROBLEM
Anrop / kommandon måste genomföras utan exakt kännedom om mottagaren eller vilken operation som skall utföras.

LÖSNING
Definiera ett gemensamt interface för alla konkreta typer av kommandon så att **requests** kan kapslas in i objekt.

DELTAGARE
- **Command** definierar ett interface för att utföra en operation.
- **ConcreteCommand** implementerar **execute**-funktionen genom att tillämpa en operation på målobjektet (**Receiver**).
- **Invoker** anropar **Command**-objektets execute-funktion.
- **Receiver** är det objekt som är målet för kommandot och som kan utföra den önskade operationen.

KONSEKVENSER
- **DP Command** ger en lös koppling mellan **Invoker** och **Receiver**. **Invoker** behöver inte veta exakt typ för **Receiver**.
- Kommandon kan behandlas som alla andra typer av objekt.
- Flera kommandon kan samlas i sammansatta kommandon, **MacroCommand**. Detta bygger på **DP Composite**.
- Nya kommandon kan lätt läggas till eftersom inga existerande klasser behöver ändras.

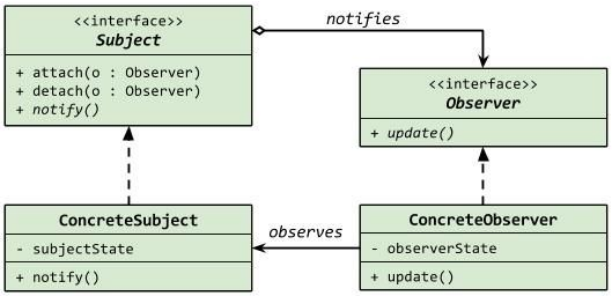
IMPLEMENTATION
- Ett konkret kommando-objekt kan ha olika grader av attribut / intelligens beroende på användningen.
- För att implementera **undo/redo** måste kommandoobjekt sparas och ha en **unExecute**-funktion. Attribut i objekten måste eventuellt spara tillståndsinformation så att **undo** kan genomföras. Även **Receivers** måste ibland kunna svara på frågor på aktuellt tillstånd.

ANVÄND NÄR
- Parametrisera objekt med en **action** som skall utföras. I ett prcedurellt språk kan det uttryckas med en **callback**-funktion. **Command** är den objektorienterade motsvarigheten till **callback**-funktioner.
- Specificera, kö, logga och utföra kommandon vid olika tider. Livstiden för ett **Command** kan vara oberoende av det ursprungliga kommandot. Om mottagaren (**Receiver**) kan representeras oberonde av den aktuella adressrymden kan ett **Command** skickas tilll en annan process och utföras där.
- Implementera **undo/redo** av kommandon.
- Logga kommandon så att de kan återexekveras vid ett senare tillfälle, t.ex. efter en systemkrasch. Detta kan göras genom att utvidga interfacet för **Command** med operationer som **load** och **store**. Lagrade kommandon kan läs läsas från disk och återexekveras med **execute**.
- Implementera transaktioner som innefattar flera ändringar av data och som skall genomföras i sin helhet eller inte alls. Alla transaktioner kan ges ett gemensamt interface och nya typer av transaktioner kan enkelt läggas till.

OBSERVER *behavioral*

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

I många sammanhang har man en uppsättning objekt (**Observers**) som skall uppdateras (ev ett **Subject**) till följd av någon händelse. Antalet och typ av objekt som skall uppdateras kan variera dynamiskt. **DP Observer** hanterar detta.



SYFTE
Definiera ett **En-till-Många**-beroende mellan objekt så att när ett objekt ändrar tillstånd så kommer alla beroende objekt att uppdateras automatiskt.

PROBLEM
Du skall 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 ansvarig för att filtrera vilka **Observers** som skall notifieras genom att använda ett **Strategy DP** för att testa om en viss **Observer** skall notifieras. En **Observer** ger då **Subject** ett lämpligt **Strategy**-objekt vid registreringen.

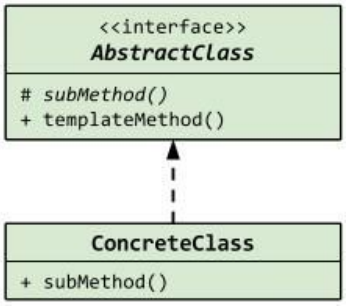
IMPLEMENTATION
Observers registrerar sig hos ett **Subject**-objekt som i sin tur notifierar alla **Observers** vid en viss händelse. Händelsen kan vara extern eller så kan **Subject** själv ta initiativ till en notifiering. **Observers** ansvarar i regel för att hämta relevant information från **Subject**. 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 skall notifieras för en viss händelse.

ANVÄND NÄR
- 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 skall uppdateras.
- När ett objekt skall kunna notifiera andra objekt utan att göra antaganden om vilka dess objekt är, dvs när du vill undvika stark koppling (**tight coupling**) mellan ett objekt och andra beroende objekt.

OOP PRINCIPER
- 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.

TEMPLATE METHOD *behavioral*

"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."



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 skall 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.

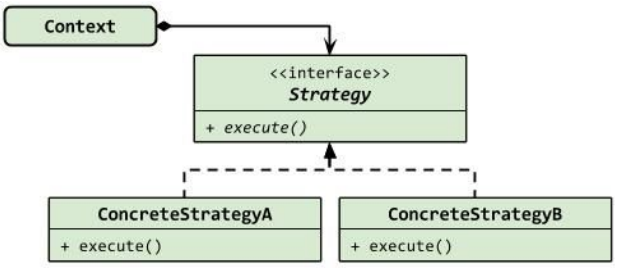
KONSEKVENSER
- Exempel på Hollywood-principen: *"Don't call us, we'll call you"*: föräldraklassen anropar metoder som implementeras i sub-klasser och inte tvärtom.
- **Template Method** är en bra teknik för återanvändning av kod, t.ex. för att åstadkomma likartat beteende hos klasser i ett klassbibliotek. En **Template Methode** binder ihop algoritmsteg som omdefinieras 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-class.
  - Konkreta **AbstractClass** operationer (som är generellt användbara i sub-klasserna).
  - **Factory Methods**
  - **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 NÄR
- 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 skall 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**.

STRATEGY *behavioral*

"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."



Strategy bygger bl.a. på att

- Objekt har ansvar för vissa uppgifter, olika specifika implementationer av dessa uppgifter kan göras genom polymorfism.
- Det finns ett behov av att hantera olika implementationer av vad som konceptuellt sett är samma algoritm.
- God designpraxis är att separera olika beteenden inom ett problemområde från varandra. Den klass som är ansvarig för ett visst beteende kan då bytas ut mot en annan utan att påverka övriga beteenden.

SYFTE

Tillåter dig att använda olika regler eller algoritmer beroende på sammanhanget.

PROBLEM

Valet av algoritm beror på klienten eller på data som skall bearbetas. Om det bara finns en algoritm som alltid skall användas behöver inte **Strategy** användas.

LÖSNING

Separera valet av algoritm från implementationerna. **Strategy** erbjuder ett sätt att konfigurera en klass med ett av flera utbytbara beteenden.

DELTAGARE

- **Strategy** specificerar interfacet till algoritmerna.
- **ConcreteStrategies** implementerar de olika algoritmerna.
- **Context** använder en **ConcreteStrategy** genom en referens till **Strategy**. **Context** vidarebefodrar en begäran från sin **Client** till sin **Strategy**.

KONSEKVENSER

- Switch- och villkorssatser kan elimineras.
- Alla algoritmer måste ha samma interface.

IMPLEMENTATION

Interfacen för **Strategy** och **Context** måste ge en **ConcreteStrategy** ett sätt att accessera data från **Context** och tvärtom.

1. **Context** kan skicka data som parametrar till **Strategy**, dvs ta data till **Strategy**. Detta ger särkoppling mellan **Strategy** och **Context**.
2. **Context** kan skicka en referens till sig själv (**this**) som parameter till **Strategy** och låta **Strategy** själv hämta de data som behövs.
3. Klassmallar (**generics**) kan användas för att konfigurera en klass med en **Strategy**. Tekniken kan bara användas om
    a. lämplig **Strategy** kan bestämmas vid compile-time och
    b. den inte behöver ändras under run-time.

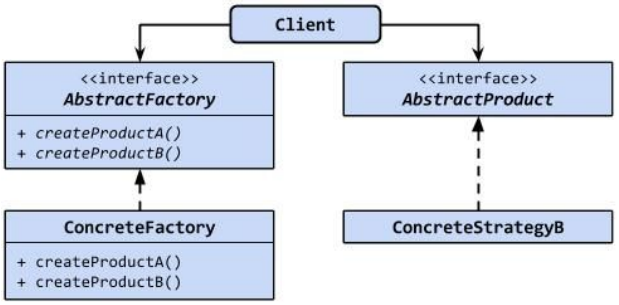
ANVÄND NÄR

- Många relaterade klasser endast skiljer sig åt i sitt beteende.
- Du behöver olika varianter av en algoritm och dessa varianter är implementerade i en klasshierarki.
- En algoritm använder data som en klient inte skall känna till. **Strategy** döljer implementeringsdetaljer!
- En klass kan ha många beteenden och dessa uppträder i multipla flervalssatser. Ta bort grenarna och låt de bli egna **Strategy**-klasser.

ABSTRACT FACTORY *creational*

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes."

Innebörden är att en familj av abstrakta klasser kan implementeras av olika konkreta familjer av klasser. T.ex. har olika GUIer olika implementationer av **Widgets**. Till varje GUI hör dock en viss uppsättning **konkreta klasser** som implementerar olika **Widgets** på ett specifikt sätt.



SYFTE
Erbjuda ett interface för att skapa familjer av besläktade eller beroende objekt utan att specificera deras konkreta klasser.

PROBLEM
Familjer av besläktade objekt skall instansieras.

LÖSNING
Abstract Factory koordinerar instansiering av objekt. Erbjuder ett sätt att flytta regler för hur instansiering skall utföras från klienterna som skall använda objekten.

DELTAGARE
Abstract Factory definierar hur varje mdelem i en familj av klasser skapas. Vanligen skapas en familj genom att ha sin egen **ConcreteFactory**.

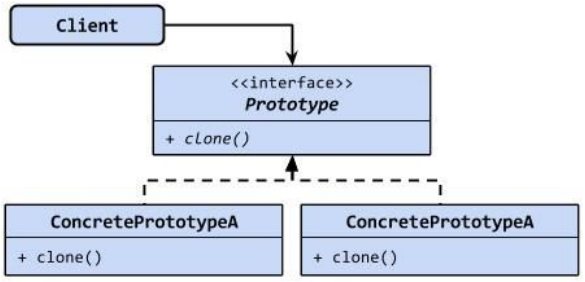
KONSEKVENSER
- Isolerar klienter från konkreta klasser. Klienter använder instanserna genom deras abstrakta interface.
- Konkreta produktnamn isoleras till implementationen av en **ConcreteFactory**, namnen används inte i klientkoden.
- Underlättar utbyte av en hel produktfamilj.
- Upprätthåller konsistens mellan de använda objekten genom att de endast skapas familjevis.
- Svårt att tillföra nya konkreta produkter i efterhand, interfacet till **AbstractFactory** och dess underklasser måste i så fall ändras.

IMPLEMENTATION
- Definiera en abstrakt klass som specificerar metoder för att skapa de olika objekten.
- En konkret klass definieras för varje familj av konkreta produkter som skall skapas.
- Eftersom en applikation endast behöver ett **Factory**-objekt för varje familj så implementeras **Abstract Factory** ofta som en **Singleton**.

ANVÄND NÄR
- En familj av objekt skall användas tillsammans och du vill framtvinga detta.
- Du vill erbjuda ett klassbibliotek med produkter och du endast vill visa interfacen men inte implementationerna.

PROTOTYPE *creational*

Purpose
Create objects based upon a template of an existing object through cloning.

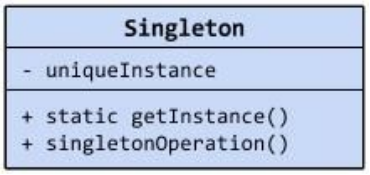


Use When

- Composition, creation, and representation of objects should be decoupled from a system.
- Classes to be created are specified at runtime.
- A limited number of state combinations exist in an object.
- Objects or object structures are required that are identical or closely resemble other existing objects or object structures.
- The initial creation of each object is an expensive operation.

SINGLETON *creational*

"Ensure a class only has one instance, and provide a global point of access to it."



Ibland är det viktigt att det finns, och bara kan finnas, EN instans av en klass i en applikation. Lösningen är att klassen själv håller kontroll på sin enda instans genom att ha:

- En icke-public konstruktor.
- En speciell accessmetod som returnerar instansen om den finns, eller skapar den om den inte finns.
- Access-metoden definieras som en statisk metod; → accessbar utan instans.

SYFTE
Att garantera att en klass endast har en instans och att ge en global accesspunkt till denna instans.

PROBLEM
Flera olika objekt behöver använda samma objekt och du vill vara säker på att du bara har en instans.

LÖSNING
Låt **Singelton** ansvara för att endast en instans skapas.

DELTAGARE
Klienter accesserar en **Singelton** enbart genom den statiska metoden **getInstance()**.

KONSEKVENSER

- Kontrollerad access till den enda instansen.
- Klienter behöver inte befatta sig med frågan om det finns någon instans av **Singelton** eller inte.
- Tillåter ett variabelt antal instanser. Det är lätt att tillåta fler än en instans om man så vill!

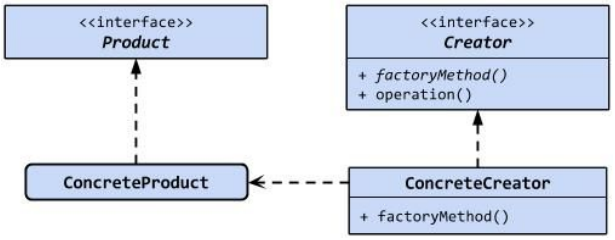
ANVÄND NÄR

- Det får finnas bara en instans av en klass och den skall vara accessbar för klienter via en känd access-punkt.
- Den enda instansen skall vara möjlig att sub-klassa och klienter skall kunna använda den sub-klassade instansen utan att modifiera sin kod.

FACTORY METHOD *creational*

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."

Används ofta i samband med framworks. Framworks motsvarar en abstrakt nivå som inte kan bestämma exakt vilka konkreta objekt som skall skapas. Superklassen vet att objekt måste skapas men beslutet om exakt typ överlåts till sub-klasserna.



SYFTE
Att definiera ett interface för att skapa ett objekt men låta sub-klasser bestämma vilken klass som skall instansieras. **Factory Method** skjuter upp instansieringen och överlåter ansvaret till sub-klasser.

PROBLEM
En klass behöver instansiera en klass från en annan hierarki men vet inte exakt vilken.

LÖSNING
En deriverad klass bestämmer vilken klass som skall instansieras och hur.

DELTAGARE
Creator är interfacet som definierar **Factory Method**. **Product** är interfacet för den typ av objekt som **Factory Method** skapar.

KONSEKVENSER
Klienter måste sub-klassa (eller implementera) **Creator** för att skapa en viss **ConcreteProduct**.

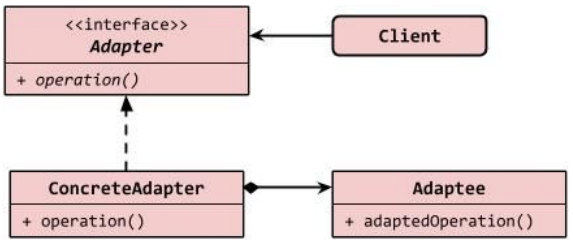
IMPLEMENTATION
Använd en eller flera **Factory Methods** i abstrakt **Creator**-klass. **Factory Method** i en **ConcreteCreator** instansierar en **ConcreteProduct**. I C++ kan **templates** användas för att minska behovet av sub-klassning.

ANVÄND NÄR
- En klass inte kan förutse klassen för det objekt den skall skapa.
- En klass vill att dess sub-klasser skall specificera vilka objekt som skall skapas.

ADAPTER *structural*

"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

Innebörden är alltså att man skapar ett nytt interface till en klass som gör rätt saker men har **fel** interface.



SYFTE
Att anpassa ett interface utanför din kontroll till ett givet interface.

PROBLEM
En klass vi vill utnyttja har önskvärda egenskaper men ett oönskat interface.

LÖSNING
Adapter erbjuder en 'wrapper'-klass med önskat interface.

DELTAGARE
Adaptern anpassar interfacet hos det adapterade objektet (**Adaptee**) så det överensstämmer med interfacet hos **adapterns**. Det tillåter klienter att använda det adapterade objektet som om det vore av **adapterns** typ.

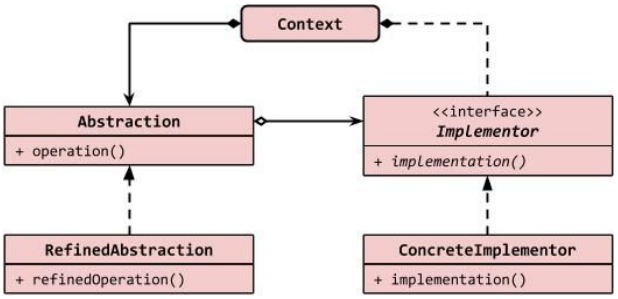
KONSEKVENSER
Adapter DP tillåter existerande klasser att passa in i nya klasstrukturer trots att interfacen inte är direkt kompatibla.

VARIANTER
- **Object Adapter**
  - Låt **Adaptee** vara ett attribut i **ConcreteAdapter**, som utnyttjar **Adaptee's** publika interface.
- **Class Adapter**
  - Bygger på multipelt arv och att man kan reglera accessnivåer i arvet. Passar därför bra i C++, men ej genomförbar i Java.
  - Genom **publikt** arv från **Adapter** får **ConcreteAdapter Adapter's** implementationer och interface.
  - Genom **privat** arv från **Adaptee** får **ConcreteAdapter** tillgång till **Adaptee's** implementation utan att dess interface blir publikt i **ConcreteAdapter**. (sk **implementationsarv**)

ANVÄND NÄR
- Du vill använda befintliga klasser och deras interface inte matchar det du behöver.
- Du vill skapa en återanvändbar klass som samarbetar med orelaterade klasser som kan ha icke kompatibla interface.

BRIDGE *structural*

"Decouple an abstraction from its implementation so that the two can vary independently."



Med **implementationer** menas de objekt som de abstrakta klasserna använder för att implementera sina abstraktioner med och **INTE** de abstrakta klassernas deriverade klasser!

Detta kan vara svårt att ta till sig. **DP Bridge** är ett kraftfullt designmönster som kan kräva lite tid att förstå. Det tillämpar principerna

- *"Find what varies and encapsulate it."*
- *"Favor aggregation over inheritance."*

SYFTE
Att koppla isär en abstraktion från dess implementation så att de kan variera oberoende av varandra.

PROBLEM
Deriverade klasser från en abstrakt klass skall använda flera olika implementationer utan att förorsaka en exponentiell tillväxt av antalet klasser.

LÖSNING
Definiera ett interface som alla implementationer skall använda och låt de deriverade klasserna använda det.

DELTAGARE
- **Abstraction** definierar interfacet för klasserna som skall implementeras.
- **Implementator** definierar interfacet för de specifika implementationsklasserna.
- Klasser deriverade från **Abstraction** använder klasser deriverade från **Implementor** utan att veta vilken **ConcreteImplementor** som används.

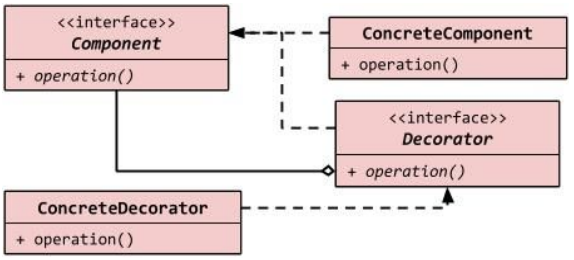
KONSEKVENSER
Minskar kopplingen mellan implementationerna och klasserna som använder dem. Klientobjekten har ingen kunskap om detaljer i implementationerna.

IMPLEMENTATION
- Kapsla in implementationerna i en hierarki med en abstrakt basklass.
- Låt basklassen för abstraktionerna innehålla en referens till implementationernas basklass. I Java kan ett interface användas istället för en abstrakt klass för implementationen.

ANVÄND NÄR
- Du vill undvika en permanent bindning mellan en abstraktion och dess implementation, t.ex. när valet av implementation skall göras under run-time. Både abstraktionerna och deras implementationer skall vara öppna för sub-classing. **Bridge** tillåter dig att kombinera de oika abstraktionerna med olika implementationer och att utvidga (**extend**) dem oberoende av varandra.
- Ändringar i implementationen inte skall påverka klienter så att de måste kompileras om.
- När du vill dela en implementation mellan flera objekt (t.ex. genom referensräkning) och detta faktum skall döljas för klienter.

DECORATOR *structural*

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."



Decorator arbetar genom att skapa en länkad kedja av objekt vars operationer skall tillämpas i en viss ordning.

- Kedjan börjar med en **Decorator** och slutar alltid med det ursprungliga objektet som skall *dekoreras*. Varje objekt i kedjan känner till nästa objekt.
- För att kunna använda en polymorfisk metod, **operation()**, måste alla **Decorators** och det ursprungliga objektet ha en gemensam basklass - **Component**.
- **Klienten** anropar **operation()** för det första objektet i kedjan.

SYFTE
Att dynamiskt lägga till extra ansvar / uppgifter till ett objekt. **Decorator** erbjuder ett flexibelt alternativ till sub-classing för utökad funktionalitet.

PROBLEM
Ett objekt utför de grundläggande uppgifterna du efterfrågar men du vill dessutom lägga till annan funktionalitet före och efter basfunktionerna.

LÖSNING
Erbjuder utökning av funktionalitet utan att använda sub-classing.

DELTAGARE
- **ConcreteComponent** får utökad funktionalitet genom **Decorators**.
- Ibland är det klasser deriverade från **ConcreteComponent** som ger basfunktionaliteten och då är **ConcreteComponent** en abstrakt klass.
- **Component** definierar interfacet för övriga klasser.

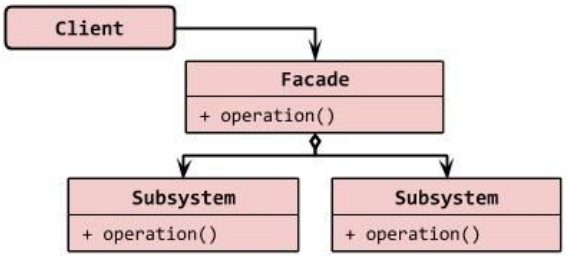
KONSEKVENSER
De utökade funktionerna ges genom små objekt som dynamiskt kan läggas till före eller efter funktionaliteten hos **ConcreteComponent**.

IMPLEMENTATION
- I **Decorator**-klasserna placeras de nya funktionerna före eller efter anrop till efterföljande objekt i kedjan för att uppnå korrekt ordning.
- Skapa en abstrakt klass (**Component**) som en basklass för den ursprungliga klassen (**ConcreteComponent**) och de nya **Decorator**-klasserna.
- Den hoplänkade kedjan av objekt skall avslutas med en **ConcreteComponent**.

ANVÄND NÄR
- Du vill lägga till uppgifter till ett objekt dynamiskt och transparent, dvs utan att andra objekt berörs.
- Utvidgning genom sub-classing är opraktiskt, t.ex. när antalet nya klasser för att täcka alla möjliga kombinationer blir för stort.

FACADE *structural*

"Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use."



Med **Facade** skapar vi alltså ett interface med endast de nödvändiga metoderna för att utnyttja delar av andra komplexa system. Kommunikationen med de underliggande systemen kapslas in i implementationen av **Facade**.

SYFTE
Du vill förenkla användningen av ett existerande system genom att definiera ett eget anpassat interface.

PROBLEM
Du vill använda bara en delmängd av ett komplext system eller du vill använda systemet på ett speciellt sätt.

LÖSNING
Facade erbjuder ett nytt interface för klienter till systemet.

KONSEKVENSER
Facade förenklar användningen av ett system. Eftersom **Facade** ger ett anpassat och förenklat interface kommer kanske viss funktionalitet att saknas.

IMPLEMENTATION
Definiera ett nytt interface med önskade egenskaper. Implementationen av den nya klassen använder det befintliga systemet.

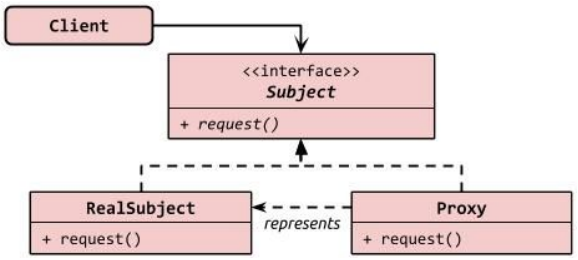
ANVÄND NÄR
- Du vill använda ett enkelt interface till ett komplext subsystem.
- Det finns många beroenden mellan klienter och implementationerna som de använder. Genom att introducera en **Facade** kan subsystem kopplas loss från klienter och andra subsystem.
- Du vill införa olika skikt för subsystemen. En **Facade** kan definiera accesspunkter till varje skikt. Om subsystemen är beroende av varandra kan de kommunicera gneom sina respektive **Facader**.

PROXY *structural*

"Provide a surrogate or placeholder for another object to control access to it."

En **Proxy**-klass erbjuder allltså ett interface för att kommunicera med en annan klass som svarar för den egentliga implementationen.

Generellt sett så ändrar **DP Proxy** kommunikationen mellan klienten och målobjektet genom att introducera ett mellanskikt.



SYFTE
Att erbjuda en **stand-in** för ett annat objekt i syfte att kontrollera access till det.

DELTAGARE
- **Proxy**
  - Innehåller en referens som tillåter **Proxy** att accessera det verkliga subjektet, **RealSubject**.
  - Erbjuder samma interface som **Subject** så att **Proxy** kan ersätta **RealSubject**.
  - Kontrollerar access till **RealSubject** och kan också ansvara för att skapa och förstöra det.
  - En **Remote Proxy** ansvarar för att skicka vidare ett anrop med argument till **RealSubject** som kan vara i en annan adressrymd.
  - En **Virtual Proxy** skapar **RealSubject** 'on demand' och kan också lagra information om det, t.ex. storleken på en bild.
  - En **Protection Proxy** kontrollerar att rätt behörighet finns för anropet.
- **Subject**
  - Definierar det gemensamma interfacet för **RealSubject** och **Proxy** så att **Proxy** kan användas var som helst där ett **RealSubject** förväntas.
- **RealSubject**
  - Definierar det verkliga objektet som **Proxy** representerar.

KONSEKVENSER
Proxy DP inför ett mellanliggande skikt i kommunikationen när en klient accesserar ett objekt. Detta extra skikt kan utnyttjas på flera sätt:

- En **Remote Proxy** bidrar till att göra accessen till ett objekt i en annan adressrymd transparent för klienten.
- En **Virtual Proxy** kan bidra med optimeringar som t.ex. att skapa ett objekt 'on demand'.
- En **Protection Proxy** utför accesskontroller.
- **Copy-on-Write** kan minska resursförbrukningen vid kopiering av stora objekt.
- En **Smart Reference** kan ersätta explicita allokera- / deallokera-operationer.

VARIANTER
- **Remote Proxy**
  - För applikationer som har två eller flera lager av kommunicerande mjukvara är det ofta praktiskt att införa en **Proxy** som mellanliggande och åtskiljande lager. Exempelvis kan man i **server-/client**-sammanhang låta klienten arbeta mot ett lokalt objekt, en **Remote Proxy**, som är en lokal representant för **servern**.
- **Protection Proxy**
  - En **Proxy** som dessutom filtrerar kommunikationen och avgör vilken information som skall skickas, och hur den skall skickas.

ANVÄND NÄR
- Du vill ha ett lokalt objekt som representerar ett objekt i en annan adressrymd.
- Du vill skapa objekt **on demand**.
- Du vill kontrollera att en klient har rätt behörighet vid access till ett objekt.
- Du vill utföra kompletterande **actions** i samband med access till ett objekt.