Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 6 – Att tänka i Patterns och DP Command

Designmönster med C++

Syfte:	Fördjupade resonemang kring designmönster. Ger exempel på hur designmönster kan kombineras och i vilken ordning de bör tillämpas. Ger även den utlovade lösningen på problemet på CAD/CAM-systemet. Slutligen presenteras DP Command.
Att läsa:	Design Patterns Explained kap. 12 – 13 sid 229-250

Kapitel 12 ’How do experts design?’

I kapitlet diskuterar författaren hur Alexander’s ideér och resonemang om designmönster kan appliceras på designmönster för mjukvara. Det är intressant läsning men inte helt lätt att förstå. Läs kursivt och begrunda.

Det vi bör ta med oss från kapitlet är de regler som utkristalliseras

- ’One at a time’
  – Om vi har flera designmönster i vår design ska vi tillämpa dem ett och ett efter varandra.

- ’Context first’
  – Tillämpa det mönster som skapar miljön (context) för ett annat mönster före det andra mönstret. Detta upprepas tills alla mönster har tillämpats.

Kapitel 13 ’Solving the CAD/CAM Problem with Patterns.’

Boken föreslår följande process för att ‘Tänka i designmönster’. Detta ska leda till en konceptuell design.

I grova drag kan den beskrivas som följer.

1. Identifiera mönstren i problemet.

2. Analysera och tillämpa mönstren.

   Upprepa för alla funna mönster:

   2a. Ordna mönstren efter hur de skapar miljön för andra mönster. Ett mönster skapar miljön för ett annat mönster, två mönster skapar inte miljön åt varandra.

   2b. Ta nästa mönster på tur och tillämpa det.

   2c. Identifiera eventuella nya mönster som har framkommit under den senaste iterationen. Lägg dem till listan av mönster som ska behandlas.

3. Arbeta vidare med förfining. Lägg till detaljer för klasser och metoder.

För att lösa problemet med CAD/CAM-systemet är fyra designmönster aktuella enligt diskussionerna i tidigare kapitel:

- Abstract Factory
- Adapter
- Bridge
- Facade

Frågan är hur de ska relateras till varandra.
Vilket skapar ’context’ för ett annat?
Vilket är ’seniormost’ (’topmost’,’outermost’, ’context-setting pattern’)?

Abstract Factory kan strykas direkt eftersom det själv kräver en ’context’ i vilken det ska skapa rätt uppsättning objekt. Erfarenheten visar också att man bör först bestämma vad man ska ha i ett system *innan* man funderar hur det ska skapas.

Återstår paren

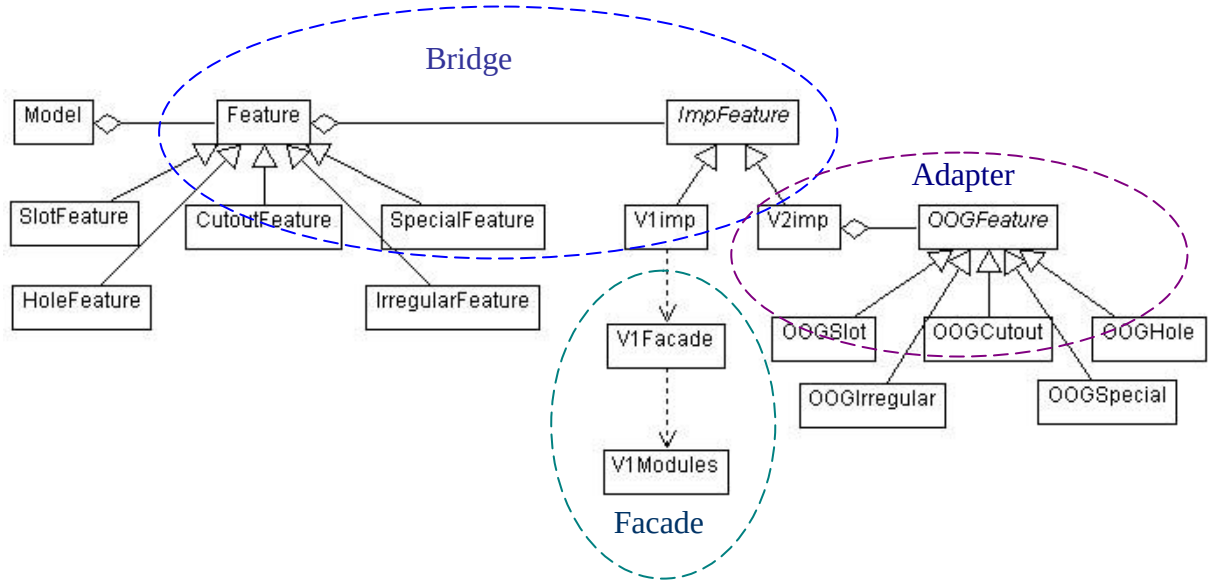
- Adapter - Bridge
- Bridge - Facade
- Facade – Adapter

Adapter – Bridge:
Bridge används ofta så att man först definerar de abstraktioner man behöver utan att då bekymra sig om de konkreta implementationerna. Dessutom kan vi inte tillämpa Adapter innan vi vet vad vi ska adaptera till. Följaktligen kan inte Adapter skapa en miljö för Bridge utan det är tvärtom så att Bridge skapar en miljö för Adapter.

Bridge – Facade:
Facade ska användas för att förenkla V1-systemets (funktions-orienterade) interface. Men det är någon av de konkreta implementationerna av Bridge som ska använda Facaden. Därför är det i detta fall Bridge som skapar miljön för Facade.

Bridge är alltså vårt ’senior-most pattern’ som bör tillämpas först.

Liknande resonemang och tillämpning av stegen i processen för att ’Tänka i mönster’ leder fram till den slutliga designen av systemet:



Du kan se att principerna
”Find what varies and encapsulate it” och *”Favour aggregation over inheritance”* har lett fram till en design där ett eventuellt tredje CAD/CAM-system kan läggas till i systemet enbart genom att skapa en ny adapter.

Jämför detta med den ursprungliga designen (fig. 13-12) som var helt baserad på arv. Här skulle ett nytt CAD/CAM-system leda till 5 nya V3xxx-klasser som ska adderas till de 10 konkreta Feature-klasser som redan finns.

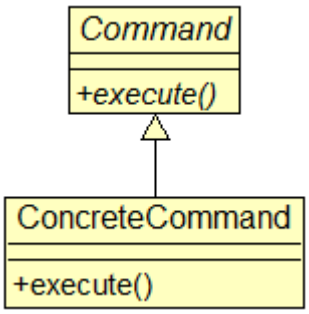
Nu ska vi återvända till det centrala temat för kursen och angripa ett nytt designmönster. Jag ska presentera DP Command. Det är ett DP som inte finns behandlat i kursboken men som kan vara mycket användbart.

DP Command

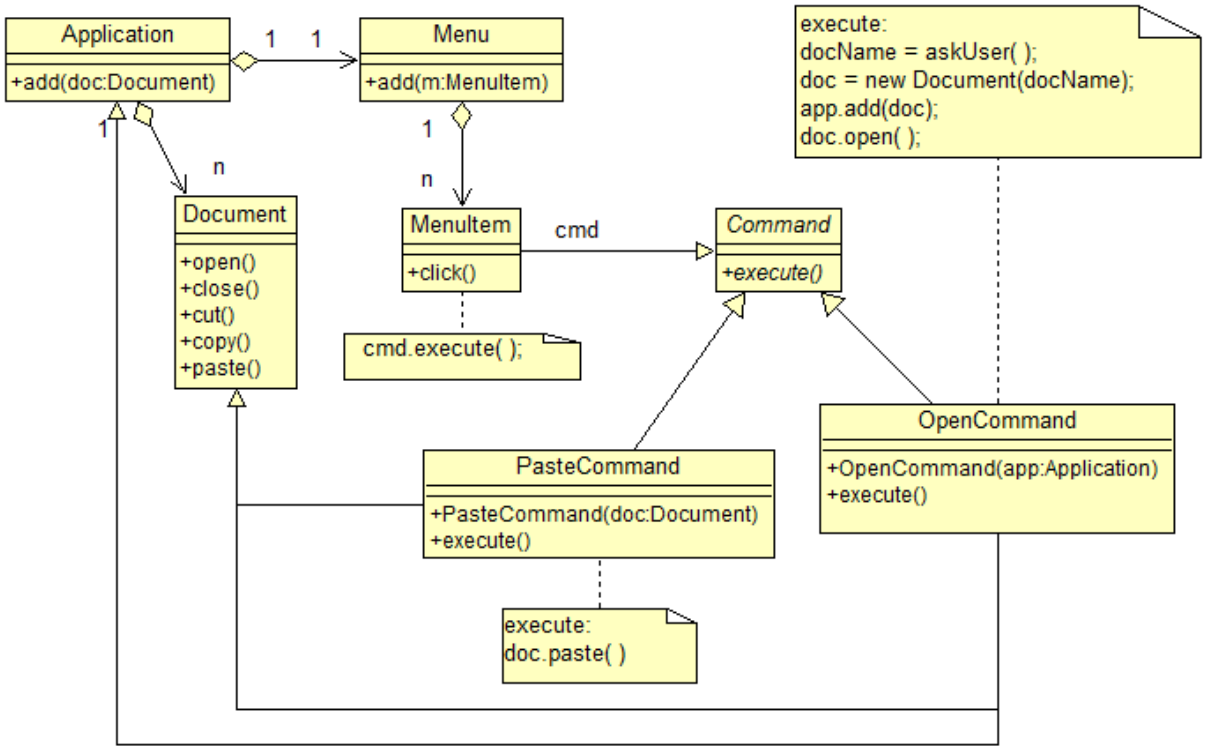
Syfte:
” *Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations* .” (GoF)

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 ska utföras, ”command receiver”. Detta kan åstadkommas genom att paketera anropet i ett objekt – ett ”Command object”. Ett sådant objekt kan skickas som parameter, sparas, köas, loggas osv.

Klassdiagrammet visar en enkel form av den abstrakta klassen Command. Ett ramverk för t.ex. ett grafiskt gränssnitt kan tillhandahålla menyer (Menu) och klickbara menyval (MenuItem) men vad som ska ske när ett MenuItem klickas är något som den aktuella applikationen måste definiera.



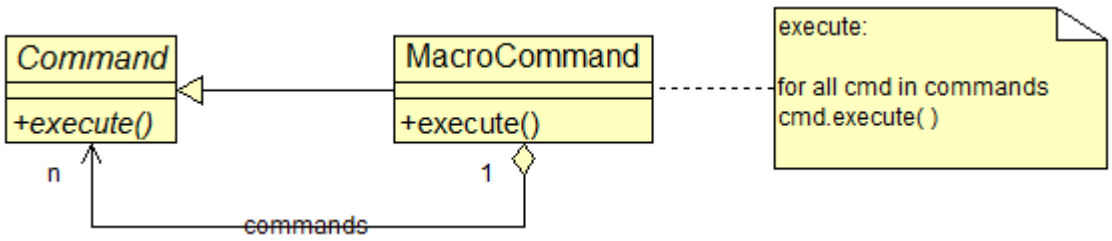
Med DP Command kan detta lösas genom att konfigurera varje MenuItem med ett konkret Command-objekt. När användaren väljer ett menyalternativ så anropar MenuItem-objektet execute-metoden för sitt Command-objekt. MenuItem vet inte (och behöver inte veta) vilket konkret Command-objekt som används. De konkreta Command-objekten vet däremot vilka objekt som är ’Command receivers’ och utför en eller flera operationer på dem. Dessa Command receivers ges oftast som argument till Commando-objektens konstruktor.
Nästa figur visar detta i ett större sammanhang med en applikation, dess dokument, meny och ett par menyalternativ (från GoF).



En viktig iakttagelse i exemplet är att metoden att konfigurera ett MenuItem-objekt med ett Command-objekt är en objektorienterad motsvarighet den funktionsorienterade programmeringens *callback-funktioner*:

→ istället för att registrera en funktion som ett ramverk ska exekvera som respons på en viss händelse så registrerar vi ett konkret Command-objekt vars execute-metod kommer att exekveras.

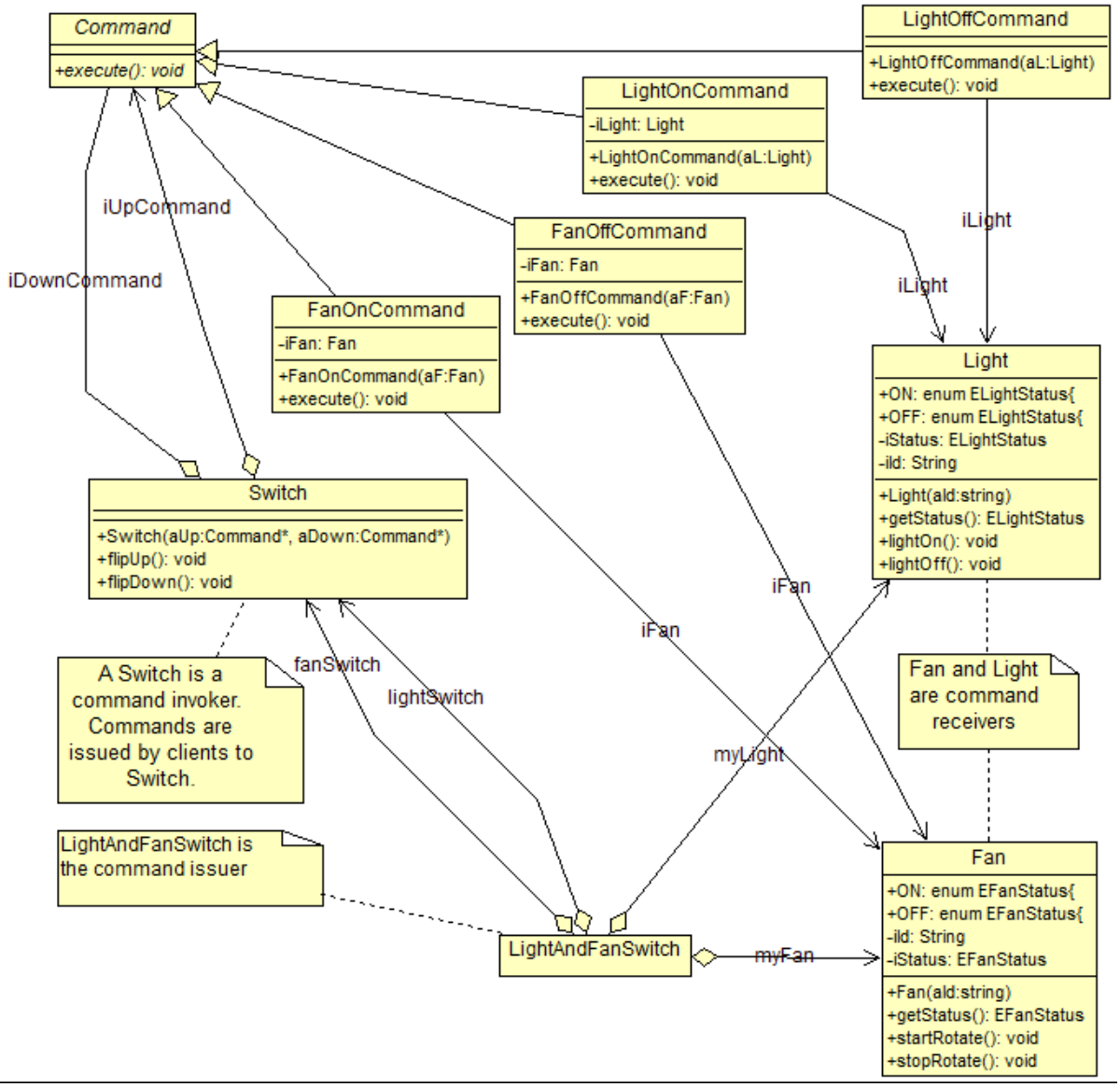
Ibland behöver ett Command-objekt exekvera en sekvens av kommandon som vart och ett representeras av ett annat Command. Genom att skapa en klass MacroCommand som en subklass till Command och låta klassen innehålla en kontainer med (referenser till) andra konkreta Command klasser kan vi lösa detta. MacroCommand.Execute anropar de övriga Command-objektens Execute-funktioner:



Konstruktionen med en subklass som aggregerar referenser till sin basklass är i sig en tillämpning av ett annat DP, nämligen det strukturella mönstret *Composite*. Effekten blir i detta fall att även ett MacroCommand kan behandlas som ett ”vanligt” Command. Se GoF för en fullständig diskussion av Composite.

I exemplet exLightAndFanSwitch, finns ytterligare en nivå av ”redirection”:

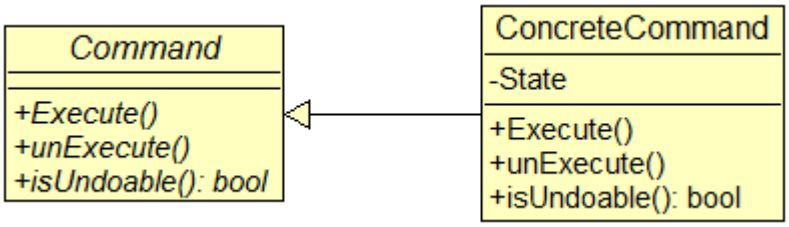
- Två ”*command recievers”*, Light och Fan, kan sättas på och stängas av via var sin Switch.
- Switcharna kan ställas UPP eller NER med operationerna flipUp och flipDown.
- Varje switch konfigureras av klientprogrammet med två Command-objekt, ett för flipUp och ett för flipDown.
- Klientprogrammet är här ’*Command issuer*’ medan switcharna har rollen av en *’Command invoker’* som anropar sina Commando-objekts execute-funktioner (se exLightAndFanSwitch).



I exemplet konfigureras Switch med sina Command-objekt via konstruktorn. De kan ”återanvändas” eftersom de i det här fallet inte innehåller några attribut med statusinformation, de är *state-less*.

Undo/Redo

En vanlig tillämpning av DP Command är implementation av ’*undo/redo*’-funktionalitet för att kunna ’ångra’ och åter-exekvera kommandon. För att klara detta kan man som grundsten använda en Command-klass som utvidgats med en *unExecute*-funktion. Den implementeras som ’omvändningen’ av execute så att tillståndet före exekveringen av execute (om möjligt…) återställs. Det innebär också att ’state’ för att kunna återställa tillstånd ibland måste sparas i en eller flera datamedlemmar. Eftersom det kan finnas Command-typer som inte går att göra ogjorda tillför vi metoden *isUndoable* som returnerar en boolean.

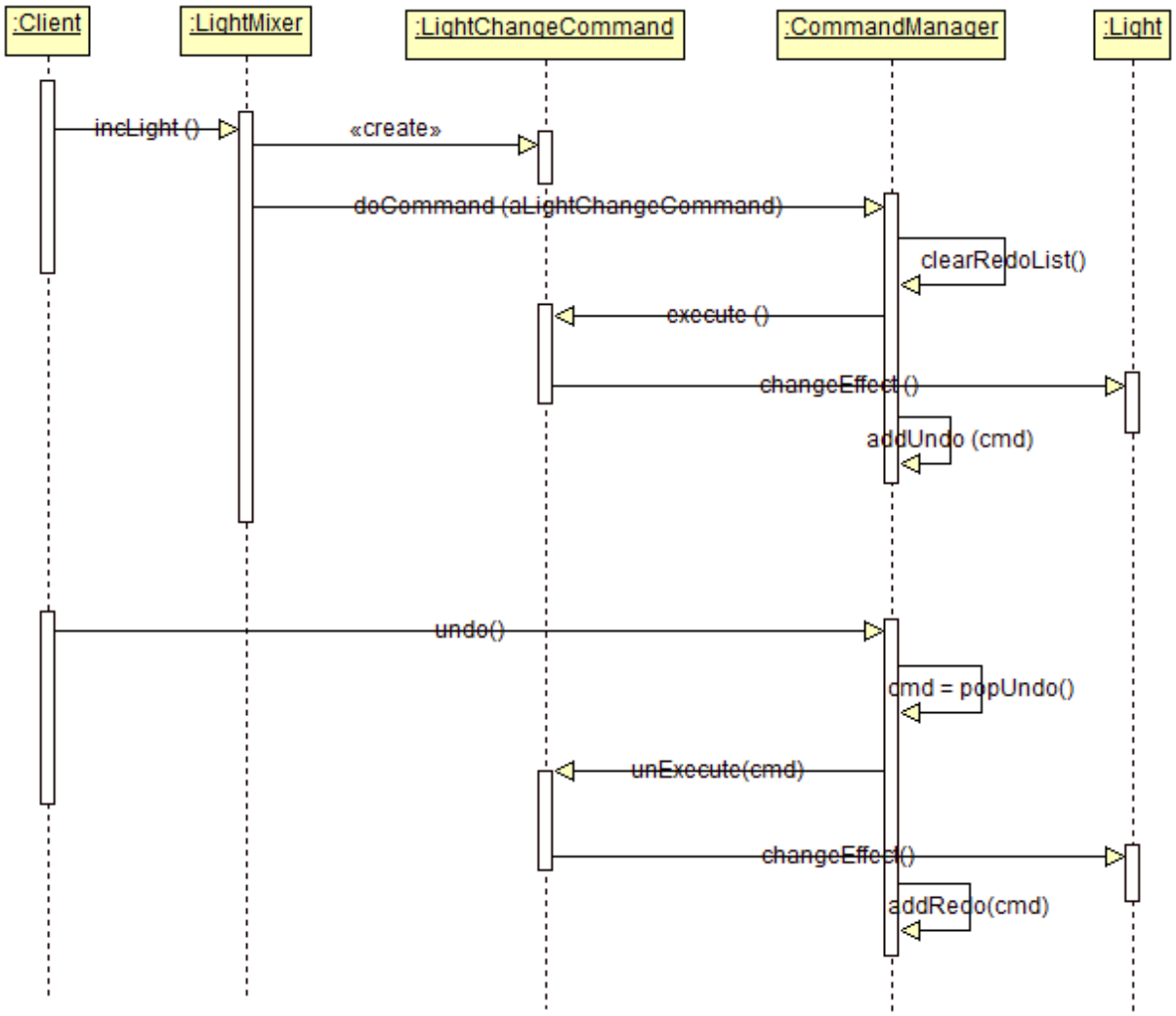


En tillämpning av detta finns i exemplet exLightMixerCommand. En LightMixer styr tre lampor, som är av klassen Light (command receiver). Intensiteten av färgerna grönt, rött och blått kan sedan ändras oberoende av varandra. LightMixer (command issuer) konfigureras med en CommandManager (command invoker).

Klientprogrammet ändrar ljuset genom operationer på en LightMixer som i sin tur skapar motsvarande Command-objekt. Dessa skickas som argument till CommandManager’s *doCommand*-metod som genomför operationen på ett Light-objekt genom att utföra execute-metoden i kommando-objektet. CommandManager implementerar dessutom undo/redo genom att spara kommandon i två LIFO-strukturer. Klientprogrammet kan utföra undo/redo genom operationer på CommandManager. I en applikation är CommandManager en stark kandidat till en Singleton, ett DP som introduceras senare i kursen.

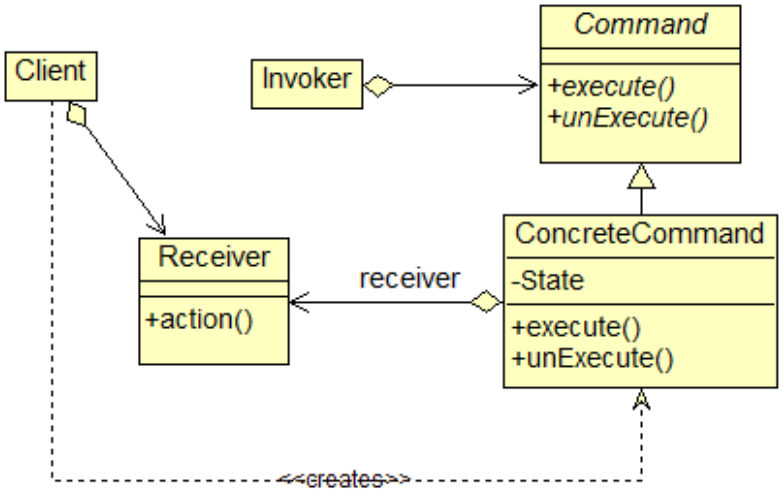
Det följande sekvensdiagrammet visar (förenklade) interaktionerna mellan inblandade objekt för

- ökning av ljusintensitet för en lampa
- undo av ökningen



Sammanfattning
DP Command (’Action, Transaction’)

- Struktur



- 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 att exakt kännedom om mottagaren eller vilken operation som ska 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 operationen 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 Command när du vill
  - Parametrisera objekt med en ’action’ som ska utföras. I ett procedurellt 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 kommandon. Om mottagaren (receiver) kan representeras oberoende av den aktuella adressrymden kan ett Command skickas till 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 läsas från disk och återexekveras med execute.
  - Specificera, kö, logga och utföra kommandon vid olika tider. Livstiden för ett Command kan vara oberoende av det ursprungliga kommandon. Om mottagaren (receiver) kan representeras oberoende av den aktuella adressrymden kan ett Command skickas till en annan process och utföras där.
  - Implementera transaktioner som innefattar flera ändringar av data och som ska genomföras i sin helhet eller inte alls. Alla transaktioner kan ges ett gemensamt interface och nya typer av transaktioner kan enkelt läggas till.