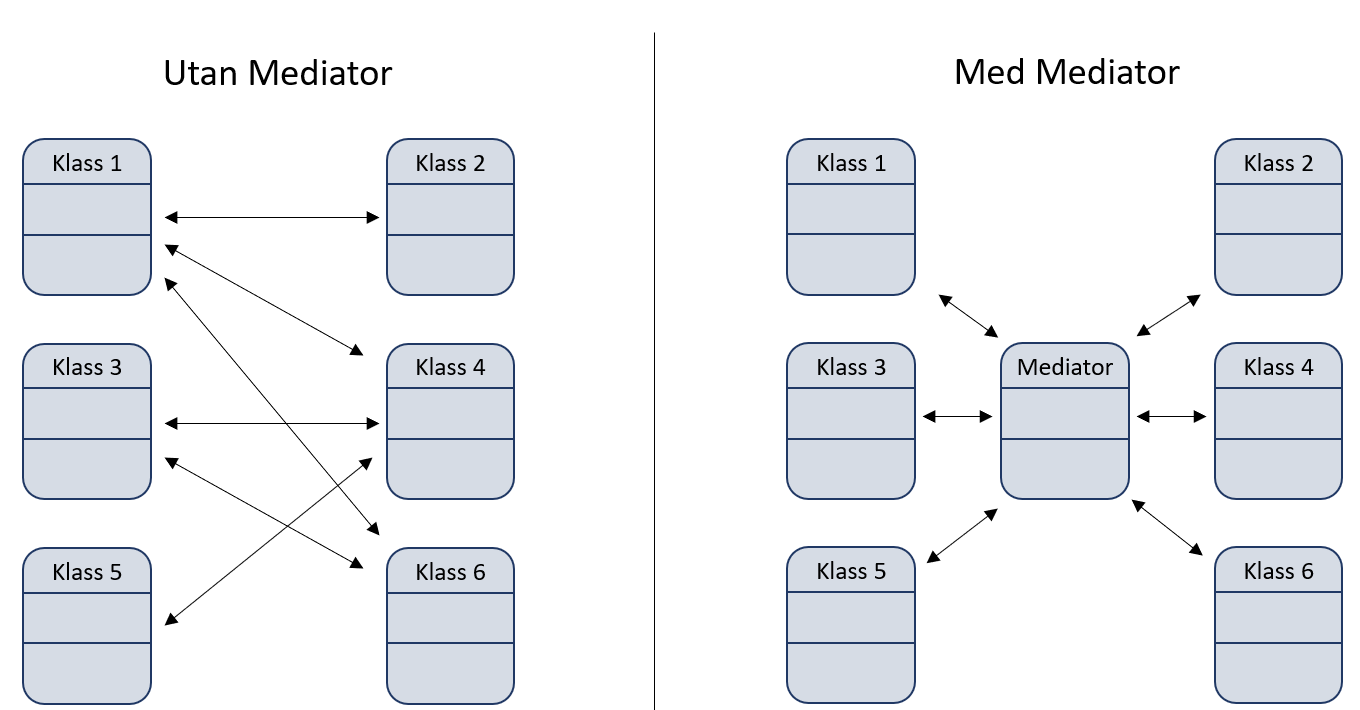
# Creator(någon av dessa)

# Mediator

Ett komplext system med många olika klasser som interagerar med varandra medför sannolikt att vissa ändringar av klassers beteende kräver korrigeringar även hos de klasser som respektive ändrad klass interagerar med. Utöver den tid som krävs för att korrigera dessa ökar risken för fel eftersom mer kod ändras samtidigt som att det kan vara svårt att spåra exakt vilka klasser som behöver korrigeras.

Genom mediator- mönstret kan vi istället abstrahera ut kommunikationen mellan klasser till en egen mediator-klass som hanterar all kommunikation mellan klasserna. Abstraheringen möjliggör att arbete med uppdateringar av en enskild klass kan begränsas till den berörda klassen samt mediator-klassen.

Exempel på relationer mellan klasser i ett system utan och med en mediator-klass:



Om vi i exemplet ovan behöver ändra klass 4 behöver vi beakta att klass 4 har relationer med klass 1, 3 och 5 vilka i sin tur med stor sannolikhet också behöver ändras.

I fallet med en mediator kan vi i många fall komma undan med att bara ändra klass 4 och möjligtvis vår mediator.

Utöver att risken för fel minskar när färre komponenter bearbetas så blir de fel som trots allt uppstår ofta mindre, och enklare att identifiera, eftersom grundidén med mediator-mönstret ofta begränsar felen till att bara beröra den komponent som bearbetas.

Sammanfattningsvis handlar mönstret om att lösgöra kopplingen mellan olika klasser vilket tjänar till att ersätta ”många till många”-relationer med ”en till många”-relationer.

# Facade

Precis som namnet föreslår så handlar mönstret om att applicera enkelhet på komplex funktionalitet genom att samla denna i en enklare funktion. I kodsammanhang handlar fasaden inte nödvändigtvis om att göra koden vacker likt fasaden på ett hus och kan kanske mer träffsäkert liknas med den kran som finns vid handfatet i ett badrum:

Bakom kulisserna (i källkoden) finns flertalet komponenter så som kopplingen till det lokala avloppet, en tank för uppvärmt vatten, en koppling till husets avloppsystem och husets koppling till stadens vattenledningar som i sin tur går genom vattenreningsverk (man behöver inte gå speciellt många steg för att uppnå en komplexitet som skulle kräva många timmars studier som ingenjör för att förstå).

För användaren, som exempelvis kan liknas med klienten, består gränssnittet endast av ett vred för varmt respektive kallt vatten samt i vissa fall ett reglage för vattentryck. Att vem som helst kan använda ett handfat är resultatet av ett lyckat gränssnitt genom en fasad.

Fasaden som designmönster syftar likt handfatet till att försköna och förenkla koden.

***Exempel 1***

Vi föreställer oss att tre funktioner skall köras i samband med en händelse i ett system enligt följande:



Genom en fasad kan vi förenkla interaktionen genom att bara kalla på en funktion:

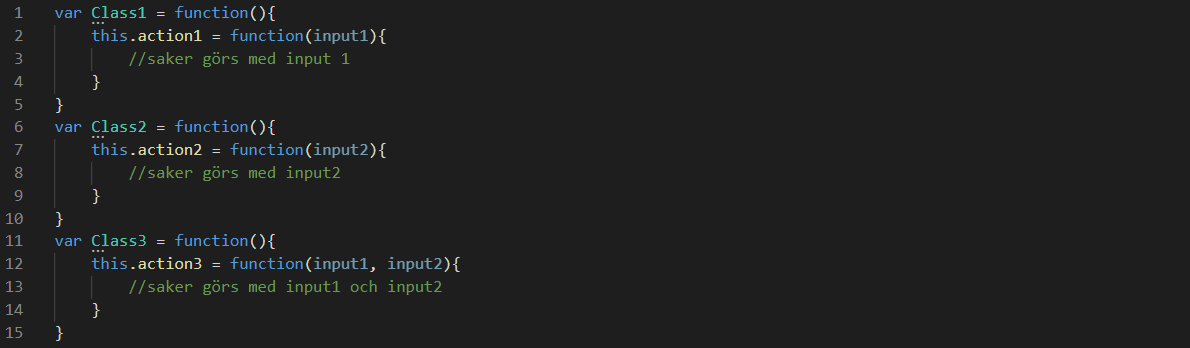


Resultatet blir att händelsen endast kräver att en funktion körs (eftersom resterande funktioner är inbakade):



***Exempel 2***

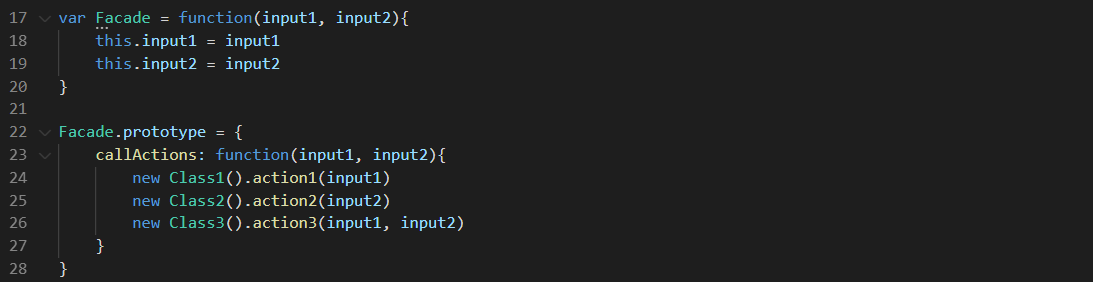
Vi föreställer oss samma scenario fast klassbaserat, där alla tre klasser skall kallas varje gång en viss händelse sker:



För att kalla på klasserna skriver vi således ut varje klass med dess tillhörande funktioner och parametrar enligt följande:



För att förenkla kallelsen på de olika klasserna skapar vi enligt fasadmönstret ett gränssnitt som kallar på klasserna åt oss genom en egen fasad-klass:



Resultatet blir att vi kan kalla på alla klasser med deras respektive funktionalitet genom att kalla på vår fasadklass (Facade). Notera även att klassgemensamma parametrar även i detta exempel inte behöver skickas med flera gånger:



**Fördelar och nackdelar**

Koden blir likt i utföranden ovan mindre repetitiv vilket leder till en mer strukturerad kod som dessutom blir enklare att underhålla. Om ytterligare en klass skall kallas på vid varje tillfälle i exempel 2 ovan så kan denna klass med parametrar enkelt läggas till i fasad-klassen (kallad ”Facade” ovan). Således minimeras antalet ställen koden måste kompletteras vilket sannolikt leder till högre hastighet av kodskrivande samt minimerar risken för fel och buggar (ju fler ställen kod måste korrigeras vid en uppdatering, desto större risk för att fel uppstår enligt egen erfarenhet).

Koden blir dessutom enklare att hantera för andra personer än författaren genom att samla funktionalitet som hör ihop i samma fasad-klasser.