*Design Patterns forfatter: Henrik*

Introduktion til design patterns:

Design patterns er en beskrivelse af den bedste løsning, til et ofte gentaget problem, i software programmering. Et design pattern er ikke et færdigt design, der direkte kan omskrives til kode. Det er mere en beskrivelse eller skabelon til løsningen, som kan benyttes i mange forskellige situationer.

Et design pattern viser den bedste tilgang til løsningen af et problem, som er gennemtestet mange gange af erfarne udviklere/ programmører. Det skal forstås som den bedste løsning på de generelle problemer, man kan støde på som udvikler/programmør.

Hele begrebet design patterns opstod i 1994, da de fire forfattere

Erich Gamma, Richard Helm, Ralph Johnson og John Vlissides udgav en bog med titlen ”Design Patterns – Elements of Reusable Object-Oriented Software”. Disse fire forfattere kendes tilsammen som Gang of Four(GoF).

Brug af design patterns:

Design patterns kan gøre udviklingsprocessen hurtigere ved at levere gennemtestede løsninger. Effektiv software design kræver, at man overvejer problemer, der måske ikke opstår før senere i implementationen. Ved at ”genbruge” kendte design patterns, kan man undgå småting, der kan give store problemer senere. Det øger desuden læsbarheden for andre programmører, der kender det anvendte design pattern i forvejen.

Da design patterns ofte er udviklet over lang tid, er den struktur de tilbyder, som regel den bedste og mest effektive måde at løse bestemte problemer på.

Typer af design patterns:

Ifølge ovennævnte bog, er der 23 design patterns der kan inddeles i tre hovedkategorier. Der er siden kommet mange flere til. I det følgende vil jeg inddele de 23 design patterns i de tre hovedkategorier, og give en kort beskrivelse af hver.

**Creational design patterns** handler om instantiering af objekter.

Det indeholder følgende design patterns:

Abstract factory bruges til at give en klient et sæt af relaterede eller afhængige objekter.

Builder skaber komplekse objekter.

Factory method kan erstatte konstruktorer og derved gøre objekt konstruktionen abstrakt.

Prototype instantierer nye objekter ved at kopiere eksisterende objekter.

Singleton sikrer at der kun eksisterer en instans af en bestemt klasse.

**Structural design patterns** handler om klasse og objekt composition.

Det indeholder følgende design patterns:

Adapter skaber forbindelse mellem to inkompatible typer.

Bridge adskiller abstrakte elementer i en klasse fra implementationen. Muliggør udskiftning af implementationen uden at ændre i det abstrakte.

Composite skaber rekursive træ strukturer af sammenhængende objekter.

Decorator kan øge eller ændre funktionaliteten i et objekt under kørsel.

Facade giver et simpelt interface til et mere komplekst objektsystem.

Flyweight reducerer ressource forbrug for komplekse modeller.

Proxy giver et erstatningsobjekt med reference til et underliggende objekt.

**Behavioral design patterns** handler om kommunikation mellem klasser og objekter.

Det indeholder følgende design patterns:

Chain of responsibility giver et container objekt, der modtager alle requests og afgør hvilket objekt, der skal varetage den.

Command lagrer requests i et command objekt, inklusiv kald med alle påkrævede parametre.

Interpreter håndterer grammatik.

Iterator giver mulighed for at gennemløbe elementerne i en collection.

Mediator reducerer kobling mellem klasser der kommunikerer sammen.

Memento gemmer den aktuelle tilstand i et objekt og kan genskabe det senere.

Observer definerer et link mellem et objekt(subject) og dens afhængige objekter(observers). Dette giver observers besked, så snart tilstanden i subject objektet ændrer sig.

State ændrer adfærden af et objekt, når dets tilstand ændres.

Strategy indeholder lignende algoritmer liggende i hver deres subklasse, den aktuelle der skal køres afgøres ved run-time udfra krav.

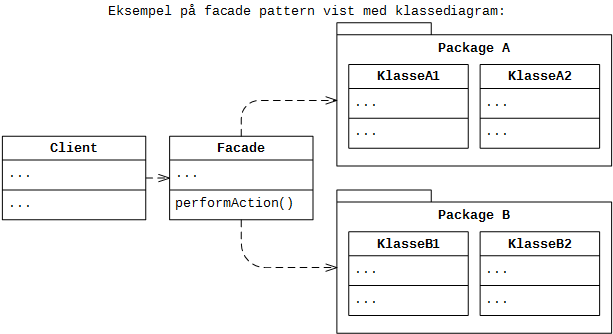
Template method definerer skridt i en algoritme og tillader

hvert skridt at blive ændret uafhængigt.

Visitor adskiller komplekst datastruktur fra den funktionalitet, der ønskes udført på det.

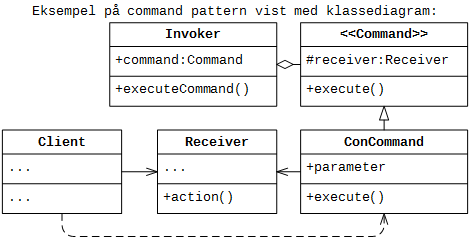
I forbindelse med vores opgave har vi benyttet tre design patterns, som jeg vil beskrive nærmere i det kommende.

**Facade pattern** giver et simpelt interface til et mere komplekst objektsystem.

Det er ideelt, når man arbejder med et stort antal af afhængige klasser, eller klasser der kræver brug af mange metoder. Facade pattern bruger en enkelt klasse, med forenklede metoder der bruges af klienten. Det delegerer de forskellige kald ud til metoder i eksisterende klasser i systemet.

**Command** **pattern** lagrer requests i et command objekt, inklusiv kald med alle påkrævede parametre.

Det indeholder ikke funktionaliteten der skal udføres, men al information der skal til for at udføre et kald. Funktionaliteten gemmes i et receiver objekt. Dette sørger for, at der er en løs kobling, mellem det objekt hvor hændelsen opstår, og det objekt der skal udføre funktionen. For at kontrollere hvornår kaldet udføres bruger man en invoker klasse.



**Observer** **pattern** definerer et link mellem et objekt(subject) og dens afhængige objekter(observers). Dette giver observers besked, så snart tilstanden i subject objektet ændrer sig.

Dette pattern giver et abstrakt forhold mellem objekter. Observer tilmelder sig hos subject for at kunne modtage besked, når tilstanden i subject ændrer sig. Subject underretter de tilmeldte observers, når en ændring opstår. Observer kan altid framelde sig disse opdateringer.

