4. State Pattern

1. **Redegør for hvad software design pattern er**
2. **Redegør for strukturen i State Pattern**
    a. Opbygget af klasser
        i. En klasse agere kontekst
        ii. En klasse der agere ”SuperState”
            1. SuperState er ofte en abstract klasse
            2. SuperState kan kaldes med *Events*
        iii. Andre klasser kan arve fra SuperState klassen
            1. Nedarvede klasser SKAL implementere funktionaliteten fra SuperState
            2. de nedarvende klasser indeholder individuelle States
            3. Implementerer de Event der er behov for
        iv. Konteksten sætter ikke State, det gør den enkelte Individuelle State klasse
    b. State pattern benyttes til at ændre adfærd for objekter
    c. Hjælper objekter med at ændre deres indre States
        i. Interfacet State ændres ikke
        ii. Overblik over elementers states opretholdes
    d. Gemt væk igennem context, hvis adfærd bestemmes af det nu aktive State
    e. Se Eksempel i disposition
    f. Gennemgå sekvensdiagram i disposition
3. **Sammenlign switch/case-implementering med GoF State**
    a. Det kan ikke betale sig at lave State Pattern I mindre systemer
        i. Her skaber en Switch/case mere overblik
        ii. I switchcases bliver State ofte bestemt af en Enum
        iii. I switchCases bliver der ofte kun benyttet én klasse
        iv. Ingen single responsibility
4. **Redegør for fordele og ulemper ved GoF State**
    a. + større overskuelighed i state machines, nemt at udvide
    b. + Adskillelse mellem actions og state machine logik
    c. + Fleksibelt og effiktivt
    d. + overholder Open/Close princippet
    e. – svært at danne overblik ud fra koden
    f. – generelt meget kode!
5. **Redegør for hvordan UML (SysML) STM mapper til GoF State.**
    a. Alle States -> SuperState
    b. Individuelle -> SubState på SuperState
    c. Starting Point -> Første satte State
    d. Events -> events der kaldes på konteksten
    e. Guards -> logik der implementeres i States
    f. Activity -> Kald der laves tilbage på Konteksten
    g. Tilstandsskifte -> Den State SetState kaldes med

4. State patterns

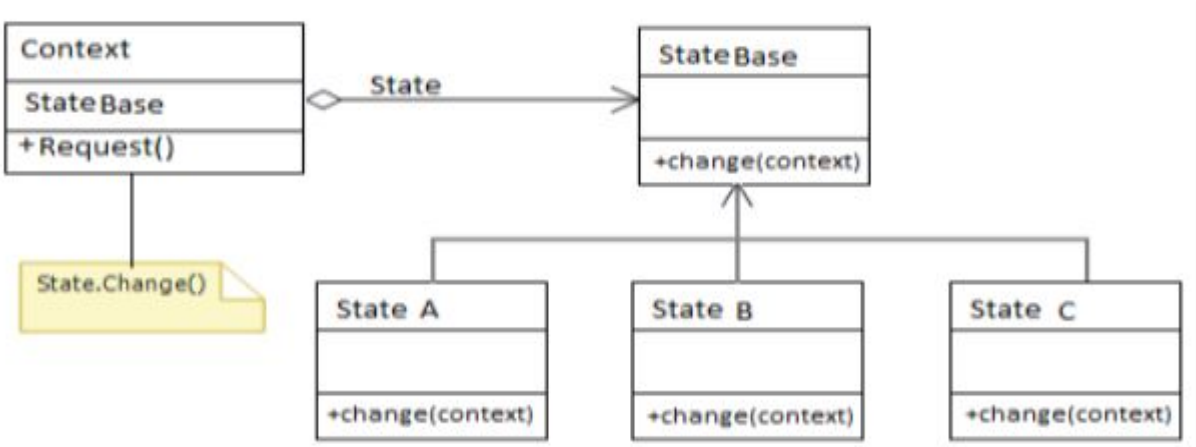
Redegør for, hvad et software design pattern er

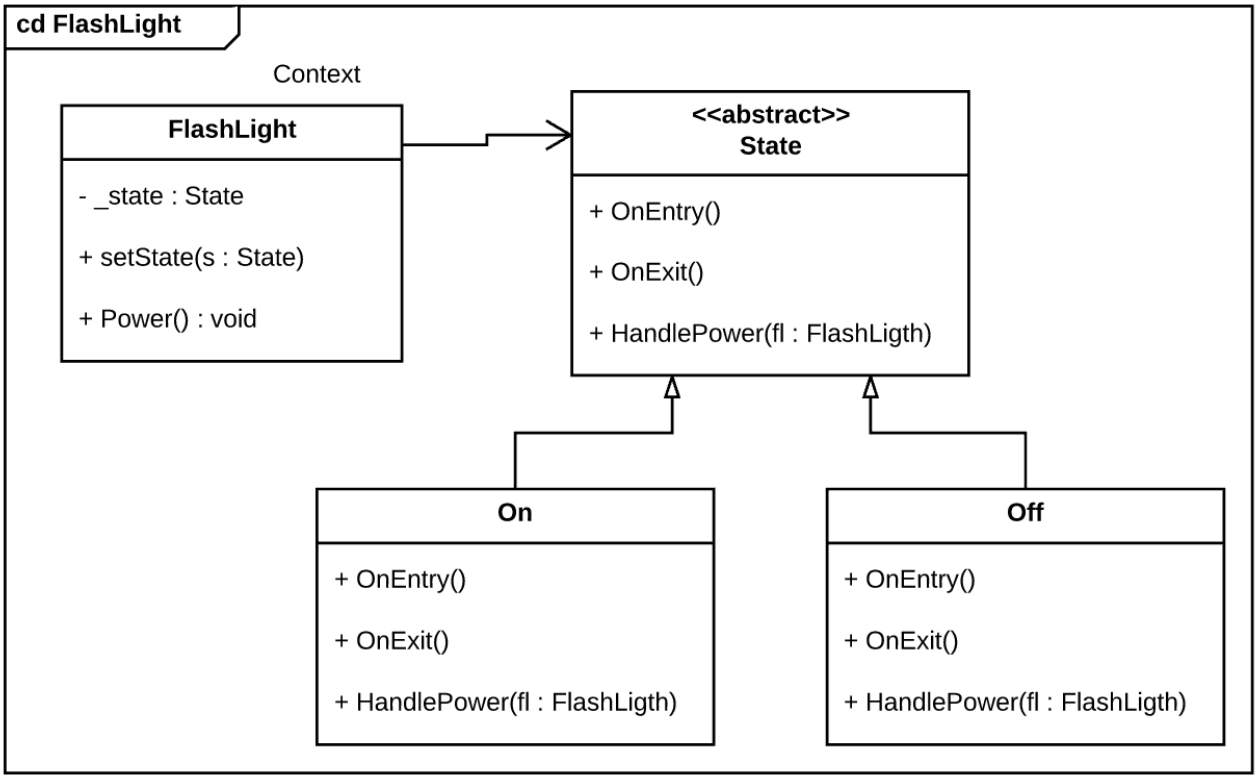
Et software design pattern er en general genbrugelig løsning til problemer der tit opstår i en given kontekst i software design. Det er **ikke** et færdigt design der kan laves direkte til kilde kode. Det er en **beskrivelse** eller **skabelon** for hvordan et problem **kan løses** i mange forskellige situationer. Det er formaliserede bedste praksisser som en programmør kan bruge til at løse problemer med.

Redegør for strukturen i GoF State Pattern

State pattern er opbygget via klasser hvor der er en klasse der agere kontekst det er denne klasse vi kalder når der sker events. Dertil er der en anden klasse som agere ”Superstate” det betyder at den kan blive kaldt med alle de mulige events der er i systemet. Denne ”Superstate” er oftest en **Abstract** klasse, sådan at funktionaliteten SKAL implementeres, i de klasser der **arver** fra den. Fra denne SuperState **nedarver** de **individuelle states** og implementer de events de har behov for. Det er **ikke konteksten** der sætter **state** det gør de **individuelle states**.

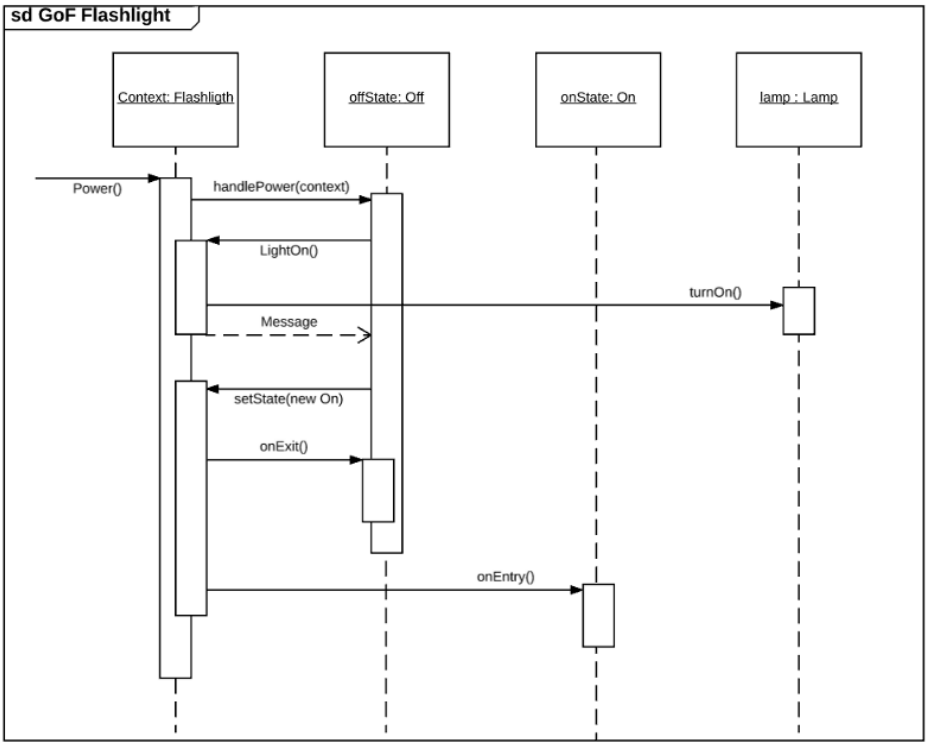
Et state pattern bruges til at ændre adfærd for de objekter hvis interne state ændres. Mønsteret hjælper med til at lade objekter ændre deres states, uden interfacet "**State**" ændres, eller miste overblik over objektets state. Alt dette er gemt væk igennem en context, som er det objekt, hvis adfærd er baseret på den nuværende aktive state, og er det eneste, der kan ses udefra. "**StateA**", "**StateB**" og "**StateC**" giver funktionaliteten, som bliver brugt til at ændre state og indeholder adfærd til context objektet.





Først oprettes et "Flashlight", som kan have en default state, eksempelvis "Off". Så kaldes "Power()", som i sin implementering skal kalde "HandlePower(flashlight)" i "State" fra hvad end det nuværende state er, og i dette tilfælde er det "Off". I implementeringen for"HandlePower()" i "Off" bliver "SetState(State)" sat til at ændre sig fra "Off" til "On", og i implementeringen for "SetState(state)" bliver det nuværende states "OnExit" funktion kaldt for "Off". Herefter bliver det nye state sat, og "OnEntry" på det nye state bliver kaldt.

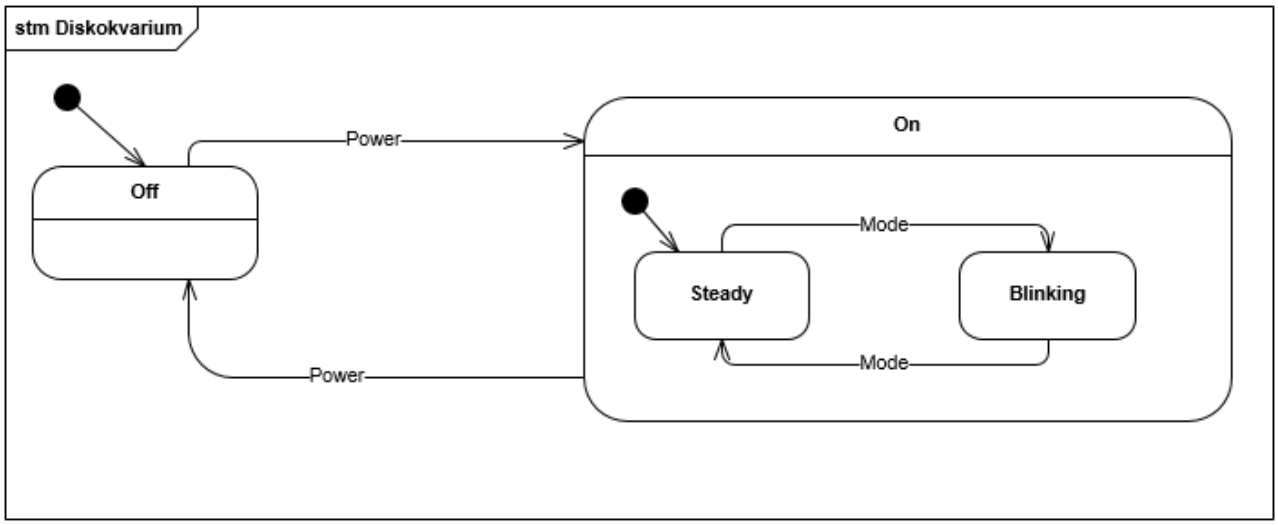
```
1   //FlashLight.cs
2   public void Power()
3   {
4       _state.HandlePower(this);
5   }
6
7   //Off.cs
8   public void HandlePower(FlashLight fl)
9   {
10      fl.setState(new On);
11  }
12
13  //FlashLight.cs
14  public void setState(State newState)
15  {
16      _state.OnExit(); //På nuværende state (Off)
17      _state = newState; //Ændrer State (fra Off til On)
18      _state = OnEntry(); //Kalder OnEntry (for On)
19  }
```



Sammenlign switch/case-implementering med GoF State

Ved mindre state maskiner kan GoF state ikke svare sig her er det mere overskueligt med en switch case.

Men har vi for eksempel nestede states giver GoF nogle fordele.



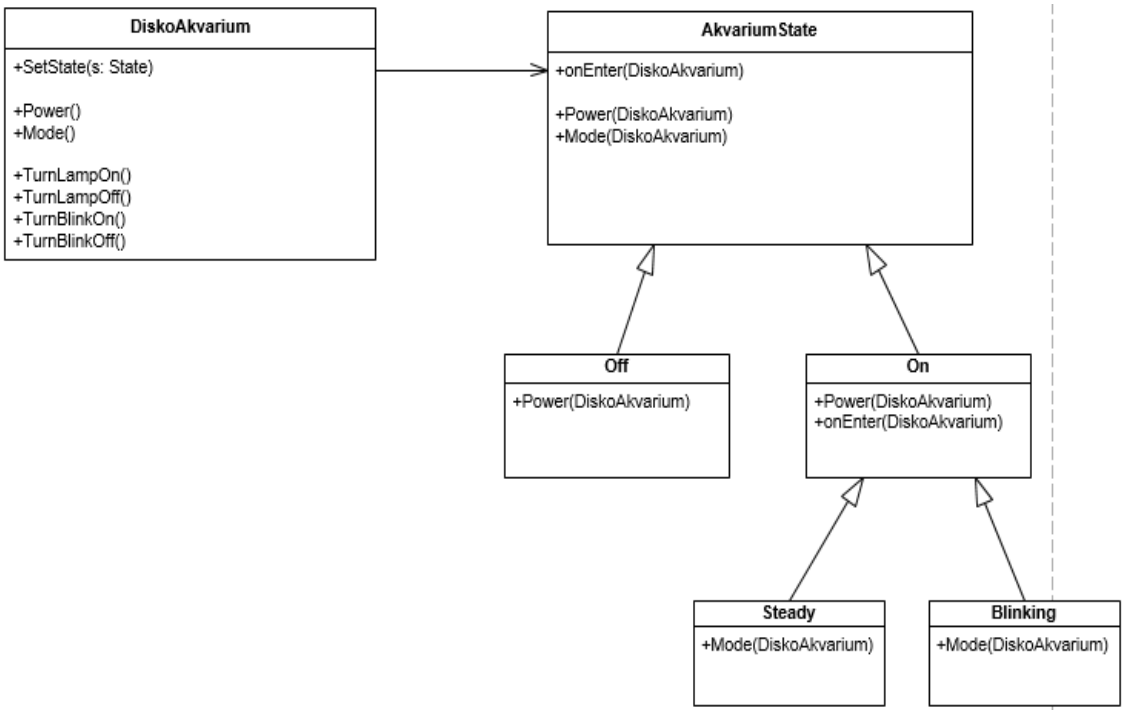
```
switch(currentState){
    case On:
    switch(Event){
        case Power: {
            currentState = Off;
            TurnLampOff();
        }

        case Mode: {
            switch(currentSubState) {
                case Steady: {
                    currentSubState = Blinking;
                    TurnBlinkOn():
                }

                case Blinking {
                    currentSubState = Steady;
                    TurnBlinkOff();
                }
            }
        }
    }
    case Off:
    switch(Event) {
        case Power: {
            currentState = on;
            currentSubState = Steady;
            TurnLampOn();
        }

        case Mode: {
            break;
        }
    }
}
```

Da dette er en simpel state maskine, kan man bruge en switch statement, som vist ovenfor. Når der bliver brugt et switch statement, så bliver *currentState* oftest bestemt via en *enum*. Når man bruger et switch statement, så har man typisk KUN 1 klasse. Dette gør at man ikke overholder **Single Responsiblity**, da alt funktionaliteten ligger i en klasse.

Switch statement kan føres over til et state diagram som ses nedenunder.

SOLID

S: Overholdes da hver state har sit eget ansvar.
O: Det er muligt at udvide med flere states ude at skulle ændre andet kode.
L: Der er muligt at substituerer alle state med alt der opfylder ”StateBase”
I : **IKKE** opfyldt da hver state ikke har sin egen interface
D: Opfyldt da der er en abstrakt klasse mellem Context og alle ”ConcreteState”



Redegør for fordele og ulemper ved anvendelsen af GoF State.

Fordele:

- Der opnås **større overskuelighed** ved store state machines. Det er **nemt at udvide**
- Der er **adskillelse** mellem **actions** og **state machine logikken**.
- Det er **fleksibelt** og effektivt
- Overholder **Open/Close princippet**

Ulemper:

- Det kan være **svært** at danne sig et **overblik** over state machinen ud fra kode.
- Der skal **meget kode** til!

Redegør for, hvordan et UML (SysML) state machine diagram mapper til GoF State.

Der kan mappes fra et statediagram til GoF state således:

- Alle states → Superstate
- Individuelle → SubState på superstate
- Starting point → Første state der bliver sat
- Events → De events der kan kaldes på konteksten
- Guards → Logik der skal implementeres i states
- Activity → Kald der laves tilbage på konteksten
- Tilstandskiftet → Hvilken State SetState skal kaldes med

Relaterede patterns

The Flyweight (218) pattern explains when and how State objects can be shared.

State objects are often Singletons (144).

Større eksempel

Fra Derek Banas state pattern video: https://www.youtube.com/watch?v=MGEx35FjBuo