1 Strategy Pattern + Template method pattern

1. **Redegør for hvad et software design pattern er.**
2. **Sammenlign strategy- og template pattern**
    a. Begge ”forsøger” at adskille algoritme fra implementering
    b. Template = **arv**
    c. Strategy = **delegation** (udskiftning under run-time)
    d. Template: compile-time algorithm (selection by **subclassing**)
    e. Strategy: run-time algorithm (selection by containment)
3. **Redegør for template**
    a. Løser problemer ved arv af abstract classes
    b. Indeholder beskyttede versioner af funktioner
    c. Indeholder funktion der kalder funktioner i ønsket rækkefølge.
    d. Benyttes hvis context skal kende til implementering af arvende klasser.
        i. Eks. Hvis man skal tjekke state på en klasse
    e. Benyttes hvis man vil have kendskab til intern struktur i en klasse
    f. Benyttes når ikke vil skifte ud i funktionaliteten under run-time
        i. alle funktioner skal overskrives/overrides
4. **Redegør for Strategy**
    a. Indkapsler funktionalitet i et interface
        i. Giver mulighed for udskiftning af funktionalitet under run-time
    b. Simpel implementering
    c. Context indeholder explicit contructor (tager imod strategy som parameter)
    d. Context benyttes til at se ændringer i adfærd
5. **GoF Strategy eksempel:**
    a. Se disposition
    b. Interface ICalculate ***(Calculate(value1 int, value2 int) : int)***
    c. Minus implementere ICalculate
    d. Plus implementere ICalculate
    e. CalculateClient har en Dependency til ICalculate
    f. KOM EVT. MED KODE EKSEMPEL?
6. **Redegør for GoF strategy og godt software design**
    a. Mulighed for ændring funktionaliteten af objekter.
    b. Nem genbrug af kode
        i. nem udvidelse af design
    c. Nem tilføjelse af nye klasser (pga. interface)
7. **Redegør for hvilke SOLID-principper der er overholdt af Strategy Pattern**
    a. OPC (open-close)
        i. Lukket for ændringer, åbent for udvidelse
    b. SRP (Single responsibility)
        i. Konteksten har kun til opgave at se på ændringer
    c. DIP (Dependency inversion)
        i. Kontekst er ligeglad med klasserne (interface)

1. Strategy pattern + Template Method pattern

Redegør for, hvad et software design pattern er.

Et software design pattern er en generel 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.

Sammenlign de to design patterns GoF Strategy og Template method

Strategy og template forsøger begge at løse problemet at adskille en algoritme fra dens implementation.

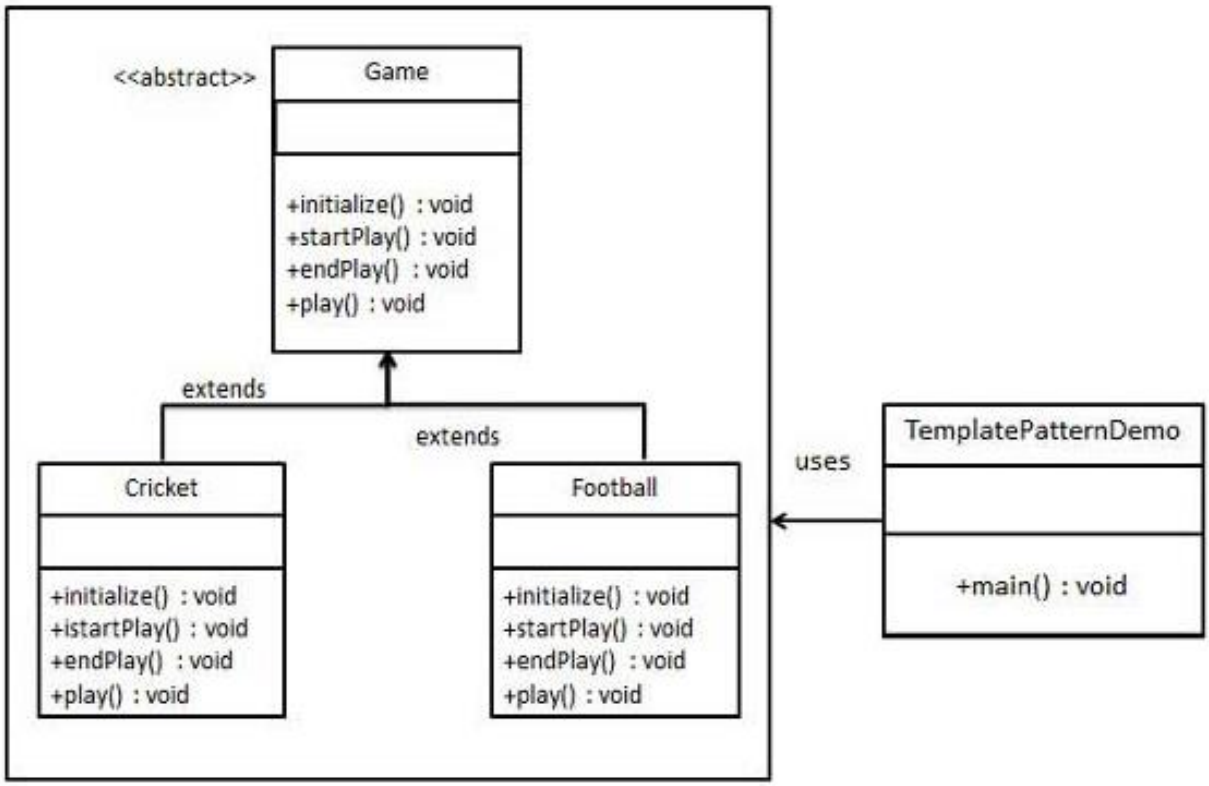
Template Method bruger **arv**

Stategy pattern bruger **delegation**, sådan der kan skiftes ud under run-time.

Template method pattern: **compile-time** algorithm selection by **subclassing**
Strategy pattern: **run-time algorithm** selection by **containment**

Template

Template forsøger at løse problemet med arv ved at lave en abstract klasse som indeholder protectede versioner af de funktioner der skal udføres samt en funktion der kalder dem i den ønskede rækkefølge. Template method bruges til at definerer en struktur / algoritme ved at give nogle metoder til sine subklasser, og lader disse subklasser omskrive funktionaliteten fra base klassen uden at ændre i base klassens struktur.



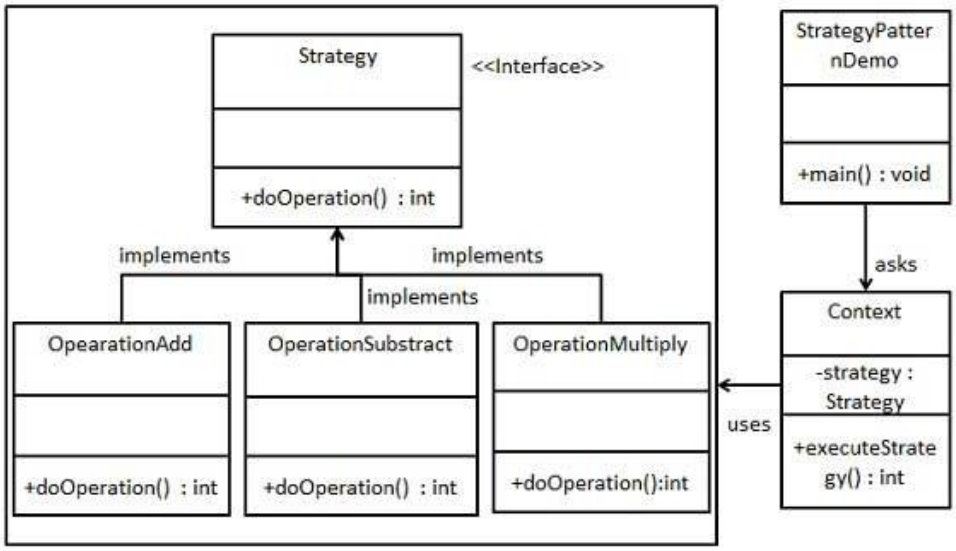

```
public class TemplatePatternDemo {
   public static void main(String[] args) {

      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
}
```

Template bruges hvis konteksten skal kende til implementeringen af de arvende klasser. Fx hvis man skal tilgå et state for at tjekke på klassen. Dette er ikke muligt med strategy pattern da klassen der bruges kan skiftes ud – og staten kan forsvinde.

Man bruger ofte template method pattern, når man har brug for at kende til den interne struktur af klassen. Der bliver også brugt Template Method pattern når man ikke er interesseret i at skifte ud i funktionaliteten under run-time.

Strategy

Indkapsler funktionalitet i et interface så det kan skiftes ud under run-time.

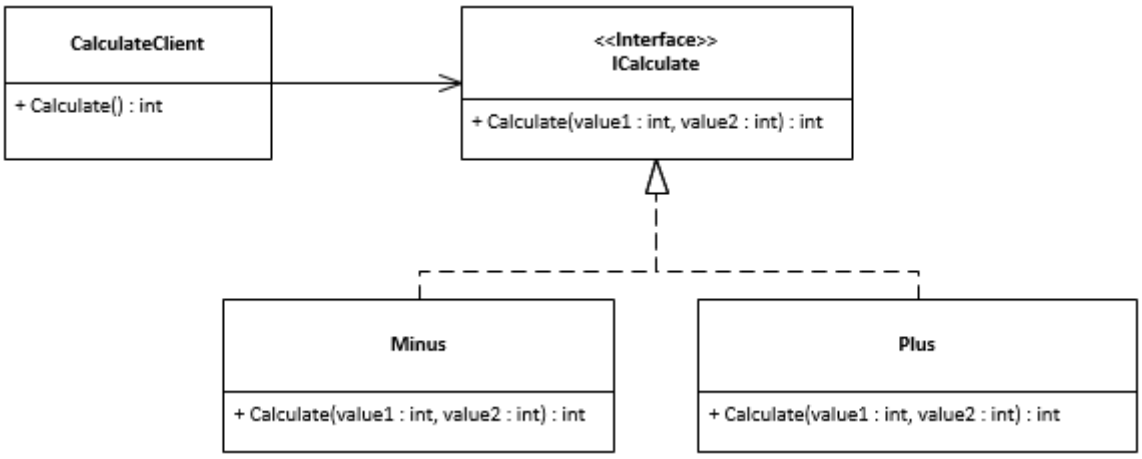


Hvornår vil vi bruge det ene frem for det andet?

Hvis det er nødvendigt at kunne **skifte den implementering** det ønskes at bruges under run-time må vi bruge strategy da template ikke kan skiftes under run-time (interface). Template er til gengæld mere simpelt at implementere.

I sin **context** klasse har en explicit contructor, som tager imod en strategy som parameter. Context klassen bruges til at se ændringer i adfærden når en strategy ændres og en executeStrategy, som vælger en metode udfra den strategy som er blevet valgt, som kunne være **OperationAdd** eller **OperationMultiply.**

GoF Strategy eksempel



Andet eksempel, med implementering taget fra: Taget fra
https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

Create *Context* Class.

Context.java

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

Altså sætter vi i vores Contexts klasses Constructor, hvilken af de konkrete strategier der skal bruges, udfra hvilken strategy der gives med som parameter.

Use the *Context* to see change in behaviour when it changes its *Strategy*.

StrategyPatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

I vores main opretter vi så en strategy af type IStrategy, og giver det med til Contexten. Så kalder vi en metode i konteksten, som så kalder strategiens metode

Redegør for, hvordan anvendelsen af GoF Strategy fremmer godt software design

Man er i stand til på run-time at ændre funktionaliteten, af objekter. Dette gør at man kan genbruge kode, og nemt udvide et design med mere funktionalitet. Da man bruger et interface til at implementere funktionaliteten, er man i stand til bare et tilføje en ny klasse der implementere dette interface.

Redegør for, hvilke(t) SOLID-princip(per) du mener anvendelsen af GoF Strategy understøtter

- Open-Close (OPC)
    - Open close fordi det er lukket for ændringer med åbent for udvidelser
- Single Responsebility (SRP)
    - Konteksten har kun til opgave hvad der skal ske men ikke hvordan det skal ske
- Dependency Inversion (DIP)
    - Konteksten er ligeglad med klasserne, den bruger et interface.

```
public class Program
{
    public static void Main()
    {
```

```
        CalculateClient client = new CalculateClient(new Minus());

        Console.WriteLine("Minus: " + client.Calculate(7, 1));

        //Change the strategy
        client.Strategy = new Plus();

        Console.WriteLine("Plus: " + client.Calculate(7, 1));
    }
}


//The client
public class CalculateClient
{
    public ICalculate Strategy { get; set; }

    public CalculateClient(ICalculate strategy)
    {
        Strategy = strategy;
    }

    //Executes the strategy
    public int Calculate(int value1, int value2)
    {
        return Strategy.Calculate(value1, value2);
    }
}
```

Inheritance vs Delegation