**Design patterns:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**INTRO:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Vi går igenom tre stycken, men det finns många fler.

* **FACADE PATTERN**
* **FACTORY PATTERN**
* **ADAPTER**

**Exempel:**

1. **Singelton**
   1. Gör så man bara kan ha en instans av en klass:

**EXEMPEL:**

* + - Om man ska ha en konfiguration, eller en räknare

1. **MVC:**
2. **Dependency injektion:**
3. **Interface:** Hur man gör en klass som bara innehåller metoddeklarationer och ev. attribut

Design patterns kan vara låst till ett visst ramverk eller språk

**STANDARDMÖNSTER:**

* Hur kan man lösa ett visst typ av problem utan att behöva berätta så mycket
* Kan vara en klass eller ett kluster av klassen
* **Är inte en färdig lösning bara ett sätt att beskriva hur man skulle kunna strukturera sitt problem utan att gå in i ett hörn**
* **Visual studio:** Utvecklingsmiljön kan ibland generera upp den struktur man vill ha till sitt pattern

**GRUPPERINGAR AV MÖNSTER:**

1. **PROGRAMMERINGSPATTERN** (det vi gör idag)
2. **ARKITEKTURPATTERN:**
   1. (typiskt är Microservices)
   2. CLIENT/SERVER

**POÄNGEN MED PATTERNS:**

* Återanvändning av kod:
  + NAMNGIVNING:Är jätteviktigt!
* Deklarera vilket pattern man använder
* **KISS-principen:** Keep It Simple (gör det inte svårare än vad det är)

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**GRUPPERING 1:  
CREATIONAL – Hur vi skapar ett nytt objekt?**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* **HANDLAR** **OM:**
  + Sättet sina objekt skapas på
  + Hur vi skapar ett nytt objekt
    - **Exempel:** Dependency injection, Factory
* **MÅLET:** Att minska beroendet mellan klasser

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**GRUPPERING 2:   
STRUCTURAL – Hur klasser samverkar?**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* **HANDLAR** **OM:**
  + Ofta kopplat till mer arkitektur situationer:
    - Beroende på hur jag gör mitt system väljer jag olika struktur mönster
    - EX: Har jag valt ett system använd inte det här pattern
  + EX: Observer mönstret
* **MÅLET:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**GRUPPERING 3:  
BEHAVIORAL – Hur objektet beter sig?**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* **HANDLAR** **OM:**
  + Vilken roll objektet har i förhållande till andra objekt
    - **Exempel:**
* **MÅLET:**

**HISTORIA:**

**GOFF-mönster:**

* 23 original mönster från boken Design patterns
* Nya mönster går att ofta härleda till de andra 23
* LÄR OSS DESSA MÖNSTER 🡪 Studera mönstren som finns

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Design pattern 1:  
FACADE: (struktur mönster – Hur ett objekt beter sig)**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Bra att använda när vi pratar om olika skikt 🡪 Vill dölja vad som händer bakom (REST interfacet gör man oftast inget i, utan man döljer)

* Sällan vi gör något i REST interfacet

**EXEMPEL:**

* Kan dölja ett asynkront anrop om man har en REST fasad
* Vill dölja upp applikationen i lager: Affärslogik, etc.
* DOMANI DRIVEN DEVELOPMENT: Anti corruption layer döljs ofta då med en fasad
* MVC: View är en fasad mot modellen. I ett MVC mönster har man ett fasad mönster (kommer se ett till senare som kopplar ihop det)

**OVE OFTA:**  
- Hur man koppla ihop ett gäng anrop till ett, man döljer de fyra anropen bakom en fasad för att få det mer effektivt/snyggt i gränssnitten

Istället för att jag som användare göra fyra anrop till sevicer varje gång jag vill till db, döljer jag det till ett

**HUR SER FACADE UT?**

1. Har någon form av klient (någon som anropar, en app, browser)
2. Sätter en Fasad framför (den man kan anropa)
   1. I den: Vad är det för klasser vi ska anropa?
   2. Sen returnerar den
3. EXEMPEL: REST: Gör en post in till servicen som i sin tur skapar massa olika typer av klasser, json (perfekt exempel på fasad)

**EXEMPEL UML:**

1. Man har en eller flera klienter som anropar något – DoSomethin()/MetodNamn() som beskriver vad den faktiskt gör (finns inget bindande på metodnamnet)
2. **DoSomethin()**: När jag får in något skapar jag upp tre klassen

**KORTLEK:** Är bara en fasad till vilka kort vi ska ta

**KOD FACADE:**

* **VAD VI HAR:**

1. **Subsystem** (man implementerat med olika operationer):
   1. **Operationer:** (subsystemet innehåller olika operationer)
2. Skapar upp instanser av subsystemet och så väljer man vilken operation
3. **ORDNINGEN KAN VARA VIKTIG INTERNT MEN INTE EXTERNT**

**EXEMPEL:**

**Class SubsystemA**

1. **OperationA1**
2. **OperationA2**

**Class SubsystemB**

1. **OperationB1**
2. **OperationB2**

**Class Facade.cs**

1. **INSTANS1: subsystemA**
2. **INSTANS2: subsystemB**
3. **OPERATION1:**
   1. **subsystemA.OperationA1**
   2. **subsystemB.OperationB1**
4. **OPERATION2:**
   1. **subsystemA.OperationA2**
   2. **subsystemA.OperationB2**

**SAMMANFATTNING:**

1. Man följer det som händer i bakgrunden och komponerar ett recept av det man behöver göra
2. **ENKAPSULERING:** Alla strukturmönster är till för att göra någon typ av enkapsulering. Vill dölja det som händer bakom och göra det lättare för användarna att anropa det
3. **UTÅTRIKTAD VS. INTERN FASAD:** Fasad behöver inte alltid vara utåtriktad, kan vara interna fasader emellan två stycken subsystem (främst när man skapar ordning och struktur).
4. **NÄR MAN MÅSTE GÖRA SAKER I VISS ORDNIG:** Om det är viktigt att man gör saker i en viss ordning så dölj det i en fasad. Det ger mer möjlighet till att ändra systemet under

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Design pattern 2:  
FACTORY: (Creational mönster – Hur vi skapar ett nytt objekt?)**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**VARFÖR:**

* Bestämmer hur man skapar olika typer av objekt
  + **EXEMPEL:**

Vi kanske inte vet när vi kommer in vad det är för typ av objekt vi ska instansiera.

* + - MVC:
    1. Vad är det vi får in? REST-anrop?
    2. Vad behöver vi skapa efteråt när vi fått in det?
    3. Man använder olika typer av fabriker för att veta vilken klass…?

**EXEMPEL:**

När man inte vet hur man ska kommunicera med motparten.

* Man kan fråga: Kan vi kommunicera på det här sättet?
  + JA: Bra då skapar vi upp en klient av den här typen.
  + NEJ: Om den andra inte stöder version 2.0 så får man använda 1.0 istället och pratar man på det kommunikationsprotokollet

**GENERELLT:**

All nätverkskommunikation bygger på att man har ett factory där man först frågar:

1. FRÅGA: Vad kan du?
   1. När man förhandlat det använder man sin network factory för att bygga rätt typ av protokoll
2. I factory klassen abstraherar man och gömmer skapandet av objektet så att man i runtime läge kan välja vilken typ av objekt som ska skapas.

**EXEMPEL UML - KLASSDIAGRAM:**

CREATOR: Ex, en abstrakt metod (du ska kunna göra det, ex skapa ett objekt av en typ). Skapar en produkt av en viss typ. Genom sin implementation av den konkreta …

**EXEMPEL UML – SPEL (Maze):**

Man vet inte vad det är för typ av rum, finns massa rum men beroende på spelet måste jag kunna skapa rum.

**LÖSNING:** Man säger att man bara ska kunna skapa rummet och göra något med det

**FACTORYN:**

* + - Med factoryn säger man:
      * MAGIC ROOM: Jag kommer implementera ett MagicRum
        + Implementerar rummet som är av definitionen av interfacet
      * LÄRAR ROOM: Kan man fortfarande ha samma spelmotor
        + Det ända jag behöver göra är att ta den som ny factory.

**KOD FACTORY:**

1. **ABSTRAKT KLASS FÖR Room**
2. **KONKRET IMPLEMENTERING AV MAGISKA RUMMET, MagicRoom**
3. **ANNAN KONKRET IMPLEMENTERING AV VANLIGA RUMMET, VanligtRoom**
4. **ABSTRAKT KLASS FÖR MAZEGAME**
   1. Innehåller en lista av rum
   2. När man skapar den säger man: Jag vill att du göra två rum till mig och kopplar ihoprum1 och rum2
      1. **ABSTRAKT METOD: MakeRoom()**
         * 1. Måste implementeras när jag gör mitt MazeGame
5. **KONKRET MAGICROOM:**
   1. **Skapa MagicMazeRoom():** Är själva factoryMetoden
6. **KONKRET ORDINARYROOM:**
   1. **Skapa OrdinaryMazeRoom():** Är själva factoryMetoden

**SKILLNAD:**

* + - Skillnaden är var jag väljer att använda mig av för typ av implementation

**BRA VID TILLFÄLLEN SOM:**

* När man hart olika varianter av samma typ
* **VANLIGT VID:** Integrationstester. All kommunication mockar man. Bara genom att ändra i konfigurationen vilken typ av man vill använda, riktig db eller mock. Bygger på factory patter. Berättar hur du vill använda det
* **OVE OFTA:** Kunna kommunicera med olika typer av personer.
  + 2 olika typer av klient implementationer för att veta vilken

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Design pattern 3:  
ADAPTER: (Structural pattern – Hur klasser samverkar?)**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**VARFÖR:**

* Hanterar konvertering av ett interface till ett annat interface
* En adapter låter klasser samarbeta med varandra som annars inte är kompatibla
* Gör så att två som inte egentligen kan kommunicera kan det

Det man gör är att konvertera från en typ till en annan.

**EXEMPEL:**

* Får in något som man inte kan hantera och gör om det till ett annat för att kunna hantera det
* Bra att använda för att få mindre lera i systemet

**UML EXEMPEL:**

CLIENT: Pratar med ett mål (target) (request)

REQUESTEN: Hanteras av adaptern

ADAPTERN:

* Implementerar själva request anropet
* Konverterar det som kommit in i requesten till en annan specifik TYP AV REQUEST

**TARGET:** Vem är det vi siktar på?

ADAPTER: Vem är det som utför saker?  
ADAPTEE: Vad är det vi översätter till?

**EXEMPEL – MEDIA PLAYER:**

Vet att den ska kunna spela någonting. Vi vet vilken fil och vad den ska spela

**AVANCERADE SPELAREN:**

* Kan hantera två typer (vlc & …)
* Tar ett filnamn

IMPLEMENTERAR VLC SPELARE: Kan jag spela Vlc, men jag kan inte hantera mp4, kommer bara kasta exception. Alltså måste ha en MP4 spelare.

**SKAPAR EN ADAPTER EMELLAN FÖR ATT HANTER VLC OCH MP4:**

* Två implementationer som kan hantera olika format
* **KONSTRUKTORN:** Bestäms vilken typ i konstruktorn
  + Kräver att få in vilken mediatyp vi ska spela (Baseras på vad vi sen skickar in Vlc/mp4)
    - Bestämmer vilken av de konkreta spelarna som ska spelas
* **TRYCKER PÅ PLAY:** Filtyp och filmnamn, kollar kan jag det?
  + Ja det kan jag
    - Gör det den ska

Baserat på vilken filtyp som kommer in vilken av de konkreta spelarna som väljs

**MÅSTE OCKSÅ HA SJÄLVA AUDIOPLAYERN:**

* + - 1. **Interface MediaPlayer**
         1. Play(string AudioType, string fileName)
      2. **AdvancedMediaPlayer:**
         1. PlayVlc(string fileName)
         2. PlayMp4(string fileName)
      3. **Konkrek VlcPlayer : AdvancedMediaPlayer**
         1. PlayVlc(string fileName)

Spelar Vlc

* + - * 1. PlayMp4(string fileName)

INGET

* + - 1. **Konkret Mp4Player : AdvancedMediaPlayer**
      2. **Konkret MediaAdapter: MediaPlayer**
         1. Instans av AdvancedMediaPlayer
         2. Konstruktor som hanterar vilken typ som kom in
         3. Play()

Instans.PlayVlc()

Instans.PlayMp4()

* + - 1. **AudioPlayer:** Kan hantera allt

**ANVÄNDARE/ADAPTER:**

* Vill jag inte veta om vlc/mp4 (bara kunna välja)
* I player skickar jag bara in den typ jag vill ha
  + MP3: Inbyggda spelaren
  + MP4: ADAPTER som gör till rätt typ
  + VLC: ADAPTER
* PLAY: Det väljs sen i systemet om det blir den inbyggda eller implementationen
  + För användaren ser det ut som att den tar alla format
  + Behöver bara veta:
    - Filtyp
    - Filnamn
      * Sen väljer min implementation rätt tolkare
* **ADAPTER:** Har skapat två konkreta spelare att välja på (adaptern skapar kopplingen emellan)

**Ska alltid se till att OM INGEN AV KATEGORIERAR som jag har stöder det som kommer in skickas ett exception**

**FACTORY PATTERN VS ADAPTER:**

* **FACTORY:** Beroende vad för typ jag får in så väljer jag den factory:
  + Om spelare som hade underfunktionalitet
  + Returnerar alltid ett objekt som någon annan använder sig av
* **ADAPTER:** Är till för konvertering
  + Om man MÅSTE ta den fil som kommer in och ska använda den så gör man en adapter
  + Om jag bara kan ta mp3 filer, så måste jag konvertera det
  + Returnerar ett form av resultat
  + Kontakt med databaser: Hur pratar jag med en ODBC databas? Översätter ett protokoll som är programmeringspråk till databasen. Typiskt sätt man använder adapter
  + Ofta när man pratar med andra system:
    - EXEMPEL: Person som ska ges till HR-systemet, Ekonomisystemet: Lägger in en adapter bakom som anpassar till det som det andra systemet måste vara i
    - EXEMPEL: Loggar: Logstatement:
      * 1. Logga detta
        + För att kunna logga och visa någonstans så måste man göra adapeter-event:

TILL EXEMPEL: Consolen, en fil, eller skickar som httpanrop till Azures interna logg

Jobbigt om jag alltid behövde veta hur eller var jag skulle logga. Man lägger en loggadapter emellan för att fånga upp

* + - * **VARFÖR ADAPTER:**
        + Jag vet HUR
        + Jag vet inte VART jag ska logga (en fil etc)

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Design pattern 4:  
STRATEGY PATTERN: (Behavioural pattern – Hur objektet beter sig?)**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**DEFINITION:**

Definierar en familj av olika algoritmer som var och en för sig kan användas oberoende av varandra. En strategi tillåter att klienten ändrar på ex. en resultatalgoritm utan att behöva ändra på den anropade koden

**EXEMPEL PÅ STRATEGY PATTERN:**

* Sorteringsalgoritmer
* Är ett beteende mönster: Är till för att välja vilken typ av algoritm man vill använda
* Helt oberoende av implementationen
  + Tillåter att en klient använder olika typer av resultatalgoritmer utan att behöva kompilera om. EXEMPEL SORTERINGSALGORITMER
    - Väljer olika sorteringsalgortimer beroende på hur många stjärnor som ska tillgås
      * Parallell sorteringsalgoritm
      * Liten maskin:
        + Låter implementationen av .NET schemat välja den algoritm som är mest effektiv
        + Det är ett exempel på strategier

**EXEMPEL STRATEGY:**

**CONTEXT**:

* Vad är det jag vill göra?
* Vad vill jag utföra?
* EXEMPEL: Algoritmet
  + AlgoritmInterface()
* Injektion: Här väljer jag vilken strategy (depency injection mönstret): I RUMTIME VET JAG VILKEN STARTEGI, INTE I KODEN

**STARTEGI INTERFACE**:

* Vad är det för någon typ av algoritm jag kommer använda mig av?

**KONKREAT STRATEGI IMPLEMETATIONE:** Här implementeras algoritmen på olika sätt

* KonkretStrategyA()
* KonkretStrategyB()
* Implementerar interfacet av Strategy() och baserat på vilken implementation så gör jag olika saker

**ANVÄNDNIG:**

1. SKpar context
2. Väljer strategi, EX StartegiA
3. Gör samma sak igen om jag vill testa andra strategier
4. Skriver ut de olika texterna
5. Olika strategier kan ta olika lång tid och då testar man för att se vilken som verkar mest effektiv med tid
6. Oliak system vet man vill ha olika typer av strategier

**INLÄMNINGNE:**

* En bana kör en match och kanske vill¨
* Har en anläggning, så har man en banservice
  + Blir mer som ett interface, mock banservice som jag injektserar in för att köra en match som är mockad, för att kunna förutsäga vad resultatet blir
  + ADAPTER: Simulerad bana eller

CUP:  
Match vinns om 3 set och högsta, men om han byter till att den som vunnit flest set så är det så man vinner. Då kan man använda det här pattern istället för if-satser. Ibland är det på snittpoäng på tre och ibland christmas challange, mest strikes

* Typiskt för startgei mönster. Beroende på startegi så väljer man olike. Blir lite som en adapter samtidigt som man väljer
* A

**INLÄMNINGEN:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**LEKTION: Design Patterns Onsdag 3/10-2018**

**BUILDER**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Separerar konstruktionen på ett komplext objekt från dens represenation

* Gör så det går att skapa olika typer av represenationer

**Participants**

Klasser & Objekt i Builder:

* **Builer (FordonBuilder)**
  + Specificerar ett abstrakt interface för att skapa delar av Produkten (objektet)
* **ConcreteBuilder (MotorcykelBuilder, CarBuilder)**
  + Skapar delar av produkten genom att implementera BuilderInterfacet
* **Director (Shop)**
  + Kontruerar ett objekt med interface
* **Product (Fordon)**
  + Representerar objektet…

**BRA FÖR ATT:**

* Blir för många kont

**RESULTATET:**

**Product Parts1:**

* PartA
* PartB

**Product Parts2:**

* PartX
* PartY

**BRA EXEMPEL:**

---------------------------  
Vehicle Type: Scooter  
 Frame  : Scooter Frame  
 Engine : none  
 #Wheels: 2  
 #Doors : 0  
  
---------------------------  
Vehicle Type: Car  
 Frame  : Car Frame  
 Engine : 2500 cc  
 #Wheels: 4  
 #Doors : 4  
  
---------------------------  
Vehicle Type: MotorCycle  
 Frame  : MotorCycle Frame  
 Engine : 500 cc  
 #Wheels: 2  
 #Doors : 0