|  |
| --- |
| Åbo Akademi |
| Vilka problem kan designmönster lösa? En analys av designmönster i programvaruutveckling |
| Fredrik Holmnäs |

|  |
| --- |
| Fredrik Holmnäs2022 |

# Referat

När man arbetar med programvaruutveckling påträffar man återkommande problem angående design och implementation. I stället för att skräddarsy en unik lösning varje gång man arbetar med dessa problem så kan man tillämpa designmönster som däremot beskriver en generell lösning. Designmönster baserar sig på etablerade designprinciper som berättar hur man gör för att hålla designen hos ett system flexibel och begriplig för andra. Användningen av designmönster brukar förenkla utvecklingsprocessen och leda till att systemet man utvecklar blir mera lätt att upprätthålla, vilket besparar tid och resurser. Dessutom förenklar designmönster kommunikationen mellan utvecklare som känner till dem, eftersom man effektivt kan beskriva sina idéer beträffande den design som man planerar genom att hänvisa till ett relaterat designmönster. Således utgör designmönster ett kraftfullt verktyg som är användbart för alla slags utvecklare.

Denna avhandling kommer att fokusera sig på objektorienterade designmönster som är relevanta för objektorienterad programmering. Dessa mönster beskriver vilka egenskaper som objekten borde innefatta och hur objekten ska bete sig sinsemellan för att uppnå en robust design. Avhandlingens avsikt är att presentera och analysera ett antal av de vanligaste objektorienterade designmönstren samt granska förknippade tillämpningsområden, begränsningar och andra väsentligheter som framgår ur litteraturen.

Innehållsförteckning

[1 Inledning 4](#_Toc99473924)

[2 Kategorisering och representation av objektorienterade designmönster 6](#_Toc99473925)

[3 Skapandemönster 7](#_Toc99473926)

[3.1 Fabriksmetoden 7](#_Toc99473927)

[3.2 Abstrakt fabrik 9](#_Toc99473928)

[4 Strukturmönster 12](#_Toc99473929)

[4.1 Fasad 12](#_Toc99473930)

[4.2 Adapter 13](#_Toc99473931)

[5 Beteendemönster 17](#_Toc99473932)

[5.1 Strategi 17](#_Toc99473933)

[5.1.1 Användning av klassarv och dess begränsningar 17](#_Toc99473934)

[5.1.2 Tillämpning av strategi och dess fördelar 19](#_Toc99473935)

[6 Sammanfattning och diskussion 22](#_Toc99473936)

[Referenser 24](#_Toc99473937)

# 1 Inledning

Inom programvaruutveckling och design återkommer särskilda problem regelbundet. Förvisso kan dessa problem lösas på flera olika sätt, men faktum är att det finns vissa bättre tillvägagångssätt än andra. Det som är avgörande är att det inom programvaruutveckling finns enbart en sak som är konstant, nämligen förändring [1], [2]. Vad det här betyder i praktiken är att man som utvecklare utan tvekan i något skede blir tvungen att göra förändringar i sin programkod och design. Det här kan bero på att systemet behöver ny funktionalitet eller bör modifieras för att upprätthålla sin prestanda. Det finns således hur många skäl som helst till att göra förändringar. Beroende på den lösning eller samling av lösningar som valts för att bygga det initiala systemet, så kan ett krav på förändring lösas med antingen så lite som en ändring i en enda kodrad eller, om den ursprungliga lösningen inte är skalbar, leda till att systemet måste utvecklas på nytt från början för att satisfiera de nya kraven. Således är det synnerligen viktigt att man väljer en god lösning för att spara tid, resurser och kapital i samband med utveckling och upprätthållning av systemet.

Idén med designmönster är att presentera en sådan lösning som går att tillämpa på dessa tidigare omnämnda problem, fastän enstaka problem som ett mönster försöker lösa kan skilja sig från varandra till en viss grad. Det här är möjligt eftersom designmönster inte beskriver konkreta implementationer av lösningen, utan snarare utgör en modell som konceptuellt avbildar en abstraktion av problemet som man vill lösa och dess generella lösning. Designmönster bygger på vedertagna designprinciper och sålunda försöker de garantera att det är möjligt att göra förändringar i systemet så smärtfritt som möjligt. Gamma et al. [3] definierar designmönstren att karakteriseras av fyra huvudegenskaper: mönstrets namn, problemet som mönstret löser, den generella lösningen och slutligen vilka konsekvenser som användningen av mönstret medför. Designmönstrets konsekvenser är betydelsefulla eftersom inget designmönster är perfekt, utan det finns så gott som alltid både för- och nackdelar med att använda ett särskilt designmönster vilket blir avgörande när man planerar designen hos ett system.

En annan fördel med designmönster är att de tillåter utvecklare att kommunicera på ett effektivt sätt om designen utan att dyka in i detaljer. Om man känner till designmönster och de termer som anknyter till dem, så kan man enkelt förmedla idéer om en design och få med andra utvecklare i sitt resonemang utan långa föredrag och förklarningar. Det här är tack vare att designmönster rör sig på en högre abstraktionsnivå än konkreta implementationer som typiskt innefattar en större mängd detaljer. Dessutom torde det här även minimera risken för missförstånd angående designen, vilket besparar tid och frustration hos utvecklarna. Som utvecklare är det därför nyttigt att känna till de vanligaste designmönstren i sitt område för att kunna uttrycka sina ståndpunkter och förstå vad andra utvecklare menar. [1]

Designmönster är ingen teknik som är exklusiv för programmering och programvaruutveckling. Gamma et al [3] citerar arkitekten Christopher Alexander när de definierar vad ett designmönster är. Alltså finns det samband i denna problemlösningsteknik som sträcker sig över flera olika branscher. Det som det i grunden handlar om är att på basen av erfarenhet kunna identifiera ett problem som motsvaras av ett designmönster och följaktligen applicera mönstret på ett lämpligt sätt i enlighet med vad den rådande kontexten tillåter.

Inom datateknik finns det massor av delområden som alla har sina egna variationer på designmönster eller någon motsvarande problemlösningsteknik. Exempelvis för olika programmeringsparadigm, som definierar egenskaper och funktionaliteten hos ett programmeringsspråk, används olika mängder av designmönster. Även maskininlärning, som är ett snabbt växande delområde, har också sin egen uppsättning av designmönster för dess unika problem. Emellertid så finns det en partiell överlappning hos de flesta samlingar av designmönster, eftersom nästan alla designmönster bygger på gemensamma designprinciper. Således så är det enklare att lära sig ett nytt områdes designmönster om man redan är bekant med en samling av designmönster.

I denna avhandling kommer det i första hand behandlas objektorienterade designmönster. Objektorienterad programmering är ett programmeringsparadigm där man bygger och representerar olika entiteter och företeelser med hjälp av objekt som definieras av klasser. Genom objektens tillstånd, beteende och interaktion med varandra så uppfylls den funktionalitet som man önskar implementera. Objektorienterade designmönster beskriver hur dessa objekt ska se ut och hur de ska interagera med varandra för att göra systemet flexibelt och lättare att upprätthålla [3]. Dessutom gör designmönstren källkoden mera överskådlig och mottaglig för andra utvecklare som läser koden, särskilt om man uttryckligen påpekar vilka designmönster som används och vilken roll de olika klasserna har för designen.

Avsikten med avhandlingen är att göra nedslag i ett antal typiska designmönster inom objektorienterad programmering och analysera deras karakteristik som tidigare beskrivits. Utöver detta granskas också andra väsentligheter såsom tillämpningar i praktiken, specialisering av mönster, kombination av mönster, problematik vid implementationer och annat som framgår ur litteraturanalysen av källorna.

# 2 Kategorisering och representation av objektorienterade designmönster

Objektorienterade designmönster delas vanligen upp i tre huvudkategorier: skapandemönster, strukturmönster och beteendemönster [3]. Mönster inom samma kategori har gemensamma egenskaper och löser problem av liknande natur. Därmed går det ibland att välja mellan olika mönster från en kategori för att lösa ett och samma problem. Valet ska ändå göras med eftertanke eftersom vissa mönster passar in bättre i designen hos det system som utvecklas än andra. I de följande kapitlen kommer de olika kategorierna kort presenteras tillsammans med ett fåtal tillhörande designmönster.

För att beskriva och representera designmönster inom objektorienterad programmering används ofta UML-diagram (unified modelling language). Dessa diagram beskriver effektivt vilka klasser och relationer mellan klasserna som är väsentliga för ett designmönster. Figur 1 visar hur ett UML-diagram kan se ut där man modellerar en representation av ankor. Även andra detaljer såsom gränssnitt, som bestämmer vilket eller vilka beteenden som ska ingå i konkreta implementationer, och multipliciteten kan framgå ur ett UML-diagram. Ibland påträffar man variationer av syntaxen som används i UML-diagram, men utgående från kontexten borde man ändå kunna få en klar uppfattning om vilken idé som diagrammet försöker förmedla. I kapitlen om specifika designmönster kommer UML-diagram att användas för att illustrera och exemplifiera de olika designmönstren.



Figur 1. Exempel på ett UML-diagram [1, s. 2, översatt från engelska]

# 3 Skapandemönster

Designmönster som hör till kategorin skapandemönster är sådana som bestämmer hur objekt skapas och representeras i systemet. Genom att abstrahera instansieringen, det vill säga skapandet at objektet utgående från en klass, så kan man uppnå en mångsidig flexibilitet där man har kontroll över vilka objekt som skapas, deras egenskaper, och i vilka tillfällen detta faktiskt händer [3], [4]. Med hjälp av dessa mönster kan man också lättare hantera kopplingen mellan klasser, alltså hur beroende en klass är av en annan klass. En låg koppling är typiskt önskvärd eftersom det höjer flexibiliteten, men emellertid så är det här inte alltid möjligt och beror på kontexten av systemet. I det här kapitlet kommer mönstren fabriksmetoden och abstrakt fabrik att behandlas.

## 3.1 Fabriksmetoden

Fabriksmetoden är ett designmönster där man skapar en skild klass som ansvarar för att skapa andra objekt av en särskild typ. I vanliga fall så använder man sig av ett nyckelord såsom ”new” i programmeringsspråket Java för att instansiera ett nytt objekt, men det här sättet har ett flertal begränsningar. Exempelvis om man vill ha en sådan design där objektens typ avgörs av i vilken ordning de skapas, så måste man implementera en skild logik för att veta vilken typ av objekt som ska instansieras nästa gång. Om man då använder en skild klass för att implementera denna logik samt instansieringen av objekten så förenklas processen avsevärt. Den här klassen som ansvarar för att skapa andra objekt brukar kallas fabrik. [1], [3]

Mönstrets funktionalitet utvidgas nästan alltid genom att skapa en abstrakt fabriksklass eller ett gränssnitt med en så kallad fabriksmetod som även gett namn till själva mönstret. Fabriksmetoden returnerar objekt av önskad typ enligt den logik som definieras av konkreta fabriksklasser som implementerar den ovanliggande fabriksabstraktionen. Tack vare att man programmerar mot abstraktionen av fabriken så kan man vid behov byta ut den konkreta implementationen utan att resten av systemet måste ändras som följd, vilket erbjuder flexibilitet och uppfyller flera av designprinciperna. [1]

Fabriksklassen kan också implementeras på sådant sätt att man kan konfigurera objekten som skapas med argument. När man instansierar objekt så kan man ge argument beroende på vilka parametrar som finns hos den motsvarande klassen för att skapa mera detaljerade och komplexa objekt. Det här är också möjligt med fabriksklasser om man skriver fabriksmetoden på sådant sätt att den tar emot vederbörande argument och sedan instansierar objektet på basis av argumenten precis som i vanliga fall. Med fabriker har man även möjlighet att inverka på fabrikernas inre tillstånd beroende på vilken logik som används i fabriksmetoden. Detta, som redan tidigare påpekats, erbjuder mera flexibilitet än att enbart direkt instansiera objekt.

Figur 2 illustrerar ett UML-diagram för ett exempel på fabriksmetoden. I det här fallet finns det en abstrakt klass eller gränssnitt vid namn Creator som utgör den abstrakta fabriksklassen. Den konkreta implementationen av denna fabriksklass är ConcreteCreator som även har den egentliga implementationen på själva fabriksmetoden. När fabriksmetoden sedan anropas så skapas och returneras ett konkret objekt benämnt ConcreteProduct av typen Product. Den flexibilitet som uppnås genom detta är att man vid behov kan göra en ny implementation på den abstrakta fabriksklassen så länge som man programmerat mot gränssnittet för fabriken framför en konkret implementation. Det här gäller också för objekten som skapas med hjälp av fabriksmetoden. Det vill säga så kan man göra flera konkreta klasser av typen Product som instansieras hos den fabrik som man väljer.



Figur 2. UML-diagram för ett exempel på fabriksmetoden [3, s. 122]

Fabriksmetoden i sig själv är ett relativt enkelt mönster. Därmed så brukar mönstret specialiseras och anpassas för att passa in i större system och utvidga mönstrets funktionalitet. Waheed et al. [5] undersöker förekomsten av designmönstervarianter genom att analysera ett antal applikationer med öppen källkod. För att identifiera designmönster hos applikationerna används ett flertal verktyg, exempelvis en metod som baserar sig på poängsättning av likheter (similarity scoring) [6]. En av de designmönstervarianter som Waheed et al. upptäckte och benämnt strategisk basfabrik (strategy base factory) är en utvidgning av den vanliga fabriksmetoden. Strategisk basfabrik kombinerar fabriksmetoden med strategimönstret för att göra systemet mera anpassat till sin kontext. Beroende på kontexten väljs en lämplig variant av en algoritm som används hos objekten som skapas av fabriken. Möjligheten att välja bland en familj av algoritmer under exekveringstid hör alltså egentligen till strategimönstret, men tack vare denna kombination så får systemet en bättre anpassningsförmåga. Det här är användbart om det finns olika tillstånd och miljöer för systemet som förutsätter att objekten som ska skapas har ett särskilt beteende för just den kontexten. Valet av beteende kan inverka på systemets beräkningsmässiga prestanda och ibland vara avgörande för om systemet fungerar överhuvudtaget.

## 3.2 Abstrakt fabrik

Abstrakt fabrik är ett designmönster som påminner mycket om fabriksmetoden. Skillnaden är att abstrakt fabrik möjliggör att fabriksobjekten kan instansiera flera olika typer av objekt. Det här skiljer sig alltså från fabriksmetoden där man enbart koncentrerar sig på skapandet av en typ av objekt. Poängen med abstrakt fabrik är att de konkreta fabrikerna kan instansiera objekt som är relaterade till eller beroende av varandra, vilket låter systemet undgå det potentiella problemet med att samtliga objekt är inkompatibla med varandra och därmed inte borde användas tillsammans. Sålunda ansvarar varje konkret abstrakt fabrik för att de objekt som skapas kan samverka på ett logiskt sätt. [1], [3]

Gränssnittet för den abstrakta fabriken definierar vanligtvis ett antal av fabriksmetoder som ansvarar för att skapa de skilda objekten, varefter konkreta implementationer av den abstrakta fabrikens gränssnitt bestämmer hur fabriksmetoderna konstruerar objekten [1]. Skillnaden hos objekt från olika fabriker kan bero på att de utgör olika implementationer av en gemensam objektstyp (illustreras i figur 3) eller helt enkelt att objekten instansieras med unika parametrar som avgör deras egenskaper.

Det är värdefullt att använda abstrakt fabrik till exempel när man vill tillhandahålla olika visuella användargränssnitt för systemets användare [3]. Exempelvis om man ville ha ett mörkt och ett ljust tema så kan man göra varsin konkret abstrakt fabrik som ansvarar för att skapa varje element hos användargränssnittet i enlighet med det aktuella temat. Ur det här exemplet framgår det också varför man vill undvika särskilda kombinationer av objekt. Det skulle nämligen inte vara någon god idé att man skapade en mörk bakgrund tillsammans med en mörk text, vilket blir svårläst och strider mot användarvänligheten. Abstrakta fabriken som används för det mörka temat ska då se till att textfärgen passar ihop med resten av objekten för att bygga upp en funktionell helhet. Ett annat analogt exempel finns hos programmering med hänsyn till operativsystemet som systemet eller applikationen körs på. Med andra ord gör man abstrakta fabriker som motsvarar enskilda operativsystem och skapar objekt som är anpassade för dessa, vilket ofta är kritiskt för att systemet ska behålla sin funktionalitet och prestanda hos det vederbörande operativsystemet.

I figur 3 presenteras användningen av abstrakt fabrik för att instansiera objekt från två olika mängder av användargränssnittskomponenter. Själva gränssnittet för den abstrakta fabriken specificerar inte vilka konkreta typer av objekten som skapas, vilket separerar klienten från de egentliga implementationerna av dessa objekt. På det här sättet inkapslas ansvaret och metoden för att skapa de önskade objekten hos de konkreta fabrikerna [3]. I figuren ansvarar PMWidgetFactory respektive MotifWidgetFactory för att skapa varsin familj av användargränssnittskomponenter, det vill säga programfönster och rullningslister i det här fallet. Med den här designen grupperas de relaterade objekten och instansieringen av dem inkapslas hos fabriksklasserna, vilket gör systemet begripligt och flexibelt.



Figur 3. UML-diagram för ett exempel på en abstrakt fabrik med två konkreta implementationer [3, s. 100]

Budgen [7] undersöker huruvida designmönster verkligen kan hjälpa oss att kommunicera kring design och bygga robusta system. Genom en kartläggningsstudie samt en undersökning av åsikter och erfarenheter hos författarna angående designmönstren så identifierar Budgen en bedömning av de ursprungliga designmönstren presenterade av Gamma et al [3]. Budgen påpekar att hans resultat inte ger oss ett fullständigt svar på om mönstren faktiskt förenklar processen med att skapa och upprätthålla system. Resultaten pekar dock på att samtliga mönster kan leda till att ett systems design blir alltför komplext i relation till det som systemet ska åstadkomma. Ett annat bekymmer som framgår ur resultaten är att en del av mönstren är för specifika och sålunda inte har en god anpassningsbarhet, vilket står i konflikt med grundtanken med designmönster. Emellertid så återfinns inte dessa problem hos abstrakt fabrik och fabriksmetoden i studien. Tvärtom så beskriver Budgen att dessa två mönster understöds nästan enhälligt i studien, vilket torde antyda att de verkligen är användbara och lever upp till sina syften som designmönster.

Dessvärre har även abstrakt fabrik sina begränsningar. Cinnéide et al. [8] beskriver ett problem benämnt ramverksuppgraderingsproblemet (framework upgrade problem) som påträffas inom utvecklingen av industriella system med syfte att satisfiera majoriteten av gemensamma krav för ett systems användare. Enligt Fayad och Schmidt [9] erbjuder sådana här ramverk modularitet, återanvändbarhet samt utvidgningsbarhet. Dessa är goda egenskaper men de förutsätter att ramverket kan uppdateras utan att användarnas anpassningar på ramverket fallerar. Att designa ett sådant ramverk är en krävande uppgift där man kan ha hjälp av designmönster. Vid användningen av abstrakt fabrik händer det sig då ofta att ramverkets användare behöver utvidga eller specialisera den ursprungliga designen hos ramverket för att konstruera önskade objekt. Figur 4 visar ett förenklat exempel där man vill skapa CustomPerson objekt som är en utvidgning av DefaultPerson hos det ursprungliga ramverket, vilket också medför en ny skild fabriksklass. Det ovannämnda problemet uppstår när ramverket uppdateras, varpå förändringar i gränssnittet för objekten som skapas eller deras klienter kan leda till konflikt med anpassningarna som gjorts hos ramverkets användare. Man blir då tvungen att undersöka och korrigera sådana konflikter för att återställa systemets funktionalitet, vilket bryter löftet hos abstrakt fabrik som säger att klienten hålls separerad från de konkreta implementationerna på objekten. Cinnéide et al. lyfter fram att ett lösningsalternativ där man i stället använder ett mönster som utgör en utvidgning av abstrakt fabrik vid namn pluggbar fabrik (pluggable factory) där man kan undgå att ändra på själva fabriksobjektet vid anpassningen av ramverket. Olyckligtvis är det här enbart en kompromiss eftersom pluggbar fabrik har sina egna unika begränsningar. Följaktligen är det här problemet någonting som man är tvungen att leva med när man arbetar med dessa ramverk.



Figur 4. UML-diagram som visar ett exempel för en anpassning (”Custom”) på ett ramverk [8, s. 6]

# 4 Strukturmönster

Ibland är det fördelaktigt eller rentav nödvändigt att bygga större strukturer som består av flera mindre individuella objekt av varierande klasser. Strukturmönster beskriver hur man kan komponera dessa strukturer på ett flexibelt sätt genom att identifiera förhållandena mellan de enskilda modulerna och sammankoppla dem till större helheter med utvidgad funktionalitet. I praktiken realiseras det här typiskt genom klassarv eller komposition.

Klassarv innebär att en klass ärver metoder och variabler från en annan klass, varpå den ärvande klassen även kan bygga vidare med sina egna metoder och variabler eller vid behov överskugga (override) ärvda metoder. Däremot finns det en designprincip benämnd ”komposition framför klassarv” (compostion over inheritance) som säger att det vanligtvis är bättre att använda komposition. Komposition betyder att objekt också innefattar referenser till andra objekt. Fördelen med komposition ligger i att man smidigt kan byta ut vilka objekt som bygger upp den större strukturen även under exekveringstid genom att ändra på referenserna. Emellertid är detta inte är möjligt med klassarv [3], [4]. Objektens kombinerade funktionalitet och tillstånd utgör stommen hos de förutnämnda strukturerna som i sin tur kan fungera som komponenter i större system. I det här kapitlet kommer strukturmönstren fasad och adapter att behandlas.

## 4.1 Fasad

Designmönstret fasad är ett relativt enkelt men användbart mönster som förenklar användningen av subsystem inom ett större system. Subsystem kan bestå av många individuella klasser som behövs för att uppfylla funktionaliteten hos subsystemet. När klienter vill använda någon funktionalitet hos detta subsystem så blir de vanligen tvungna att delegera anrop till de individuella klasserna hos subsystemet. Det vill säga så tvingas klienterna att arbeta med ett flertal gränssnitt hos de olika komponenterna i subsystemet för att kunna använda det. Fasad erbjuder däremot ett enat gränssnitt för hela subsystemet vilket låter klienterna använda subsystemet genom att enbart interagera med gränssnittet hos fasaden. [1]–[3]

Figur 5 illustrerar informellt hur relationerna mellan klienterna och subsystemets komponenter förändras när ett fasadgränssnitt tas i bruk. Utseendet av relationerna mellan klasserna i subsystemet kan variera men ett fasadgränssnitt borde kunna implementeras oavsett hur det underliggande subsystemet ser ut. Om en klient behöver utnyttja många klasser från subsystemet utan fasad kan det bli en stark koppling som för med sig onödig komplexitet vilket lätt undviks med användningen av fasad. En annan fördel med fasad är att de enskilda klasserna och deras gränssnitt hos subsystemet fortfarande är tillgängliga för klienterna. Det är alltså inte tvång på att använda fasadgränssnittet, vilket är praktiskt om en klient vill åstadkomma någonting specifikt i subsystemet vilket enbart kräver funktionaliteten hos en enda komponent.



Figur 5. Diagram som visar hur fasad förenklar användningen av ett subsystem [3, s. 208]

Subsystem kan också byggas upp av mindre subsystem som är tillgängliga med sina egna fasadgränssnitt. På det här sättet kan man naturligt konstruera olika abstraktionsnivåer i ett större system genom att gruppera samverkande beståndsdelar och bilda en hierarki inom systemet [10]. Det här gör systemet mera överskådligt och begripligt jämfört med att granska individuella klasser en åt gången, vilket är särskilt nyttigt för utvecklare som bekantar sig med systemet men inte deltagit i utvecklingen från början och därmed inte känner till varje detalj i systemet. I studien av Waheed et al. [5], som tidigare omtalats i kapitel 3.1, identifieras även en variant av fasad som bygger på denna företeelse som beskrivits ovan. Waheed et al. benämner varianten universell fasad (universal fasad) eftersom tanken är att introducera ett fasadgränssnitt som tillhandahåller alla andra subsystems gränssnitt. Utöver att göra systemet lättare att förstå så hävdar Waheed et al. att detta även förenklar implementationen, testningen, underhållningen och användningen av systemet. Emellertid är en sådan design inte nödvändigtvis realiserbar hos alla system och dessutom kan det ibland vara mera praktiskt att hålla de olika komponenterna isär.

## 4.2 Adapter

Emellanåt kan man stöta på ett sådant problem där en klass vill använda sig av en annan klass men deras gränssnitt är inte kompatibla med varandra. Därmed kan klienten inte anropa metoder direkt från den begärliga klassen eftersom metoderna hos den inte ser ut som förväntat med hänsyn till gränssnitten. Lyckligtvis kan det här hindret enkelt lösas med hjälp av designmönstret adapter som även ibland kallas för omslag (wrapper). Detta uträttas genom att införa en adapterklass som ansvarar för att göra gränssnittet hos den eftersökta klassen tillgängligt genom det gränssnitt som klienten förväntar sig. För att anropa en metod från den begärda klassen anropas i stället den vanliga metoden som klienten känner till hos adapterklassen, varpå adapterklassen i stället delegerar anropet till den motsvarande metoden hos den begärda klassen. På det här sättet kan man förena gränssnitten hos klasser som i vanliga fall är inkompatibla. Detta förutsätter dock att beteendet hos den begärda klassen nöjaktigt motsvarar det som klienten faktiskt vill åstadkomma. [1]–[3]

Figur 6 visar en generalisering av adapter. I figuren är specificRequest() den metod som klienten Client vill använda sig av, men klienten känner bara till att det finns en metod som kallas request() hos gränssnittet Target. Adapterklassen Adapter implementerar gränssnittet hos Target och innehåller en referens till en instans av klassen Adaptee som hyser den önskade metoden. I implementationen av request() hos Adapter anropas i sin tur specificRequest() med hjälp av referensen till Adaptee. Tack vare detta kan klienten nu utnyttja specificRequest() genom att anropa request() hos Adapter utan att känna till det gränssnitt som är definierat hos Adaptee-klassen.



Figur 6. UML-diagram för designen hos objektadapter [1, s. 245, bildtext utelämnad]

I exemplet ovan realiseras mönstret genom att använda komposition. Adapterklassen upprätthåller således en referens till Adaptee-klassen som används för att anropa dess metoder med främmande gränssnitt. Däremot är det här inte det enda alternativet på att förverkliga adapter, utan det är också möjligt genom att använda multipelt klassarv. De två adpatervarianterna kallas objektadapter, som utnyttjar komposition såsom i det tidigare exemplet, och klassadapter som i stället använder multipel klassarv. En generaliserad implementation av klassadapter illustreras i figur 7 på motsvarande sätt såsom i figur 6. Grundidén med klassadapter är den samma men i stället för att anropa specificRequest() med hjälp av en referens till Adaptee-klassen så kan metoden anropas direkt från Adapter tack vare klassarv. Det finns både för- och nackdelar med att använda klassarv för att implementera adaptermönstret. Till exempel kan man vid behov även överskugga metoder som härstammar från Adaptee vilket inte är möjligt med komposition. Däremot kan man använda vilken som helst subklass av Adaptee i adapterklasen om man utnyttjar komposition, vilket i sin tur inte är möjligt med klassarv.



Figur 7. UML-diagram för designen hos klassadapter [1, s. 246, bildtext utelämnad]

Emellertid medför multipelt klassarv en räcka av unika problem som uppstår vid upprepat klassarv. Särskilt besvärligt är ett problem som är känt som diamantproblemet (”diamond problem”) där klasshierarkin som uppstår har formen av en diamantfigur. Detta leder till en växande komplexitet där man blir tvungen att ta ställning till hur man undgår situationer såsom att en klass ärver en gemensam metod med samma namn från två olika klasser som har olika implementationer på metoden. Detta är en av orsakerna till att samtliga programmeringsspråk såsom Java inte tillåter multipelt klassarv, sålunda går det inte att använda klassadapter hos dessa programmeringsspråk (åtminstone med multipelt klassarv). För övrigt föredras objektadapter i stor mån just på grund av de ovannämnda begränsningarna och nackdelarna med multipelt klassarv. Sådant är även fallet hos programmeringsspråk där multipelt klassarv är tillgängligt och klassadapter kan implementeras, såsom hos C++. [1], [11]

I praktiken kan adapter till exempel användas när man arbetar med externa kodbibliotek som erbjuder bland annat färdigskrivna klasser, metoder och variabler. Det händer då ibland att man växlar mellan olika bibliotek om man exempelvis upptäcker någon klass hos ett annat bibliotek som är mera ändamålsenlig för systemets design. Här kan adaptermönstret fungera utmärkt eftersom vid bytet av Adaptee-klassen med hänsyn till de tidigare figurerna så behöver man enbart ändra på adapterklassen för att klienterna ska kunna börja använda den nya klassen. Den här aspekten är mycket relevant eftersom man i mjukvaruutveckling arbetar mycket med att återanvända och tillämpa färdiga bibliotek för att utveckla nya system och applikationer. [1]

Ett annat konkret exempel illustreras i figur 8. I Java finns det två skilda gränssnitt som används för att iterera över elementen hos en datasamling (collection). Äldre varianter på datasamlingar använder typiskt gränssnittet Enumeration medan nyare datasamlingar i stället använder Iterator. Beroende på vilka datasamlingar man arbetar med så kan man hamna i den situationen att man tvingas arbeta med gränssnittet hos Enumeration när man egentligen förväntar sig Iterator. Adapterklassen EnumerationIterator i figuren löser konflikten genom att koppla ihop motsvarande metoder hos bägge gränssnitt, det vill säga delegerar hasNext() ett metodanrop vidare till hasMoreElements() och på motsvarande sätt kopplas next() ihop med nextElement(). Enda problemet är att metoden remove() inte stöds hos Enumeration och därmed kan denna metod inte implementeras hos adapterklassen inom adaptermönstrets ramar. Lyckligtvis kan detta lösas genom att i stället kasta ett undantag (throw an exception) i adapterklassens remove(), vilket vid metodanrop meddelar klienten om felet och låter klienten med sin egen logik bestämma hur problemet ska hanteras. Således överlåts en del av ansvaret av gränssnittsanpassningen till klienten, vilket inte är den egentliga tanken med designmönstret. Alltså är adapter inte heller något perfekt designmönster utan man är tvungen att vara aktsam och göra anpassningar i sitt system för att understödja användningen av mönstret.



Figur 8. UML-diagram för designen hos objektadapter [1, s. 251, bildtext utelämnad]

# 5 Beteendemönster

Till skillnad från skapandemönster och strukturmönster så fokuserar beteendemönster på interaktionen och kommunikationen mellan objekt i stället för deras uppbyggnad. Således handlar beteendemönster mera om algoritmer och uppdelning av olika ansvarsområden mellan objekten i ett system. Det här bygger på enda-ansvarsprincipen (single-responsibility principle) som säger att objekt, metoder och moduler ska i första hand ansvara för att representera endast en enskild företeelse. Följaktligen när man eventuellt har behov av denna företeelse så delegerar man ett anrop eller en förfrågan till modulen som ansvarar för just denna uppgift. Beteendemönster erbjuder sålunda kontroll över flödet av kommunikationen i systemet och försöker garantera att detta görs på ett smidigt och flexibelt sätt, vilket kan förverkligas genom ett flertal olika tillvägagångssätt med hjälp av de olika beteendemönstren [4]. I det här kapitlet så kommer beteendemönstret strategi att behandlas.

## 5.1 Strategi

Designmönstret strategi är ett mönster som kraftigt stöder designprincipen komposition framför klassarv som omtalats i tidigare kapitel. Grundtanken med mönstret är att identifiera och definiera en samling av relaterade algoritmer som förekommer hos ett system och kapsla in dessa individuella algoritmer med till exempel motsvarande klasser. Det här minskar kopplingen mellan algoritmerna och deras klienter samt gör det möjligt för klienterna att välja den algoritm som passar bäst med hänsyn till kontexten och klientens behov. [1]–[3]

### 5.1.1 Användning av klassarv och dess begränsningar

Det är inte omöjligt att åstadkomma designen som beskrivits ovan med hjälp av klassarv utan att utnyttja strategi. Däremot medför klassarv en uppsättning med klara begränsningar och problem som strategi förbigår. Nackdelarna med klassarv kan belysa varför strategi är mycket användbart och mera flexibelt. Observera figur 9 som presenterar ett påhittat exempel med en design som använder klassarv. I exemplet representeras fyra olika djur med varsin klass som ärver av en superklass med namnet Animal. Alla djuren ärver metoden makeNoise() och överskuggar den med sina egna implementationer på metoden. Redan nu påträffas problemet med att det förekommer kodduplikation eftersom ForestCat och BengalCat har en identisk implementation på metoden (metoden markerat med blå text) och likaså för LabradorDog och CollieDog (metoden markerat med röd text). Detta strider mot designprincipen DRY (don’t repeat yourself) som lyder att man ska undvika att skriva samma kod på flera platser eftersom man blir tvungen att ändra på varje plats där koden finns om man måste ändra på denna gemensamma kod. I det här fallet med enbart en upprepning av koden skulle det inte bli något stort besvär att göra en ändring. Däremot om man hade många av dessa upprepningar så skulle det blir avsevärt svårare att hålla reda på allt när man behöver ändra på något, vilket är fallet hos många system i verkligheten därför att det typiskt finns många klienter som har behov av gemensam kod.



Figur 9. UML-diagram som illustrerar användningen av klassarv hos ett system.

Det är ändå möjligt att undvika kodduplikationen hos klassarv genom att gruppera det gemensamma beteendet med hjälp av användningen av en ytterligare superklass, vilket illustreras i figur 10. Den identiska implementationen av makeNoise() hos ForestCat och BengalCat är nu definierad på en enda plats, det vill säga hos superklassen Cat (och på motsvarande sätt för LabradorDog och CollieDog med sin superklass Dog). Dilemmat med kodduplikation är nu löst eftersom klienterna för den gemensamma algoritmen ärver den från en superklass, vilket betyder att en ändring i superklassens implementation leder till att alla klienterna också påverkas av ändringen. Olyckligtvis är detta dock ingen skalbar lösning, eftersom det hos verkliga system nästan alltid finns mycket flera beteenden och algoritmer som används av klasserna. Följaktligen blir det mycket svårare att gruppera beteendena med hjälp av klassarv och det leder till att man får klasshierarkier som snabbt växer sig mycket komplexa i takt med ökande antal beteenden och entiteter som bör representeras i systemet.



Figur 10. UML-diagram som illustrerar gruppering av ett gemensamt beteende med hjälp av nya superklasser.

### 5.1.2 Tillämpning av strategi och dess fördelar

Strategi kan åstadkomma det ovannämnda men på ett mycket smidigare sätt. Strategi fullbordas genom att kapsla in de relaterade beteendena eller algoritmerna bakom ett gränssnitt, varpå varianter av beteendet utgörs av konkreta klasser som implementerar gränssnittet. De konkreta implementationerna brukar ofta kallas för strategier för algoritmen och därav har denna benämning också gett namn åt själva designmönstret. Strategiernas klienter får åtkomst till beteendena med hjälp av komposition. En klient upprätthåller alltså en referens till en implementation av den vederbörande strategin och kan på så sätt anropa dess metoder. Därmed kan man välja vilka beteenden som ska användas hos en klient och vid behov kan man även byta ut dem under exekveringstid, vilket är möjligt tack vare användningen av komposition. Till skillnad från användningen av klassarv är det även lättare att konfigurera klienter som har behov av flera olika beteenden som härstammar från olika beteendefamiljer, eftersom allt man behöver är en referens till en implementation av beteendet som behövs för varje individuell beteendefamilj. Den generella designen hos strategi presenteras i figur 11 där det finns en godtycklig kontext som innefattar en referens till en av de konkreta implementationerna (ConcreteStrategtA, ConcreteStrategtB och ConcreteStrategtC) av den aktuella algoritmen som sedan kan anropas när den behövs. [1]–[3]



Figur 11. UML-diagram som visar den generella designen hos strategi. [3, s. 351]

Genom att applicera strategi på det tidigare leksaksexemplet så skulle man kunna komma fram till en design som exemplifieras av figur 12. Beteendet hos makeNoise() har nu kapslats in hos implementationerna av gränssnittet NoiseBehavior. När en klass av typen Animal anropar makeNoise() delegeras anropet vidare till den egentliga metoden för beteendet som finns hos instansen av NoiseBehavior som varje subklass har. Till exempel har ForestCat en referens till en instans av CatNoiseBehavior och när makeNoise() anropas hos ForestCat är det egentligen performNoise() hos CatNoiseBehavior som används (markerat med blått) för att förverkliga det önskade beteendet. Likadant fungerar det för BengalCat, men däremot har LabradorDog och CollieDog i stället en instans av DogNoiseBehavior som innefattar en annan variant (markerat med blått) av performNoise() som används när dessa klasser anropar makeNoise(). Sålunda har algoritmerna och deras klienter separerats vilket sänker på graden av koppling i systemet och höjer på flexibiliteten. Genom att definiera nya gränssnitt och implementationer av dessa så kan man trivialt introducera nya beteendefamiljer som klienterna kan utnyttja med hjälp av motsvarande referenser, vilket möjliggör skalbarhet som blev problematisk vid användningen av klassarv.



Figur 12. UML-diagram för samma system såsom i figur 5 och 6 men med användningen av designen hos strategi.

Strategi är ett synnerligen användbart mönster eftersom det kan tillämpas på väldigt många sätt och bidrar till en betydligt lägre koppling hos designen. Till exempel för datastrukturer, som ansvarar för att lagra och representera mängder av data, så används olika sorteringsalgoritmer för att behandla och ordna den data som lagras. Med hjälp av strategi kan man välja den sorteringsalgoritm som passar bäst med hänsyn till de rådande omständigheterna hos systemet för att bespara tid och beräkningsresurser hos arkitekturen som systemet körs på. Det är alltså inte nödvändigt för klienterna att följa samma beteende hos en strategi under alla omständigheter. En ursprungsstrategi konfigureras typiskt vid den första instansieringen av en klient med hjälp av en konstruktormetod, men därefter kan man i ett senare skede byta ut strategin mot en annan med hjälp av andra metoder och logik i systemet. [1]

Tack vare dess flexibilitet kan strategi dessutom införlivas med andra designmönster för att etablera en mera komplex design. Designmönster strävar efter att undvika onödig komplexitet men beroende på systemets krav kan det likväl leda till behov av att kombinera mönster för att gottgöra systemets funktionalitet. McNatt och Bieman [12] undersöker olika mönsterkopplingar i sin studie där de analyserar litteratur kring tillämpningar av kombinerade mönster. Ur deras resultat framgår det att strategi kan användas tillsammans med ett flertal andra designmönster för att åstadkomma olika specialiserade mönsterkopplingar. Emellertid så är en del av dessa kopplingar strama, vilket är en egenskap som bör beaktas vid användningen av mönsterkopplingar eftersom det betyder att klasserna hos mönstren är beroende av varandra. Således går man miste om en del av flexibiliteten i designen med en sådan mönsterkoppling. Däremot kategoriserar McNatt och Bieman majoriteten av mönsterkopplingar med strategi som korsningskopplingar (intersection) där relationen mellan mönstren karakteriseras av informationsflöde och ömsesidig interaktion. McNatt och Bieman medger att sådana kopplingar förvisso inte garanterar en god design men är mera skonsamma gentemot flexibiliteten än de andra kopplingskategorierna som definierats i studien.

# 6 Sammanfattning och diskussion

Designmönstren som behandlats kan summeras enligt följande: fabriksmetoden och abstrakt fabrik kapslar in instansieringen och konfigureringen av objekt, fasad förenklar interaktionen med subsystem som består av flera komponenter, adapter möjliggör samverkan mellan klasser som har inkompatibla gränssnitt samt strategi som kapslar in återkommande beteenden och algoritmer hos ett system. Det finns ytterligare många fler designmönster bland de som ursprungligen presenterats av Gamma et al. [3]. Dessa mönster löser i sin tur andra problem som även är av intresse, dock överskrider de omfattningen i det här arbetet och är således utelämnade. Däremot har åtminstone ett mönster från varje mönsterkategori avhandlats för att ge en överblick över mönstrens avsikt och nytta. Kategorierna och deras mönster kan sammanfattas så här: skapandemönster hanterar hur objekt kan skapas och representeras i ett system, strukturmönster möjliggör konstellationer av mindre komponenter för att bygga större helheter och slutligen beteendemönster som ansvarar för att effektivisera och modularisera kommunikationen mellan objekt samt deras beteenden.

Fastän enbart fem av de 23 klassiska objektorienterade designmönstren som lyfts fram av Gamma et al. [3] har behandlats i denna avhandling framgår det ändå hur användbara designmönster är och vilken betydelse de har för mjukvaruutveckling. Mönstren utgör inte färdiga lösningar på problemen som de vill lösa, utan i stället beskriver mönstren en generell lösning som kan skräddarsys för att passa in i sitt aktuella system. Genom att tillämpa mönstren från de olika kategorierna uppnås en förbättrad flexibilitet och modularitet hos systemet som utvecklas, vilket i sin tur leder till att systemet blir lättare att underhålla och vidareutveckla. Utöver detta gör mönstren också designen hos system mera överskådlig och begriplig, vilket följaktligen även förenklar samarbetet och kommunikationen mellan systemets utvecklare.

Bekräftelse på designmönstrens generaliserbarhet uppmärksammas bland annat hos exemplet med ramverkssystem i kapitel 3.2 samt användningen av adapter för att lösa gränssnittskonflikten presenterad i kapitel 4.2. Det finns ett flertal med andra tillämpningar för designmönster. Ett tillämpningsområde som inte tidigare omnämnts är datorspelsutveckling där man ofta blir tvungen att överväga hur man ska implementera olika spelmekaniker som innefattar regler och logik hos spelet. Ampatzoglou et al. [13], [14] beskriver i sina studier om användningen av designmönster inom spelutveckling hur designmönster är utmärkta för att implementera spel. Dels eftersom mönstren är användbara för att realisera spelmekanikerna själva, dels för att det inom den moderna spelbranschen är mycket vanligt att spel vidareutvecklas efter sin initiala lansering och därmed förutsätter skalbarhet i sin mjukvarudesign. Detta är precis vad designmönstren vill åstadkomma och således är det här ett annat bra exempel på hur mönstren kan tillämpas.

Dessvärre är designmönstren inte perfekta på grund av att när man anpassar dem till sitt system blir man ofta tvungen att göra kompromisser för att uppnå en funktionell implementation. Detta beror på att strukturen och designen hos verkliga system är mycket mera komplex än de förenklade och ideala omständigheter vari individuella designmönster presenteras. När man gör sådana kompromisser leder det ofta olyckligtvis till att man förlorar en del av fördelarna och flexibiliteten som utlovas av designmönstren. Exempel på detta observeras bland annat i kapitlen 3.2 om abstrakt fabrik respektive kapitel 4.2 om adapter. Förvisso existerar det även specialiserade mönster som kunde lösa ovannämnda situationer, men däremot är dessa mindre kända och utforskade. Dessutom strider vissa specialiserade mönster i viss mån mot grundtanken med designmönster, nämligen att de ska vara generaliserbara och att kärnan i deras resonemang kring designen är tydlig. Specialiseringarna hamnar alltså i ett gränsområde mellan designmönster och konkreta lösningar. Därmed blir det tveksamt huruvida samtliga av dessa specialiseringar kan anses vara egentliga designmönster.

En annan mängd av begränsningar hos designmönster belyses av Wirfs-Brock [15] i sin essä där hon beskriver sin 15-åriga erfarenhet av forskning och arbete med designmönster. Wirfs-Brock påpekar att mönstrens generaliserbarhet är ändamålsenlig men att det vid konkreta tillämpningar förutsätts ytterligare erfarenhet och kunskap för att kunna fatta mera detaljerade designbeslut. Wirfs-Brock kritiserar därför presentationen av samtliga designmönster eftersom de inte täcker dessa detaljer som kan vara väsentliga vid implementationen av särskilda system. Förutom detta anmärker Wirfs-Brock också att det är svårt för nya mönster att få fotfäste inom mjukvaruutveckling oavsett hur användbara de är, vilket beror på att de fördunklas och glöms bort bland konferenser och deras förfaranden. Därmed nås i första hand enbart engagerade akademiker och sålunda förblir majoriteten av dessa mönster okända och outnyttjade.

# Referenser

[1] E. Freeman, E. Robson, K. Sierra, and B. Bates, *Head first design patterns: building extensible & maintainable object-oriented software*, 2nd edition. Bejiing: O’Reilly, 2021.

[2] S. Jiang and H. Mu, “Design patterns in object oriented analysis and design,” in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, Jul. 2011, pp. 326–329. doi: 10.1109/ICSESS.2011.5982229.

[3] E. Gamma, Ed., *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995.

[4] V. Sarcar, “Introduction to Design Patterns,” in *Interactive Object-Oriented Programming in Java*, Berkeley, CA: Apress, 2020, pp. 443–492. doi: 10.1007/978-1-4842-5404-2\_15.

[5] A. Waheed, G. Rasool, S. Ubaid, and F. Ghaffar, “Discovery of design patterns variants for quality software development,” in *2016 International Conference on Intelligent Systems Engineering (ICISE)*, Jan. 2016, pp. 185–191. doi: 10.1109/IN℡SE.2016.7475118.

[6] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, “Design Pattern Detection Using Similarity Scoring,” *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, Nov. 2006, doi: 10.1109/TSE.2006.112.

[7] D. Budgen, “Design Patterns: Magic or Myth?,” *IEEE Softw.*, vol. 30, no. 2, pp. 87–90, Mar. 2013, doi: 10.1109/MS.2013.26.

[8] M. Ó. Cinnéide and P. Fagan, “Design patterns: the devils in the detail,” in *Proceedings of the 2006 conference on Pattern languages of programs*, New York, NY, USA, Oct. 2006, pp. 1–9. doi: 10.1145/1415472.1415511.

[9] M. Fayad and D. C. Schmidt, “Object-oriented application frameworks,” *Commun. ACM*, vol. 40, no. 10, pp. 32–38, Oct. 1997, doi: 10.1145/262793.262798.

[10]P. Gahlyan and S. Narayan Singh, “Analysis of Catalogue of GoF Software Design Patterns,” in *2018 8th International Conference on Cloud Computing, Data Science Engineering (Confluence)*, Jan. 2018, pp. 814–818. doi: 10.1109/CONFLUENCE.2018.8442878.

[11]A. C. Kak, “Multiple Inheritance in C++,” in *Programming with Objects: A Comparative Presentation of Object-Oriented Programming With C++ and Java*, IEEE, 2003, pp. 721–787. doi: 10.1109/9780470547144.ch16.

[12]W. B. McNatt and J. M. Bieman, “Coupling of design patterns: common practices and their benefits,” in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, Oct. 2001, pp. 574–579. doi: 10.1109/CMPSAC.2001.960670.

[13]A. Ampatzoglou and A. Chatzigeorgiou, “Evaluation of object-oriented design patterns in game development,” *Inf. Softw. Technol.*, vol. 49, no. 5, pp. 445–454, May 2007, doi: 10.1016/j.infsof.2006.07.003.

[14]X.-C. Kounoukla, A. Ampatzoglou, and K. Anagnostopoulos, “Implementing Game Mechanics with GoF Design Patterns,” in *Proceedings of the 20th Pan-Hellenic Conference on Informatics*, New York, NY, USA, Nov. 2016, pp. 1–4. doi: 10.1145/3003733.3003779.

[15]R. Wirfs-Brock, “Should we stop writing design patterns?,” in *Proceedings of the 27th Conference on Pattern Languages of Programs*, USA, Oct. 2020, pp. 1–11.