Strategy en Bridge

Introductie *73*

Leerkern *73*

9 The Strategy pattern *74*
- 9.1 An approach to handling new requirements *74*
- 9.2 The international e-commerce system case study: initial requirements *74*
- 9.3 Handling new requirements *74*
- 9.4 The Strategy pattern *75*
- 9.5 Field Notes: using the Strategy pattern *76*

10 The Bridge pattern *76*
- 10.1 Introducing the Bridge pattern *76*
- 10.2 Learning the Bridge pattern: an example *77*
- 10.3 An observation about using design patterns *77*
- 10.4 Learning the Bridge pattern: deriving it *78*
- 10.5 The Bridge pattern in retrospect *80*
- 10.6 Field notes: using the Bridge pattern *80*

Terugkoppeling *83*

– Uitwerking van de opgaven *83*

Leereenheid 5

Strategy en Bridge

INTRODUCTIE

Na een samenvatting van de principes van objectoriëntatie, de beschrijving van een casus en een inleiding in design patterns, hebben we in de vorige leereenheid kennisgemaakt met twee relatief eenvoudige patterns, het Facade pattern en het Adapter pattern.

In deze leereenheid komen het Strategy en het Bridge pattern aan bod. Het Strategy pattern is een logische voortzetting van de stelregel om delegatie boven overerving te prefereren: in het Strategy pattern wordt gedrag gedelegeerd naar een speciaal object en kan men, door middel van polymorfie, een object verschillende vormen van gedrag laten vertonen. Het Bridge pattern is een heel stuk lastiger te begrijpen dan de tot nu toe behandelde patterns.

Het Bridge pattern wordt geïllustreerd aan de hand van de CAD/CAM-casus die in leereenheid 2 is geïntroduceerd. Ter illustratie van het Strategy pattern wordt een nieuw voorbeeldsysteem geïntroduceerd.

LEERDOELEN
Na het bestuderen van deze leereenheid wordt verwacht dat u
- kunt aangeven wat voor problemen het Strategy en het Bridge pattern oplossen
- het Strategy en het Bridge pattern kunt beschrijven en toepassen
- situaties kunt herkennen waarin u het Strategy en het Bridge pattern kunt gebruiken
- kunt beschrijven hoe het Bridge pattern gecombineerd kan worden met het Adapter pattern
- het Strategy en het Bridge pattern kunt implementeren.

Studeeraanwijzingen
Hoofdstukken 9 en 10 van het tekstboek behoren tot de stof.

Terminologie

Engels	*Nederlands*
abstraction	abstractie
compound	samengesteld
decoupling	ontkoppeling
implementation	implementatie, praktische invulling van een abstractie
implementor	object dat een praktische invulling geeft aan een abstractie

LEERKERN

9 The Strategy pattern

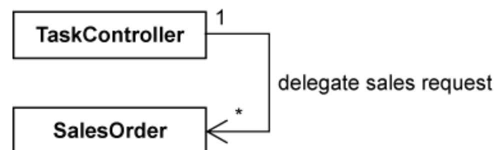
9.1 AN APPROACH TO HANDLING NEW REQUIREMENTS

De richtlijnen voor ontwerpen, die we eerder in de cursus ook al tegenkwamen, zorgen ervoor dat een ontwerp beter bestand wordt tegen veranderingen; ook tegen veranderingen die niet expliciet meegenomen zijn bij het opstellen van het ontwerp.

In de vorige leereenheden hebt u al gezien dat de juiste formulering van 'Favor object aggregation over class inheritance' is:
Favor delegation over inheritance.

9.2 THE INTERNATIONAL E-COMMERCE SYSTEM CASE STUDY: INITIAL REQUIREMENTS

Het klassendiagram van figure 9-1 stemt niet overeen met de beschrijving in de tekst. De tekst vertelt dat de `TaskController` een sales request doorgeeft aan een `SalesOrder` object. De `TaskController` delegeert dus het afhandelen van de sales request aan een `SalesOrder` object. Er moet daarom een associatie bestaan tussen `TaskController` en `SalesOrder`, in plaats van een afhankelijkheidsrelatie. Figuur 5.1 laat een klassendiagram zien dat overeenkomt met de tekst.



FIGUUR 5.1 TaskController en SalesOrder

9.3 HANDLING NEW REQUIREMENTS

Een delegate in C# is een variabele waaraan als waarde een methode kan worden gegeven (zolang de signatuur van de methode overeenstemt met de signatuur van de delegate). Een function pointer in C is een referentie naar een functie. Java kent iets dergelijks niet.

Wanneer u figure 9-4 en de code van example 9-1 bekijkt, zou u kunnen denken dat het probleem nu alleen maar verschoven is van `SalesOrder` naar `TaskController`: de switches zijn nu inderdaad verdwenen uit `SalesOrder`, maar zijn nodig in `TaskController`. Dat is juist. Het doel is in feite dan ook niet om switches kwijt te raken, maar om ervoor te zorgen dat een object (van `SalesOrder`) dat een ander object (van `CalcTax`) gebruikt, niet zelf verantwoordelijk is om uit te zoeken van welke klasse dat object precies moet zijn.

Die verantwoordelijkheid is in dit geval in handen van `TaskController`. Het voordeel is dat `SalesOrder` nu van het `CalcTax` object alleen maar weet dat het een concrete subklasse is van `CalcTax`. Het voordeel is

ook dat `TaskController` de kennis over het land ook kan gebruiken om – bijvoorbeeld – een object te instantiëren dat weet hoe het bedragen moet afdrukken.

De opmerking over parameters in klassendiagrammen, waarbij een keyword 'in' wordt toegevoegd aan parameters, is onjuist: dat is niet nodig.

OPGAVE 5.1

Het komt vaak voor dat er binnen een systeem een aantal klassen aanwezig is die personen voorstellen. In een systeem dat de administratie rond studiebegeleiding ondersteunt, zullen bijvoorbeeld klassen voorkomen als `Docent`, `Student`, `Studiebegeleider` en `Examinator`.
Het lijkt dan op het eerste gezicht handig de gemeenschappelijke kenmerken te generaliseren tot een superklasse `Persoon`.
Maar dan doet zich het scenario voor dat een docent ook studiebegeleider is. Een object zou dan van klasse moeten kunnen wisselen of van twee klassen tegelijkertijd moeten kunnen zijn.
a Hoe zou u dat probleem oplossen?
b Welke strategie wordt met dit (veelvoorkomende) voorbeeld geïllustreerd?

9.4 THE STRATEGY PATTERN

Creëren van een ConcreteStrategy

Voor het Strategy pattern geldt: de verantwoordelijkheid voor het *creëren van een object van de juiste ConcreteStrategy klasse* ligt buiten de verantwoordelijkheid van het pattern. Er bestaat een aantal patterns voor het creëren van objecten. Enkele daarvan komen later in deze cursus aan bod.

In het stuk 'The Strategy pattern: key features' wordt aangevoerd dat switches geëlimineerd zouden worden. Dat is dus niet geheel juist: het creëren van de juiste objecten kan nog steeds met switches gebeuren. Of daarvoor een design pattern wordt ingezet, is een aparte vraag.

OPGAVE 5.2

Java kent de klasse `java.awt.Container`. Deze klasse kent onder andere de methode `setLayout`, waarbij een object van een klasse die de interface `java.awt.LayoutManager` implementeert als argument wordt meegegeven. Voorbeelden van zulke klassen zijn `java.awt.GridLayout` en `java.awt.FlowLayout`.
De klasse `java.awt.Container` kent ook een methode `doLayout`, die over het algemeen niet direct door de programmeur wordt aangeroepen, maar die binnen de klasse `Container` zelf gebruikt wordt om methoden als `add` en `remove` te implementeren (om een `java.awt.Component` toe te voegen of te verwijderen). In de implementatie van de methode `doLayout` van `java.awt.Container` wordt de methode `layoutContainer` van het `java.awt.LayoutManager`-object aangeroepen.
a Teken een klassendiagram waarin genoemde klassen voorkomen (`Container`, `LayoutManager`, `GridLayout` en `FlowLayout`).
b Laat zien dat hier het Strategy pattern wordt gebruikt, door het klassendiagram te vergelijken met figure 9-6 van het tekstboek.

OPGAVE 5.3
Voor een tekstverwerker zijn verschillende manieren nodig om te formatteren: links uitgelijnd, rechts uitgelijnd of gecentreerd.
a Noem een aantal nadelen van de oplossing waarbij de klasse Document een methode format krijgt waarin de manier van formatteren als argument wordt meegegeven.
b Geef een klassendiagram van een oplossing met behulp van het Strategy pattern.

9.5 FIELD NOTES: USING THE STRATEGY PATTERN

OPGAVE 5.4
De 'wet van Demeter' is een 'wet' voor objectgeoriënteerd ontwerpen, die – kort gezegd – neerkomt op de waarschuwing: praat alleen met bekenden.
In een methode m van object o mag gecommuniceerd worden met:
– objecten die als parameter worden meegegeven in de aanroep van methode m
– objecten die gecreëerd worden binnen methode m
– objecten die instantievariabelen zijn van object o.
a Welke van de drie in het tekstboek gepresenteerde oplossingen om de leeftijd van de klant te betrekken in de berekening van de belasting, zou tegen de wet van Demeter in kunnen gaan?
b Wat zou het nadeel zijn van het schenden van die wet?

OPGAVE 5.5
Stel: u bent bezig een herbruikbare klasse `Bedrag` te schrijven. Een van de verantwoordelijkheden van deze klasse bestaat uit het op de juiste manier formatteren van een bedrag volgens regels die per land verschillen, en met de juiste munteenheid voor of achter het bedrag.
De bouwsteen bij deze opgave bevat twee klassen, `NlFormatteerder` en `UsaFormatteerder`, met methoden die dat probleem oplossen voor respectievelijk Nederland en de Verenigde Staten.
a Teken een klassendiagram waarin het Strategy pattern wordt toegepast op dit probleem.
b Schrijf, gebruikmakend van het Strategy pattern, een klasse `Bedrag` die als Context fungeert, en een interface plus klassen die als Strategies fungeren.

OPGAVE 5.6
Als u op de GoF-cd het State pattern opzoekt, ziet u onder het kopje 'Structure' een structuur die, afgezien van de namen, in feite identiek is aan de structuur van het Strategy pattern.
Lees het stuk over het State pattern door en probeer te formuleren wat het verschil is tussen het State pattern en het Strategy pattern.

10 The Bridge pattern

10.1 INTRODUCING THE BRIDGE PATTERN

Implementatie in het objectgeoriënteerde paradigma

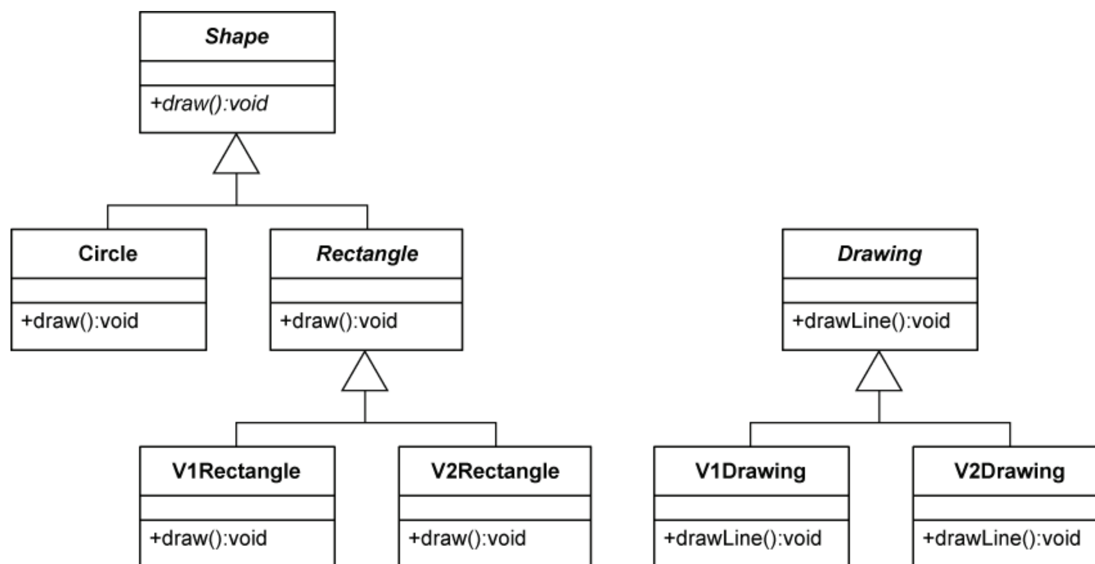
De uitleg over waar 'implementation' voor staat, is niet erg duidelijk. Het probleem zit hem in het feit dat *implementatie* in het *objectgeoriënteerde paradigma* staat voor de concrete klassen die als 'type' een abstracte klasse of een interface hebben. Die concrete klassen vormen de implementatie van de interface of van de abstracte klasse.

Implementatie bij het Bridge pattern

De term *implementatie* wordt bij de behandeling van het *Bridge pattern* gebruikt om de fysieke representatie van iets aan te duiden, de praktische invulling van een abstractie. Een voorbeeld: stel u een tabel voor met namen en e-mailadressen van studenten. Binnen een programma kan zo'n tabel abstract gerepresenteerd worden door een verzameling `Student`-objecten. Fysiek kan de tabel worden opgeslagen in een Word-bestand, in een Excel-bestand, in een tekstbestand of in bijvoorbeeld een database. De verschillende bestanden vormen in dit geval verschillende vormen van implementatie in de betekenis zoals die bij het Bridge pattern wordt gehanteerd; de `Student`-objecten vormen de abstractie.

10.2 LEARNING THE BRIDGE PATTERN: AN EXAMPLE

Figure 10-2 stemt niet overeen met de tekst: in de tekst is sprake van twee subklassen van `Rectangle`: `V1Rectangle` en `V2Rectangle`. Ook het feit dat `V1Drawing` en `V2Drawing` verschillende methoden hebben om een cirkel of een lijn te tekenen, is niet te zien in het diagram. Wanneer we van dat laatste feit afzien, ziet het klassendiagram er uit als in figuur 5.2.



FIGUUR 5.2 Design for rectangles and drawing programs (DP1 and DP2)

De eerste oplossing van het probleem, geschetst in figure 10-3 van het tekstboek, illustreert de noodzaak van het Bridge pattern.

OPGAVE 5.7
Laat zien, aan de hand van de criteria die zijn opgesteld in leereenheid 2, wat er minder goed is aan de oplossingen zoals die hier geschetst worden.

10.3 AN OBSERVATION ABOUT USING DESIGN PATTERNS

Probleem en context

Houd deze paragraaf in gedachten bij het doornemen van design patterns uit andere bronnen: het *probleem* en de *context* zijn de belangrijkste onderdelen van een beschrijving van een pattern.

In de sjabloon die op de GoF-cd wordt gebruikt om patterns te beschrijven, is onder de kopjes ‘Intent’ en ‘Applicability’ te vinden welk soort problemen het pattern oplost. Nadere informatie en uitleg daarover zijn te vinden onder het kopje ‘Motivation’.

10.4 LEARNING THE BRIDGE PATTERN: DERIVING IT

Wanneer u figure 10-11 bekijkt, vraagt u zich misschien af hoe een `Rectangle` object met het juiste `Drawing` object wordt verbonden: dat is niet duidelijk uit de figuur. Het antwoord is dat die verbinding buiten de verantwoordelijkheden van het Bridge pattern ligt. Zoals u nog zult leren in deze cursus, is het goed om het creëren van objecten te scheiden van het gebruiken van objecten. Voor het creëren van objecten zult u aparte patterns gaan gebruiken.

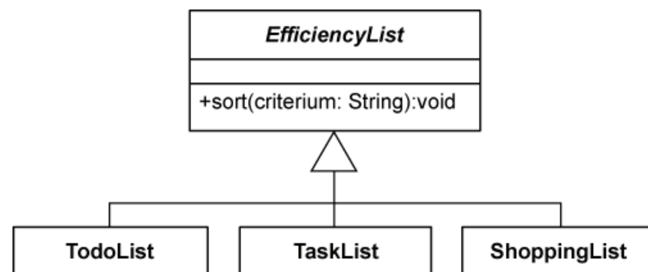
Bij het bekijken van een pattern als het Strategy of Bridge pattern, gaat het er dus om dat u begrijpt hoe een en ander werkt; hoe de objecten geïnstantieerd worden, en hoe daarbij de juiste relaties gelegd worden, is een ander probleem.

In de code van example 10-3 zit bij de `Shapes` een aantal foutjes. De juiste code van dat gedeelte is:

```
public class Client {
  static public void main() {
    Shape myShapes[];
    Factory myFactory = new Factory();
    myShapes = myFactory.getShapes();
    for (int i=0; i < myShapes.length; i++) {
      myShapes[i].draw();
    }
  }
}
abstract public class Shape {
  protected Drawing myDrawing;
  abstract public void draw();
  Shape(Drawing drawing) {
    myDrawing = drawing;
  }
  protected void drawLine(double x1, double x2, double y1,
                          double y2) {
    myDrawing.drawLine(x1, x2, y1, y2);
  }
  protected void drawCircle(double x,
        double y, double r) {
    myDrawing.drawCircle(x, y, r);
  }
}

public class Rectangle extends Shape {
  private double _x1, _x2, _y1, _y2;
  public Rectangle(Drawing dp, double x, double xx,
                   double y, double yy) {
    super(dp);
    this.x1 = x; _
    this.x2 = xx; _
    this.y1 = y; _
    this.y2 = yy;
  }
```

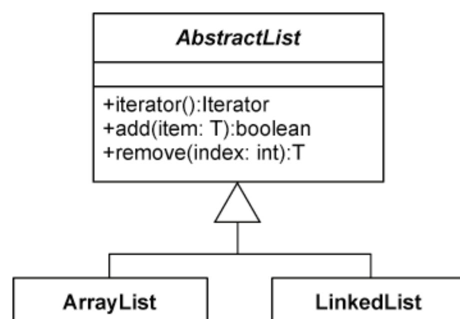
```
  public void draw() {
    drawLine(_x1, _y1, _x2, _y1);
    drawLine(_x2, _y1, _x2, _y2);
    drawLine(_x2, _y2, _x1, _y2);
    drawLine(_x1, _y2, _x1, _y1);
  }
}

public class Circle extends Shape {
  private double _x, _y, _r;
  public Circle(Drawing dp,
          double x, double y, double r) {
    super(dp);
      this.x = x; _
      this.y = y; _
      this.r = r;
  }
  public void draw() {
    drawCircle(_x, _y, _r);
  }
}
```

OPGAVE 5.8
U bent een programma voor allerlei soorten lijstjes aan het schrijven: een todolist, een shoppinglist, een tasklist. U hebt een abstracte klasse `EfficiencyList`, met concrete subklassen `TaskList`, `TodoList` en `ShoppingList`. De abstracte klasse kent onder andere de methode `sort`, zie figuur 5.3.



FIGUUR 5.3 Lijstjes

Het moet mogelijk zijn om de `TasksList` soms te implementeren met behulp van een `ArrayList`, en soms met behulp van een `LinkedList`: zie figuur 5.4.



FIGUUR 5.4 ArrayList en LinkedList

a Laat zien hoe u een Bridge kunt gebruiken voor dit doel. Teken het klassendiagram. U kunt er van uitgaan dat u de getoonde drie methoden kunt gebruiken; u hoeft niet precies uit te denken hoe u `sort` daarmee implementeert.
b Geef aan wat de Abstraction is en wat de Implementor.

10.5 THE BRIDGE PATTERN IN RETROSPECT

De verschillen tussen een abstracte klasse en een interface zijn in leereenheid 1 behandeld.

Kiezen van de juiste implementator

Een aspect van het Bridge pattern dat niet aan bod komt, is de vraag hoe de *juiste concrete implementator wordt gekozen*. Het Bridge pattern laat zien hoe de structuur is van de klassen bij een abstractie met verschillende mogelijke implementaties; het creëren van die structuur is een andere kwestie, die uitdrukkelijk niet binnen het Bridge pattern valt. De verantwoordelijkheid voor het creëren van de objecten die onderdeel zijn van het Bridge pattern, situeert u dus buiten de onderdelen van het pattern.

In de volgende leereenheid wordt een pattern behandeld dat te maken heeft met het kiezen van de juiste klassen in dergelijke situaties.

10.6 FIELD NOTES: USING THE BRIDGE PATTERN

OPGAVE 5.9
Laat zien hoe het Adapter pattern geïntegreerd is in de oplossing uit figure 10-13 van het tekstboek.

Refactoring

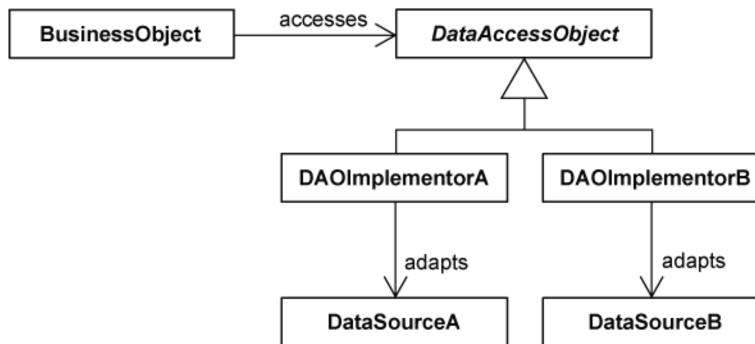
Refactoring is in feite het herstructureren van (stukjes) software, zonder iets aan de functionaliteit te veranderen. Twee voorbeelden zijn het splitsen van een lange methode en het invoegen van een superklasse voor het gemeenschappelijke deel van een aantal afzonderlijke klassen. Het is belangrijk om het aanbrengen van nieuwe functionaliteit gescheiden te houden van de activiteit van refactoring. Het doel van refactoring is om de code beter onderhoudbaar te maken. Design patterns kunnen daar vaak bij helpen, omdat ze flexibiliteit bieden in verband met mogelijke toekomstige variaties.

De design patterns die beschreven worden op de GoF-cd, waarvan we er in het tekstboek een aantal tegenkomen, worden tot de ‘klassiekers’ gerekend. Sinds het verschijnen van dat eerste boek over design patterns van de Gang of Four, zijn er veel meer design patterns beschreven. Vaak zijn dat variaties van, uitbreidingen op of specialisaties van de patterns van het GoF-boek.

Een voorbeeld van een beschrijving van een verzameling design patterns zijn de J2EE-patterns van Sun: ‘best practices’ voor het gebruiken van Java 2 enterprise edition. J2EE staat voor Java 2 platform, enterprise edition, en bevat naast het standaard Java 2 platform onder meer Enterprise JavaBeans-componenten, de Java servlets API, JavaServer Pages en XML-technologie. Informatie over de patterns voor J2EE is te vinden via de link op Studienet.

OPGAVE 5.10

Een pattern uit de J2EE-wereld is het Data access object pattern, waarbij een `BusinessObject` nooit rechtstreeks verbonden is met een database (of andersoortig systeem voor het opslaan van gegevens), maar via een speciaal object van type `DataAccessObject`. Het Data access object pattern kan op een aantal manieren worden uitgewerkt. Figuur 5.5 laat zo'n uitwerking zien.



FIGUUR 5.5 Data access object pattern

Het `BusinessObject` staat voor een willekeurige klasse waarvan de objecten gebruik willen maken van een database.
De klasse `DataAccessObject` wordt geïmplementeerd door verschillende `DAOImplementor`-klassen, die verantwoordelijk zijn voor de communicatie met het systeem voor persistentie. De gegevens worden uiteindelijk bewaard in bijvoorbeeld een database, hier aangeduid met `DataSourceA` en `DataSourceB`.

a Geef aan hoe het Bridge pattern wordt gebruikt in het Data access object pattern, door de overeenkomsten te laten zien met figure 10-15 van het tekstboek.
Aanwijzing: het Bridge pattern is in dit geval niet helemaal volledig!
b Laat zien welk ander pattern in combinatie met het Bridge pattern wordt gebruikt.

OPGAVE 5.11

Als hulp voor het testen onder allerlei omstandigheden hebben ontwikkelaars een abstracte klasse `Log` geschreven, met subklassen `ErrorLog`, `DebugLog` en `TestLog`. De abstracte klasse `Log` beschikt over twee abstracte methoden `log`: een met een `String` als argument en een met een `Object` als argument.
De ontwikkelaars willen op een gemakkelijke manier kunnen instellen dat de loggegevens naar het scherm of naar een bestand worden gestuurd. Ze beschikken over de klassen `Scherm` en `Bestand`, met een methode `schrijf`, die een `String` als argument verwacht. Deze klassen hebben een abstracte superklasse `Uitvoer`.
U vindt deze klassen in de bouwsteen bij deze opgave.

a Geef aan hoe het Bridge pattern ingezet kan worden om de ontwikkelaars te hulp te komen, door figure 10-15 van het tekstboek als uitgangspunt te nemen en te laten zien hoe een en ander ingevuld kan worden met de genoemde klassen.

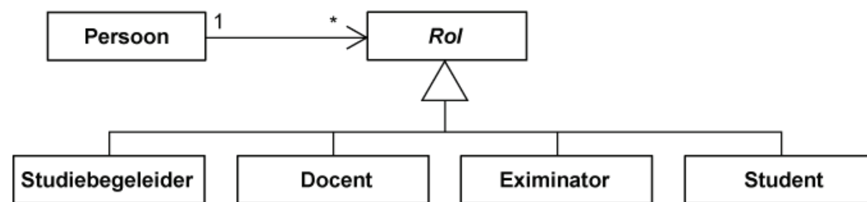
b Implementeer het loggingsysteem, gebruikmakend van de bouwsteen bij deze opgave.
Bedenk daarbij dat de taak van het creëren van objecten en het aan elkaar koppelen ervan niet behoort tot het Bridge pattern. Leg de verantwoordelijkheden daarvoor dus buiten de klassen die onderdeel zijn van het Bridge pattern.
c Beschrijf wat er moet gebeuren om een nieuw soort logging te ondersteunen.
d Beschrijf wat er moet gebeuren om een nieuwe manier van uitvoer te ondersteunen.

OPGAVE 5.12
U hebt bedacht dat steeds meer mensen met een notebook op schoot naar de televisie kijken. U hebt daarom het plan opgevat om een afstandbedieningsprogramma te schrijven.
We gaan er even van uit dat de hardware geen probleem is. De software is lastiger: u wilt één programma schrijven dat alle tv's kan aansturen, maar elke tv-fabrikant heeft een eigen protocol en een eigen commandoset. U mag ervan uitgaan dat elk merk een driver voor het notebook kan leveren (of ooit zal kunnen leveren) die de infraroodpoort op de juiste manier aanstuurt. Wat per fabrikant verschilt, zijn de interfaces van de drivers.
Teken een klassendiagram waarin u het Bridge pattern voor dit probleem inzet, met voor elk merk driver een Adapter. De abstractie wordt in dit geval gevormd door het `AfstandBedieningsProgramma`. Later kunnen daar nog RefinedAbstractions aan worden toegevoegd door klassen voor de televisie, voor de cd-speler, de dvd-speler en de radio te schrijven. Die hoeven niet in het klassendiagram terecht te komen.

TERUGKOPPELING

Uitwerking van de opgaven

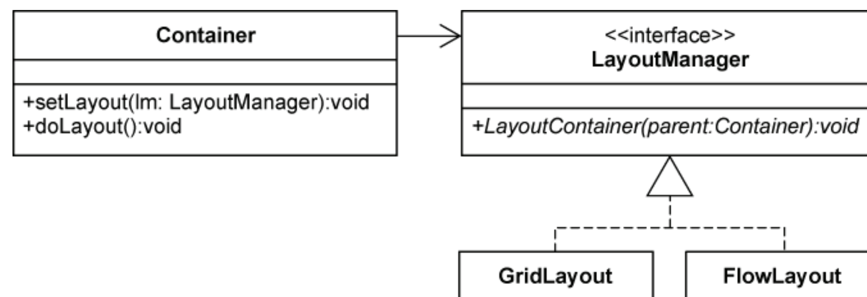
5.1 a Een object van de klasse `Persoon` kan een associatie krijgen met een object van de klasse `Docent`, `Studiebegeleider`, `Examinator` of `Student`. Zo'n `Persoon`-object kan eventueel associaties met meerdere rollen vormen. De klasse `Persoon` krijgt dus niet `Docent`, `Student` en dergelijke als subklassen, maar een associatie met een abstracte klasse `Rol`, die geïmplementeerd wordt door `Studiebegeleider`, `Docent`, `Examinator` en `Student`. Zie figuur 5.6.



FIGUUR 5.6 Persoon met Rol

b Deze oplossing is een voorbeeld van de strategie 'Favor delegation over inheritance' en laat zien wat de extra flexibiliteit is van delegatie ten opzichte van overerving.

5.2 a Een klassendiagram dat de relaties laat zien tussen `Container`, `LayoutManager` en een aantal klassen die `LayoutManager` implementeren, is weergegeven in figuur 5.7.

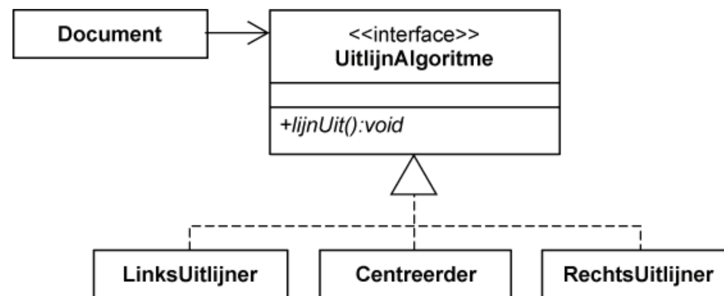


FIGUUR 5.7 Strategy pattern in java.awt

b `Container` komt overeen met Context in figure 9-6 van het tekstboek. `Container` bevat een `LayoutManager` bij wijze van Strategy en gebruikt de methode `layoutContainer` daarvan voor de implementatie van de methode `doLayout`.

5.3 a Nadelen van het meegeven van de manier van formatteren als parameter bij een methode zijn de volgende.
– De implementatie van de methode wordt ingewikkeld: er zijn switches nodig.
– Uitbreiden van het aantal manieren om te formatteren wordt steeds moeilijker naarmate er meer manieren zijn. Het kan in de toekomst bijvoorbeeld wenselijk zijn om tekst om een plaatje heen te kunnen formatteren. Dat zou dan allemaal binnen diezelfde implementatie van die ene formatteermethode moeten gebeuren.
– De klasse Document krijgt op deze manier wel een heel uitgebreide verantwoordelijkheid.

b Wanneer bij de oplossing van het tekstverwerkerprobleem gebruik wordt gemaakt van het Strategy pattern, krijgt de klasse `Document` een associatie met een abstracte klasse `UitlijnAlgoritme`. De concrete subklassen van deze abstracte klasse implementeren het uitlijnen (zie figuur 5.8).

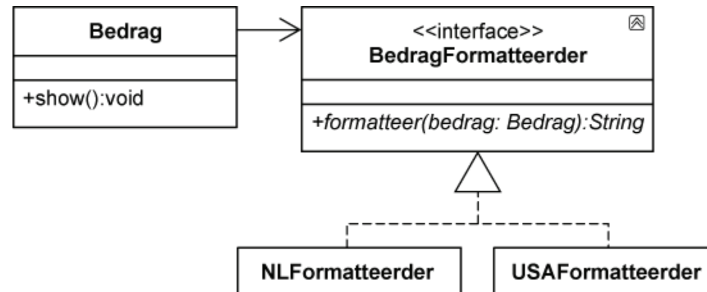


FIGUUR 5.8 Strategy pattern toegepast in de tekstverwerker

5.4 a Wanneer aan `SalesOrder` een referentie naar het `Customer`-object gevraagd moet worden om achter de leeftijd te komen, zoals in oplossing 3 gebeurt, kan het voorkomen dat de wet van Demeter geschonden wordt. Dat zou gebeuren wanneer `CalcTax` een referentie zou krijgen van `SalesOrder` naar het `Customer` object dat de leeftijd kan geven. Alleen wanneer `SalesOrder` zelf de verantwoordelijkheid heeft voor de leeftijd van de `Customer`, wordt de wet van Demeter niet geschonden. Met een apart `Customer`-object ligt het niet voor de hand dat de verantwoordelijkheid voor de leeftijd van de klant ligt bij `SalesOrder`.

b Het schenden van de wet van Demeter maakt het mogelijk dat er ‘objectgeoriënteerde spaghetticode’ ontstaat. De restricties die aan de communicatie tussen objecten worden gesteld, zorgen ervoor dat duidelijk blijft welke afhankelijkheden er bestaan tussen klassen. Zonder die restricties zou het vrijwel onmogelijk worden om te bepalen op welke klassen een verandering in een bepaalde klasse van invloed zou kunnen zijn.

5.5 a Een klassendiagram waarin het Strategy pattern is toegepast op het `Bedrag`-probleem, is getekend in figuur 5.9.
`Bedrag` vormt daarin de Context en heeft een aggregatierelatie met een Strategy in de vorm van een `BedragFormatteerder`.



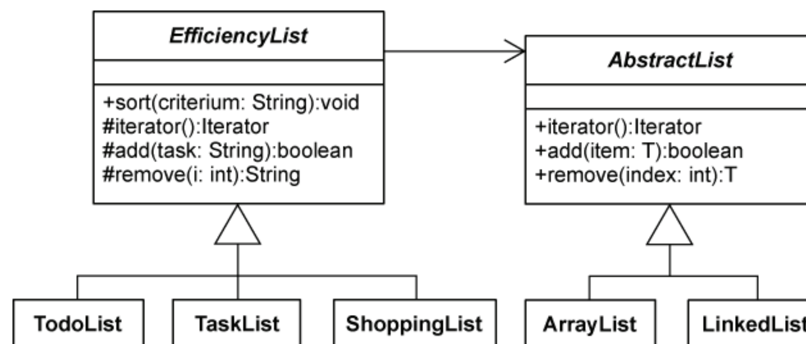
FIGUUR 5.9 Strategy pattern voor het Bedragprobleem

b De bouwsteen bij deze uitwerking geeft een heel simpele interface `BedragFormatteerder`, met alleen de methode `formatteer`, plus de klassen `NlFormatteerder` en `UsaFormatteerder` die nu de interface `BedragFormatteerder` implementeren. Daarnaast bevat de bouwsteen de klasse `Bedrag`, die als Context fungeert.

5.6 Het Strategy pattern kapselt de variatie van de implementatie van een algoritme in, in verschillende klassen voor de verschillende algoritmen.
Het State pattern is bedoeld voor situaties waarin een object zich in verschillende toestanden kan bevinden. Bij sommige methoden van zo'n object varieert de implementatie afhankelijk van de toestand. De concrete State objecten bevatten alle methoden waarvan de implementatie afhankelijk is per toestand. Vooral wanneer zo'n State klasse slechts één methode kent, lijkt het pattern erg op het Strategy pattern.
Een klasse `Lichtschakelaar` bijvoorbeeld, kent twee toestanden: Aan en Uit. Zo'n klasse beschikt over een methode `klik`, die in het ene geval het licht uit doet en tot gevolg heeft dat de toestand in Uit verandert, en in het andere geval het licht aan doet en tot gevolg heeft dat de toestand in Aan verandert.
Toepassen van het State pattern houdt in dat we een abstracte klasse `Toestand` maken, met twee subklassen, `Aan` en `Uit`, beide met de methode `klik`. Het verschil met het Strategy pattern is dan niet aan het klassendiagram te zien; dat verschil zit hem alleen in het feit dat de vraag welke concrete subklasse van `Toestand` gebruikt wordt, afhankelijk is van de toestand van `Lichtschakelaar`.

5.7 In figure 10-3 van het tekstboek worden de methoden `drawLine` en `drawCircle` geïmplementeerd door respectievelijk `Rectangle` en `Circle`. Elke nieuwe vorm die een lijn of een cirkel nodig heeft om zichzelf te kunnen tekenen, moet deze methoden zelf ook implementeren. Dat betekent redundantie. Dat probleem is in figure 10-7 opgelost. Daar is echter weer redundantie in de tekenmethoden van bijvoorbeeld `V1Rectangle` en `V2Rectangle`: in beide methoden wordt viermaal een lijn getekend en dat algoritme staat nu dus dubbel in de oplossing.
– Overzichtelijkheid
Zowel aan figure 10-3 als 10-7 is te zien dat de overzichtelijkheid kwijt zal zijn bij het introduceren van een nieuwe vorm of een nieuwe versie van het tekenprogramma.
– Lage mate van koppeling
Aan de elkaar kruisende associaties in figure 10-3 is al te zien dat de mate van koppeling (te) hoog is. Wat dat betreft ziet figure 10-7 er beter uit.
– Hoge cohesie
Wat betreft cohesie is er niet veel mis met figure 10-3: elke klasse heeft een netjes afgebakende verantwoordelijkheid. In figure 10-7 worden `V1Shape` en `V2Shape` laag in cohesie wanneer er meer vormen worden toegevoegd.
– Intuïtie
Intuïtie is sterk afhankelijk van ervaring, maar de meeste mensen zullen in beide oplossingen niet snel een elegante oplossing zien.

5.8 a Figuur 5.10 toont de gevraagde Bridge.



FIGUUR 5.10 Klassendiagram van een Bridge

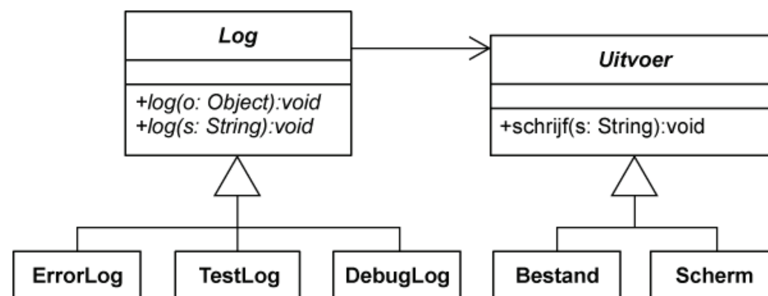
b Het Bridge pattern is te herkennen in de vorm van de abstracte klasse `EfficiencyList` (de Abstraction uit figure 10-15 van het tekstboek) en de interface `List` (de Implementor). `EfficiencyList` is de abstractie voor de lijsten die het programma gaat ondersteunen en `List` is de implementatie van die lijsten. Klasse `ToDoList` is een refined abstraction. Een concrete implementatie van `List` is bijvoorbeeld `ArrayList`.

5.9 `V1Drawing` en `V2Drawing` hebben een Adapter-achtige functie: ze zorgen ervoor dat `Shape` de methoden `drawLine` en `drawCircle` kan gebruiken bij een aanroep van de implementatieklassen, terwijl de eigenlijke implementatieklassen, `DP1` en `DP2`, die methoden niet bieden (of niet noodzakelijkerwijs).

5.10 a `BusinessObject` komt overeen met RefinedAbstraction van figure 10-15 van het tekstboek. Er is in dit geval geen Abstraction. `DataAccessObject` komt overeen met de Implementor. `DataSourceA` en `–B` komen overeen met ConcreteImplementatorA en -B.
NB: tussen `BusinessObject` en `DataAccessObject` is een gewone associatie getekend, terwijl in figure 10-15 tussen Abstraction en Implementor een aggregatie wordt getekend. Dat is geen wezenlijk verschil: beide patterns worden op zo'n hoog abstractieniveau beschreven dat het niet mogelijk is te beslissen of in alle gevallen een aggregatie meer op zijn plaats is dan een gewone associatie of andersom.

b Tussen `DataAccessObject` en `DataSourceA` en `–B` zitten nog twee Adapters, `DAOImplementorA` en `–B`.

5.11 a De situatie dat er verschillende klassen zijn voor het samenstellen van loggegevens, waarbij de loggegevens op verschillende manieren aan de buitenwereld kenbaar gemaakt kunnen worden, is bij uitstek een situatie waarin het Bridge pattern van pas komt.
Figuur 5.11 laat zien hoe het loggingsysteem er uitziet. Wanneer we figure 10-15 uit het tekstboek er naast houden, zien we dat de klasse `Log` voor de Abstraction staat, de klassen `ErrorLog`, `DebugLog` en `TestLog` voor de RefinedAbstractions, dat de klasse `Uitvoer` voor de Implementor staat en de klassen `Bestand` en `Scherm` ConcreteImplementors zijn.



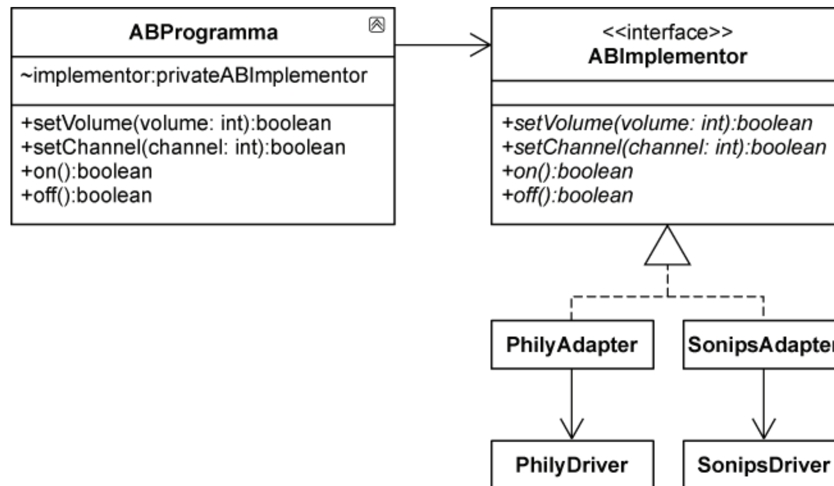
FIGUUR 5.11 Bridge voor het loggingsysteem

b De implementatie van het loggingsysteem is te vinden in de bouwsteen bij de uitwerking van deze opgave.
De implementatie houdt in dat de abstracte klasse `Log` een instantievariabele heeft gekregen van type `Uitvoer`.
De implementatie van de methode `log` van de drie subklassen van `Log` maakt gebruik van die instantievariabele.
Daarnaast is van belang dat het creëren van objecten en het koppelen ervan buiten de Bridge wordt gehouden. In het loggingsysteem gebeurt dat in de methode `main` van `LogApplicatie`.

c Om een nieuw soort logging te ondersteunen, wordt er een nieuwe subklasse van `Log` gemaakt. In de methode `log` van die subklasse wordt gebruikgemaakt van de instantievariabele `uitvoer`.

d Om een nieuw soort uitvoer te ondersteunen, is er alleen een nieuwe subklasse van `Uitvoer` nodig.

5.12 Het afstandsbedieningsprogramma is de abstractie in dit probleem. Er is geen refined abstractie (dat kan ooit nog komen wanneer we het programma uitbreiden en ook de dvd-speler willen aansturen en dergelijke). Voor elk merk (en soms type) televisie zal een driver geschreven moeten worden die de infraroodpoort op de juiste wijze aanstuurt, op dezelfde manier als de afstandsbediening voor de televisie zich gedraagt. We gebruiken een adapter om ervoor te zorgen dat alle drivers op dezelfde manier aangesproken kunnen worden. Het klassen-diagram ziet er dan uit als in figuur 5.12.



FIGUUR 5.12 Bridge pattern voor de afstandsbediening

Ter illustratie geven we een voorbeeld van een implementatie van de klasse `ABProgramma` en de interface `ABImplementor`:

```
public class ABProgramma {

  ABImplementator implementor;
  boolean impSet = false;

  public ABProgramma() {
  }

  /**
   * @param imp is een object van een klasse die
   * ABImplementator implementeert, met een
   * adapter naar de driver.
   */
  public void setImp(ABImplementator imp) {
    this.implementor = imp;
    impSet = true;
  }

  /**
   * @param vol is een absolute maat voor het volume.
   * @return true als alles naar wens gaat, en
   * false als het volume niet op de gewenste sterkte
   * gezet kan worden.
   */
```

```
  public boolean setVolume(int vol) {
    if (impSet) {
      return imp.setVolume(vol);
    }
    return false;
  }

  /**
   * @param channel is een int die het gewenste kanaal
   * aangeeft.
   * @return true als alles naar wens gaat, en
   * false als het gewenste kanaal niet bestaat.
   */
  public boolean setKanaalsetChannel(int channel) {
    if (impSet) {
      return imp.setChannel(channel);
    }
    return false;
  }

  /**
   * @return true als alles naar wens gaat, en
   * false als de tv niet aangezet kan worden.
   */
  public boolean on() {
    if (impSet) {
      return  imp.on();
    }
    return false;
  }

  /**
   * @return true als alles naar wens gaat, en
   * false als de tv niet uitgezet kan worden.
   */
  public boolean off() {
    if (impSet) {
      return imp.off();
    }
    return false;
  }
}

public interface ABImplementator {
  public boolean setVolume(int vol);
  public boolean setKanaal(int channel);
  public boolean aan();
  public boolean uit();
}
```