Tentamen i EDAF25

29 augusti, 2024

Skrivtid: 8-13

Lösningsförslag

Grafteori

Uppgift 1

(a) Djupet-först-sökning

(b) Här ville vi testa om ni praktiskt kan anpassa/tillämpa en av grafalgoritmerna ur kursen till/på ett givet problem. Vi ger ett par olika alternativa lösningar som skiljer sig åt såtillvida vägen sätts ihop när vi går ner i rekursionen eller när vi kommer tillbaka ur densamma.

Alternativ 1

```
hamilton(g) =
    foreach node u in g:
        m = a set containing all nodes in g except u
        initial_path = a list containing only node u
        path = find_path(u,m,initial_path)
        if path is not null:
            return path
    return null


find_path(x,s,p) =
    if the set s is empty:
        return p
    foreach neighbour n to node x in set s:
        new_set = a set containing all nodes in s except x
        new_path = p
        add n to the end of new_path
        found_path = find_path(n,new_set,new_path)
        if found_path is not null:
            return found_path
    return null
```

Alternativ 2

```
hamilton(g) =
    foreach node u in g:
        m = a set containing all nodes in g except u
        path = find_path(u,m)
        if path is not null:
            return path
    return null


find_path(x,s) =
    if the set s is empty:
        return a list containing x
    foreach neighbour n to node x:
        new_set = a set containing all nodes in s except x
        found_path = find_path(n,new_set)
        if found_path is not null:
            return a list containing x followed by the elements in found_path
    return null
```

Uppgift 2

(a) B, E, A, D, F ,C, G *eller* B, E, A, F, D, C, G (D och F har samma avstånd från B).

(b) Bellman-Fords algoritm klarar även bågar med negativa vikter, vilket Dijkstras algoritm inte gör.

(c) Dijkstras algoritm har en bättre tidskomplexitet än vad Bellman-Fords algoritm har och kräver färre beräkningar.

(d)

	A	B	C	D	E	F	G
A		2	6	1			
B	2			4	1		
C	6					1	6
D	1	4					
E		1				2	6
F			1		1		
G			6		6		

Design

Uppgift 3

(a) Först definierar vi ett kommando-interface:

```
interface DrawCommand {
    void execute(Plotter plotter);
    void undo();
}
```

Vi har här låtit plotter vara parameter till execute-anropet – det är OK om ni skickar in plotter som parameter till konstruktorn i era rit-kommandon (det är egentligen mest en smaksak – man kan tänka sig fall då man inte har tillgång till plottern när vi definierar våra kommandon, men vi kan alltså i uppgiften anta att vi har det).

Därefter kan vi deklarera en Template-klass med det som blir gemensamt för våra rit-kommandon:

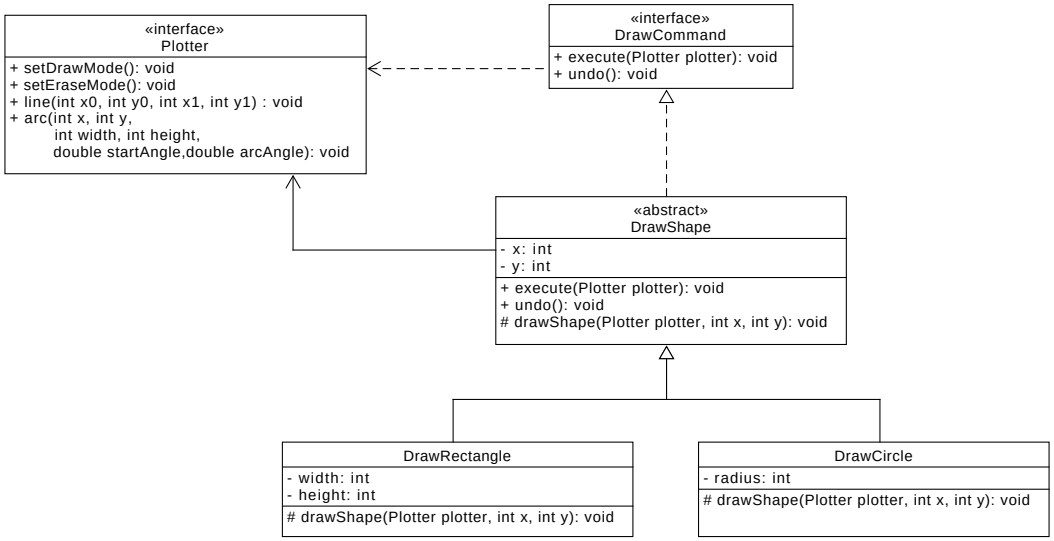
```
abstract class DrawShape implements DrawCommand {

    private Plotter plotter;
    private int x, y;

    protected DrawShape  (int x, int y) {
        this.x = x;
        this.y = y;
    }

    protected abstract void drawShape(Plotter plotter, int x, int y);

    public void execute(Plotter plotter) {
        this.plotter = plotter;
        plotter.setDrawMode();
        this.drawShape(plotter, x, y);
    }

    public void undo() {
        plotter.setEraseMode();
        this.drawShape(plotter, x, y);
    }
}
```

Vi har här låtit plotter och x- och y-koordinaterna vara private-deklarerade, och skickar dem som parameter till drawShape, vi skulle även ha kunnat låta dem vara protected och hämta dem direkt i våra subklasser när vi skall rita.

Nu räcker det att våra kommandon för att rita rektanglar och cirklar implementerar en konstruktor, och metoden drawShape:

```
class DrawRectangle extends DrawShape {

    private int width, height;

    public DrawRectangle  (int x, int y, int width, int height) {
        super(x, y);
        this.width = width;
        this.height = height;
    }

    protected void drawShape(Plotter plotter, int x, int y) {
        plotter.line(x - width/2, y - height/2, x + width/2, y - height/2);
        plotter.line(x + width/2, y - height/2, x + width/2, y + height/2);
        plotter.line(x + width/2, y + height/2, x - width/2, y + height/2);
        plotter.line(x - width/2, y + height/2, x - width/2, y - height/2);
    }
}

class DrawCircle extends DrawShape {

    private int radius;

    public DrawCircle  (int x, int y, int width, int radius) {
        super(x, y);
        this.radius = radius;
    }

    protected void drawShape(Plotter plotter, int x, int y) {
        plotter.arc(x, y, radius, radius, 0, 2*Math.PI);
    }
}
```

(b)



(c) Det som vi skall dekorera är alltså plottern, vi sparar den dekorerade plottern och vår PrintStream som parametrar, och implementerar metoderna i vår dekorerande Plotter genom att skriva ut och delegera vidare:

```
class LoggingPlotter implements Plotter {

    private Plotter plotter;
    private PrintStream output;

    public LoggingPlotter  (Plotter plotter, PrintStream output) {
        this.plotter = plotter;
        this.output = output;
    }

    public void setDrawMode( ) {
        output.println("Set Draw Mode");
        plotter.setDrawMode( );
    }

    public void setEraseMode( ) {
        output.println("Set Erase Mode");
        plotter.setEraseMode( );
    }

    public void line(int x0, int y0, int x1, int y1) {
        output.printf("Drawing line from (%d, %d) to (%d, %d)\n", x0, y0, x1, y1);
        plotter.line(x0, y0, x1, y1);
    }

    public void arc(int x, int y,
             int width, int height,
             double startAngle, double arcAngle) {
        output.printf("Drawing arc with radius %d around (%d, %d)\n");
        plotter.arc(x, y, width, height, startAngle, arcAngle);
    }
}
```

Uppgift 4

Vi behöver en builder-klass, och enklast är att låta den vara en publik statisk inre klass i Account. Vår builder-klass behöver ha:

- samma attribut som den omgivande klassen,
- metoder som låter oss sätta dessa attribut på ett säkert sätt, och
- en build()-metod som skapar ett objekt.

Vi vill dessutom ha ett enkelt sätt att skapa en builder – enklast är att ha en statisk metod i Account för att göra det (vi vill samtidigt ta bort den publika konstruktorn i klassen Account, så att man måste använda vår builder för att skapa konton).

```
class Account {

    private String accountNumber;
    private String customerId;
    private String bankBranch;
    private double balance;
    private double interestRate;
```

```
    private Account  (Builder builder) {
        this.accountNumber = builder.accountNumber;
        this.customerId = builder.customerId;
        this.bankBranch = builder.bankBranch;
        this.balance = builder.balance;
        this.interestRate = builder.interestRate;
    }

    // Här definieras typiska Account-metoder

    // ... utelämnat ...

    // Builder-relaterat:

    public static Builder builder() {
        return new Builder();
    }

    public static class Builder {

        private String accountNumber;
        private String customerId;
        private String bankBranch;
        private double balance;
        private double interestRate;

        public Builder accountNumber(String accountNumber) {
            this.accountNumber = accountNumber;
            return this;
        }
        public Builder customerId(String customerId) {
            this.customerId = customerId;
            return this;
        }

        public Builder bankBranch(String bankBranch) {
            this.bankBranch = bankBranch;
            return this;
        }
        public Builder balance(double balance) {
            this.balance = balance;
            return this;
        }
        public Builder interestRate(double interestRate) {
            this.interestRate = interestRate;
            return this;
        }
        public Account build() {
            return new Account(this);
        }
    }
}
```

Med hjälp av deklarationerna i (a) kan vi skapa vårt konto med:

```
class MainBuilder {

    public static void main(String[] args) {
        var account =
            Account
            .builder()
            .accountNumber("12-1334236")
            .customerId("645-564-654")
            .bankBranch("Lund")
            .balance(2015045)
            .interestRate(4.0)
            .build();
        // ...
    }
}
```

Uppgift 5

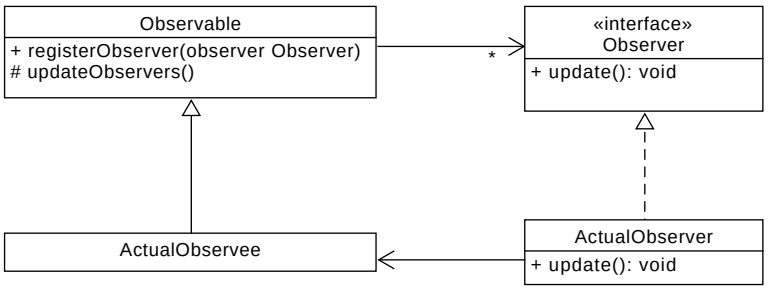
(a) **Coherence** Med coherence (eller cohesion, sammanhang) menar man hur pass stort beroende det finns mellan olika enheter i en modul. Saker som ofta används tillsammans passar bra tillsammans i en modul (hög coherence) medan saker som inte har med varandra (låg coherence) att göra och inte används tillsammans inte bör ligga i samma modul.

Coupling Moduler som har ett starkt beroende av varandra och inte fungerar på egen hand sägs vara starkt kopplade (hög coupling). Om det finns mycket sådana beroenden är det dåligt för då blir det svårare och otympligare att återanvända, eller ändra i, mjukvaran.

(b) Idén bakom Observer/Observable är att öppna upp för andra objekt (observerare – Observer) att bli informerade om saker som händer i ett objekt (den observerade – Observable) utan att den som skriver koden för det observerade objektet behöver känna till vilka objekt/klasser som i framtiden kommer att observera detsamma. på detta sätt undviker man beroenden från klassen/modulen som observeras till modulerna/klasserna/objekten som observerar denna.

Man åstadkommer detta genom att tillhandahålla ett interface som observatörerna kan implementera och som beskriver en metod som den observerade kan anropa när något händer. Observatörerna registrerar sig hos den observerade som sedan går igenom de registrerade observatörerna och anropar metoden i interfacet.

En enkel, nerskalad, version av mönstret kan se ut så här:



Här känner `ActualObserver` till `ActualObservee` och kan registrera sig hos denna och anropa dess metoder. `ActualObservee` behöver däremot inte känna till `ActualObserver` men kan ändå anropa dess `update()`-metod via superklassens metod `updateObservers()`.