Lösningar till tentamen i EDAF60, 28 oktober 2019

Christian Söderberg

Lösning 1

Det fanns en övre gräns på 75 ord på de första två deluppgifterna, för att ge er en uppfattning om ungefär hur noga ni skulle beskriva mönstren.

(a) Man kan naturligtvis beskriva Template Pattern på många sätt, ett förslag är (66 ord):

> *I Template Method definierar vi en klass som innehåller attribut och programkod som är gemensam för två eller flera potentiella subklasser (den blir ett slags mall-klass) – det som skiljer subklasserna åt implementeras i en eller flera metoder som deklareras (men inte implementeras) i superklassen, och som anropas från metoder i superklassen. Vi kan använda template method för att slippa upprepa kod i flera klasser (DRY).*

Det som vi vill få med i beskrivningen är följande punkter (vi gör markeringar för dem i rättningsprotokollet):

- Vi lyfter gemensam kod till superklass.
- Vi deklarerar i superklass metod(er) för det som skiljer subklasser åt.
- Vi anropar i superklassen metoder för det som skiljer subklasserna åt.
- Vi implementerar i subklasserna metoder för det som skiljer dem åt.
- Template Method kan hjälpa oss att undvika duplicerad kod (DRY).

Anmärkningsvärt många skrivande tar bara upp den första och sista punkten ovan i sina beskrivningar, men då har man i princip bara beskrivit vanligt arv (oavsett om superklassen är abstrakt eller inte) – för att det skall bli Template Pattern behövs de tre punkterna däremellan. Man får lite tröstpoäng för de första och sista punkterna, eftersom arv naturligtvis är en komponent i Template Pattern, men för full poäng måste man ha med de tre övriga punkterna (och om man beskriver de övriga punkterna behöver man inte ta med DRY-punkten, den är ju en följd av de andra punkterna).

Man kan egentligen använda Template Pattern både för att slippa upprepa kod, och för att skapa en mall för en eller flera tänkta framtida subklasser (det är egentligen därifrån namnet *Template* Pattern kommer), men det räcker med punkterna ovan i er beskrivning.

(b) På samma sätt som för Template Pattern kan Strategy Pattern beskrivas på många sätt, här är ett förslag till kortfattad beskrivning (62 ord):

> *Strategy är ett sätt att under exekvering välja hur något skall göras. Vi introducerar ett interface för hur man anropar någon algoritm, och låter en eller flera klasser implementera gränssnittet. Istället för att låta ett objekt själv implementera algoritmen, kan det nu anropa metoden i ett annat objekt som implementerar gränssnittet, och vi kan under exekvering välja vilket objekt det skall vara.*

Vi gör markeringar för följande punkter:

- Strategy gör det möjligt att välja algoritm vid exekvering.
- Vi har ett interface som beskriver hur vi anropar vår algoritm.
- Flera klasser kan implementera interfacet, och kan implementera algoritmen på olika sätt.

Förvånansvärt många har använt ett exempel med djur som antingen kan flyga eller inte kan flyga, men exemplet är tyvärr olyckligt valt eftersom dess poäng verkar vara att olika slags djur har olika förmågor (och därmed olika interfaces), medan poängen med Strategy är att de utbytbara klasserna har ett gemensamt interface, så att vi kan anropa dem på samma sätt.

(c) Anledningen att vi hade denna uppgift var att den skulle ge er en möjlighet att reflektera över:

- inlämningsuppgift 1, som är en viktig del av kursen,
- begreppet 'utvidgning' i OCP.

Tre naturliga utvidgningar för vilka vi har OCP:

- Nya ordtyper – vi kan enkelt implementera nya subklasser till `Word`, och dessutom skriva nya `WordFactory`-klasser som ger de nya typerna av ord. Alla gamla instruktioner kan hantera våra nya ord, eftersom de aldrig anropar någon metod som inte redan finns i `Word`-interfacet. Genom att skicka in våra nya `WordFactory`-objekt till programklasserna får vi ett program som använder den nya ordtypen utan att vi behöver ändra vår övriga kod.
- Nya program (subklasser till `Program`) som använder de instruktioner som redan finns, eller som använder instruktioner som är lätta att lägga till (se nedan).
- Nya instruktioner som inte är relaterade till orden, eller inte använder andra `Word`-metoder än de som redan definieras i programmet. Exempel på sådana instruktioner kan vara `Double` (dubblar värdet i en adress), `Panic` (avbryter programmet med ett felmeddelande), eller `JumpNotEq`. Tack vare att vi använder Command Pattern räcker det att vi defininerar en klass för varje ny instruktion för att vi skall kunna använda dem i våra program.

(d) Med utvidgning menar vi ett sätt att lägga ny funktionalitet till vårt program – ganska många har i sina svar föreslagit ändringar som innebär en refaktorisering av den befintliga koden (exempelvis en effektivare implementation av klassen `Memory`), men en refaktorisering innebär per definition att man ändrar befintlig kod utan att lägga till någon ny funktionalitet, så begreppet OCP kommer inte riktigt i spel[1]. Det som vi i första hand söker i uppgiften är utvidgningar som är relaterade till OCP, och som på grund av sin natur tvingar oss att ändra befintlig kod – typexempel på sådana är:

- Nya `Word`-instruktioner, exempelivs `div` och/eller `sub`, eller `lessThan` – dessa skulle vi behöva implementera i samtliga befintliga `Word`-klasser.
- Nya aritmetiska instruktioner, typ `Div` och `Sub`, eftersom de skulle kräva att vi implementerar nya metoder i våra gamla `Word`-klasser (se ovan).

Observera att det finns ett slags utväxling här, för varje ny metod/instruktion skulle vi tvingas skriva om många befintliga klasser.

Det går att hitta på andra rimliga 'utvidgningar' till projektet som kräver att vi ändrar på någon eller några få klasser i programmet, och man kan få poäng även för sådana utvidgningar, om man motiverar på vilket vis de bryter mot OCP.

Lösning 2

(a) Vi har *Command Pattern* (interfacet `Statement` med metoden `execute` som implementeras i subklasserna), och *Composite* (`Block` har en aggregering av andra `Statements`).

Några har svarat *Strategy* istället för Command, och det är delvis rätt, Command kan ses som ett slags Strategy, så man får lite poäng för det (däremot får man inte poäng för både Command och Strategy).

Andra svar som förekom var:

- *Template Pattern*, men det finns inga abstrakta metoder uppe i `Statement`, och alla subklasser implementerar `execute` själva, så det finns i diagrammet inget tecken på Template Method,
- *Decorator Pattern*, och visserligen kan aggregeringen från `Block` till `Statement` (som alltså visar att vi har Composite Pattern) se ut som aggregeringen i Decorator Pattern, men vi har i Decorator normalt multipliciteten 1 istället för `*` för aggregeringen, dessutom borde namnet på klassen varit något i stil med `DecoratedStatement` – namnet `Block` visar tydligt att det rör sig om en samling av satser.

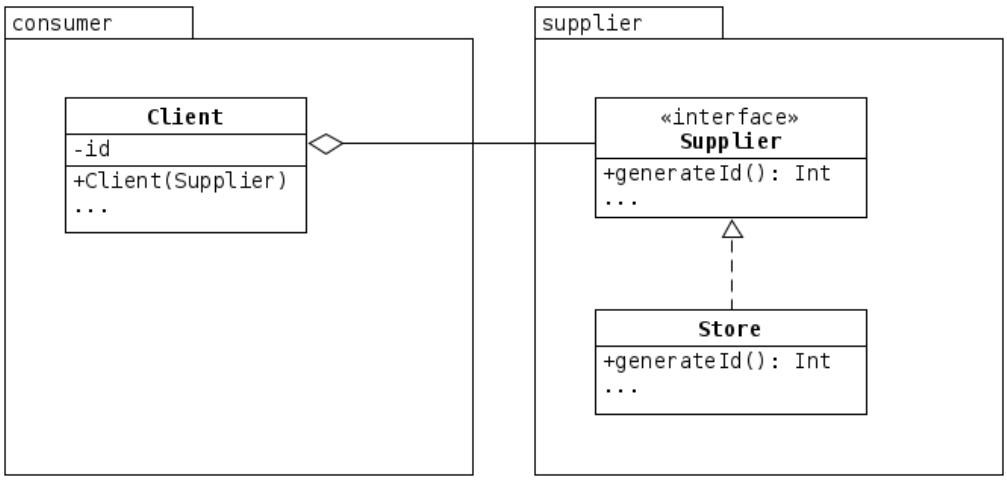
[1] Vår design är dock gjord för att underlätta även sådana ändringar, vi använder bland annat inkapsling för att göra klasserna så oberoende av varandra som möjligt (dvs ge dem så låg coupling som möjligt) – vi kan ändra vår i `Memory`-klass utan att någon annan klass behöver ändras. Men klassen `Memory` själv måste naturligtvis ändras.

(b) Det bryter mot *Dependency Inversion Principle*, som säger att vi bör programmera mot interfaces hellre än konkreta klasser.

Ganska många svarade *Single Responsibility Principle*, eftersom klassen `Store` genererar id-numret för klienterna, men det i sig är inget brott mot SRP – om en butik vill att dess klienter skall ha olika identiteter, så är det naturligt att butiken genererar dem (detta var inte gjort för att förvirra er, men många tycks ha blivit förvirrade av det).

Och oavsett om klienten själv genererar id-numret[2] eller inte, så vill vi ganska säkert ha en referens till butiken i vår klient, och då måste vi fortsätta kompilera om `Client` så snart `Store` kompileras om (så vi har inte hanterat problemet i frågeställningen).

(c) Vi inför ett interface `Supplier`, och låter `Store` implementera `Supplier` (i diagrammet har jag markerat `Supplier` som en aggregation, eftersom vi säkert vill spara en referens, det hade gått utmärkt att rita bara en beroendepil istället):



Vi får Dependency Inversion genom att låta `Client` bero av interfacet `Supplier` istället för av den konkreta klassen `Store` – vi kan nu använda olika slags `Supplier`-klasser, och kan kompilera `Client` utan att ha tillgång till någon specifik konkret klass.

Det var denna princip som gjorde att vi kunde kompilera den utdelade koden i XL-projektet innan vi hade skrivit någon egen klass för kalkylarket – tack vare att vi hade gränssnittet `Environment` inne i `expr`-paketet, kunde klasserna där anropa metoder som hämtade värdet i en given adress, trots att den klass som skulle hålla reda på alla adresser inte hade skrivits än.

Vi behöver kompilera om `Client` om interfacet `Supplier` ändras, men det är mycket mindre vanligt att ett interface ändras än att implementationen av en konkret klass ändras.

Många av dem som svarade SRP i (b), och även några av dem som svarade DIP, hade i (c) en lösning där man klippte bort kontakten mellan `Client` och `Store` helt och hållet, men det skulle antagligen innebära att vi tvingades låta någon/några andra klasser hålla reda på kopplingen mellan `Client` och `Store`, och det skulle i slutändan antagligen leda till högre coupling i programmet, och lägre cohesion.

Lösning 3

Tyvärr fanns det i snabbreferensen för `Stream` en olycklig beskrivning av hur `filter`-funktionen fungerar – funktionen returnerar egentligen en ström med de värden som uppfyller ett givet villkor, men vi kommer att ge rätt även för dem som tolkat funktionen 'tvärtom'.

(a)

```
List<String> wordsOfLength(String text, int wordLength) {
    return
```

[2] I så fall skulle vi behöva ett sätt att se till att inte två butiker får samma id, det är dock enkelt om vi använder exempelvis ett UUID.

```
        Arrays
        .stream(text.split(" "))
        .filter(s -> s.length() == wordLength)
        .collect(Collectors.toList());
}
```

(b)

```
int partialSum(List<String> strings, int min, int max) {
    return
        strings
        .stream()
        .filter(s -> isInt(s))
        .mapToInt(s -> toInt(s))
        .filter(n -> min <= n && n <= max)
        .sum();
}
```

Lösning 4

Vår klass `DecoratedLever` gör om gränssnittet för `Lever`-dekoratörerna så att det blir bättre anpassat för olika slags logging, det är ett exempel på ett standardmönster som brukar kallas *Adapter Pattern* (mönstret kallas ibland för *Wrapper*, vilket är lite lustigt, eftersom även Decorator Pattern ibland kallas Wrapper).

I uppgiften använder vi dessutom *Template Pattern*, när vi inför två abstrakta metoder som anropas från vår `raise()`-metod.

(a) Vi börjar med att introducera ett interface för spakar:

```
interface Lever {
    void raise();
}
```

Vi kan sedan definiera en abstrakt dekoratorklass som kapslar in en `Lever`, och ser till att vi alltid anropar `raise()` på denna lever, och att vi alltid anropar `preRaise()` precis före, och `postRaise()` precis efter:

```
abstract class DecoratedLever implements Lever {

    private Lever decoratedLever;

    public DecoratedLever  (Lever lever) {
        decoratedLever = lever;
    }

    protected abstract void preRaise();
    protected abstract void postRaise();

    public final void raise() {
        preRaise();
        decoratedLever.raise();
        postRaise();
    }
}
```

(b) Vi behöver bara skriva en konstruktor och de två log-metoderna:

```
class LoggingLever extends DecoratedLever {

    public LoggingLever  (Lever lever) {
        super(lever);
```

```
    }

    public void preRaise() {
        System.out.println("Before raising...");
    }

    public void postRaise() {
        System.out.println("After raising...");
    }
}
```

Vi kan anropa detta genom att skriva något i stil med:

```
void run() {
    Lever lever = new PrintingLever();
    Lever loggingLever = new LoggingLever(lever);
    loggingLever.raise();
}
```

(c) När vi anropar `raise()` på den yttre loggaren (`SwedishLogger`), så anropas metoden som vi ärvt ifrån `DecoratedLever`, och den anropar `preRaise()` i `this` (som alltså är en `SwedishLogger` här) – den första utskriften blir därför '`före`'. Därefter anropar koden i `DecoratedLever` `raise()` i den inkapslade spaken, och där anropas återigen först `preRaise()` – eftersom `this` denna gång är en `FinnishLogger`, så blir utskriften '`ennen`'. När vi denna gång anropar `raise()` kommer vi att göra det på en `PrintingLever`, så vi får utskriften '`Raising the lever`', varpå vi ramlar tillbaks till den inre loggaren (vår `FinnishLogger`), som kör sin `postRaise()` och ger utskriften '`mukaan`'. Till slut återvänder vi till vår yttre logger, som avslutar sin ärvda `raise()` genom att anropa `postRaise()` och skriva ut '`efter`'. Utskriften blir alltså:

```
före
ennen
Raising the lever
mukaan
efter
```

Lösning 5

(a) Denna uppgift är i allt väsentligt samma som uppgift 2 på seminarium 2 – den går att lösa väldigt enkelt med lambdauttryck, se längre ner i texten, men vi kan även lösa den med hjälp av abstrakta klasser.

Det som gör att `OriginalPlayer` inte är OCP är `if`-satserna i `handleButtonPress`, vi skulle vilja ha knappar som automatiskt anropar rätt metod på mp3-kretsen (alltså någon form av *Command Pattern*).

Ett sätt att göra det är att introducera en abstrakt klass `ActionButton`, som själv tar hand om knapptryckningar, och automatiskt anropar någon metod på kretsen när det sker. För att det skall fungera måste vi ha en referens till kretsen, så vi låter den bli ett attribut i vår klass:

```
abstract class ActionButton extends Button implements ButtonPressHandler {

    private Mp3Circuit circuit;

    public ActionButton  (String label, Mp3Circuit circuit) {
        super(label);
        this.circuit = circuit;
        addButtonPressHandler(this);
    }

    protected abstract void execute(Mp3Circuit circuit);

    public final void handleButtonPress(ButtonEvent e) {
```

```
        execute(circuit);
    }
}
```

Vi kan nu skapa en knapp för ”play/pause”:

```
class PlayPauseButton extends ActionButton {

    public PlayPauseButton  (Mp3Circuit circuit) {
        super(”play/pause”, circuit);
    }

    protected void execute(Mp3Circuit circuit) {
        circuit.togglePlay();
    }
}
```

De två knapparna för framåt och bakåt blir väldigt lika varandra, så vi kan lyfta upp en del gemensam kod till en superklass (till och med jag inser att det kan verka lite hysteriskt att introducera en extra superklass här, ni behöver inte göra det i era lösningar):

```
class SkipButton extends ActionButton {

    private int step;

    public SkipButton  (Mp3Circuit circuit, String label, int step) {
        super(label, circuit);
        this.step = step;
    }

    protected final void execute(Mp3Circuit circuit) {
        circuit.skip(step);
    }
}

class PreviousButton extends SkipButton {

    public PreviousButton  (Mp3Circuit circuit) {
        super(circuit, ”previous”, -1);
    }
}

class NextButton extends SkipButton {

    public NextButton  (Mp3Circuit circuit) {
        super(circuit, ”next”, 1);
    }
}
```

Vi kan nu skriva vår nya spelare som:

```
class SolidPlayer extends OmdGui {

    private Mp3Circuit circuit;

    public SolidPlayer  (Mp3Circuit circuit) {
        this.circuit = circuit;
        addButton(new PlayPauseButton(circuit));
        addButton(new PreviousButton(circuit));
        addButton(new NextButton(circuit));
    }
```

```
    public void addButton(ActionButton button) {
        add(button);
    }
}
```

Vi använder inte attributet `circuit` annat än i konstruktorn, så i princip skulle vi kunna ta bort det.

En betydligt enklare lösning är att använda lambdauttryck, om vi inför ett *Single Abstract Method*-interface (SAM) motsvarande den abstrakta metoden ovan:

```
interface Mp3CircuitAction {
    void execute(Mp3Circuit circuit);
}
```

så kan vi enkelt koppla ihop knappar och 'actions':

```
class SolidPlayer extends OmdGui {

    private Mp3Circuit circuit;

    public SolidPlayer  (Mp3Circuit circuit) {
        this.circuit = circuit;
        addButton("play/pause", c -> c.togglePlay());
        addButton("previous", c -> c.skip(-1));
        addButton("next", c -> c.skip(1));
    }

    public void addButton(String label, Mp3CircuitAction action) {
        var button = new Button(label);
        button.addButtonPressHandler(e -> action.execute(circuit));
        add(button);
    }
}
```

Denna lösning skulle även fungera bra om vi ändrade designen på uppgiften lite, så att `Button` var ett interface, och att vårt `Gui` returnerade nya knappar med en Factory-metod:

```
interface Gui {
    Button createButton(String label);     // skapar en ny knapp med given text
    void add(Button button);               // visar en knapp i GUI:
}
```

Vi har nu fått *Dependency Inversion* (DIP) för vårt GUI, och kan skriva:

```
class SolidPlayer extends OmdGui {

    private Mp3Circuit circuit;

    public SolidPlayer  (Mp3Circuit circuit) {
        this.circuit = circuit;
        addButton("play/pause", c -> c.togglePlay());
        addButton("previous", c -> c.skip(-1));
        addButton("next", c -> c.skip(1));
    }

    public void addButton(String label, Mp3CircuitAction action) {
        var button = createButton(label);
        button.addButtonPressHandler(e -> action.execute(circuit));
        add(button);
    }
}
```

Med `Button` som ett interface hade lösningen med abstrakta `Button`-subklasser inte fungerat lika enkelt som ovan (jag lät `Button` vara deklarerad som en klass i uppgiften för att göra den lite enklare att lösa även om man inte kom på idén att använda lambdauttryck).

(b) Beroende på vilken av ovanstående lösningar vi väljer får vi lite olika huvudprogram. Om vi inför våra `ActionButton`-subklasser så kan vi först implementera nya knappar:

```
class ChangeVolumeButton extends ActionButton {

    private int amount;

    public ChangeVolumeButton  (Mp3Circuit circuit, String label, int amount) {
        super(label, circuit);
        this.amount = amount;
    }

    protected final void execute(Mp3Circuit circuit) {
        circuit.changeVolume(amount);
    }
}

class VolumeDownButton extends ChangeVolumeButton {

    public VolumeDownButton  (Mp3Circuit circuit) {
        super(circuit, "volume down", -10);
    }
}

class VolumeUpButton extends ChangeVolumeButton {

    public VolumeUpButton  (Mp3Circuit circuit) {
        super(circuit, "volume up", 10);
    }
}
```

och sedan skriva huvudprogrammet som:

```
var mp3circuit = new OmdMp3Circuit();
var player = new SolidPlayer(mp3circuit);
player.addButton(new VolumeDownButton(mp3circuit));
player.addButton(new VolumeUpButton(mp3circuit));
```

Vi utnyttjar här den externa referensen till vår krets (`mp3circuit`), vilket inte är jättesnyggt, men det gör lösningen enklare.

Den andra lösningen ovan gör att vi inte behöver utnyttja `mp3circuit` i huvudprogrammet för att skapa våra knappar, vi kan skriva:

```
var mp3circuit = new OmdMp3Circuit();
var player = new SolidPlayer(mp3circuit);
player.addButton("volume up", c -> c.changeVolume(10));
player.addButton("volume down", c -> c.changeVolume(-10));
```