Tentamen i Objektorienterad modellering och diskreta strukturer

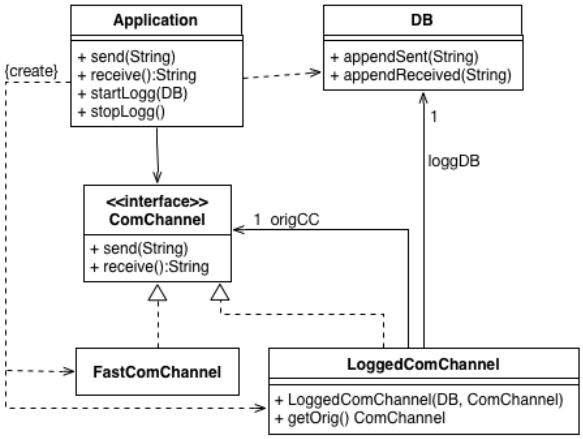
Lösningar

1. B, A, B, D

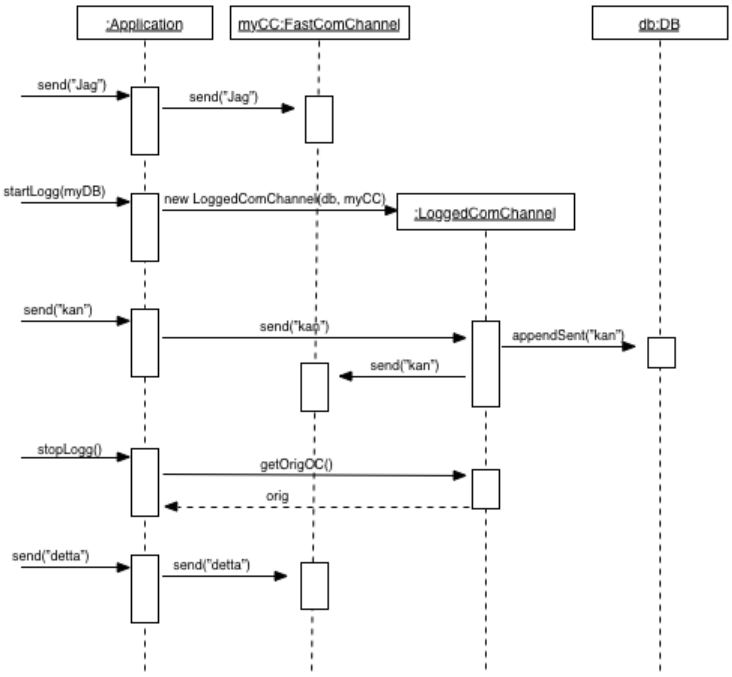
2. Template Method

3. **a**. Decorator

b. Klassdiagram



c. Sekvensdiagram



Problem 4

(a) Om vi skall få veta när något händer med ett underliggande värde bör vi i kursen använda *Observer-Observable*-mönstret (att bara testa när någon bestämmer sig för att anropa `currentYield()` räcker inte, det skulle inte varna oss ens om portföljen hade tvärdykt en längre tid, om vi inte valt att anropa `currentYield()`).

Både `Stock` och `Portfolio` skulle enligt uppgiften utvidga `Observable`, på något sätt måste vi även få `Portfolio` att observera ändringarna i de underliggande aktierna. Vi kan göra det antingen genom att låta `Portfolio` implementera `Observer` själv, eller genom att låta den skapa lokala objekt som ligger och lyssnar på de ingående aktierna – vi kan börja med det senare alternativet (här har jag använt ett lambda-uttryck för att registrera min observatör, man kan även explicit skapa en anonym klass eller till och med skriva och instantiera en namngiven inre klass):

```
class Stock extends Observable {

    private double value;

    void update(double value) {
        this.value = value;
        setChanged();
        notifyObservers();
    }

    public double value() {
        return value;
    }
}

class Portfolio extends Observable {

    private List<Stock> stocks = new LinkedList<>();
    private double buyValue;
    private double yieldLimit;

    public Portfolio  (double yieldLimit) {
        this.yieldLimit = yieldLimit;
    }

    public void add(Stock stock) {
        stocks.add(stock);
        buyValue += stock.value();
        stock.addObserver((obs,obj) -> {
                check();
            });
    }

    public double currentYield() {
        double value = stocks
            .stream()
            .mapToDouble(s -> s.value())
            .sum();
        return value / buyValue;
    }
```

```
    private void check() {
        if (currentYield() < yieldLimit) {
            setChanged();
            notifyObservers();
        }
    }
}
```

Ett annat alternativ är alltså att låta `Portfolio` själv implementera `Observer`, en möjlig invändning är att det ger klassen själv två uppgifter: både att hålla reda på portföljen, och att lyssna på aktierna. Det var dock helt OK att göra så i uppgiften, vi kan göra det så här (jag passar även på att byta ut användningen av `Stream` i beräkningen av portföljens värde mot en loop – personligen föredrar jag att använda en `Stream`, men det är en smaksak):

```
class Portfolio extends Observable implements Observer {

    private List<Stock> stocks = new LinkedList<>();
    private double buyValue;
    private double yieldLimit;

    public Portfolio  (double yieldLimit) {
        // som tidigare
    }

    public void add(Stock stock) {
        stocks.add(stock);
        buyValue += stock.value();
        stock.addObserver(this);
    }

    public void update(Observable obs, Object obj) {
        check();
    }

    public double currentYield() {
        // som tidigare
    }

    private void check() {
        // som tidigare
    }
}
```

I rättningsprotokollet framgår att man framförallt får poäng för den del av lösningen som har med *Observer-Observable*-mönstret att göra: 0.75 p för att `Portfolio` deklarerar något slags Observer (antingen själv, eller invärtes, som ovan), 0.90 p för att anropa `addObserver` på rätt objekt, och 0.90 p för att implementera `update(Observable, Object)` (för att inte missgynna dem utan snabbreferens var vi snälla när vi tittade på metodrubriken, så länge ni såg ut att förstå vad ni höll på med).

(b) Här kan vi först i klassen `Portfolio` göra följande ändringar:

- Ta bort attributet `yieldLimit` (och motsvarande parameter ur konstruktorn) – hela poängen med att göra ändringen är ju att vi vill bli av med dem från `Portfolio`.
- Ändra uppdateringen så att vi inte längre testar innan vi anropar `setChanged()` och `notifyObservers()`.

```
class Portfolio extends Observable {

    private List<Stock> stocks = new LinkedList<>();
    private double buyValue;

    public Portfolio  () {
        // nu tom!
    }

    public void add(Stock stock) {
        // som tidigare
    }

    private void update() {
        setChanged();
        notifyObservers();
    }

    public double currentYield() {
        // som tidigare
    }
}
```

Den test som vi tidigare gjorde inne i `Portfolio` flyttar vi till en ny klass, `PortfolioTracker`, som även den får utvidga `Observable`. Återigen måste någon lyssna, denna gång på en portfölj, och även denna gång kan vi välja om den skall vara `PortfolioTracker` själv, eller ett anonymt objekt inuti – här låter jag det vara ett objekt inuti (men alternativet går precis lika bra på tentan – lägg dock märke till att vi då i klass-rubriken även måste skriva `implements Observer`):

```
class PortfolioTracker extends Observable {

    private Portfolio portfolio;
    private double yieldLimit;

    public PortfolioTracker  (Portfolio portfolio,
                              double yieldLimit) {
        this.portfolio = portfolio;
        this.yieldLimit = yieldLimit;
        this.portfolio.addObserver((obs,obj) -> {
                PortfolioTracker.this.check();
            });
    }

    private void check() {
        if (portfolio.currentYield() < yieldLimit) {
            setChanged();
            notifyObservers();
        }
    }
}
```

Även i 4(b) är det huvudsakligen Observer-Observable som ger poäng, 0.60 p för att göra `addObserver` på rätt portfölj, och 0.45 p för att implementera något som ser ut som en `uppdate`-metod.

Problem 5

(a) Två klasser för att rita olika slags figurer kommer rimligtvis att bli ganska lika varandra, så vi börja med att definiera en klass `DrawShape`:

```
abstract class DrawShape implements DrawCommand {

    protected int x, y;
    protected Drawing d;

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

    protected abstract void shape();

    public void draw(Drawing d) {
        this.d = d;
        d.useForegroundColor();
        moveToAndDraw();
    }

    public void undo() {
        d.useBackgroundColor();
        moveToAndDraw();
    }

    private void moveToAndDraw() {
        d.moveTo(x, y);
        shape();
    }
}
```

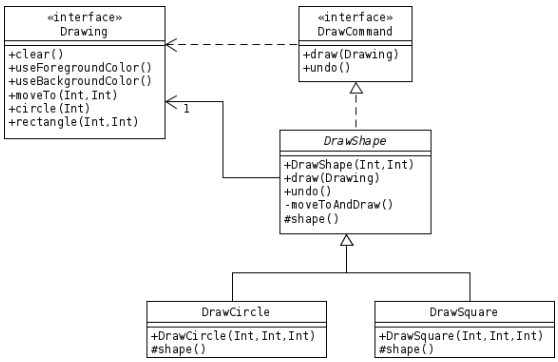
Med denna klass blir våra båda figur-klasser enkla:

```
class DrawCircle extends DrawShape {

    private int radius;

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

    protected void shape() {
        d.circle(radius);
    }
}

class DrawSquare extends DrawShape {

    private int side;

    public DrawSquare  (int x, int y, int side) {
```

```
        super(x, y);
        this.side = side;
    }

    protected void shape() {
        d.rectangle(side, side);
    }
}
```

Några har på tentan implementerat två separata, nästintill identiska, ritklasser – de får fungerande (eller nästan fungerande) program, och får en del poäng på uppgiften, men på en tenta i OMD kan vi inte blunda för så mycket kod-duplicering, så man kan bara få ungefär 50% av maxpoängen om man inte använder en gemensam superklass för de båda ritklasserna. Den mest naturliga lösningen är *Template Method* (som ovan), men även de som skriver en gemensam superklass, och sedan låter `draw` och `undo` i subklasserna överskugga superklassens `draw`- och `undo`-metoder och bara lägga till ett ritkommando får ganska mycket poäng.

- Klasserna ovan ser ut så här i UML:



Problem 6

(a)
```
class DrawAction implements Action {

    private DrawCommand command;

    public DrawAction  (DrawCommand command) {
        this.command = command;
    }

    public void execute(Drawing d, Stack<DrawCommand> history) {
        history.push(command);
        command.draw(d);
    }
}

class Undo implements Action {

    public void execute(Drawing d, Stack<DrawCommand> history) {
        if (!history.isEmpty()) {
            history.pop().undo();
```

```
        }
    }
}

class Exit implements Action {
    public void execute(Drawing d, Stack<DrawCommand> history) {
        System.exit(0);
    }
}
```

(b) Den metod `run` som drar runt hela systemet kan skrivas som:

```
class DrawDispatcher {

    void run(GUI gui, Drawing d) {
        Stack<DrawCommand> history = new Stack<>();
        while (true) {
            gui.next().execute(d, history);
        }
    }
}
```