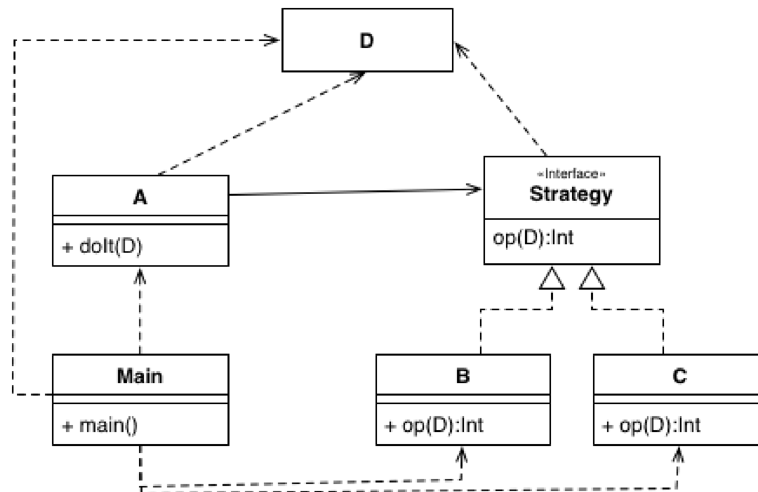
Lösningar till tentamen i EDAF60

24 oktober 2017

Lösning 1

(a)
```
public class A {
    private Strategy myStrategy;

    public void doIt(D d) {
        myStrategy.op(d);
    }
    public void setStrategy(Strategy newStrategy) {
        myStrategy=newStrategy;
    }
}

public class B implements Strategy {
    public int op(D d) {
        // omitted code
        return 25;
    }
}

public class C implements Strategy {
    public int op(D d) {
        // omitted code
        return 32;
    }
}

interface Strategy {
    int op(D)
}

public class D {
}

public class main {
    public static void main(String[] args) {
        A a = new A();
        a.setStrategy(new B());
        a.doIt(new D());
        a.setStrategy(new C());
        a.doIt(new D());
    }
}
```

(b) Klassdiagram



Lösning 2

ISP (Interface Segregation Principle). Klassen A implementerar innan designförändringen hela gränssnittet I, men efter förändringen endast delmängden I1, vilket uppenbarligen räcker eftersom inten viktig funktionalitet har ändrats. Det var m a o onödigt att implementera de delar som finns i I2, dvs man bröt tidigare mot ISP.

Lösning 3

Om vi bara tar med de metoder som nämns i uppgiften så får vi:

(a)
```
interface Company {
    Optional<Department> findDepartment(String deptCode);
}

interface Department {
    Optional<Employee> findEmployee(int empCode);
}

interface Employee {
    String name();
}
```

(b) Det enklaste sättet att lösa uppgiften är nog att bara koppla ihop `flatMap`, `map` och `orElse`-satserna som nedan, men man får gärna göra det i flera steg, och spara returvärdet efter varje steg (det ger inget poängavdrag).

```
String findName(Company company, String deptCode, int empCode) {
    return
        company
        .findDepartment(deptCode)
        .flatMap(d -> d.findEmployee(empCode))
        .map(e -> e.name())
        .orElse("Not found");
}
```

Observera att `findDepartment`-anropet ger oss en `Optional<Department>`, att `findEmployee`-anropet inne i `flatMap`-anropet ger oss en `Optional<Employee>`, men att `flatMap` 'lyfter upp' detta värde så att vi får en `Optional<Employee>` (istället för en `Optional<Optional<Employee»`), att `map`-anropet ger oss en `Optional<String>`, och att `orElse`-anropet slutligen ger oss en `String`, vilket är precis vad `findName`-metoden skall returnera.

Lösning 4

(a) Våra två Decorator-klasser kommer att bli lika varandra, vi kan undvika DRY genom att använda *Template Method*, och definera en abstrakt dekorator-klass (man måste inte göra så på tentan, men får någon liten poäng för det):

```java
abstract class DecoratedRobot implements Robot {

    protected Robot robot;

    public DecoratedRobot  (Robot robot) {
        this.robot = robot;
    }

    public final Point2D getPosition() {
        return robot.getPosition();
    }

    public final void move(double dx, double dy) {
        robot.move(dx, dy);
        decorate(dx, dy);
    }

    protected abstract void decorate(double dx, double dy);
}
```

Denna klass kan vi använda för att att skapa en enkel `RobotLogger`-klass:

```java
class RobotLogger extends DecoratedRobot {

    public RobotLogger  (Robot robot) {
        super(robot);
    }

    protected void decorate(double dx, double dy) {
        System.out.println(String.format("Moving to %s",
                                         robot.getPosition()));
    }
}
```

(b) Vår `RobotGuardian` måste hålla reda på sin ursprungsplats, sitt maximala avstånd från ursprungsplatsen, och vem som är supervisorn:

```java
class RobotGuardian extends DecoratedRobot {

    private Point2D base;
    private double limit;
    private RobotSupervisor supervisor;

    public RobotGuardian  (Robot robot,
                           Point2D base, double limit,
                           RobotSupervisor supervisor) {
        super(robot);
        this.base = base;
        this.limit = limit;
        this.supervisor = supervisor;
    }

    protected void decorate(double dx, double dy) {
        if (robot.getPosition().distanceTo(base) > limit) {
            warn();
        }
    }

    private void warn() {
        supervisor
```

```
            .alert(String.format("WARNING: The robot is now %s m from base.",
                                 robot.getPosition()));
        }
    }
```

(c) Som vanligt med Decorator Pattern kan vi dekorera våra dekoratorer, i detta fall kan vi dekorera en `RobotLogger` med en `RobotGuardian`, eller tvärtom:

```
VacuumCleaner vacuumCleaner = new VacuumCleaner(...);
controller.run(new RobotGuardian(new RobotLogger(vacuumCleaner),
                                 new Point2D(0,0),
                                 15,
                                 supervisor));
```

Lösning 5

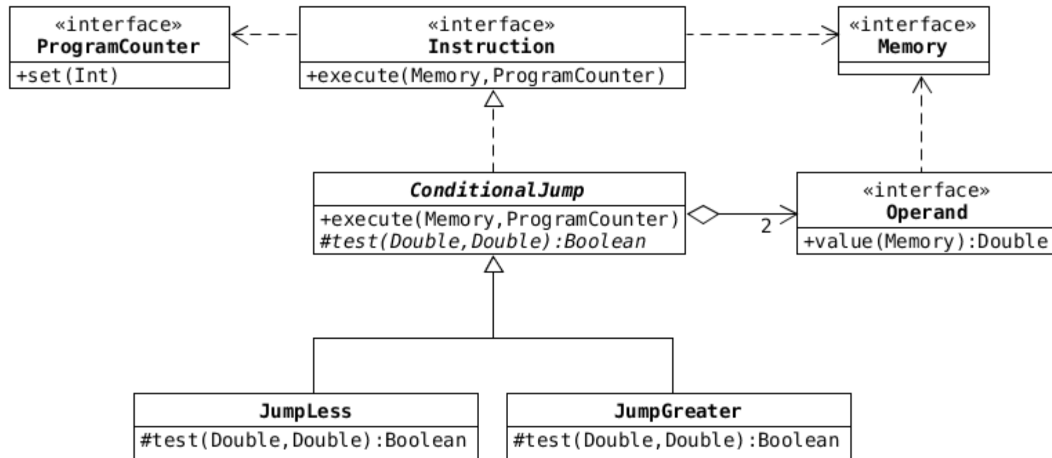
(a) Vi får först superklassen:

```
abstract class JumpConditional implements Instruction {

    private int label;
    private Operand op1, op2;

    public JumpConditional  (int label, Operand op1, Operand op2) {
        this.label = label;
        this.op1 = op1;
        this.op2 = op2;
    }

    protected boolean test(double lhs, double rhs);

    public void execute(Memory memory, ProgramCounter pc) {
        if (test(op1.value(memory), op2.value(memory))) {
            pc.set(label);
        }
    }
}
```

och med hjälp av denna kan vi skriva `JumpLess` som:

```
class JumpLess extends JumpConditional {

    public JumpLess  (int label, Operand op1, Operand op2) {
        super(label, op1, op2);
    }

    protected boolean test(double lhs, double rhs) {
        return lhs < rhs;
    }
}
```

(b) Det framgår inte av uppgiftstexten om `Instruction`, `Operand` och `Memory` är klasser eller gränssnitt – i figuren antar vi att de är gränssnitt:



Lösning 6

	Påstående	Anledning	Svar A,B,C, D,E	Motivering
T1	I figur 6.1 är Paket B helt stabilt.	Paket B beror inte på något annat paket.	A	Om ett paket inte beror pånågot annat paket behöver det inte kompileras om eller testas när andra paket ändras - det är stabilt.
T2	Class C kan ändra beteendet i Paket B utan att koden i Paket B behöver ändras.	Class B2 och Class C beror inte på varandra	C	Class C beror på Class B2, men eftersom inte B2 beror på C så kan C ändras utan att B och därmed paket B behöver ändras.
T3	Det enda sättet att följa OCP för en klass är genom att definiera subklasser för den.	Subklasser kan utvidga basklassens beteende utan att man behöver ändra i dess kod.	D	Delegation till ett interface som vid t.ex. strategimönstret är ett annat sätt att uppnå OCP
T4	Enligt SRP ska en klass bära hela ansvaret för det ansvarsområde den ansvarar för.	Enligt SRP ska det bara finnas en anledning för en klass att ändras	B	Både påståendet och anledningen är delar av definitionen av SRP som båda ska gälla, men där anledningen inte förklarar påståendet.
T5	Figur 6.2 visar två sätt att modellera ett Hus som består av Rum av olika typ. I alternativ A är det lättare att lägga till nya rumstyper än i alternativ B	I alternativ A medför abstraktionen ”Rum” att klassen Hus inte behöver känna till vilka rumstyper som finns.	D	Anledningen är ett korrekt uttalande, men det är lika enkelt att lägga till nya rumstyper i Alternativ B