Tentamen i EDAF60

24 oktober 2017

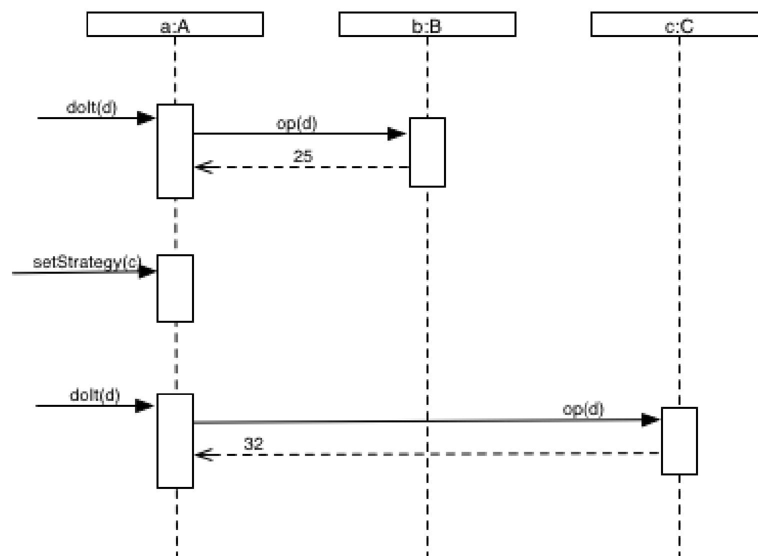
Skrivtid: 8-13

- Skriv bara på ena sidan av pappret – tentorna kommer att scannas in, och endast framsidorna rättas.
- Skriv **inte** med färgpenna – enda tillåtna färg är svart/blyerts.
- Skriv tydligt – om texten inte går att läsa kan du inte få några poäng.
- Sätt identitet och sidnummer på varje inlämnat blad, kontrollera att sidnumret på din sista sida är samma som det antal blad du markerar på omslagspappret.

Hjälpmedel:
- Andersson: UML–syntax
- Holm: Java snabbreferens

Uppgift 1

Nedan ett sekvensdiagram som visar exekveringen av en del av ett program där Strategy-mönstret har använts. ”d” är en referens till en instansiering av klassen D. Returvärdena ”25” och ”32” är av typen `int`.

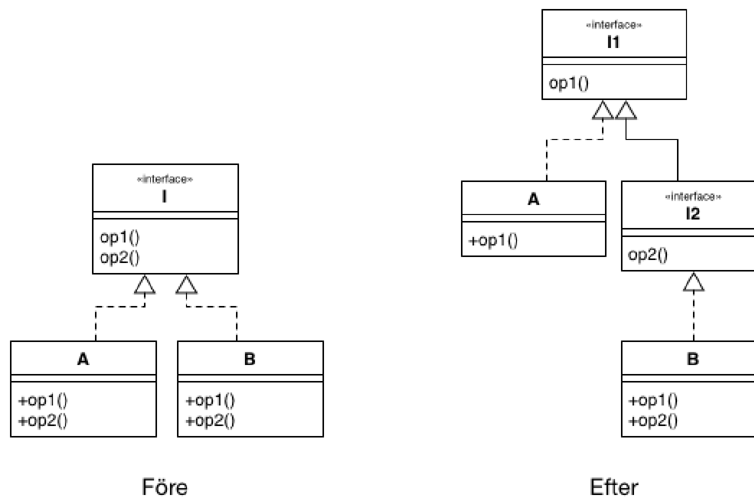


(a) Skriv Java-kod som resulterar i detta sekvensdiagram och där hela Strategy-mönstret är korrekt implementerat, inklusive ett huvudprogram som gör anropen till a. Du ska ta med allt som går att utläsa från sekvensdiagrammet och ibland kan du själv behöva hitta på namn på klasser, objekt, attribut som behövs för att få med allt. Du behöver däremot inte hitta på annan kod som inte bidrar till att göra sekvensdiagrammet komplett. *3 p*

(b) Rita klassdiagrammet för din lösning *2 p*

Uppgift 2

I klassdiagrammen nedan visas en ändring i design för att bättre följa en av de principer som lärts ut i kursen. Ingen för klasserna A och B viktig funktionalitet har ändrats. Vilken är principen? Motivera ditt svar! *2 p*



Uppgift 3

Denna uppgift behandlar Optional-mönstret, för att få poäng måste du använda Javas `Optional`-klass på lämpligt sätt i lösningen – en del av dess specifikation är:

```
class Optional<T> {
    public static <T> Optional<T> empty();
    public static <T> Optional<T> of(T t);
    public static <T> Optional<T> ofNullable(T t);
    public boolean isPresent();
    public T orElse(T defaultValue);
    public <U> Optional<U> map(Function<T, U> f);
    public <U> Optional<U> flatMap(Function<T, Optional<U>> f);
}
```

Vi har ett företag med olika avdelningar, varje avdelning har en avdelningskod (en sträng) och en grupp anställda, och varje anställd har ett namn.

För ett företag vill vi bland annat kunna:

- hämta avdelningen med en given kod (en sträng) – det är inte säkert att det finns en avdelning med den givna koden.

För en avdelning vill vi bland annat kunna

- hämta en anställd, med ett givet anställningsnummer (heltal) – det är inte säkert att det finns en anställd med det givna anställningsnumret.

För en anställd vill vi bland annat kunna

- hämta namnet.

(a) Definiera interfaces för `Company`, `Department` och `Employee` med bara de metoder som beskrivs ovan – använd `Optional` på lämpliga ställen.

2 p

(b) Implementera en metod med signaturen

```
String findName(Company company, String deptCode, int empCode)
```

som med hjälp av `Optional`-metoder ger namnet på en anställd med en given anställningskod på avdelningen med en given avdelningskod – om ingen sådan avdelning eller anställd finns skall texten `"Not Found"` returneras.

2 p

Uppgift 4

Vi har ett interface `Point2D` med följande specifikation:

```
interface Point2D {
    Point2D move(double dx, double dy);
    double distanceTo(Point2D other);
    String toString();
}
```

Metoden `toString()` ger punkterna som `(3.25, 4.80)`.

Vi har dessutom ett interface för robotar som kan röra sig på en plan yta:

```
interface Robot {
    Point2D getPosition();
    void move(double dx, double dy);
}
```

En färdig klass `VacuumCleaner` beskriver dammsugare (som är ett slags robotar):

```
class VacuumCleaner implements Robot {
    public VacuumCleaner  (Point2D pos);
    public void move(double dx, double dy);
    public Point2D getPosition();
}
```

För att köra våra robotar har vi ett interface `RobotController` med en metod `run` som styr roboten:

```
interface RobotController {
    void run(Robot robot);
}
```

Så för att köra en given dammsugare kan vi skriva:

```
controller.run(vacuumCleaner);
```

där `controller` är ett objekt som implementerar `RobotController` (exakt hur det går till behandlas inte i uppgiften).

Vi kan inte göra några ändringar i vår `VacuumCleaner`-klass, men vill kunna göra några extra saker med vår dammsugare, och med alla andra robotar vi har:

- Vi vill kunna logga deras rörelser (dvs göra en log-utskrift varje gång `move` anropas). Exempel på utskrift:

  ```
  Moving to (3.25, 4.80)
  ```

- Vi vill kunna bli varnade när de kommer för långt ifrån en given punkt. Exempel på utskrift:

  ```
  WARNING: The robot is now 15.05 m from base.
  ```

  Varningen skall ges varje gång `move` anropas och roboten hamnar för långt bort (så det kan bli många varningar).

Vi skall göra detta genom att använda *Decorator Pattern*, och vill kunna dekorera en godtycklig `Robot` – använd lämpliga mönster från kursen när du skriver din programkod för att lösa uppgifterna nedan.

(a) Skriv en klass `RobotLogger` som dekorerar en `Robot` genom att göra utskrifter så fort den rör sig (dvs varje gång `move` anropas). Vi vill kunna starta en körning med en `RobotLogger` på följande sätt:

```
VacuumCleaner vacuumCleaner = new VacuumCleaner(...);
controller.run(new RobotLogger(vacuumCleaner));
```

2 p

(b) Vi har ett interface `RobotSupervisor` som ser ut så här:

```java
interface RobotSupervisor {
    void alert(String message);
}
```

Skriv en klass `RobotGuardian` som dekorerar en `Robot` genom att anropa `alert` på en `RobotSupervisor` så snart roboten avlägsnar sig mer än en given sträcka från en given punkt. Vi vill kunna starta en körning med en `RobotGuardian` på följande sätt (vi varnar `supervisor` så snart dammsugaren är mer än 15 meter ifrån punkten `origin`, där `origin` är en `Point2D`):

```java
VacuumCleaner vacuumCleaner = new VacuumCleaner(...);
controller.run(new RobotGuardian(vacuumCleaner,
                                 origin,
                                 15,
                                 supervisor));
```

3 p

(c) Hur startar vi en körning med ett objekt som *både* loggar och varnar? *1 p*

Uppgift 5

Klassen `JumpLess` är implementerad på följande sätt:

```java
public class JumpLess implements Instruction {

    private int label;
    private Operand op1, op2;

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

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

Det finns en liknande klass, `JumpGreater`, där tecknet `<` bytts ut mot `>`. Metoden `value` i `Operand` returnerar ett `double`.

(a) Vi vill skriva om klasserna `JumpLess` och `JumpGreater` med hjälp av *Template Method* – subklasserna får inte ges tillgång till `memory` eller `pc`. Redovisa Javakod för den gemensamma superklassen och för klassen `JumpLess`. *3.5 p*

(b) Rita klassdiagram för den nya lösningen, de klasser, gränssnitt och metoder som framgår av uppgiften skall finnas med (du behöver inte ta med konstruktorer). *1.5 p*

Uppgift 6

Denna del innehåller uppgifter med påståenden och anledningar. För varje uppgift svara med ett av följande alternativ:

A Både påståendet och anledningen är korrekta uttalanden och anledningen förklarar påståendet på ett korrekt sätt.

B Både påståendet och anledningen är korrekta uttalanden, men anledningen förklarar inte påståendet.

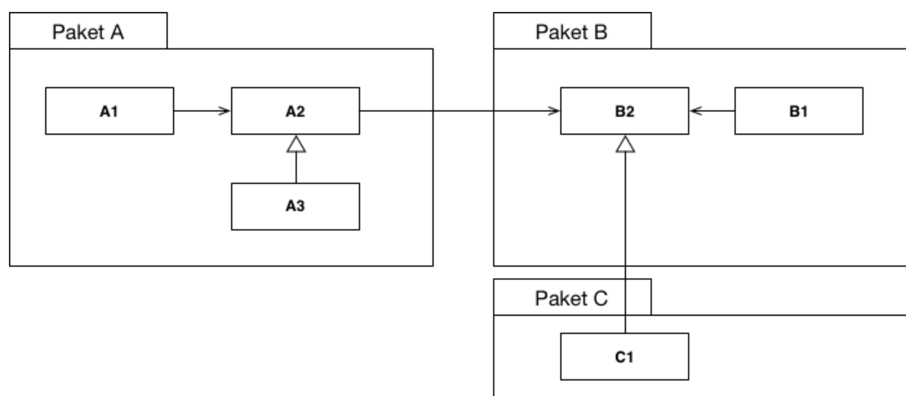
C Påståendet är ett korrekt uttalande, men anledningen är falsk.

D Påståendet är falskt, men anledningen är ett korrekt uttalande.

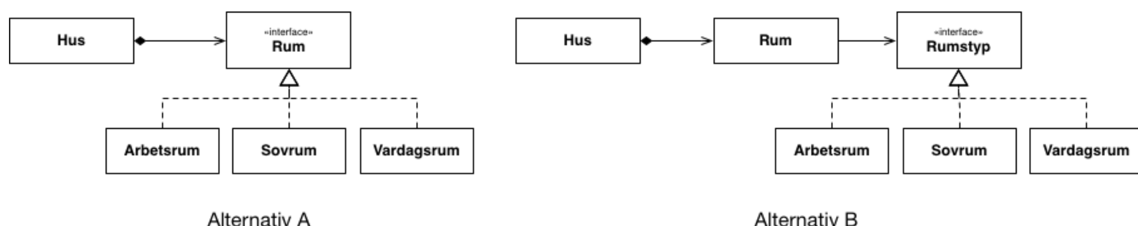
E Både påståendet och anledningen är falska.

Det går bra att svara direkt i formuläret. **Glöm då inte att lämna in denna sida av formuläret tillsammans med övriga lösningar.** *5 p*

	Påstående	**Anledning**	**Svar A,B,C, D,E**
T1	I figur 6.1 är Paket B helt stabilt.	Paket B beror inte på något annat paket	
T2	Klassen C1 kan ändra beteendet vid exekvering av kod i Paket A, utan att Paket B behöver modifieras. (figur 6.1)	Class B2 och Class C1 beror inte på varandra	
T3	En klass `Vehicle` bryter mot OCP. Är enda sättet att följa OCP att skapa subklasser till denna klass.	Subklasser kan utöka basklassens beteende utan att man behöver ändra i dess kod.	
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	
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.	



Figur 6.1 till T1



Figur 6.2 till T4