Tentamen i Objektorienterad modellering och design (OMD) Helsingborg

EDAF25

3 juni 2019, 8 – 13

- Skriv bara på ena sidan av pappret – tentorna kommer att scannas in, och endast framsidorna rättas.
- 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.

Tentamen består av uppgifter med totalt 30 poäng. För godkänt betyg kommer att krävas högst hälften av poängen. Vid bedömningen kommer hänsyn att tas till lösningens kvalitet. UML-diagram skall ritas i enlighet med UML-häftet.

Hjälpmedel:
- Andersson: UML–syntax

Uppgift 1 12 P.

Denna uppgift innehåller ett antal deluppgifter, där varje deluppgift har ett påstående och en anledning. För varje deluppgift, 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 ett felaktigt uttalande.

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

E Både påståendet och anledningen är felaktiga uttalanden.

- Svara inte i uppgiftshäftet, utan skriv ditt svar på ett vanligt lösningspapper.
- Varje korrekt besvarad deluppgift ger 1.0 poäng
- Vi vill inte att ni skall chansa, så ni får *minuspoäng* (-0.25 poäng) på en deluppgift om ni svarar fel, men aldrig minuspoäng totalt sett på hela uppgift 1.

	Påstående	Anledning
T1	OMD handlar ytterst om att fördela ansvar och samarbete mellan klasser av objekt. Önskvärda egenskaper är hög grad av sammanhang (cohesion) och en låg grad av koppling (coupling).	Cohesion mäter graden av sammanhållning mellan elementen i en enskild modul eller klass. Hög grad av sammanhang (cohesion) och lös/låg koppling (coupling) ger mjukvara som är lätt att utvidga eller dela upp och återanvända eftersom inte alla delar är inbördes beroende av varandra.
T2	Det är viktigt att ett UML-klassdiagram innehåller alla detaljer som rör implementeringen av de ingående klasserna.	Det bör gå att entydigt generera ett komplett program utifrån ett UML-klassdiagram.
T3	Refaktorisering är ett led i TDD.	TDD leder till hög koppling (coupling) mellan modulerna i mjukvaran.
T4	Att ersätta ett konstruktor-anrop med en fabriksmetod är ett exempel på en refaktorisering.	Refaktorisering innebär att ändra strukturen på koden utan att ändra dess beteende.
T5	Hög seghet (viscosity) är ett exempel på illaluktande design (bad smells).	Seghet är ett mått på hur jobbigt det är att implementera lösningar på 'rätt' sätt (bibehålla en god design). Om segheten är hög innebär det att det finns genvägar (hack) som är lättare att ta för att lösa ett problem vilket i sin tur leder till att designen försämras.
T6	Dekoratörsmönstret erbjuder en möjlighet att förändra ett objekts beteende dynamiskt.	Dekoratörsmönstret lägger till ytterligare ansvar till instanser av en klass.
T7	Både Strategimönstret (Strategy) och Mallmetodmönstret (Template Method) erbjuder sätt att följa OCP (Open Closed Principle).	Båda öppnar för logisk variation: Strategimönstret förändrar logiken för enskilda objekt medans Mallmetodmönstret gör det för hela klasser.
T8	Fabriksmetodmönstret (Factory Method Pattern) använder fabriksmetoder för att kunna skapa objekt utan att specificera den exakta klassen för objektet som ska skapas.	Fabriksmetoder är exempel på mallmetoder i Mallmetodmönstret (Template Method Pattern) där skapandet av objektet utgör variationspunkterna som kapslas in och lyftas ut ur kontexten.
T9	ISP (Interface Segregation Principle) liknar SRP (Single Responsibility Principle) men tar modulanvändarens(klientens) perspektiv snarare än moduldesignerns.	Brott mot ISP innebär även brott mot SRP. En klass kan inte ha ett tydligt ansvar om den implementerar flera interface.
T10	Om man vill ha en klass av objekt som kan övervakas av andra klasser av objekt i ett program utan att den blir tätt kopplad till dessa kan man använda sig av 'Observer/Observable' mönstret.	Kopplingen mellan en observatör och det observerade objektet kan göras i observatörens konstruktor.

	Påstående	Anledning
T11	I 'observer'-mönstret finns två sätt att propagera data från det observerade objektet till dess observatörer, antingen genom att det observerade objektet skickar data till observatören (push-modellen), eller genom att observatören hämtar data från det observerade objektet (pull-modellen). Push-modellen ökar kopplingen mellan det observerade objektet och dess observatörer.	I Push-modellen kommer observatörerna att behöva beräkna vilka förändringar som gjordes eller anta att något har förändrats.
T12	När man tillämpar Tillståndsmönstret måste man välja vilka klasser som ska vara stängda för förändring när systemet vidareutvecklas.	Tillståndsövergångarna kommer att kräva beroenden antingen mellan de olika tillståndsklasserna eller från tillståndsklasserna till kontexten.

Uppgift 2 6 P.

Studera följande Java-kod.

```
protected abstract class Employee {
   private String name;
   //Omissions
   protected abstract String role();
   public String hallo() {
      return "Hej! Jag heter " + name + " och är " + role();
   }
}
public class PM extends Employee {
    //Omissions
   protected String role() {
     return "project manager";
   }
}
public class Dev extends Employee {
    //Omissions
    protected String role() {
        return "developer";
    }
}
```

a) Gör en bättre design med hjälp av `Strategy`-mönstret, där bl a följande kodsekvens ska fungera:

```
Employee emp = new Employee("Ulf");
Role r = new PM();
emp.setRole(r);
emp.hallo();
```

Redovisa med Java-kod.

b) Det finns också en klass `Company` som har en metod `lastIn()` som returnerar ett objekt av typen `Employee` (den senast anställde på företaget). Om företaget inte har någon anställd returneras null. Metoden `lastIn()` används i sin tur av klassen `UnionReports` i en en metod `countRoles()` som presenterar en sammanställning av hur många utvecklare vs hur många projektledare som är senast anställda på sina respektive företag.

Klasserna `Company` och `Employee` finns i samma paket (`company`) medans `UnionReports` är ett exempel på användning av detta paket. På vilket sätt kan ovan beskrivna design vara ett problem för användaren av paketet? Föreslå en bättre design som utnyttjar antingen *Null Object*-mönstret eller Javas `Optionals`. Redovisa din lösning i ord och klassdiagram tillräckliga för att beskriva hur designen löser problemet.

Uppgift 3 7 P.

Använd *komposit-* och *kommando*-mönstret för att representera instruktioner i ett datorprogram enligt följande beskrivning:

Det finns tre sorters instruktioner, *hoppinstruktioner, adderare* och *makron*. Alla instruktioner behöver en referens till datorns minne. En hoppinstruktion behöver ett index som anger vilken instruktion som ska utföras i nästa exekvering. En adderare har dels en adress som pekar ut en plats i minnet och dels ett värde som läggs till det värde som redan finns på utpekad minnesplats. Ett makro innehåller en eller flera instruktioner.

a) Gör ett fullständigt klassdiagram för denna beskrivning som visar alla klasser, relationer (inklusive riktning, typ och multiplicitet), attribut och metoder.

b) Rita ett objektdiagram som exemplifierar designen ovan som innehåller en instruktion av varje typ.

Uppgift 4 5 P.

Vid en stor stadsfestival behöver trafiken förbi festivalområdet dirigeras om. Arrangörerna vill räkna ut hur stor påverkan festivalen kommer att ha på den normala trafiken (Hur många fordon kommer att behöva välja en alternativ rutt?). Normalt passerar i genomsnitt 2000 fordon per dygn genom området.

Under festivalen finns det bara en väg in (vid Knutpunkten) respektive ut (korsningen Hälsovägen/Kopparmöllegatan) ur festivalområdet. Vägarna mellan infarten och utfarten har en begränsad framkomlighet och alla är enkelriktade. Trådgårdsgatan är farbar från Knutpunkten till Essenska villan. Därifrån kan man sedan följa Bergaliden till Skolmuseet och vidare Kopparmöllegatan till festivalens utfart. En alternativ väg går från knutpunkten via Järnvägsgatan till Stortorget och vidare längs Drottninggatan till Stadsteatern. Från stadsteatern kan man följa Hälsovägen till festivalens utfart. Från Stortorget kan man dessutom välja att åka via Södra Storgatan till Essenska villan eller via Möllegränd till Skolmuseet. Slutligen är det också möjligt att ta sig från skolmuseet till statsteatern via Nedre Långvinkelsgatan.

De farbara vägarna i ovanstående förslag har kapaciteter enligt följande: Järnvägsgatan, Trädgårdsgatan, Kopparmöllegatan och Hälsovägen klarar var och en 1000 fordon per dygn medans framkomligheten på övriga vägar är mer begränsad. Bergaliden tar 800 fordon per dygn, Möllegränd 600, Drottninggatan och Nedre långvinkelsgatan 400 vardera och slutligen Södra storgatan 300.

a) Beskriv detta som ett grafproblem. Det vill säga, översätt problembeskrivningen till något av de typiska grafproblem vi har tittat på i kursen. Ange vilket grafproblem som behöver lösas samt vad noder och bågar och deras eventuella värden representerar.

b) En algoritm som kan användas för att lösa problemet är Ford Fulkersons algoritm. Föreslå en lämplig representation av grafen vid implementation av algoritmen.

c) Algoritmen använder sig av residualgrafen för att lösa problemet. Visa hur den inititala residualgrafen i exemplet skulle se ut enligt föreslagen grafrepresentation.

d) I första steget av algoritmen finner man en möjlig väg via Trädgårdsgatan, Bergaliden och Kopparmöllegatan. Hur ser residualgrafen ut efter första steget? Motivera ditt svar.

Lösningsförslag till uppgift 1

T1 A, Högt sammanhang leder till lösare kopplingar mellan de ingående modulerna. Desto högre sammanhang i en applikation desto mer väldefinierat är ansvaret för varje modul. Ju mer väldefinierat en moduls ansvar är desto lättare är det att lyfta den ut ur sin kontext och låta den göra samma jobb i en annan kontext. Modulen kan sköta sitt jobb utan att veta något om kontexten. Genom att tillämpa principer och mönster kan vi få en bättre design.

T2 E, Syftet med UML är visualisering för att göra det möjligt att förstå komplexiteten i mjukvaran. Därför är det viktigt att detaljnivån är anpassad till situationen. UML kan användas på olika sätt 1) Konceptuellt, som snabba skisser (hand-ritade på whiteboards) som ett hjälpmedel vid designdiskussioner 2) Specificerande, som specifikation före implementering eller i efterhand (reverse-engineering), för att förstå existerande kod eller 3) Implementerande, som ett programspråk (kompletta exekverbara specifikationer av ett mjukvarusystem. Detaljnivån varierar för dessa tre scenarier.

T3 C, En effekt av TDD är att man tvingas fokusera på gränssnitt och funktionalitet vilket i sin tur leder till att man tvingas till låg coupling av mjukvaran. Refaktorisering är ett led i TDD och behovet av refaktorisering en konsekvens av TDD.

T4 A, Refaktorisering görs för att förbättra koden med avseende på kvaliteter som t.ex. begriplighet, flexibilitet, återanvändbarhet och utvidgningsbarhet och är vanligtvis kopplad till en specifik förbättring av designen. Ytterst handlar det om att genomföra små beteendebevarande omstruktureringar (refaktoriseringar) av koden, en i taget. Exempel på refaktoriseringar kan vara:

- Ta bort duplicerad kod.
- Extrahera en hjälpmetod utav en del av en lång metod.
- Ersätta ett konstruktor-anrop med en fabriksmetod

T5 A

T6 D Strategimönstret erbjuder en möjlighet att *förändra* ett objektets beteende (genom valet av implementation/strategi) medans Dekoratörsmönstret *lägger till* ytterligare beteenden till objektet dynamiskt utan att förändra beteendet för det ursprungliga objektet. I det första fallet påverkas alla konsumenter av förändringen medans i det senare fallet kan tillägg göras utan att de ursprungliga konsumenterna av objektet påverkas.

T7 A Både Strategimönstret och Mallmetodmönstret erbjuder sätt att följa OCP. De skiljer sig dock i granularitet och metod. Mallmetodmönstret använder arv för att variera delar av en algoritm. Strategimönstret använder delegation för att variera hela algoritmen. Strategimönstret modifierar

logiken för enskilda objekt medan mallmetodmönstret modifierar logiken för en hel klass.

T8 C Fabriksmetodmönstret (Factory Method Pattern) använder fabriksmetoder för att kunna skapa objekt utan att specifiera den exakta klassen för objektet som ska skapas. En fabriksmetod är inte en mallmetod utan liknar mer hjälpmetoderna (Hooks) i Mallmetodmönstret (Template Method). Däremot kan de anropas från en mallmetod och fabriksmetodmönstret används ofta tillsammans med en variant av mallmetodmönstret.

T9 C ISP säger att ett interface ska vara sådant att dess användare/klienter inte ska vara tvingade att bero på metoder som de inte använder. Anledningen är att man vill undvika onödiga beroenden. Om en klient beror på en klass som innehåller metoder som klienten inte använder, men som andra klienter använder kommer denna klient ändå att påverkas av förändringar som dessa andra klienter påtvingar klassen. Ett sätt att undvika sådana beroenden/kopplingar är att separera interfacen.

ISP liknar SRP men tar modulanvändarens(klientens) perspektiv snarare än moduldesignerns. En klass kan ha ett tydligt ansvar även om den implementerar flera interface. Ett exempel är Sheet i XL-projektet. Användaren i Expression-paketet behövde bara de metoder som Environment-interfacet tillhandahöll alltså behövdes inget särskilt Sheet-interface. Att ersätta Environment-interfacet med ett sådant skulle bryta mot ISP men inte nödvändigtvis mot SRP som handlar om Sheet-klassens ansvar i det här fallet (som i sin tur utgör ett gränssnitt gentemot GUIt ...).

T10 B Om man vill ha en klass som övervakas av andra klasser i ett program utan den klassen blir tätt kopplad till dessa andra klasser kan man använda sig av 'observer/observable' mönstret.

Man vill t.ex. att vissa ändringar i klassens tillstånd ska förmedlas till resten av programmet. För att göra detta måste någon form av metod anropas. Dock behöver inte den observerade klassen veta vilken typ av objekt den pratar med. Genom att låta alla observatörer implementera ett gränssnitt, `Observer`, räcker det att den observerade klassen känner till dess metoder.

För att hålla reda på vilka observatörer som är intresserade behöver den observerade klassen ha en lista över dessa och någon metod, `addObserver`, som gör det möjligt för en observatör att börja prenumerera på uppdateringar. När en uppdatering sker går den observerade klassen igenom listan av observatörer och informerar dem om ändringen genom att anropa dess anmälningsmetod.

T11 C I 'observer'-mönstret finns två sätt att propagera data från det observerade objektet till dess observatörer, antingen genom att det observerade objektet skickar data till observatören (push-modellen), eller genom att observatören hämtar data från det observerade objektet (pull-modellen).

Push-modellen kräver att det observerade objektet känner till vilken data observatörerna behöver. Detta ökar kopplingen mellan subjekt och observatörer. Pull-metoden minskar kopplingen men kan vara mindre effektiv. Observatörerna kommer att behöva beräkna vilka förändringar som gjordes eller anta att något kunde ha förändrats. Observera att i båda fallen måste observatörerna ha kunskap om det observerade objektet (antingen för att tolka parametern som skickats i uppdateringsmetoden eller för att fråga efter dess uppdaterade tillstånd). Denna koppling kan minskas om observatörerna känner till det observerade objektet via en abstrakt klass.

T12 A: I tillståndsmönstret kan tillståndsövergångarna skötas antingen i tillståndsklasserna eller i kontexten. Beslutet om var man ska lägga dem är ett beslut om vilka klasser som ska vara stängda för förändring (tillståndsklasserna eller kontexten) när systemet vidareutvecklas.

Nackdelen med att ha dem i tillståndsklasserna är att man då skapar beroenden mellan dessa. Läggar man dem i kontexten blir tillstånden istället beroende av kontexten.

Lösningsförslag till uppgift 2

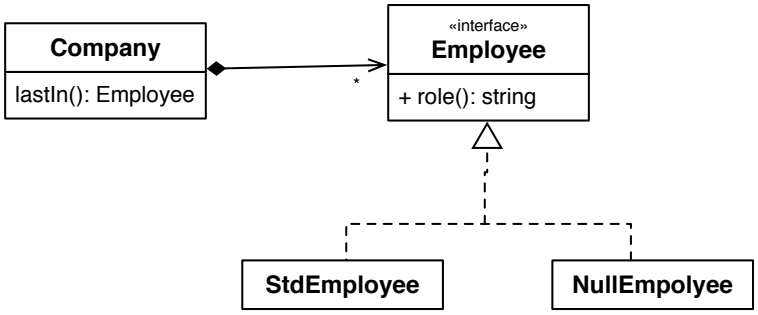
a)

```
public class Employee {
   private String name;
   private Role role;

   public Employee(String name) {
      this.name = name;
      this.role = role;
   }
   public void setRole(Role newRole) {
      this.role = newRole;

   public string role(){
      return role.title();
   }

   public String hallo() {
      return "Hej! Jag heter " + name + " och är "
                                        + role.title();
   }
}

public interface Role {
   string title();
}

public class PM implements Role {
   public String title() {
      return "project manager";
   }
}

public class Dev implements Role {
    public String title() {
        return "developer";
    }
}
```

Strategiinterface med strategimetod (1p) implementeras av två strategier (1p). I klassen Employee som nu är konkret finns en metod för att byta strategi (1p).

b) I den givna designen måste användaren av paketet hålla reda på om null är ett möjligt returvärde för metoderna hen använder och tänka på att hantera null. En miss kan leda till svårupptäckta fel. (1P)

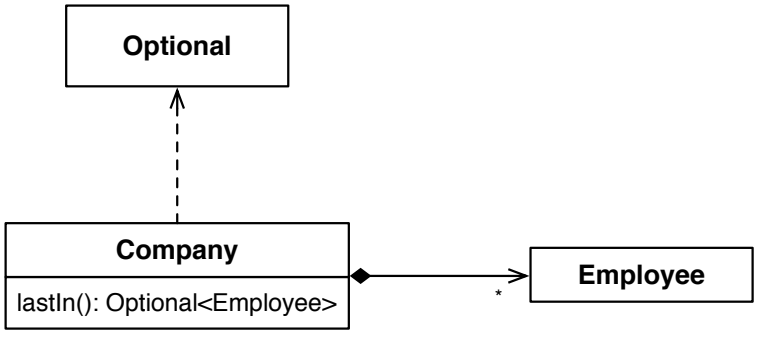
I en lösning med *Null Object*-mönstret bestämmer utvecklaren av `company-paketet` vad som är ett rimligt beteende för en icke-befintlig anställd:



(1p)

Istället för null returnerar metoden `lastIn()` ett objekt av typen `NullEmployee` om företaget saknar anställda. `NullEmployee` implementerar ett neutralt beteende för role() t.ex. kan strängen "None" returneras. Användaren behöver inte själv göra en nullkontroll. (1p)

I en lösning med javas `Optional` tvingas användaren av `company-paketet` definiera vad som är ett rimligt beteende för en icke-befintlig anställd:

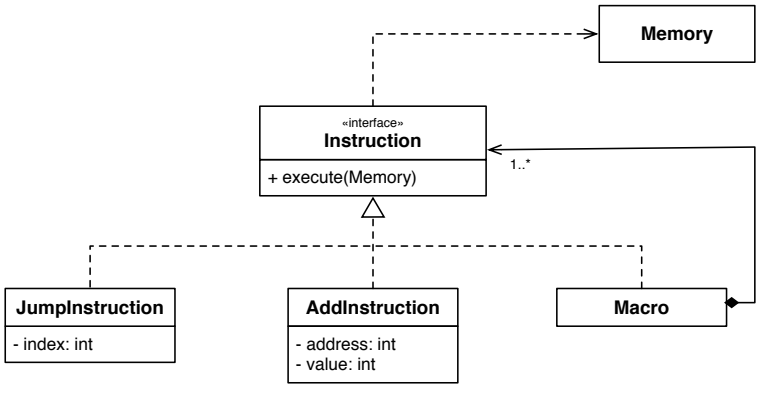


(1p)

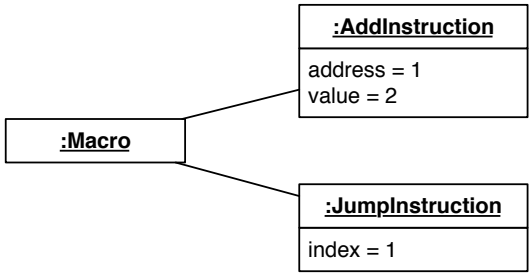
Med denna lösning returnerar metoden `lastIn()` alltid ett objekt av typen `Optional<Employee>` oavsett om det finns en anställd eller om företaget saknar anställda. Detta objekt kan vara tomt eller innehålla en `Employee`. Användaren kan då inte skippa (eller missa) en null-kontroll utan kan behöver utnyttja metoderna i `Optional`-klassen för att definiera hur befintliga respektive icke-befintliga anställda ska hanteras. (1p)

Lösningsförslag till uppgift 3

a)



b)



- Interface `Instruction` med en metod `execute()` (1p)
- Tre subklasser `JumpInstruction, AddIstruction` och `Macro` (2p)
- attribut som beskriver subklassernas parametrar (1p)
- Komposition med multiplicitet (1p)
- Objektdiagram (2p)

Lösningsförslag till uppgift 4

a) Vi vill beräkna det maximala flödet i ett nätverk. (1p) Genom att ta reda på hur många bilar som kan passera området på ett dygn kan vi uppskatta hur många av de normalt 2000 fordon som passerar som behöver välja en alternativ väg. Låt det aktuella vägnätet representeras av en riktad graf där noderna representerar korsningar och bågarna vägar (1p). Bågarnas kapacitet motsvarar antal fordon per tidsenhet som maximalt kan passera. Låt infarten vara källnod och utfarten vara sänka.

b) I Ford Fulkersson brukar man forutsatta att noderna ar numrerade fran 0 och uppat och att natverket representeras av en heltalsmatris. I matrisen anger elementet pa rad i och kolumn j kapaciteten for den bage som forbinder noderna i och j. Om det saknas bagar mellan vissa par av noder sa ar motsvarande element i matrisen 0.(1p)

c) Residualgrafen visar upp resterande kapacitet. Initialt har inget flöde lagts på grafen och därför är residualgrafen samma som den ursprungliga grafen. (1p)

Noder	Knutpunkten	Essenska villan	Skolmuseet	Stortorget	Stadsteatern	Utfart
Knutpunkten	0	1000	0	1000	0	0
Essenska villan	0	0	800	0	0	0
Skolmuseet	0	0	0	0	400	1000
Stortorget	0	300	600	0	400	0
Stadsteatern	0	0	0	0	0	1000
Utfart	0	0	0	0	0	0

d) Eftersom flaskhalsen längs denna väg är Bergaliden, med en kapacitet på 800 bilar per dygn, läggs ett tänkt flöde av storlek 800 på längs hela sträckan. I residualgrafen syns detta genom att 800 dras från det ursprungliga värdet från de tre vägarna och 800 läggs till i motsatt riktning för var och en av de tre. (1p)

Noder	Knutpunkten	Essenska villan	Skolmuseet	Stortorget	Stadsteatern	Utfart
Knutpunkten	0	200	0	1000	0	0
Essenska villan	800	0	0	0	0	0
Skolmuseet	0	800	0	0	400	200
Stortorget	0	300	600	0	400	0
Stadsteatern	0	0	0	0	0	1000
Utfart	0	0	800	0	0	0