Mittuniversitetet

MID SWEDEN UNIVERSITY

DT063G, Designmönster med C++

Skriftlig Tentamen

Innehållsförteckning

Tentamensdata

Ämne, nivå	Datateknik GR (B)
Kurs	DT063G, Designmönster med C++, 7.5hp
Tidpunkt	Torsdag 18 mars 2021, 09:00 – 15:00

Betygsgränser (poängdistribution)

A	**28-30**	
B	**25-27**	
C	**22-24**	
D	**19-21**	
E	**16-18**	
F(x)	**15**	*Underkänd, kan kompletteras!*
F	**0-14**	*Underkänd!*

Tillåtna Hjälpmedel

Kursmaterial + utförda laborationer!

Studentdata

Namn	*Marcus Roos*
StudentID	*Maro1904*
Personnummer	*940219-8130*

Tentamensfrågor

Uppgift 1 (2p)

Nedan finner du två implementationer av **Singleton DP**. Beskriv vad som skiljer dem åt, samt när och varför en bör föredras över den andra. ***Dina svar måste motiveras!***

```
class Singleton {
  ... // Attributes
  Singleton() = default;

public:
 Singleton(const Singleton&) = delete;
 void operator=(const Singleton&) = delete;

  static Singleton& getInstance() {
    static Singleton instance;
    return instance;
  }
  ... // Operations
};
```

```
class Singleton {
  ... // Attributes
  static Singleton* instance;
  Singleton() = default;

 public:
  Singleton(const Singleton&) = delete;
  void operator=(const Singleton&) = delete;

  static Singleton* getInstance() {
    if (instance == nullptr)
      instance = new Singleton();
    return instance;
  }
  ... // Operations
};
```

Svar

Den högra implementationen föredras generellt och ses som en standard singleton implementationen.

Den vänstra implementationen kommer alltid att returnera en statisk instans varje gång getInstance kallas medan den högra kollar om instance är samma som nullptr, om så är fallet så skapas en ny singleton som returneras. Om en instans redan finns skapad så kommer den nuvarande instansen att returneras.

Skillnaden mellan de två implementationerna är att den vänstra implementationen inte får eller kan deallokeras under programmets körtid, medan den högra tillåter deallokering av instansen när den inte används. Den vänstra kommer användas när man förväntar sig att ha en singleton som allokeras ibörjan av programmet och innan själva programmet exekveras, den högra används när singleton instansen kan behöva deallokeras under programmets gång och inte behöver vara tillgänglig genom hela programmets livstid.

Uppgift 2 (6p)

Ange vilket designmönster du skulle tillämpa för varje scenario beskrivet nedan. Varje punkt ger totalt `1` poäng, där korrekt mönster ger `0.5` och en korrekt motivering `0.5` poäng.

a) Byta ut / alternera beteenden under exekvering beroende på aktuella omständigheter.

b) Reducera komplexiteten av ett existerande gränssnitt.

c) Utöka specifikationen av en existerande klass för att stödja ny funktionalitet.

d) Att utifrån en basimplementation av en algoritm låta detaljer kring exekveringsordning variera beroende på situation.

e) Utöka funktionalitet av objekt under exekvering.

f) Du önskar ersätta komplexa villkorssatser med en mer förenklad och skalbar lösning.

Svar

a) Strategy. Strategy används när olika regler används beroende på aktuella omständigheter. Strategy består av delarna Strategy, ConcreteStrategy samt context, där Strategy är gränssnittet, ConcreteStrategy är de konkreta implementationerna (kan ha hur många konkrete implementationer som helst), medan Context avgör vilken omständighet vi för närvarande finner oss i. Beroende på dessa omständigheter så kommer Context att skicka sin förfrågan till Strategy som sedan vidarebefordrar förfrågan till rätt konkret implementation, beroende på omständigheten som lösningen behöver fungera för.

b) Facade. Facade används för att skapa ett gränssnitt som använder några av de mer komplexa metoderna i andra komplexa gränssnitt, då vi inte vill använda allting som finns i nuvarande gränssnitt så väljer vi istället att bygga vidare ett extern gränssnitt som inte behöver ta hänsyn till de mer komplexa fuktionerna. För att det här ska funka så kaplsas implementationerna in i det nya gränssnittet, facade, som sedan delas upp i subsystem.

c) Decorator. I en tidigare laboration vi hade så byggde vi en kaffemaskin, denna kaffemaskin skapade en grundkomponent som i sin tur skapade en dryck som kunden kunde välja, varesig det var te, kaffe, espresso etc. Sedan skulle ytterligare funktionalitet läggas till denna klass, vi la då till tillägg för drycken, socker, mjölk, grädde etc. Med hjälp av denna struktur som vi redan hade satt upp så hade det varit enkelt att lägga till ännu mer funktionalitet till dryckesmaskinen. Vi kanske skulle vilja lägga till tillbehör så som kanelbulle, weinerbröd, gifflar etc. Decorator används när en existerande klass redan finns men fler implementationer måste läggas till, och det är dynamiskt så det kan konstant läggas till ny funktionalitet, det används flitigt inom spelvärlden där man konstant bygger på innehållet som finns.

d) Template Method (möjligen Strategy), Template Method låter en implementera en basimplementation och sedan skicka vidare vissa steg av implementationen till subklasser, dessa subklasser kan kallas vid behov, alltså när exekveringsordningen är beroende på situationen. I en template method så är det basimplementationen som kallar på subklasserna när de behövs, template method har en så kallad hook-implementation i sina subklasser som definieras om de behövs. Jag skulle personligen ha använt mig av strategy pattern här igen då strategy tillåter en att implementera olika beteende beroende på situationen som man för närvarande finner sig i, som jag beskrev under punkt ”a)” är strategy perfekt att använda när klasserna är relaterade men utförandet är olika.

e) Decorator igen, decorator ger en möjligheten att utöka funktionalitet av objekt under exekvering, precis som det beskrevs under punkt ”c)” så kan man i detta scenario första nöja sig med att endast skapa en kaffe, men om man vill utöka objektets funktionalitet så kan man lägga till mjölk och socker i sitt kaffe, samt beställa en kanelbulle vid sidan om.

f) Facade eller Command fungerar. Jag skulle valt Command då vi även vill ersätta villkorssatser. När vi använder DP Command så har vi en receiver och invoker som först kollar vart i programmets exekveringsprocess vi är, och med hjälp av det villkoret skickar den vidare till Invoker som kallar på rätt exekvering av programmet. Om vi vill lägga till ny funktionalitet till programmet så behöver inga existerande klasser ändras utan det räcker att lägga till nytt.

Uppgift 3 (5p)

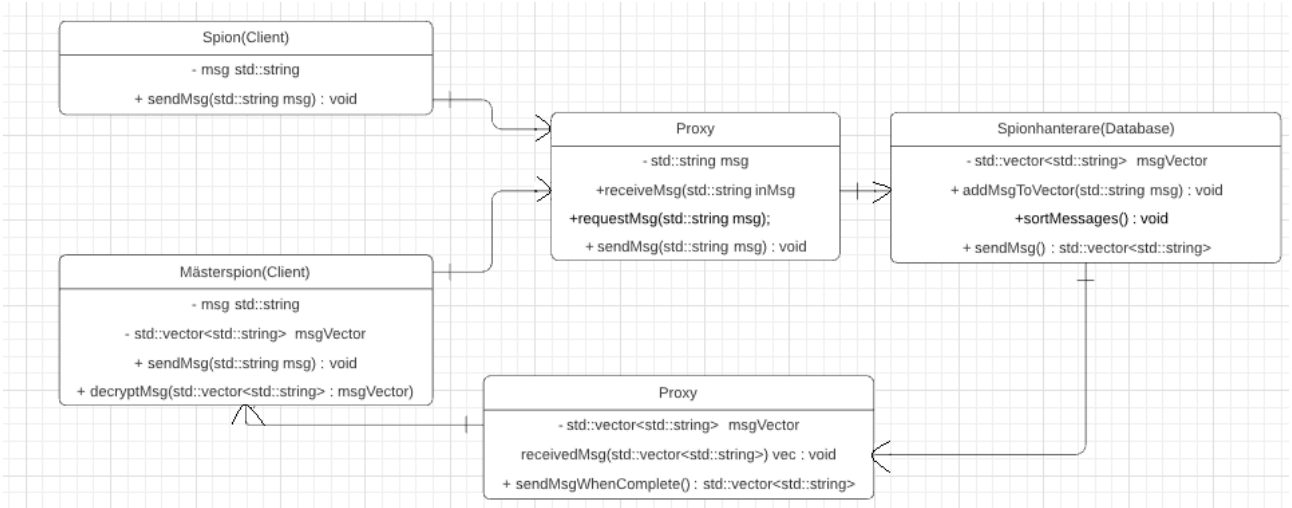
Du har blivit kontaktad av den topphemliga organisationen **Spies'R Us** som behöver din hjälp att begränsa hur information delas mellan agenter verksamma i fält. Det finns två typer av agenter; **spion** och **Mästarspion**. Varje spion tillhandahåller en liten del av ett större meddelande, men som är krypterat för att försäkra att ingen av dem förstår innehållet. Mästarspionen är den ende som har nyckel för att dekryptera meddelandet, och en spionhanterare på **Spies'R Us** är den enda som vet i vilken ordning informationens delar skall sättas samman för att meddelandet skall bli komplett.

Din uppgift blir att tillhandahålla en design som uppfyller ovan beskrivna krav på beteende och ansvarsfördelning, och det skall inte spela någon roll hur omfattande ett meddelande kan vara. Hur möter du detta problem och på vilket sätt skulle en lämplig design se ut? **Förklara** och **motivera** ditt tillvägagångssätt och det / de mönster du tillämpar. Bifoga även ett klassdiagram och / eller klassdefinitioner (i kod) som illustrerar din lösning.

Svar

Eftersom spionerna är plural och ska rapportera till spionhanteraren med sina meddelanden så skapar det ett många-till-ett scenario, dessa meddelanden ska alla till spionhanteraren, som sedan sätter ihop meddelandena i rätt ordning och vidarebefordrar det till mästerspionen som sedan använder sin nyckel för att dekryptera meddelandet. Däremot så kan mästerspionen också ha meddelande som ska sättas ihop så att skicka alla meddelande från spion till spionhanteraren, som sedan sätter ihop meddelandet och vidarebefordrar meddelandet till mästerspionen är inget alternativ.

Jag skulle använt DP Proxy i det här scenariot, på det sättet kan båda spionerna skicka sina meddelanden till proxyn, som behandlar det inkommande meddelandet, när meddelandet är mottaget så skickas det vidare till spionhanteraren som sätter samman meddelandet i en vector i rätt ordning, när meddelandet är ihopsatt i rätt ordning så skickas det tillbaka till mästerspionen via en annan proxy, dekrypteras och meddelandet är klart. I det här fallet så spelar det ingen roll hur omfattande meddelandet är då det krypterade meddelandet lagras i ordning hos spionhanteraren, som i sin tur skickar vidare det rätt ihopsatta meddelandet via en proxy, till mästerspionen som sedan dekrypterar det.



Uppgift 4 (2p)

Avsluta nedanstående mening med någon av alternativen, samt en kort motivering för detta val. Korrekt val av alternativ ger **1** poäng och tydlig motivering **1** poäng.

Template Method DP liknar **Factory Method DP** eftersom...

a) … de båda definierar abstrakta operationer som ska implementeras i subklasser.

b) … de båda är inblandade i instansiering av objekt.

c) … de båda delegerar ansvar för instansiering av objekt.

Svar

Andra punkten, b) är korrekt. Varken a eller c passar in då endast Factory Method delegerar ansvaret för instansiering av objekt till sina subklasser, något som template method inte gör. Template method är inte inblandad i instansiering av objekt på samma sätt som factory method är. Medan båda mönstrena definierar abstrakta operationer i basklassen som sedan implementeras i subklasserna, men i template methods fall så behöver inte alla algoritmer implementeras utan det räcker om några implementeras. Det gör att både a (operationerna *ska* inte implementeras), och c (Template method delegerar inte ansvaret för instansiering av objekt, men det gör factory method) blir inkorrekta. Rätt svar är b, template method instansiera sitt objekt i den abstrakta klassen medan factory method instansierar sina objekt i subklasserna.

Uppgift 5 (3p)

Studera klassen `MyComplex` och hur den är implementerad.

```
class MyComplex {
public:
  static MyComplex fromCartesian(double real, double imag) {
    return MyComplex(real, imag);
  }

  static MyComplex fromPolar(double length, double angle) {
    return MyComplex(length*cos(angle), length*sin(angle));
  }

private:
  MyComplex(double a, double b) :x(a), y(b) {}
  double x, y;
};
```

Denna klass är exempel på en variant av ett vanligt designmönster. Vilket av nedanstående alternativ handlar det mest troligt om? Du får enbart välja ett alternativ, och ditt svar måste motiveras!

a) Singleton DP

b) Abstract Factory DP

c) Strategy DP

d) Factory Method DP X

Svar

Det är en basklass vi kollar på som inte är abstrakt eller virtuell, implementationerna görs alltså i basklassen. Abstract Factory använder sig av en ett abstrakt gränssnitt, så abstract factory kan det inte vara. Objektet instansieras olika beroende på vilka värden som skickas med i konstruktorn, något som factory method gör då den låter sub klasserna instansiera objekten. En singleton är det inte då getInstance saknas och flera instanser av objektet kan existera samtidigt, trots att static returneras så är själva objektet inte statiskt utan endast själva funktionerna. Strategy kan det också vara då strategy behandlar olika algoritmer, men då det är objekt som returneras och implementationerna är gjorda i basklassen bör det inte vara strategy heller.

Jag tror det är Factory Method som beskrivs i koden.

Uppgift 6 (2p)

Ange vilka två av nedanstående designmönster som tillämpar rekursiv komposition. Notera att du enbart får välja två alternativ, och varje korrekt svar ger **1** poäng. Ingen svarsmotivering är nödvändig!

a) Strategy DP

b) Composite DP

c) Proxy DP

d) Decorator DP

e) Abstract Factory DP

Svar

Composite DP & Decorator DP.

Uppgift 7 (10p)

Ange huruvida nedanstående påståenden är korrekt eller ej. Varje rätt svar ger +1p, varje felaktigt svar ger -1p medan varje uteblivet svar ger 0p (minsta möjliga poäng på uppgiften är 0p)!

	Påstående	Svar
a)	Begreppet ***sammanhållning*** relaterar enbart till Objekt Orienterad Programmering.	FALSKT
b)	En av de främsta anledningarna att föredra komposition före arv är för att undvika ***kombinatorisk explosion*** av antalet nödvändiga klasser.	SANT
c)	Utifrån ett högre designperspektiv så ökar sammanhållningen (cohesion) av en modul om dess ingående entiteter har stark koppling (tight coupling) till varandra.	FALSKT
d)	Principen ***find what varies and encapsulate it*** kan bli uppfyllt genom att definiera likheter mellan subklasser i en gemensam abstrakt bas.	SANT
e)	Strategy DP kombineras ofta tillsammans med andra mönster för att abstrahera implementationsdetaljer och möjliggöra flexibla designlösningar.	FALSKT
f)	Composite DP använder både arv och komposition mellan klasser.	FALSKT
g)	En klassadapter konstrueras i C++ genom multipelt arv från minst två icke-publika basklasser.	FALSKT
h)	Utökning av det publika gränssnittet i deriverade klasser strider mot **Liskov's Substitution Principle** (LSP).	SANT
i)	**Single Responsibility Principle** (SRP) kan summeras såsom: *det bör finnas enbart en anledning till förändring*.	SANT
j)	Enligt **Interface Segregation Principle** (ISP) så ska en design tillämpa generiska gränssnitt som täcker in ett stort spektrum av möjliga användningsområden.	FALSKT

Redovisning

Skriv dina svar under avsedda sektioner direkt i tentadokumentet. Vid eventuella problem där detta inte är möjligt, är det accepterat att svaren skrivs i separata textdokument. Lägg då till ditt namn och studentid i varje dokument och relatera tydligt varje svar till respektive fråga / uppgift.

Kodexempel / implementationer kan förläggas till källfiler och eventuella diagram kan sparas i externa bildfiler. Dock måste relationen mellan dina svar och dessa filer tydligt framkomma (inget är underförstått gällande dessa delar).

Alla filer packas till en **zip**-fil som skickas in i avsedd inlämningslåda i Moodle senast angiven deadline. Lycka till!

Genom att du skickar in denna tentamen försäkrar du att alla svar är dina egna, att dina lösningar inte innehåller plagiat från andra källor samt att samarbete med andra studenter inte har förekommit under provtiden!