Instuderingsuppgifter läsvecka 3 - LÖSNINGAR

1.

Funktionell dekomposition innebär att bryta ned koden i ändamålsenliga metoder. Koden blir lättare att läsa – ett beskrivande namn på metoden är lättare att förstå än en sekvens av kodrader. Ökar abstraktionsnivån – programmerarna kan betrakta metoder som inbyggda högnivå konstruktioner i programspråket. Reducerar duplicering av kod – kodavsnitt som upprepas på flera ställen kan ersättas med ett metodanrop.

2.

The Separation of Concerns Principle (SCP) säger att en metod skall utföra endast en sak och utföra denna sak bra. Följs denna princip erhålls korta metoder som är lätta att överblicka och förstå. Den kongnitiva komplexiteten reduceras. SCP anger slutmålet vid funktionell dekomposition.

3.

En sidoeffekt är en observerbar modifikation av data till följd av ett metodanrop. En oönskad sidoeffet är en oavisiklig sidoeffekt, dvs en sidoeffekt som inte anges i metodens specifikation och är således alltid en bug.

4.

T*he Command-Query Separation Principle* säger att en metod antingen skall vara enaccessor eller en mutator, inte både och. En metod som returnerar information om ett objekts tillstånd skall inte ändra tillståndet hos objektet. En metod som ändrar tillståndet hos ett objekt skall inte returnera information om objektet.

5.

Kontraktsbaserad design innebär att det upprättas ett formellt kontrakt mellan ett objekt/klass och klienten som använder objektet/klassen. Ett kontrakt anger för varje publik metod vad metoden gör samt vilka för- och eftervillkor metoden har. Klienten ansvarar för att förvillkoren är uppfyllda då en metod anropas. Klienten måste alltså ha möjlighet att testa om ett förvillkor är uppfyllt eller inte. Är förvillkoren ansvarar metoden för att eftervillkoren uppfylls. Kontraktsbaserad design medför en tydlig och dokumenterad ansvarsfördelning, förenklar designen, samt underlättar debugging och testning.

6.

Defensiv programmering förespråkar att man lägger in kontroller i varje programmodul för att upptäcka och hantera oförutsedda situationer. Detta leder till redundanta kontroller (t.ex. kan både klient och server utföra samma kontroller). Stora mängder kontroller gör att programmet bli mer komplext, svårare att underhålla och långsammare.

Vid design by contract är ansvarsfördelningen tydlig och är en del av gränssnittet för modulen, vilket förebygger redundanta kontroller och förenklar underhåll.

7.

Innan man lämnar varje konstruktor och innan man lämnar varje publik mutator-metod.

8.

Nej! Det är metoden (servern) som ansvarar för eftervillkor och klassinvarianter varför dessa kan uttryckas med hjälp av den interna implementationen eller det publika gränssnittet.

9.

Assertions är till stor hjälp i felsökningsprocessen för att kunna rätta fel som beror på att för- och eftervillkor samt klassinvarianter inte har uppfyllts.

10.

a) Designen bryter mot Open/Close-principen. Om man skulle införa ytterligare en klass måste man ändra i metoden chooseSupplier.

b) Inför ett gränssnitt som klasserna SupplierA, SupplierB och SupplierC implementerar.

```
public interface Supplier {
   public void doIt();
}

public class SupplierA implements Supplier {
   public void doIt() {
      System.out.println("Done by supplier A");
   }
}

public class SupplierB implements Supplier {
   public void doIt() {
      System.out.println("Done by supplier B");
   }
}

public class SupplierC implements Supplier {
   public void doIt() {
      System.out.println("Done by supplier C");
   }
}

public static void chooseSupplier(Supplier obj) {
   obj.doIt();
}
```

Alternativt kan man införa en abstrakt klass som klasserna SupplierA, SupplierB och SupplierC utökar.

11.

Förvillkoret amount > 250 i klassen CreditCardPayment är starkare än i dess superklass Payment.

12.

Specifikation a) är undermålig och särklassigt sämst. Värdet 0 som returneras om fältet är **null** eller är tomt kan ju även vara ett giltigt värde för ett fält som innehåller element.

Specifikationerna b) och c) är båda fullvärdiga och vilken som är att föredra beror på om man utgår från klientens eller serverns perspektiv. Från klientens perspektiv är specifikation b) att föredra.

13.

a) Exponering av den interna representationen innebär att den interna representationen oavsiktligt blir direkt tillgänglig för modifieringar utanför klassen, t.ex. att en instansvariabel borde vara privat blivit publik.

b) Här uppstår exponeringen genom att referensen till instansvariablerna startPoint och endPoint är tillgänglig utanför klassen. Vi måste därför se till att konstruktorn skapar kopior av referenserna som fås som parametrar, samt att metoden getStartpoint och getEndpoint istället för att returnera referenser till instansvariablerna returnerar referenser till kopior av instansvariablerna.

```
import java.awt.Point;
public class Line {
 private Point startPoint;
 private Point endPoint;
 public Line(Point startPoint, Point endPoint) {
    this.startPoint = (Point) startPoint.clone();
    this.endPoint= (Point) endPoint.clone();
 }
 public Point getStartpoint() {
    return (Point) startPoint.clone();
 }
 public Point getEndpoint() {
    return (Point) endPoint.clone();
 }
}//Line
```

14.

Icke-muterbara objekt har många fördelar:

- De är lätta att förstå. Ingen risk att de hamnar i inkonsistenta tillstånd. Det är mycket enklare att resonera om och bevisa egenskaper för kod som använder icke-muterbara objekt.
- Inga oväntade alias-updateringar. Det är ofarligt att lämna ut referenser till dem. Det gör inget om en massa olika objekt råkar använda ett och samma icke-muterbara objekt.
- Högre maintainability (lättare att underhålla) – alias-buggar är bland de allra svåraste att spåra.
- Deras interna data kan återanvändas. Om ett nytt icke-muterbart objekt ska skapas där vissa fält inte skiljer sig från de i ett befintligt objekt av samma klass kan alla oförändrade fält referera till exakt samma instanser som i det befintliga objektet.
- Det är lätt att använda icke-muterbara objekt som tillstånd i andra objekt.
  - De är alltid trådsäkra.
- Det är ofarligt att skicka dem till andra processer i distribuerade system.

15.

En klass som anges med **final** får/kan inte utöka, dvs klassen kan inte ha några supklasser.

En metod som anges med **final** får/kan inte överskuggas i en subklass.

En parameter som anges med **final** får/kan inte ädra sitt värde i metoden.

En instansvariabel som anges med **final** får/kan inte ändra sitt värde.

16.

En klass är icke-muterbar om:

- alla attribut (instansvariabler) är **final private**
- klassen inte innehåller några mutator-metoder
- konstruktorn skapar klonade kopiaor av muterbara argument
- access-metoder returnerar klonade kopior av muterbara attribut
- alla instansmetoder är **final.**

För att förhindra att klassen kan ärvas sätts klassen till **final**. Detta måste göras om det expicit specificeras att klassen är icke-muterbar, annars kan skulle en subklass kunna vara muterbar och därmed strida mot Liskov Substitution Principle.

17.

Principen säger att en klass endast skall ha ett ansvarsområde. Implicit innebär detta att klassen har hög kohesion – samtliga komponenter i klassen samverkar för ett gemensamt mål. Om en klass har flera ansvarsområden blir automatisk kohesionen lägre, vissa komponenter sköter vissa ansvarsområden och andra komponenter andra ansvarsområden. En klass med flera ansvarsområden blir sårbar och bräcklig, det kan uppstå fler anledningar till att klassen behöver förändras. En klass med flera ansvarsområden får ochså automatisk högre koppling, flera andra klasser blir beroende av klassen. Om klassen behöver ändras berörs de klasser som har beroenden till klassen.

A class should have only one reson to change. (Martin)

18.

Att en gränssnitt för en klass är konsistent innebär att gränssnitten för publika metoder har samma strukturella uppbyggnad avseende namn, parametrar, ordning på parametrar, returvärden och beeteende.

19.

Information Expert Pattern säger att den objekt (den klass) som har den information som behövs för att utföra en uppgift skall utföra uppgiften. Handlar också i gruden om kohesion.

20.

Law of Demeter (LoD) säger att en metod m i klassen C endast skall samverka med objekt som

- är instansvariabler i C
- är argument till m
- skapas av m
- med sig själv (**this**)

Dessa är de objekt som finns i metodens scope (= explicit kända i metoden), dvs dessa objekt är metodens ”immediate friends”.

LoD innebär är att en metod inte skall anropa någon av sina ”vänner” för att få tillgång till ett annat objekt, med vilket metoden kan samverka. Behövs en sådan samverkan är det ”vännen” som skall förmedla detta. Genom att använda LoD *minskar graden av kopplingar* i programsystemet.

21.

The Interface Segregation Principle (ISP) säger att ett objekt/klass inte skall vara beroende av metoder som objekt/klassen inte använder. Säg t.ex. att klassen Service innehåller de publika metoderna m1, m2, m3 och m4, samt att klassern A och B båda nyttjar tjänster från klassen Service. Dock nyttjar klassen A endast m1 och m2, medan klassen B endast nyttjar m3 och m4. Man skall då införa två interface IA och IB som specifierar metoderna m1 och m2 respektive m3 och m4, samt låta klassen Service implementera implementera interfacen IA och IB. Detta medför att klassen A kan betrakta ett objekt av typen Servic som ett objekt av typen IA medan klassen B betrakta ett objekt av typen Service som ett objekt av typen IB. Varken klassen A eller B blir därigenom beroende av metoder de inte använder.

Något mer kortfattat säger (ISP) att feta interface (= interface som specificerar många metoder) skall delas upp i flera tunna interface (= interface som specificerar få metoder).