Lösningsförslag till instuderingsuppgifter läsvecka 1

1.

Vid *Programming in the small* är programmen vanligtvis mycket enkla, programmen har kort livslängd och endast en eller fåtal programmerare är delaktiga i utvecklingsarbetet

Vid *Programming in the large* är programmen vanligtvis mycket stora, programmen har lång livslängd och 100-tal programmerare kan vara delaktiga i utvecklingsarbete och underhåll.

2.

Att koden är *korrekt* innebär att programmet fullgör det som anges i specifikationen.

Att koden är *robust* innebär att programmet inte skall krasch om det uppstår situationer som inte finns inkluderade i specifikationen. Detta betyder att programmet måste kunna förutse att fel kan inträffa och reagera på ett lämpligt sätt då de inträffar.

Koden vi producerar skall alltid vara korrekt kod och vi skall alltid försöka göra koden så robust som det är möjligt.

3.

Viktigaste aspekterna för utvecklarna är att koden går att underhålla (vilket inkludera att designen skall vara enkel att förstå samt vara lätt att modifiera och utöka med ny funktionalitet), går att återanvända och är enkel att testa..

En förutsättning för att kunna underhålla koden är att koden är lätt att läsa och förstå. Ett system som har en modulär uppbyggnad där varje modul har ett väl specificerat ansvarsområde, tydliga gränssnitt samt hög kohesion och låg koppling är mer överblickbart och lättare att bygga ut än ett system där modulerna har låg kohesion och hög koppling.

4.

Stelhet (*rigidity*) betyder att en förändring i en programmodul är svår att göra p.g.a. att det finns många beroenden med andra programmoduler vilka påverkas av förändringen.

Bräcklighet (*fragility*) innebär att det är svårt att göra att förändring i en programmodul utan att introducera nya oväntade fel i andra programmoduler som inte har någon konceptuell koppling till den programmodul som förändras.

5.

En design blir unken (= ruttnar) p.g.a. att *olämpliga beroenden* introduceras mellan de ingående i systemet. Detta orsakas ofta av att akuta problem löses med snabba hack istället för att åtgärda eventuella problem i designen. Ett programsystem kommer under sin livstid att förändras, vilket betyder att det även kan finnas behov av att förändra den underliggande designen.

6.

Synliga tecken i koden på att en design har eller håller på att ruttna är till exempel:

- duplicerad kod
- stora klasser
- långa metoder
- långa parameterlistor
- långa if- och switch-satser
- obegriplig kod
- . . .

7.

För att motverka att en design ruttnar måste man från början inse att kraven kommer att förändras, och använda beprövade tekniker och principer för att åstadkomma en så framåtkompatibel design som möjligt.

Det är dock omöjligt att från början designa ett perfekt system. För att hålla en hög kvalitén på designen under systemets hela livstid måste därför koden refaktoriseras vid behov (=ständigt?).

8.

Refaktorering innebär en omskrivning av koden i syfte att förbättra strukturen, utan att förändra kodens funktionalitet, i syfte att göra koden lättare att förstå, lättare att underhålla och bygga ut samt att återanvända. Refaktorering sker i små steg , och testas efter varje steg, för att säkerställa att inga fel introducerar och att funktionaliteten upprätthålles.

9.

När man utvecklar ett programsystem tillsammans behöver man ibland läsa och förstå varandras kod. Om alla följer en kodstandard blir detta mycket enklare och ingen behöver reta sig på någon annans kodningsstil (vilket annars är vanligt). Naturligtvis är det också minst lika viktigt för de programmerare som senare skall underhålla programsystemet att de endast har en kodningsstil att förhålla sig till.

10.

Ett namn skall väljas på så sätt att namnet underlättar för den som läser kod att förstå vad koden gör. Namnet skall väljas med stor omsorg och skall avspegla vilken roll en entitet har och vara (tillräckligt men inte överdrivet) självförklarande.

Namngivningen skall vara på engelska (eftersom vi i kursen behandlar *programming in the large*).

11.

En modulär design innebär att systemet är uppbyggt av moduler. Varje modul har en väldefinierade funktionalitet, ett väldefinierat gränssnitt, hög kohesion samt liten koppling till andra moduler.

Fördelar med en välgjord modulär design:

uppdelning av ansvar
lätt att utvidga systemet
återanvändbarhet p.g.a. hög kohesion och låg koppling
utbytbarhet pga. hög kohesion och låg koppling.

12.

Kohesion (*cohesion*) är ett mått på den inre sammanhållningen i en modul. En modul skall ha hög kohesion - vilket innebär att samtliga komponenterna i en modulen sinsemellan samverkar för att lösa den uppgift modulen har, utan att samverka med andra moduler.

K*oppling* (*coupling*) är ett mått på hur beroende en modul är av andra moduler. En modul skall ha låg grad av koppling – vilket betyder att modulen har hög kohension och därmed är återanvändbar och utbytbar.

Ju lägre grad av koppling som finns mellan modulerna i ett system, ju mer flexebelt och förändringsbart blir systemet.

13.

För att en modul skall vara återanvändvar krävs att modulen har hög kohesion och låg koppling.

14.

Vid implementationsarv ärver en klass implementationen (koden) från en annan klass. Vid specifikationsarv ärver en klass specifikationer från ett interface.

15.

Multipelt arv innebär att en klass ärver från mer än en klass eller ett interface. Java tillåter implementationsarv endast från en klass, varför multipelt arv i Java alltid involverar specifikationsarv från minst ett interface.

16.

Specialisering innebär att man från en superklass skapar subklasser – subklasserna utökar den uppsättning egenskaper och beteenden som superklassen har. Generalisering innebär att gemensamma egenskaper och beteenden hos flera klasser samlas i en gemensam superklass.

17.

Här är ett problem med multipelt arv:

```
public class Medic {
   private Date birthDate;
   private int examinationYear;
   public void operate() { /*...*/ }
}// Medic

public class Engineer {
   private Date birthDate;
   private int examinationYear;
   public void construct() { /*...*/ }
}// Engineer

public class MedicAndEngineer extends Medic, Engineer {
   . . .
}//MedicAndEngineer
```

En MedicAndEngineer är både Medic och Engineer, och klassen MedicAndEngineer ärver därför egenskaperna birthDate och examinationYear *två* gånger, från Medic respektive Engineer. Hur ska dessa kollisioner hanteras? Hur många födelsedatum och examensår ska en MedicAndEngineer ha?

För födelsedatum är det naturligt att slå ihop de två instansvariablerna till en enda variabel, för en person har bara ett födelsedatum, och detta är samma oavsett yrke. Men en person som är både läkare och ingenjör har (förmodligen) två olika examensår, så examensåren ska inte slås ihop till en enda variabel. Det är svårt för kompilatorn att förstå skillnaden.

Det kan även uppstå namnkonflikter på metoder. Två klasser som ärvs ifrån kan ha metoder med samma namn och profil, men metoderna gör helt olika saker.

18.

En abstrakt klass kan föredras framför ett interface om det finns beteenden som delas mellan alla subklasser och som man vet inte kommer att förändras.

Abstrakta klasser används för att faktorisera ut duplicerad kod i arvshierarkier eller då man vill tvinga subklasser att implementera en metod (t.ex. som i designmönstret Template method).

19.

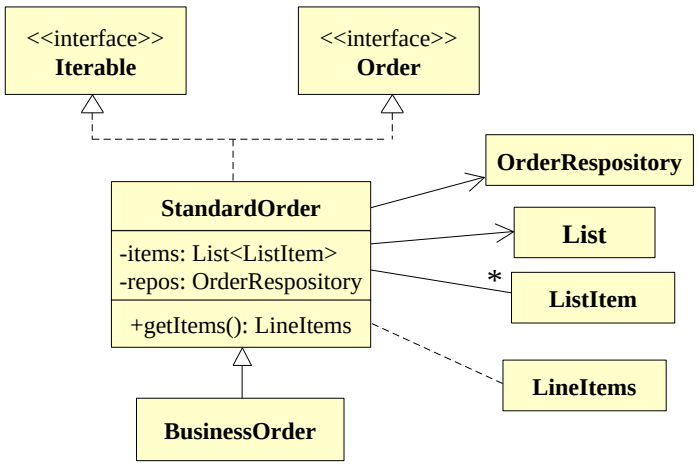
Fem relationer finns definierade: Generalisering, realisering, association, inkapsling samt beroende.

20.

Vid aggregation kan ett objekt vara associerat med flera ägarobjekt. Försvinner ägarobjektet/ägarobjekten kommer det associerade objektet att leva vidare. Aggregation beskriver ”känner till”-relationen.

Vid komposition kan en objekt endast vara associerat med ett ägarobjekt. Försvinner ägarobjektet, så försvinner också det associerade objekt. Komposition beskriver ”har”-relationen (eller ”del av”-relationen).

21.



22.

Alias innebär att det finns två eller flera referenser till samma objekt. Detta innebär att objektet är åtkomligt från flera ställen i koden. Om en förändring av objektets tillstånd görs via ett alias påverkas allså andra delar av koden, vilket kanske inte var avsikten. Alias är alltså en potentiell källa för buggar.

23.

Nya typer definieras genom att definiera nya klasser och nya interface.

24.

En typ är en värdemängd samt en uppsättning operationer som kan utföras på denna värdemängd.

25.

Om S är en subtyp till T, så är mängden av objekt i typen S en delmängd till mängden av objekt i typen T, och mängden av operationer i typen T är en delmängd av mängden operationer i typen S.

26.

Ett programmeringsspråk är starkt typat om kompilatorn kan avgöra typkompabilitet för alla uttryck som representerar värden. Starkt typade programspråk har således statisk typkontroll och kräver att typen anges för alla variabel och metoder.

27.

Explicit typomvandling innebär att programmeraren säger åt kompilatorn att åsidosätta typkontrollen för ett uttryck och själv anger vilken typ uttrycket har.

Faran med explicit typomvandling är att det visar sig under exekveringen att uttrycket inte har den typ som programmeraren angivit, varvid ett exekveringsfel uppkommer.

28.

Med operationen **instanceof** kan man kontrollera huruvida ett objekt är av en viss typ. Operationen är t.ex. användbar för att avgöra om en explicit typomvandling är korrekt eller inte.