Skolan för datavetenskap och kommunikation

Objektorienterad Programkonstruktion, DD1346

Tentamen 2013–06–01, kl. 10.00-13.00

Tillåtna hjälpmedel: Papper, penna och radergummi.

Notera: Frågorna i del I ska besvaras på för ändamålet lämnad plats i tentamenslydelsen. Frågorna i del II besvaras på separat papper. Använd gärna både fram- och baksida, men behandla högst en uppgift per sida. Kom ihåg att skriva namn och personnummer på alla inlämnade blad. Skriv tydligt!

Betygsgränser: Betyg XF: $\geq$ 17p i del I
Betyg E: $\geq$ 20p i del I
Betyg D: $\geq$ 20p i del I **och** $\geq$ 5p i del II
Betyg C: $\geq$ 20p i del I **och** $\geq$ 10p i del II
Betyg B: $\geq$ 20p i del I **och** $\geq$ 15p i del II
Betyg A: $\geq$ 20p i del I **och** $\geq$ 20p i del II

Ansvarig: Christian Smith (ccs@kth.se)

Lycka till!

Del I - flervalsfrågor

1. I denna uppgift finns 5 program(-delar) i form av Java-kod. *Generiska namn används i stället för de mer vanliga namn som avslöjar vilket mönster det är.* För varje program, ange det designmönster som bäst beskriver det. Välj från listan nedan. Varje korrekt angivet designmönster ger 1 p. (5 p)

   MVC **Singleton** **Adapter** **Proxy** **Composite** **Flyweight** **Threadpool**
   Lock **Observer** **Factory** **Builder** **Prototype** **Facade** **Socket**

   a) Prototype

```
public class A implements Cloneable{

  private static myA = new A();
  SomeClass mySC = new SomeClass();
  DataHoldingClass myDHC;

  private A(){
    mySC.DoSomeReallyDemandingStuff();
    myDHC = mySC.RunHeavyCalculation();
  }

  public static A getA(){
    return myA.clone();
  }

  public A clone() throws CloneNotSupportedException{
    A cloneA = super.clone();
    A.myDHC = myDHC.clone();
  }

}
```

   b) Lock

```
public class B{

  int a = 0;

  public synchronized void setA(int newA){
    a = newA;
  }

}
```

c) Singleton

```
public class C{

  private static C myC = null;

  private C(){
  }

  public static C getC(){
    if (myC == null){
      myC = new C();
    }
    return myC;
  }

}
```

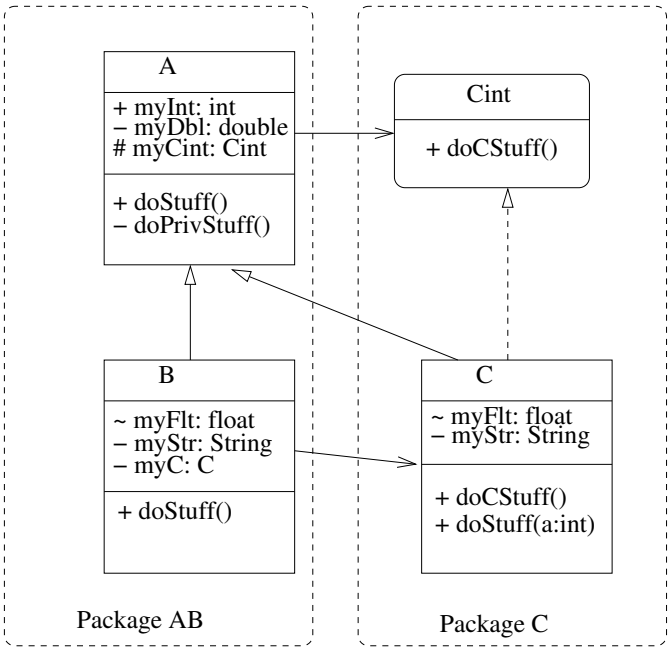
d) Flyweight

```
public class D{

  private static EnormousClass myEnormObject = new EnormousClass();
  private SomeSmallClass mySmallObject;

  public D(){
    mySmallObject = new SomeSmallClass();
  }
}
```

e) Proxy

```
public class E{

  private SomeOtherClass myObject = null;
  private String trivialString;

  public E(){
    trivialString = "Trivial";
  }

  public String getTrivialStuff(){
    return myTriv;
  }

  public NonTrivialClass getNonTrivialStuff(){
    if (myObject == null){
      myObject = SomeOtherClass.getObject();
    }
    return myObject.getNonTrivialStuff();
  }

}
```

2. Nedan finns ett UML-diagram som beskriver några klasser och deras relationer.



För objekt av typen B, ange vilka av följande fält- och metodanrop som är tillåtna. Antag att alla fält är initialiserade. 0.5 p för varje korrekt svar. (4 p)

a `myC.myFlt = 0.4;` Ej tillåten

b `myCint.doStuff(2);` Ej tillåten

c `myCint.doStuff();` Ej tillåten

d `myDbl = 2.1;` Ej tillåten

e `myC.doCStuff();` Tillåten

f `super.doPrivStuff();` Ej tillåten

g `myC.doStuff((int)myC.myDbl);` Ej tillåten

h `super.myInt = (int)myFlt;` Tillåten

3. Nedan följer 4 exempel på Java-program. För varje program, ange om programmet alltid ger samma utskrift, aldrig ger någon utskrift, eller ger kan ge olika beteende vid olika körningar. Varje korrekt analyserat program ger 1 p. (4 p).

   a) Samma

```
public class A extends Thread{

    public static void main(String[] args){
        A myA = new A();
        myA.start();
        System.out.println("Finished!");
    }

    public void run(){
        while(true);
    }

}
```

   b) Olika

```
public class B extends Thread{

    public static void main(String[] args){
        B myB = new B();
        myB.start();
        System.out.println("Finished!");
    }

    public void run(){
        System.out.println("Running!");
    }

}
```

c) Olika

```
public class C extends Thread{

    private static int c = 0;

    public static void main (String[] args){
        C myC1 = new C();
        C myC2 = new C();
        myC1.start();
        myC2.start();
        System.out.println(c);
    }

    public void run(){
        incrementC();
    }

    private synchronized void incrementC(){
        c++;
    }

}
```

d) Aldrig någon utskrift

```
public class D extends Thread{

    public static void main(String[] args){
        D myD = new D();
        myD.start();
        myD.join();
        System.out.println("Finished!");
    }

    public void run(){
        while(true);
    }

}
```

4. För varje påstående om fält och variabler i Java, ange om det är sant eller falskt. 5 korrekta svar ger 4p, 4 korrekta svar ger 2p, 3 korrekta svar ger 1p, 2 eller färre korrekta svar ger 0 p (4 p)

   a) Om man är osäker på vilken åtkomst man ska ge ett fält bör man välja `public` för att vara säker på att alla klasser (som kanske ännu inte implementerats) som behöver kan komma åt fältet.Falskt

   b) Ett fält som deklarerats som `static` får inte ändras efter att det skapats.Falskt

   c) En lokal variabel kan ges samma namn som ett fält som redan definierats i klassen.Sant

   d) Det är god programmeringssed att ge variabler namn med inledande versal.Falskt

   e) Om ett fält i ett visst objekt har deklarerats som `private` kan det inte läsas direkt av något annat objekt.Falskt

5. För varje påstående om metoder i Java, ange om det är sant eller falskt. 5 korrekta svar ger 4p, 4 korrekta svar ger 2p, 3 korrekta svar ger 1p, 2 eller färre korrekta svar ger 0 p (4 p)

   a) Två metoder i samma klass kan ha samma signatur förutsatt att de returnerar olika typer.Falskt

   b) Statiska metoder kan anropas även om man inte instansierat något objekt av klassen de definierats i.Sant

   c) Konstruktorer bör alltid synkroniseras för att undvika trådinterferens vid skapandet av ett nytt objekt.Falskt

   d) Om klassen **A** ärver från **B**, så kan inte **A** anropa metoder som deklarerats som `final` i **B**.Falskt

   e) En metod som deklarerats som `abstract` i en superklass måste ges en konkret implementation i alla ärvande icke-abstrakta klasser.Sant

6. För varje påstående om designmönster nedan, ange om det är sant eller falskt. 5 korrekta svar ger 4p, 4 korrekta svar ger 2p, 3 korrekta svar ger 1p, 2 eller färre korrekta svar ger 0 p (4 p)

   a) Det fämsta syftet med `Iterator` är att göra loopar snabbare.Falskt

   b) `Threadpool` används för att minska overhead i parallella program.Sant

   c) *Överskuggning* är ett problem som man ofta kan undvika med `Builder`.Falskt

   d) Ett vanligt mål med att använda `Prototype` är att göra programmet snabbare.Sant

e) I princip alla vanliga designmönster finns i Javas standardbibliotek i form av abstrakta klasser som man kan ärva ifrån.Falskt

Del II - fördjupningsfrågor

Följande uppgifter besvaras på separat papper.

7. förklara vad följande fel/problem innebär, hur de uppstår, och hur man kan lösa dem:

   a) Thread Interference (2 p) Thread Interference är när två trådar stör varandra, t.ex genom att ändra på samma variabel samtidigt. Problemet kan lösas med hjälp av lås, som endast medger åtkomst för en tråd i taget

   b) Deadlock (2 p) Deadlock uppstår när två trådar båda håller ett lås som den andra tråden behöver för att fortsätta. Kan lösas genom att en tråd släpper sitt första lås om den inte lyckas ta det andra, eller genom att låsa alla relaterade delar med samma lås.

   c) Starvation (2 p) Starvation är när trådar i programet tillbringar oproportionerlig stor tid med att vänta på någon otillräcklig resurs i stället för att göra nytta. Ett exempel kan vara att alla trådar väntar på samma lås, som bara tillåter en tråd att köra i taget trots att det kanske finns flera lediga CPU-kärnor. Detta kan ibland lösas genom att man refaktoriserar koden så att väntande trådar kan utföra andra uppgifter under tiden (kanske måste inte allt göras i en bestämd ordning?), eller så att stora kodblock som är låsta med samma lås delas upp i flera mindre med olika lås.

8. Vad menas med *lazy initialisation*? Ge exempel på när det kan vara bra respektive dåligt. (2 p) Lazy initialization innebär att man väntar med att initialisera objekt tills de behövs. Detta kan vara bra när man har ett stort antal objekt där de flesta kanske aldrig någonsin behövs. Det kan vara dåligt om objekten tar lång tid att initialisera, och den delen av koden där de behövs är tidskritisk.

9. Antag att du har fått i uppgift att konstruera en server för fotobearbetning. Tanken är att olika klienter ska kunna ansluta sig till servern över nätverk, och skicka en förfrågan som innehåller dels bilden som skall modifieras, och dels önskemål om vilka modifikationer som skall göras (t.ex ändra upplösning, köra olika grafikfilter, hitta ansikten i bilden, mm). I den första versionen av servern använder du dig av ett flertal olika bibliotek för bildbearbetning som du har laddat ner gratis från olika källor på internet. Beställaren tror att tjänsten har potential att bli väldigt populär så småningom.

   Klienterna kommer att kodas av någon annan, så du behöver inte inkludera dem i ditt program, men du måste förstås ange hur de ska kommunicera med servern.

   Beskriv en bra struktur för serverprogrammet. Förklara vilka designmönster som används, i grova drag hur de implementeras (vilken information finns var, hur kommunicerar programmets olika delar med varandra, hur styrs olika programflöden, osv). Motivera varför din lösning är bra! För full poäng skall hänsyn ha tagits till alla delar som beskrivs ovan. Du får använda UML eller (pseudo-)kod i ditt svar om du tycker att det förenklar presentationen, men det är inte ett krav. (8 p)

   Till att börja med bör servern kunna ta emot anslutningar i ett så öppet format som möjligt, så att vi inte tvingar på klienterna någon speciell utformning. T.ex kan klientanrop ske via TCP/IP i enlighet med en väldokumenterad standard. Man kan inleda med metadata i t.ex XML-format, för att avsluta med en binär kodning av själva bilden.

   Själva servern bör vara multitrådad för att effektivt kunna hantera flera anrop. Dessa trådar bör hanteras av en threadpool för att undvika att tung belastning genererar onödigt mycket overhead. Inkommande anrop ges som indata till en Factory-metod som returnerar Runnable-objekt som kan vara director gentemot ett Builder-objekt som utför alla önskade operationer på bilden, och sedan returnerar den färdiga bilden på önskat format. Färdiga bilder levereras till en separat kommunikationstråd som skickar tillbaka bilden enligt bestämt format.

   De olika nedladdade biblioteken kommer troligtvis att behöva förses med adapters för att kunna kommunicera med varandra, och vi kan sätta ihop vanliga anropskombinationer med en Facade för att underlätta kodandet.

10. En stor del av denna kurs handlar om designmönster, och varför man bör använda dem. Finns det några nackdelar med designmönster? Förklara! (2 p)

    Om man förlitar sig för mycket till befintliga lösningar riskerar man att tvinga på sina program en suboptimal form, om man vill göra något som inte passar in i kända mönster.

11. Vad är *lös koppling*? Är det generellt bra eller dåligt? Varför? (3 p)

    Med lös koppling menas att man undviker att explicit ange vilka typer som ska användas eller vilka metoder som ska anropas. Som exempel kan man ta Javas knappar, där man kan koppla knapplyssnare till knappar av den befintliga typen utan att behöva ändra någonting i den befintliga knappklassen. Att programmera mot gränssnitt i stället för mot klasser gör också att kopplingen blir lösare. Lös koppling är generellt bra, eftersom det gör det lättare att byta ut eller återanvända delar av koden utan att behöva ändra på många olika ställen.

12. Antag följande situation:

    En av dina vänner har försökt skriva en variant av tetris i Java, som tyvärr är förödande långsam och inte går att spela. Hen beklagar sig över att Java är ett så långsamt språk att det inte går att använda till något praktiskt. Du menar att Java förmodligen är tillräckligt bra för att att kunna köra tetris på en modern dator, och erbjuder dig att fixa programmet. Din vän tycker det är pinsamt och vill dock inte visa sin kod, men tar gärna emot generella tips för hur man optimerar kod. Skriv en enkel lathund för kodoptimering som din vän kan använda! (3 p)

    Det första och viktigaste steget bör vara att undersöka varför koden är långsam. Detta görs lämpligast med hjälp av profilering.

    När man vet vilka delar av koden som tar mest tid bör man koncentrera sig på att fixa dessa. Man kan sedan refaktorisera koden så att man byter ut långsamma algoritmer eller datastrukturer mot snabbare. Går det mycket tid till att skapa nya objekt bör man man försöka skapa dessa i förväg. Behöver man inte ändra i datastrukturerna i pågående spel kan man byta ut strukturer som är lätta att ändra i mot sådana som går fort att läsa.

    Finns det någon eller flera delar av programmet som tillbringar mycket tid med att vänta på andra delar, eller på svar från systemet eller från användaren, bör dessa läggas i separata trådar.