Skolan för datavetenskap och kommunikation

Objektorienterad Programkonstruktion, DD1346

Tentamen 2014–03–20, 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 FX: $\geq$ 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)

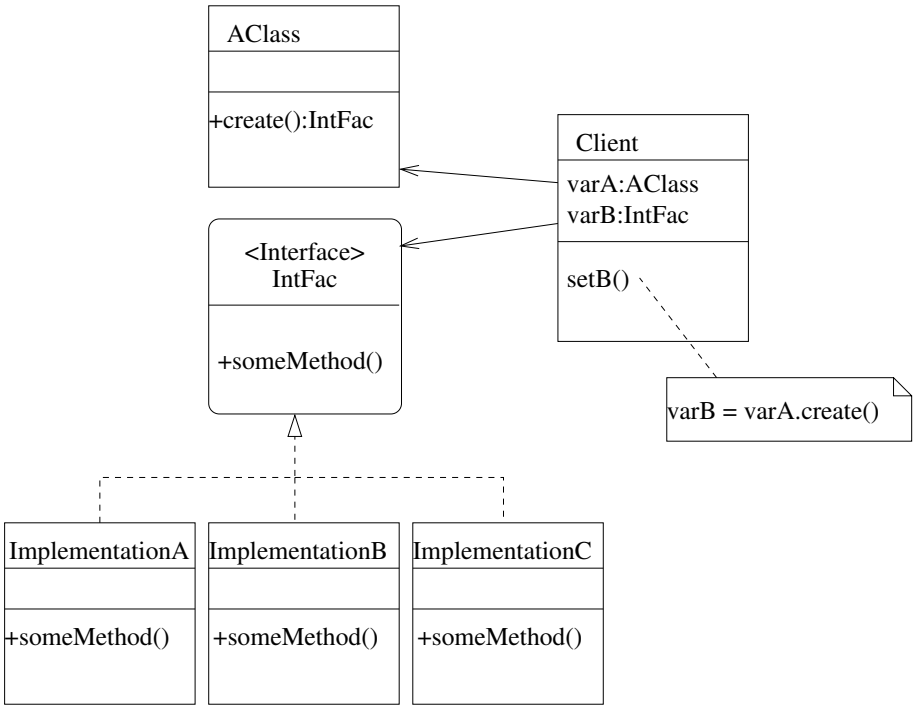
Lycka till!

Del I - flervalsfrågor

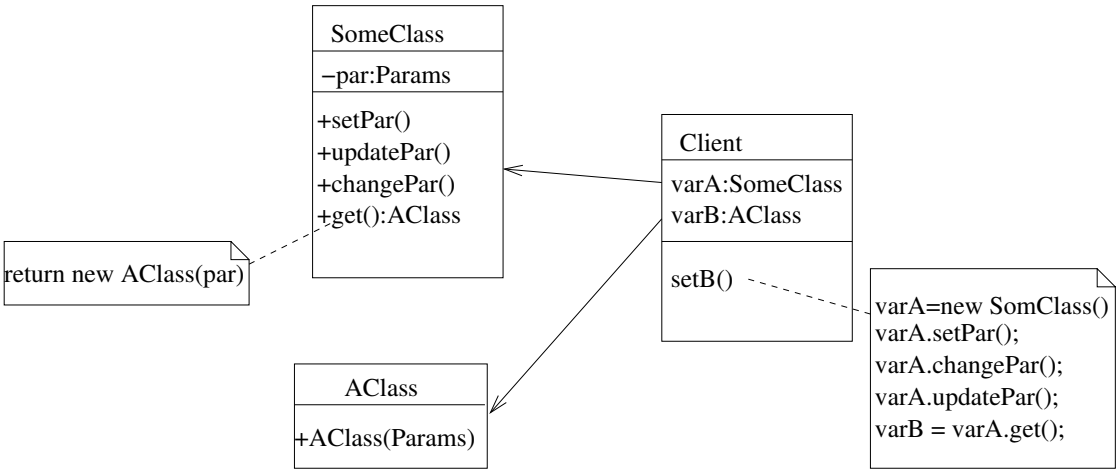
1. I denna uppgift finns 5 uppsättningar UML-diagram. *Generiska namn används i stället för de mer vanliga namn som avslöjar vilket mönster det är.* För varje diagram, ange det designmönster som bäst beskriver det. Välj från listan nedan. Varje korrekt angivet designmönster ger 1 p. (5 p)

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

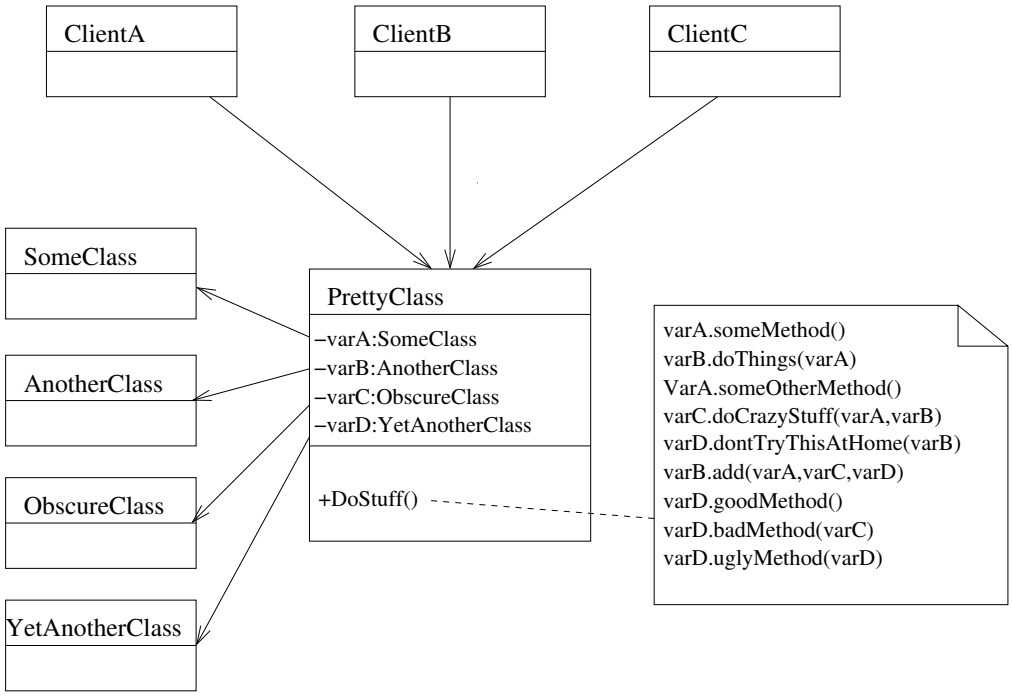
a) factory



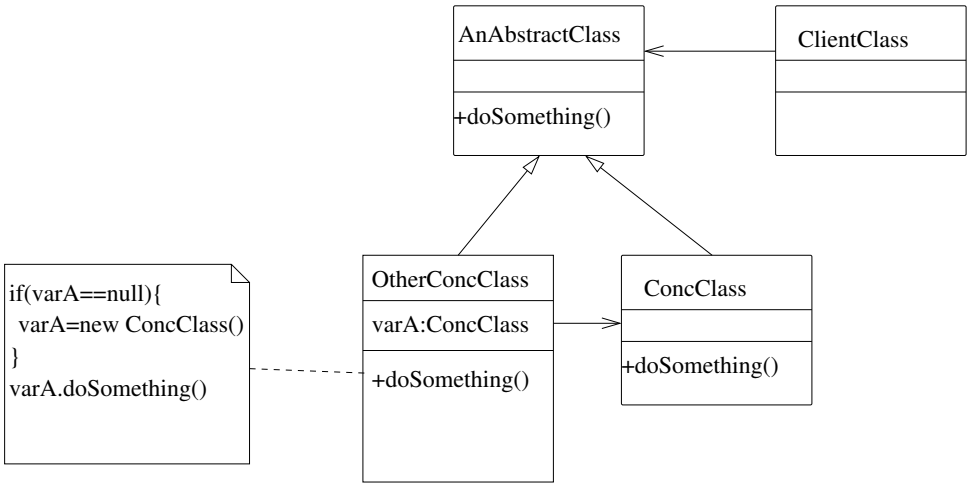
b) builder



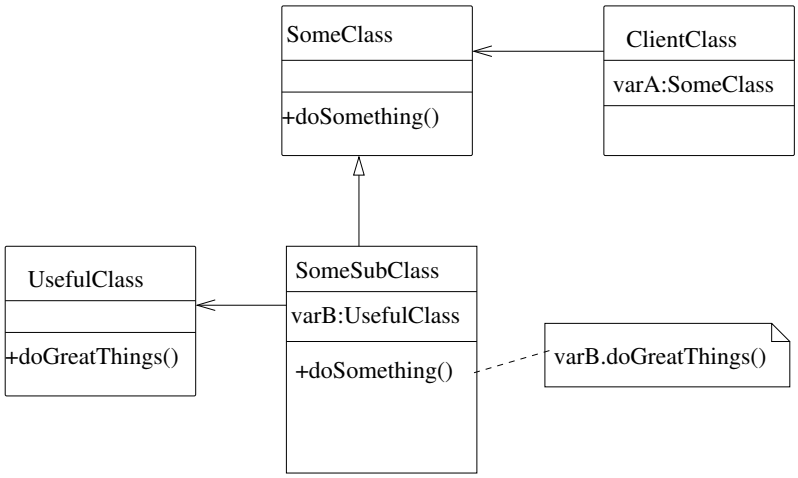
c) facade



d) proxy



e) adapter



2. Nedan följer 3 exempel på Java-program. För varje program, ange om programmet ger en korrekt implementation av en **singleton**. Varje korrekt analyserat program ger 1 p. (3 p).

a) korrekt

```
public class SingletonA{

    private static SingletonA theInstance = new SingletonA();

    private SingletonA(){}

    public static SingletonA getInstance(){
        return theInstance;
    }
}
```

b) EJ korrekt

```
public class SingletonB{

    private static SingletonB theInstance = null;

    private SingletonB(){
        if(theInstance == null){
            theInstance = new SingletonB();
        }
    }

    public static SingletonB getInstance(){
        return new SingletonB();
    }
}
```

c) korrekt

```
public class SingletonC{

    private static SingletonC theInstance = null;

    private SingletonC(){}

    public static SingletonC getInstance(){
        if(theInstance == null){
            theInstance = new SingletonC();
        }
        return theInstance;
    }
}
```

3. Nedan följer 3 exempel på Java-program. För varje program, ange om programmet ger fungerande **trådsäker variabelåtkomst**. Varje korrekt analyserat program ger 1 p. (3 p).

a) EJ korrekt

```
public class ThreadSafeA{

    private static volatile int a = 0;

    public void incA(){
        a++;
    }

    public void decA(){
        a--;
    }

    public int getA(){
      return a;
    }

}
```

b) EJ korrekt

```
public class ThreadSafeB{

    private static int a = 0;

    public synchronized void incA(){
        a++;
    }

    public synchronized void decA(){
        a--;
    }

    public synchronized int getA(){
        return a;
    }


}
```

c) `public class ThreadSafeC{` korrekt

```
    private static int a = 0;
    private static Object lock = new Object();

    public void incA(){
        synchronized(lock){
            a++;
        }
    }

    public void decA(){
        synchronized(lock){
            a--;
        }
    }

    public int getA(){
        return a;
    }

}
```

4. För varje påstående om text och strängar 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) `String` är en av Javas inbyggda primitiva datatyper. falskt

   b) En fördel med att använda strängkonkatenering med '+'-operatorn är att det ger snabbare kod om man hanterar långa strängar med många sammanslagningar. falskt

   c) En `Scanner` är en *iterator* för `String`-objekt. sant

   d) Ett `String`-objekt i Java går inte att ändra i efterhand, dvs den är vad man kallar *immutable*. sant

   e) Eftersom en `String` är implementerad som en array av `char` så kan den bara innehålla text kodad som `ascii`. falskt

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

   a) XML är ett utrymmeseffektivt sätt att lagra komplexa datastrukturer, som t.ex mätdataloggar från experiment. falskt

   b) XML passar bra för att lagra data med en explicit trädstruktur. sant

   c) En anledning till att lagra data i XML-format kan vara att man vill att det ska vara lättfattligt för en mänsklig läsare. sant

   d) Alla XML-taggar måste alltid förekomma i par, med en start-tag och en slut-tag. falskt

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 `Flyweight` är att minska minnesanvändning. sant

   b) `Composite` används för att minska overhead i parallella program. falskt

   c) *Polymorfism* är ett problem som man ofta kan undvika med `Threadpool`. falskt

   d) Ett vanligt mål med att använda `Observer` är att göra programexekveringen snabbare. falskt

   e) `MVC` är ofta ett lämpligt mönster när man gör program med grafiska användargränssnitt. sant

7. För varje programlistning nedan, ange om koden går att kompilera eller ej. (4 p)

a) `public class ParentClass{` Kompilerar

```
    private int a = 0;

    public class SubClass extends ParentClass{

        private int b;

        public SubClass(){
            b=a;
        }

    }
}
```

b) `public class ParentClass{` Kompilerar EJ

```
    public static void main(String[] args){
        SubClass sc = new ParentClass();
    }

    private class SubClass extends ParentClass{
    }

}
```

c) `public class ParentClass{` Kompilerar

```
    public static void main(String[] args){
        IntFace ifce = new SubClass();
    }

    private static class SubClass extends ParentClass implements IntFace{
    }

    private interface IntFace{
    }

}
```

d) `public final class ParentClass{` Kompilerar EJ

```
    private ParentClass(){
    }

    public static void main(String[] args){
        ParentClass pc = new ParentClass();
        SubClass sc = pc.new SubClass();
    }

    private class SubClass extends ParentClass{
    }

}
```

Del II - fördjupningsfrågor

Följande uppgifter besvaras på separat papper.

8. a) Vad menas med *Lös koppling*? Är det bra eller dåligt? Varför? (2 p)

   Med lös koppling menar man att det explicita beroendet mellan olika klasser är så litet som möjligt, dvs de har så få explicita referenser till varandra som möjligt. Om man har lös koppling kan man ersätta delar av ett program utan att behöva ändra på andra delar. Detta är bra eftersom det blir lättare att underhålla koden, och att byta ut en implementation av en del mot en annan, t.ex byta ut en länkad lista mot en array.

   b) Ge exempel på hur man kan koda för att åstadkomma respektive undvika lös koppling. (2 p)

   Exempel på hur man kan åstadkomma lös koppling är genom att deklara alla variabler som den mest generella typ som stöder de operationer man vill utföra på den, helst bör man deklarera med gränssnitt. Ett antal designmönster, som till exempel Observer, MVC, Iterator, Facade mm har som syfte att åstadkomma lös koppling.

   Om man till exempel skriver kod där man på ett stort antal ställen explicit namnger andra klasser, och använder explicita metodanrop och fältåtkomster på låg nivå i dessa klasser, får man inte lös koppling. OBS, detta är alltså dålig kod!

9. Förklara skillnaden mellan *abstrakta klasser* och *gränssnitt*, och när man bör använda vilken. (2 p)

   Abstrakta klasser är som vanliga klasser i allt utom att de inte går att instansiera. Detta betyder att de kan ärva direkt från en annan klass, de kan innehålla fält och metoder av alla typer som är tillåtna i vanliga klasser, men även icke-implementerade metoder, sk. *abstrakta* metoder som måste implementeras io ärvande subklass om den ska gå att instansiera.

   Gränssnitt innehåller inga fält annat än konstanter, och innehåller bara abstrakta metoder (och klassmetoder och sk default-metoder från och med java 8), men inga "vanliga" instansmetoder. Detta gör att gränssnitt tillåter multipelt arv, samt att alla metoder som finns i ett gränsnitt måste implementeras explicit av implementerande klass.

   Man bör använda gränssnitt när man bara behöver gemensamma metodsignaturer för sina olika klasser, men inte är beroende av gemensamma iplementationer. Abstrakta klasser används när vi antingen vill kunna komma åt gemensamma fält i de implementerande subklasserna, eller då vi vill att en eller flera metoder ska ha samma implemtation. Ett vanligt sätt att göra på kan vara att ha ett gränssnitt som definierar publika metoder, en abstrakt klass som implementerar gränssnittet och

tillhandahåller de gemensamma metodimplementationerna och fälten, och konkreta subklasser som ärver från den abstrakta.

10. Förklara vad en **Threadpool** är, och hur och varför man använder den. (2 p)

    En threadpool betyder att man skapar ett antal trådar som kan hämta uppgifter från en kö. När det kommer in nya uppgifter att lösa så läggs dessa på uppgiftskön i stället för att läggas på en explicit nystartad tråd. När en tråd ur poolen är färdig med sin uppgift hämtar den nästa uppgift ur kön. Om kön är tom läggs tråden på en trådkö ur vilken trådar kan plockas om det kommer in nya uppgifter.

    När man skapar en threadpool brukar man ange hur många trådar man vill ha, där antalet ofta är baserat på vad som är optimalt ur system-/hårdvaruperspektiv. Varje gång en ny uppgift kommer in så ges denna till ett nytt `Runnable`-objekt, som man sedan lägger på uppgiftskön.

    Anledningen till att använda threadpool är att man vill optimera resursanvändning, och undvika att allt för många trådar skapas om det tillfälligt kommer in väldigt många nya uppgifter. Om man inte använder threadpool kan i värsta fall overheaden för att skapa nya trådar äta upp all tillgänlig kapacitet, så att ingen kapacitet alls finns kvar till att utföra själva uppgifterna.

11. Nedan följer fem olika designmönster. Välj ut tre olika kombinationer av två mönster ur denna lista, och beskriv hur dessa mönsterpar kan kombineras med varandra, och varför man skulle vilja göra det. Notera att det ska handla om en direkt kombination av mönster, inte bara en applikation som råkar innehålla båda mönstren. (6 p)

    Builder Factory Singleton Lock Proxy

    Nedan följer ett par exempel, man kan tänka sig fler möjligheter:

    Builder - Factory Man använder buildermönstret inne i fabriksmetoden för att skapa de efterfrågade objekten.

    Factory - Singleton När man genererar kryptonycklar ser man till att nyckelfabriken är en singleton, för att garantera att man aldrig genererar likadana nycklar.

    Lock - Factory Om man har en fabrik för kryptonycklar kan man dessutom låsa fabriksmetoden så att bara en tråd i taget kan generera nycklar, återigen för att garantera att man inte korrumperar slumpfröna, och råkar generara likadana nycklar.

    Lock - Singleton Man ser till att Lock-objektet är en singleton, så kan man garantera att olika metoder i olika klasser bara kan anropas med samma lås, och därmed kan de låsas av en specifik tråd.

    Proxy - Builder Om vi behöver komma åt det objekt som vi bygger i buildern innan det är klart så kan vi använda ett proxyobjekt som ger oss åtkomst till de delar som är färdiga, och som sedan byts ut mot det slutgiltiga objektet när det är klart.

12. Du har bestämt dig för att bli rik och berömd och tänker åstadkomma detta genom att ge dig in i spelbranschen och börja producera enkla billiga spel för olika plattformar som stöder Java. Din marknadsresearch säger dig att det är ganska små skillnader mellan framgångsrika och misslyckade spel, men att slumpen verkar spela stor roll. För att komma runt detta tänkte du börja med att programmera ett praktiskt ramverk som du kan återanvända för att göra olika spel, så att du enkelt kan producera ett stort antal olika spel med förhoppningen att något av dem slår igenom. För att minimera risken att någon stämmer dig på alla dina miljoner när framgången väl kommer, har du beslutat att inte använda några tredjepartsbibliotek.

    Du tänker dig att alla dina spel ska vara enkla och actionbetonade, och ska innehålla ett antal olika figurer och föremål som kan interagera med varandra enligt olika regler, i stil med Angry Birds, Cut The Rope, Flappy Bird, Worms eller liknande succéspel.

    Beskriv hur du konstruerar ditt ramverk för att göra dina enkla spel i, så att du får en bra arkitektur med så mycket återandvändbar och lättmodifierad kod som möjligt. Vilka designmönster använder du, hur, och varför? (8 p)

    Denna uppgift tillåter till sin natur flera olika lösningar. Nedan följer några exempel på vad dessa kan/bör innehålla.

    Till att börja med kan vi konstatera att spel-program till sin natur är väldigt beroende av sina användargränsnitt, därför passar det bra med en MVC-arkitektur.

    Vi skapar en abstrakt `View`, i vilken vi till exempel samlar funktioner för att ladda, visa och panorera över bakgrundsbilder, samt metoder för att animera och rita ut sprite-bilder, samt visa poäng och annan aktuell status på skärmen. De olika spelen får sedan ha konkreta view-klasser som implementerar det som är specifikt för just det spelet.

    På liknande vis skriver vi en `model`-klass för att hålla reda på speltillståndet. Den har information om alla föremål i spelet (t.ex spelarfigurer, hinder, flyttbara föremål, fiender, mm), speltillståndet (t.ex poäng, vilka banor som finns, vilka man har klarat av, osv). Denna har också metoder för hur man sparar eller laddar in relevanta speldata, som t.ex hur långt man har kommit, bästa poäng, mm.

    Vi skapar också en abstrakt `model`-klass med möjlighet att lagra data om alla olika föremål i spelet (t.ex spelarfigurer, hinder, flyttbara föremål, fiender, mm), hur de rör på sig, hur de interagerar med varandra, och metoder för att läsa in och lagra spritedata. Vi kan skapa ett par abstrakta subklasser till denna för att definiera olika huvudtyper av figurer och föremål. För att strukturera upp sammansatta föremål använder vi ett Composite-mönster till våra föremål. Det betyder att det blir lättare att tillämpa spelets fysikmotor på sammansatta föremål, eftersom vi kan behandla dessa på samma sätt som alla andra föremål.

    Varje spel kommer att behöva en `controller`-klass som kopplar ihop modellen med det grafiska och användarens styrsignaler.

En stor del av kommunikationen mellan de olika delarna kan utföra med hjälp av Observer-mönstret: Controller lyssnar på användarens insignaler (knappar, mus, pekskärm, etc), och View lyssnar på model, för att alltid visa spelets nuvarande tillstånd. Vi kan ha en datalogg-klass som lyssnar på modellen och ser till att spara alla relevanta ändringar till en fil så att dessa finns kvar även om spelet skulle krascha eller stängas av oväntat. För dataloggern så använder vi en push-modell för datan, så att modellen kan ange vad den vill att vi ska logga. Därmed kan dataloggerklassen vara gemensam för alla spel.

13. Du hjälper en vän att skriva ett enkelt fotoredigeringsprogram i Java, och enligt din rekommendation använder hen färdiga Swingkomponenter till användargränssnittet. I ritprogrammet kan man applicera olika filter på sin bild genom att trycka på någon av ett antal olika knappar. Vissa filter kan dock ta väldigt lång tid att köra, och din vän lägger för säkerhets skull till en "avbryt"-knapp. Tyvärr fungerar det inte som hen hade tänkt sig, eftersom det inte går att trycka på någon knapp, inklusive "avbryt", medan filterkoden körs, utan resultatet av knapptryckningarna kommer först när filtreringen är färdig. Din vän undrar vad som händer och hur hen ska fixa det. Ge ett hjälpsamt svar! (3 p)

    Det som troligtvis inträffar är att filterkoden anropas direkt från `ActionPerformed` i den klass som lyssnar på knappen. Det betyder att filterkoden kommer att köras av event-tråden i Swing, och den kommer inte att kunna göra något annat (t.ex reagera på andra knapptryckningar) förrän filtermetoden har returnerat. Ett sätt att lösa detta på är att låta `ActionPerformed` anropa en metod som knoppar av en separat tråd för att köra filtret, och returnerar omedelbart. Filtret kan då avbrytas genom att man från eventtråden (genom ett tryck på "avbryt"-knappen) anropar lämplig avbrottsmetod på filterobjektet, och t.ex sätter en avbrottsvariabel till `true`, som får filtrets huvudloop at avbryta.