HÖGSKOLAN
I GÄVLE

AKADEMIN FÖR TEKNIK OCH MILJÖ
Avdelningen för industriell utveckling, IT och samhällsbyggnad

Objektorienterad design och programmering

Instruktioner till laborationerna

Version 7 september 2015, perjee

Innehåll

1 Allmänt

1.1 Introduktion

Laborationerna i kursen OODP har som mål att ge stundenterna möjligheten att öva användningen av olika element i programmeringsspråket Java och ett antal principer för att utveckla välstrukturerade, fungerande och underhållningsbara applikationer. Detta sker genom att utveckla en enkel applikation steg för steg:

Del 1: Klasserna som behövs för att representera programmets datamodell skapas.

Del 2: Styrningen som förmedlar mellan användargränssnitt och datamodellen byggs upp.

Del 3: Programmering av användargränssnitt.

Varje del motsvarar en laboration; laboration 1 kommer alltså att handla om klasser och interface och hur de kan användas för att bygga upp en datamodell osv.

1.2 Problembeskrivning

Uppgiften som ska lösas inom kursens gång är att skapa ett Java-program som kan användas för att rita ut enkla geometriska figurer i ett plan[1]: Punkt, linje, triangel, cirkel, rektangel, ellips, kvadrat.

Användaren ska kunna skapa figurer. Till varje figur ska man kunna bestämma den initiala positionen och storleken.

Vidare ska det vara möjligt att förflytta alla figurer tillsammans, att ändra alla figurernas storlek ('skalning') och att rotera alla figurer. Rotationen ska utföras med figurens medelpunkt som rotationscentrum. Även skalningen ska beräknas med hänsyn till figurens medelpunkt.

Dessutom ska användaren kunna ta bort alla figurer.

Användaren ska kunna kommunicera med programmet via ett grafiskt gränssnitt.

Ett enkelt exempel för ett GUI visas i figur 1. (Obs: Programmet är inte komplett, man kan bara använda det för att skapa punkter och linjer.)

1.3 Användningsfall

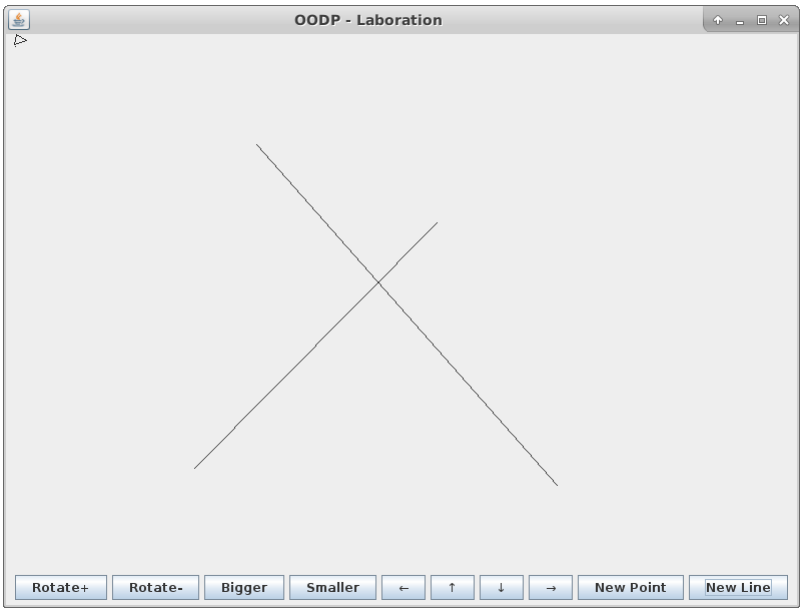
Med utgångspunkt i beskrivningen har man skapat ett användningsfallsdiagram som återges i figur 2 samt användningsfallsbeskrivningar som följer nedan.

Beskrivningar till användningsfallen:

- AF 1: Skapa figur.

  Användaren väljer 'skapa figur' och sedan vilken typ figuren ska ha. Möjliga värden är: Punkt, linje, triangel, rektangel, cirkel och kanske ellips och kvadrat.

[1] Figurerna är alltså definierade i ett kartesiskt koordinatsystem; en position i planet bestäms av en x- och en y-komponent.



Figur 1: Exempelprogram.

Systemet ger användaren möjlighet att mata in värden som behövs för att definiera den önskade figuren. Systemet använder sedan värden för att skapa ett objekt av vald typ och tilldelar objektet till passande *objektlista* (t.ex. figur-lista, lista med flyttbara figurer, lista med roterbara figurer mm. - se nedan).

- AF 2: Ta bort alla figurer.

  Användaren väljer 'Ta bort allt'. Systemet tömmer alla listor.

- AF 3: Rita ut alla figurer.

  Användaren väljer 'Rita ut allt'. Systemet visar alla figurer som kan ritas.

  Obs: Detta användningsfall kommer att behandlas först i laboration 3.

- AF 4: Skriv ut data till alla figurer.

  Användaren väljer 'Skriv ut allt'. Systemet skriver ut data till alla figurer på konsollen.

- AF 5: Förflytta alla flyttbara figurer.

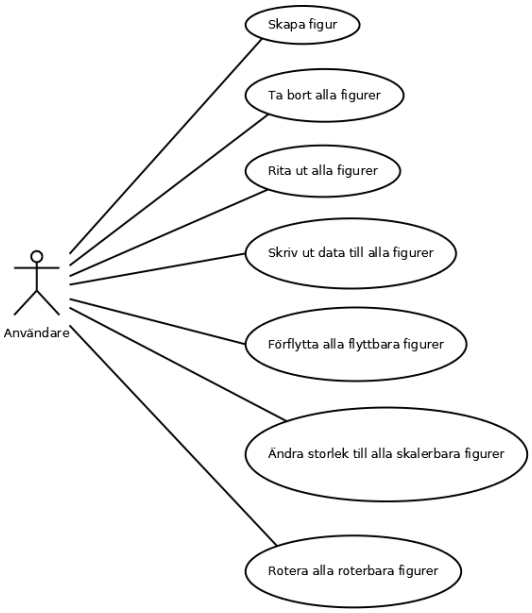
  Användaren väljer 'Förflytta allt'. Systemet ger användaren möjlighet att mata in värden d_x, d_y. Systemet förflyttar alla figurer som kan förflyttas.

- AF 6: Ändra storlek till alla skalerbara figurer.

  Användaren väljer 'Ändra storlek till alla'. Systemet ger användaren möjlighet att mata in värden s_x, s_y. Systemet ändrar storleken till alla figurer som tillåter detta.

- AF 7: Rotera alla roterbara figurer.

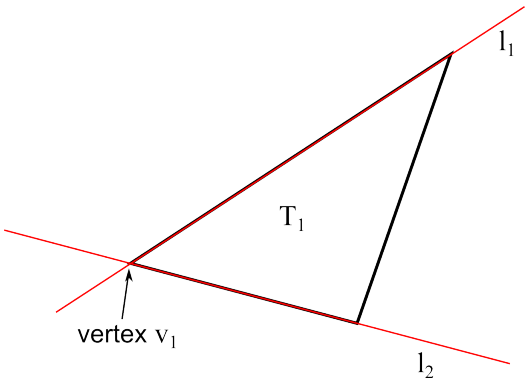
  Användaren väljer 'Rotera alla'. Systemet ger användaren möjlighet att mata in ett vinkel-värde. Systemet roterar alla figurer som kan roteras.



Figur 2: Användningsfallsdiagram.

1.4 Bakgrund

För att modellera figurerna i programmet ska man använda sig av s.k. *noder* (eng. *vertex*). En nod beskriver positionen där t.ex. två linjer i en triangel skär varandra i ett hörn - se fig. 3.



Figur 3: Triangel och vertex.

En nod är alltså en sorts ”markör” för en position. Eftersom en geometrisk figur befinner sig på en position eller sträcker sig mellan flera positioner, så kan man säga att den innehåller (eller består av) en eller flera noder - en punkt en enda nod (den har en position), en linje två (går från en position till en annan), en triangel tre osv.

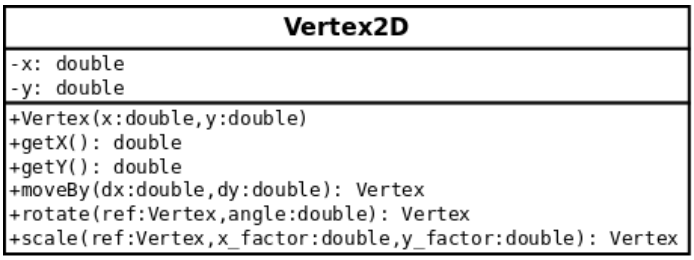
Noder kan man alltså anse som grundläggande element för alla figurer när man vill skapa en datamodell för programmet. Därför finns det en klass `Vertex2D` som definierar en datatyp motsvarande ’nod’ i beskrivningen ovan.

`Vertex2D`-objekt har som egenskaper (resp. attribut) x- och y-koordinat.

`Vertex2D`-objekt ska kunna skapas; ett `Vertex2D`-objekt ska kunna 'frågas' efter sin position.

Att förflytta eller rotera en figur resp. ändra figurens storlek realiseras genom att förflytta/rotera figurens alla noder (möjligtvis med hänsyn till en referenspunkt). Därför är det praktiskt att implementera dessa operationer också i `Vertex2D`-klassen.

Figur 4 visar klassdiagrammet till klassen `Vertex2D`. Klassen är redan imple-



Figur 4: UML-klassdiagram för klassen `Vertex2D`.

menterad. Koden finns tillsammans med en testklass och dokumentationen i filen `oodp_lab_ht2015.zip`.

Rotation och skalning (dvs. ändra storlek) på figurer måste beräknas med hänsyn till en referenspunkt. Specifikationen ovan kräver att operationerna alltid ska utföras med hänsyn till figurens centrum. Referenspunkten är alltså medelpunkten av en figur.

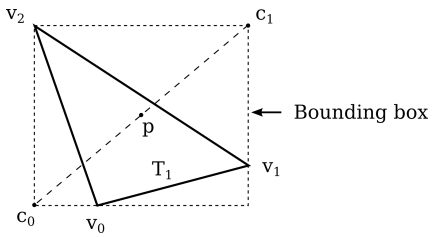
För vissa figurer finns det ett givet centrum: En linje har en medelpunkt, en cirkel likaså.

För andra figurer som triangel eller polygon kan man approximera positionen med hjälp av en inneslutande rektangel ('bounding box'). Rektangelns egenskaper är följande:

- Kanterna är parallella till koordinataxlarna i det tvådimensionella kartesiska koordinatsystemet.
- Positionen p (se figur 5) är medelpunkten av diagonalen från rektangelns nedre hörn till vänster (c_0) till rektangelns övre hörn till höger (c_1).

För att bestämma den inneslutande rektangeln måste ni söka koordinaternas extremvärden från alla noder i en figur:

- y-koordinaten av nedre hörnet till vänster är lika med det minsta av alla y-värden - y_0 (i exemplet i figur 5: y-värdet till v_0);
- x-koordinaten av nedre hörnet till vänster är lika med det minsta av alla x-värden - x_0 (figur 5: x-värdet till v_2);
- y-koordinaten av övre hörnet till höger är lika med det största av alla y-värden - y_1 (figur 5: y-värdet till v_2);
- x-koordinaten av övre hörnet till höger är lika med det största av alla x-värden - x_1 - (figur 5: x-värdet till v_1);



Figur 5: Triangel och 'bounding box'.

Medelpunktens koordinater beräknas sedan på följande sätt:

$$x_p = x_0 + \frac{x_1 - x_0}{2}$$

$$y_p = y_0 + \frac{y_1 - y_0}{2}$$

2 Regler

Varje laboration består av tre obligatoriska uppgifter och en extra, frivillig, uppgift som ger bonuspoäng. Problemen ska lösas i grupper av 2 studenter.

Lämna in lösningarna via inlämningslänken i Blackboard senast vid angiven tid.

Lösningarna skall bestå av en rapport (PDF-fil) som innehåller bl.a. svaren till uppgifterna. En mall som beskriver hur en rapport ska vara strukturerad finns i bilaga A.1. Ifall det krävs att skriva kod i en uppgift, så ska ni även lämna in den tillsammans med tillhörande dokumentation. Spara i så fall koden tillsammans med rapporten i en ZIP-fil. Filen som ni lämnar in ska ha ett namn som beskriver till vilken laboration filen hör och vilken grupp som är avsändare till filen i följande format:

`oodp_lab<lab-nr>ht15_grupp<gruppnamn>.zip`

Exempel: `oodp_lab1ht15_gruppA1.zip`

På morgonen efter en inlämningstids utgång publiceras kl. 8:00 tre rapporter som väljs ut slumpmässigt. Varje student ska skriva en kort text där man diskuterar lösningen[2] till uppgift 1 i rapport 1, uppgift 2 i rapport 2 och uppgift 3 i rapport 3. Texterna ligger till grund för seminariet där laborationen utvärderas. Dokumentet ska lämnas in via Blackboard innan resp. seminarium börjar.

Obs: Eftersom det lottas ut vem som kommer att redovisa sin diskussion i seminariet så är det inte möjligt att delta i seminariet om man inte har lämnat in sin diskussionstext resp. lösningar till laborationen i tid eller inte är närvarande när seminariet börjar.

För reglerna angående betygsättningen se kursinformationen (i dokumentet `oodp_kursinfo_ht15.pdf` på Blackboard). Där står det bland annat att "En laborationsuppgift blir godkänd när lösningen motsvarar specifikationen, laborationen lämnas in i tid och författaren har deltagit i seminariet där laborationen utvärderas."

Det måste förklaras vad det betyder att "lösningen motsvarar specifikationen":

- För det första så måste det finnas en lösning till problemet (svar på fråga resp. kod som löser uppgiften som har ställts).
- Ifall det handlar om en text som innehåller t.ex. förklaringar, diskussioner eller liknande, så ska texten vara läsbar och förståelig.
- När det gäller kod så är det underförstått att koden kan kompileras och exekveras.

  Varning: Lämnar man in kod som inte ens uppfyller dessa grundkrav, så blir laborationen underkänd direkt!

  Men utöver det så ska koden motsvara kraven som ställs (alltså implementerar rätt funktionalitet) och vara skriven i en enhetlig stil. För mer information om programmeringsstil se avsnitt A.2 i bilagan.

Observera att det inte är tillåtet att kopiera andras lösningar!

Ifall det används material (såsom kod, texter eller bilder) i program eller rapport som inte skapades av studenterna i gruppen, så är det ett krav att källorna redovisas!

[2] Handledning för detta arbete finns i avsnitt A.3 i bilagan till detta dokument.

3 Laboration 1
Att modellera typer: Klass, interface, arv. Datamodell till ett program.

3.1 Introduktion

I nästan varje applikation är man konfronterad med uppgiften att hantera data om objekt som finns 'ute i världen' - konkreta objekt som bilar, byggnader, personer..., eller abstrakta som i vårt fall geometriska figurer.

Eftersom kursen handlar om objektorienterad design och programmering, så ska ni modellera geometriska figurer med hjälp av klasser. Alla klasser som representerar någon geometrisk figur ingår i en del av applikationen som man kan kalla för *datamodell* eller *datahållningsskikt*.

3.2 Uppgifter

1.

(a) Modellera klasserna som representerar *konkreta* geometriska figurer som ska kunna hanteras i programmet och därför ska ingå i programmets datamodell: Punkt, linje, triangel, cirkel, rektangel.

Beskriv klasserna på ett liknande sätt som i kursboken, kap 2.2, på sida 65 - tredje upplaga. Skapa dessutom ett UML-klassdiagram för modellen.

Motivera era beslut: Varför skapade ni modellen som den är? Varför har klasserna de attribut och operationer så som ni valde?

(b) Implementera klasserna som ingår i klassdiagrammet och skapa JUnit-tests för att testa koden!

Tips:

Filen `oodp_lab_ht2015.zip` innehåller ett Eclipse-projekt som ni ska använda för ert arbete. I projektet finns klassen `Vertex2D` (se ovan); använd klassen när ni skapar modell-klasserna.

Som redan sagts så roteras t.ex. en figur runt sin medelpunkt genom att rotera figurens noder runt den. En linje kan alltså roteras runt sin medelpunkt `p` genom att anropa `rotate` på start- och slutnod (`angle` lagrar rotationsvinkeln i grader):

```
// rotera startnod omkring positionen:
v0.rotate (p, angle);

// rotera slutnod omkring positionen:
v1.rotate (p, angle);
```

Samma princip går att använda vid storleksändring (med `scale`) och flytt (med `moveBy` - men här behövs ingen referenspunkt).

(c) Vilken relation ser ni mellan klassen `Vertex2D` och figurklasserna resp. mellan instanser av `Vertex2D` och instanser av figurklasserna? På vilket sätt återspeglas relationen i klassdiagrammet? Ge en förklaring!

2.

(a) En superklass kan innehålla attribut och operationer som är gemensamma för ett antal klasser.

Beskriv en superklass till klasserna ni skapade i uppgift 1! Beskrivningen ska innehålla information om vilka attribut det ska finnas i superklassen och vilka meddelanden (kommandon och frågor) man kan skicka till instanser av superklasstypen.

Förklara varför ni bestämde er för just den uppsättning attribut/operationer som ni valde!

(b) Ska superklassen vara en abstrakt klass eller inte? Diskutera[3] vad som talar emot och vad som talar för!

(c) Uppdatera klassdiagrammet från uppgift 1, så att den nya klassen ingår i modellen samt relationerna mellan superklassen och befintliga klasser!

(d) Implementera modellen som ni skapade i (c), dvs. aktualisera koden från uppgift 1 så att den motsvarar klassdiagrammet från (c)!

Använd testen som ni utvecklade i uppgift 1 för att visa att klasserna fungerar som förut!

3.

(a) Betrakta nu modellen en gång till: Ser ni likheter när det gäller objektens beteenden? Vilka är dessa? Ge en beskrivning!

(b) Utöka nu klassdiagrammet en gång till: Lägg till minst ett *interface* som deklarerar någon metod som passar till det som ni beskrev ovan. Låt klasserna implementera interfacet ifall det passar!

(c) Uppdatera koden från uppgift 2 så att den motsvarar den utökade modellen från (b)!

4. - Bonus

Uppgiften kan ge upp till tre bonuspoäng.

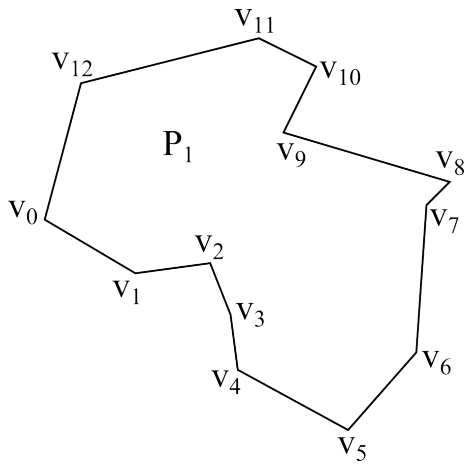
(a) En polygon är en figur som består av ett godtyckligt antal noder med linjedrag mellan noderna. Noderna är ordnade; ett linjedrag ritas ut bara mellan en nod och nodens efterföljare (efterföljare av sista noden är den första) - för ett exempel se figur 6.

Modellera en klass `Polygon` (beskrivning, klassdiagram) och implementera klassen samt skapa lämpliga testfall på ett liknande sätt som i uppgifterna ovan!

Det finns två möjligheter att tilldela noderna till en `Polygon`-instans: En är att använda konstruktorn, den andra en metod som tar emot en position, skapar en nod och lägger den till noderna som redan finns.

Vilken du väljer bestämmer du själv (kanske att det finns andra möjligheter också), men du ska förklara ditt val och diskutera för och nackdelar med det!

[3]Med utmaningen 'diskutera' menas här att ni ska skriva ner för- och nackdelar och även slutsatserna som ni drar när ni jämför dem.



Figur 6: Polygon.

Tips:

För att lagra `Vertex2D`-instanserna kan ni använda klassen `ArrayList`:

```
ArrayList<Vertex2D> vertices = new ArrayList<Vertex2D> ();
```

(b) Diskutera: Vilken klass ärver av vilken (det är möjligt att vissa klasser varken är sub- eller superklass)

- Punkt
- Ellips
- Triangel
- Kvadrat
- Linje
- Polygon
- Cirkel
- Rektangel

Använd svaret på frågan om en viss klass B är mer specialiserad än en viss klass A resp. omvänt: Representerar A något som är mer generellt än B?

(c) Utgå från resultaten från (b) och uppdatera klassdiagrammet en gång till: Lägg till klasser som representerar typerna ellips och kvadrat! Uppdatera även relationerna mellan klasserna.

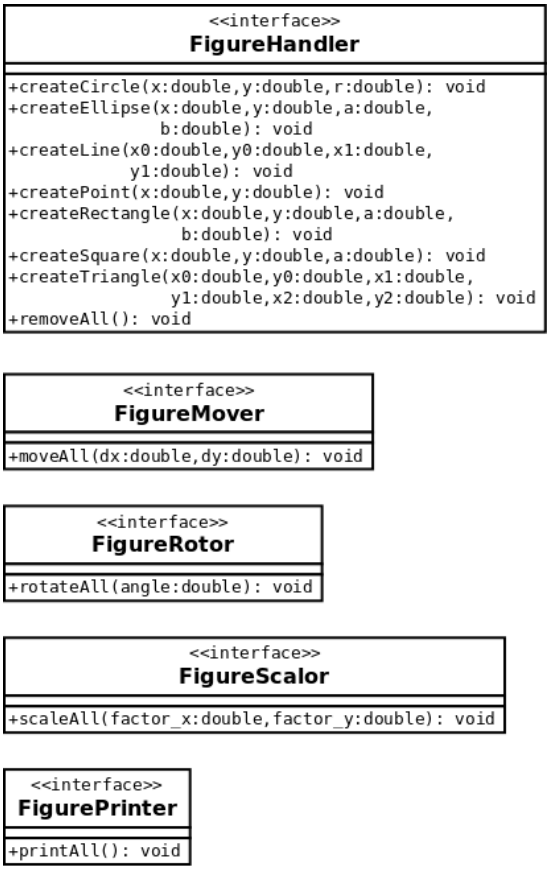
Implementera ändringarna i kod: Lägg till klasserna för figurtyperna ellips och kvadrat till datamodellen. Uppdatera de andra klasserna så att förhållandena i koden motsvarar UML-klassdiagrammet.

4 Laboration 2
Relationer mellan objekt.
Program-styrning.

4.1 Introduktion

Laboration 2 handlar om att skapa koden för logiken som behövs för att kunna förmedla mellan användarens input (via UI) och objekten i datamodellen. På vilket sätt detta ska ske måste bestämmas med utgångspunkt i änvändningsfallsdiagrammet resp. användningsfallsbeskrivningar - se avsnitt 1.3.

För att underlätta arbetet har man definierat en uppsättning interface som beskriver API:et till programstyrningen. Interfacen visas i klassdiagrammet i figur 7.



Figur 7: Styrnings-interface.

Interfacen ska implementeras i den del av systemet som ansvarar för applikationens logik. Samtidigt används interfacen i koden för användargränssnittet. På så sätt blir det möjligt att utveckla styrnings-logik och UI oberoende av varandra.

Ur AF-beskrivningarna framgår att figur-objekten ska lagras i olika listor. I en sådan lista lagras objekt som kan hanteras på liknande sätt[4] (vissa objekt som kan förstoras/förminskas samlas i en lista för sig, andra som kan roteras samlas i en lista för sig osv).

I specifikationen finns ingen information om en möjlig klass i vilken listorna ska förvaltas; det är alltså utvecklarens ansvar att fatta detta beslut. Om man betraktar interfacen, så ser man att `FigureHandler` innehåller alla metoder för att skapa figurer. Även metoden `removeAll` finns här. Därför är det en bra möjlighet att förvalta listorna i klassen som implementerar `FigureHandler`.

Java-API:et erbjuder att antal container-klasser som t.ex. `java.util.ArrayList`. En sådan klass kan användas i klassen `FigureHandlerImpl`. Flyttbara objekt t.ex. skulle kunna lagras i en instans av klassen `ArrayList` som parametriseras med typen för flyttbara objekt.

4.2 Uppgifter

1.

Som sagts ovan, så förväntar man sig att kunna gruppera figur-klasserna: Objekt som kan flyttas hamnar i gruppen 'flyttbara', sådana som kan roteras i gruppen 'roterbara' osv.

En sådan gruppindelning kan förverkligas med hjälp av interface: T. ex. kan ett objekt som kan roteras vara en instans till en klass som implementerar ett visst interface. Interfacet i sin tur deklarerar en metod för rotationen med en viss vinkel.

(a) I laboration 1 definierade ni redan *ett* interface. Kontrollera om interfacet passar in i mönstret som beskrevs ovan och gör ändringar ifall interfacet måste anpassas.

(b) Modellera alla interfacen som behövs utöver interfacet från (a) (beskrivning och UML-diagram; uppdatera klassdiagrammet).

(c) Skriv interfacen från (b) i Java.

Rekommendation: För att kunna hålla isär klasserna som används i dataskiktet, klasserna som tillhör styrningen och GUI-klasser så kan ni använda *packages*. Definiera ett paket för varje grupp och flytta klasserna som ni skapade hittills till respektive paket.

2.

(a) Ändra modellklasserna från laboration 1 så att de implementerar interfacen ifall det passar.

(b) Varje klass i dataskiktet som kan instansieras ska kompletteras med en metod `toString` (för mer information se PDF-filen `Chapter3.pdf`, 'Item 10: Always override `toString`').

(c) Diskutera: Varför används typer som `List<FigureType>`[5] i `FigureHandler`?

Kan man inte använda bara `List`? Eller en array?

[4] Observera att det är möjligt att ett objekt både kan roteras och förflyttas och dessutom ändrar storleken. Det måste alltså tilldelas olika listor.

[5] `FigureType` är bara en platshållare här.

3.

(a) När metoden `createPoint` anropas på ett objekt av typen `FigureHandler`, så måste det skickas ett antal meddelanden mellan olika objekt. Visa meddelanden med hjälp av ett sekvensdiagram!

(b) Skriv klasserna som implementerar interfacen i styrnings-API:et enligt figur 7.

Tips: Paketet `se.hig.oodp.lab3.control` innehåller som ett exempel både interfacet `FigurePrinter` och klassen `FigurePrinterImpl` som implementerar interfacet.

4. - Bonus

Uppgiften är inte obligatorisk och kan ge två bonuspoäng.

(a) Objekten av klasserna i styrnings-API:et är beroende av varandra på olika sätt: Klasserna måste instansieras i en viss ordning och instanserna måste tilldelas korrekt.

Det är önskvärt att initieringen implementeras t.ex. i en särskild klass. Att starta programmet skulle då bestå av instansieringen av en enda klass i programmets `main`-metod, följd av några metodanrop för att få tillgång till objekten i styrningen. Möjligtvis kan man hitta en lösningsmall i form av ett designmönster som beskriver en generell lösning.

Undersök följande designmönster:

- 'Abstract factory'
- 'Builder'
- 'Factory method'

Diskutera om de går att använda i det här sammanhanget!

(b) Skriv kod som löser problemet som beskrivs i (a)!

5 Laboration 3
Kommunikation mellan objekt.
'Putting it all together' - GUI.

5.1 Introduktion

I denna laboration ska ni avsluta arbetet med applikationen: Resultatet ska vara ett program som kan användas och som uppfyller de kraven som beskrivs i avsnitt 1.2.

Det betyder att ni måste skapa ett grafiskt användargränssnitt som innehåller två delar:

- Styrelement för att skapa resp. ta bort figurer och manipulera figurerna (flytta, rotera,...)
- En ritpanel där figurerna ritas ut.

Figur 1 visar ett exempel som ni kan utgå ifrån.

Dessutom ska ni koppla ihop GUI-klasserna med styrningen som ni skapade i laboration 2.

Slutligen måste AF 3 implementeras. När ni löser denna uppgift, så är det särskilt viktigt att ni implementerar ritmetoderna på ett sätt som låter klasserna i styrningen resp. datamodellen vara oberoende av GUI-klasserna. Dvs. de ritmetoder som finns i GUI:et får inte användas direkt i styrningen resp. dataskiktet.

5.2 Uppgifter

1.

Skapa alla GUI-klasser som behövs för att implementera funktionerna som anges i användningsfallsbeskrivningarna (se avsnitt 1.3).

(a) Börja med ett fönster (t.ex. `JFrame`) som innehåller ett antal paneler (t.ex. `JPanel`) och lägg till inmatnigselementen som ni behöver. Implementera även koden för händelsehanteringen, men använd den bara för att skriva ut textmeddelanden på konsollen. Testa så att allt funkar!

Obs: Koden för ritning av figurer ska ni skriva först i uppgift 2!

(b) Koppla därefter koden i styrnings-API:et till metoderna för händelsehantering i GUI:et - lägg till anrop av metoder i styrningen.

2.

Ritningar av geometriska figurer kan man betrakta som sammansatta av enkla element - linje och cirkel. När en punkt ska ritas ut, så kan den symboliseras med hjälp av en liten cirkel (5px i diameter, t.ex.), för en triangel eller en rektangel ritas ut linjer mellan varje hörn. Idéen är att visualisera abstrakta geometriska figurer med hjälp av några få grafiska element (som också kallas för 'primitiver').

Att rita ut grafiska objekt i ett GUI kräver att figurernas data överförs till GUI:ets grafikrutiner. Sådana metoder är kopplade till en s.k. *grafik-kontext* som representeras i Java t.ex. av instanser till klassen `java.awt.Graphics`.

Exempel: Det finns en metod `drawLine (int x1, int y1, int x2, int y2)` i klassen `java.awt.Graphics`. För att kunna använda en `java.awt.Graphics`-instans för att 'rita ut' t.ex. ett linjeobjekt så behövs kunskap om koordinaterna till start- och slutnod av linjen. Man är alltså tvungen att anropa nodinstansernas metoder `getX ()` resp. `getY ()` på något sätt.

Svårigheten som uppstår här är att bestämma vart detta ska hända. Om man returnerar noderna från figurinstanserna till ritningskoden, så bryts kanske kapslingen i figurklasserna.

Om man däremot tilldelar grafikkontexten till figurerna, så skapar man en beroenderelation mellan figurklasserna och GUI. Att byta ut användargränssnittet skulle i så fall kräva ändringar i datamodellen. Dvs. beroenderelationen skulle försvåra återanvändning av koden.

För att hålla designen flexibel bestämdes istället att figur-klasser och grafikkontext inte ska kopplas ihop direkt. Istället skall ritningsrutinerna implementeras på följande sätt:

(a) Första steget är att skapa ett interface `Drawable` som definierar en metod `draw (PrimitivesPainter ppainter)` (se (c) för typen `PrimitivesPainter`). Alla figurer som kan ritas ut ska implementera interfacet, dvs. sådana objekt ska vara av typen `Drawable`.

(b) Fortsätt med att skriva ett interface som definierar en typ `FigurePainter` med en enda metod `paintAll ()`. En klass som implementerar interfacet innehåller en lista med `Drawable`-instanser; när metoden `paintAll ()` anropas, så itereras genom listan och metoden `draw` anropas på alla `Drawable`-objekt. Det betyder också att en `FigurePainter`-objekt måste tillhandahålla en `PrimitivesPainter`.

(c) Det har bestämts att kommunikationen mellan figur-instanser och grafikkontext skall förmedlas av instanser av klasser som implementerar interfacet `PrimitivesPainter` (del av `se.hig.oodp.lab.control`):

```
public interface PrimitivesPainter
{
    public void paintPoint (Vertex2D v);
    public void paintLine (Vertex2D v0, Vertex2D v1);
    public void paintEllipse (Vertex2D v, double a, double b);
}
```

En klass som implementerar `PrimitivesPainter` måste innehålla en referens till en grafik-kontext och använda den i metoderna. Exempel:

```
public class PrimitivesPainterImpl
                  implements PrimitivesPainter
{
    private Graphics g;

    public void setGraphics (Graphics g)
    {
        this.g = g;
```

```
    }


    // Exempel för en möjlig implementering av en metod:
    @Override
    public void paintLine (Vertex2D v0, Vertex2D v1)
    {
        // Obs: Konvertering mellan figur- och grafik-
        // koordinatsystemet saknas!
        g.drawLine ((int)v0.getX(), (int)v0.getY(), (int)v1.getX(), (int)v1.getY());
    }

    // Resten av koden borttagen
}
```

`PrimitivesPainter`-objekt används sedan av instanser av klassen som implementerar interfacet `FigurePainter` från (b).

Här är uppgiften att implementera alla metoder i klassen `PrimitivesPainterImpl`. Ifall ni anser att det behövs fler metoder, så ska ni lägga till även dem till interfacet resp. klassen.

(d) Komplettera nu figurklasserna: Varje klass som representerar en figur som kan ritas ut ska implementera interfacet `Drawable`.

I en klass som implementerar `Drawable` används sedan den `PrimitivesPainter`-metoden som passar till figuren (exempel: `Point`):

```
public class Point implements Drawable
{
    private Vertex2D p;

    // Kod borttaget

    @Override
    public void draw (PrimitivesPainter gpp)
    {
        gpp.paintPoint (p);
    }

    // Resten av koden borttagen
}
```

Tips: För att kunna testa om alla komponenter i kedjan funkar så kan ni skapa en klass `TextOutputPainterImpl` som implementerar `PrimitivesPainter`. Klassen ska kunna 'rita ut' figurer på konsollen, dvs. när en ritmetod anropas, så ska det skrivas ut ett meddelande. Exempel (i `paintPoint`):

```
System.out.println(Drawing Point@("+ v.getX() + ":"+ v.getY() + ")");
```

3.

Genom att lösa uppgift 2 så har ni avslutat implementeringen av applikationen.

(a) Om ni nu betraktar hela programmet: Hur skulle ni beskriva programmets struktur (eller arkitektur)?

(b) Finns det något i designen av programmet som gjorde det särskilt svårt att implementera det? Ifall det är så: Vad är orsaken? Kan ni föreslå en förbättring?

(c) Det finns ett designmönster som beskriver ett annat sätt att strukturera ett system: *Model-View-Controller*. Beskriv i era egna ord MVC-mönstret! Vilka fördelar har en sådan struktur? Vilka nackdelar?

(d) Om ni jämför strukturen som ert program har med MVC: Skulle det finnas fördelar med att använda MVC-mönstret? Varför, eller varför inte?

4. - Bonus

Uppgiften är inte obligatorisk och kan ge två bonuspoäng.

Lägg till funktioner för att exportera innehållet av ritpanelen till en bildfil. Avsikten är att kunna spara ritningar som digitala bilder i JPEG- eller PNG-format.

Tips: Använd klasserna `BufferedImage` och `ImageIO`.

A Appendix

A.1 Mall för rapporter

Laborationsrapporter ska vara strukturerade på följande sätt:

1. Titel, författarna.
2. "Inledning" (problembeskrivning - vad handlar laborationen om).
3. "Förutsättningar och krav" (på del av programmet som bearbetas i resp. laboration).
4. "Resultat" (Till varje uppgift: Kort beskrivning av lösningen, dvs. implementering och test. Svar på frågorna).

A.2 Om programmeringsstil

När ni skriver kod så ska ni alltid tänka på att ni inte skriver för kompilatorn - den vet ingenting om betydelsen av variabelnamn, och den kommer att processa programkoden oavsett om ni skriver allt på en rad eller helt oformaterad. Det är era kollegor och inte minst ni själva som är programtextens läsare och som antagligen kommer att vara tacksamma om koden är välformaterad och dessutom innehåller meningsfulla namn.

Det finns olika möjligheter att göra kod läsbar och lättare att förstå.

Läsbar kod *indenteras*. Jämför följande två exempel:

`Binomial.java`, version 1:

```
import java.util.Scanner;

public class Binomial {
public static int factorial (int n) {
int sum = 1;
if (n > 1) {for (int i = 1; i <= n; i++) {sum *= i;
}}

return sum;
} public static int metod (int a, int b) {
int v1, v2, v3;
/* Metodanrop I: */
v1 = factorial (a);
/* Metodanrop II: */
v2 = factorial (b);
/* Metodanrop III: */
v3 = factorial (a - b);
/* Resultatet beräknas och returneras: */
return v1 / (v2 * v3);
}}
```

`Binomial.java`, version 2:

```
import java.util.Scanner;

public class Binomial {
    /**
     * Metoden factorial beräknar funktionen n!, dvs. n! = 1*2*3*...*(n-1)*n
     * till ett icke-negativt värde som har tilldelats variabeln 'n'.
     * TODO: Lägg till kod för att skicka ett felmeddelande
     * ifall n < 0.
     * @param n - heltalsvärde
     * @return n!
     */
    public static int factorial (int n) {
        int product = 1;

        if (n > 1) {
            for (int i = 1; i <= n; i++) {
                product *= i;
            }
        }
        return product;
    }

    /**
     * Metoden binomialCoeff beräknar binomialkoefficienten till talen n och k.
     * @param n - heltalsvärde
     * @param k - heltalsvärde
     * @return binomialkoefficient till talen n och k
     */
    public static int binomialCoeff (int n, int k) {
        int v1, v2, v3;

        /* Metodanrop I: */
        v1 = factorial (n);
        /* Metodanrop II: */
        v2 = factorial (k);
        /* Metodanrop III: */
        v3 = factorial (n - k);

        /* Resultatet beräknas och returneras: */
        return v1 / (v2 * v3);
    }
}
```

Det behövs inte någon kommentar här.

Det är alltså ett krav att ni indenterar. Hur är däremot upp till er, det kan vara två blanksteg som ni använder, tre eller fyra. Huvudsaken är att ni är konsekventa. Tabbar är däremot inte tillåten, för de måste tolkas, och då vet man inte hur resultatet kommer att se ut när filen öppnas på en annan dator/i en annan editor.

När ska man indentera? När man öppnar ett nytt block. Och när blocket är slut, så måste indraget minskas igen.

Utöver indragningen så är det bra att positionera '{' och '}' ('måsvingesparenteser') på ett enhetligt sätt. Programmerare har använt olika stilar som t.ex.

```
public void call (int number){
    for (int i = 0; i < number; i++){
        System.out.println ("Calling " + i);
    }
}
```

eller

```
public void call (int number)
{
    for (int i = 0; i < number; i++
    {
        System.out.println ("Calling " + i);
    }
}
```

eller också

```
public void call (int number)
  {
  for (int i = 0; i < number; i++
    {
      System.out.println ("Calling " + i);
    }
  }
```

Vilken stil ni använder bestämmer ni själva - huvudsaken är även här att ni är konsekventa.[6]

En viktig del av ett program är betecknare, dvs. namn till klasser, variabler och metoder. Ett namn ska återspegla betydelsen av objektet som betecknas. Det blir inte lättare att förstå meningen med en metod om metodens namn är just - `metod`...

Ni ska inte heller vilseleda folk genom att beteckna t.ex. en variabel där ni vill lagra en produkt som `sum`.

Välj alltså namn som passar till objektets ändamål!

Namn kan skrivas på olika sätt, och här finns det några konventioner som är värda att användas, för de underlättar att följa exekveringsflödet i ett program:

- Klassnamn börjar *alltid* med en versal (stor bokstav). Ifall ett klassnamn är sammansatt utav flera begrepp, så är det bra att använda "CamelCase"-stavningen. Exempel: `PageRequest` (men inte `pagerequest`).
- Metod- och variabelnamn börjar med en liten bokstav (gemen). Använd även här gärna "camelCase"-stavningen. Exempel: `processRequest` (men inte `processrequest` eller `Processrequest`).
- Enskilda bokstäver som t.ex. `i`, `j` eller `k` ska reserveras som namn till räknevariabler (i loopar) eller när man lagrar index till arrayer. Det är också konvention att använda `x`, `y` och `z` inom kod för matematiska beräkningar eller när man vill lagra geometriska värden (koordinater).

[6] Kommentar: Författaren föredrar mellersta versionen (man ser lätt block-strukturen) och använder den i sin kod.

A.3 Rekommendationer för seminarier - Att granska andras inlämningsuppgifter

Det är inte lätt att granska material som lämnats in av en annan student, därför följer här några rekommendationer.

För att få en uppfattning om rapporten kan man försöka hitta svaren till några frågor:

- Tolkas uppgifterna på rätt sätt?
- Om man nu betraktar en uppgift: Är svaret relevant, dvs. passar svaret till frågan eller känns det som 'goddag yxskaft'?
- Är svaret fullständigt? Hur är det med korrektheten?
- Är rapporten i sin helhet läsbar (formatering, skrift)?
- Är språket förståeligt?
- Har man tillgång till koden? Det kan vara viktigt som komplement till rapporten (om det är något som man inte förstår så får man kanske ytterligare information där).

I själva seminariet:

Försök att kommentera delrapporten som du talar om i sin helhet. Börja med det som är bra, sedan kan man ta upp det som skulle kunna förbättras.

Det är också möjligt att ställa frågor till rapportens författare!

Det viktigaste är att vara konstruktiv. Undvik att fokusera på stavningsfel eller rapportens yttre form eller dylikt. Vokabler som 'värdelöst' eller liknande är inte tillåtna!

A.4 Testdata

Punkt

Initiala värden: $x = 2.0$, $y = 1.0$

- Test konstruktor
  Förväntad position: $x = 2.0$, $y = 1.0$
- Test `moveBy`
  Indata: $dx = 3.0$, $dy = 3.0$
  Förväntad position: $x = 5.0$, $y = 4.0$

Linje

Initiala värden: $x_0 = 0.0$, $y_0 = 0.0$, $x_1 = 4.0$, $y_1 = 2.0$

- Test konstruktor
  Förväntad position: $x = 2.0$, $y = 1.0$
- Test `moveBy`
  Indata: $dx = 3.0$, $dy = 3.0$
  Förväntad position: $x = 5.0$, $y = 4.0$
  Förväntad v_0: $x = 3.0$, $y = 3.0$
  Förväntad v_1: $x = 7.0$, $y = 5.0$

- Test `scale`
  Indata: $dx = 1.21$, $dy = 1.21$
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = -0.42$, $y = -0.21$
  Förväntad v_1: $x = 4.42$, $y = 2.21$
- Test `rotate`
  Indata: $angle = 30.0$
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = 0.768$, $y = -0.866$
  Förväntad v_1: $x = 3.232$, $y = 2.866$

Triangel

Initiala värden: $x_0 = -1.0$, $y_0 = 1.0$, $x_1 = 3.0$, $y_1 = -1.0$, $x_2 = 5.0$, $y_2 = 3.0$

- Test konstruktor
  Förväntad position: $x = 2.0$, $y = 1.0$
- Test `moveBy`
  Indata: $dx = 3.0$, $dy = 3.0$
  Förväntad position: $x = 5.0$, $y = 4.0$
  Förväntad v_0: $x = 2.0$, $y = 4.0$
  Förväntad v_1: $x = 6.0$, $y = 2.0$
  Förväntad v_2: $x = 8.0$, $y = 6.0$
- Test `scale`
  Indata: $dx = 1.21$, $dy = 1.21$ Examensarbete för högskoleexamen
  Titel
  Abstract
  Inledning (problembeskrivning)
  Förutsättningar och krav (på program/system)
  Beskrivning av lösningen
  Implementering och test
  Resultat
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = -1.63$, $y = 1.0$
  Förväntad v_1: $x = 3.21$, $y = -1.42$
  Förväntad v_2: $x = 5.63$, $y = 3.42$
- Test `rotate`
  Indata: $angle = 30.0$
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = -0.598$, $y = -0.5$
  Förväntad v_1: $x = 3.866$, $y = -0.232$
  Förväntad v_2: $x = 3.598$, $y = 4.232$

Cirkel

Initiala värden: $x_0 = 2.0$, $y_0 = 1.0$, $r = 1.0$

- Test konstruktor
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad radie: $r = 1.0$

- Test `moveBy`
  Indata: $dx = 3.0$, $dy = 3.0$
  Förväntad position: $x = 5.0$, $y = 4.0$
  Förväntad radie: $r = 1.0$
- Test `scale`
  Indata: $d = 1.21$
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad radie: $r = 1.21$

Rektangel

Initiala värden:

Position bestäms av

- antingen koordinater till övre hörn till vänster: $x = -1.0$, $y = 3.0$
- eller koordinater till centrum: $x = 2.0$, $y = 1.0$

Längder sidor: $a = 6$, $b = 4$

- Test konstruktor
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = -1.0$, $y = -1.0$
  Förväntad v_1: $x = 5.0$, $y = -1.0$
  Förväntad v_2: $x = 5.0$, $y = 3.0$
  Förväntad v_3: $x = -1.0$, $y = 3.0$
- Test `moveBy`
  Indata: $dx = 3.0$, $dy = 3.0$
  Förväntad position: $x = 5.0$, $y = 4.0$
  Förväntad v_0: $x = 2.0$, $y = 2.0$
  Förväntad v_1: $x = 8.0$, $y = 2.0$
  Förväntad v_2: $x = 8.0$, $y = 6.0$
  Förväntad v_3: $x = 2.0$, $y = 6.0$
- Test `scale`
  Indata: $dx = 1.21$, $dy = 1.21$
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = -1.63$, $y = -1.42$
  Förväntad v_1: $x = 5.63$, $y = -1.42$
  Förväntad v_2: $x = 5.63$, $y = 3.42$
  Förväntad v_3: $x = -1.63$, $y = 3.42$
- Test `rotate`
  Indata: $angle = 30.0$
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = 0.402$, $y = -2.232$
  Förväntad v_1: $x = 5.598$, $y = 0.768$
  Förväntad v_2: $x = 3.598$, $y = 4.232$
  Förväntad v_3: $x = -1.598$, $y = 1.232$

Kvadrat

Initiala värden:

Position bestäms av

- antingen koordinater till övre hörn till vänster: $x = 0.0$, $y = 3.0$
- eller koordinater till centrum: $x = 2.0$, $y = 1.0$

Längd sida: $a = 4$

$x_0 = 0.0$, $y_0 = -1.0$, $x_1 = 4.0$, $y_1 = -1.0$, $x_2 = 4.0$, $y_2 = 3.0$, $x_3 = 0.0$, $y_3 = 3.0$

- Test konstruktor
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = 0.0$, $y = -1.0$
  Förväntad v_1: $x = 4.0$, $y = -1.0$
  Förväntad v_2: $x = 4.0$, $y = 3.0$
  Förväntad v_3: $x = 0.0$, $y = 3.0$
- Test `moveBy`
  Indata: $dx = 3.0$, $dy = 3.0$
  Förväntad position: $x = 5.0$, $y = 4.0$
  Förväntad v_0: $x = 3.0$, $y = 2.0$
  Förväntad v_1: $x = 7.0$, $y = 2.0$
  Förväntad v_2: $x = 7.0$, $y = 6.0$
  Förväntad v_3: $x = 3.0$, $y = 6.0$
- Test `scale`
  Indata: $dx = 1.21$, $dy = 1.21$
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = -0.42$, $y = -1.42$
  Förväntad v_1: $x = 4.42$, $y = -1.42$
  Förväntad v_2: $x = 4.42$, $y = 3.42$
  Förväntad v_3: $x = -0.42$, $y = 3.42$
- Test `rotate`
  Indata: $angle = 30.0$
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad v_0: $x = 1.268$, $y = -1.732$
  Förväntad v_1: $x = 4.732$, $y = 0.268$
  Förväntad v_2: $x = 2.732$, $y = 3.732$
  Förväntad v_3: $x = -0.732$, $y = 1.732$

Ellipse

Initiala värden: $x_0 = 2.0$, $y_0 = 1.0$, $a = 1.0$, $b = 0.7$

- Test konstruktor
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad storlek: $a = 1.0$, $b = 0.7$
- Test `moveBy`
  Indata: $dx = 3.0$, $dy = 3.0$
  Förväntad position: $x = 5.0$, $y = 4.0$
  Förväntad storlek: $a = 1.0$, $b = 0.7$
- Test `scale`
  Indata: $dx = 1.21$, $dy = 1.21$
  Förväntad position: $x = 2.0$, $y = 1.0$
  Förväntad storlek: $a = 1.21$, $b = 0.847$