DVG303
Objektorienterad design och programmering
Laboration 1

Jonas Sjöberg
860224
Högskolan i Gävle
tel12jsg@tudent.hig.se
https://github.com/jonasjberg

Datum: 2015-10-15
Kursansvarig lärare: Peter Jenke

Sammanfattning

Laborationsrapport för *DVG303 – Objektorienterad design och programmering*, Högskolan i Gävle, Höstterminen 2015.

Innehåll

Figurer

Introduktion

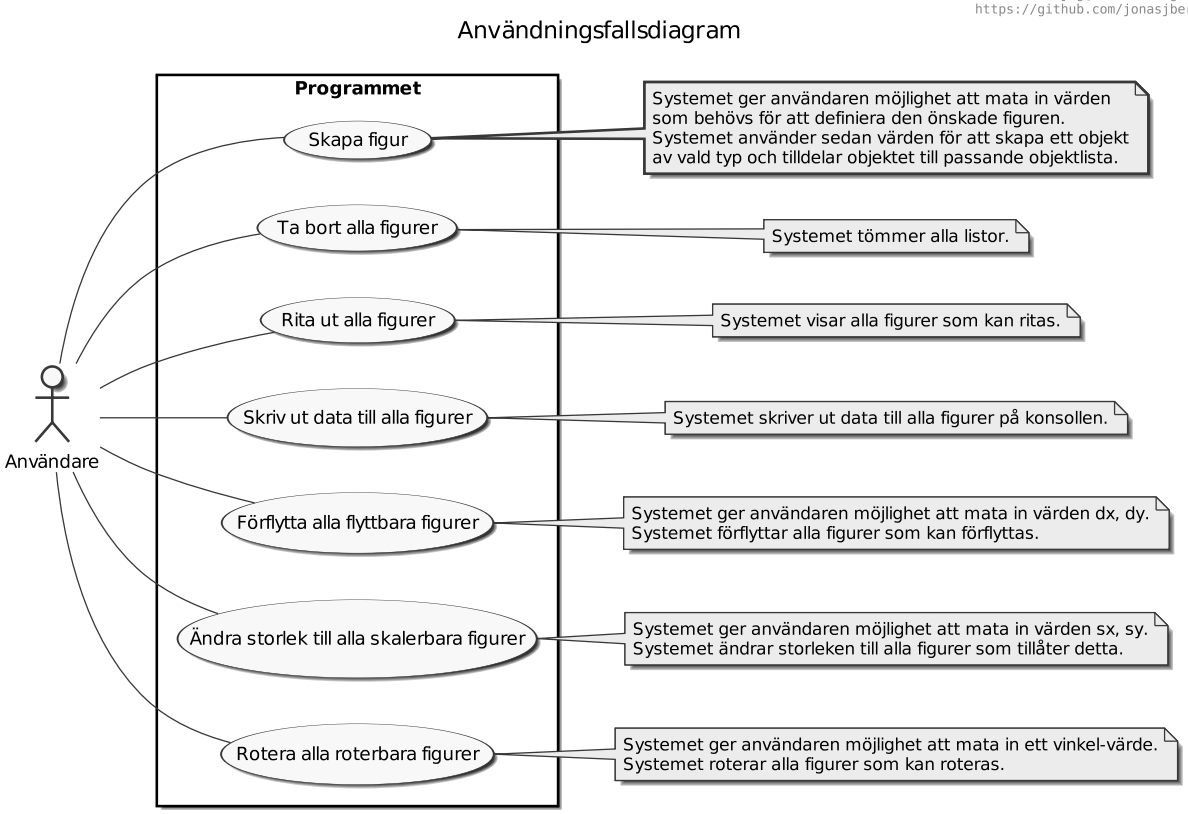
Övergripande beskrivning

Det här är den första av tre laborationer i objektorienterad design och programmering. Ett fullständigt program kommer att utvecklas under laborationerna. Processen kommer att innehålla många element av professionell mjukvaruutveckling; design, dokumentation, revisionskontroll, etc., och syftar till att utveckla praktiska färdigheter i mjukvaruutveckling.

Specifikation

Den funktionalitet som efterfrågas beskrivs enligt UML-standardiserat användningsfallsdiagram (*"use case diagram"*) i Figur 1.

Jonas Sjöberg 860224
Högskolan i Gävle
tel12jsg@student.hig.se
https://github.com/jonasjberg



Figur 1: Användningsfallsdiagram (`diagram/usecase.eps`)

Arbetsmetod

- Koden skrivs i utvecklingsmiljön `Intellij IDEA` under `Linux 3.19.0-28-generic` och kompileras samt exekveras med följande `Java`-version:

```
> $ java -version
java version "1.7.0_79"
OpenJDK Runtime Environment (IcedTea 2.5.6) (7u79-2.5.6-0ubuntu1.15.04.1)
OpenJDK Server VM (build 24.79-b02, mixed mode)
```

- Rapporten skrivs i med texteditorn `Vim` och kompileras till pdf med `latexmk`.

  Diagram och figurer skrivs i `PlantUML`-format och renderas med `Graphviz`. Resultatet förhandsgranskas i realtid med hjälp av plugins i `Intellij IDEA`.

- För revisionskontroll används `Git`.

UML-diagram och dokumentation uppdateras löpande parallellt med källkoden. Förändringar i källkoden har kommit att följa uppdaterade diagram likväl som diagrammen har behövt uppdateras för att reflektera förändrad källkod eller funktionalitet.

Varje uppgift finns representerad som en separat utvecklingsgren (*branch*) i `Git`. På så vis kan varje uppgift utvecklas oberoende utan redundans och filduplicering. Det är också mycket enkelt att propagera förändringar mellan grenar och individuella *commits* med hjälp av ändamålsriktiga "diff"-verktyg. Ett sådant ingår i standarddistributioner av `Git`, ett annant exempel är `Meld`.

Källkod

Källkod till programmet och rapporten finns att hämta på `https://github.com/jonasjberg`.

1 Uppgift 1

(a)

Frågeställning

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?

Lösning

Figurerna beskrivs av ett visst antal punkter eller positioner som representeras av `Vertex2D`-instanser. Alla figurer består av minst ett `Vertex2D`-objekt som representerar figurens mittpunkt. Figurerna kan delas upp i två separata grenar i arvshierarkin, `Figure` och `SimpleFigure`. Skillnaden mellan Figure och SimpleFigure är hur många punkter de lagrar för att beskriva figuren de representerar. Figurer som kan beskrivas med bara en `Vertex2D`-instans, figurens position, ärver från `SimpleFigure`; punkt, cirkel och ellips. Figurer som består av fler `Vertex2D`-objekt, ärver från `Figure`. Klassen `Figure` lagrar sina punkter i en lista för enkel åtkomst. Det möjliggör också skapandet av figurer med ett godtyckligt antal punkter.

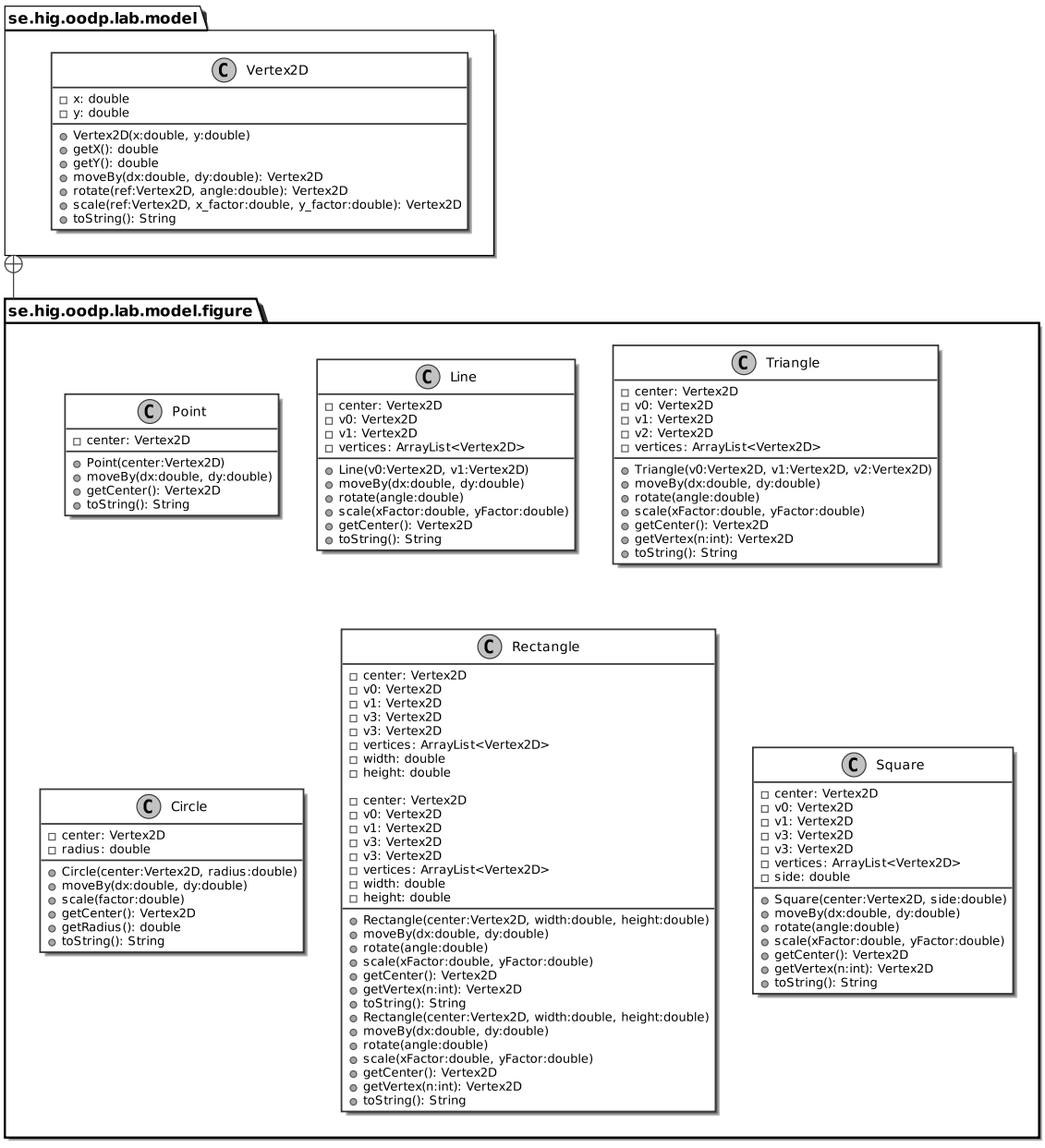
UML-klassdiagrammet återfinns i Figur 2, samt bifogad fil.

(b)

Frågeställning

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

Jonas Sjöberg 860224
Högskolan i Gävle
tel12jsg@student.hig.se
https://github.com/jonasjberg



Figur 2: Uppgift 1(a): UML-diagram för geometriska figurer (`diagram/uppgift1.eps`)

Lösning

Se bifogad källkod, klasserna finns i paketen `se.hig.oodp.lab.model.simplefigure` och `se.hig.oodp.lab.model.figure`.

(c)

Frågeställning

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!

Lösning

Klassen `Vertex2D` är en del av figurklasserna. Figurklasserna består av minst en `Vertex2D` som utgör figurens mittpunkt. Figurklasserna har en varsin lista där ett godtyckligt antal `Vertex2D`-objekt kan lagras, antalet beror på vilken figur subklassen representerar.

2 Uppgift 2

(a)

Frågeställning

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!

Lösning

Figurerna beskrivs av ett visst antal punkter eller positioner som representeras av `Vertex2D`-instanser. Alla figurer består av minst ett `Vertex2D`-objekt som representerar figurens mittpunkt. Figurerna kan delas upp i två separata grenar i arvshierarkin, `Figure` och `SimpleFigure`. Skillnaden mellan Figure och SimpleFigure är hur många punkter de lagrar för att beskriva figuren de representerar. Figurer som kan beskrivas med bara en `Vertex2D`-instans, figurens position, ärver från `SimpleFigure`; punkt, cirkel och ellips. Figurer som består av fler `Vertex2D`-objekt, ärver från `Figure`. Klassen `Figure` lagrar sina punkter i en lista för enkel åtkomst. Det möjliggör också skapandet av figurer med ett godtyckligt antal punkter.

(b)

Frågeställning

Ska superklassen vara en abstrakt klass eller inte? Diskutera vad som talar emot och vad som talar för!

Lösning

Klasserna `SimpleFigure` och `Figure` är både abstrakta av den anledningen att vi sannolikt inte kommer att behöva instantiera någon odefinerad figur. Möjligheten att skapa t.ex. en polygon med ett godtyckligt antal punkter från figurklasserna talar emot att de är abstrakta, det skulle vara möjligt att skapa odefinerade figur-objekt och efter att de skapats namnge dem till rätt figur (punkt, kvadrat, triangel, etc..) efter det antal punkter figuren instantierats med och dessa punkters position. I det här fallet valde jag ändå att göra figurklasserna abstrakta och låta subklasserna punkt, kvadrat, triangel, etc.. ärva från superklasserna och utgöra faktiska, instantierbara objekt. De olika figurerna har fält och metoder som gör dem unika, en cirkel har t.ex. en radie medan en ellips har en bredd och höjd.

(c)

Frågeställning

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

Lösning

UML-klassdiagrammet och arvshierarkin återfinns i Figur 3, samt bifogad fil.

(d)

Frågeställning

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!

Lösning

Se bifogad kod för implementering.

3 Uppgift 3

(a)

Frågeställning

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

Lösning

Alla figurer behöver någon mekanism för att flytta på sig och det kan tänkas att alla figurer skulle kunna använda sig av sammma logik för att utföra förflyttningen. Exempel på hur det skulle kunna implementeras är med en `move()`-metod i en gemensam superklass som itererar genom en lista av `Vertex2D`-punkter och flyttar varje punkt för sig. Alla figurer kan uttryckas med en lista av punkter och kan således ärva från en gemensam superklass.

Ett annat exempel är rotering, som inte är applicerbart för vissa figurer, t.ex. cirkeln. Det kan vara konceptuellt möjligt att rotera en perfekt cirkel, men i det här fallet så skulle cirkelns

interna läge inte förändras, och den grafiska representationen skulle inte heller komma att ändras. Alternativet till ett interface `Rotatable` i fallet med cirkeln är att `rotate()`-metoden i cirkel-klassen blir ersatt med en tom metod genom *override*.

(b)

Frågeställning

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!

Lösning

Jag bestämde mig för att lägga till interfacen `Movable`, `Rotatable` och `Scalable`. Detta med motiveringen att det är operationer som efterfrågas i användningsfallsdiagrammet och som de olika figurerna kan komma att behöva utföra på olika vis, beroende på figur. Avvägning gjordes mot att göra en enklare design som förlitar sig mer på arv och *override* av superklassers metoder. Det uppdaterade klassdiagrammet återfinns i Figur 4.

(c)

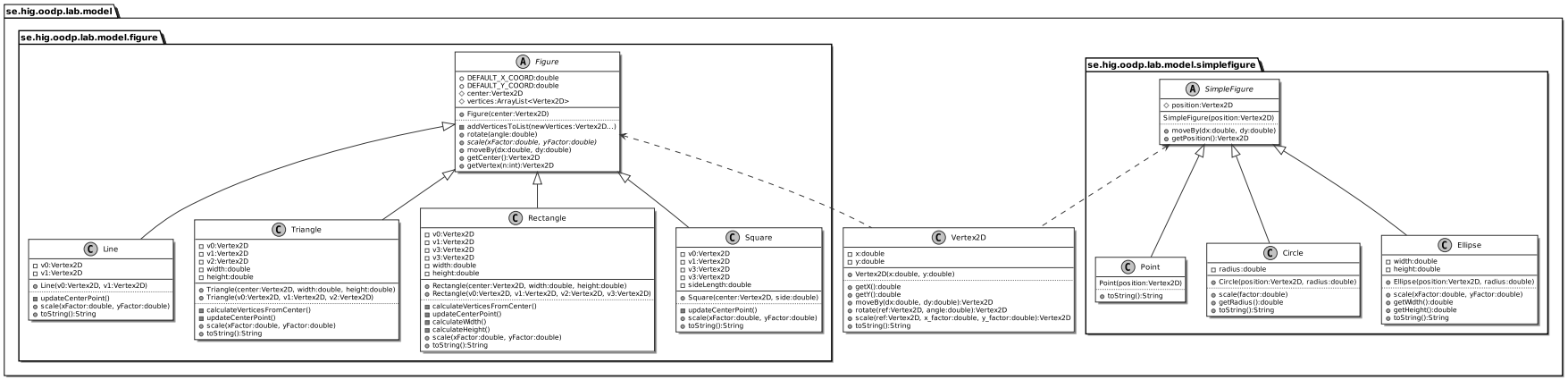
Frågeställning

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

Lösning

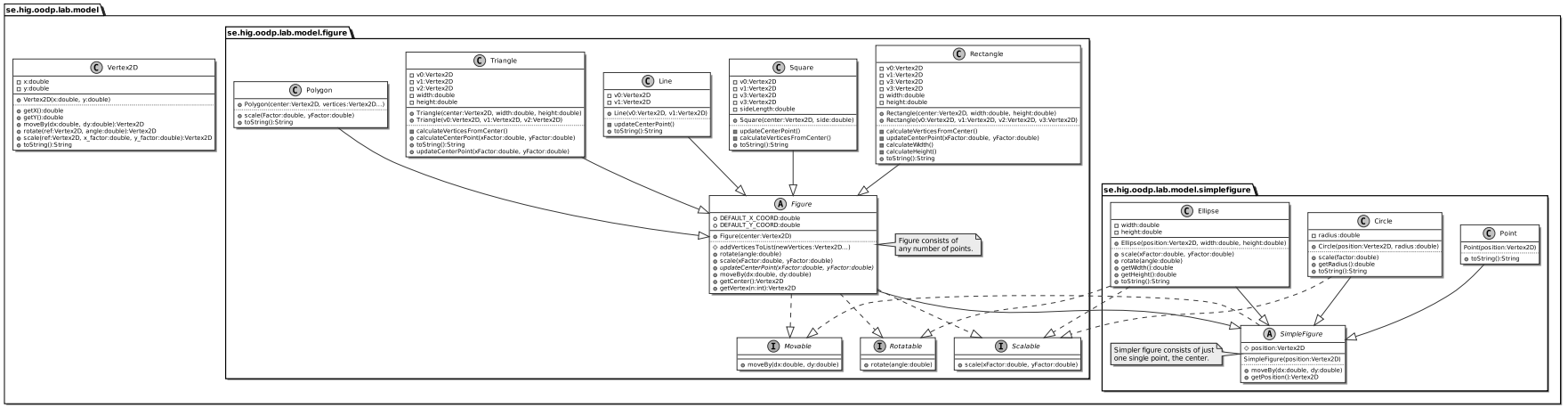
Se bifogad kod för implementering.

Jonas Sjöberg 860224
Högskolan i Gävle
tel12jsg@student.hig.se
https://github.com/jonasjberg



Figur 3: Uppgift 2(c): UML-diagram för geometriska figurer (`diagram/uppgift2.eps`)

Jonas Sjöberg 860224
Högskolan i Gävle
tel12jsg@student.hig.se
https://github.com/jonasjberg



Figur 4: Uppgift 3(c): UML-diagram för geometriska figurer (`diagram/uppgift3.eps`)