HÖGSKOLAN
I GÄVLE

AKADEMIN FÖR TEKNIK OCH MILJÖ

Avdelningen för industriell utveckling, IT och samhällsbyggnad

Objektorienterad Design och

Programmering, 7.5HP, HT15

Laboration 1

av

Miran Batti

Innehåll

Inledning 3
Förutsättningar och krav 3
Resultat 3
Referenser 6

Inledning

Labben handlar om grundläggande idéer om objektorienterad design och programmering. Denna labb går ut på att skapa klasser till geometriska figurer. Den handlar om arv och interface.

Förutsättningar och krav

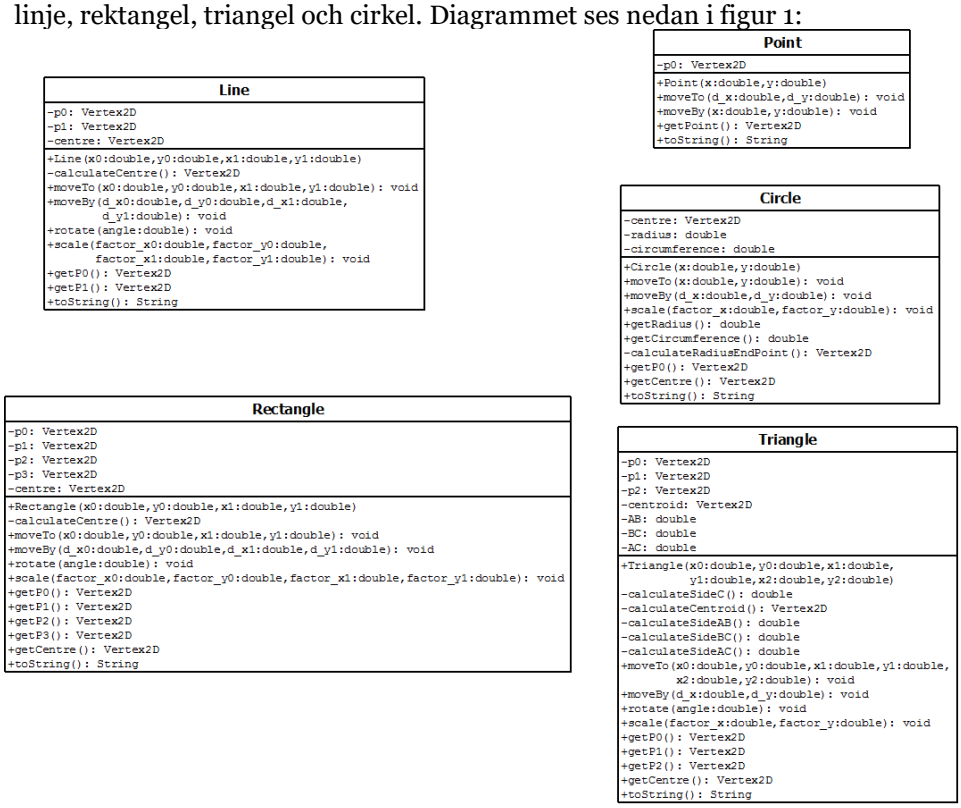
Det ska skapas en applikation där konkreta objekt modelleras. De objekt som ska modelleras är: punkt, linje, rektangel, triangel och cirkel. Dessutom ska uppgifter med frågor besvaras. Dessa uppgifter framgår i laboration instruktionerna[1].

Klasserna ska testas i JUnit[2]. De ska vara exekverbara och uppfylla kraven enligt labbinstruktionerna. Dessutom ska frågor i instruktionerna besvaras.

Till labbens förfogande finns en färdig klass, `Vertex2D`, som objekten ska utnyttja.

Resultat

1. A) Till uppgiften skapades ett UML[3] diagram som beskriver objekten punkt, linje, rektangel, triangel och cirkel. Diagrammet ses nedan i figur 1:



Figur 1. Ett UML-klassdiagram till programmets datamodeller

Objektens attribut skapades med avseende till figurernas antal hörn och figurernas centrum, och i cirkelns fall, en radie med en ändpunkt och ett centrum.

Cirkeln och triangeln har attribut för omkrets och sidor, dessa är möjligtvis överflödiga men det ansågs att de kan vara användbara till framtida laborationer.

Operationerna delas in i en ”Command” kategori och en ”Queries” kategori.

De flesta Command operationerna är metoder från `Vertex2D` eftersom figurerna som skapas är definierade av noderna som bildas från den klassen. Alla klasser förutom Point har någon typ av beräkningskommando där centrum eller omkrets eller sidor till figuren beräknas.

Queries operationerna är enkla ”getters” och ”toStrings” som används huvudsakligen för testandet av klasserna.

B) Implementeringen av klasserna ledde till ändringar och additioner till UML-klassdiagrammet. Det är värt att nämna att diagrammet från figur 1 är den ändrade versionen.

Varje figur har ett antal noder som skapas genom att skicka x och y koordinater till konstruktorn, x och y koordinaterna skickas till `Vertex2D` som skapar dessa noder(eller ”är” noderna). Figurerna har ett centrum som beräknas i en egen metod. Eftersom cirkelns punk är centrum så är det en radie och en ändpunkt som beräknas.

C) Figur klasserna har en stark ”har en” relation till klassen `Vertex2D`. Figurerna har ansvaret att skapa noderna som ska bilda figurerna.

Relationen ses i UML diagrammet endast från attributen. Operationerna och konstruktorn visar implicit att attributen tilldelas värden, men det finns ingen `Vertex2D` klass i diagrammet som pekar på de andra klasserna och visar att ”den här skapar ett `Vertex2D` objekt”.

2. A) Klasserna är 2D figurer. Därför skapas det en superklass, `Shapes2D`. Superklassen har attribut och operationer som är gemensamma till alla klasser.

För att få så många operationer och attribut som möjligt i superklassen så ingår punktklassen inte i `Shapes2D`. Sedan kan frågan om en ”punkt” är en figur också kunna ställas.

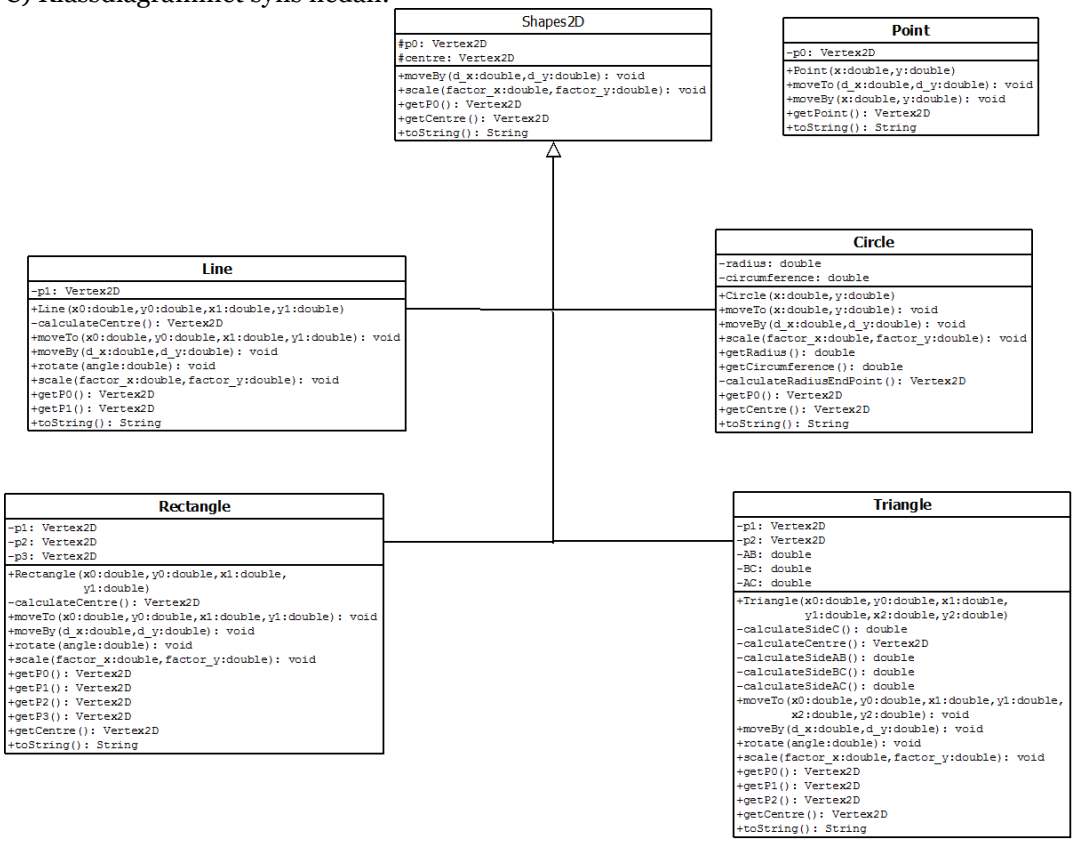
Operationen `moveTo` ingår inte `Shapes2D` eftersom varje klass ser olika ut när det gäller parameter till metoden.

`Shapes2D` har attributen `p0` och `centre` eftersom alla klasser har dessa attribut, dessutom ingår ”get” metoderna för dessa. Operationerna `moveBy` och `scale` är med, men inte `moveTo` och `rotate`. Det beror på att `moveTo` skiljer sig när det gäller parametrarna mellan klasserna, `rotate` ingår inte eftersom inte alla figurer är roterbara. En `toString` metod tillhör också `Shapes2D`.

B) Superklassen är abstrakt. Metoderna är tomma och abstrakta. Anledningen till det är att vi inte vill ha en figur; vi vill ha en rektangel eller triangel, osv. Implementeringen i de flesta metoderna skiljer sig mellan alla klasser, vilket betyder att ha abstrakta metoder är en fördel i den här situationen. Alltså, alla subklasser implementerar egna versioner av metoderna, därför är det bäst att ha en abstrakt klass.

Nackdelen med en abstrakt klass är att den inte förenklar vår kod. Meningen med arv är ett minska antal kod och att göra klasserna bättre förstådda.

C) Klassdiagrammet syns nedan:



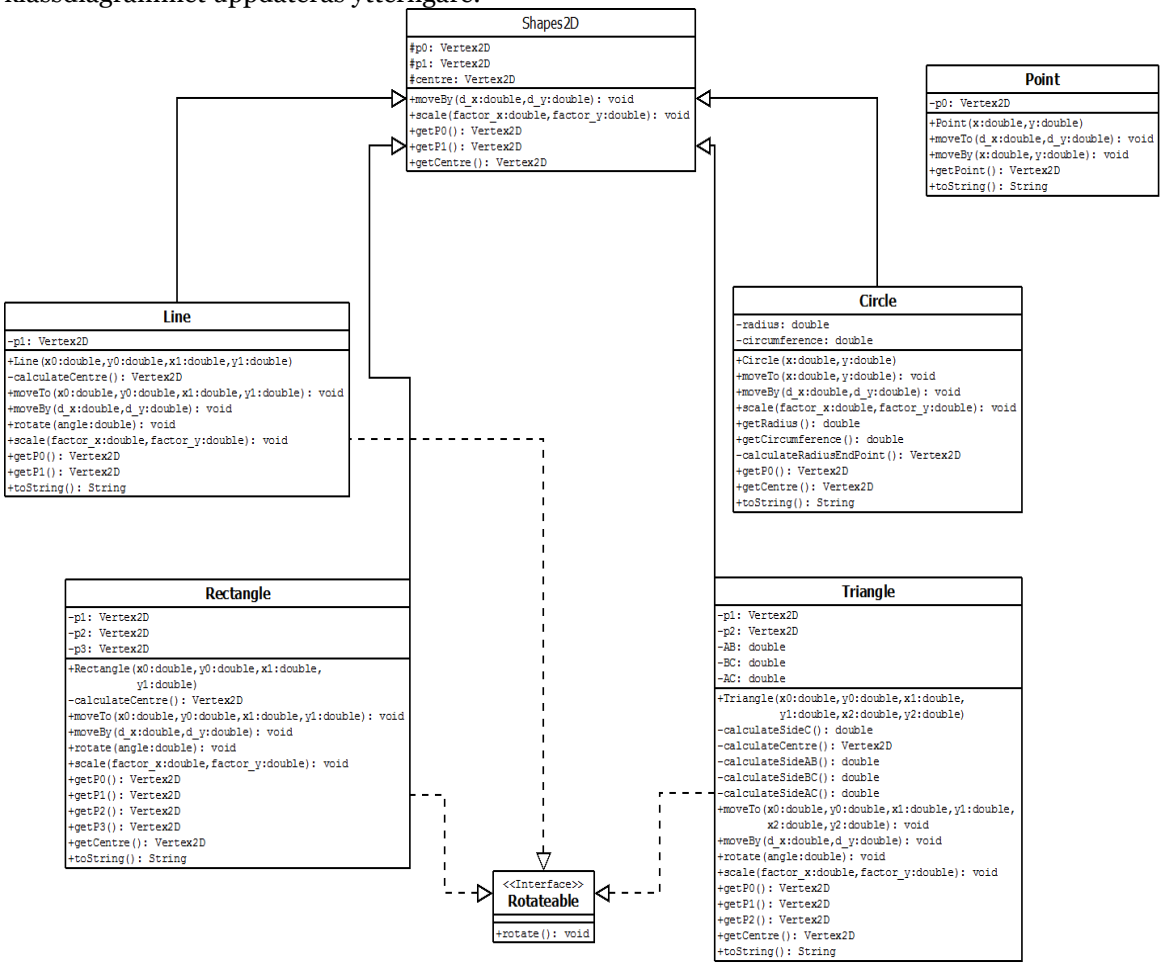
Figur 2. Uppdaterat UML-klassdiagram

D) Koden i superklassen kan ses i den bifogade .zip filen. `Shapes2D` är en abstrakt superklass med abstrakta metoder. Vilket betyder att metoderna i superklassen måste implementeras i subklasserna. Metoder och attribut som redan finns i superklassen är borttagna från subklasserna.

3. A) `Line`, `Rectangle` och `Triangle` har metoden `rotate` gemensamt men finns inte i superklassen. Det beror på att `Circle` klassen inte är roterbar.

B) Vissa objekt är roterbara. Därför finns interfacet `Rotateable`. Nu kan

klassdiagrammet uppdateras ytterligare.



Figur 3. UML-klassdiagrammet är figur 2 med ett interface.

UML-klassdiagrammet visar att klasserna `Line`, `Rectangle` och `Triangle` är roterbara.

C) Interfacet har endast en metod utan implementering. I slutet av klassnamnen till de roterbara klasserna har nu nyckelorden `implements` och `Rotateable`. Vilket innebär att klasserna måste innehålla metoden `rotate`.

Referenser

[1] ”Objektorienterad design och programmering: Instruktionerna till laborationerna”. Jenke, Peter. [Online]. Tillgänglig: https://lms.hig.se/bbcswebdav/pid-343196-dt-content-rid-1408191_1/courses/HT15_18402/oodp_lab_instruktioner_ht15v4.pdf
[Använd 2015-09-20]

[2] ”JUnit”. *JUnit*. Tillgänglig:
http://junit.org/
[Använd 2015-09-21]

[3] ”Unified Modeling Language Resource Page”. UML. Tillgänglig:
http://www.uml.org/
[Använd 2015-09-21]