DVG303
Objektorienterad design och programmering
Laboration 2

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

Datum: 2015-10-27
Kursansvarig lärare: Peter Jenke

Sammanfattning

Laborationsrapport för *DVG303 – Objektorienterad design och programmering.*

Innehåll

Introduktion **3**
Övergripande beskrivning . . . 3
Uppgifter . . . 3
Arbetsmetod . . . 3
Källkod . . . 3

1 Uppgift 1 **4**
(a) . . . 4
(b) . . . 4
(c) . . . 4

2 Uppgift 2 **4**
(a) . . . 4
(b) . . . 5
(c) . . . 5

3 Uppgift 3 **5**
(a) . . . 5
(b) . . . 6
(c) . . . 6

Figurer

1 Uppdaterat UML-diagram, sida 1 av 2. . . . 8
2 Uppdaterat UML-diagram, sida 2 av 2. . . . 9
3 Sekvensdiagram för test av alla figurer. . . . 10
4 Sekvensdiagram för `Point`. . . . 11
5 Sekvensdiagram för `Line`. . . . 12
6 Sekvensdiagram för `Rectangle`. . . . 13
7 Objektdiagram för test av alla figurer. . . . 14
8 Objektdiagram för `Point`. . . . 15
9 Objektdiagram för `Line`. . . . 15
10 Objektdiagram för `Rectangle`. . . . 16

Introduktion

Övergripande beskrivning

Det här är den andra 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.

Uppgifter

Instruktionerna är till viss del kopierade från källfilen `oodp_lab_instruktioner_ht15v4.pdf`.

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 LaTeX 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`.

- Analys av program under exekvering sker med hjälp av `JIVE` – Java Interactive Visualization Environment. [1]

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 annat exempel är `Meld`.

Källkod

Källkod till programmet och rapporten finns att hämta på `https://github.com/jonasjberg/DVG303_lab2`. Hämta hem repon genom att exekvera följande från kommandoraden:

```
> $ git clone git@github.com:jonasjberg/DVG303_lab2.git
```

[1] `http://www.cse.buffalo.edu/jive/`

1 Uppgift 1

(a)

Frågeställning

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

Lösning

Ett nytt paket `component` (`se.hig.oodp.lab.model.component`) innehåller de tre interface som ska användas för att manipulera alla typer av figurer; `Movable`, `Rotatable` och `Scalable`. Dessa tre interface skapades i den förra laborationen och har inte ändrats.

(b)

Frågeställning

Modellera alla interfacen som behövs utöver interfacet från Uppgift **??** (beskrivning och UML-diagram; uppdatera klassdiagrammet).

Lösning

Uppdaterade UML-diagram återfinns i Figur 1 samt Figur 2.

(c)

Frågeställning

Skriv interfacen från Uppgift (b) i Java.

Lösning

Se bifogad källkod.

2 Uppgift 2

(a)

Frågeställning

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

Lösning

Den enklaste figur-klassen `Figure` implementerar interfacet `Movable`. Underklasserna `Point`, `Circle` och `Ellipse` implementerar sedan `Scalable` och `Rotatable` beroende på vad som är lämpligt för respektive klass.

- `Point` behöver varken roteras eller skalas och implementerar indirekt `Movable` genom superklassen `Figure`.
- `Circle` kan skalas men behöver inte roteras och implementerar följdaktligen `Movable` (ärvs från superklassen) och `Scalable`.

- `Ellipse` kan både skalas och roteras och implementerar `Movable` (ärvs från superklassen), `Rotatable` och `Scalable`.

Jag inser nu att arvshierarkin är onödigt komplex, det finns ingen större vits med att ha två separata arvsträd av figurtyper, `SimpleFigure` och `Figure`. Men vid det här laget finns inte tid för någon omstrukturering.

(b)

Frågeställning

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').

Lösning

Se bifogad källkod. Lösningen använder `StringBuilder` för att konkatenera resultatet från superklassens `toString`-metod med klassens egna data.

(c)

Frågeställning

Diskutera: Varför används typer som `List<FigureType>` i FigureHandler? Kan man inte använda bara List? Eller en array?

Lösning

Användningen av en `List` framför en vanlig array motiveras med att en array inte kan ändra storlek dynamiskt. Storleken definieras vid skapande av arrayen och är därefter fixerad. En `List` har flera användbara funktioner som saknas för arrays. Genom att skriva något i stil med `List<Figure>` specifieras vilken typ av objekt listan kan innehålla. Det fungerar som en slags felkontroll där kompilatorn ger varningar om ett inkompatibelt objekt stoppas in.

3 Uppgift 3

(a)

Frågeställning

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!

Lösning

Programmet analyserades med `JIVE` – Java Interactive Visualization Environment. [2]

`JIVE` installeras som en plugin i `Eclipse Mars` version 4.5.0 och körs i ett interaktivt debug-"perspective". Programmet kan generera olika modeller av ett exekverande program; [3]

1. Contour Model.

[2] `http://www.cse.buffalo.edu/jive/`

[3] `http://www.cse.buffalo.edu/jive/presentations/cse505-10-12-01.pdf`

2. Object Diagram.

3. Sequence Model.

4. Sequence Diagram.

5. Event Log.

En särskild klass vid namn `FigureHandlerTest` användes vid körning av `JIVE`, som presenterar resultatet på olika sätt. Jag valde att exportera som `csv`-textfil samt `png`-bild.

Sekvensdiagram för programmet `FigureHandlerTest` återfinns i Figur 4, Figur 5, Figur 6 och Figur 3. Resultatet av testerna är finns också bifogade i katalogen `diagram`.

(b)

Frågeställning

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

Lösning

Se bifogad källkod. Lösningen använder `StringBuilder` för att konkatenera resultatet från superklassens `toString`-metod med klassens egna data.

(c)

Frågeställning

Diskutera: Varför används typer som `List<FigureType>` i FigureHandler? Kan man inte använda bara List? Eller en array?

Lösning

Användningen av en `List` framför en vanlig array motiveras med att en array inte kan ändra storlek dynamiskt. Storleken definieras vid skapande av arrayen och är därefter fixerad. För det här användningsområdet är en datastruktur med flexibla metoder för att "sätta in" och "ta ut" enskilda element att föredra. En `List` har flera användbara funktioner som man ofta måste skriva själv vid användning av arrays, ofta relaterade till sökning och åtkomst.

Java version 1.5 introducerade "typparametrisering". Det löser ett återkommande problem med listor och objekts typer. För en lista som klarar av att lagra objekt av generell typ förlorar objekten sin specifika typinformation då de stoppas in i listan. Då objektet ska plockas ut ur listan måste de "castas" för att återfå sin ursprungliga typ. Ofta behövs särskilda lösningar för att lagra objekt av olika typer, typparametriseringen erbjuder en generell lösning genom att direkt i språket definera listor som lagrar en viss typ av objekt. Genom att skriva något i stil med `List<Figure>` specifieras vilken typ av objekt listan kan innehålla. Det fungerar även som en slags felkontroll där kompilatorn ger varningar om ett inkompatibelt objekt stoppas in.

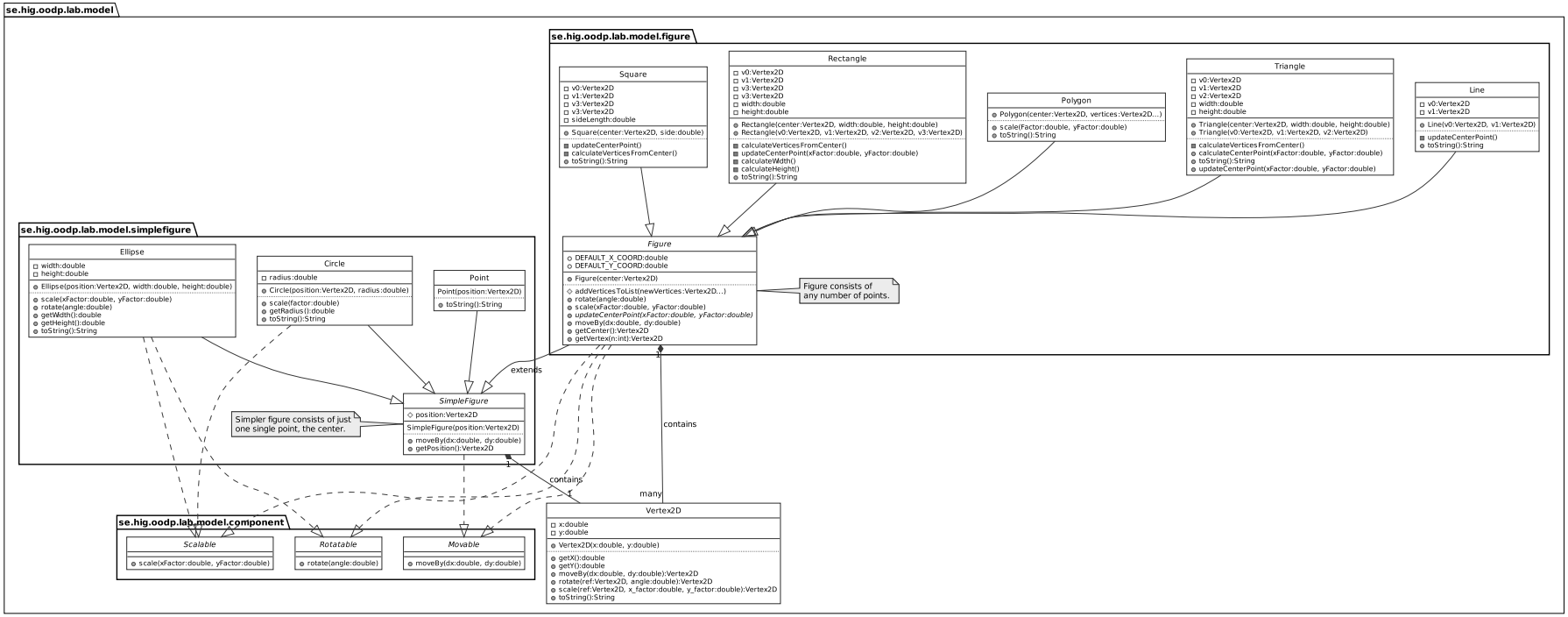
```java
/* Elimination of casts.
 * The following code snippet without generics requires casting:
 */
    List list = new ArrayList();
    list.add("hello");
    String s = (String) list.get(0);

/* When re-written to use generics, the code does not require casting:
 */
    List<String> list = new ArrayList<String>();
    list.add("hello");
    String s = list.get(0);   // no cast
```

Java SE Tutorials – Learning the Java Language > Generics [4]

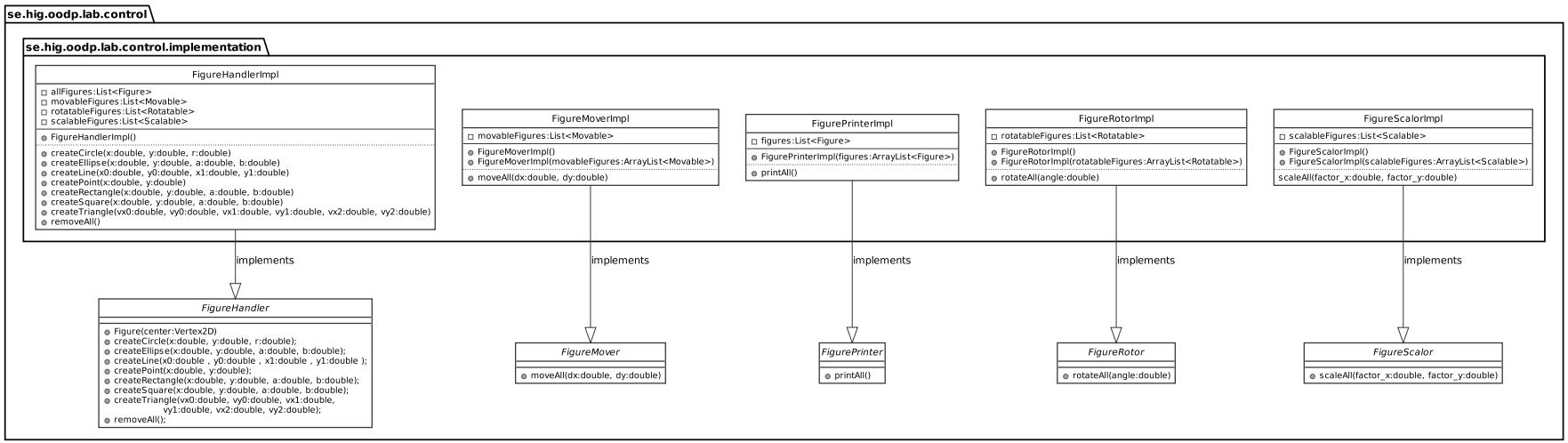
[4] https://docs.oracle.com/javase/tutorial/java/generics/why.html

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

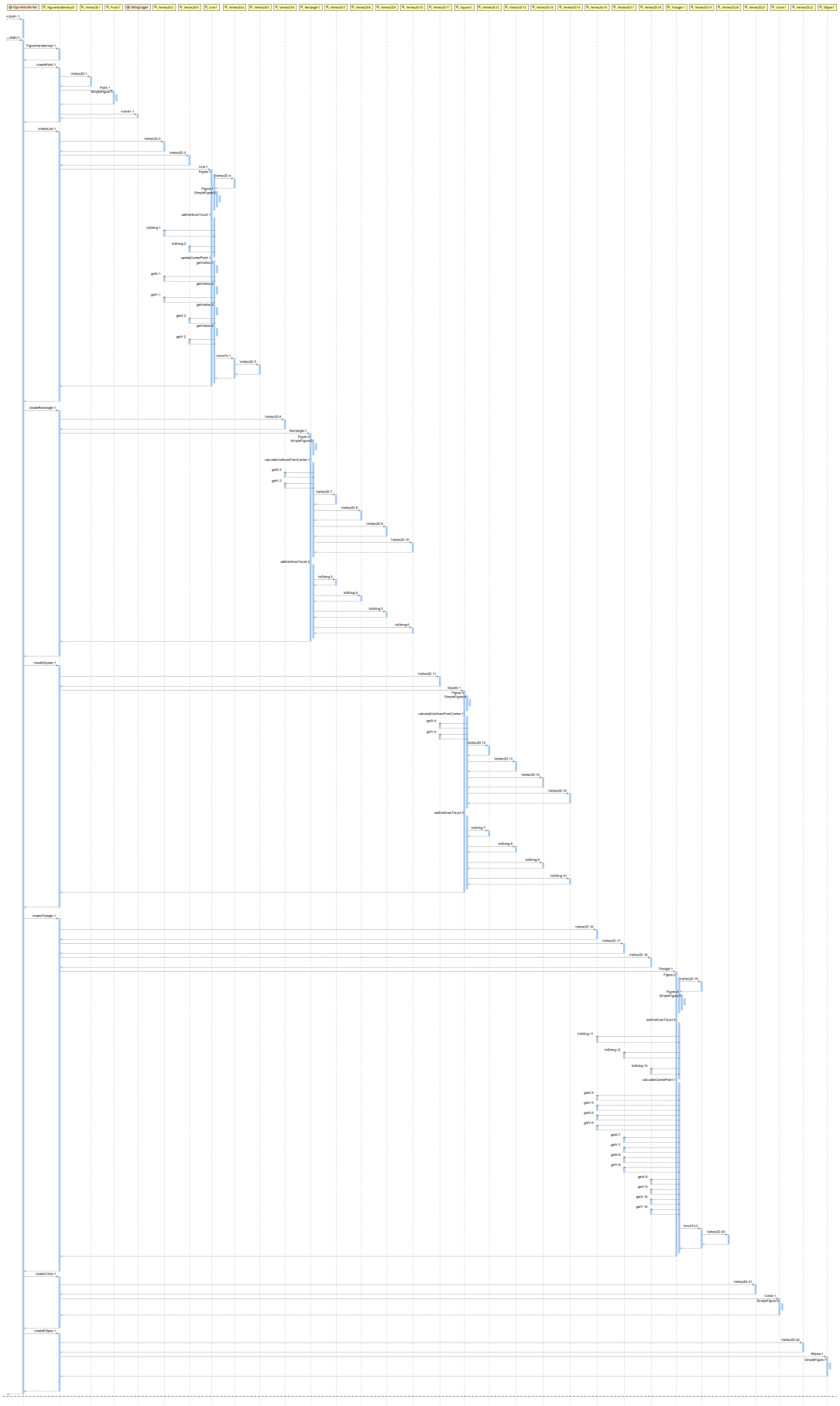


Figur 1: Uppdaterat UML-diagram, sida 1 av 2.

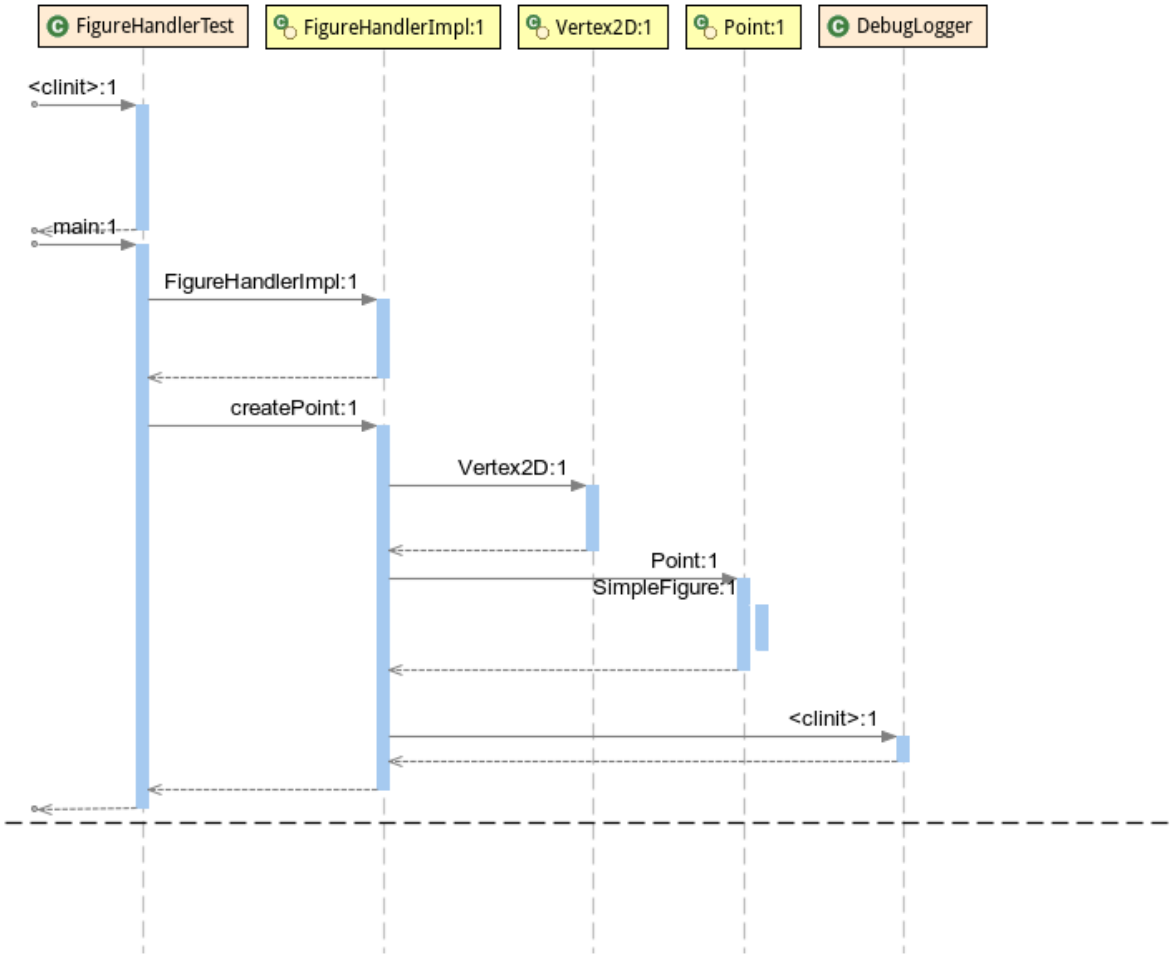
Jonas Sjöberg 860224
Högskolan i Gävle
tel12jsg@student.hig.se
https://github.com/jonasjberg



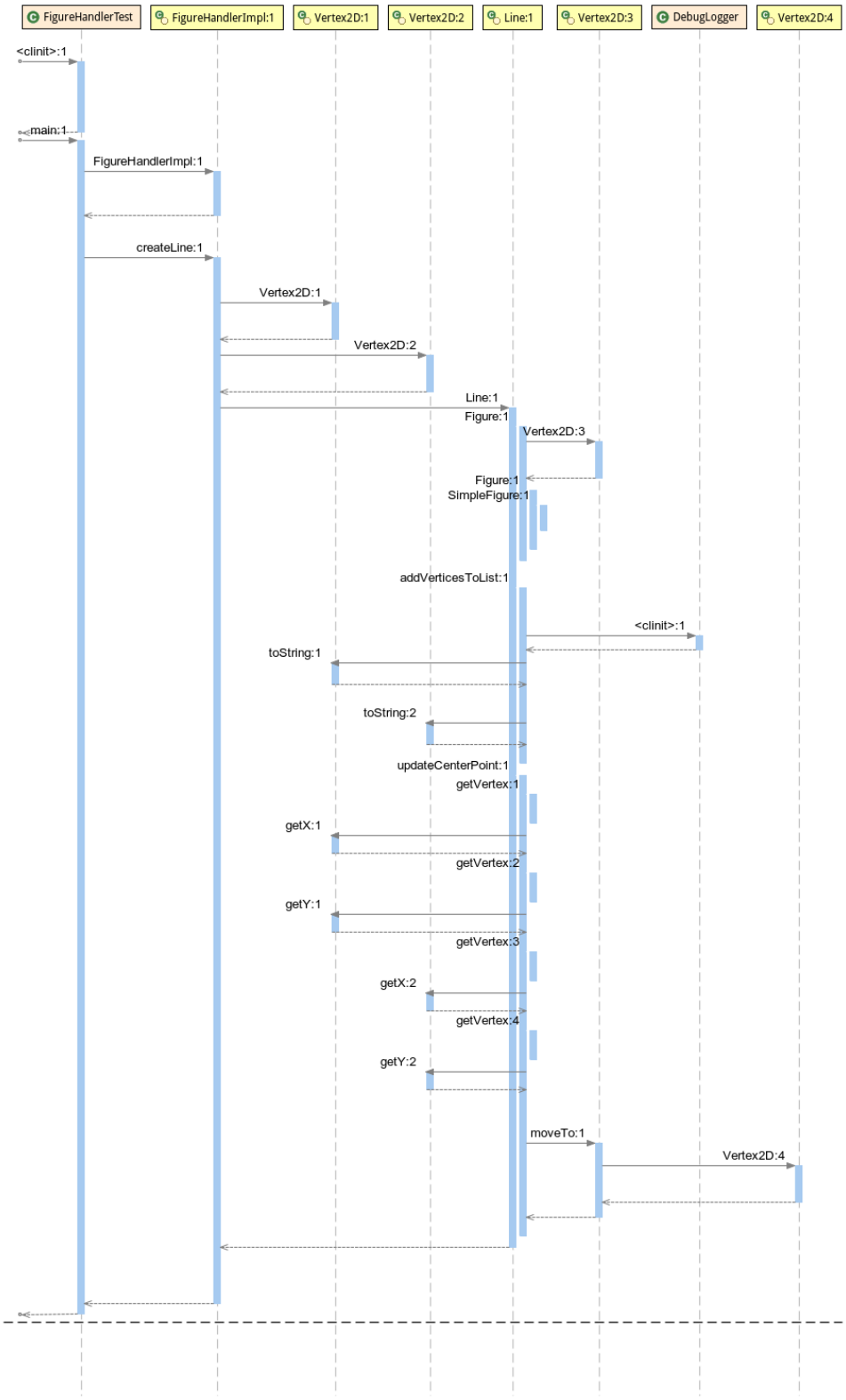
Figur 2: Uppdaterat UML-diagram, sida 2 av 2.



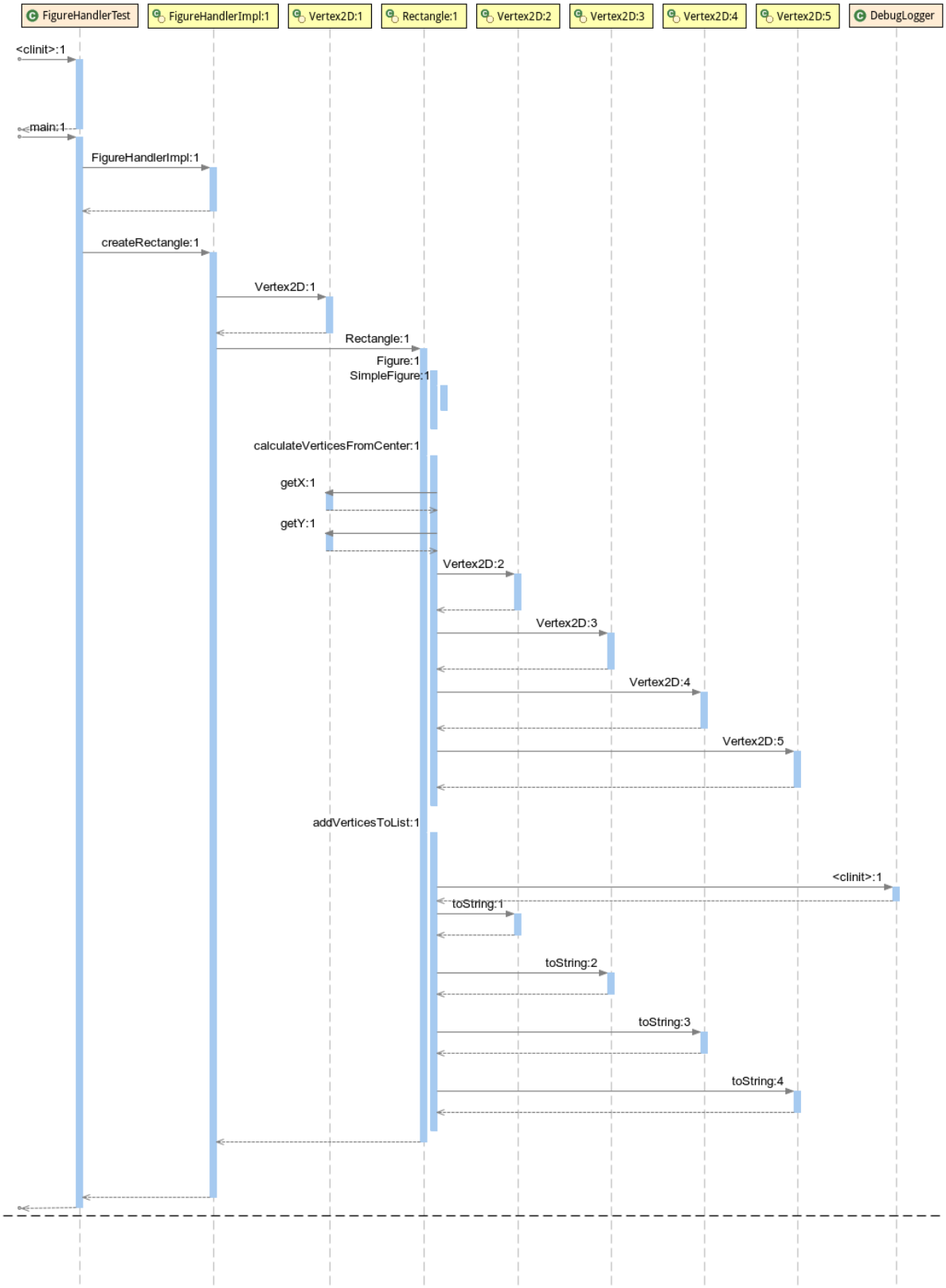
Figur 3: Sekvensdiagram för test av alla figurer.



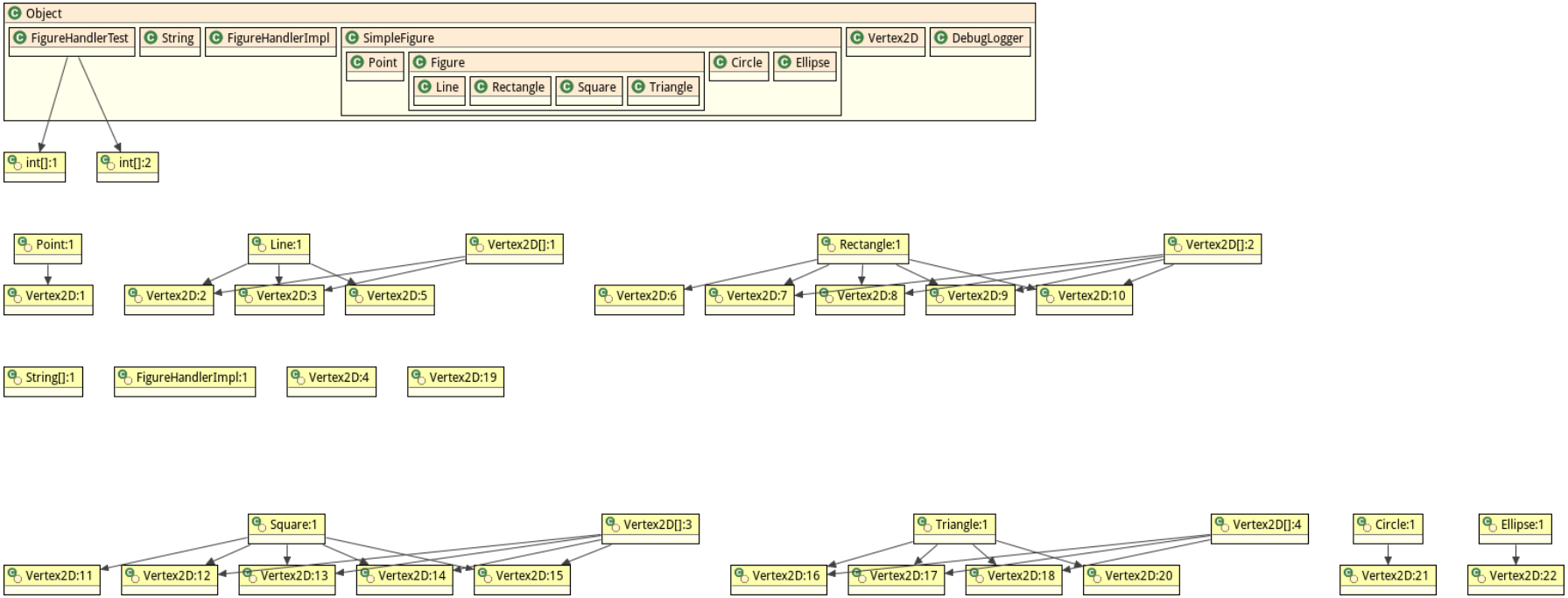
Figur 4: Sekvensdiagram för `Point`.



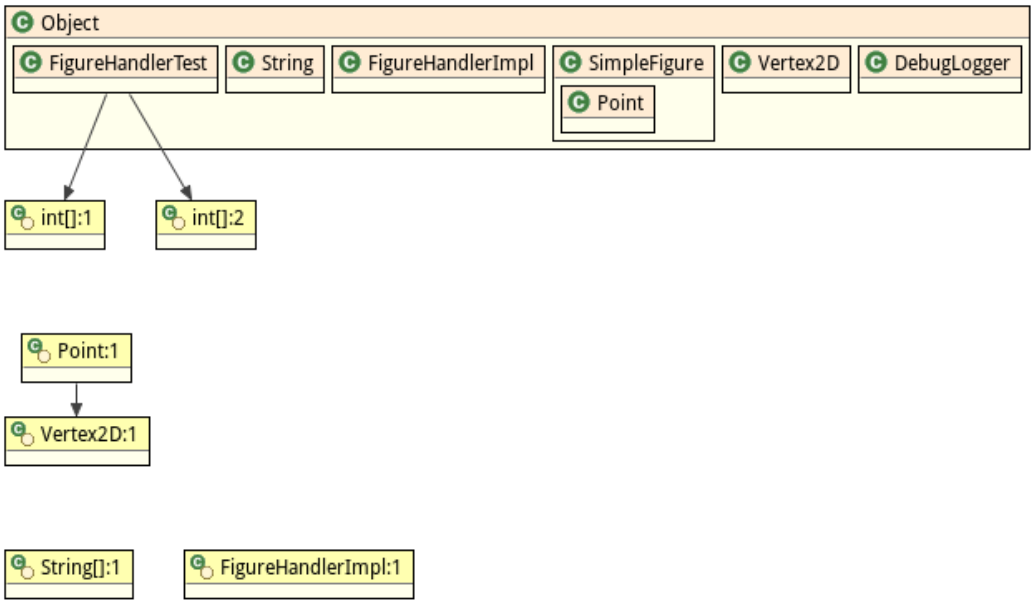
Figur 5: Sekvensdiagram för `Line`.



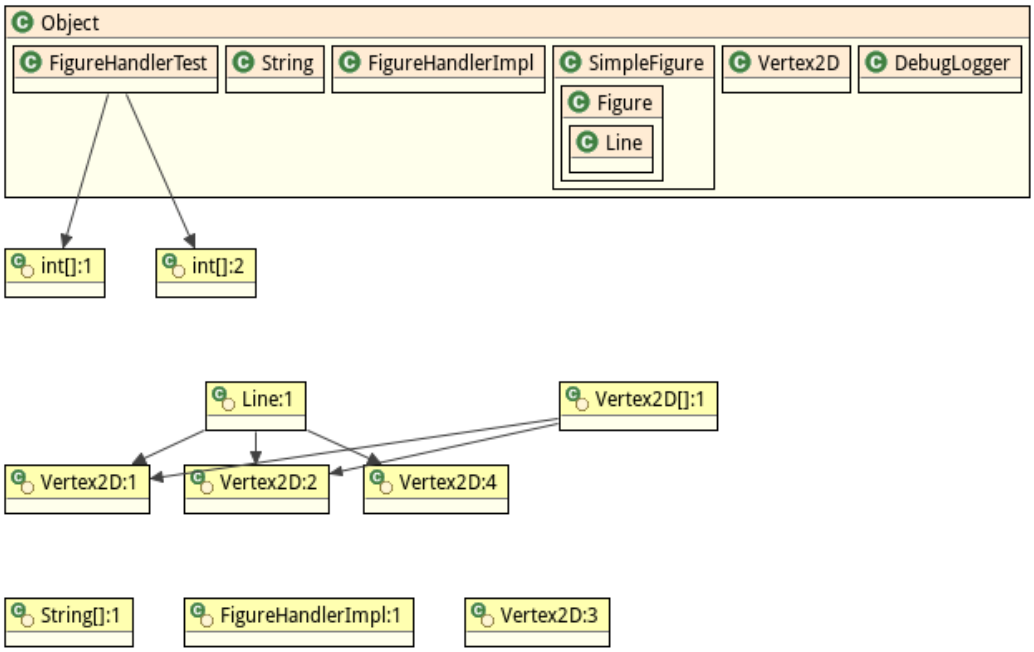
Figur 6: Sekvensdiagram för `Rectangle`.



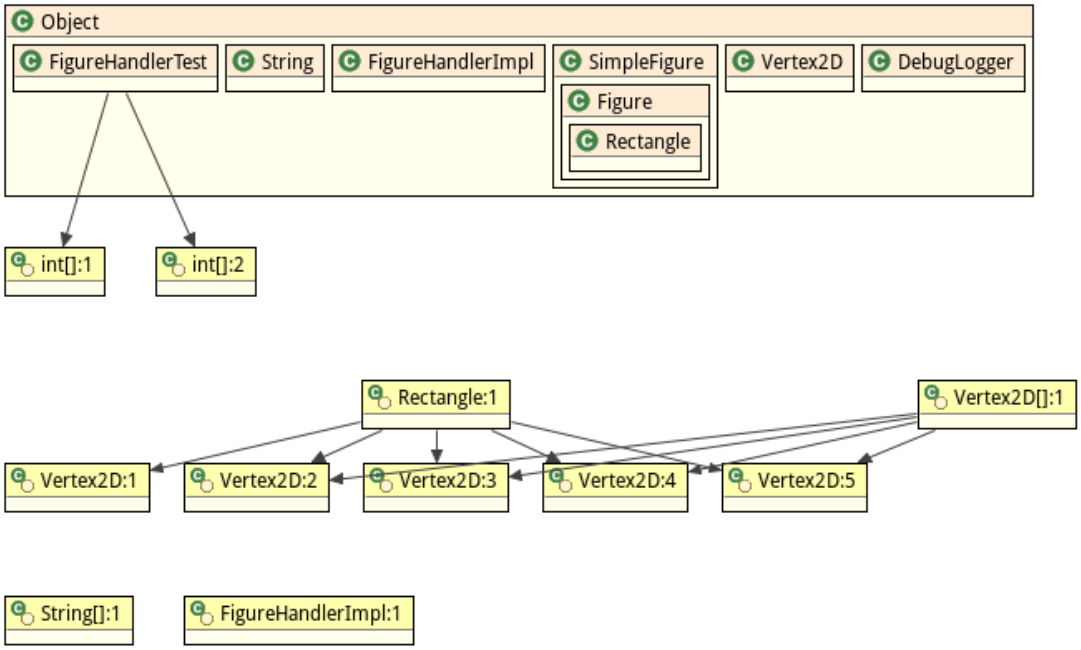
Figur 7: Objektdiagram för test av alla figurer.



Figur 8: Objektdiagram för `Point`.



Figur 9: Objektdiagram för `Line`.



Figur 10: Objektdiagram för `Rectangle`.