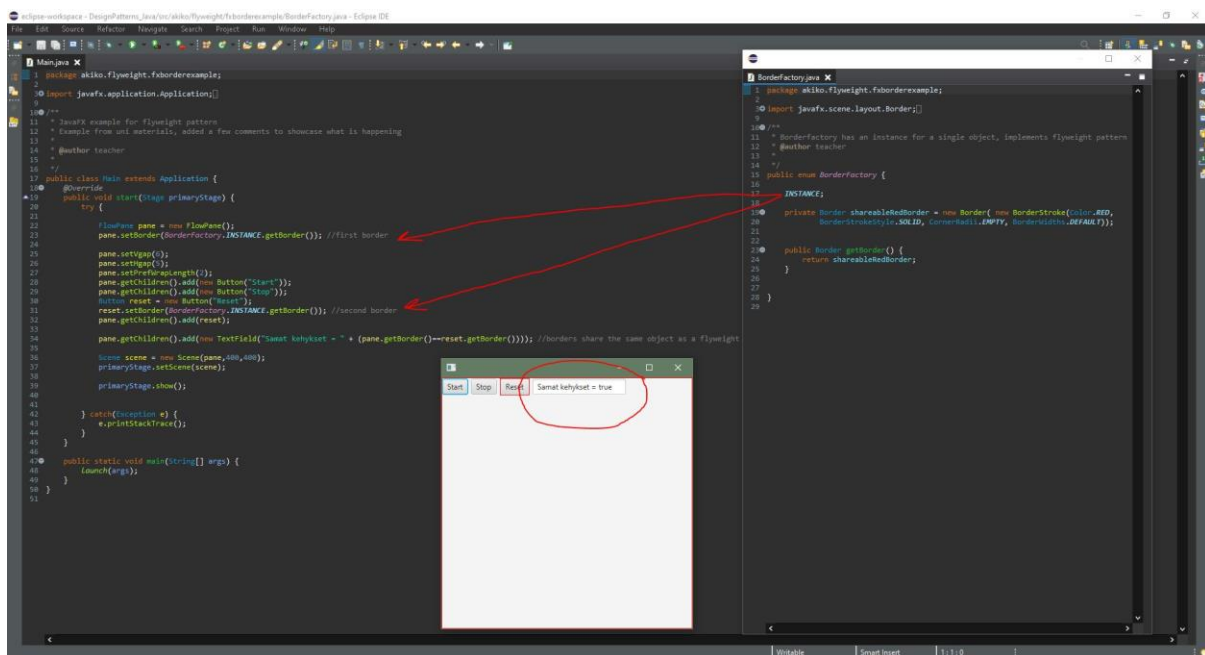


Flyweight lyhyesti

Flyweight suunnittelumallissa pyritään erottamaan samoja asioita sisältävät asiat yhdeksi olioksi niin että oliota voidaan uudelleen käyttää eikä sitä tarvitse luoda uudestaan. Olion ei tarvitse olla täysin samanlainen sillä sille yhteiskäyttöiset piirteet sisälletään Flyweight-olion sisäiseen tilaan, mutta ulkoiseen tilaan voidaan asettaa kontekstin mukaista dataa. Ajonaikaisesti voidaan käydä vaikkapa flyweight graafinen olio, ja asettaa se tiettyihin koordinaatteihin ulkoisten koordinaattitietojen perusteella. Seuraavissa kolmessa kappaleessa käyn läpi kolme eri esimerkkiä flyweight mallin käytöstä.

JavaFX elementin rajat flyweight mallin mukaisesti

Tässä flyweight esimerkissä käytiin hakemassa tietyn tyyppiset rajat ikkunassa olevalle näkymälle itselleen, sekä reset-nappulalle. Esimerkki on haettu opettajan jakamasta materiaalista.



Kuva 1. Borderfactoryn yhteiskäyttö

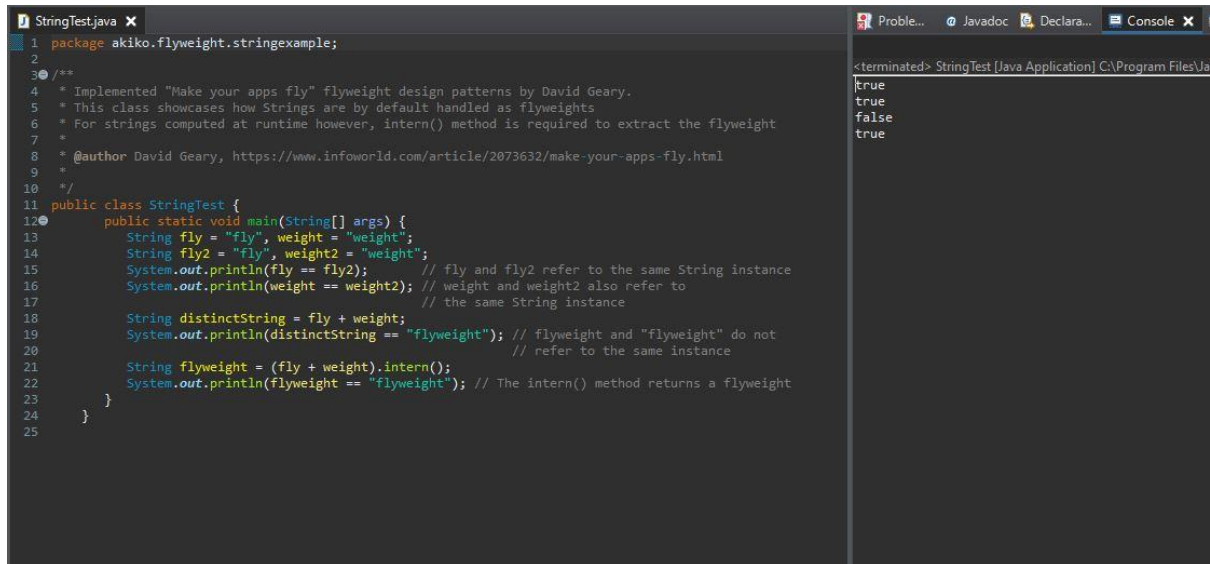
Kuvasta 1 voi nähdä, että samaa instanssia käytetään sekä näkymälle että nappulalle. Näennäisesti resursseja säästyy koska oliota ei tarvitse luoda kahta kertaa.

String olio flyweightinä

Tämä esimerkki näyttää kuinka String oliot toimivat flyweight mallin mukaisesti aina normaalioloissa. Ne ovat immutable (eli muuttumattomia) ja niillä voi olla vain yksi arvo, jota ei voi muuttaa. Jos arvon vaihtaa, on kyseessä uusi olio. Saman sisällön sisältävä Stringit

ovat kuitenkin vertailtavasti samoja olioita, kuten kuvasta 2 käy ilmi. Esimerkki on peräisin David Gueryn artikkelista "Make your apps fly"

(<https://www.infoworld.com/article/2073632/make-your-apps-fly.html>).



```
1 package akiko.flyweight.stringexample;
2
3 /**
4  * Implemented "Make your apps fly" flyweight design patterns by David Geary.
5  * This class showcases how Strings are by default handled as flyweights
6  * For strings computed at runtime however, intern() method is required to extract the flyweight
7  *
8  * @author David Geary, https://www.infoworld.com/article/2073632/make-your-apps-fly.html
9  *
10 */
11 public class StringTest {
12     public static void main(String[] args) {
13         String fly = "fly", weight = "weight";
14         String fly2 = "fly", weight2 = "weight";
15         System.out.println(fly == fly2); // fly and fly2 refer to the same String instance
16         System.out.println(weight == weight2); // weight and weight2 also refer to
17                                                // the same String instance
18         String distinctString = fly + weight;
19         System.out.println(distinctString == "flyweight"); // flyweight and "flyweight" do not
20                                                            // refer to the same instance
21         String flyweight = (fly + weight).intern();
22         System.out.println(flyweight == "flyweight"); // The intern() method returns a flyweight
23     }
24 }
25
```

Console output:

```
<terminated> StringTest [Java Application] C:\Program Files\Ja
true
true
false
true
```

Kuva 2. String flyweightinä

Poikkeus tähän käyttäytymiseen ovat kuitenkin ajonaikaiset merkkijonot, joista pitää erityisesti kutsua flyweight esitys, jotta vertailu samanlaiseen merkkijonoon antaisi vastaavuutta osoittavan tuloksen. Tämä on nähtävissä kuvan 2 intern() metodin käytöllä.

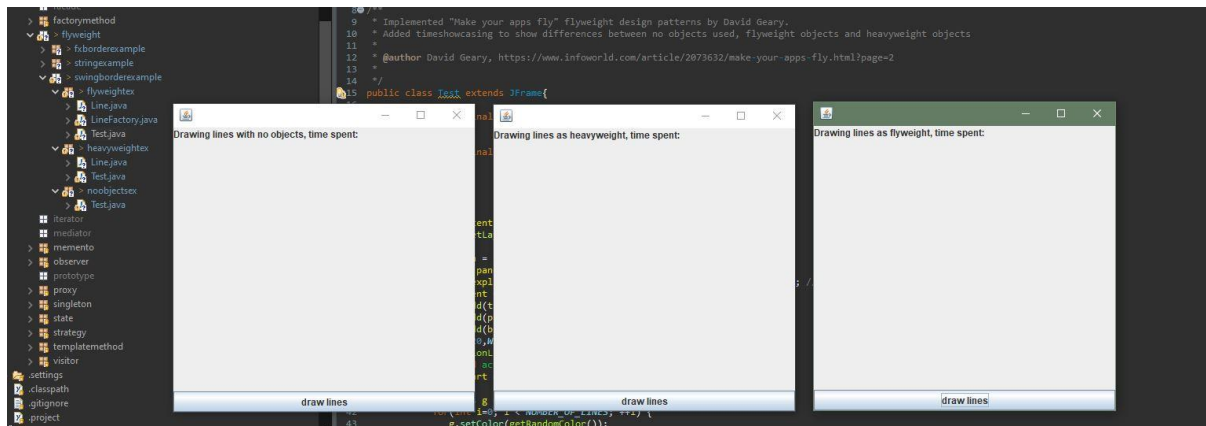
Swing kirjaston avulla piirtelyn vertailua heavyweight, flyweight ja ei oliota toteutuksissa

David Gueryn artikkelissa "Make your apps fly"

(<https://www.infoworld.com/article/2073632/make-your-apps-fly.html?page=2>) oli esimerkin omaisesti näytetty, kuinka 10000 viivaa piirretään eri keinoilla. Yksi näistä keinoista oli piirtää viivat ilman olioiden hyväksikäyttöä, toinen oli luoda jokaiselle viivalle oma olio ja kolmas tapa oli luoda uusi olio vain jokaiselle uniikille viivalle, jolla oli tietty väri.

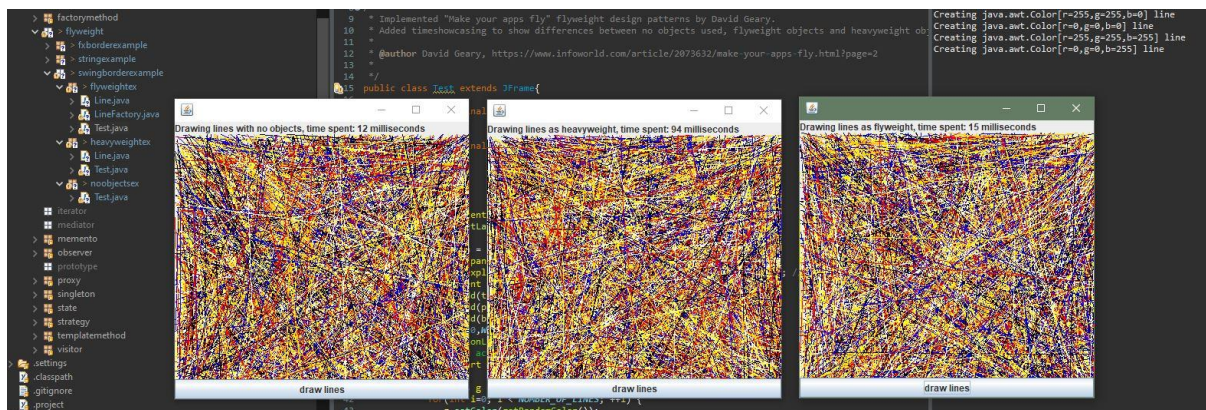
Tämä kolmas tapa on flyweight mallin mukainen ja se varmistaa, että olioita luodaan vain yhtä monta kuin sisäiseltä tilaltaan erilaisia viivoja on. Sisäinen tila kertoo tässä tapauksessa viivan värin. Koska värejä on 6 kappaletta, syntyy uusia olioita myös vain 6 kappaletta joita 10000 viivaa yhteiskäyttää.

Avaan saadut tulokset seuraavan kuvasarjan avulla.



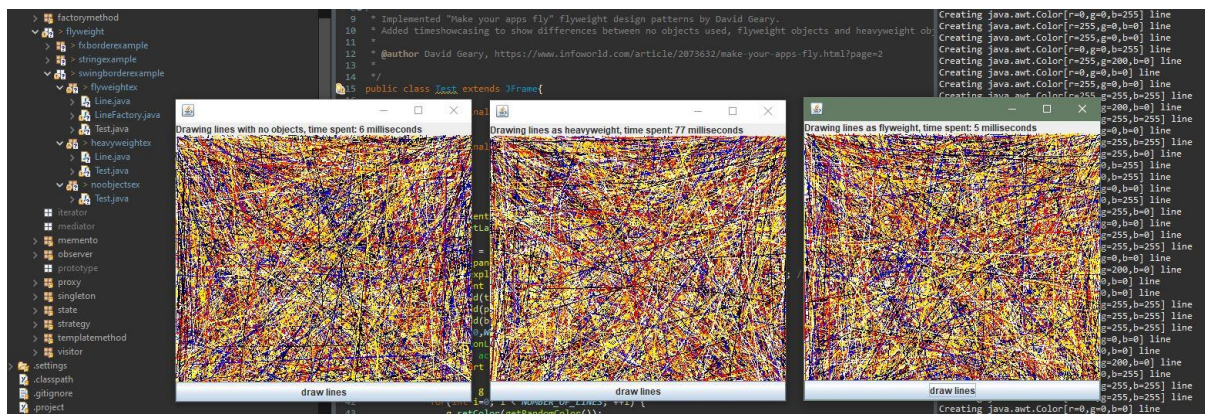
Kuva 3. Alkutilanne ennen viivojen piirtämistä

Kuten kuvasta 3 voi nähdä, lisäsin esimerkkikoodiin esityksen viivojen piirtämiseen kuluvasta ajasta millisekunteina. Ikkunat vasemmalta oikealle esittävät: olioton ratkaisu, raskas ratkaisu ja flyweight ratkaisu.



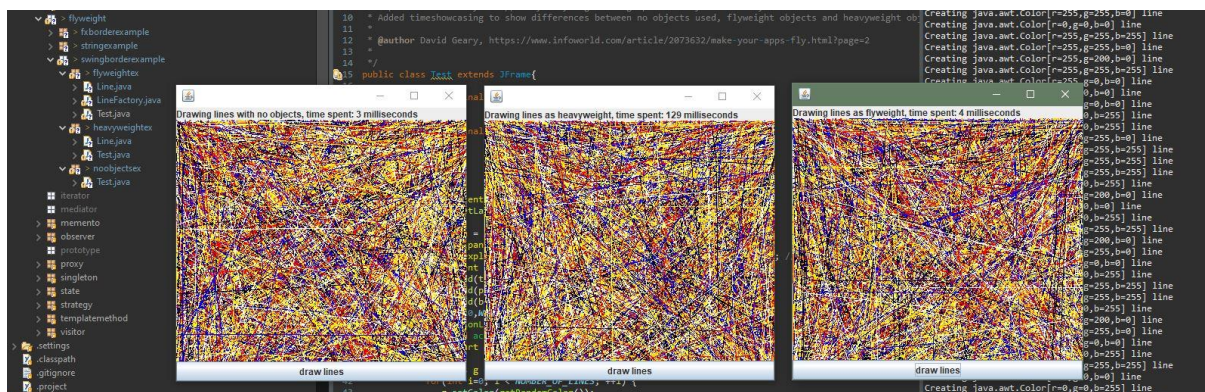
Kuva 4. Tilanne kun viivat on piirretty kerran jokaiselle ikkunalle

Kuvasta 4 voidaan nähdä, että olioton ratkaisu suoritti piirtämisen 12 millisekunnissa, raskas ratkaisu 94 millisekunnissa ja flyweight metodi 16 millisekunnissa. Voidaan helposti todeta että raskas ratkaisu, eli olioiden luonti jokaiselle viivalle, vie huomattavasti enemmän resursseja. Olioton ja flyweight ratkaisu ei tuota suuria eroja, mutta Java olio ohjelmointi kielenä toki kannustaisi käyttämään oliopohjaisia toteutuksia sulavuuden ja uudelleen käytettävyyden kannalta.



Kuva 5. Tilanne kun viivat on piirretty toisen kerran jokaiselle ikkunalle

Katsotaan miten tilannu muuttuu kun piirtäminen suoritetaan uudelleen. Kuvasta 5 nähdään, että olioton ratkaisu suoritti piirtämisen 6 millisekunnissa, raskas ratkaisu 77 millisekunnissa ja flyweight metodi 5 millisekunnissa. Ajan huomattava tippuminen flyweight metodissa käy järkeen sillä luout oliot ovat edelleen muistissa viime piirtelyn jäljiltä. Näin ollen uusia viiva olioita ei tarvitse luoda lainkaan. Olioton ratkaisu näyttää seurailevan samaa trendiä.



Kuva 6. Tilanne kun viivat on piirretty kolmannen kerran jokaiselle ikkunalle

Tehdään vielä yksi testi lisää ja ajetaan kaikissa ikkunoissa viivojen piirtäminen kolmannen kerran. Kuvasta 6 nähdään, että olioton tapa ja flyweight tapa vievät entistä vähemmän aikaa (3ms ja 4 ms). Raskaan metodin käyttämä aika jopa kasvaa (129ms). Tätä seuraavat piirtämiset näyttävät tasaantuvan oliottoman ja flyweight tavan osalta näihin lukemiin. Raskas tapa pysyy hyvin vaihtelevana ajan suhteen mutta kuitenkin aina hyvin pitkäkestoisena.

Lopuksi

Tuloksien perusteella voi päätellä, että olioiden yhteiskäyttöisyys parantaa tehokkuutta merkittävästi.