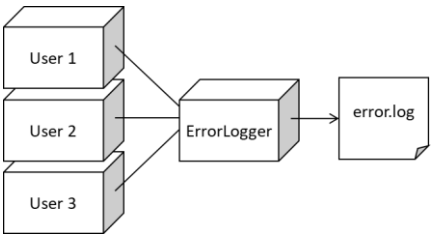
	Pattern: Singleton		AnPr	V 1.0
	Name	Klasse	Datum	

1 Problemstellung

Objekte werden immer durch einen Konstruktor erzeugt. Somit können wir durch mehrfachen Konstruktoraufruf auch mehrere Objekte erzeugen. Dies ist nicht immer gewünscht – bspw. wenn das Objekt den Zugriff auf eine zentrale Ressource kontrolliert. Wir wollen also folgende Funktionalität haben:

- Erzeuge das Objekt mit einem Aufruf, sofern es nicht schon existiert und gebe es zurück
- Gibt es das Objekt bereits, gebe es zurück

Diese Funktionalität soll aber gekapselt vorliegen, ohne die Existenz von irgendwelchen zentralen, globalen Variablen. Ein Beispiel wäre eine Klasse, welche Fehler loggt.

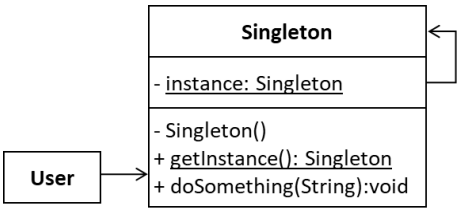


Das Objekt soll erst erzeugt werden, wenn es wirklich benötigt wird, aber dann nur einmal.

2 Problemlösung

Wir bedienen uns hier zwei Dingen:

- Privater Konstruktor
- Statische Variable für das Objekt

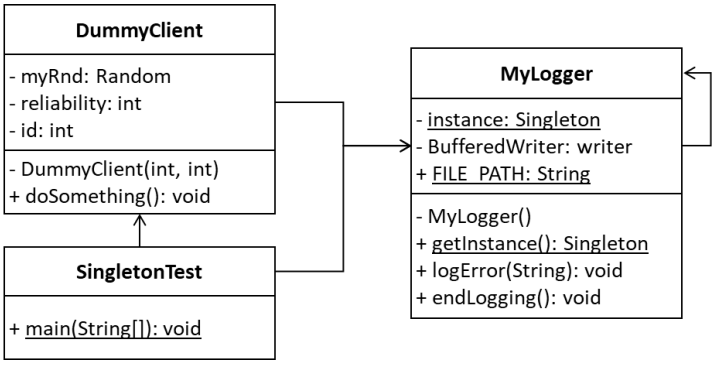


Durch das Setzen des Konstruktors auf „private“, kann er nicht mehr von außen erreicht werden. Der eigentliche Konstruktoraufruf wird in „`getInstance()`“ durchgeführt und das erzeugte Objekt wird in die statische Variable „`instance`“ geschrieben. Wenn dort allerdings bereits ein Wert vorhanden ist, wird dieser lediglich zurückgegeben:

```
public class MySingleton {
   private static MySingleton singleton = null;

   private MySingleton() {
      // do initialization
   }

   public static MySingleton getInstance() {
      if (singleton == null) {
         singleton = new MySingleton();
      }
      return singleton;
   }

   public synchronized void doSomething(String value) {
      System.out.println("Print something: " + value);
   }
}
```

3 Aufgabenstellung

Erzeugen Sie eine Klasse „`MyLogger`“ als Singleton, welche mit Hilfe eines `BufferedWriter` Objektes ein Logfile beschreibt. Sobald der Logger das erste Mal genutzt wird, soll das File offenbleiben, bis das gesamte Programm beendet wird.



Der „`DummyClient`“ soll einfach nur die Rechnung 100 / Zufallszahl rechnen und ausgeben. Wenn die Zufallszahl 0 ist, soll die Rechnung nicht erfolgen,

sondern ein Logeintrag erfolgen. Mit Hilfe der „`reliability`“ Variable kann die „Zuverlässigkeit“ eingestellt werden. Ist bspw. die `reliability = 2`, dann wird im Schnitt jede 2. Zahl 0 sein, bei `reliability = 10` jede zehnte:

```
public class DummyClient {
   private Random myRnd;
   private int reliability = 1;
   private int id = 0;

   public DummyClient(int reliability, int id) {
      myRnd = new Random();
      this.reliability = reliability;
      this.id = id;
   }

   public void doSomething() {
      int i = myRnd.nextInt(reliability);
      if (i == 0) {
         try {
              MyLogger logger = MyLogger.getInstance();
              logger.logError(" Client " + id + " division by zero");
         } catch (IOException e) {
              e.printStackTrace();
         }
         return;
      }
      System.out.println("Client " + id + ": Result: " + (100.0 / i));
      return;
   }
}
```

Die main-Methode erzeugt nun drei Clients und ruft in einer zufälligen Reihenfolge 100 mal „`doSomething()`“ auf.

```
public static void main(String[] args) {
   final int[] REALIBILITIES = {2, 10, 20};
   final int NO_OF_RUNS = 100;

   Random myRnd = new Random();
   DummyClient[] clients = new DummyClient[REALIBILITIES.length];

   for (int i = 0; i < REALIBILITIES.length; i++) {
      clients[i] = new DummyClient(REALIBILITIES[i], i);
   }

   for (int i = 0; i < NO_OF_RUNS; i++) {
      clients[myRnd.nextInt(clients.length)].doSomething();
   }

   try {
      MyLogger logger = MyLogger.getInstance();
      logger.endLogging();
   } catch (IOException e) {
      e.printStackTrace();
   }
}
```

***Hinweis**: in unserem Beispiel kann der Logger nicht parallel genutzt werden, da wir singlethreaded arbeiten. In einer multithreaded Umgebung sollten die Methoden „`getInstance`“, „`logError`“ und „`endLogging`“ als `synchronized` Methoden umgesetzt werden, da es sonst zu Fehlverhalten beim Parallelzugriff kommen kann.*

Realisieren Sie die Klasse `MyLogger`.