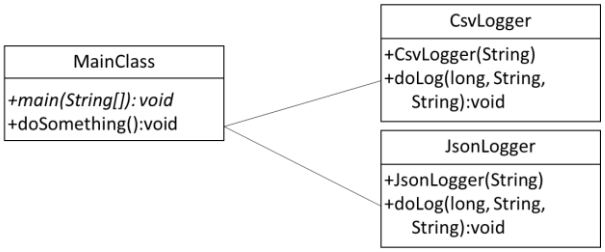
1 Problemstellung



Objekte werden immer durch einen Konstruktor erzeugt. Hierbei muss der Konstruktoraufruf (und somit die Klasse des zu erzeugenden Objektes) in dem nutzenden Code vorhanden sein.

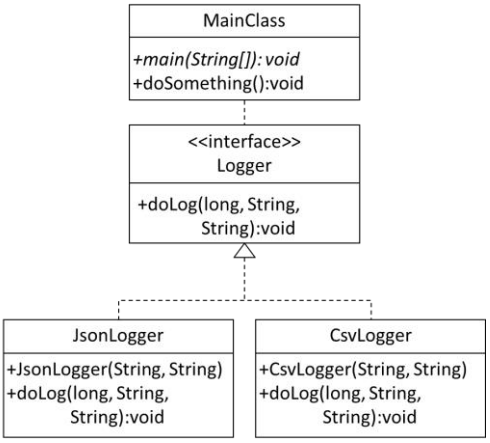
Dies hat nun zwei Nachteile. Der offensichtliche ist, dass wir in der `MainClass` jeden Logger berücksichtigen müssten:

```
public class MainClass {
   private CsvLogger myCsvLogger;
   private JsonLogger myJsonLogger;

//…
   private void doSomething() {
      // …
      if(myCsvLogger != null) {
         myCsvLogger.doLog(1, "Error", "something went wrong");
      } else if(myJsonLogger != null) {
         myJsonLogger.doLog(1, "Error", "something went wrong");
      }
   }
//…
}
```

2 Entkoppelung via Interface

Dieses Problem würden wir einfach über einen polymorphen Ansatz mit Hilfe eines Interfaces oder einer geeigneten Elternklasse realisieren können.



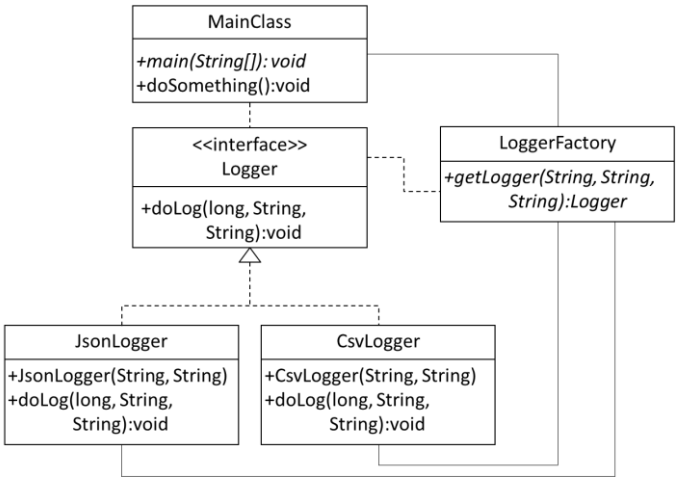
Das Interface kümmert sich um die Abstraktion der Logging-Funktionalität und die einzelnen Logger um die Implementierung.

Es gibt jedoch nun noch eine weitere Problematik. Wenn wir bei den Konstruktoren Änderungen vornehmen, oder gar neue Logger einfügen würden, dann würde dies wiederum zwangsläufig zu Änderungen in der `MainClass` Implementierung führen.

3 Statische Factory Methode

Auf den ersten Blick scheint dies unumgänglich. Jedoch könnten wir nun Gedanken für die Erzeugung unseres Loggers realisieren:

Wir lagern die Konstruktoraufrufe in eine statische Methode einer eigenen Klasse aus, welche sich exklusiv um die Objekterzeugung kümmert – eine sogenannte „Factory Methode“. Diese Methode gehört architektonisch zu den Logger-Klassen. In der Praxis würde man ein eigenes Package generieren, in dem sämtliche logging-relevanten Funktionalitäten zusammengefasst wären, also eine Art „Logging Biblitothek“



Der Code einer solchen Methode könnte wie folgt aussehen:

```
public class LoggerFactory {

//…
   private Logger getLogger(String type, String path, String fileName) {
      switch(type) {
         case "json":
              return new JsonLogger(path, filename);
         case "csv":
              return new CsvLogger(path, filename);
      }
      System.out.println("unknown logger type – creating csv Logger instead);
      Return new CsvLogger(path, filename);
//…
}
```

In der nutzenden `MainClass` würde dann die Erzeugung eines Loggers ohne eine explizite Nennung des Konstruktors ablaufen:

```
public class MainClass {
   private Logger myLogger;

//…
   private void doSomething() {
      // …
      myLogger = LoggerFactory.getLogger("csv", "C:\\log", "error.log");
      myLogger.doLog(1, "Error", "something went wrong");
      // …
   }
//…
}
```

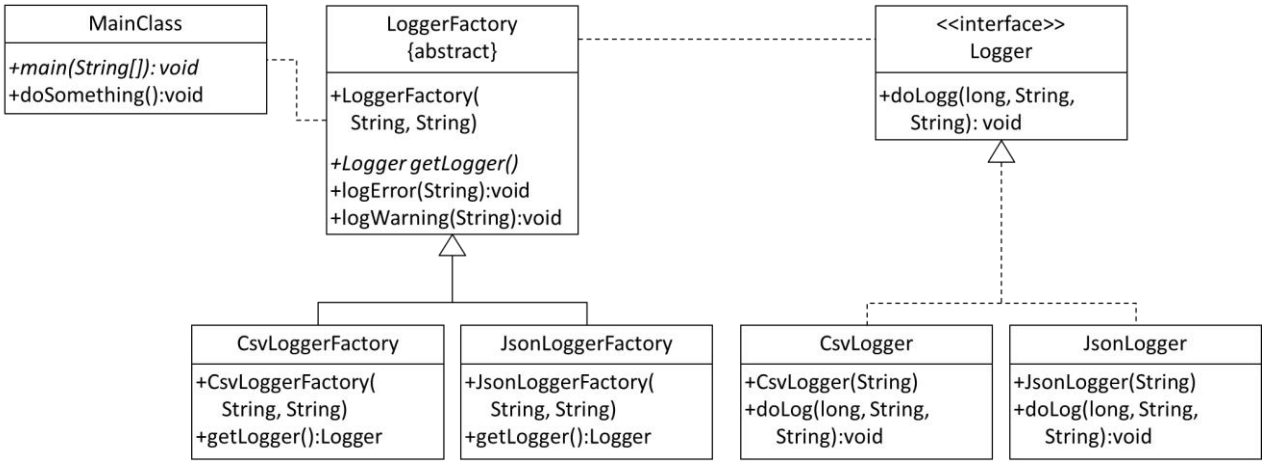
Wenn wir nun einen dritten Logger ergänzen wollen, dann können wir im logging Package diesen Ergänzen, die Factory erweitern und müssten dann lediglich beim Aufruf in der `MainClass` den 1. Parameter der `getLogger` Methode ändern, bspw.:

```
   private void doSomething() {
      // …
      myLogger = LoggerFactory.getLogger("xml", "C:\\log", "error.xml");
      // …
   }
```

Dieses Konzept der statischen Factorymethode (mitunter auch „einfaches Factory Pattern“ genannt) ist zwar kein klassisches „Gang of Four“ Pattern, findet sich jedoch recht häufig in Code wieder. Die GoF hat jedoch ein eigenes Factory Pattern vorgeschlagen.

4 Factory Pattern

Hierbei wird die Abstraktion noch einen Schritt weiter – in Richtung Factory getrieben:

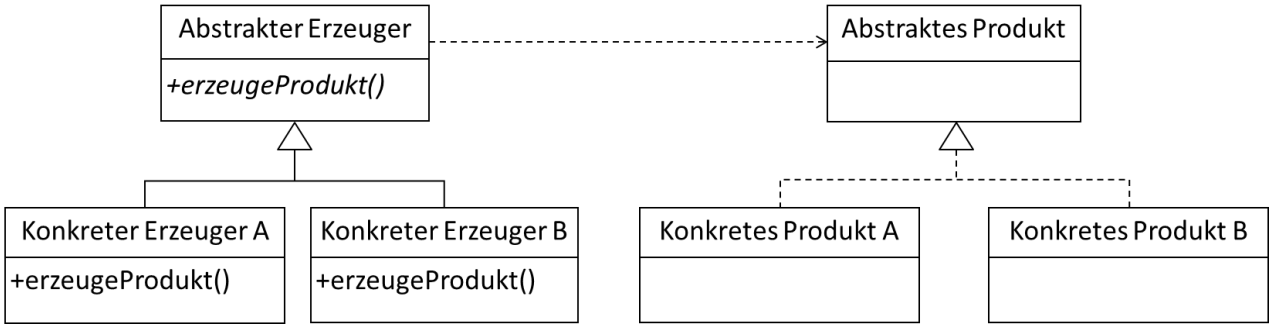


Die Idee hier ist, dass auch die Factory erstmal abstrakt vorgegeben wird. Dadurch können eventuelle Eigenheiten bei der Erzeugung der einzelnen Produkte individueller durch die Factory abgebildet werden. Weiterhin kann die Nutzung des tatsächlichen Loggers auch durch die Factory abgebildet werden. Das heißt, die Factory erzeugt die ausführenden Objekte und kümmert sich um die korrekte Nutzung dieser. In der `MainClass` würde die Nutzung beim Logging wie folgt aussehen:

```
public class MainClass {
   private LoggerFactory logFact;

//…
   private void doSomething() {
      // …
      logFact = new CsvLoggerFactory("C:\\log", "error.log");
      logFact.logError("something went wrong");
      // …
   }
//…
}
```

Das Detailwissen über die einzelnen Logger kann somit in die Factory ausgelagert werden. Das ist zwar nicht zwingend notwendig, kann aber durchaus so implementiert werden. Würde man auf die Factorymethode „`getLoger()`“ verzichten, kann die Nutzung der eigentlichen Logger komplett von der `MainClass` „versteckt“ werden. Das allgemeine Bild für das Konzept des Factory Patterns sieht dieses „verstecken“ jedoch nicht vor, sondern kümmert sich primär um die Objekterzeugung:



Anmerkung: Es existiert neben dem Factory Pattern auch die „Abstract Factory“. Hierbei versucht man zusammengehörige Produkte mittels einer Factory zu erzeugen. Wenn also mehrere, zusammengehörige Produkte erzeugt werden sollen, dann übergibt man diese Aufgabe an eine Factory für die Produkte der Gruppe A und eine zweite Factory würde die Produkte der Gruppe B erzeugen.