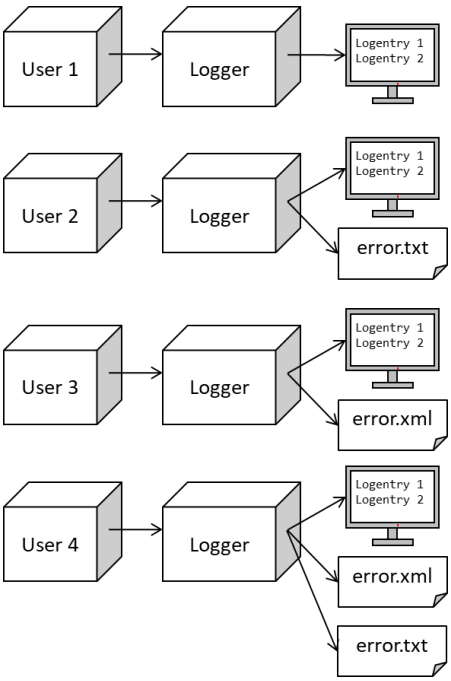
	Pattern: Decorator		AnPr	V 1.0
	Name	Klasse	Datum	

1 Problemstellung



Objekte erfüllen immer eine definierte Funktionalität. Mitunter möchte man jedoch diese Funktionalität um weitere Elemente erweitern. Eine Möglichkeit wäre die Erweiterung durch Kindklassen. Das Problem hier ist, dass die Erweiterung statisch erfolgt – sprich wir müssen im Code bei der Erzeugung des Objektes die Funktionalität festlegen, nicht während der Laufzeit. Weiterhin ist das Problem, dass wir bei einer hohen Anzahl von möglichen Erweiterungen (und deren Kombinationen) extrem viele Kindklassen erzeugen müssten.

Folgende Funktionalität wird jedoch erwünscht:

- Erzeuge das Objekt mit einer Basisfunktionalität
- Wenn notwendig, erweitere sie zur Laufzeit um weitere Funktionalitäten

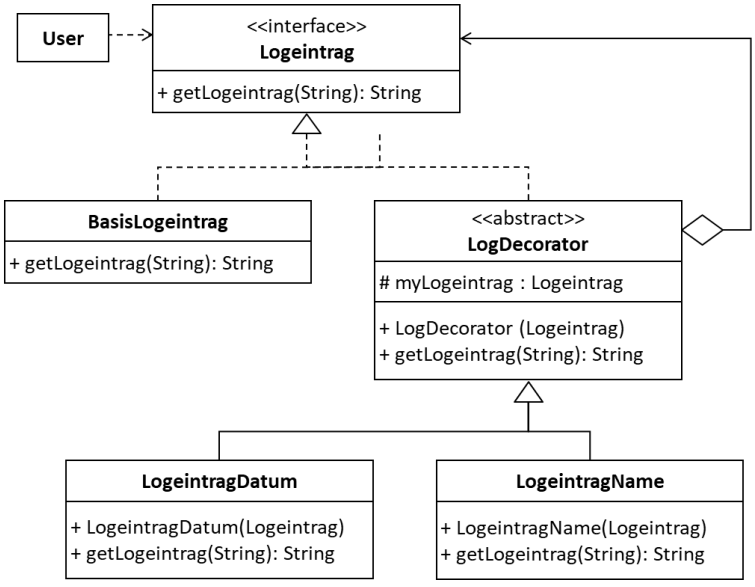
Beispielsweise können wir eine Klasse schreiben, welche Loginformationen auf dem Bildschirm anzeigt („Basisfunktionalität“). Ggf. möchte man nun zusätzlich ein zentrales Logfile schreiben. Wiederum andere Nutzer benötigen ein Logfile im XML Format, oder gar mehrere Formate. Dekorierer ermöglichen diese Funktionserweiterung mit relativ wenig Aufwand.

Die Zusatzfunktionalität soll in einer zentralen Stelle definiert werden und die Basisfunktionalität erst ergänzt werden, wenn der Zusatz wirklich benötigt wird.

2 Problemlösung

Im Wesentlichen sind drei Dinge wichtig:

- Interface (oder abstrakte Klasse), welches die zu dekorierende Methode festlegt
- Basisfunktionalität, welche immer benötigt wird
- Decorator, welcher als Elternklasse für die unterschiedlichen Decorator-Erweiterungen fungiert



Sehen wir uns hierfür ein vereinfachtes Beispiel an. Eine Klasse soll Logeinträge generieren. Hierzu realisieren wir das Interface für die Methodensignatur und eine Basisklasse für die Erzeugung eines grundlegenden Logeintrages:

```
public interface Logeintrag {
   String getLogeintrag(String errorReason);
}
```

```
public class BasisLogeintrag implements Logeintrag {

   @Override
   public String getLogeintrag(String errorReason) {
      return "Error: " + errorReason;
   }
}
```

Nun wollen wir die Logfunktionalität um ein Logdatum und der Angabe des aktuellen Users erweitern. Hierzu erzeugen wir uns die abstrakte Decorator Klasse, welche eine Referenz auf die Basisklasse halten kann:

```
public abstract class Decorator implements Logeintrag {
   protected Logeintrag myLogeintrag;

   public Decorator(Logeintrag myLogeintrag) {
      this.myLogeintrag = myLogeintrag;
   }

   @Override
   public String getLogeintrag(String errorReason) {
      return basisLog.getLogeintrag(errorReason);
   }
}
```

Nun erfolgen die Implementierungen der eigentlichen Dekoratoren:

```
public class LogeintragDatum extends Decorator {

   public LogeintragDatum(Logeintrag myLogeintrag) {
      super(myLogeintrag);
   }

   @Override
   public String getLogeintrag(String errorReason) {
      String date = "| Timestamp: " + (new Timestamp(System.currentTimeMillis()));
      return super.getLogeintrag(errorReason + date);
   }
}
```

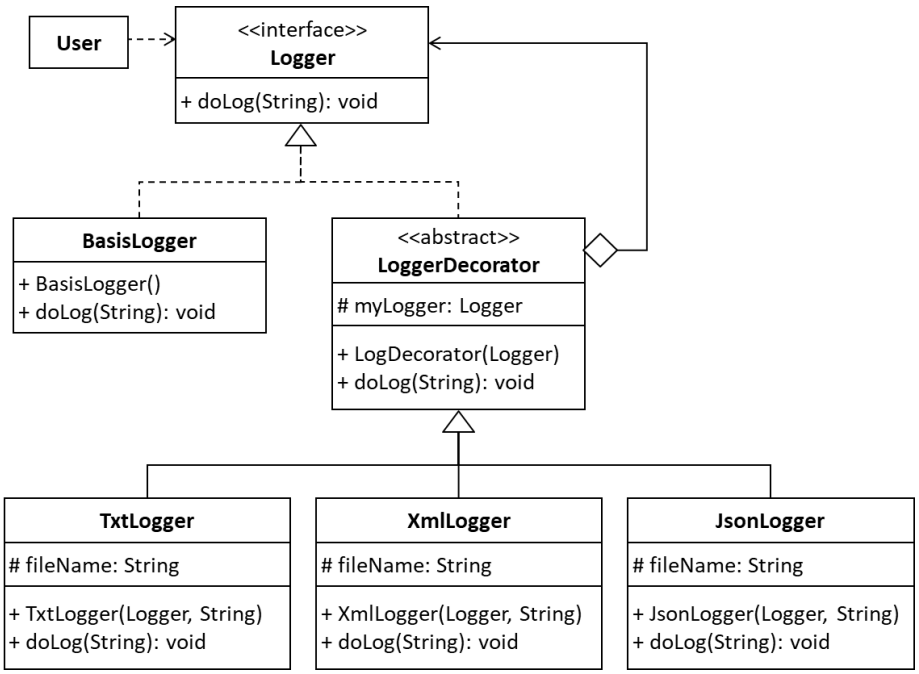
```
public class LogeintragName extends Decorator {

   public LogeintragName(Logeintrag myLogeintrag) {
      super(myLogeintrag);
   }

   @Override
   public String getLogeintrag(String errorReason) {
      String userName = "| User: " + System.getProperty("user.name");
      return super.getLogeintrag(errorReason + userName);
   }
}
```

Nun können wir in der Variablen `myLog` einmal das Basislogging testen, anschließend um den Namen und zum Schluss noch um das Datum erweitern:

```
public class DecoratorTest {

   public static void main(String[] args) {
      Logeintrag myLog = new BasisLogeintrag();
      System.out.println(myLog.getLogeintrag("some Error"));

      myLog = new LogeintragName(myLog);
      System.out.println(myLog.getLogeintrag("some Error"));

      myLog = new LogeintragDatum(myLog);
      System.out.println(myLog.getLogeintrag("some Error"));
   }
}
```

3 Aufgabenstellung

Schreiben Sie eine Klasse `BasisLogger`, welche Fehlermeldungen auf der Konsole ausgibt. Diese soll durch Dekoratoren um ein Text-Logfilewriter, einem JSON-Logfilewriter und einen XML-Logfilewriter ergänzt werden:



Eine Ausgabe der Lognachricht „MyMessage“ sieht auf der Konsole wie folgt aus:

```
Log (2022-09-30 09:44:27.445): MyMessage
```

Der Textlogger ergänt die Ausgabe zusätzlich in ein Textfile. Der XML Logger gibt folgendes aus:

```
<log>
  <time>2022-09-30 09:44:27.445</time>
  <msg>MyMessage</msg>
</log>
```

Im JSON-File soll die Nachricht wie folgt aussehen:

```
{
  "time":"2022-09-30 09:44:27.445",
  "msg":"MyMessage"
},
```

Die Zeitstempel generiert jede Klasse selbst. Die Testmethode sollte wie folgt laufen:

```
public static void main(String[] args) {
   Logger myLogger = new BasisLogger();
   myLogger.doLog("Fehler 1");

   Logger myTxtLogger = new TxtLogger(myLogger, "C:\\tmp\\error.txt");
   myTxtLogger.doLog("Fehler 2");

   Logger myXmlLogger = new XmlLogger(myTxtLogger, "C:\\tmp\\error.xml");
   myXmlLogger.doLog("Fehler 3");

   Logger myJsonLogger = new JsonLogger(myLogger, "C:\\tmp\\error.json");
   myJsonLogger.doLog("Fehler 4");
}
```

Auf der Konsole erwarten wir dann:

```
Log (2023-10-23 09:01:39.33): Fehler 1
Log (2023-10-23 09:01:39.347): Fehler 2
Log (2023-10-23 09:01:39.358): Fehler 3
Log (2023-10-23 09:01:39.362): Fehler 4
```

Im `error.txt`:

```
Log (2023-10-23 09:01:39.353): Fehler 2
Log (2023-10-23 09:01:39.359): Fehler 3
```

Im `error.xml`:

```
<log>
<time>2023-10-23 09:01:39.359</time>
<msg>Fehler 3</msg>
</log>
```

Und in `error.json`:

```
{
"time":"2023-10-23 09:01:39.363",
"msg":"Fehler 4"
},
```