Wartung

Wartung

- Die Wartung bezieht sich auf den Zeitraum, in dem die Software beim Kunden in der produktiven Umgebung läuft

> *operation and maintenance phase – The period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements*

> *maintenance – The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment*

IEEE Std 610.12 (1990)

- Diese Definitionen bieten aber keine exakte Abgrenzung zu den Aktivitäten bei einer inkrementellen Software-Entwicklung

- Während bei einer inkrementellen Entwicklung die Tätigkeiten geplant sind, so zeichnet sich die Wartung durch einen hohen Anteil an nicht planbaren Aktivitäten aus
- Dies führt zu der folgenden Definition der Wartung [LL10]

> *Software-Wartung ist die Arbeit an einem bestehenden Software-System, die nicht von Beginn der Entwicklung an geplant war oder hätte geplant werden können, und die unmittelbaren Auswirkungen auf den Benutzer der Software hat*

Zur Abgrenzung vom Refactoring

- Die Bedeutung der Wartung ergibt sich durch ihren hohen Kostenanteil
- Fast alle Studien beziffern die Wartungskosten auf > 50 Prozent der Gesamtkosten einer Software (siehe [LL10])

- Es werden mehrere Arten der Wartung unterschieden (aus [LL10]):
  - Adaptive Wartung (Anpassung): Die Software wird so verändert, dass sie neue oder geänderte Anforderungen erfüllt; diese können funktional und nichtfunktional sein
  - Korrektive Wartung (Korrektur): Die Software wird so verändert, dass ein beobachteter Fehler nicht mehr auftritt

Es wurden noch weitere Arten der Wartung definiert. Diese zusätzlichen Definitionen haben sich aber als nicht hilfreich erwiesen

- Eingriffe in ein bestehendes Softwaresystem sind sehr fehleranfällig
- Die Wartung muss daher systematisch erfolgen
  1. Der Ausgangszustand wird präzise dokumentiert und archiviert
     ← eigentlich eine Grundvoraussetzung
  2. Der zu modifizierende Teil wird identifiziert
     ← wird durch eine geeignete Architektur erleichtert
  3. Testfälle für die durchzuführenden Änderungen werden systematisch entworfen
     ← die bereits bestehenden Testfälle bleiben für den Regressionstest bestehen
  4. Die zu ändernden Komponenten werden aus der Konfigurationsverwaltung entnommen, bearbeitet und lokal geprüft
     z.B. durch Modultest, Review oder Metriken →
  5. Die veränderten Komponenten werden integriert und getestet

Abweichungen beim Regressionstest müssen durch Spezifikation begründet sein ↗

  6. Der neue Zustand wird dokumentiert und archiviert
  7. Der benötigte Aufwand wird erfasst

- Im Rahmen der Konfigurationsverwaltung müssen Änderungen systematisch verwaltet werden (Change Management)
- Am Anfang einer jeden Änderung im Rahmen der Wartung steht eine Problemmeldung (Software Problem Report, SPR)
- Die Problemmeldung muss klassifiziert werden
  - Irrtum (z.B. falsche Verwendung der Software)
  - Fehlermeldung (bekannt, unbekannt)
  - Änderungswunsch
- Eine Problemmeldung kann dann eine Änderung der Software erfordern
- Eine Änderung erfordert einen entsprechenden Änderungsantrag (Change Request, CR)

- Die Verwaltung von Problemmeldungen und Änderungsanträgen erfordert den Einsatz von Werkzeugen

z.B. Bugzilla oder Trac

- Der Umgang mit Änderungs- und Erweiterungswünschen hängt von der Organisation ab
  - Externe Kunden: Abschluss von Wartungsverträgen oder neuer Vertrag pro Änderung (Budget muss vorhanden sein)
  - EDV-Projekt: Ein Gremium (*Change Control Board*, CCB) entscheidet

- Bearbeitung von Problemmeldungen
  1. Problemmeldung erfassen (inkl. Kennung und Datierung)
  2. Problemmeldung analysieren
     - Irrtum und Fehlbedienung ausschließen
     - Auswirkung eines Fehlers bzw. Nutzen einer Anpassung bestimmen
  3. Entscheidung des CCB vorbereiten
     - Finanzierung klären
  4. Absender über die Entscheidung des CCB informieren
  5. Änderungen durchführen und prüfen
  6. Problemmeldung schließen

- Um Wartungsaktivitäten effizient und effektiv durchführen zu können müssen die folgenden Fragen schnell beantwortet werden können
  a) Liegt aktuell ein Problem (Defekt, Ausfall, Störung) vor?
  b) Was ist die Ursache eines Problems?
  c) Wie wird die Anwendung genutzt? Gibt es Verbesserungspotenzial?

- Wenn das Programm geeignete Kontrollausgaben erzeugt, wird die Beantwortung der Fragen erleichtert (oder gar erst ermöglicht)

Logging

- Die Erzeugung von Kontrollausgaben (die für den Anwender nicht direkt ersichtlich sind) wird als *Logging* bezeichnet
  - Die Kontrollausgaben müssen persistent gespeichert werden, um eine spätere Analyse zu ermöglichen
    - Die Verarbeitungshistorie kann zudem wichtige Hinweise auf die Ursachen eines Problems liefern

- Das Schreiben/Speichern von *Logging*-Informationen benötigt Ressourcen
  - Rechenzeit
  - Speicherplatz
- Die *Logging*-Informationen werden daher nach ihrer Wichtigkeit klassifiziert
  - Jeder Meldung wird ein *Log-Level* zugeordnet
  - Nur Meldungen ab einem bestimmten *Log-Level* werden ausgegeben/gespeichert
- Während der Programmlaufzeit muss es möglich sein, zu bestimmen, welcher *Log-Level* mindestens für eine Meldung erforderlich ist
  - z.B. über Konfigurationsdateien, die in regelmäßigen Abständen eingelesen werden

– Beispiel: Die *Log-Level* der *Logging-Frameworks log4j* (`DEBUG` hat die geringste Wichtigkeit)

Log-Level	Bedeutung
`DEBUG`	Informationen zum Aufspüren von Fehlerursachen
`INFO`	Protokollierung des Programmflusses
`WARN`	Betriebssituation weicht vom erwarteten Verhalten ab, wird aber kontrolliert
`ERROR`	Programmablauf wird gestört (z.B. unvollständig behandelte `Exception`)
`FATAL`	Schwerwiegender Fehler. Programm wird beendet

- Eine *Logging*-Meldung sollte zumindest die folgenden Informationen enthalten
  1) Datum der Meldung
  2) Uhrzeit der Meldung
  3) Log-Level der Meldung
  4) Thread / Prozess
  5) Kategorie (z.B. Komponentenname, Klassenname)
  6) Zusätzliche Informationen (Freitext)

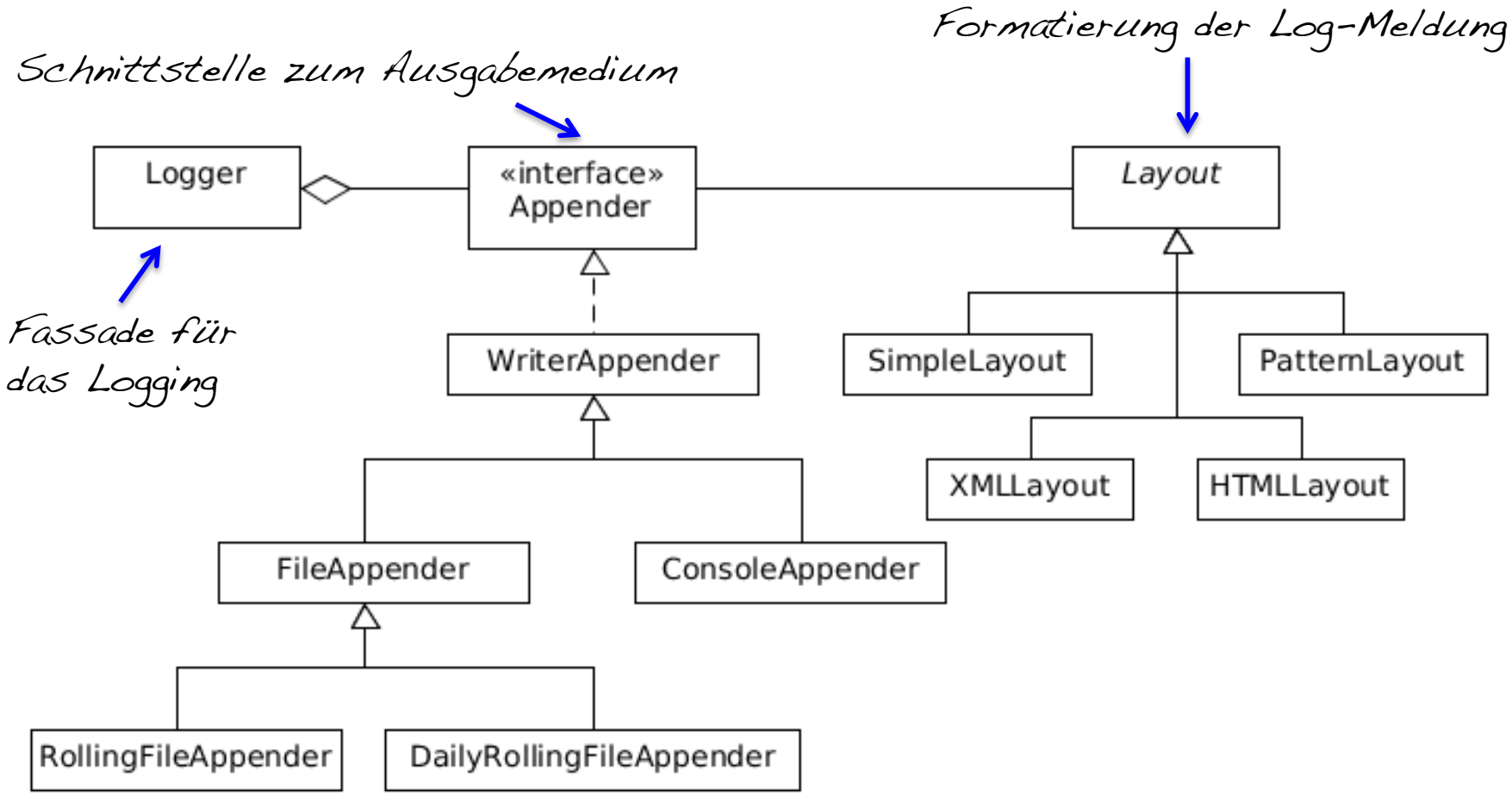
```
2015.11.25-15:08:54 WARN  [main] de.fhdortmund.swtd.Util:
istErstesHalbjahr wurde mit dem ungültigen Monat 0 aufgerufen.
```

Achtung: Rechtliche Aspekte sind zu beachten. Nicht alle Daten dürfen einer Auswertung zugänglich gemacht werden

- Als Beispiel betrachten wird das *Logging-Framework log4j*
  - Die zugehörige `jar`-Datei muss sich im Klassenpfad des Projekts befinden

```
<dependency>
   <groupId>log4j</groupId>
   <artifactId>log4j</artifactId>
   <version>1.2.17</version>
</dependency>
```

– Schnittstellen und Klassen aus dem Paket `org.apache.log4j`



a) Logging mit einer Grundkonfiguration

```
public class Util {
   private static Logger LOGGER = Logger.getLogger(Util.class);

   public static boolean istErstesHalbjahr(int monat) {
      LOGGER.info("istErstesHalbjahr(" + monat + ") aufgerufen.");
      if ((monat < 1) || (monat > 12)) {
         LOGGER.warn("istErstesHalbjahr mit ungültigem Monat");
         throw new IllegalArgumentException();
      }
      if (monat <= 6) return true;
      return false;
   }
   public static void main(String[] args) {
      BasicConfigurator.configure();
      if (istErstesHalbjahr(0)) { System.out.println("Ja"); }
    }
}
```

Kategorie (Logger-Name): hier Klassenname

Grundkonfiguration:
ConsoleAppender (Ausgabe über System.out)
+ einfaches PatternLayout

b) Programmatische Konfiguration eines *Loggers*

1) *Layout* erzeugen
2) *Appender* erzeugen und mit *Layout* versehen
3) *Appender* zum *Logger* hinzufügen
4) *Log-Level* einstellen

```
final Layout layout =
  new PatternLayout("%d{YYYY.MM.dd-HH:mm:ss} %-5p [%t] %c: %m%n");
try {
   final RollingFileAppender rfa =
               new RollingFileAppender(layout, "./app.log", true);
   rfa.setMaxFileSize("1MB");
   LOGGER.addAppender(rfa);
} catch (IOException e) {
   e.printStackTrace();
}
LOGGER.setLevel(Level.DEBUG);
```

c) Verwendung einer Konfigurationsdatei (hier: `log4j.properties`)

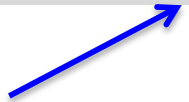
```
log4j.rootLogger=INFO, MyAppender

log4j.appender.MyAppender=org.apache.log4j.RollingFileAppender
log4j.appender.MyAppender.File = app.log
log4j.appender.MyAppender.MaxFileSize = 1000KB
log4j.appender.MyAppender.layout =org.apache.log4j.PatternLayout
log4j.appender.MyAppender.layout.ConversionPattern=%d{YYYY.MM.dd-
HH:mm:ss} %-5p [%t] %c: %m%n
```

log4j.properties

- Einlesen der Konfiguration

```
PropertyConfigurator.configureAndWatch("log4j.properties", 5000);
```



Zeitintervall für das Prüfen auf Änderungen in der Konfiguration

- Einstellen von *Log-Level* auf Paket- und Klassenebene

```
log4j.logger.<paket>.<klasse>=<log-level>
```

- Abfrage der *Log-Level*

Wann ist eine solche Abfrage sinnvoll?

```
if (LOGGER.isInfoEnabled()){
   LOGGER.info("Aufruf der Service-Methode");
}
```

- Logging von Exceptions (*Stacktrace*)

```
try{
   writeBufferToFile(b);
} catch (IOException e){
   LOGGER.warn("Fehler bei der Ausgabe", e);
}
```