Fragen, Antworten und Kommentare zur aktuellen Vorlesung

Achtung, ab der zweiten Lernnotiz beziehen sich die Seitenangaben auf die 5. Auflage von [Kle25].

[Kle25] S. Kleuker, Grundkurs Software-Engineering mit UML, 5. aktualisierte Auflage, Springer Vieweg, Wiesbaden, 2025

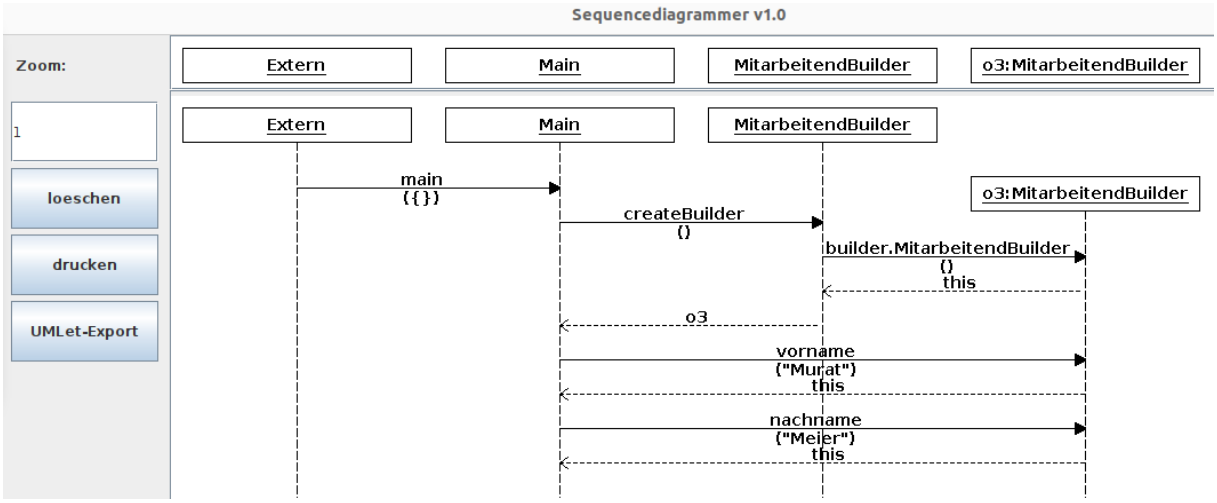
herunterladbar unter https://link.springer.com/book/10.1007/978-3-658-46534-6 mit „Log in via an institution“

Frage: Haben die Begriffe Controller und Entity was mit dem Boundary-Controller-Entity (BCE)-Pattern zu tun?

Ja, wobei Pattern sich meist auf sehr konkrete Arten von Klassen beziehen, es bei BCE eher um einen Architekturansatz geht. Wichtig ist aber, dass Controller und Entity auch in anderen Ansätzen nutzbar sind. Generell bleibt es dabei, dass Entity-Klassen nur mit anderen Entity-Klassen verknüpft sind und Controller die Entitys verwalten, also alle CRUD-Methoden (create-read-update-delete) für die Entitys anbieten. Soll dann der Begriff Boundary noch genutzt werden, gibt es eine Klasse, die den Zugriff auf den Controller koordiniert. Dies kann z. B. eine Klasse sein, die Anfragen per REST (vereinfacht HTTP-Befehle) annimmt, diese aufbereitet, den Controller mit den aufbereiteten Daten nutzt und dessen Antworten passend formatiert zurückgibt. Eine Boundary-Klasse kann aber auch eine einfache Nutzungsschnittstelle für Ein- und Ausgaben auf der Konsole sein.

Frage: Ich erinnere mich an das erste Aufgabenblatt mit dem Builder. Warum waren eigentlich zwei MitarbeitendBuilder-Objekte im Sequenzdiagramm, obwohl nur eines erzeugt wird?

Antwort:



Ja, Achtung, das erste MitarbeitendBuilder-Objekt ist kein Objekt vom Typ MitarbeitendBuilder sondern ein sogenanntes Klassenobjekt. Dies hat was mit dem Thema Reflexion zu tun, das u. a. behandelt, wie Klassen und Objekte entstehen. Etwas vereinfacht formuliert, wird der Code zunächst in ByteCode kompiliert und dann geladen. Dabei wird bereits festgestellt, welche Klassen benötigt werden und für diese Klassen jeweils immer genau ein Klassenobjekt angelegt. Dieses

Klassenobjekt hat zwei wesentliche Aufgaben, die Erzeugung von Objekten, also was beim Aufruf von new passiert und die Ausführung von Klassenmethoden. Dazu verwaltet das Klassenobjekt alle Klassenvariablen (static). Beim Lesen der letzten beiden Sätze könnte Ihnen die richtige Idee kommen, das „new“ ist eigentlich auch nur eine normale Klassenmethode mit anderer Syntax.

Da diese Klassenobjekte vor dem Start des Programms existieren, werden sie im Sequenzdiagramm in die Kopfzeile der bereits existierenden Objekte eingetragen. Im Sequenzdiagrammer können Klassenobjekte und „normale“ Objekte dadurch unterschieden werden, dass normale Objekte immer einen Namen, hier o3, bekommen. Aus dem Sequenzdiagramm ist damit unmittelbar herauslesbar, dass createBuilder eine Klassenmethode sein muss, in der ein Objekt erzeugt wird, das dann zurückgegeben wird.

In der Literatur gibt es sehr wenig Informationen, wie Klassenmethoden in Sequenzdiagrammen dargestellt werden. Vereinzelt werden Sie direkt mit Objekten verbunden, was aber nicht die Frage klärt, wie ein Aufruf aussehen sollte, wenn noch kein Objekt existiert. Im UML-Standard ist der Fall ebenfalls im graphischen Teil nicht thematisiert. Dies hat aber den Grund, dass die formale Semantik ein normales Metaklassenmodel für die Objektorientierung nutzt, das eine Klasse hat, die die erwähnten Klassenobjekte erzeugt. Dadurch wird ein Klassenobjekt zu einem normalen Objekt bezüglich der Objektorientierung und wird, wie dargestellt als normales Objekt im Sequenzdiagramm dargestellt. Zum Start von Java-Programmen wären damit auch spannende Sequenzdiagramme darstellbar, die nach dem Laden das Erzeugen der Klassenobjekte zeigen.

Die Klassenobjekte spielen auch in Java-Programmen direkt eine Rolle und sind über MitarbeitendBuilder.class oder o3.getClass() als Ergebnis nutzbar. Das Objekt kennt u. a. alle Objektvariablen und Objektmethoden eines Objekts und kann diese verändern und aufrufen.

Das Thema Reflexion ist bei der ersten Betrachtung komplex, allerdings elementar, um Objektorientierung im Detail zu verstehen. Wir machen noch einen kleinen Einstieg mit einer Praktikumsaufgabe.

Falls jemand interessante Literatur zum Thema „Darstellung von Klassenmethoden in Sequenzdiagrammen“ findet, bin ich immer interessiert.

Frage: Unter Linux (Mac?) scheint die Konfigurationsdatei des Sequencediagrammers keinen Einfluss zu haben?

Antwort: Wenn Sie die Jar-Datei aus der KleukersSEU nutzen, kann das passieren, da diese für Windows erstellt wurde. Für andere Betriebssysteme direkt die Jar-Datei von der Seite http://kleuker.iui.hs-osnabrueck.de/gsdet/index.html laden.

Frage: Soll ich die Tests eigentlich alle verstehen können?

Antwort: Generell ja, da Software ohne Tests nicht existiert, da sie potenziell unzuverlässig ist. Eventuell müssen Sie nicht in alle Details schauen. Grob sind immer JUnit-Tests gegeben, deren grundsätzlicher Aufbau u. a. mit @BeforeClass, @AfterClass, @BeforeMethod, @AfterMethod, @Test für jede mit Java entwickelnde Person zum Grundwissen gehört. Es werden zwei

Erweiterungen genutzt. Mit system-lambda.jar können recht einfach Eingaben über die Konsole und Ausgaben auf der Konsole erzeugt und gelesen werden.

```
private void neuesBild(String serie, int nr) throws Exception {
  String[] inputs = { serie, "" + nr };
  SystemLambda.withTextFromSystemIn(inputs)
      .execute(() -> {
        this.dialog.sammelbildHinzufuegen();
      });
}
```

Der obige Test ruft z. B. die Methode sammelbildHinzufuegen() auf, die einen Nutzungsdialog enthält, der aus zwei Eingabeschritten besteht. Die genutzten Eingaben stehen im Array inputs.

```
@Test
public void testHinzu1() throws Exception {
  this.neuesBild("XYZ", 100);
  String systemOut = SystemLambda.tapSystemOutNormalized(() -> {
    this.dialog.gesamtbestand();
  });
  Assertions.assertTrue(systemOut.contains("XYZ")
      , "Bild (XYZ,100), XYZ nicht befunden ");
  Assertions.assertTrue(systemOut.contains("100")
      , "Bild (XYZ,100), 100 nicht befunden ");
}
```

Der obiger Test liest in die Variable systemOut alle Konsolenausgaben, die die Methode gesamtbestand() ausgibt.

Da es sich bei Tests um gewöhnliche Java-Klassen handelt, sind sie ebenfalls mit Hilfsmethoden zu strukturieren.

In anderen Tests wird eine Klasse Alle genutzt, die auf alle Eigenschaften einer Klasse, genaue alle Variablen, Konstruktoren und Methoden mit zugehörigen Detailinformationen, wie Typen von Parameterlisten zugreift.

```
private static String[][][] geforderteVariablen = {
  {{"Zugriffsverwaltung"},{"aktuelleNutzung"},{"Nutzung"}, {"private"}}
  , {{"Nutzung"},{"login"},{"String"}, {"private"}}
  , {{"Nutzung"},{"passwort"},{"String"}, {"private"}}
};

private static String[][][] variablen(){
  return geforderteVariablen;
}

@ParameterizedTest
@MethodSource("variablen")
public void testVariablenExistieren(String[][] var){
  String klasse = var[0][0];
  String name = var[1][0];
  String typ = var[2][0];
  String[] sichtbar = var[3];
  Assertions.assertTrue(Alle
        .klasseHatVariableVomTypMitSichtbarkeitUndArt(klasse, name
                                                      , typ, sichtbar)
      , "Geforderte Variable " + name + " in Klasse"
```

```
            + klasse + " mit Typ " + typ + " fehlt");
//    }
  }
```

Der obige Test ist parametrisiert und nutzt die Methode variablen() zur Berechnung der Parameter. Für jedes angegebene Parametertupel wird dann der obige Test ausgeführt. Konkret wird geprüft, ob die gewünschten Objektvariablen mit den genannten Typen und der geforderten Sichtbarkeit in der zu untersuchenden Klasse vorhanden sind.