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: Soll ich die Tests eigentlich alle verstehen können.

Antwort: Generell ja, eventuell müssen Sie nicht in alle Details schauen. Grob sind immer JUnit-Tests gegeben, deren grundsätzlicher Aufbau u. a. mit @BeforClass, @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.