HAW-Logistics-System

im Auftrag der Firma
HAW Logistics
c/o Prof. Dr. Stefan Sarstedt
Software Experience Lab
Fakultät Technik und Informatik
Berliner Tor 7
20099 Hamburg

Testkonzept

Prof. Dr. Stefan Sarstedt

Version: 1.1
Status: Abgeschlossen
Stand: 24.09.2013

Zusammenfassung

Dieses Dokument beschreibt das Testkonzept und die zugehörigen Test-Konventionen des HAW-Logistics-Systems.

Historie

Version	Status	Datum	Autor(en)	Erläuterung
1.1	Abgeschlossen	24.09.2013	Stefan Sarstedt	Initiale Version für das WP 2013/2014.

Inhaltsverzeichnis

1 Einleitung

1.1 Ziele

Ziel dieses Dokuments ist die Definition der Testarten und Strukturen im HLS.

1.2 Randbedingungen

Keine.

1.3 Konventionen

Sourcecode wird folgendermaßen dargestellt, Schlüsselwörter und Typen sind dabei farbig markiert:

```
void CreateGeschaeftspartner(ref GeschaeftspartnerDTO gpDTO);
```

3 Testen im HLS

Das Testen im HLS geschieht auf vier verschiedenen Ebenen:

- Komponententests
- Integrationstests
- Systemtests
- Verbundtests

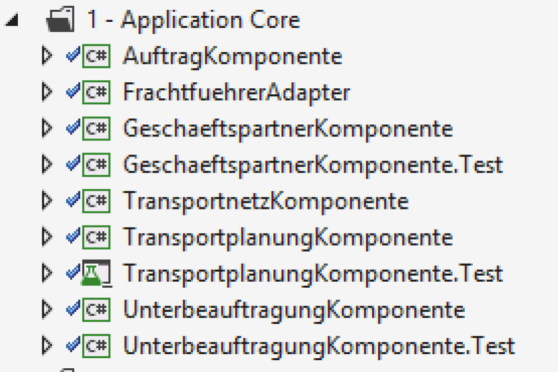
Die Testarten und deren Umsetzung im HLS werden in den folgenden Abschnitten beschrieben.

3.1 Komponententests

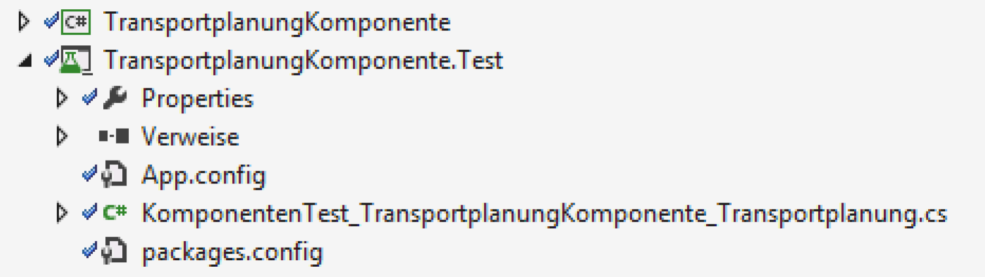
Komponententests testen das Verhalten einer einzelnen Komponente in einem isolierten Kontext, d. h. ohne die realen „Nachbar"komponenten, mit denen diese Komponente interagiert. Während der Entwicklung eines großen Softwaresystems werden im Normalfall mehrere Komponente parallel durch verschiedene Mitarbeiter entwickelt – um nicht abwarten zu müssen, dass die Nachbarkomponenten verfügbar sind, sollte jeder Entwickler seine Komponente zunächst isoliert testen. Dazu benötigt er jedoch Funktionalität der Nachbarkomponenten, welche er durch sogenannte Mock-Objekte selber zur Verfügung stellt. Diese Mocks (http://de.wikipedia.org/wiki/Mock-Objekt) simulieren die Funktionalität der benötigten Komponenten für die Testläufe der eigenen Komponente.

3.1.1 Struktur in Visual Studio

Für die Implementierung der Tests nutzen wir das Microsoft Test-Framework, welches in Visual Studio integriert ist. Für jede Komponente wird ein zugehöriges Testprojekt angelegt, welches wir `<Komponentenname>.Test` nennen:



Jedes Testprojekt enthält mindestens eine Datei mit Tests. Die Namenskonvention lautet `KomponentenTest_<Komponentenname>_<Art der enthaltenen Tests>.cs`:



3.1.2 Aufbau der Testdatei

Die Testdatei ist dem Namespace `Tests.KomponentenTest.<Komponentenname>` zugeordnet. Das Attribut `[TestClass]` markiert die Testklasse:

```
namespace Tests.KomponentenTest.TransportplanungKomponente
{
    [TestClass]
    public class KomponentenTest_TransportplanungKomponente_Transportplanung
    {
        [ClassInitialize]
        public static void InitializeTests(TestContext testContext)
        {
        }

        [ClassCleanup]
        public static void CleanUpClass()
        {
        }

        [TestCleanup]
        public void CleanUpTest()
        {
        }
    }
}
```

Die statischen Methoden `InitializeTests` und `CleanUpClass` übernehmen die Intialisierung der Testklasse. Sie werden genau einmal aufgerufen. Die Initialisierung kann bspw. das Anlegen von allen Tests gemeinsam genutzen Testdaten und die Initialisierung der Persistenz übernehmen. Dagegen übernimmt die Methode `CleanUpTest` das Aufräumen nach einem Test – diese wird also nach jedem einzelnen Test aufgerufen.

Beispiel der Initialisierung auf der Komponente `GeschaeftspartnerKomponente.Test`. Hier wird eine Persistenz initialisiert und die Komponente `GeschaeftspartnerKomponente` instanziiert. Diese wird dann in den folgenden Tests verwendet:

```
[ClassInitialize]
public static void InitializePersistence(TestContext testContext)
{
    log4net.Config.XmlConfigurator.Configure();

    PersistenceServicesFactory.CreateSimpleMySQLPersistenceService(
        out persistenceService,
        out transactionService);

    gpK_AC = new GeschaeftspartnerKomponenteFacade(
        persistenceService,
        transactionService);
}
```

Jeder Test wird in einer eigenen Testmethode definiert. Testmethoden sind mit dem Attribut `[TestMethod]` markiert:

```
[TestMethod]
public void TestCreateGeschaeftspartnerSuccess()
{
    GeschaeftspartnerDTO gpDTO = new GeschaeftspartnerDTO()
                                 { Vorname = "Max", Nachname = "Mustermann" };
    Assert.IsTrue(gpDTO.GpNr == 0, "Id of Geschaeftspartner must be null.");
```

```
    gpK_AC.CreateGeschaeftspartner(ref gpDTO);
    Assert.IsTrue(gpDTO.GpNr > 0, "Id of Geschaeftspartner must not be 0.");
    Assert.IsTrue(gpDTO.Version > 0, "Version of Geschaeftspartner must not be 0.");
}
```

Jede Testmethode beginnt mit dem Wort `Test`, gefolgt von der inhaltlichen Beschreibung des Tests (CRUD-Operationen bleiben Englisch), gefolgt von `Success` bei Erfolgsszenarien oder `Fail` bei Misserfolgsszenarien.

Jeder Test prüft seine Ergebnisse mit den statischen Methoden der `Assert`-Klasse.

Hinweis: Die Ausführungsreihenfolge der Tests ist zufällig und kann sich jederzeit ändern! Daher sind die Tests so zu implementieren, dass sie unabhängig voneinander sind.

3.1.3 Erstellen und Verwendung von Mock-Objekten

Wie bereits diskutiert verwenden wir in Komponententests sogenannte Mock-Objekte, die das Verhalten von noch nicht zur Verfügung stehenden Komponenten simulieren. Hierzu verwenden wir das Moq-Framework [Moq].

Beispiel der Erstellung und Konfiguration eines Mock-Objekts mit Moq:

```
var unterbeauftragungsServicesMock = new
            Mock<IUnterbeauftragungServicesFürTransportplanung>();
unterbeauftragungsServicesMock.Setup(ubs => ubs.BeauftrageTransport(
    It.IsAny<int>(), It.IsAny<DateTime>(), It.IsAny<DateTime>(), It.IsAny<int>(),
    It.IsAny<int>()))
   .Returns(1);
```

Die Komponententests der Transportplanungskomponente müssen eine Mock-Instanz der Unterbeauftragung-Komponente erstellen, da die Transportplanung diese Komponente verwendet. Hierzu wird in obigem Beispiel zunächst ein Mock-Objekt erzeugt (der Typparameter definiert die Schnittstelle, die simuliert werden soll). Dann wird dem Mock in ein oder mehreren `Setup` -Aufrufen mitgeteilt, wie es sich bei bestimmten Aufrufen verhalten soll. Siehe [Moq] für Details zur Verwendung des Moq-Frameworks.

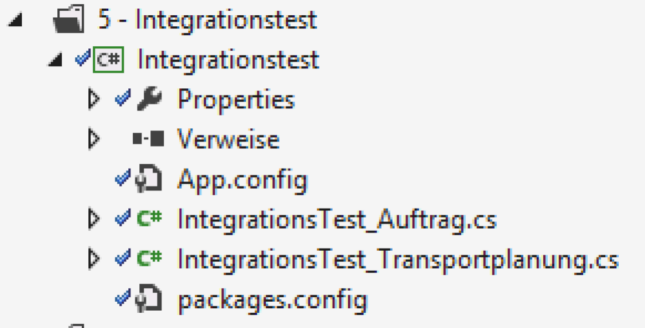
Hinweis: aus Praktikabilitätsgründen verwenden wir in Komponententests immer die „richtige“ Persistenz. Diese wird also niemals gemockt.

3.2 Integrationstests

Integrationstests testen das Zusammenspiel von implementierten Komponenten. Dies ist die zweite Teststufe nach den Komponententests und erfolgt in einer separaten Testphase in einem Projekt.

3.2.1 Struktur in Visual Studio

Für die Integrationstests gibt es ein einziges separates Testprojekt in Visual Studio. Es enthält mehrere Dateien der Form `IntegrationsTests_<Art der enthaltenen Tests>.cs`:



3.2.2 Aufbau der Testdatei

Grundsätzlich ist der Aufbau zu dem der Komponententests identisch (siehe Abschnitt 3.1.2). Der Namespace heißt hier allerdings `Tests.IntegrationsTest`:

```
namespace Tests.IntegrationsTest
{
    [TestClass]
    public class IntegrationsTest_Auftrag
    {
    }
}
```

Jedes Testmethode ist außerdem die Testkategorie `IntegrationsTest` zugeordnet. Dies ermöglich eine separate Ausführung der Integrationstests:

```
[TestMethod, TestCategory("IntegrationsTest")]
public void TestPlanungAngebotAblehnenSuccess()
{
}
```

Integrationstests testen für bestimmte Szenarien das Zusammenspiel von Komponenten, jedoch nicht das System als Ganzes. Dies übernehmen die Systemtests.

3.3 Systemtests

Systemtests testen das Gesamtsystem. Meistens sind dies automatisierte und manuelle GUI-Tests (es sollte soweit wie möglich automatisiert werden). Die realen Nachbarsysteme (bspw. die Anbindung der Frachtführer) werden allerdings in die Testszenarien noch nicht mit einbezogen. Zu groß ist die Gefahr, dass fehlerhafte Daten die Nachbarsysteme korrumpieren könnten.

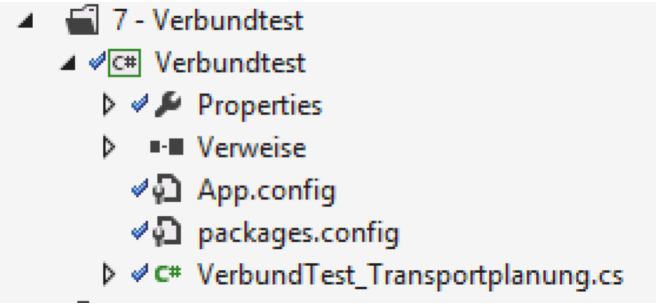
Im HLS ist in der aktuellen Ausbaustufe noch keine GUI vorhanden, deswegen verzichten wir vorerst auf Systemtests.

3.4 Verbundtests

Nachdem die Systemtests sicherstellen, dass das System in seiner Gesamtheit „korrekt“ funktioniert, werden in den Verbundtests die realen Nachbarsysteme in die Tests mit einbezogen. Die Verbundtestphase muss in einem Projekt gut geplant sein – genaue Abstimmungen mit den Betreibern der Nachbarsysteme sind nötig um die Tests in bestimmten Zeitfenstern und in Kooperation mit den externen Mitarbeitern durchzuführen.

Im HLS simulieren wir Verbundtests in der Art, dass wir nach Testläufen in den Queues prüfen, ob entsprechende Daten dort vorhanden sind – also an das fiktive Nachbarsystem geschickt wurden.

Verbundtests sind im HLS in einem separaten Testprojekt definiert:



4 Offene Punkte

- Systemtests und Verbundtests detaillieren.

5 Literatur

Moq https://code.google.com/p/moq/