Workshop 2 : Testning med JUnit

Ramverket JUnit finns med som en plugin i Eclipse.

1 Förberedelse

Vi skall arbeta med en klass som vi testar, felsöker och bygger på vartefter.

1. Hämta Eclipse-projektet test.ep.zip från kursidan. Importera till Eclipse (File > Import > General > Existing Project...). Inspektera och läs kommentarer. Lägg märket till toString-metoden i klassen Node, bra vid utveckling!

2. Vi skiljer på programmets kod och testkoden. Skapa en ny Source folder för testkod (Högerklick > Build Path > New Source Folder > Namn: test)

3. Skapa samma paketstruktur i test som i src-foldern (så kan vi även testa paketprivata klasser).

2 Testning av Klassen List

1. Inspektera klassen List. Det finns ett par metoder klara. Vi skall testa add-metoden.

2. Nu skapar vi en testklass för List. Markera List-klasses > New > Junit Test Case > Name : TestList. Browsa till test-foldern och rätt paket, välj. Skall vara "New Junit 4 test" och "Class Under test: edu.chl.hajo.test.list.List". Låt resten vara > Next > Markera add(Integer) > Finish

   OBS! Eventuellt dyker det upp en fråga (Add Junit 4 ....). Svara Ok! Det visas en JUnit-ikon i Packaget vyn.

3. Inspektera den genererade klassen i test. Det finns en färdig testmetod för add (testmetod = metoden har @Test som annotation).

4. Ändra metoden till följande (alla testmetoder måste vara parameterlösa och public void);

```
import static org.junit.Assert.*;   // Possible add this row
@Test
public void testAdd() {
    List l = new List();
    l.add(1);
    assertTrue(l.getLength() == 1); // The logical check
}
```

   Sista raden är ett påstående om att det booleska uttrycket i parentesen skall vara sant. Om så är testen godkänd. Om ej kommer vi att få ett felmeddelande.

Produced with Lyx, the open source wordprocessor

OBS! Så fungerar alla tester. Testen går igenom eller inte. Ingen manuell inspektion skall behövas (t.ex. System.out.println)[1].

5. Nu skall vi köra testen: Markera TestList > Högerklicka > Run As > JUnit Test, klicka.

6. Ett nytt fönster dyker upp: JUnit. Om testen är ok visas en liggande grön stapel m.m.. Man kan klicka framför testklassens namn (framför ListTest) för att visa alla testmetoder som körts, gör det (testAdd skall synas)!

   Vid fel syns utskrifter längst ned i fönstret (klass och rad för testen som gick fel, ... eller exception).

7. Man kan köra testen från JUnit fönstret, högerklicka i fönstret > Run. Gör det! I detta fall är alltså testen ok.

   OBS! Här testade vi en void-metod, vi fick inget returvärde som vi kunde använda i testen (i det booleska uttrycket)! Om man testar void metoder måste det finnas någon annan metod för att avläsa tilståndet (i vårt fall getLength). Ev. får man lägga till en sådan metod enbart för att göra klassen testbar (skall inte ingå i något interface).

8. Nu skall vi test remove-metoden i List. Metoden tar bort första Noden i listan och returnerar värdet för denna. Skapa en metod testRemove().

   Kolla flera olika saker i testen! Använd returvärdet i testen. Du skall kunna skapa en test som inte går igenom! Vad är felet i List?? Fixa.

9. Nästa test är get-metoden. Skapa en test. Lägg till 5 värden i listan och returnera värdet för index 2 (lite skumt att värdena läggs till i omvänd ordning ...). Metoden fungerar inte och vi antar att vi inte vet varför. Vi skall därför avlusa metoden (vi kör alltså testen i debugläge).

   a) Sätt en brytpunkt vid raden : Node<Integer> pos = head; i List-klassen (dubbelklicka i vänstermarginalen, skall ge en blå punkt).

   b) I kodfönstret i TestList: Högerklick > Debug As > JUnit test. Eclipse byter till Debug-perspektivet och körningen stannar vid brytpunkten (en pil och färgmarkering).

   c) Klicka dig fram (gula pilar upp till i Debug-fönstret, välj "Step Over") genom loopen och inspektera pos (i fönstret Variables, kan även inspektera this.head.next.next, o.s.v. klicka framför ...)

   d) Avsluta avlusningen genom att klicka på alla "röda fyrkanter" du hittar (minst 2, inte bra att köra debug samtidigt som man kör programmet på vanligt sätt).

   e) Upprepa till du hitta felet. Rätta till.

[1] Ev. vid utvecklandet av själva testen men därefter skall det bort (kommenteras ut).

Produced with Lyx, the open source wordprocessor

10. Vi vill även testa att vi verkligen får de fel vi förväntar oss. Skapa en metod testGetBadIndex. I testen skickar vi in felaktiga index till get-metoden och förväntar oss då en IllegalArgumentException. Metoden måste se ut som följande (ingen assertTrue behövs).

```
@Test(expected=IllegalArgumentException.class)
public void testGetBadIndex() {
    // Get a list then ...
    list.get(-1); // Exception!!!
}
```

11. Skapa nu en egen metode copy() i List-klassen som ger en djup kopia av hela listan. Hur testa? Skapa metod och test för denna.

3 Fixturer

1. Vissa tester behöver fräsh indata eller behöver fixa något innan de körs eller efter det att de har körts, kallas en testfixtur. Kan göras med speciella metoder som körs innan/efter den första/sista testmetod har körs eller innan/efter varje testmetod. Lägg till följande i TestList (vi skall normalt inte ha System.out, detta är bara som en demonstration);

```
@BeforeClass
public static void beforeClass(){  //First of all
    System.out.println("Before class");
}
@AfterClass
public static void afterClass(){   //Last of all
    System.out.println("After class");
}
@Before
public void before(){  //Before each method
    System.out.println("Before");
}
@After
public void after(){ //After each method
    System.out.println("After");
}
```

4 Övrigt

- Man kan naturligtvis testa flera samverkande klasser på liknande sätt som ovan, s.k. integrationstestning. Görs inte här man kan vara värdefullt för projektet.

Produced with Lyx, the open source wordprocessor

- Även händelsebaserade testning är möjligt. Låt testklassen (eller någon inre hjälpklass) fungera som observatör. Händelser kan sparas i en lista. Anropa metoder och kontrollera förväntade händelser.

Produced with Lyx, the open source wordprocessor