Workshop 2 : JUnit och TDD

I denna övning ska vi gå genom grunderna i testramverket JUnit och hur man använder det för att skriva enhetstester (unit tests). Vi kommer också att snabbt utvärdera TDD (Test-Driven Development)[1] och diskutera fördelarna med denna arbetsmetod. Det är rekommenderat att TDD övningen görs i grupp. Testramverket 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 kurssidan. Importera till Eclipse (File > Import > General > Existing Project...). Inspektera och läs kommentarerna. Lägg märket till toString() metoden i klassen Node, bra vid utveckling!

2. Vi skiljer på programmets kod och testkod. 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 : ListTest. 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 paketvyn.

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

[1] Wikipedia, Test-driven development: http://en.wikipedia.org/wiki/Test-driven_development

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.

   OBS! Så fungerar alla tester. Testen går igenom eller inte. Ingen manuell inspektion skall behövas (t.ex. System.out.println)[2].

5. Nu skall vi köra testen: Markera ListTest > 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 tillståndet (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 testa remove() metoden i List. Metoden tar bort första Noden i listan och returnerar värdet för denna. Skapa en metod testRemove().

   Kontrollera 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 debug-läge).

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

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

b) I kodfönstret i ListTest: 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" ni hittar (minst 2, inte bra att köra debug samtidigt som man kör programmet på vanligt sätt).

e) Upprepa till ni hittar felet. Rätta till.

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 metod 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 fräsch indata eller behöver göra något innan de körs eller efter det att de har körts; detta kallas för en testfixtur. Kan göras med speciella metoder som körs innan/efter den första/sista testmetoden har körs eller innan/efter varje testmetod. Lägg till följande i ListTest (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 test method
    System.out.println("Before");
}
@After
public void after(){ //After each test method
    System.out.println("After");
}
```

4 En snabb introduktion till TDD

Test-Driven Development går ut på att låta tester driva utvecklingen av koden, istället för tvärtom (vilket brukar vara fallet). Det betyder att man måste skriva tester innan man skriver implementationen (klassen som testas). Testen fungerar då som en mall över hur implementationen är tänkt att fungera. Detta görs normalt i små iterationer där man hoppar mellan att skriva på testet och implementera klassen som testas[3]. Tanken med detta är att man vill förhindra att man missar något i implementationen och att antalet nya buggar under projektets gång minimeras.

1. En populär metod för att fort lära sig att effektivt använda TDD är att dela upp arbetet så att testet skrivs av en part medan implementationen skrivs av en annan part. Vi ska testa denna arbetsmetod i den här övningen. Det är markerat nedan vilken uppgift som ska göras av vilken part.

2. *[Part 1]* Skapa en klass; PrimerTest med testmetoden isPrimeTest() som testar metoden med signaturen *boolean isPrime(int number)*. Testet ska kontrollera att isPrime returnerar sant eller falskt beroende på om ett primtal skickas in i metoden. Skapa sedan Primer klassen men gör endast så mycket som behövs för att få testet att kompilera och köra (det skall inte lyckas). Du skall alltså skapa en tom isPrime() metod i Primer klassen. Var noga med att lägga klasserna på rätt ställe i källkodsträdet.

3. *[Part 2]* Få PrimerTest att passera genom att implementera metoden Primer.isPrime().

4. *[Part 2]* Skapa testmetoden PrimerTest.getPrimeTest() som testar metoden med signaturen *int getPrime(int sequenceIndex)*. Även nu skapar vi endast en tom getPrime() metod i Primer klassen. Metoden är senare tänkt att returnera primtalet

[3] Wikipedia, Test-driven development cycle: http://en.wikipedia.org/wiki/Test-driven_development#Test-driven_development_cycle

sequenceIndex inom sekvensen av primtal. Testet kan förslagsvis kontrollera om följande uttryck håller:

a) getPrime(0) == 2

b) getPrime(9) == 29

c) getPrime(< 0) ska kasta en IllegalArgumentException

5. *[Part 1]* Få PrimerTest att passera genom att implementera metoden Primer.getPrime(). Implementera metoden på ett så enkelt sätt som möjligt.

   OBS! Ett enkelt (men inte så snabbt) sätt att implementera metoden ovan är att helt enkelt anropa isPrime() metoden i en loop.

6. Optimera och faktorisera om Primer klassen om ni ser förbättringar som kan göras. Detta borde nu kunna göras utan risk att nya buggar introduceras. Kör testerna kontinuerligt medan ni ändrar i koden.

5 Heltäckande tester

Enhetstester skall vara heltäckande och förhindra att nya buggar uppstår i klasser i samband med tillägg till koden. De skall också fungera som bra instruktionsklasser som beskriver hur metoder och klasser som testas är tänkta att uppföra sig. För att ett test skall fungera på ett bra sätt och uppfylla dessa krav behöver de vara så heltäckande som möjligt. Det betyder att så mycket av koden som möjligt som ett unit test avser skall traverseras och testas.

Det kan självklart vara väldigt svårt att uppnå full kodtäckning då en klass som testas ofta innehåller många olika villkor och skrymslen. Lyckligtvis så finns det hjälp att få. Det finns många olika verktyg för Java som hjälper till att visa hur kodtäckningen i ett program ser ut. Två sådana verktyg (som är väldigt bra) är Cobertura[4] och JaCoCo[5]. Verktyg som dessa är ett ovärderligt hjälpmedel under projektets gång.

6 Övrigt

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

[4] Cobertura: http://cobertura.sourceforge.net/
[5] JaCoCo Java Code Coverage Library: http://www.eclemma.org/jacoco/