Testning med JUnit

1 Inledning

JUnit är ett ramverk för enhetstestning av Javakod. Det är utvecklat av Kent Beck och Erich Gamma. Här beskrivs hur man använder verktyget JUnit för att testa de klasser man skriver. Mera information finns på:

`https://github.com/junit-team/junit/wiki/`
`http://junit.org/javadoc/latest/`

2 Enhetstestning

När man implementerat en klass måste man, innan den kan användas, testa att den uppfyller sin specifikation. Denna typ av testning brukar kallas enhetstestning. Vid s.k. black-box testning ser man en klass som en svart låda d.v.s. man vet ingenting om hur metoderna är implementerade. Det som testas är att metoderna ger det förväntade resultatet enligt sin specifikation. Antag t.ex. att vi har implementerat en klass som representerar ett bankkonto med följande specifikation:

```
public class BankAccount {
    /**
     * Create new account with balance 0.
     */
    public BankAccount();

    /**
     * Returns balance of account
     * @return balance of account
     */
    public int getBalance();

    /**
     * Deposits the specified amount.
     * pre: The specified amount is >= 0
     * post: The specified amount is added to balance
     * @param n The amount to deposit
     * @throws IllegalArgumentException if the specified amount is < 0
     */
    public void deposit(int n);

    /**
     * Withdraws the specified amount.
     * pre: The specified amount is >= 0 and <= balance
```

```
     * post: Balance is decreased by the specified amount
     * @param n The amount to withdraw
     * @throws IllegalArgumentException if the specified amount is < 0
     * or > balance of account
     */
    public void withdraw(int n);
}
```

För att testa att en klass uppfyller sin specifikation måste man skriva programkod som anropar metoderna i olika situationer och kontrollera att resultatet blir det förväntade. Det är viktigt att man noga tänker igenom vilka olika fall som måste testas. För klassen BankAccount finns t.ex. följande fall som bör kontrolleras:

- `getBalance()` ska ge resultatet 0 för ett nyskapat konto
- `deposit(n)` följt av `getBalance()` ska ge resultatet n
- `deposit(t1); withdraw(t2); ...; deposit(tn);` (med legala parametervärden för uttag) följt av `getBalance()` ska ge resulatet t1-t2...+tn
- `withdraw(n);` där `n > getBalance()` ska generera `IllegalArgumentException`
- ....

De nämnda testfallen är bara en delmängd av alla fall som måste testas för att man ska veta att klassen uppfyller sin specifikation. Det är viktigt att testfallen också omfattar "onormala" fall och gränsfall. För bankkontoklassen är ett sådant fall att man försöker ta ut mer pengar än vad som finns. Enligt specifikationen ska då `IllegalArgumentException` genereras. Ytterligare onormala fall är att försöka sätta in eller ta ut ett negativt belopp. För en klass som hanterar listor kan intressanta gränsfall vara att ta bort det enda elementet i en lista eller att sätta in ett element allra först eller sist i listan.

2.1 Enhetstestning med JUnit i Eclipse

Ett sätt att genomföra testning enligt ovan är att skriva en `main`-metod i den klass som ska testas och låta den innehålla de kombinationer av metodanrop som motsvarar de olika testfallen. För att förenkla testförfarandet, speciellt när många klasser ska enhetstestas, kan man ha hjälp av ett verktyg. Vi ska här beskriva ett sådant verktyg, JUnit. I JUnit kan man samla alla test som rör en viss klass i en testklass och utföra alla tester och få en grafisk presentation av resultatet.

Till varje klass som ska testas med hjälp av JUnit skriver man en motsvarande testklass som kan se ut så här:

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class TestMyClass {

    @Before
    public void setUp()  {
        ...
    }

    @After
    public void tearDown()  {
        ...
    }
```

```
    @Test
    public final void testXXX() {
        ...
    }

    @Test
    public final void testYYY() {
        ...
    }
}
```

Metoderna `setUp` och `tearDown` är annoterade med `@Before` respektive `@After`. Metoder med dessa annotationer anropas av ramverket före, respektive efter varje test i testklassen. I `setUp` kan man därför initiera det tillstånd man vill utgå ifrån i varje testfall, t.ex. skapa en instans av den klass som ska testas. I `tearDown` kan man "städa upp" efter varje test t ex genom att frigöra resurser som man allokerat i `setUp`. Det är inte nödvändigt att ha med metoderna `setUp` och `tearDown`. Man kan i stället skapa instans(er) av den klass man vill testa i varje enskild testmetod.

För varje test man vill utföra skriver man en metod som annoteras med `@Test`. Dessa metoder kommer att köras av ramverket. Metoderna `testXXX, testYYY`,... i exemplet ovan ska alltså innehålla de sekvenser av metodanrop som motsvarar testfall man vill genomföra. Observera att testmetoderna måste vara publika för att ramverket ska kunna anropa dem.

För att informera testramverket om utfallet finns ett antal statiska metoder i ett paket `org.junit.Assert`. Dessa kan användas i testmetoderna för att kontrollera att resultat efter anrop av metoder är det förväntade. En del av dessa metoder räknas upp nedan. För fullständig förteckning se dokumentationen av klassen `Assert` se dokumentation an jUnit på nätet.

```
assertTrue(String message, boolean condition)
assertEquals(String message, TYPE expected, TYPE actual);
assertNotNull(String message, Object obj);
assertNull(String message, Object obj);
assertSame(String message, Object expected, Object actual);
fail(String message);
```

Assert betyder ungefär att hävda, påstå. I metoderna ovan är `message` det meddelande som används för att generera felmeddelande i det grafiska gränssnittet vid körning av testfallet om utfallet inte är det förväntade. `TYPE` är någon av typerna `boolean, byte, int, short, long, float, double` eller `Object`. Om påståendena inte är sanna så genererar ramverket meddelanden om att testfallet inte gav förväntat resultat. `fail`-metoden används när man utan att behöva kontrollera någonting vet att man misslyckats.

2.2 Exempel på testklass i JUnit 4

Vi visar som exempel test av bankkontoklassen med hjälp av JUnit. Vi utgår ifrån att vi har ett projekt i Eclipse med klassen `BankAccount` enligt ovan.

Man skapar en testklass på följande sätt i Eclipse:

1. Markera med högerknappen det paket där du vill placera testklassen och välj New -> JUnit Test Case. Då öppnas ett dialogfönster.

2. Överst i dialogfönstret finns knappar för att välja om man vill skapa en testklass för en äldre version av JUnit eller en testklass för den senaste versionen JUnit 4. Markera JUnit 4.

3. Fyll i det namn du vill ge din testklass, t ex `BankAccountTest`, i textrutan Name.

4. Kryssa i att du vill generera metoderna `setUp()` och `tearDown()`.
5. Fyll i namnet på den klass som ska testas. Om denna klass befinner sig i ett annat paket i projektet måste du kvalificera namnet med paketets namn. Om du valt att placera en testklass för bankkontoklassen (som finns i paketet `bank`) i ett annat paket i projektet anger du alltså `bank.BankAccount`. Om du valt att ha testklassen i samma paket som klassen `BankAccount` behöver du bara ange `BankAccount`
6. Klicka på Next i dialogfönstret
7. Markera de metoder i `BankAccount` du vill testa, vilket vanligtvis är alla metoder.
8. Klicka på Finish.
9. Om det är den första testklassen som skapas i projektet så kommer det att öppnas ett nytt dialogfönster med texten ”JUnit 4 is not on the build path. Do you want to add it?”.
   - Låt alternativet ”perform the following action: Add JUnit library to the build path” vara markerad och klicka på OK. Då stängs dialogfönstren.

Nu öppnas din nyskapade testklass i editorn. Det finns stubbar för metoderna `setUp` och `tearDown`. Dessutom finns det stubbar för ett antal testmetoder, en för varje metod i klassen `BankAccount`, som du markerat enligt punkt 7 ovan. Man vill naturligtvis i allmänhet ha mer än en test av varje metod i den klass som ska testas. Ytterligare test skapas enkelt genom att kopiera en testmetod, klistra in kopian i testklassen och byta dess namn och innehåll.

Alla testmetoder innehåller en rad:

```
fail("Not yet implemented");
```

Denna rad ska ersättas med din egen testkod. Redan innan du gör detta kan du köra testen. Markera tesklassen med högerknappen och välj Run As -> JUnit Test.

Följande klass innehåller testmetoder motsvarande några av de fall som behöver testas:

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class BankAccountTest {
    private BankAccount theAccount;

    @Before
    public void setUp() throws Exception {
        theAccount = new BankAccount();
    }

    @After
    public void tearDown() throws Exception {
        theAccount = null;
    }

    @Test
    public final void testGetBalance() {
        assertEquals("New account, balance not 0", 0, theAccount.getBalance());
    }

    @Test
    public final void testDeposit() {
        theAccount.deposit(100);
```

```
        assertEquals("Wrong balance after deposit", 100, theAccount.getBalance());
    }

    @Test
    public void testWithdraw() {
        theAccount.deposit(100);
        theAccount.withdraw(20);
        assertEquals("Wrong balance after withdraw", 80, theAccount.getBalance());
        theAccount.withdraw(80);
        assertEquals("Wrong balance after withdraw", 0, theAccount.getBalance());
    }

    @Test(expected=IllegalArgumentException.class)
    public final void testOverDraft() {
        theAccount.deposit(100);
        theAccount.withdraw(200);
    }
}
```

I testklassen deklareras ett attribut `theAccount` som är det konto som testas. I metoden `setUp` skapas före varje test ett nytt konto som `theAccount` refererar till. I `tearDown` tas det bort (egentligen onödigt här eftersom `setUp` ändå skapar ett nytt före nästa test).

Den första testen `testGetBalance` kontrollerar att ett nytt konto har behållningen 0. Om så inte är fallet kommer ett meddelande ”New account....” att visas av ramverket. Metoden `testDeposit` kontrollerar att behållningen är korrekt efter en enda insättning.

Lägg märke till hur det sista testet `testOverDraft` testar om det verkligen genereras Exception vid försök till att ta ut för mycket pengar. Annotationen `@Test` har i detta fall fått attributet `expected` med värdet `IllegalArgumentException.class`. Om raderna i `testOverDraft` *inte* genererar `IllegalArgumentException` misslyckas testet.

Man exekverar alla testmetoderna i testklassen `BankAcounmtTest` i Eclipse genom att markera klassen med högerknappen och välja Run As -> JUnit Test.

Testresultatet visas i Eclipse enligt Fig. 1. När testen exekveras rapporterar JUnit vilket/vilka testfall som eventuellt gick fel. Härav kan man oftast dra slutsatser om vilken/vilka metoder det är i den testade klassen som är felaktiga. Skulle t ex testfallet `testGetBalance` ovan ge felmeddelande så måste något vara fel i antingen konstruktorn i `BankAccount` eller i metoden `getBalance`. Man får i så fall försöka korrigera och köra testerna på nytt tills man känner sig nöjd med funktionaliteten i klassen. Eventuellt kommer man under denna process på ytterligare test som bör utföras. I så fall får man komplettera testklassen med fler testmetoder.

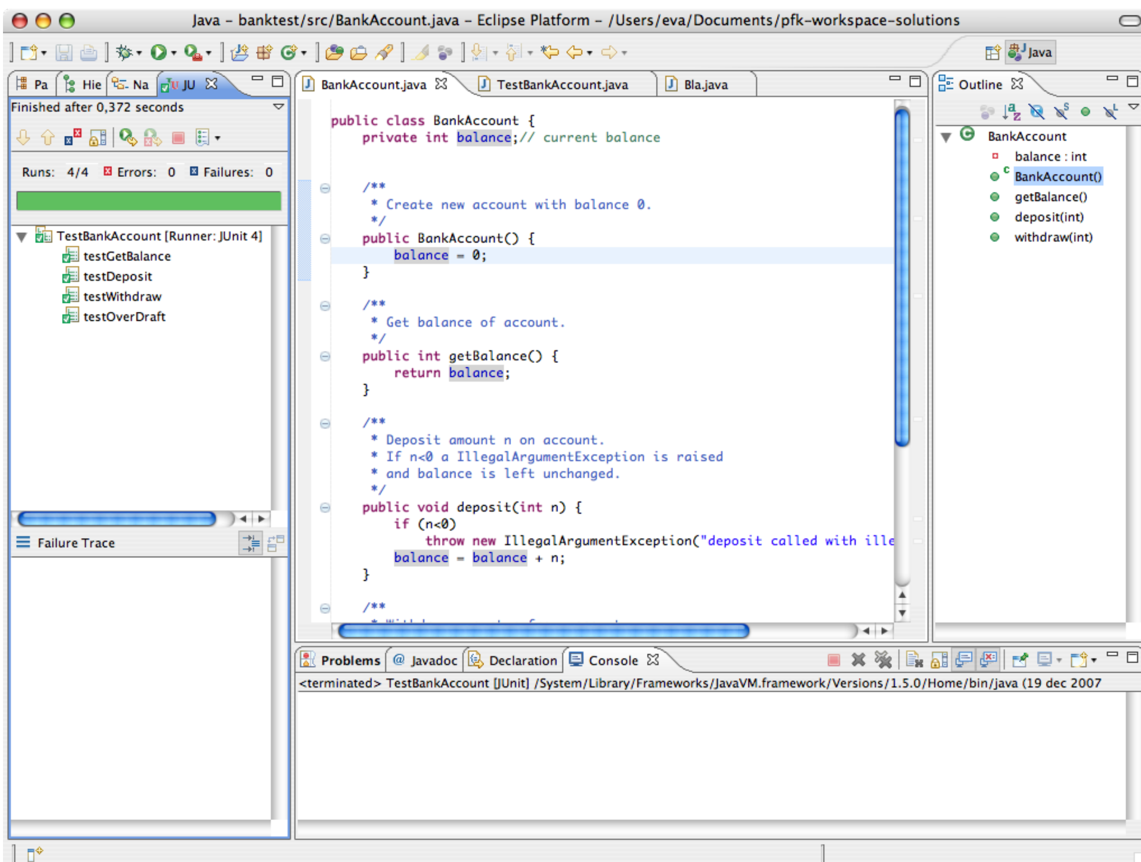
3 Prova JUnit

Det finns ett förberett arbetsområde för laborationerna som kan hämtas från kursens hemsida. I detta arbetsområde finns, förutom de sex laborationsprojekten, ett projekt med namnet `banktest`, som innehåller en färdig implementering av bankkontoklassen och dess testklass enligt ovan.

Exekvera `BankAccountTest` genom att markera klassen och välja Run As -> JUnit Test.

Alla tester går igenom. Verktyget rapporterar att 4 Runs genomförts (det finns fyra testmetoder i klassen `BankAccountTest`) samt att antalet Failures och Errors är 0. Dessutom visas grönt ljus. Se Fig. 1.

JUnit skiljer mellan två sorters fel; Failures och Errors. Failures är den typ av fel som upptäcks genom `assert`-satserna. Errors är oförutsedda fel som kan bero på att man skrivit kod som leder till exekveringsfel i klassen som testas eller i testmetoderna, t.ex. råkar ut för att få `ArrayIndexOutOfBoundsException` eller dylikt.

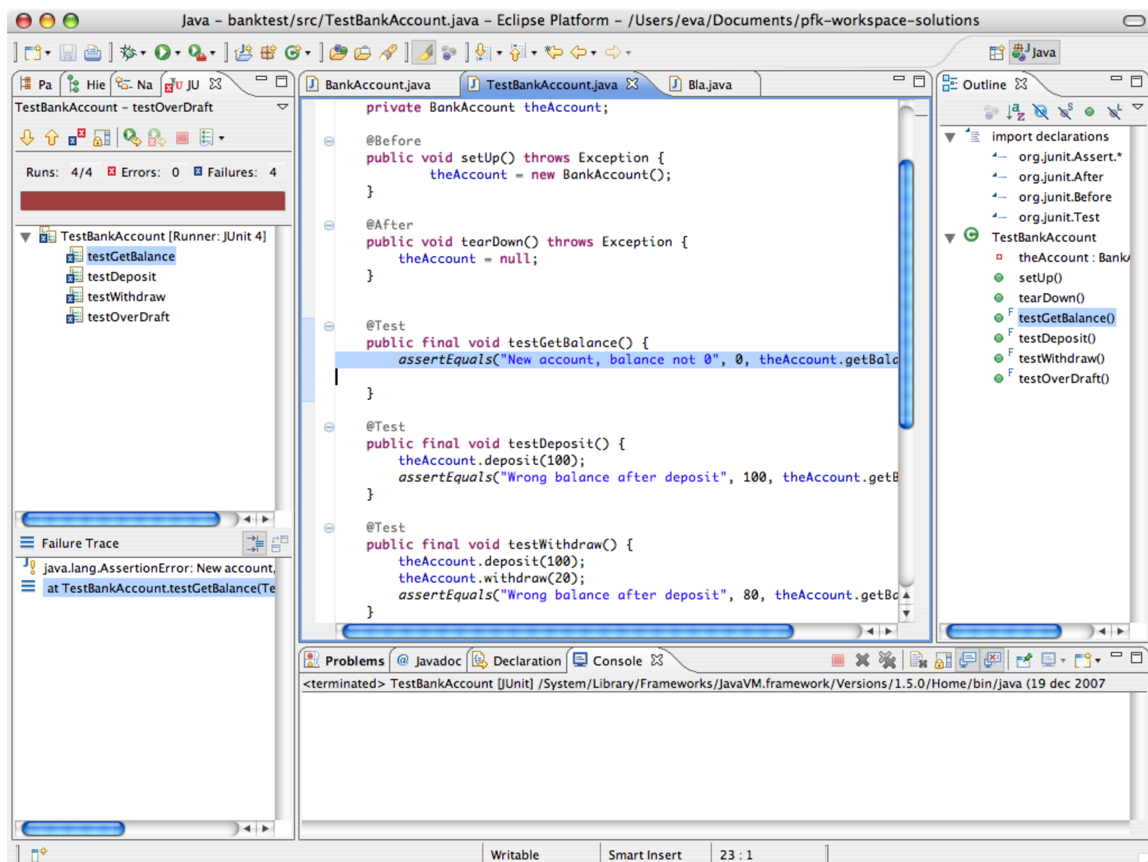


Figur 1: Resultat efter exekvering av 4 test avseende klassen `BankAccount`

För att få se hur det blir när test går fel kan du introducera något fel i klassen `BankAccount`. Låt t.ex. tillgången på ett nyskapat konto bli 100 genom att ändra i konstruktorn i klassen. Kör sedan testerna på nytt. Nu rapporteras inte mindre än 4 failures. Se Fig 2. Alla fyra testen ger fel resultat. Genom att markera ett av de test som gått fel får man i den undre textrutan det genererade felmeddelandet för den markerade metoden. Man får också uppgift om på vilken rad felet inträffat. Dubbelklickar du på raden som anger var felet inträffat så markeras motsvarande rad i testklassen i editorfönstret.

4 JUnit utanför Eclipse

JUnit kan också användas utanför Eclipse. Information om hur man gör finns på webbsidan `https://github.com/junit-team/junit/wiki/Test-runners`.



Figur 2: Testklassen `BankAccountTest` med 4 testfall har körts. Fyra fel (failures) rapporteras. Det första test som gick fel är markerat (`testGetBalance`). Ett meddelande som genererats från `assert`-satsen i `testNewAccount` visas i den undre textrutan tillsammans med en uppgift om på vilken rad felet inträffade.