Peter Lindberg
Computer Programmer, Oops AB
mailto:peter@oops.se
http://oops.se/

Några principer för effektiv enhetstestning

Enhetstester (“unit tests”) är en central del av extremprogrammering (XP). Man skriver enhetstester före man skriver “produktionskod” och fokuserar på en mindre portion funktionalitet åt gången.

Syftet med enhetstester är att möjliggöra snabba (och säkra) förändringar av produktionskod. Enhetstester kan också bidra till att öka kvaliteten i ett system, samt göra att man kan skjuta designbeslut framför sig och fatta dem först i det ögonblick då implementationsarbetet påbörjas (med vissa undantag då man behöver göra en “spike” eller ha en designdiskussion innan).

Enhetstestning kan dock motverka snabb förändring och skapa en falsk känsla av trygghet om det inte görs på rätt sätt. En viktig princip är att varje testfall ska testa en individuell aspekt av en klass istället för att testa helheten. Detta är något som blir mer uppenbart ju komplexare en klass är: förändringar kräver uppdateringar av samtliga testfall för klassen. En annan princip är att en klass måste ha så få beroenden som möjligt till kringliggande klasser, eftersom testfallen snabbt ökar i komplexitet för varje extra objekt som behöver skapas för att testa en klass.

Jag ska här gå igenom några viktiga principer för effektiv enhetstestning.

Skriv testerna först

Vi utgår från principen att skriva testerna först. Med utgång från testfallen är det rätt uppenbart att produktionskoden tar sig en form som gör den mer testbar. Du skriver ett testfall för en första portion funktionalitet, kompilerar och får kompileringsfel. Du skriver tillräckligt med kod för att få kompileringen att gå igenom, kör testerna och får testfel, varpå du skriver kod tills testerna går igenom.

Jämfört med när du skriver tester i efterhand så har du inte som mål att testerna ska gå igenom. Du vet inte ens hur testerna kommer se ut. Varje metod du skriver fungerar förvisso i sitt sammanhang, men att skriva ett testfall som verifierar just den metodens ansvarsområde blir vanligtvis svårare.

För att illustrera detta kan du tänka dig att du behöver skriva kod för att skicka epost till en epostadress, med ett ärende och en text. Du tänker: “Ok, jag behöver en klass som har en metod för att skicka epost och som tar mottagaradress, ärende och text som argument.” Du börjar skriva koden (med hjälp av JavaMail) och kommer fram till något i stil med detta:

```
public class Emailer {
     public static void sendEmail(String emailAddress, String
               subject, String text) throws MessagingException {
          Properties properties = new Properties();
          properties.put("mail.smtp.host", "mail.oops.se");
          Session session
               = Session.getDefaultInstance(properties, null);
          MimeMessage message = new MimeMessage(session);
          message.setSentDate(new Date());
          message.setFrom(new InternetAddress("noreply@oops.se"));
          InternetAddress[] recipients = new InternetAddress[]
               {new InternetAddress(emailAddress)};
          message.setRecipients(Message.RecipientType.TO,
               recipients);
          message.setSubject(subject);
          message.setText(text);
          Transport.send(message);
     }
}
```

Detta fungerar utmärkt! Men föreställ dig att du i efterhand ska skriva ett test för denna klass. Du behöver verifiera att epostmeddelandet har rätt ärende och text, samt att det skickas till rätt mottagare. Hur gör du?

Förmodligen resonerar du så att du behöver göra omfaktorisering (“refactoring”) så att du kan testa `MimeMessage`-objektet innan det skickas iväg. I detta fall kanske det är acceptabelt att göra omfaktorisering utan att ha testfall uppsatta för klassen, men hade det varit fråga om en mer komplex klass kanske det hade krävt mycket manuell testning av systemet för varje omfaktoriseringssteg om inte tester fanns.

Om du däremot skriver testerna först så kanske det går till såhär: “Ok, jag ska skriva en klass som skickar epost. Jag behöver kunna verifiera att epostmeddelandet har rätt mottagare, ärende och text. Jag börjar med att testa att mottagaradressen är korrekt.” Du börjar med testerna:

```
public void testHasRightRecipient() throws MessagingException {
     Emailer emailer = new Emailer();
     emailer.setRecipient("jeff_bridges@hotmail.com");
     MimeMessage mimeMessage = emailer.createMimeMessage();
     InternetAddress[] recipients = (InternetAddress[])
          message.getRecipients(Message.RecipientType.TO);
     assertEquals("jeff_bridges@hotmail.com",
          recipients[0].getAddress());
     assertEquals(1, recipients.length);
}
```

För att uppfylla testet behöver du skapa en metod som sätter mottagaradress samt en som skapar ett `MimeMessage`-objekt:

```
public void setRecipient(String recipient) {
     this.recipient = recipient;
}
```

```
MimeMessage createMimeMessage() throws MessagingException {
     Properties properties = new Properties();
     properties.put("mail.smtp.host", "mail.oops.se");
     Session session
          = Session.getDefaultInstance(properties, null);
     MimeMessage message = new MimeMessage(session);
     InternetAddress[] recipients = new InternetAddress[]
          {new InternetAddress(recipient)};
     message.setRecipients(Message.RecipientType.TO, recipients);
     return message;
}
```

I detta skede går ditt testfall igenom utan fel och du fortsätter med nästa steg: att verifiera att ärendet är korrekt:

```
public void testHasRightSubject() throws MessagingException {
     Emailer emailer = new Emailer();
     emailer.setSubject("Hello, world!");
     MimeMessage mimeMessage = emailer.createMimeMessage();
     assertEquals("Hello, world!" mimeMessage.getSubject());
}
```

Sedan skriver du koden för att uppfylla testfallet, vilket bara är en sättmetod och ett tillägg i slutet av metoden `createMimeMessage()`:

```
public void setSubject(String subject) {
     this. subject = subject;
}

MimeMessage createMimeMessage() throws MessagingException {
     ...
     message.setSubject(subject);
     return message;
}
```

Sedan gör du samma sak med texten i epostmeddelandet. För att göra klassen bekväm att använda skriver du en likadan statisk metod som i exemplet utan tester:

```
public static void sendEmail(String emailAddress, String
          subject, String text) throws MessagingException {
     Emailer emailer = new Emailer();
     emailer.setRecipient(emailAddress);
     emailer.setSubject(subject);

     emailer.setText(text);
     emailer.sendEmail();
}
```

Men vänta! Ska vi inte skriva ett test för metoden `sendEmail()`? Nej, det behövs inte eftersom det enda den innehåller är ett anrop till JavaMail för att skicka epost-meddelandet. Vi skulle bara testa att JavaMail och SMTP-servrarna fungerar, vilket dessutom är krävande. (Se nedan för mer om detta.)

```
public void sendEmail() throws MessagingException {
     Transport.send(createMimeMessage());
}
```

Det jag har försökt illustrera med detta exempel är hur testarbetet skiljer sig mellan när tester skrivs i efterhand och när de skrivs i förhand: att koden blir testbar när testerna skrivs först och att arbetet att i efterhand lägga till tester tar mer tid.

Skriv enbart kod för att få testfall att gå igenom

Denna princip kan tyckas vara densamma som att man ska skriva testerna först, men det finns en viktig skillnad: i exemplet ovan så skulle det vara "förbjudet" att skriva koden för att sätta ärende när det bara finns testfall för att testa att mottagaren är korrekt. Även om du tycker att du vet att testerna kommer gå igenom när du väl skriver dem, så är det förbluffande lätt att i mer komplexa sammanhang smyga in små fel när man inte har satt upp testfallet i förhand.

Fråga kompilatorn/testerna

Många reagerar negativt när de ser någon som först skriver enhetstester och därefter startar kompilatorn trots att det är uppenbart att det inte kommer gå igenom, eller exekverar testfallen fastän vem som helst kan se att de inte kommer lyckas.

Men detta handlar egentligen om att hela tiden låta kompilatorn och testverktyget berätta för en vad som behöver göras: "Ok, det finns ingen `getFullName()`, så jag får väl skriva den då." eller "Hm, `getAge()` ska returnera 57 men jag får -1, så jag får fixa den." Och förr eller senare kommer du stöta på ett fall då `getFullName()` redan finns eller `getAge()` redan returnerar korrekt ålder trots att du trodde att den inte skulle det.

Denna princip innebär också att man "fångar sig själv" när man märker att man sitter och försöker tänka igenom hur programmet kommer fungera, istället för att bara skriva ett testfall och se vad som händer. Det har hänt att jag har suttit och tänkt ett tag över hur programmet skulle bete sig i ett visst sammanhang, sedan tagit fram ett papper och försökt rita för att lättare kunna förstå – och först därefter (påmind av mig själv eller min parprogrammeringspartner) insett att jag bara behövt sätta upp ett enkelt testfall och se vad resultatet blir.

Dessa tre principer – 'Skriv testerna först', 'Skriv enbart kod för att få testfall att gå igenom' samt 'Fråga kompilatorn/testerna' – bidrar till att skapa ett arbetssätt som "drivs på" av testfallen, av kompilatorn och av testverktyget. Jag själv tycker det bidrar till att skapa "flow" i arbetet, genom att man kan ta diskussion och eftertanke först när koden "ber om det".

Mindre portioner

Exemplet ovan visar också principen att fokusera på en mindre portion funktionalitet åt gången: vi koncentrerar oss i tur och ordning på epostmottagare, ärende samt text och bryr oss i varje sammanhang inte så mycket om resten. I mer komplexa sammanhang än i exemplet ovan gör detta att metoder blir mer oberoende av varandra, vilket minimerar den omfattning som en klass påverkas av en ändring.

Dessutom vill jag påstå att enhetstestning bidrar till att metoderna blir mer sammanhållna (“cohesive”), dvs att deras ansvarsområden blir tydligare. Man blir mindre benägen att låta en metod göra något som ligger utanför ansvarsområdet, eftersom det blir tydligare att något inte är som det borde när man skriver testfallet: *ser* det konstigt ut så *är* det konstigt. Ett hårddraget exempel:

```
public void testChangeLastNameToCostelloWhenFirstNameIsElvis() {
     Person person = new Person();
     person.setLastName("Presley");
     assertEquals("Presley", person.getLastName());
     person.setFirstName("Elvis");
     assertEquals("Costello", person.getLastName());
}
```

För att istället knyta an till `Emailer`-exemplet ovan: skulle klassen utökas med möjligheten att kunna skapa X.400-meddelanden så skulle det förmodligen resultera i en ny metod istället för att låta `createMimeMessage()` beroende på sammanhang ibland returnera ett `X400Message`-objekt.

Ensam är stark

Det är en grundprincip inom god design att minska antalet beroenden en klass har till sin omgivning. På engelska kallas detta “low coupling”. I enhetstestningssammanhang blir detta kanske tydligare än annars: arbetet att sätta upp nya testfall blir mer omfattande för varje beroende och förhoppningsvis medverkar principen ‘Skriv testerna först’ till att minska antalet beroenden.

Då tester ska sättas upp i efterhand för en klass med för många beroenden kan arbetet bli oproportionerligt stort för ändringar som ska göras. En stor mängd objekt ska sättas upp för att objektet som ska testas ska fungera. Består klassen dessutom (liksom i det första `Emailer`-exemplet ovan, då testerna skulle skrivas i efterhand) av svårtestade metoder är situationen ännu värre.

Testa bara det som behöver testas

I exemplet ovan skrev jag att man inte behövde testa metoden `sendEmail()`. Anledningen till detta är att det enda som testfallet skulle testa vore att JavaMail fungerar, vilket är en tredjepartsprodukt och ligger utanför det område som omfattas av våra tester. I sammanhanget räknas också att det inte är något som är trivialt att testa – man skulle kanske få sätta upp någon sorts “mock mailserver” så att man kan få tag i det “ivägsända” meddelandet, men det skulle ändå inte testa hela kedjan.

Skulle det dock vara trivialt att testa `sendEmail()` så kan man för all del testa det, men enhetstestning handlar alltid om en gränsdragning för testernas omfattning. Det är en avvägning som baserar sig på hur mycket arbete som krävs för att testa något, hur sannolikt det är att det kan uppstå fel, hur allvarliga fel är som kan uppstå, osv.

Principerna ‘Skriv testerna först’ och ‘Skriv endast kod för att få testfall att gå igenom’ innebär att man ibland kommer skriva tester som man i andra sammanhang inte skulle skriva. Mest uppenbart är det med sättmetoder – ibland kommer du skriva testfall som detta:

```
public void testAmount() {
     Transaction transaction = new Transaction();
     transaction.setAmount(4000.0);
     assertEquals(4000.0, transaction.getAmount(), 0.0);
}
```

Detta skulle kunna vara ett fall där `Transaction`-klassen precis har skapats och inte ännu används av någon klass. Utgår du däremot från `Account`-klassen och testar beräkning av saldo så kanske du slipper skriva testfall för `setAmount()`, eftersom det i sammanhanget är för trivialt:

```
public void testBalance() {
     Transaction transaction = new Transaction();
     transaction.setAmount(4000.0);
     Account account = new Account();
     account.addTransaction(transaction);
     assertEquals(4000.0, account.getBalance(), 0.0);
}
```

Detta med gränsdragning för enhetstestning är någonting som ditt team behöver enas om. Frågor om huruvida det är det ok att låta bli att testa databaskopplingen, tredjepartsbibliotek, användargränssnitt, osv, måste ventileras och bedömas. Kanske kan man inom vissa områden på något sätt kompensera för saknaden av enhetstester?

Generellt sett kan man säga att enhetstestning är effektiv så länge som testarbetet inte blir för krävande. En miljö där testerna skriver till en databas som återställs inför varje enskilt testfall, inbillar jag mig är svår att få effektiv: enstaka förändringar i en klass som kräver förändringar i tabellstrukturen kan kräva stora uppdateringar av testklasserna.

Avslöja buggen innan du fixar den

Principen ‘Skriv enbart kod för att få testfall att gå igenom’ medför att du när du ska fixa en bugg måste börja med ett nytt testfall (eller modifiera ett redan existerande). Om vi antar att du skriver ett nytt testfall så handlar det om att återskapa det sammanhang i vilket buggen inträffade, samt skriva testet så att buggen avslöjas genom att du får ett förväntat fel (“failure”) när du exekverar testerna:

```
public void testNoNullPointerExceptionInGetInitials() {
     Person person = new Person();
     person.setFirstName("Jeff");
     person.setLastName(null);
     try {
          person.getInitials();
     } catch (NullPointerException unexpected) {
          fail("Shouldn't throw NullPointerException");
     }
     assertEquals("J?", person.getInitials());
}
```

Detsamma gäller buggar i tredjepartskod: skriv ett test för att avslöja buggen. Då kanske du inte kan fixa buggen utan får vänta på en ny version, men förhoppningsvis kan du skriva en "workaround" så att testfallet åtminstone går igenom.

Sammanfattning

I inledningen skrev jag att enhetstestning kan motverka snabb förändring samt ge en falsk känsla av säkerhet. Jag har försökt visa hur tester skrivna i efterhand kan göra arbetet med systemet trögare, hur testfall som alltid testar helheten kan göra att testsviten tar för lång tid att exekvera samt att testkoden kräver mer underhåll vid förändring av systemet – kort sagt: hur testsviten och önskan att underhålla och utöka den kan bli en fotboja.

För att motverka detta handlar det främst om att följa principerna 'Skriv testerna först' och 'Mindre portioner' och att aldrig glömma att det högsta målet med enhetstestning är att möjliggöra snabba förändringar i systemet. Därtill är det viktigt att diskutera gränsdragningen för testernas omfattning (se principen 'Testa bara det som behöver testas').

När det gäller enhetstester som en falsk känsla av säkerhet så motverkas det av att följa principen 'Skriv testerna först' samt att man påminner sig om poängen med 'Skriv enbart kod för att få testfall att gå igenom'. Det handlar om "code coverage", dvs att de tester som finns verkligen kommer åt samtliga delar av produktionskoden som behöver testas.

Avslutningsvis vill jag lätta upp denna text med att påminna om att enhetstestning när man gör det på "rätt" sätt är fantastiskt roligt. *Man får programmera hela tiden!* Och till sist en liten övning i "code coverage" till dig: i exemplet med `Emailer`-klassen finns det något som jag har glömt testa och som borde testas – vad?

Appendix: Föreslagen läsning

JUnitTest Infected: Programmers Love Writing Tests, Kent Beck. Den ursprungliga artikeln om enhetstestning omarbetad för JUnit. Beskriver såväl hur enhetstestning går till som hur JUnit fungerar. *http://junit.org* (Medföljer i JUnit-distributionen.)

Unit Tests. En sida med diskussioner om enhetstestning och testaförstprogrammering. Wiki-sajten är ett öppet diskussionsforum där vem som helst kan redigera en sida. Det finns oerhörda mängder intressant information om man bara tar sig tid att utforska sajten. *http://c2.com/cgi/wiki?UnitTests*

Endo-Testing: Unit Testing with Mock Objects, Tim Mackinnon, Steve Freeman, Philip Craig. En artikel som definierar tekniken att arbeta med "mock objects". Mycket läsvärd! *http://www.sidewize.com/company/mockobjects.pdf*

Diskutera denna artikel på det svenska XP-communityts egen Wiki.

http://oops.se/cgi-bin/wiki?ExtremProgrammering