Endo-Testing: Unit Testing with Mock Objects

Tim Mackinnon, Steve Freeman, and Philip Craig. Presenterad på *eXtreme Programming and Flexible Processes in Software Engineering - XP2000.*

Syfte med artikeln

Författarna förespråkar användandet av vad de kallar “Mock Objects” (mocks eller mockobjekt) för att förenkla testing av mjukvara.

Presentation av arbetet

Artikeln ger en detaljerad beskrivning av hur mocks kan användas vid enhetstestning. Inflikat i texten visas flera kodexempel på det de beskriver.

Resultat av arbetet

Författarna drar slutsatser utifrån sina egna erfarenheter av att arbeta med mocks.

Om mockobjekten blir för komplexa, till exempel genom att en mock kallar på en annan mock, kan det indikera att testet inbegriper mer än en enskild del i mjukvaran eller att mjukvaran som testas behöver refaktoreras.

En fördel med mocks är att det går att lägga assertions direkt i dem istället för att återskapa det i varje testfall. Men om dessa assertions fallerar oväntat så kan det indikera att antingen mjukvaran som testas eller mockobjektet behöver refaktoreras.

Mocks minskar behovet av tillgång till den testade mjukvarans struktur.

När mocks används uppmuntrar det till att skriva kod som tar in objekt istället för att använda singletons.

Mocks är bra att använda för att visa hur effektiva enhetstest skrivs.

Författarna kom fram till en generell arbetsmetod för mocks:

- Skapa mockinstanser
- Sätt states i mockobjekten
- Sätt expectations i mockobjekten
- Kör koden som ska testas med mockobjekten som parametrar
- Verifiera mockobjekten

Begränsningar med mocks

Mocks är främst användbara vid enhetstester då små delar av mjukvaran testas. Funktionstester kräver mer logik än mockobjekten kan leverera (utan att duplicera mjukvaran som ska testas).

Komplexa objekt kan vara svåra att ersätta med mocks.

Mocks kan bara användas om mjukvaran som ska testas är byggd med interfaces (under förutsättning att det är ett statiskt typat språk).

Omdöme av artikeln

Artikeln har skrivits av skaparna av Mock Objects och i samtliga andra artiklar om Mock Objects jag stött på så har denna varit med som källa. Artikeln har citerats 222 gånger enligt Google Scholar. Artikeln presenterades först på en konferens om eXtreme Programming och har därefter tryckts i boken *Extreme programming examined*.

Mock Roles, Not Objects

Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. I *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 236-246. ACM, 2004.

Syfte med artikeln

Artikeln är en uppföljning av artikeln ovan, *Endo-Testing: Unit Testing with Mock Objects*, skriven 4 år senare. Författarna visar hur deras arbete med Mock Objects har utvecklats under åren och hur de tänker kring begreppet nu (läs: då, 2004). De presenterar även ett mockramverk de skapat, *jMock*.

Presentation av arbetet

Även den här artikeln bygger till största delen på en genomgång av hur man använder sig av mocks. De går in ännu mer i detalj och visar med hjälp av ett komplett testfall hur man använder sig av *jMock* för att bygga sina tester med mocks.

Resultat av arbetet

En av de främsta resultaten tycks vara att författarna börjat utveckla enligt en metod de kallar *Need-Driven Development*. Kortfattat innebär den att göra tester utifrån-och-in. På det sättet växer den kompletta hierarkin för programmet fram på ett naturligt sätt. Tänk att mjukvaran som ska testas är en virtuell bil. Då kan testningen ske enligt följande steg:

1. Testa *Volvo.undvikKrock()*. En Volvo instansierar i sintur troligtvis en Bil. Skapa ett mockobjekt för Bil och kalla på lämplig metod: mock*Bil.tryckPåTutan()*.
2. Testa *Bil.tryckPåTutan()*. En Bil instansierar en Tuta. Skapa ett mockobjekt med metoden: mock*Tuta.aktivera()*.
3. Testa *Tuta.aktivera()*. Den här metoden skriver bara ut en texten “TUT!” på skärmen och behöver inte mockas utan kan testas direkt.

Att använda mocks enligt *Need-Driven Development* resulterar i platta hierarkier. Det leder till att välkända problem, som exempelvis *Fragile base class*[1], lättare kan undvikas.

Författarna upplevde att deras erfarenhet var att de följde en bakvänd ordning när de skrev sina tester. De började med objektets metodanrop, identifierade tjänster som objektet krävde och representerade dessa med mocks, skapade kontexten för testet, definierade eventuella slutkrav och verifierade att mockobjekten använts som tänkt av metodanropet.

Begränsningar med jMock

jMock är inte anpassat att användas med exempelvis integrationstestning eller metoder som är klasslösa (procedurell programmering).

Omdöme av artikeln

Åter igen är artikeln skriven av skaparna av Mock Objects. Artikeln verkar dock inte ha haft samma genomslag som den första, endast 95 citat enligt Google Scholar. Både Freeman och Pryce är erkända namn inom Test-Driven Development.

[1] https://en.wikipedia.org/wiki/Fragile_base_class

How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study

Joseph Lawrence, Steven Clarke, Margaret Burnett, and Gregg Rothermel. I *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pp. 53-60. IEEE, 2005.

Syfte med artikeln

Att studera hurvida mjukvarutester skrivna av professionella utvecklare blir mer effektiva när visualiseringsverktyg för Code Coverage, specifikt blockkodcoverage, används samt hurvida utvecklare som använder dessa verktyg har en annan strategi när de utveckar test än de som inte använder sig av Code Coverage-visualisering.

Metod för insamling av data

Innan den riktiga studien påbörjades gjordes en pilotstudie där 4 utvecklare ingick. Utvärderingen av pilotstudien ledde till följande metod.

30 professionella C#-utvecklare delades in i två lika stora grupper, varav den ena gruppen använde sig av Code Coverage-visualisering men inte den andra. Samtliga deltagare fick först besvara ett frågeformulär, därefter fick de ett antal mindre program som de skulle skriva tester för. Programmen innehöll en del medvetna fel. Efter testfasen fick de besvara ytterligare ett frågeformulär.

Under skapandet av testerna observerades utvecklarna av författarna och audio samt video spelades in.

Resultat av arbetet

Författarna kunde inte finna någon större skillnad vad gällde effektiviteten i testerna mellan grupperna. Båda grupperna skapade dessutom ungefär lika många tester, däremot så skiljde sig utformingen av testerna åt mellan grupperna.

En tydlig skillnad i arbetssätt var att de som arbetade med Code Coverage-visualiseringen fortsatte att skriva tester tills att coverage var godkänd, därefter slutade de skriva tester (vilket enligt författarna var helt korrekt tillvägagångssätt).

Författarna tittade också på om det fanns någon skillnad i hur grupperna uppskattade hur stor andel fel de ansåg sig ha hittat i programmen. Båda grupperna överskattade sina testers förmåga att hitta fel men Code Coverage-gruppen visade sig ha en högre uppskattningsnivå.

Begränsningar i studien

30 personer är ett mycket litet underlag och det är svårt att dra några definitiva slutsatser utifrån det. Programmen som skulle testas var också mycket små och speglar inte reella fall. En mycket större testgrupp och riktiga, stora program skulle behövas för att bekräfta eller dementera slutsatserna i den här studien.

Studien undersöker endast effektiviten baserat på ett coveragemått, nämligen blockcoverage. Flera mått skulle kunna tillföras studien för att få ett bredare resultat.

Omdöme av artikeln

Artikeln har citerats 28 gånger enligt Google Scholar, vilket verkar högt inom ämnet. Burnett och Rothermel har runt 150 publiceringar var enligt ACM Digital Library. Burnett har fått utmärkelse flera gånger för “mest inflytanderika artikel”. Rothermel är professor i mjukvaruutveckling och sitter som editor för flera kända forskningstidskrifter.