EXAMENSARBETE VID CSC, KTH

Svensk titel

Engelsk titel

Jonas Frogvall
frjo02@kth.se

Exjobb i: exjobbsämne

Handledare: Stefan Arnborg

Examinator: Stefan Arnborg

Uppdragsgivare: Sigma

Svensk titel

Sammanfattning

Svensk sammanfattning.

Sammanfattningen på det språk rapporten är skriven på placeras först.

Engelsk titel

Abstract

Engelsk sammanfattning.

Om båda sammanfattningarna får plats på en sida, placerar du dem på samma sida. Sätt annars den andra på en ny sida.

Förord

Förordet sätter du på ny sida. Du behöver inte ha förord.

Innehållsförteckning

Problembeskrivning

”Kan man verifiera ett program - avsett för att verifiera testares testkompetens - på ett sådant sätt att en ickeprogrammerare kan avgöra att kraven på programmet är rimliga?”

Testning av programvara är fundamentalt för att verifiera att den fungerar som den skall och att den uppfyller specifikation och krav framtagna av kunden.

Men hur vet man att de test som utvecklarna tagit fram är korrekta? Om kunden inte själv är programmerare och kan läsa kod, hur vet han att de test som presenteras för honom är rimliga? Bara för att test utförts så betyder det inte att de testar rätt saker eller bekräftar önskad funktionalitet.

Hur överkommer man denna klyfta mellan utvecklare/testare och kunden? Kan man skriva kod på ett sådant sätt att det för kunden framstår som ren engelska? Om så, vilka verktyg behöver man för att realisera detta?

Jag kommer i det här arbetet att analysera och jämföra några verktyg som kan användas för att uppnå detta mål.

Bakgrund

En gång i tiden testades det saker. Det gör det fortfarande. Det här stycket behöver utvecklas något.

Testlager

Testverktyg

Verktyg för att driva test

JUnit

Robot Framework

Cucumber

Verktyg för att styra GUI

FEST Swing

Project Sikuli

UISpec4J

Bräcklighet

Det största problemet med automatiserade end-to-end-test, är att de är bräckliga. Project Sikuli är ett mycket bra exempel på hur lätt det är att ha sönder ett av dess test, utan att ändra i funktionalitet. Byter man t.ex. färg på en knapp, så kommer den gamla bilden av knappen inte längre att matcha den nya. Detta medför att mycket små ändringar kan kräva att väldigt många test skrivs om. Att just Sikuli är extra bräcklig beror ju naturligtvis på att det fullt utgår från programmets utseende. Är programmet designat på ett sådant sätt att det körs på olika språk, beroende på plattformens nationella inställningar, eller om färgscheman, teckensnitt, etc. beror på plattformen, så kommer testet gå sönder bara man kör det på en annan dator än den man designade testet på.

FEST och UISpec4J hittar komponenter på ett annat sätt, genom att antingen titta på deras namn – som då måste ha satts av den som programmerade komponenten och vara känt för den som skriver testkoden – eller genom en sökning genom alla komponenter och därefter matcha dem med olika attribut. Detta gör ju att ändring av teckensnitt eller knappfärg inte påverkar testen, såvida dessa inte är en del av matchningen i sökningar, men det gör det ju inte heller helt säkert. Programmeraren kan få för sig att döpa om en komponent, utav en eller annan anledning, och då kommer alla test som använder sig av den komponenten haverera och behöva skrivas om. En annan sak som kan ha sönder ett test är om matchningen efter en komponent helt plötsligt

genererar fler än ett alternativ. Om man tänker sig att man har en panel där man har en lista som listar anställda på ett företag. Bredvid listan har man knapparna ”Lägg till”, ”Redigera” och ”Radera”. Det kan finnas andra knappar med samma text på andra paneler i programmet, som inte syns just nu, och vill vi söka efter en knapp med texten ”Lägg till” och sedan klicka på den, så måste vi se till att även kolla att knappen är synlig, för att inte riskera att hitta andra komponenter med samma text. Vi vill antagligen försäkra oss om att vi hittar en knapp också, och inte en komponent av en annan typ. I FEST är detta ganska enkelt, då de använder sig av *Generics*.

```
public void clickButton(final String buttonText) {
    GenericTypeMatcher<JButton> textMatcher =
            new GenericTypeMatcher<JButton>(JButton.class) {

                @Override
                protected boolean isMatching(JButton button) {
                    return ((button.isShowing())
                            && (buttonText.equals(button.getText())));
                }
            };
    window.button(textMatcher).click();
}
```

Genom att deklarera redan när vi skapar objektet *textMatcher* att det är en *JButton* så behöver vi inte oroa oss för att vi får fel sorts komponent, så FEST bara kommer söka i komponenter av klassen *JButton* eller dess subklasser [Kontrollera att detta är sant]. Detta ger oss också direkt tillgång till fält som är unika för *JButton*, som t.ex. *getText()*. Sedan kontrollerar funktionen huruvida knappen är synlig och har rätt text och när vi hittat en och endast en komponent så klickar vi på den.

I UISpec4J så är det lite krångligare, då *Generics* inte används, utan man får istället själv kontrollera vad man får in för objekt.

```
public void clickButton(final String buttonText) {
    ComponentMatcher buttonMatcher = new ComponentMatcher() {

        @Override
        public boolean matches(Component component) {
            String componentText;
            try {
                componentText = ((JButton) component).getText();
            } catch (ClassCastException ex) {
                return false;
            }
            return ((componentText.equals(buttonText))
                    && (component.isShowing()));
        }
    };
    Button button = window.getButton(buttonMatcher);
    button.click();
}
```

Vad man då istället gör är att man försöker kasta komponenten till en *JButton* och misslyckas det så fångar man upp det *ClassCastException* som kastas och returnerar falskt, då komponenten uppenbarligen inte var den vi letade efter. Utöver det så är det i princip exakt samma kod som för FEST.

När det sedan visar sig att kunden vill ha separata listor för de som jobbar på Örebrokontoret och de som jobbar på Stockholmskontoret, men fortfarande vill ha dem listade på samma panel, så uppstår ett problem. I början verkar det enkelt. Programmeraren skalar ned storleken på den första listan, döper om den till Örebro och slänger in en ny listkomponent under den första och döper den till Stockholm. Han lägger sedan till tre nya knappar, ”Lägg till”, ”Redigera” och ”Radera”, precis intill Stockholmstabellen. Han binder knapparna till funktioner som interagerar med Stockholmslistan och kör testen och upptäcker att helt plötsligt så går de inte igenom. Det finns nu nämligen fler än en knapp som har texten ”Lägg till” och är synlig. Man behöver då antingen hitta en annat unikt attribut att jämföra med, eller börja använda sig av komponentnamn för sökning, istället för att söka på knappens text.

Är det samma programmerare som skriver koden som skriver testen så är det inte svårt att ge alla komponenter ett namn och sedan söka på det, men är det en helt annan person som skriver testkoden och inte känner till komponentnamnen så blir det klurigare. Han kommer antingen behöva dokumentation som listar komponenternas namn på ett överskådligt vis, eller leta efter komponentnamnen i koden. Båda alternativen är suboptimala, då de är ineffektiva och dyra [Uppbackning]. Att hitta ett ytterligare attribut som testkodsskrivaren kan se direkt på skärmen är inte heller lätt, såvida man inte gör en synlig skillnad på de båda komponenterna, t.ex. genom att färga den ena röd och den andra grön. Är knapparna lika så är det svårt att komma på andra attribut att jämföra med än texten på knappen och huruvida den är synlig. Är bara den ena knappen aktiv, så kan man förstås även kontrollera *isEnabled()*, men det skulle antagligen inte fungera i det här fallet. [Nu finns risk att jag lyckats deraila för mycket, men omskrivningar behövs väl oavsett med tiden].

Bräcklighet är ett allmänt problem för test på den här nivån (referens), och inte något som är lätt att komma runt. [Nu stannar jag här i nuläget för att inte fastna och skriva i cirklar, bättre att fortsätta skriva på något annat, när jag har farten uppe]

Litteraturlista

ANDERSSON, P. 1999. Testing algorithms. *Journal of algorithms*, Vol. 4, 1999, s. 217–223.

ANGEL E. 2011. *Computer graphics, a top down approach.* Prentice Hall. ISBN 123456789.

Modern och välkänd lärobok i datorgrafik som används vid KTHs kurser.

MORKES J. OCH NIELSEN J. 1997. *Consise, scannable, and objective: how to write for the web.* http://www.useit.com/papers/webwriting/writing.html

Artikel av bl.a. webbgurun Jakob Nielsen om hur webbsidor bör skrivas. Beskriver två experiment med läsning av webbsidor.

PETTERSSON, A. 2000. *Att skriva vetenskapliga rapporter.* Studentlitteratur. ISBN 123456789

Detta är en fejkad bok som inte finns.