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 1
Bakgrund 2
Testnivåer 3
Enhetstest 3
Integrationstest 3
Systemtest 3
Acceptanstest 3
Regressionstest 3
Testautomatisering 4
Testdriven utveckling 4
Testverktyg 5
Verktyg för att driva test 5
Verktyg för att styra GUI 5
Bräcklighet 7
Litteraturlista 9

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.

Testnivåer

Testa mjukvara kan man göra på olika nivåer. Vanligtvis är tester uppdelade i *enhetstest*, *integrationstest*, *systemtest*, och *acceptanstest* (Dustin et al. 1999; Craig & Jaskiel 2002; Burnstein 2003). Syftet med de olika testnivåerna är att undersöka och testa mjukvaran från olika perspektiv och hitta olika sorters defekter (Burnstein 2003).

Enhetstest

Mjukvarans minsta komponenter kallas för enheter. Traditionellt ses funktioner och procedurer som enheter, medan man i objektorienterade språk kan syfta till metoder och klasser/objekt. Även små komponenter eller bibliotek kan ses som enheter. Målet med enhetstester är att hitta funktionella och strukturella fel i dessa enheter.

Fel som uppstår i enhetstest är ofta enkla att lokalisera och reparera eftersom de bara rör en enhet åt gången (Burnstein 2003). Det är därmed billigast att leta efter och laga defekter på enhetstestnivå.

Integrationstest

När man kombinerar olika enheter till grupper av enheter, så får man s.k. *subsystem* eller *kluster*. Målet med integrationstest är att verifiera att dessa subsystem fungerar som de skall och att kontroll- och dataflödet mellan de olika komponenterna beter sig korrekt (Burnstein 2003).

Systemtest

Kombinerar man alla subsystemen så får man det slutgiltiga systemet, det vill säga applikationen så som kunden skulle se den, om den släpptes där och då. Systemtest validerar både funktionella och ickefunktionella kvaliteter hos systemet. Målet är att verifiera att systemet fungerar som det skall i sin sammansatta form.

Acceptanstest

Acceptanstestsnivån finns där för att kontrollera att en produkt uppfyller kundens krav. Acceptanstest borde utvecklas i samarbete med kunden för att verifiera att produkten uppfyller de mål och förväntningar som kunden har. I vissa fall är det inte möjligt att arrangera kundspecifik acceptanstest och då brukar man dela upp acceptanstesten i två faser, alfa- och betatestning. Under alfafasen så testas produkten av ev. kunder och medlemmar av utvecklingsorganet på utvecklingsorganets premisser. Efter defekter man funnit under alfafasen rättats till, så skickas produkten ut till ett urval av användare som använder produkten i de miljöer den färdiga produkten kommer köras i och rapporterar de defekter de finner (Burnstein 2003).

Regressionstest

Syftet med regressionstest är att verifiera att ändringar i mjukvaran inte har fått nya defekter att uppstå eller att element som fungerade tidigare inte upphör att fungera. Regressionstestning är inte en testnivå, utan något som kan utföras i alla nivåer. Regressionstestning är särskilt viktigt när ett system släpps flera gånger. Funktionaliteten som fanns i en tidigare utgåva skall fortfarande fungera även efter att ny funktionalitet lagts till och att verifiera detta kan vara väldigt tidskonsumerande.

Testautomatisering

La la la.

Testdriven utveckling

Test-driven development (TDD) går i korthet ut på att man skriver sina enhetstest före man skriver sina enheter. Man skriver ett test som inte går igenom, rättar till koden för sin enhet tills den går igenom, refaktorerar och skriver därefter ett nytt test som inte går igenom, rättar till koden, refaktorerar och fortsätter på samma sätt tills man inte kan komma på fler exempel som inte går igenom.(Beck 2003).

Hur mycket kod skall man då implementera i varje steg? Exakt så lite som möjligt. Man begår de synder som behövs för att testen skall gå igenom med så lite ansträngning som möjligt. Man skall inte oroa sig för att koden blir ful eller för att det skall uppstå duplikationer. Detta löser man i refaktoreringssteget (Beck 2003).

Om vi, som exempel, vill implementera en funktion som tar två tal och adderar dem till varandra, så kanske vi börjar med att skriva ett test som adderar 4 och 6, där vi förväntar oss värdet 10. Med minsta möjliga ansträngning så skriver vi en funktion som returnerar talet 10. Funktionen gör inget annat. Testet går igenom och det finns ingenting att refaktorera, då funktionen består av endast en rad. Vi skriver sedan ännu ett test. Den här gången adderar vi 2 och 4 och kör våra test. Funktionen returnerar bara 10, så vårt andra test går inte igenom. Vi skriver då om koden till att istället lägga ihop de två variablerna och kör igen, testen går igenom. I och med att addition redan finns implementerat i de flesta programmeringsspråk så blir inte just det här exemplet jätteavancerat, men tanken är att man fortsätter på samma sätt tills allt man kan tänka sig går igenom.

Testverktyg

Verktyg för att driva test

JUnit

JUnit är ett verktyg för enhetstest av Javaapplikationer.

Robot Framework

Robot Framework är ett verktyg för automatisering av acceptanstest och acceptanstestdriven utveckling. Tre olika sorters syntax för att skriva test stöds, nyckelordsdrivet, datadrivet eller beteendedrivet.

Cucumber

Cucumber är ett beteendedrivet testverktyg för automatiserade acceptanstest. Det påminner mycket om Robot Frameworks sätt att skriva beteendedrivna test, med den visuella skillnaden att man valt att implementera nyckelordshanteringen med Javas *Annotations*, istället för att ha med nyckelordshanteringen i den motsvarande feature-filen. Cucumber har Annotations som stödjer över 40 språk, vilket kan öka läsförståelsen eller uttrycksmöjligheten för personer som inte förstår engelska.

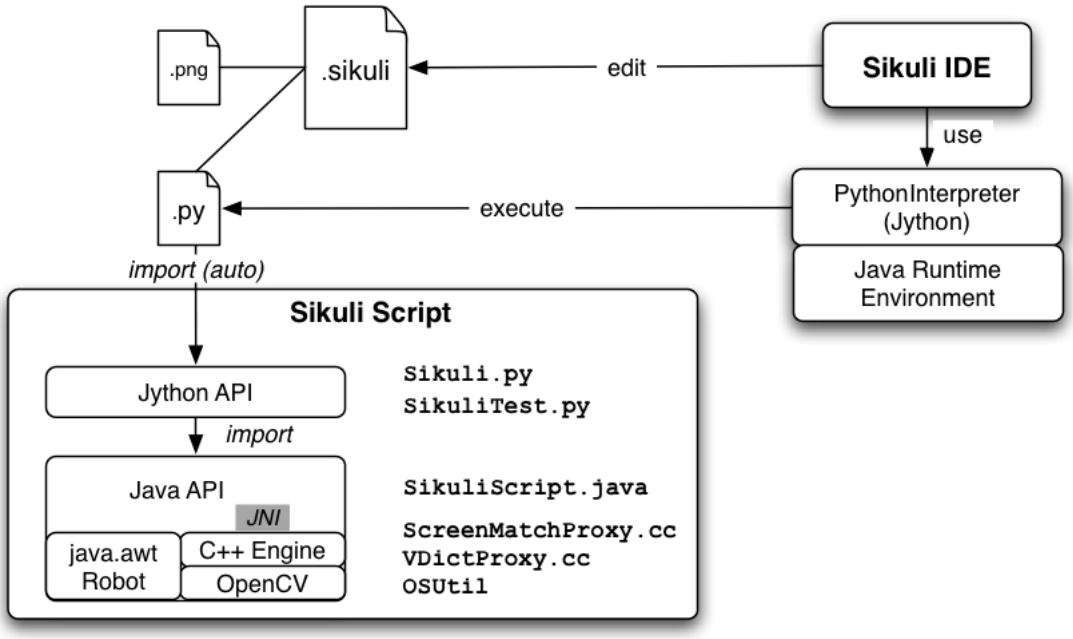
Verktyg för att styra GUI

FEST-Swing

FEST-Swing är ett verktyg utvecklat i Java som testar grafiska användargränssnitt skrivna i Java, och mer specifikt skrivna med Swingklasser. FEST söker efter komponenter genom att antingen gå på deras lokala namnattribut eller genom att söka genom samtliga komponenter av en angiven typ och matcha ett eller flera av dess attribut mot förbestämda värden. Det finns sedan möjlighet att interagera med dessa komponenter på olika sätt, t.ex. genom att klicka på dem eller fylla i ett värde i ett fält.

När test körs så startar FEST applikationen som skall testas, och tar sedan över kontrollen över mus och tangentbord, för att styra gränssnittet. Detta gör att man kan se hela förloppet av testet, om så önskas. Det finns också möjligheten att ta en skärmdump när ett test går fel, så att en testare i efterhand kan se hur det såg ut då testet gick snett.

Project Sikuli



Sikuli Script är ett Jython- och Javablibliotek som automatiserar GUI-interaktion genom bildmatchning för att styra tangentbord och mus. Dess kärna är ett Javabibliotek bestående av två delar: *java.awt.Robot*, som fångar upp tangentbords- och mushändelser och en C++-motor baserad på OpenCV som söker av skärmen för att matcha mot bilder. C++-motorn kopplas till Java via JNI. Ovanpå det hela körs ett lager av Jython som är där för att användare skall kunna skriva enkla skript, men det går även bra att importera biblioteket i din Javaapplikation och skriva test-/automatiseringskod direkt i Java istället.

Ett sikuliskript är en mapp som innehåller en Pythonfil med kod, samt alla bilder som används av skriptet. Mappen kan även innehålla en html-fil, då Sikulis IDE automatiskt sparar ned en webbversion av skriptet, för enkel publicering.

Med Project Sikuli följer ett IDE, Sikuli IDE, som bistår med verktyg och stöd för att skriva sikuliskript enklare. IDE:t förenklar väsentligt tagandet av bilder för skärmen. När Sikuli IDE kör så lyssnar den på en global tangentbordskombination (Ctrl+Shift+2) och när denna trycks ned, så minimerar IDE:t sig, för att sedan frysa skärmen eller skärmarna. Man markerar sedan ett stycke av skärmen med musen och en bild sparas, med ett interaktionsankare i bildens mitt. Vill man kan man sedan i Sikuli IDE redigera bildens ankare att hamna någon annanstans än i mitten (för att försäkra att Sikuli klickar på rätt ställe t.ex.).

UISpec4J

UISpec4J är väldigt likt FEST-Swing. Det är skrivet i Java, testen skrivs i Java och sökningen efter komponenter är snarlik. UISpec4J söker också efter komponenter baserad på dess namnattribut eller, alternativt, söker den igenom alla komponenter (till skillnad från FEST som söker igenom ett urval) och jämför attribut. Vill man jämföra attribut som är unika för en viss komponent, så får man kasta komponenten till rätt sorts komponent och själv se till att försäkra sig om att det *ClassCastException*, som kastas om komponenten inte var av rätt typ, tas omhand om.

En stor skillnad mellan FEST-Swing och UISpec4J är det sätt som UISpec4J hanterar själva testprocessen på. UISpec4J implementerar något de kallar en *WindowInterceptor*, som fångar upp de fönster som applikationen genererar och kör alla test i bakgrunden. Detta gör att testen körs snabbare, men man kan inte visuellt bekräfta att saker går rätt till. Vill man försäkra sig att

en textruta blir röd om man skriver in fel värden i den, så måste man skriva ett test som utröner att så är fallet. Man kan dock argumentera för att ett sådant test borde finnas i vilket fall.

Bräcklighet

Ett stort problem 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> buttonMatcher =
            new GenericTypeMatcher<JButton>(JButton.class) {

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