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.

Historia

Erfarenheter av testverktyg

Sikuli

- Det är viktigt att ställa in minsta similaritet mellan en bild och den testade ytan, annars klickar programmet ibland på fel knapp.
- Objekt, så som knappar, byter utseende när man för musen över den, eller när den har fokus. Måste kolla alla variationer för att vara säker på att objektet kan identifieras.
- Utgår ifrån att användaren kör med amerikansk tangentbordslayout. Med svensk layout kan den t.ex. inte skriva ’@’. Blir istället ””.
- Klarar inte av byte av t.ex. färgschema eller operativsystem.
- Kan hantera UI som den inte känner till strukturen bakom. T.ex. flash, excel, etc.
- Finns ej i maven-repositoryt. Kräver lokal installation.
- Möjlighet att slå på tydliga grafiska indikationer på när ett objekt interageras med.

Cucumber

- Fungerar på över 40 språk (även om det inte var helt tydligt hur, och vissa språk är fiktiva)
- Är väldigt läsbart. Väldigt otekniskt på framsidan (med tekniskt klister under).
- Ej färdigt. Måste ständigt ändra i beroendestrukturen i dess pom.
- Väldigt petigt/beroende av val av teckentabell (UTF-8 or die).
- Var inte helt trivialt att få att fungera till en början, men när det väl var igång så var det väldigt smidigt att utöka det. Nästan helt utan ansträngning alls.
- Många små petigheter som kanske inte nödvändigtvis behöver kontrolleras (att det står ”Feature” istället för ”Egenskap” kanske inte Cucumber behöver bry sig om då filen redan berättat att den är på svenska). Det är klart att det är bättre om hela filen är på svenska, för läsaren, men borde inte vara ett krav för att testet skall köras (och att ge felutskriften ”There are no tests.” är kanske inte heller optimalt i fallet).
- Gick mycket smidigt att pussla ihop med den befintliga strukturen för en ”GUIRunner”. När det var färdigpusslat så gick det utan problem att byta mellan Sikuli och FEST som GUI-testverktyg.

UISpec4J

- ComponentMatcher tar inte generics (så som FEST gör) och man kan därför bara jämföra komponenter på variabler som tillhandahålls av klassen Component, och inte unika fält/funktioner för klassen man jämför, såvida man inte kastar om klasserna, men då måste man se till att man gör detta säkert, eftersom alla klasser som påträffas testas, inte bara t.ex knappar.
- Går snabbt. Behöver inte slösa tid på att rita upp komponenter och simulera musrörelser. Bra när man bara vill se att testet går igenom och inte vad som faktiskt händer.

- Kan köra i bakgrunden. Tar inte kontroll över mus och tangentbord.
- Väldigt likt FEST, syntaxmässigt, efter den ursprungliga initiationen.
- Gör inga antaganden, så som FEST gör. FEST utgår ifrån att om man letar efter en knapp, så letar man efter en synlig knapp, medan UISpec4J letar någon knapp, vilken som helst. Man får själv säga till om att man vill att knappen skall vara synlig (bara ett problem om man har fler knappar med samma text). Om detta är bra eller dåligt kan man argumentera om.
- Så här en vecka i efterhand känner jag att UISpec4J är det bästa som hänt sedan skivat smör. Det skall gå fort att testa!

Robot Framework

- Väldigt likt Cucumber. Klisterkoden kunde nästan återanvändas helt. Specifikationsfilen var dock kladdigare/mer teknisk
- Något är vajsigt med ClassLoadern, då Sikuli inte funkar alls pga att dess resursmapp inte hittas. Borde gå att lösa, men det är sannolikt krångligt och inte alls uppenbart.
- Går inte att köra mvn test på modulen. Måste testas med eget kommando, men det finns en mavenplugin för det, så den körs automatiskt vid build.
- Väldigt känslig för skräp i filen, men varningarna vid kompilering är inte tydliga med vad som är fel. T.ex. så råkade jag skriva ”*** Keywords ***”” istället för ”*** Keywords ***” vid ett tillfälle och då klagade den över att den inte kunde hitta några nyckelord som matchade mina testfall. Har man inte exakt två mellanslag mellan argument bland nyckelorden, så förstår den inte heller vad som menas.
- Vid kastande av undantag, så ges inte all information. Man får felet ”NullPointerException”, som anledning till att testet misslyckades, men inte mer information än så. Man får manuellt gå in och fånga upp undantaget om man vill få ut dess stackspår.

Allmän dumpplats

Här dumpar jag sådant som jag skriver i nuläget, så får vi placera ut det där det skall vara senare.

Dynamisk klassladdning och Java Webstart

Dynamisk laddning av klasser i java är inget problem. Vet man vad klassen heter och vart den ligger så är man bara en rad kod bort.

```
ClassLoader loader = new URLClassLoader(new URL[]{}, Glitch.class.getClassLoader());
glitchList.add((Glitch) (loader.loadClass("com.sigma.qsab.glitches.customglitches." +
                glitchName)).newInstance()));
```

Vill man ladda in alla klasser i ett paket, vilket vi vill om vi skall få den dynamiska inladdningen av buggar att fungera, så går det också utan större bekymmer genom att ta reda på programmets *classpath*.

```
String path = System.getProperty("java.class.path");
```

Sedan går man igenom mappen där klasserna ligger och laddar in dem en efter en.

```
File[] files = (new File(path + "/com/sigma/qsab/glitches/customglitches")).listFiles();
for (File f : files) {
    if (f.getName().endsWith(".class")) {
        glitchList.add((Glitch) (loader.loadClass("com.sigma.qsab.glitches.customglitches." +
                        f.getName().substring(0, f.getName().length() - 6)).newInstance()));
    }
}
```

Detta funkar alldeles utmärkt lokalt, men när man distribuerar koden via Java Webstart och kör det så upptäcker man att klasserna inte har laddats.

Det är i sig kanske inte så konstigt, då Java Webstart kräver att man har hela programmet sparat i jar-filer. Då får man istället tänka om och börja ladda klasserna ur en jar-fil.

```
JarFile jar = new JarFile(path);
Enumeration<JarEntry> entries = jar.entries();
while (entries.hasMoreElements()) {
    JarEntry entry = entries.nextElement();
    if (entry.isDirectory()) continue;
    if (!entry.getName().startsWith("com.sigma.qsab.glitches.customglitches.")) continue;
    glitchList.add((Glitch) (loader.loadClass(entry.getName()).newInstance()));
}
```

Provar man nu att köra programmet lokalt från en jar-fil, så funkar det alldeles utmärkt, men när man kör det via JWS så fungerar det fortfarande inte.

Problemet inser man om man tittar på vad JWS använder för *classpath*.

```
C:\\Program Files (x86)\\Java\\jre6\\lib\\deploy.jar
```

Classpath är inte en sökväg till programmet, utan istället Javas *deploy.jar*, där JWS finns och körs från.

Hur löser vi då problemet? Hur skall man kunna hitta den jar-fil som klasserna ligger i? Svaret är tyvärr att det gör man inte.

Hur har vi då löst problemet?

[alt 1]

Vi kom fram till att man kommer bli tvungen att kompilera om hela projektet oavsett när man skapat en ny bugg-klass, för att den skall hamna i webstart-jaren, så då skrev vi en egen generator som sökte igenom mappen med klassfiler och gjorde i ordning en property-fil som

innehöll namnet på alla bugg-klasser. När vi känner till namnet på alla klasserna vid kompilering, så är det sedan en enkel sak att ladda in dem i runtime.

```
Properties glitchMap = new Properties();
glitchMap.load(GlitchLoader.class.getResourceAsStream("/glitches.properties"));
ArrayList<Glitch> glitchList = new ArrayList<Glitch>();
ClassLoader loader = new URLClassLoader(new URL[]{}, Glitch.class.getClassLoader());
for (int i = 0; i < Integer.valueOf(glitchMap.getProperty("nrofglitches", "0")).intValue();
     i++) {
    glitchList.add((Glitch) (loader.loadClass("com.sigma.qsab.glitches.customglitches." +
                   glitchMap.getProperty("" + i)).newInstance()));
}
```

[alt 2]

Skriv något om att ha klasserna i en separat jar-fil som man lägger en länk till i jnlp-filen. Ej testat ännu.

Testmoduler och cirkulära beroenden

Tanken var att skriva en separat modul med ett testbibliotek för att hantera GUI-testning. Denna modul var menad att vara den enda modul som har kontakt med FEST-biblioteket och abstrahera bort kännedom om FEST från den som skriver testkod. En annan fördel detta skulle medföra är att man kan byta testmiljö utan att testen behöver skrivas om, eller ens känna till förändringen.

Det uppstod dock ett problem med denna abstrahering. FEST kräver att den känner till den Frame den skall jobba mot; och är således beroende av implementationsmodulen. Eftersom testen i sig är beroende av testbiblioteksmodulen och är en del av implementationsmodulen, så uppstår ett cirkulärt beroende och koden går inte att kompilera.

Ett antal lösningsförslag diskuterades.

Man kan ta bort kravet om abstraktion och låta testen känna till och sköta initiering av FEST-biblioteket och därmed ta bort testbibliotekets beroende till implementationsmodulen.

En annan, bättre, lösning är att lägga till ytterligare ett abstraktionslager och låta integrationstesten ligga i en egen modul, vilket resulterar i att implementationen inte längre är beroende av testbiblioteket.

Oavsett vilken väg man väljer att gå så har man löst problemet med cirkulära beroenden.

TDD

Att skriva kod testdrivet är att skriva testen först och sedan med minsta möjliga möda anpassa koden så att testen går igenom. Har man till exempel, som utgångspunkt, ett test som skickar in en sträng i en funktion och förväntar sig att strängen antingen godkänns (funktionen returnerar sant) eller inte godkänns (funktionen returnerar falskt), så är det enklaste sättet att skriva en funktion som skall klara testet genom att bara returnera det förväntade värdet. Lägger man sedan till ett till test som förväntar sig ett annat värde så får man skriva om funktionen på ett sådant sätt att den på enklaste sätt klarar av det både det gamla och det nya testet. Sedan skriver man ett nytt test och modifierar koden så att testet går igenom, o.s.v.

Fördelar

Det blir rätt från början. Blir det inte rätt så går inte testet igenom och man upptäcker genast att något är galet.

Ändrar man eller optimerar i koden vid något tillfälle så upptäcker man direkt om det helt plötsligt finns ett test som inte går igenom. Hade man inte haft några test så hade man kanske inte hittat buggen alls och den hade nått release.

Nackdelar

Det tar längre tid att skriva funktioner som är väldigt simpla.

Det kan hända att man missar att skriva ett test för något som borde ha testats och i och med att man implementerar funktionerna allt eftersom testen inte går igenom, så kanske man missar ett fall man hade täckt om man hade skrivit funktionen rätt från början.

Jag har nu skrivit funktioner utifrån test, men jag har även provat motsatsen. Jag har skrivit funktioner som jag sedan skrivit test för (och modifierat kod när den inte gått igenom).

Fördelen med att skriva koden innan testet märker man väl framförallt på väldigt enkla funktioner, där det är lätt att göra fel från början och inte allt för mycket meck att rätta till om man upptäcker ett fel. En funktion som jämför två strängar kanske man inte behöver lägga ned en massa tid och energi på att skriva ”fel” från början för att sedan rätta till.

Samtidigt gäller motsatsen också. Om en funktion är, eller riskerar bli, mer avancerad, så kan det vara bra att ta små steg i rätt riktning hela tiden, istället för att skriva en avancerad metod där det sedan visat sig att man inte tänkt på allt och man måste skriva om stora partier och kanske till och med behöva skriva om hela funktionen helt och hållet (och då kunde man ju gjort så redan från början).

Dynamiskt buggtillägg

För att buggar skall gå att lägga till dynamiskt i programmet, så krävs det att programmet skall vara skrivet på ett sådant sätt att det är enkelt att lägga till en ny bugg och att programmet man inte skall behöva uppdatera koden för själva programmet vid skapelse av buggar.

Därför låter jag varje funktion som skall kunna gå fel, via en s.k. *GlitchManager*, Varje funktion i managern gör två saker. Den kontrollerar om det finns en inladdad *Glitch* som vill definiera om funktionens beteende. Hittas inte någon *Glitch* så kallar den den funktion som i ursprungsläget hanterar situationen på ett – för ett fungerande program – korrekt sätt. Hittas det en *Glitch* så kallas den funktion, definierad i glitchen, som tar över den förväntade funktionaliteten.

Vad är då en *Glitch*? Hur fungerar den?

För att en glitch skall fungera så måste den innehålla följande saker:

- Ett ID som berättar vilken funktion den vill ta över.
- En kort och en lång text som beskriver vad den gör (och som programmet listar vid val av buggar).
- En funktion som kan anropas av en *GlitchManager*.

Jag har skrivit en abstrakt klass *Glitch* som man helt enkelt ärver när man skriver sin egen *Glitch*.

```
public abstract class Glitch implements Comparable<Glitch> {

    <STATIC_FIELDS>

    private String longDescription, shortDescription;
    private int overrideID;

    protected Glitch() {
        shortDescription = "Short Description";
        longDescription = "Long Description";
        overrideID = -1;
    }

    protected Glitch(String shortDescription, String longDescription, int overrideID) {
        this.shortDescription = shortDescription;
        this.longDescription = longDescription;
        this.overrideID = overrideID;
    }

    public String getLongDescription() {
        return longDescription;
    }

    public void setLongDescription(String longDescription) {
        this.longDescription = longDescription;
    }

    public int getOverrideID() {
        return overrideID;
    }

    public void setOverrideID(int overrideID) {
        this.overrideID = overrideID;
    }

    public String getShortDescription() {
        return shortDescription;
    }

    public void setShortDescription(String shortDescription) {
        this.shortDescription = shortDescription;
    }

    public abstract Object performGlitch(Object... args);

    @Override
    public int compareTo(Glitch glitch) {
        return shortDescription.compareTo(glitch.getShortDescription());
    }
}
```

När du skriver en egen glitch, så ärver du *Glitch*, skriver en konstruktor som anropar

`super(String shortDescription, String longDescription, int overrideID);`

och åsidosätter funktionen `performGlitch(Object... args)`.

GlitchManagern kommer därefter automatiskt anropa funktionen i den glitch du precis skrivit, förutsatt att den laddats in (valts av superadmin), istället för att låta programmets standardfunktion anropas.

Ett exempel på en glitch:

```
public class IgnoreCasePasswordGlitch extends Glitch {

    private static String shortDesc = "Kontrollera inte versaler och gemener i lösenord";
    private static String longDesc = "Kontrollerar att lösenord överensstämmer vid "
            + "registrering, men gör inte skillnad på versaler och gemener.";

    public IgnoreCasePasswordGlitch() {
        super(shortDesc, longDesc, REGISTER_AREPASSWORDSEQUALGLITCH);
    }

    @Override
    public Object performGlitch(Object... args) {
        String password = (String)args[0];
        String passwordRepeat = (String)args[1];
        return password.equalsIgnoreCase(passwordRepeat);
    }
}
```

I exemplet, så åsidosätts funktionen som jämför två lösenord och kontrollerar att de överensstämmer, med en funktion som jämför lösenorden, men inte skiljer på gemener och versaler.

Den största nackdelen med den här lösningen är att funktionen får in en lista av objekt, vilka som helst, och returnerar ett objekt, vilket som helst. Det är upp till den som skriver buggen att veta vad som kommer in och förväntas komma ut, men detta kommer vi inte ifrån hur vi än gör.

[[UTVECKLA DET SISTA PÅSTÅENDET]]

Någon slags diskussion

Implementation

Slutsats

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.