OU1

Enhetstestningsframework

Emmy Lindgren
emli0277@ad.umu.se
id19eln@cs.umu.se

17 november 2022

Innehåll

1 Introduktion **1**

2 Användarhandledning **1**
2.1 Utformning av testklass 1
2.2 Hur startar man programmet? 1
2.3 Hur använder man programmet? 2

3 Systembeskrivning **3**
3.1 Designmönster 3
3.2 Java Reflections 4
3.3 UML-klassdiagram 4
3.4 Systemets uppbyggnad 5

4 Testkörningar **6**

1 Introduktion

För denna uppgift i kursen Applikationsutveckling (Java) implementerades ett ramverk för enhetstestning likt JUnit 3, där användare ska kunna köra testklasser och se resultatet av dessa. För att genomföra detta användes Javas *Reflection API* (JavaTPoint, 2021a), ett API som möjliggör att man kan undersöka, manipulera och skapa objekt av en klass utan att i förväg veta klassens namn eller dess metoder. Ett grafiskt användargränssnitt implementerades för körningen med hjälp av Java Swing (JavaTPoint, 2021b).

Till uppgiften genomfördes också en kodgranskning där man fick läsa och bedöma andra kurskamraters kod inför inlämning.

Syftet med uppgiften var att lära sig om Java Reflections, testa göra ett grafiskt användargränssnitt med hjälp av Javas swing och slutligen att öva på att läsa och bedöma ett programs kvalité.

2 Användarhandledning

2.1 Utformning av testklass

För att ha användning av programmet behövs en testklass som ska köras. Den testklassen har vissa kriterier som behöver uppfyllas för att testmetoder ska kunna köras.

- Testklassen behöver implementera interfacet `TestClass.java` som finns under mappen *unittester* i filstrukturen för programmet.
- Testklassen måste ha en konstruktor och ska inte ta in några parametrar.
- Metoder setUp och tearDown är valfria och behöver inte implementeras i klassen om det ej finns behov av detta.
- Testmetoder som du vill ska köras i programmet måste vara publika, deras namn måste börja med `test` (exempelvis `testAddNumber`), inte ta in några parametrar samt returnera en boolean.
- Tester kommer att räknas som avklarade om testmetoden returnerar sant, om testet returnerar falskt eller kastar ett undantag räknas testet som misslyckat.
- Testklass-filen måste ligga i samma mapp som jar-filen för programmet för att denna ska kunna hittas.

2.2 Hur startar man programmet?

För att skapa en jar-fil av programmet används IntelliJ. I menyn File -> Project Structure -> Artifacts kan man där klicka på en plus-ikon. Add -> JAR -> From modules with dependencies. Där får man välja main-klassen för programmet, sedan ok. Nu är en jar-fil definierad och den behöver ny byggas. Välj Build -> Build Artifacts -> main-

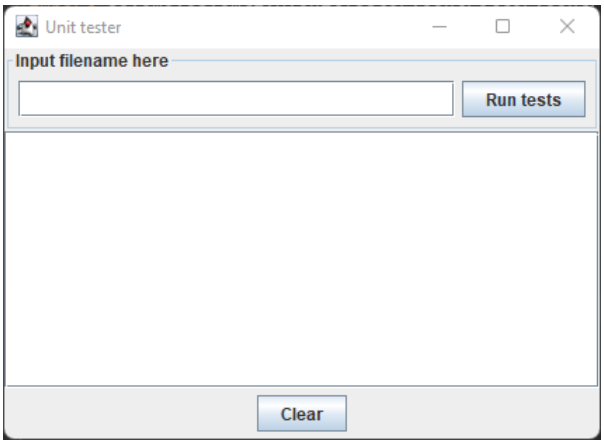
klassnamn.jar -> Build. Då byggs jar-filen. Denna behöver byggas om varje gång något är ändrat i koden i programmet.

Sedan kan programmet köras. Se först till att testklassen uppfyller kraven som listas under rubrik 2.1. Testklassen ska ligga i samma mapp som jar-filen för programmet.

Starta programmet genom att skriva in kommandot `java -jar MyUnitTester.jar` i en kommandotolk. Det går också att dubbelklicka på jar-filen för att starta programmet.

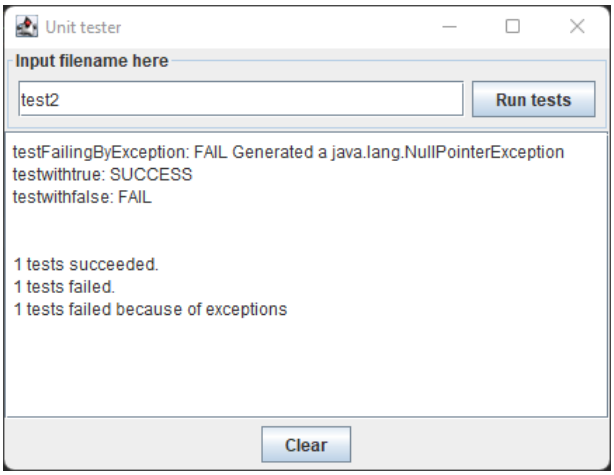
2.3 Hur använder man programmet?

När man startat programmet ser det ut enligt följande på Windows.



Figur 1: Start av programmet

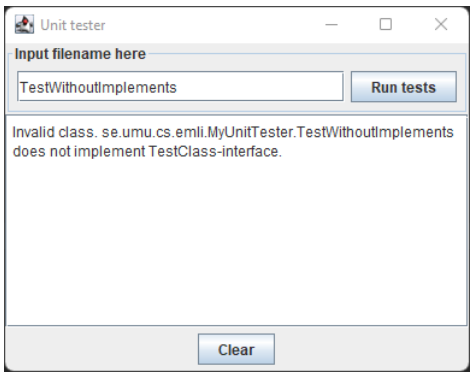
Testklassens namn matas nu in under *Input filename here*. Var noga med att ange det korrekta namnet för testklassen, gemener och versaler spelar också roll. För att sedan köra testerna på testklassen tycker man på knappen med texten *Run tests*. Då körs testerna och resultatet av dem skrivs ut. Kastas ett undantag från någon av testmetoderna skrivs typen på undantaget också ut. I slutet skrivs även ett sammanfattat resultat ut. Det kan se ut enligt följande.



Figur 2: Testkörning av testklass med namn Test2

Utskrifterna kan rensas på knappen med texten *Clear*. Skärmen rensas även mellan varje testkörning.

Om testklassen inte följer kriterierna för en testklass så visas felmeddelanden upp. Det visas även upp ifall testklassen inte hade några testmetoder som kunde köras (ifall de inte fanns några eller de som fanns var privata). Felmeddelanden kan se ut enligt följande.



Figur 3: Felmeddelande när testklassen ej implementerar TestClass

3 Systembeskrivning

3.1 Designmönster

Som designmönster har MVC använts, Model-View-Controller. MVC syftar till att separera modell och vy-objekt för att skapa en så modulär kod som möjligt (JavaTPoint, 2021c). I mitten finns istället en så kallad controller, som sköter kommunikationen mellan

dem. Vardera objekt ska (vid en ren MVC-implementation) inte ha någon vetskap om den andre.

Modellen representerar objektet som bär data och också kan innehålla en del logik relaterad till den datan. Vyn är det som presenteras i applikationen, som i detta fall det grafiska gränssnittet som visas upp. Här ska modellen visualiseras. Controller ligger mellan dessa, och sköter själva flödet i applikationen, uppdaterar modell och vy med data till och från varandra.

I detta program finns två controller-klasser (för att dela upp koden på ett rimligt sätt), **Controller** och **TestWorker**. En vyklass, **UnitTestView**, samt två modellklasser, **ClassHolder** och **ResultHolder**.

3.2 Java Reflections

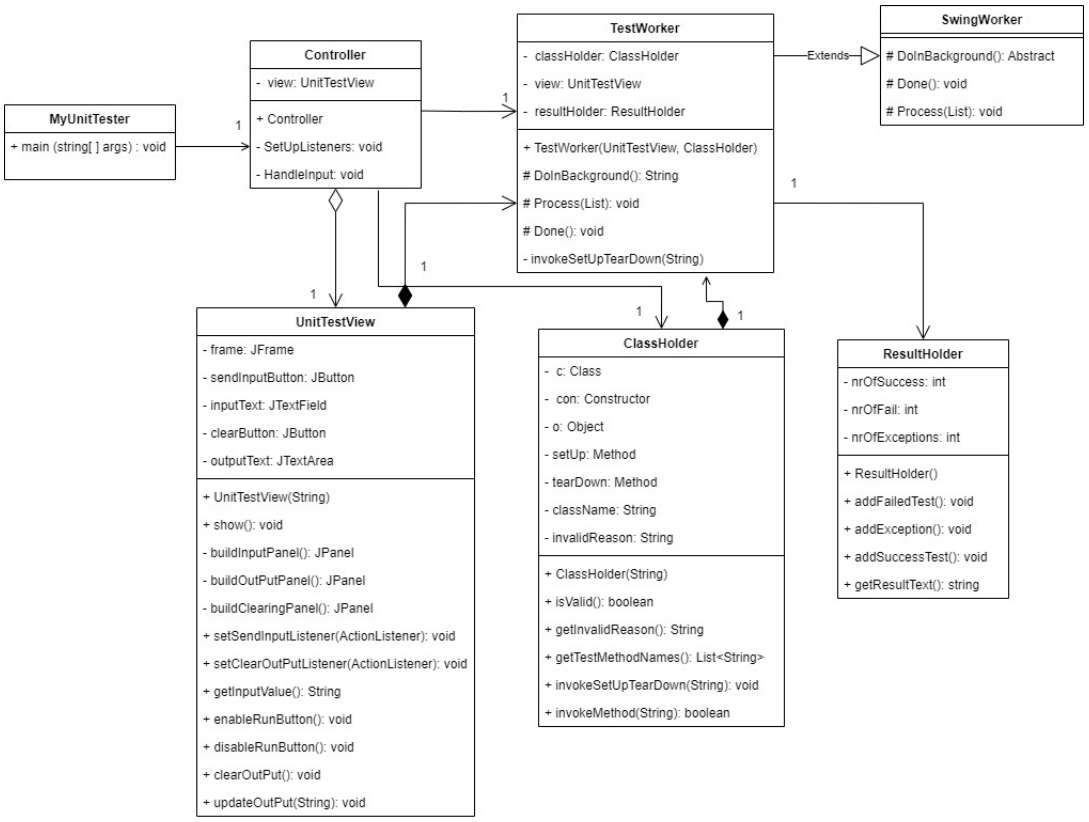
För att kunna köra testklasser som systemet inte vet någonting om används Java Reflections. Som tidigare nämnt är Reflections ett API som möjliggör att man kan undersöka, manipulera och skapa objekt av en klass utan att i förväg veta klassens namn eller dess metoder. Det är omöjligt att innan man kör programmet veta vilken testmetod som användaren kommer att vilja ange och vilka metoder denna kan tänkas ha.

Med Reflections kan klassen hämtas med hjälp av dess namn och metoden `Class.forName(namnet)`. Sedan kan en konstruktor hämtas från den klassen vilken i sin tur kan användas för att skapa en instans av klassen. Metoder som finns i klassen kan även dem hämtas från klassen, med hjälp av `klass.getMethods()` eller `klass.getMethod(metodnamn)`, ifall man vet namnet på metoden man vill hämta. Dessa metoder kan sedan köras på en instans av klassen genom att anropa `klass.invoke(instans)`.

Reflections kan även användas för att ta reda på antal parametrar, deras typer, vad metoder returnerar och ifall en klass implementerar ett interface exempelvis. Detta är bara en bråkdel av vad man kan göra, dock det mesta som används för just detta system.

3.3 UML-klassdiagram

Nedan syns ett klassdiagram som beskriver klasserna som finns i systemet samt vilka attribut och metoder dessa har.



Figur 4: UML-klassdiagram

3.4 Systemets uppbyggnad

Systemet är implementerat med Java 17. Det körs på EDT, Event Dispatch Thread. Med Java Swing bör allting som har med Swing att göra köras på EDT på grund av att Swing inte är trådsäkert (Oracle, 2022). Att köra Swing på andra trådar än EDT kan då skapa trådsäkerhetsproblem. EDT startas i mainklassen och det första som händer är att en controllerklass-instans skapas på den tråden. För att göra detta används koden:

```
SwingUtilities.invokeLater(Controller::new);
```

Det är sedan controllern som skapar en instans av vyobjektet. Vyobjektet bygger det grafiska gränssnitt som sedan visas för användaren. Men det är sedan controllern som sätter lyssnare på knapparna i vyn. Vyobjektet ska inte själv sköta vad som händer då en användare klickar på en speciell knapp, det bestäms istället i controllern för att vidhålla MVC-struktur. När användaren anger ett klassnamn och trycker *run tests* så skapar controllern en instans av Classholder med hjälp av klassnamnet. Sedan ska testmetoderna i den klassen köras, och resultatet av testerna skrivas ut.

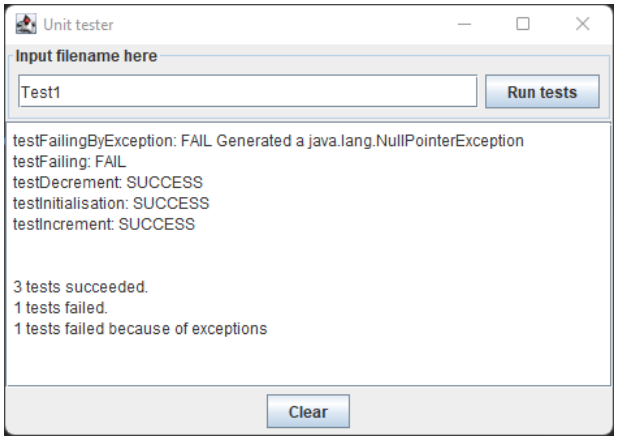
Att utföra testerna kan dock vara väldigt tidskrävande och bör därför utföras på en annan

tråd än EDT. Körs testerna på EDT så kommer gränssnittet att låsas tills dess att testet är genomfört. Det betyder att användaren inte kommer kunna klicka på någon knapp eller ens ändra storleken på rutan som innefattar gränssnittet för systemet. För att köra testerna på en annan tråd används en Java SwingWorker (Oracle, 2020).

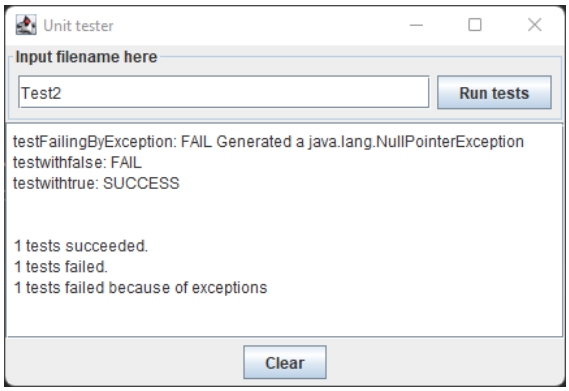
Därför skapas och startas i Controllern en TestWorker som utför tester och uppdaterar vyn med resultaten. Klassen TestWorker i systemet utökar (**extends**) klassen SwingWorker och skriver över metoderna `doInBackground, process` och `done`. Det som implementeras i `doInBackground` körs på en annan tråd medan metoderna `process` och `done` körs på EDT. Eftersom vyn bara får uppdateras från EDT betyder det att vyn får uppdateras från `process` och `done`. För att uppdatera vyn medan `doInBackground` körs så används `publish` som skickar data till metoden `process` som sedan uppdaterar vyn. I systemet används `publish` för att uppdatera vyn med testresultaten efter varje testkörning. När `doInBackground` har kört klart anropas `done`, där man kan hämta det som `doInBackground` har returnerat. I systemet skrivs slutresultatet av körningarna ut i metoden `done`.

Resultaten av testerna skrivs ut, och även ifall något undantag skett under körningen. Om ett undantag kastats under körning får TestWorker tillbaka ett `InvocationTargetException`, från vilket man kan hämta det undantag som kastats av klassen med hjälp av `getCause`. För att se till att utskrifter från tester kommer i rätt ordning är systemet uppbyggt så att bara en testklass går att köra åt gången. Knappen för att köra en testklass blir inaktiv då ett test körs och blir sedan aktiv igen då ett test har kört färdigt.

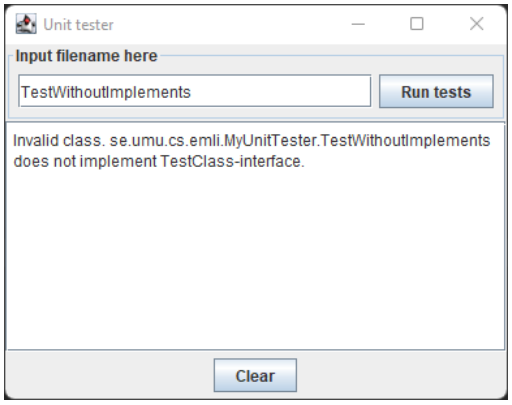
4 Testkörningar



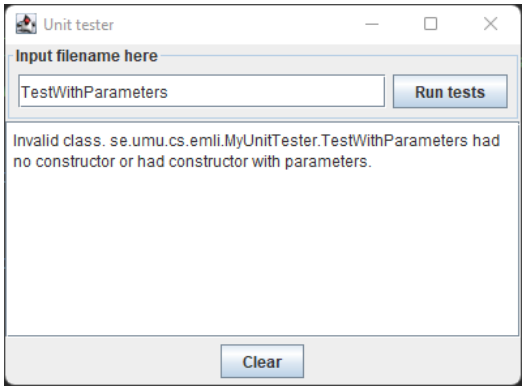
Figur 5: Körning av test 1



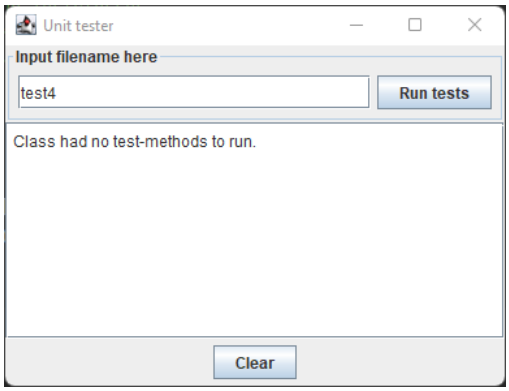
Figur 6: Körning av test 2



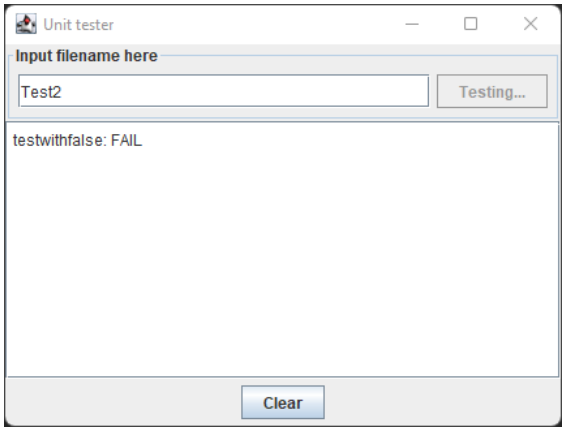
Figur 7: Körning av testclass som ej implementerar TestClass



Figur 8: Körning av testclass med parametrar



Figur 9: Körning av testclass utan publika testmetoder



Figur 10: Under en körning som tar lång tid att utföra

Referenser

JavaTPoint. (2021a). *Java Reflection API*. https://www.javatpoint.com/java-reflection

JavaTPoint. (2021b). *Java Swing Tutorial*. https://www.javatpoint.com/java-swing

JavaTPoint. (2021c). *MVC Architecture in Java*. https://www.javatpoint.com/mvc-architecture-in-java

Oracle. (2020). *Class SwingWorker<T,V>*. https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html

Oracle. (2022). *The Event Dispatch Thread*. https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html