

# OU1

## Enhetstestningsframework

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

1 december 2022

# Innehåll

<b>1</b>	<b>Introduktion</b>	<b>1</b>
<b>2</b>	<b>Användarhandledning</b>	<b>1</b>
2.1	Utformning av testklass . . . . .	1
2.2	Hur startar man programmet? . . . . .	1
2.3	Hur använder man programmet? . . . . .	2
<b>3</b>	<b>Systembeskrivning</b>	<b>3</b>
3.1	Designmönster . . . . .	4
3.2	Java Reflections . . . . .	4
3.3	UML-klassdiagram . . . . .	4
3.4	Systemets uppbyggnad . . . . .	5
<b>4</b>	<b>Testkörningar</b>	<b>6</b>
4.1	JUnit-tester . . . . .	9

## 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 kvalitet.

## 2 Användarhandledning

Under denna rubrik finnes användarhandledning. Hur en testklass ska utformas för att kunna köras i programmet, hur programmet startas samt hur programmet används.

### 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

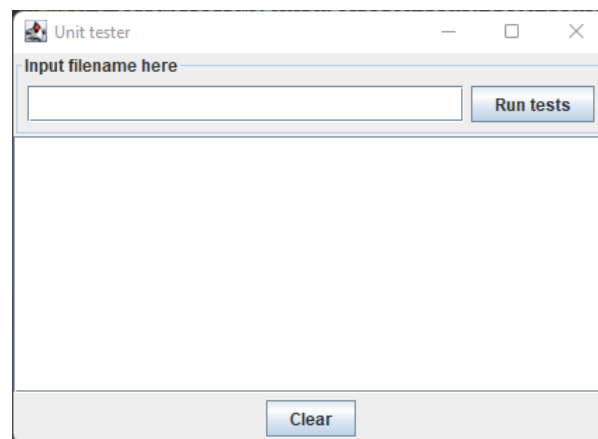
with dependencies. Där får man välja main-klassen för programmet, sedan ok. Nu är en jar-fil definierad och den behöver ny byggas. Välj Build -> Build Artifacts -> main-klassnamn.jar -> Build. Då byggs jar-filen. Denna behöver byggas om varje gång något är ändrat i koden i programmet.

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.

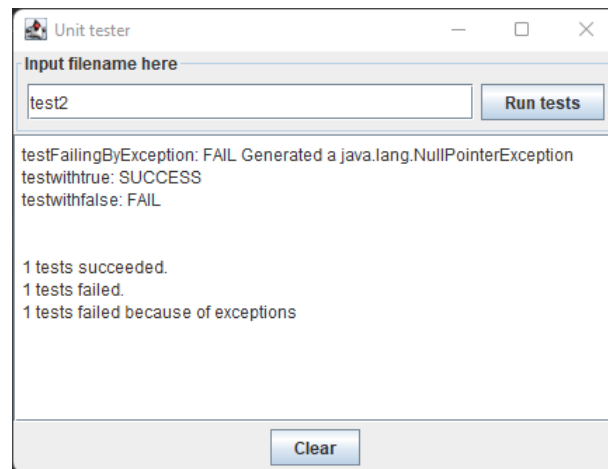
### 2.3 Hur använder man programmet?

När man startat programmet ser det ut enligt figur 1 på Windows.



Figur 1: Start av programmet

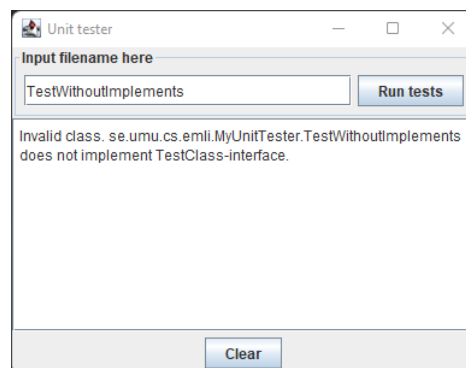
Testklassens namn matas nu in under *Input filename here*. Var noga med att ange det korrekta namnet för testklassen, gemener och versaler spelar också roll. För att sedan köra testerna på testklassen tycker man på knappen med texten *Run tests*. Då körs testerna och resultatet av dem skrivs ut. Kastar 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 figur 2.



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 figur 3.



Figur 3: Felmeddelande när testklassen ej implementerar TestClass

### 3 Systembeskrivning

Systembeskrivning beskriver hur systemet är uppbyggt och hur det implementerats. Vilket Designmönster som har följts, hur Java Reflections har använts och hur systemet är uppbyggt. Systembeskrivningen innehåller även ett UML-klassdiagram för att beskriva systemet ytterligare med dess klasser och metoder.

### 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

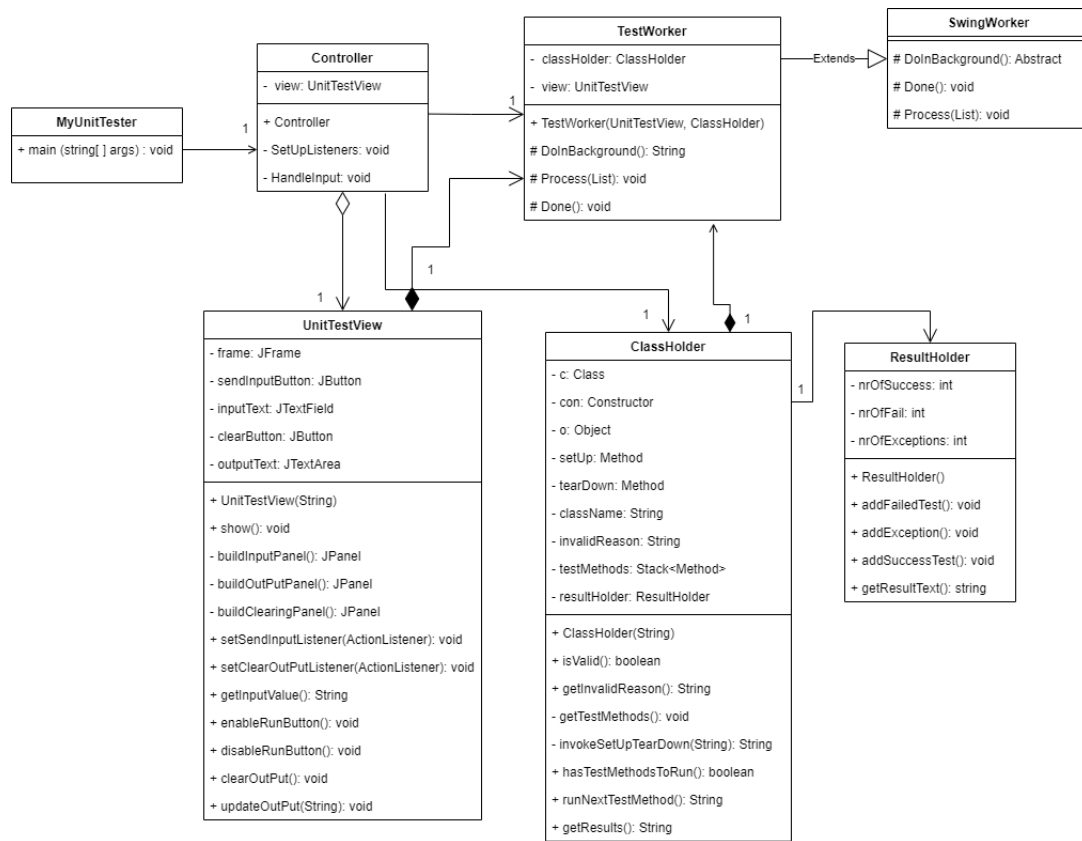
För att kunna köra testklasser som systemet inte vet någonting om används Java Reflections. Som tidigare nämnt är Reflections ett API som möjliggör att man kan undersöka, manipulera och skapa objekt av en klass utan att i förväg veta klassens namn eller dess metoder. Det är omöjligt att innan man kör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

I figur 4 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 kontrollern som skapar en instans av vyobjektet. Vyobjektet bygger det grafiska gränssnitt som sedan visas för användaren. Men det är sedan kontrollern 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 kontrollern för att vidhålla MVC-struktur. När användaren anger ett klassnamn och trycker *run tests* så skapar kontrollern 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).

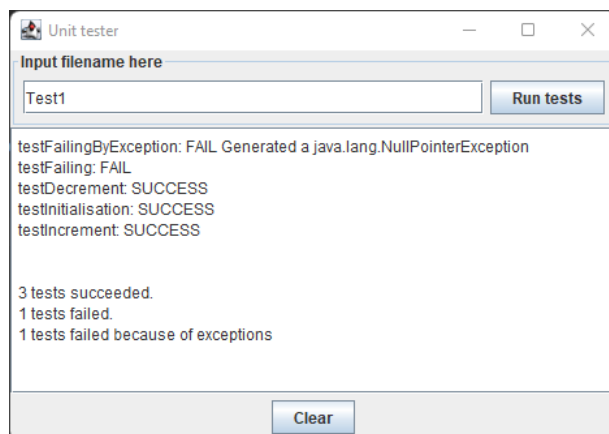
Därför skapas och startas i Controllern en TestWorker som utför tester och uppdaterar vyn med resultaten. Klassen TestWorker i systemet utökar (**extends**) klassen SwingWorker och skriver över metoderna **doInBackground**, **process** och **done**. Det som implementeras i **doInBackground** körs på 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

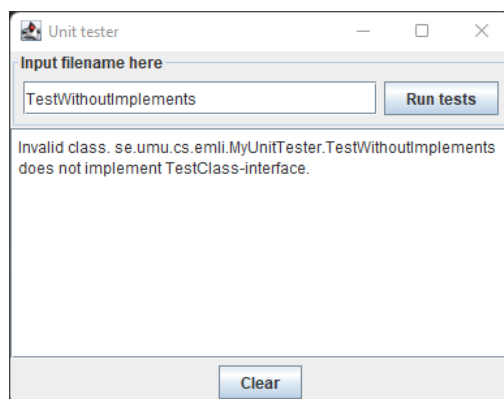
Första testet som kördes var Test 1. Där testas att undantag kastas vidare från testmetoder upp till GUI. Sedan en metod som misslyckas, samt tre stycken metoder som lyckas. Tre stycken lyckade test för att testa att slutresultatet som visas i GUI visas på korrekt sätt. Se resultatet av körningen i figur 5. Testet testar en int. Det test som kastar ett undantag försöker öka på en int som är satt till null och kastar därför ett **nullpointerexception**. Testet som misslyckas returnerar bara falskt, vilket indikerar att det misslyckats. Testen som lyckas kollar så att int:en först blir initialiserad, sedan att det går att öka på samt minska värdet på int.





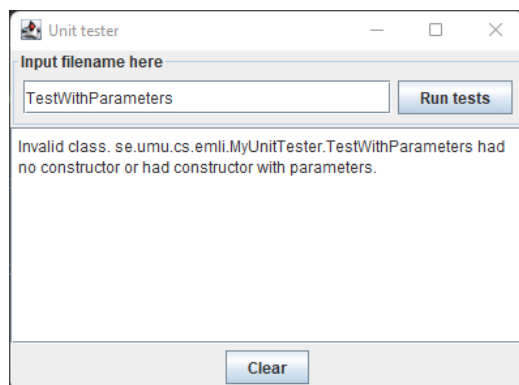
Figur 5: Körning av test 1

Ett test skrevs också för att se till så att programmet upptäcker när en Testklass inte implementerar interfacet TestClass. En testklass skrevs då utan att implementera TestClass. Resultatet av körningen syns i figur 6.



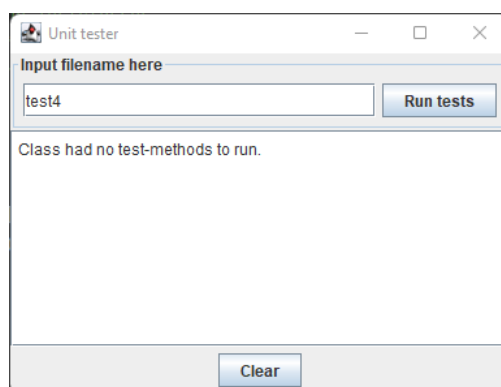
Figur 6: Körning av testclass som ej implementerar TestClass

Programmet ska även upptäcka om en Testklass tar in parametrar, vilket inte är tillåtet för en Testklass. En testklass skrevs som tog in två parametrar i konstruktorn. Resultatet av körningen på testklassen syns i figur 7.



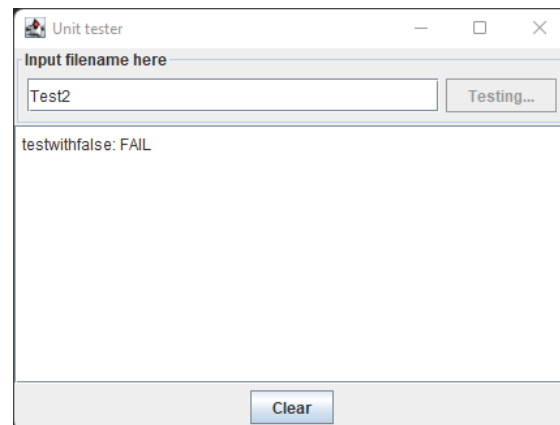
Figur 7: Körning av testclass med parametrar

Eventuellt kan en Testklass med bara privata testmetoder eller inga testmetoder alls skickas in till programmet. Då hanteras det genom en informativ utskrift för att hjälpa användare att förstå varför inga testmetoder körs. För att testa detta skrevs en Testklass som bara hade privata testmetoder. Resultatet av körningen syns i figur 8.



Figur 8: Körning av testclass utan publika testmetoder

Medan en Testklass körs måste en användare av programmet vänta tills testerna kört klart innan en annan Testklass får köras. För att testa att detta fungerar som det ska lades en `Thread.sleep()` på Swingworker mellan att denne kör varje testmetod. Under körningen ser det då ut enligt figur 9.



Figur 9: Under en körning som tar lång tid att utföra

## 4.1 JUnit-tester

Två modeller har skapats under utvecklingen av programmet. För att testa dessa har två JUnit-testklasser utvecklats, en för varje modell.

Först skapades tester för modellen **ResultHolder**. Där skrevs 5 testmetoder för att testa den publika funktionaliteten hos klassen. I figur 10 syns resultatet av testkörningarna.

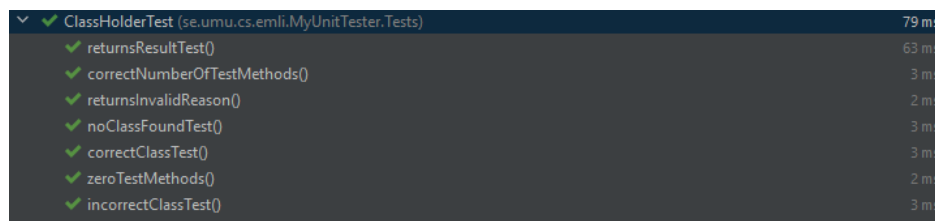
A screenshot of JUnit test results in a dark-themed IDE. The results are for the class "ResultHolderTest (se.umu.cs.emli.MyUnitTester.Model)". There are five test methods listed, each with a green checkmark icon indicating success. The execution time for each test is shown in milliseconds (ms) on the right side of the table.

✓ ResultHolderTest (se.umu.cs.emli.MyUnitTester.Model)	73 ms
✓ getResultStringWhenNoResults()	47 ms
✓ addingExceptionFailedAndPassedTest()	17 ms
✓ addingFailedTest()	5 ms
✓ addingExceptionTest()	2 ms
✓ addingPassedTest()	2 ms

Figur 10: Körning av test av Resultholder-klassen

Här behövde testmetoderna testa att det gick att lägga till ett test som misslyckats, ett som misslyckas på grund av ett undantag samt om ett test avklarats. TestResult ska även returnera en resultatsträng ifall inget resultat lagts till, vilket även testas i en testmetod. Detta testades genom att testa lägga till dessa saker och sedan kontrollera strängen som returneras som resultat, att denne motsvarar det som förväntas.

Sedan skapades tester för modellen **ClassHolder**. Där skrevs 7 testmetoder för att testa den publika funktionaliteten hos klassen. I figur 11 syns resultatet av testkörningarna.



✓ ClassHolderTest (se.umu.cs.emli.MyUnitTester.Tests)	79 ms
✓ returnsResultTest()	63 ms
✓ correctNumberOfTestMethods()	3 ms
✓ returnsInvalidReason()	2 ms
✓ noClassFoundTest()	3 ms
✓ correctClassTest()	3 ms
✓ zeroTestMethods()	2 ms
✓ incorrectClassTest()	3 ms

Figur 11: Körning av test av Resultholder-klassen

Här behövde testmetoderna testa att klassen kunde ladda in en giltig klass, samt “säger ifrån” ifall en ogiltig testklass försöker laddas in. Ifall en testklass är ogiltig ska det också gå att hämta en *InvalidReason* från *ClassHolder*, även detta testas med en testmetod. Ifall användare matar in ett klassnamn som inte kan hittas, ska *ClassHolder* returnera ett *ClassNotFoundException*. Detta testas också i en testmetod.

En testmetod testar också de fall där det inte finns några testmetoder i testklassen, då ska metoden `hasTestMethodsToRun()` returnera falskt. *ClassHolder* ska också köra testmetoderna. För att testa att alla metoder i en klass körs skickades en *TestKlass* in med känt antal metoder, och sedan kontrollerades att testmetoden kunde köras lika många gånger som det fanns test.

Till sist testades även att *ClassHolder* returnerar ett resultat efter att alla test kört klart, den sista resultatsträngen som sammanfattar hur många test som avklarats respektive misslyckats. Det testades genom att köra en testklass testmetoder och sedan hämta resultatet och jämföra det mot en tom sträng, vilken den skulle skilja sig från.

## 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>