Mjukvarutestning

Redaktör: Johan Isaksson

Version 0.1

Status

Granskad	Johan Isaksson	-
Godkänd	Andreas Runfalk	-

Innehåll

1 Inledning

Då jag (Johan Isaksson) är testledare i gruppen faller det naturligt att skriva om något testrelaterat. Därför ska denna del av rapporten handla om hur kod och andra aspekter av ett program ska testas. Även ett par olika sätt man kan testa på samt reflektioner över hur bra de fungerar i praktiken mot vad som står i böcker. Informationen som kommer att behandlas i denna del i denna del kommer att komma från läroböcker, internet och från egna erfarenheter under projektet.

1.1 Syfte

Syftet med denna del av rapporten är att klargöra hur bra olika metodiker inom mjukvarutestning fungerar samt hur och när de ska användas. Även att visa hur lätt eller svår en metod kan vara i praktiken samt vilka andra aspekter det finns som påverkar resultatet av metoden.

1.2 Frågeställning

I denna rapport kommer följande frågor att besvaras:

1. Hur, var, när och varför ska programmet testas?
2. Gjorde vi på bästa sätt?
3. Vad kan göras bättre?

1.3 Avgränsningar

De delar som valts att tas bort i detta avsnitt av rapporten är de testmetoder som inte använts under projektets gång. Detta för att tiden är begränsad och för att få så mycket förståelse som möjligt för just det som tas med. Huvudsakligen kommer det att handla om områdena Enhetstester, Integrationtester (Modultester och Systemtester) och Acceptanstester.
Anledningen till att rapporten har denna inriktning är för att mjukvarutestning är ett väldigt brett ämne. Vissa delar måste skippas för att hålla rapporten till en rimlig storlek. Dessutom så känns det mest relevant att diskutera de aspekter som har utövats under projektet, något man har erfarenhet utav.

2 Bakgrund

Dagens datorsystem blir mer och mer avancerade och får hela tiden en allt större beräkningskapacitet. Därför öppnas dörrarna upp för att skapa allt komplexare applikationer som har betydligt mer funktionalitet än förut. I och med detta krävs det en större del testning för att garantera funktionaliteten hos koden, och framförallt mer planering.
Jag har valt att skriva om just testning av ett program eftersom det just är en sådan kritisk del av ett projekt. Dessutom så har erfarenheterna i detta projekt tydligt visat mig hur det kan underlätta arbetet och betydligt öka effektiviteten när man med säkerhet kan lita på de funktioner man använder. Samtidigt minskar man det området där buggar kan uppstå vilket gör att även debugtiden i senare skriven kod minskar.

3 Teori

Eftersom testprocessen sträcker sig över hela projektet är det viktigt att tidigt tänka på vad som ska göras. Att veta hur testningen går till och information om när man ska använda en viss metod är kritiskt. Utan det skulle ingen som hade ett system gå säker. När som helst skulle ett fel kunna uppstå vilket skulle kunna leda till dyra skador beroende på användningsområdet.

Som det står skrivet i The art of software testing (referens) så spenderas vanligen ca 50% av den totala arbetstiden i ett projekt på testning av mjukvaran. 50% är en väldigt stor del vilket gör att det är viktigt att veta hur man spenderar denna tid på bästa sätt. I följande del kommer det bekrivas hur man ska gå till väga för att kunna uppnå ett bra resultat för de metoder som denna del avgränsat sig till.

4 Metod

För att ta reda på all information som behövs har ett flertal böcker tittats igenom. Ingen av dem har läst igenom till 100%, utan endast de intressanta delarna har läst med mer nogrannhet. För att verifiera att det som lästs stämmer har vissa delar testats i praktiken, i detta fall på projketet. Till exempel så har enhetstester körts på matris bibliotekts funktioner såsom matrisaddition, sytemtester och modultester har körts på den kvadratiska problem-lösaren och acceptanstester har körts på alla delar i projektet för att se till att alla krav är uppfyllda. Mer om detta kan läsas i resultat.

4.1 Enhetstest

”Software Unit Testing”, ett papper från IV&V Australia definierar ett enhetstest som ett test på den minsta möjliga samlingen kod som fortfarande går att testa nyttofullt, alltså att koden är tillräckligt stor för att fel ska kunna uppstå. Dessa test är bra då de är riktade mot en så liten portion med kod och i och med detta sällan missar buggar och fel. Det som kan vara krångligt med enhetstest är att just hitta dessa minsta samlingarna med kod, speciellt om det är en extern part (en annan programmerare) som ska skapa och utföra testen. I och med att man kör enhetstester på så små delar av kod har de i teorin en tendens till att bli väldigt många om man ska täcka tillräckligt stor funktionalitet. Detta är dock oftast inget problem i praktiken eftersom många små funktioner ofta är triviala och har ytterst få uppgifter. Då behöver inte så många test skrivas, om ens ett enda.

4.2 Modultest

Det finns flera olika definitioner av vad en modul är, och i och med det blir det då svårt att definiera vad ett modultest är. I (tAoST) definierar de att moduler och enheter är samma sak men jag har valt att istället gå efter Kristian Sandahls definiton. Han definierar att ett modultest är ett integrationtest utav två eller fler enheter. Det går ut på att man kombinerar ett antal enheter och sedan testar dem tillsammans. Testen i sig kan var väldigt lika enhetstest men täcker nu en komplexare funktionalitet. Precis som enheter kan moduler vara svåra att definiera. Om allt för många enheter förs samman och modulen får mer funktionalitet kanske den till slut kan klassas som ett system, och då förlorar modultestningen sitt syfte. Om man har ett så litet projekt som detta så är moduldefinitionen ofta inte så svår, eftersom både komplexiteten och utvecklingsmöjligheterna är begränsade. Det är kritiskt vid dessa test att underliggande funktioner redan är testade. Om inte, kan i stort sett inga slutsatser, angående vad som är fel, dras när ett test misslyckas.

4.3 Systemtest

Ett systemtest är precis som modultest ett integrationstest, men nu utav ett antal moduler isället. Testet inriktar sig nu också vanligtvis på hela produkten som har utvecklats. Detta för att säkerställa dess funktionalitet och för att hitta fel som uppstått vid kommunikation mellan moduler. Exempel på systemtest i vårt projekt är test av lösaren. Anledningen till att lösaren ses som ett eget system och inte är ihopbakad med GUI:t är att den ska kunna fungera separat. Givetvis är de båda ihopbakade också ett system, som också kräver systemtest.

4.4 Accpetanstest

Ett acceptanstest är ett test som har målet att testa om programmet är accepterbart. Med det menas att testen kollar ifall programmet uppfyller alla krav som ställs på det, och nu inte endast funktionalitetskraven. I vårt projekt är det huvudsakligen prestandan som behöver acceptanstestas. Då det prestandakrav produkten har är relativt vagt (Produkten ska ha likvärdig prestanda med gurobi") är inte testen av så stor prioritet. Men för att ha någon koll på produktens hastighet och för att garantera att kunden blir nöjd krävs ändå någon form utav test.

4.5 Övrigt

I projektet har även ett byggsystem använts, som smidigt kompilerat all kod automatiskt. Systemet har därefter även kört alla tester som skapats genom projektet. Med hjälp av byggsystemet och Continuous Integration har all kod då kunnat testas direkt när någon har skrivit ny kod och därav har det gått att se när och var feluppstått. Och eftersom sytemet även har kört alla gamla test har koden säkerställts med att alla andra funktioner, de som inte har rörts, också fortfarande fungerar.

5 Resultat

Följande del beskriver hur arbetet med efterforskningen gick samt hur testen utfördes.

5.1 Efterforskning

Det finns otroligt mycket information om mjukvarutestning men samtidigt är ämnet ganska vagt då testning beror så mycket på just vad som ska testas. I vårt projekt visade det sig att "Black Box"-testning var den metod som överlägset lämpade sig bäst. "Black Box" går ut på att man sätter en "svart låda" över det som ska testas, så att man endast kan se in- och utdatan. Sedan kollar man ifall udatan är den förväntade. "Black Box" anses bra i detta projekt eftersom hela Quadopt är uppbyggd utav många små funktioner och resultateten som de ska ge tillbaka är oftast kända på förhand. Ett exempel på detta är matrisaritmetiken där resultatet, av till exempel en multiplikation, går att räkna ut ganska enkelt på papper. Enligt (tAoST) ska dessa test utgå ifrån kravspecifikationen och andra dokumnet som beskriver vad produkten ska ha för funktionalitet. Boken beskriver också "Black Box" som en utmattande testteknik. Med menar de att man borde testa alla möjliga indata till den svarta lådan och se så att svaren stämmer. Detta är precis som det står i boken i praktiken omöjligt. Speciellt i vårt projekt där det enda som begränsar antalet olika sätt en matris, bestående utav tal, kan se ut på är datorns minne.
I projektet fanns dock ofta behovet av se en funktions lösningsgång, och då är "Black Box" en väldigt dålig metod. En metod som då lämpar sig bättre är "White Box" testning, som innebär att man kollar på den interna strukturen i en funktion. Därefter kan man kolla, efter vald indata, om lösningsgången är den väntade. i (tAoST) står det att även denna metod kan problematisk då antalet lösningsgångar kan vara väldigt många. För att se om det ens är rimligt att utföra dessa test kan man kolla på funktionens cyklomatiska komplexitet. Cyklomatisk komplexitet innebära att man gör en graf över funktionen där de möjliga stadierna i lösningsgången är noder, och de möjliga lösningsvägarna är bågar. I (Structured testing, Arthut H. Watson, Thomas J. McCabe) står det att om denna komplexitet är för stor ökar antalet fel som programmeraren gjort väldigt fort, och samtidigt blir det i stort sett omöjligt att uföra några "White Box"-test då fel kan uppstå på så många ställen.

För att säkerställa projektets krav behövdes bara en testmetod till, och det var en metod för att mäta prestanda. Den som valdes var "Load testing" som innebär att man belastar programmet med mycket data och så kollar man på hur bra det fungerar. I projektets fall gavs lösaren många problem och kollade på hur fort det gick i förhållande till andra lösare.
Det skulle kunnat varit så att GUI:t hade haft högre prioritet än vad det hade. I det fallet hade

olika typer av UX- användartest varit nödvändiga för att kvalitetsssäkra produkten. Men eftersom GUI:t beställdes utifrån kundens personliga behov, var det tydligt definierat redan från början att det var av låg prioritet.

5.2 Praktik

Som beskrivet tidigare finns det i stort sett oändligt många kombinationer av in- och utdata. För att då kunna utföra ”Black Box”-testerna behövdes antalet testfall reduceras. Detta åstakoms genom att ha möten med kunden som klargjorde att indata till programmet alltid skulle vara giltig. Det reducerade antal testfall enormt mycket, men som beskrivet i resultatet av efterforskningen finns det även väldigt mycket giltig indata. Exempelvis för matrisaritmetiken. Dessa test gick också att reducera genom att de flesta operationer är triviala och endast kräver numeriska test såsom nolldivision och flyttalsfel. Genom att även utnyttja ”White Box”-tekniken gick det att utforma olika ”Black Box”-test som tog olika vägar genom koden och på så sätt bara skapa ett test för varje väg. Denna teknik utnyttjades endast på mindre funktioner såsom moduler för att antalet olika fall skulle begränsas till något som var rimligt.
För lösaren gick det att applicera samma metod, eftersom dess funktionalitet bygger mycket på underliggande funktioner. Det som skilde sig gentemot småfunktionerna var att nu behandlades oftast rader eller kolumner i matriser istället för enskillda element. Detta ledde till att de flesta test kollade på kanterna utav det tillåtna området. Alltså kunde ett test vara att försöka hämta ut en radvektor på rad -1 ur en matris, vilket skulle vara ogiltigt.
Vid granskning av testresultat från git, Travis (verktyg för Continuous integration) och gruppmedlemmar visade det sig att majoriteten fel av bestod utav två typer: ”Assertion”s som misslyckats och ”Segmentations fault”. En ”Assertion” är ett test inuti koden som avbryter exekveringen om testet inte går bra. ”Seqmentation fault” är ett programmeringsfel som resulterat i en ogiltig läsning eller skrivning till minnet. Felen som uppstod var väldigt utspridda och olika. Det som de flesta hade gemensamt var dock att de låg på en låg nivå, alltså i bottenfunktionerna.
För att testa algoritmens hastighet stötte gruppen på oväntade problem, lösarna var för snabba. Då varje testkörning tog 0.00 sekunder för alla lösarna förutom MATLAB behövdes testen köras många gånger för att se en tydlig skillnad. Anledningen till att MATLAB är långsammare är för att den är oerhört generell men förmodligen inte gjorts med fokus på att vara snabb. Gurobi är snabb eftersom dens enda uppgift är att lösa sådana här problem och har arbetats på under lång tid. Vår algoritm är snabb eftersom den inte är generell, alltså bryr vi oss inte om vissa specialfall som vi aldrig kommer stöta på.
För att då testa deras prestanda fick lösarna lösa olika optimeringsproblem många gånger, ofta upp emot 1000 gånger, för att kunna skilja deras egentliga hastighet. Att köra testen på vår lösare och i matlab var enkelt då vi hade funktionalitet för att konvertera matriser från MATLABs definition till vår, och vice versa. Däremot så var det krångligare i gurobi eftersom tiden som var allokerad för utbildning av programmet var begränsad. Det tvingade gruppen till att mata in prblemet på ekvationsform vilket gjorde att vi var tvungna att köra testen med få variabler. Till exempel så skulle testfallet med 92 variabler ta väldigt lång tid att konvertera och mata in.

5.3 Enhetstester

Under projektet har många enhetstest skrivits (framförallt för matrisbiblioteket). Dessa test har skrivits innan och under kodningen och sedan utförts direkt efter att koden blivit klar. Det som var intressant med dessa test var mängden av fel som upptäcktes direkt. Det ledde till att utvecklingen blev mycket effektiv då fel kunde åtgärdas direkt.

5.4 Integrationstester

De modultester som planerades och utfördes under projektet var framförallt lösarens funktioner. De var uppbyggda utav sammansättningar av enheter från matrisbiblioteket och andra strukturer.

Dessa test utfördes vanligtvis samtidigt som implementeringen pågick. Detta för att hela tiden se till att rätt protokoll och gränsnitt användes.
De systemtester som utförts är tester utav lösaren och sublösaren. Dessa ansågs vara tillräcklig komplexa för att ses som system.

5.5 Acceptanstester

De acceptanstester som utförts under projektet är framförallt prestandatester. Detta för att det enda kravet lösaren hade var att den skulle vara ungefär lika snabb som den kommersiella lösaren gurobi. De delar som testats då var underfunktioner till lösaren, såsom matrisbiblioteket och subproblemslösaren.

5.6 Misstag

Det hände att vissa modul- och systemtester skedde innan de underliggande enheterna blivit testade. Ett exempel på det är lösaren som vi var ivriga att få igång och började testa tidigt. Då den inte fungerade korrekt gjorde detta att det tog lång tid att hitta felet som förmodligen hade upptäckts mycket snabbare om bara rätt testprocess hade använts.

6 Diskussion

Denna sektion kommer att förklara varför resultatet blev som det blev och vad som kan förbättras.

6.1 Resultat

Då detta är första gången som gruppen jobbar med testning som en aktiv del av ett projektet var planering och utförandet ganska svårt att ta sig in i. I början testades precis som planerat små enheter men på grund utav bristfällig utbildning påbörjades sytem- och modultesterna för tidigt. Gruppen ville tidigt testa lösaren då dens funktionalitet var av högsta prioritet, men eftersom subproblemslösaren var bristfälligt testad (ty den bristfälliga utbildningen) och i själva verket inte fungerade, även om den klarade de test som var skrivna, så var stora delar av koden tvungen att testas om. Detta hade lätt undvikts om vi hade lagt mer tidigt på utbildningen. Eftersom det var i slutet av iteration 2 som subproblemslösaren först blev färdigställd gjordes många test i onödan, men dock berodde inte det på något testrelaterat utan snarare på de beslut som togs.

6.2 Metod

Tanken var att genom hela projektet använda oss utav en ”Bottom-up”-strategi. Men eftersom organisationen i gruppen angående vem som skulle göra vad var dålig i början glömdes större delar av testningen bort. Detta ledde till att vissa funktioner felaktigt antogs fungera och att en ”Top-down”-strategi användes senare vilket inte fungerade så bra då majoriteten av fel hittades på låg nivå i koden. Då projektet ändå var ganska entydigt, med att vi skulle implementera en algoritm som löser konvexa kvadratiska optimeringsproblem, så var det ett enkelt beslut att köra enhetstester och integrationstester. Det framgick tidigt att det var framförallt funktionalitet som behövde testas, och därför behövde planeringen av testningen endast göras grovt i början och sedan specificeras när testningen närmade sig. Det som skulle behövas göras tidigare är framförallt att organisera och schemalägga testning för att testare skall kunna veta vad de har för ansvar.

7 Slutsatser

Det som jag har lärt mig utav detta projekt angående testning är vikten av att tidigit definiera projektets omfång och syfte. Utan en tydlig förståelse av projektet blir testplannering en väldigt svår uppgift och en mindre lyckad plannering påverkar arbetet genom hela projekttiden på ett

negativt sätt. Dock krävs det även en tydlig kravspecifikation och en idé om hur arkitekturen ser ut för att kunna definiera målen med testningen. Detta för att sedan kunna plannera och skapa de testfall som ska säkerställa mjukvarans funktionalitet.
Det som skulle kunna förbättras är som nämnt tidigare organisationen. När gruppen arbetade med testningen enligt beskrivningar i testplanen flöt arbetet på bra, men så fort en enhet/modul/system ansågs vara klar behövde gruppen organiseras upp igen för fortsatt testarbete. Om mer tid allokerats till testplanering i början hade förmodligen testningen gått som på räls.
Valet av strategi med ”Black Box”-test var utan tvivel rätt då ingen annan metodik passade lika bra in med vårt project. Däremot så hade testen kunnat utformas snabbare och bättre. Det som gjorde att vissa test var mindre bra var bristen på utbildning. När en grupp med väldigt lite förkunskaper får en uppgift som denna måste mer tid allokeras till utbildning. Om mer tid hade lagts på utbildning hade testarbetet kanske inte kunnat börja ända från början, men betydligt tidigare.

7.1 Sammanfattande svar på frågeställningar

1. Programmet ska testas med hjälp utav ”Black Box”-test på de delar där komplexiteten inte är för stor och resultatet är känt på förhand. På de delar där insikt i kodens lösning krävs bör istället ”White Box”-test användas. ”Black Box”-test kan därefter tas fram med hjälp utav dessa. Testen bör utföras direkt efter implementationen och helst utav utvecklaren själv, eftersom denne oftast är mest insatt. Programmet ska testas på detta sätt om det mestadels är funktionellt uppbyggd. Om det istället är inriktat på användarbarhet eller liknande är denna strategi dålig.

2. De flesta test utfördes på korrekt sätt och i rätt ordning. Men som beskrivet under ”Misstag” så användes ibland fel strategi vid fel tidpunkt.

3. För att förbättra resultatet utav testprocessen krävs en djup förståelse utav både uppgiften och den implementerade koden. Dessa faktorer skulle i detta projekt kunnat förbättras genom mer utbildning.