**Planering**

**Testplan**

För testplan så måste jag först kolla igenom minst en föreläsning och söka efter någon bra mall att använda för att göra testet. Efter som jag måste kolla på i alla fall två föreläsningar så avsätter jag 3 timmar till det. Sen avsätter jag 3 timmar till för att söka efter information och skriva arbetet.

Planerad tid: 6h

**Design och implementation**

Det här med att skriva all kod till ett användningsfall lär bli rätt knepigt. Jag måste undersöka lite i vilket språk jag ska använda och hur jag ska gå tillväga. Jag tror dock inte att jag kommer kunna skriva all kod utan det blir nog stora delar som skrivs istället. Här kommer det gå åt mycket tid till att söka information.

Planerad tid: 4h

**Enhetstestning**

Först måste jag kolla igenom föreläsningen som innehåller xUnit. Därefter blir det att fördjupa sig inom ämnet för att försöka förstå bättre vad som ska göras och hur. Mycket tid kommer nog gå åt att skriva och dubbelkolla så att jag är på rätt väg, så det blir lite iterationer av det.

Planerad tid: 5h

**Implementera testsviten och kör**

Den här lär gå rätt fort för jag bör veta vad testsvit är för något när jag nått uppgiften. Men om jag inte vet vad det är för något så lägger jag till 2 timmar som en buffert. Sen lika många timmar till att göra analysen och skriva ner allt.

Planerad tid: 4h

**Integrationstestning**

Här måste jag avsätta en del tid för att kolla upp hur jag ska skriva ner allt. Eftersom jag inte vet hur många användningsfall jag kommer välja i uppgift 3. Så för att vara säker så tar jag i lite mer med tidsplaneringen bara för att vara på den säkra sidan.

Planerad tid: 5h

**Testplan**

**Referenslista**

Referens till koden som är på github:

https://github.com/al222jf/Iterativ-Test

**1. Introduktion**

Den här testplanen är en fortsättning på gymnastikligan och ska beskriva hur arbetet ska fortsätta med testningen.

Testplanen användas för att säkerställa att slutprodukten fungerar som planerat när den når kunden. Planen kommer att användas för att upptäcka fel och buggar innan varan skickas till kund.

**2.1 Mål**

Målet är att säkerställa så att slutprodukten fungerar som planerat när den når kunden. Testplanen kommer att användas för att upptäcka fel och buggar så att man hinner fixa dom innan varan skickas till kund.

**2.2 Risker**

**Personnr**

Det är viktigt att systemet är säkert och att informationen som sparas inte går att komma åt. Systemet kommer att spara person nr och annan känslig data som inte får vara åtkomlig därför är detta en viktigt punkt. Användaren ska känna sig säker när dom fyller i sin information.

**Inlogg**

Inloggen måste förvaras säkert så att ingen utomstående kan komma åt dom. Inloggnings uppgifter är personligt och därför en risk.

**Email**

Ifall man kan komma åt användarnas email så kan dom användas till spam och allt möjligt från den som får tag på dom. Därför måste emailen sparas på ett säkert sätt så att det inte går att komma åt dom.

**Testprocess**

**Ansvarig**

I dom första testerna så kommer utvecklaren vara ansvarig för att testerna utförs. Utvecklaren kommer att utföra testerna själv också. I senare releaser kommer använderen att få göra testerna för att se hur en användare använder systemet och på så sätt kanske upptäcka fel som inte utvecklaren tänkt på.

**Enhetstest**

Enhetstestet kommer först bestå av att kontrollera koden som är skiver för att först se om man hittar några självklara felskrivningar. Efter det kommer jag skriva test kod som kontrollerar att systemet fungerar så som det är tänkt, att det visar fel medellanden där det ska och att den utför rätt åtgärder när man gör rätt.

**Intregationtest**

Det här testet kommer kanske inte fungera i den första releasen men den beskrivs ändå.

Den ska testa så att data sparas på rätt sätt när man t.ex. registrerar ett nytt konto. Den kollar så att informationen som ska sparas verkligen gör det och att systemet fungerar att användas i olika webbläsare och dess versioner.

**Testtekniker.**

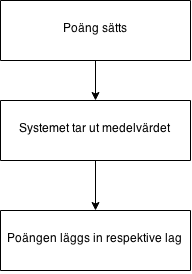
JavaScript kommer att användas för att koda testningen för att det känns bäst att göra det i nuläget.

**Design och implementation**

**Beskrivning**

Poängsystemet är skapat för att göra det mesta automatiskt och därför avlasta arbetsuppgifterna för domarna och sekreteraren. Det är designat för att göra samma arbete som utförs nu fast snabbare och säkrare. Systemet tar bort väntetiderna som kan uppstå när arbetet utförs manuellt.

**Användningsfallsmodell**



**Krav**

Poängen ska tilldelas till den tävlande och laget som den tävlar för.

Domarna ska kunna mata in nya poäng direkt efter dom skickat ett, det får inte vara några laddningstider.

Användare ska bara behöva skriva in poäng och lag för att förminska risken att något blir eller går fel.

**Begränsningar**

* Systemet får inte ta emot poäng som är noll eller har ett minus värde.
* Det kan heller inte ta emot bokstäver som poäng.
* Systemet tar bara emot siffror från ett och uppåt.

**Scenario**

1. Domaren väljer poäng för den tävlande
2. Domaren matar in poängen systemet(**A**).
3. Poängen läggs in i en array med andra domares poäng.
4. Systemet tar ut alla poäng ur arrayen och tar ut ett medelvärde av dessa.
5. Medelvärdet skickas vidare och sparas till den tävlandes lag.
6. Domaren får medelande om att poängen sparats.

**A1**

* Domaren matar in poäng över noll.
* Systemet tar emot poängen.

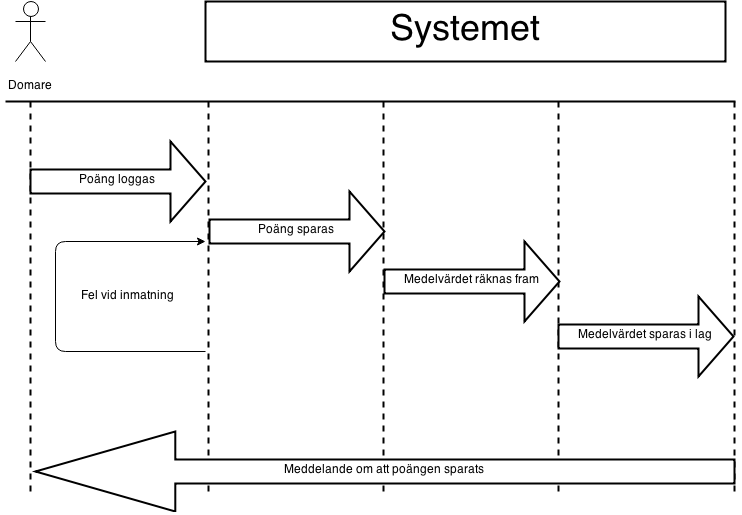
**A2**

* Domaren matar in poäng under ett.
* Systemet visar ett fel meddelande.
* Domaren får försöka igen.

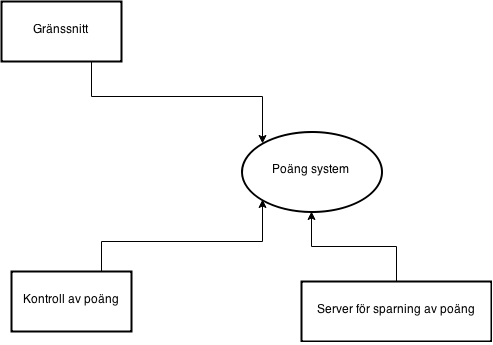
**A3**

* Domaren matar in bokstäver som poäng.
* Systemet visare ett felmeddelande.
* Domaren får försöka igen.

**Sekvensdiagram**



**Implementeringsdiagram**



**Klasser med beskrivning**

**GetPoints**

getpoints tar antalet domare som har dömt tävlingen och sparar poängen som dom matar in och kollar sen så att poängen är större än noll. Den sparar poängen i en array som sen retuneras tillbaka.

**Avrage**

Avrage tar emot den returnerade arrayen och plockar ut medelvärdet. Medelvärdet returneras sen.

**SaveToTeam**

SaveToTeam tar sen emot medelvärdet och sparar den till laget. Poängen sparas i en lista så man lätt kan läsa ut dem sen.

**Enhetstestning**

**Testsvitsdokummentering**

För att kunna genomföra testfallen så måste alla klasser vara färdiga med kod som tar hand om den data dom tar emot. Klasserna består också av tester som kollar att datan som skickas till dom är korrekt.

Testerna kommer bestå av att fel data matas in och sen se till att felmeddelande visas korrekt. Om inte ett felmeddelande kommer upp då är det bevis på att något är felskrivet eller att det finns en bugg i koden.

Dom klasserna som jag har valt att testa är GetPoints och Avrage för dom känns mest relevanta nu.

**Testsvit med testfall**

Jag har kommit fram till att jag kommer använda tre testfall

|  |  |  |  |
| --- | --- | --- | --- |
| **Testfall** | **Egenskaper som testas** | **Värde** | **Förväntat resultat** |
| TestQuantity | GetQuantity | 0 | När 0 skickas in till GetQuantity bör ett felmeddelande visas. |
| TestPoints | SavePoints | 0 | När 0 skickas in till SavePoints så ska ett felmeddelande visas att poängen måste vara mer än 0. |
| TestArray | ProcessAverage | Tom array | Ett felmeddelande ska visas som säger att det inte finns några poäng att räkna medelvärdet på. |

**Testsvit 1 (skapa)**

* Skapa GetPoints klassen.
* Skapa GetPoints konstruktor som tar emot antal domare.
* Skapa GetQuantity engenskap.
* Skapa SavePoints egenskap som hämtar poängen och sparar dom i en array.
* Skapa klassen Average.
* Skapa en default konstruktor Average.
* Skapar egenskapen ProcessAverage som tar ut medelvärdet från arrayen.

**Testsvit 2 (Testa)**

* Testar så att kontruktorn GetPoints kallar på GetQuantity.
* Testar så att GetQuantity kastar ett undantag om det är mindre än en domare.
* Testar så att SavePoints visar felmeddelande om poängen är mindre än ett.
* Testar så att ProcessAverage kastar undantag om det inte finns några poäng.

**Motivering**

Jag valde att skapa två testsviter som var och en gör en sak för att få en lättare överblick enklare förstå. Det känns bättre att dela upp dom för att då gör en testsvit bara en sak och testerna kommer bli lättare att utföra.

**Implementera testsviten och kör**

**Testklassen**

I testklassen har jag tre egenskaper som testar olika saker för att se att felmeddelanden visas när något inte stämmer. Dom egenskaperna om finns i testklassen är.

**TestQuantity**

TestQuantity testar antalet domare som dömer och ska visa felmeddelande om det är mindre än en domare. Det måste vara minst en domare som dömer annars finns det inga poäng att skicka vidare. För att testa detta sätter jag TestQuanitity till att skicka in så att antalet domare blir noll. Funkar det ska ett felmeddelande komma upp och säga till att det måste finnas minst en domare.

Resultatet blev som förväntat. Ett felmeddelande kastas upp när domarna är mindre än ett vilket betyder att testet fungerar som det ska och testet godkändes.

**TestPoints**

TestPoints testar antalet poäng som en domare skriver in. Poängen måste vara minst ett för att man ska inte kunna få noll eller minus som poäng. För att testa detta tar TestPoints och sätter poängen som noll och fungerar det som förväntat ska ett felmeddelande visa att den inskrivna poängen inte är större än noll.

Resultatet fungerade och ett felmeddelande visar att noll är inte högre än noll vilket det måste vara.

**TestArray**

TestArray testar hur stor arrayen med poäng får vara. Arrayen får inte vara tom för då går det inte att räkna ut medelvärdet på poängen. Den minsta tillåtna längden på arrayen är 1.Testet skapar en ny tom array och skickar in den till egenskapen Average. Om arrayens längd är mindre än ett ska ett felmeddelande visas som säger att det inte finns några poäng att ta ut medelvärdet från.

Resulatet från testet blev godkänt. TestArray skickade in en tom array och felmeddelandet visades som förväntat.

**Integrationstestning**

**Testbeskrivning**

Jag kommer här att testa integrationen mellan olika egenskaper för att kontrollera så att dom fungerar som det är tänkt. Testningen kommer göras i samma testklass som innan och bara lägga till nya egenskaper för dom nya testerna.

|  |  |  |  |
| --- | --- | --- | --- |
| **Testfall** | **Egenskaper som testas** | **värde** | **Förväntat resultat** |
| Integrationen mellan SavePoints och ProcessAverage | ProcessAverage | Array med ett värde 0 | Ett felmeddelande ska visas att poängen är mindre än 1 |

**Testning med TestIntegration**

Jag kommer testa att ProcessAverage tar emot arrayen från SavePoints och att felmeddelandet visas. Testet kommer fungera så att först skapas en array med ett värde och det värdet kommer vara 0. Detta görs i SavePoints och sedan kommer arrayen skickas in till ProcessAverage för att kontrollera om det fungerar eller ej. Fungerar som förväntat så kommer ett felmeddelande visas. Detta görs för att ifall det skickas en array med poäng mindre än 0 så ska det inte gå att kunna ta ut något medelvärde på detta. Helt enkelt för att det ska inte gå att ge poänget 0.

Resultatet blev också som förväntat. Det gick bra att skapa arrayen med SavePoints och när den sen skickades in till ProcessAverage så fungerade felhanteringen och felmeddelandet visades som förväntat.

**TestIntegration**

TestIntegration består av en try/catch sats som skickar in arrayen från SavePoints till ProcessAverage. Den ser för till att skapa två instanser från dom berörda klasserna för att sedan kunna komma åt dom två egenskaperna som ska integrationstestas. Catch tar sen emot felmeddelandet som ska visas och presenterar det enkelt med en Console.WriteLine.

**Reflektion**

Det har verkligen varit svårt att planera en sån här planering. För att jag hade inte alls koll på hur man gjorde en testplan eller hur man testade kod. Det har sagts väldigt mycket att det inte finns något rätt sätt att skriva en testning på. Det är inte ett arbetsätt som jag gillar speciellt mycket. Jag vill ha ett rätt sätt att skriva på för att göra det mindre förvirrande. Nu när man har letat efter guider eller mallar på nätet så har en del varit olika och alla är på engelska vilket gör det svårare att komma in i och förstå hur saker ska göras. För mig känns det som att det finns ett rätt sätt att skriva på. Om vi ska kunna använda det här sen i arbetslivet så vill man vara säker på att man skriver en korrekt testning för att då ska det vara professionellt annars lär man skrämma bort kunder. En annan svår bit av den här uppgiften var att skriva testkoden. Eftersom C# är mycket mer lämpad åt den här uppgiften för att klasser och liknande måste användas i planeringen och utförandet. Så jag valde att använda C# men nu om jag fick ändra hade jag nog valt att göra det i JavaScript istället. Jag känner mig mer säker i JavaScript för att vi håller på med det just nu. Sen tappar man väldigt mycket när man sitter med ett annat språk. Testklassen var väldigt klurig att skriva för att vi har aldrig gjort något sånt innan och därför kunde man inte gå tillbaka till dom gamla uppgifterna för att få lite hjälp.

Jag vet inte om jag har lärt mig något efter uppgiften, jag är i alla fall inte säker på att jag kan skriva en korrekt testplanering med test.

Adam Lundbeg