GURPS Lite

Introduktion till testing av it-system 2015-12-16

|  |  |
| --- | --- |
| Anton Friedmann | anfr6960 |
| Beatrice Beta | bebe5678 |
| Caesar Gamma | caga9012 |
| Diana Delta | dide3456 |

Obs! Ändra inte på formatet, det ska vara liggande A5.

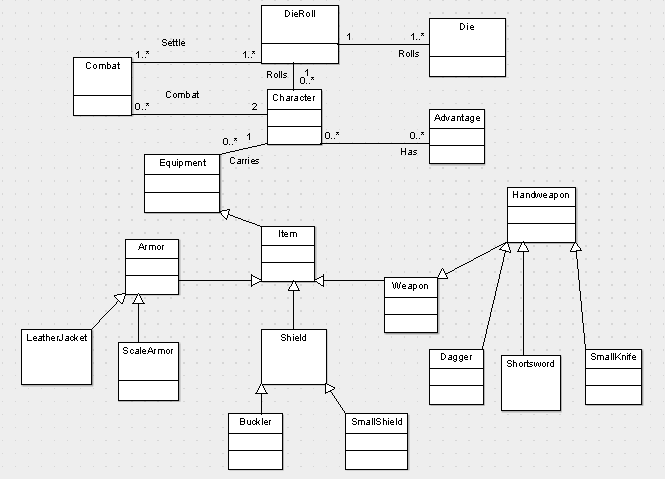
# Introduktion

En kort introduction till projektet där ni också listar de verktyg ni använt. Om ert versionshanteringssystem går att komma åt ska adressen dit finnas med, annars ska det finnas en länk från vilken man kan ladda hem källkoden.

Vi valde att utveckla spelet något förenklad version av GURPS som är ett rollspel går ut på att leva ut sina fantasier i en av spelarna bestämd värld. Spelarna ska kunna skapa karaktärer i spelet som ska kunna spendera poäng för att forma sig till den typ av karaktär de vill vara. Karaktärerna ska även kunna bära föremål samt slåss mot varandra tills en segrare koras. Vi har utvecklat spelet med Java som språk. Övriga verktyg som används listas nedan.

* Eclipse
* Git
* Ant
* FindBugs
* EclEmma

# Slutlig design



# Testdriven utveckling – process

En översikt över hur ni tillämpat TDD med exempel från olika personer och olika faser i projektet. Om ni har använt versionshanteringssystemet ordentligt bör all information som efterfrågas här finnas i det. Tänk på att kodexemplen ska vara läsbara.

Vi har använt oss av TDD genom projektets gång och försökt skriva test och utifrån dessa utveckla koden. Ett exempel nedan visar på hur det har gått till i en tidig upplaga av projektet. Denna commit visar en simpel början av en karaktärsklass vilket gav oss en förståelse för hur TDD används korrekt. Då det är en enkel del i arbetet misslyckades ej testen men dock säkerställde den att set metoderna fungerade.

<https://github.com/hadiaq/INTE/commit/874c628e82b3525a6c742fff0d23a7876fe6e29c>

Commiten nedan visar hur hur en konstruktor för föremål byggs upp utifrån tester. Testen gav grönt ljus och visade att konstruktorn fungerade som vi ville.

<https://github.com/hadiaq/INTE/commit/67784ba3b4c1da1ebc55283513e8de7f42cfc3e5>

# Testdriven utveckling – erfarenheter

En diskussion om vilka era erfarenheter ni dragit av att tillämpa TDD. Det finns inget rätt eller fel här. Enda sättet att bli underkända är att bara fuska över punkten och säga något pliktskyldigt.

Anton: Till en början uppfattade jag testdriven utveckling som något vi bara var tvungna att göra samt onödigt tidskrävande. Efter att jag gjort några testfall uppfattade jag dock det nyttiga med tekniken. Att ställa upp testfallen innan kodningen gjorde mig mer medveten om vad jag faktiskt vill att min kod ska göra. Den direkta feedbacken huruvida man får det förväntade resultatet eller inte gjorde det lätt att hitta var de logiska felen i koden skulle kunna ligga. Jag antar att det skulle vara till väldigt stor nytta i större projekt där felen kan staplas och kanske ta längre tid att fixa än den tid man lägger ner på att skriva korta testfall.

Erik: I början av projektet så var man ivrig att komma igång snabbt med skapandet av klasser och metoder som man brukar göra och inte skriva massa testfall först, men efter ett tag så inser man varför TDD är en bra utvecklingsmetod att tillämpa. När programmet växer i storlek har man tryggheten att kodbasen är väl testad från början och fungerar som man tänkt. Det kändes inte så jättemotiverat att strikt använda sig av TDD i detta projekt då det utfördes i en utbildningsmiljö och systemet tog längre tid att bygga, men i industrin är det säkert en jättebra teknik. Man är så van att inte skriva ett testfall före riktig kod, eller inget alls för den delen.

# Ekvivalensklassuppdelning – namn på del

En kort presentation av vad ni valt ut för att tillämpa ekvivalensklassuppdelning på. Ni ska kort motivera valet, och ge tillräckligt med information för att det ska gå att bedöma er. Detta avsnitt och de tre föjande (till och med testmatrisen) ska finnas för samtliga delar ni tillämpat ekvivalensklassuppdelning på.

Vi har tillämpat ekvivalensklassuppdelning på metoden createShield som används för att skapa en ny sköld som inte redan är hårdkodad i spelet. Metoden tar in en String för namnet, en int för dess värde, en double för dess vikt samt en int för dess passiva försvar.

Metoden ska kunna skapa sköldar vars namn kan vara antingen bokstäver eller siffror men inte null. Värdet får vara 0 eller positivt men inte negativt. Vikten måste vara positiv och det passiva försvaret måste vara 0 eller högre men inte positivt. Om ett eller flera argument inte är valida kastas IllegalArgumentException.

Signaturen för metoden är void createShield(String name, int value, double weight, int passiveDefence).

# Ekvivalensklasser – namn på del

Samtliga ekvivalensklasser för denna del presenterade på ett tydligt sätt.

1. Namn är en String av bokstäver – Valid.
2. Namn är en String av siffror – Valid.
3. Namn är null – Invalid.
4. Value är <0 – Invalid.
5. Value är 0 – Valid.
6. Value är >0 – Valid.
7. Weight är <0 – Invalid.
8. Weight är 0 – Invalid.
9. Weight är >0 – Valid.
10. PassiveDefence är <0 – Invalid.
11. PassiveDefence är 0 – Valid.
12. PassiveDefence är >0 – Valid.

# Testfall – namn på del

Testfallen som ni fått fram från ekvivalensklasserna. Observera att vi inte vill ha någon kod här, utan bara en tydlig presentation av testfallen i någon lämplig tabellform.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Testfall | Name | Value | Weight | PassiveDefence | Valid |
| 1 | ”The Shield” | 10 | 5.0 | 2 | X |
| 2 | ”1234567” | 0 | 5.0 | 0 | X |
| 3 | null | 0 | 5.0 | 2 |  |
| 4 | ”The Shield” | -2 | 5.0 | 2 |  |
| 5 | ”The Shield” | 3 | 0 | 2 |  |
| 6 | ”The Shield” | 3 | -2.0 | 2 |  |
| 7 | ”The Shield” | 3 | 5.0 | -3 |  |

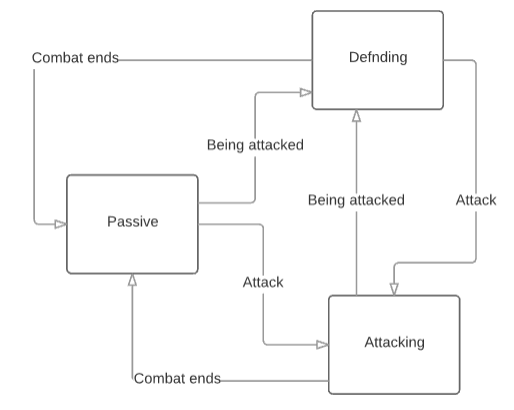
# Testmatris – namn på del

En testmatris som visar sambandet mellan ekvivalensklasserna och testfallen för denna del.

|  |  |
| --- | --- |
| Testfallsnummer | Ekvivalensklasser |
| 1 | 1, 6, 9, 12 |
| 2 | 2, 5, 11 |
| 3 | 3 |
| 4 | 4 |
| 5 | 8 |
| 6 | 7 |
| 7 | 10 |

# Tillståndsbaserad testning

Vi har valt att göra ett tillståndsdiagram för en karaktär som slåss. Vi valde just denna del för att det är den enda del vi tyckte passade in på att skapa testfall utifrån ett tillståndsdiagram. Som täckningskriterium har vi valt branchcoverage vilket innebär att alla övergångar ska täckas.



# Testfall för tillståndsbaserad testning

Testfallen som ni fått fram från tillståndsmaskinen. Observera att vi inte vill ha någon kod här, utan bara en tydlig presentation av testfallen i någon lämplig tabellform. Det ska enkelt gå att mappa testfallen till tillståndsmaskinen.

|  |  |  |
| --- | --- | --- |
| Testfall | Beskrivning | Täckta övergångar |
| 1.1 | Två karaktärer skapas ”1” och ”2”. Båda är i passive state per default. Karaktär 1 attackerar karaktär 2. | Övergång från passive till attacking(1). Övergång från passive till defending(2). |
| 1.2 | Karaktär ”2” attackerar karaktär ”1”. | Övergång från attacking till defending(1).  Övergång från defending till attacking(2). |
| 1.3 | Karaktär ”1” väljer att fly. | Övergång från attacking till passive(2).  Övergång från defending till passive(1). |
|  |  |  |

# Granskning

En kort presentation av den del av koden ni valt ut för att göra en formell granskning av och processen ni använt er av inklusive eventuella checklistor, scenarier, edyl. Ni ska kort motivera valen, och ge tillräckligt med information för att det ska gå att bedöma er.

Vi har valt att formellt granska vår combatklass där två karaktärer ska kunna slåss mot varandra. Combatklassen valdes för granskning då det är den innehaller de svåraste metoderna. Vi utgick från att läsa igenom checklistan som finns i iLearn för granskning under workshopen. En av oss tog rollen som sekreterare och den person som skrivit koden tog an rollen som granskningsledare. Vi alla var med och gick igenom koden tillsammans och diskuterade kring problemen samt möjliga lösningar för svagare delar i koden.

# Granskningsrapport

En lista över de påträffade felen och hur pass allvarliga ni bedömer dem.

Combat

# Granskning – erfarenheter

En diskussion om vilka era erfarenheter ni dragit av att tillämpa granskning. Det finns inget rätt eller fel här. Enda sättet att bli underkända är att bara fuska över punkten och bara säga något pliktskyldigt. Ni förväntas förhålla er till såväl kursboken som utdelat material och IEEE Std 1028.

Under granskningen insåg vi ganska snabbt att det är lättare att hitta fel i en kod som inte är skriven av en själv. Vi uppfattade granskningen som nyttig för att hitta fel samt för att få en förståelse för svårare kodpartier där den utvecklaren som skrivit fick förklara koden rad för rad. Hela granskningsprocessen upplevdes som tidskrävande men nyttig då det är bra med flera kritiska ögon som kollar samtidigt på samma kod.

Under projektets gång har vi även gjort kortare informella granskningar för att reda ut frågor angående kod och klassupplägg. Dessa informella granskningar uppstod naturligt under projektets gång och var nyttiga för att få teamet på samma kurs och var långt ifrån lika tidskrävande som det formella formatet.

# Kodkritiksystem

En presentation av de problem som hittats med hjälp av verktyg för statisk analys och en diskussion av dem enligt anvisningarna. Det räcker alltså inte med att bara lista problemen, ni måste förhålla er till dem också. Tänk också på att ni ska göra detta både på koden som den såg ut före granskningen och på koden efter att ni rättat det som kommit fram under granskningen.

Vi använde oss av Findbugs som kodkritiksystem för att göra en statisk analys av koden. Innan vår granskning fann vi dessa 29 buggar där samtliga hade ”Troubling” som prioritet. 4 av dessa var av hög prioritet, 2 hade medlhög prioritet och resten var av låg prioritet.

## Bad practice

Vi fick fem buggar som rörde namngivningskonventionen. Samtliga av dessa buggar rörde våra test som skulle tvinga fram illegal argument exceptions där vi hade döpt testningsmetoderna med stor bokstav. Det är enkla problem att åtgärda vilket vi kunde göra med detsamma.

## Malicious code vulnerability

FindBugs hittade fem buggar som rör variabler i main och die klasserna som borde vara konstanter. Efter att ha kikat närmre på de berörda delarna i koden höll vi med och ändrade dessa till final då de ej borde vara variabler. Det var 4 av dessa buggar som hade hög prioritet.

## Dodgy code

FindBugs hittade 19 buggar kring ämnet dodgy code. Dessa delades upp i olika typer som listas neda

## Casting from integrial casting

Två buggar med medelhög prioritet hittades angående avsaknad av formell kastning I vår getMovementSpeed metod so returnerar en double baserat på uträkning av två integers. Programmet i nuläget gjorde en implicit kastning på returvärdet men det är enkelt åtgärdat genom att implementera en explicit kastning istället.

## Dead store to local variables

Vi fann 14 buggar som rörde variabler som skapades men ej användes. Anledningen var att variablerna skapades men inte användes i testfallen. Hälften av dessa fel rörde låg i tester som skulle generera exceptions och dessa var lätta att åtgärda då vi bara tog bort variabeln där det sparades.

Den andra hälften var mer komplicerad då den utgick från vår advantageklass som var något underutvecklad utan färdiga hårdkodade advantages. Vi beslöt oss för att göra om den klassen med färdiga hårdkodade förmågor i stil med vår equipmentklass.

## Test for floating point equality

En bugg hittades i en metod där vi jämför två doubles(movementspeed) för att avgöra vem som inleder en combat. Det beror på att det finns en risk att en vanlig större än(<) kan missa skillnader på två flyttal som ligger mycket nära varandra och se dessa som lika. Problemet kan lösas genom att använda doubleklassens compare-metod istället för ’<’.

## Unused fields

Ett problem vi fann är att en map i vår mainklass där karaktärerna ska lagras inte läses. Anledningen är att vi ännu inte lagrat några karaktärer och därmed beslöt vi oss att ignorera denna bugg.

## Useless code

En bugg vi fann påstod vi redan visste att arg0 > 0 I en metod I vår equipment-klass. Vi var något förvirrade över buggen och kommer titta närmre på koden där buggen ligger.

# Statiska mått

En presentation och diskussion kring ett antal lämpliga statiska mått på koden. Att vi inte specificerar exakt vilka mått som ska tas upp beror på att olika verktyg har olika uppsättningar, men vi förväntar oss fler och mer intressanta mått än bara rena storleksmått som LOC, #klasser, #metoder, etc. Även här är det viktigt att förhållas sig till måtten, inte bara lista dem.

Verktyget metrics 1.3.6 användes för att utföra en statisk analys av olika mått. Vi valde att presentera dessa mått:

1. Nested block depth: Vi hade ett medelvärde 1,097 med en standardavvikelse på 0,408. Dessa värden får ses som acceptabla då det maximala värdet som kan accepteras innan man bör bryta upp blocket med kod är 5. Detta mått innebär att vi inte har så mycket djup nesting i våra metoder vilket gör koden lättare att följa.
2. Lines of code(metodnivå): Medelvärde på 6,392 med en standardavvikelse på 16,991. Det var bara en metod som ansågs vara för stor enligt metrics 1.3.6 riktlinjer vilket säger att en metod inte bör vara större än 50 rader kod. Det var vår calcutateBasicWeaponDamage-metod i equipmentklassen som hade hela 92 rader kod. Anledningen är att det är en metod som tar hänsyn till många variabler och vi bör kolla närmre på om det är möjligt att dela upp metoden för att förenkla förståelse vid läsning av koden.
3. McCabes cyklomatic complexity: Vi fick ett medelvärde på 1,716 och en standardavvikelse på 3,94. Detta mått visar på komplexiteten i programmet i form av hur många vägar som finns att gå i metoderna. Anledningen till den höga standardavvikelsen beror precis som i måttet ovan på vår calcutateBasicWeaponDamage-metod som hade ett värde på hela 46. Måttet bör enligt riktlinjerna definitivt inte vara över 10 vilket ger oss ännu mer anledning till att försöka bryta upp metoden.

# Täckningsgrad

En översikt över vilken täckningsgrad era testfall uppnått. Denna kan antagligen tas rakt av från verktyget ni använt för att mäta den. Om ni inte uppnått fullständig täckning så ska detta förklaras och motiveras.

# Profiler

En kort presentation av hur ni gått tillväga för att testa koden med en profiler och vilka resultat ni fick fram.

# Byggscript

Byggscriptets första (seriösa) version, och den slutliga.

<project>

<path id="application" location="GURPS.jar"/>

<path id="classpath">

<fileset dir="lib" includes="\*\*/\*.jar"/>

</path>

<target name="clean">

<delete dir="bin"/>

<mkdir dir="bin"/>

<delete file="GURPS.jar"/>

</target>

<target name="compile">

<javac srcdir="src" destdir="bin">

<classpath>

<path refid="classpath"/>

</classpath>

</javac>

</target>

<target name="jar" depends="compile">

<jar destfile="GURPS.jar" basedir="bin">

<manifest>

<attribute name="Main-Class" value="GURPSmain"/>

</manifest>

</jar>

</target>

<target name="run" depends="jar">

<java jar="GURPS.jar" fork="true"/>

</target>

<target name="junit" depends="jar">

<junit printsummary="yes">

<classpath>

<path refid="classpath"/>

<path refid="application"/>

</classpath>

<batchtest fork="yes">

<fileset dir="src" includes="\*Test.java"/>

</batchtest>

</junit>

</target>

</project>

# Övrigt

Här kan ni ta upp övrigt av relevans för bedömningen av ert arbete. Om avsnittet inte behövs kan det plockas bort.