QA-Report: Meilenstein 5

1. Konzept

Unser grösstes Problem, welches wir mit der Zeit immer wieder beheben mussten, war die Anzahl Zeilen pro Methode. Da unsere Schach Engine sich auf die jeweiligen spezifischen Winkelberechnungen und mehrere Vergleiche zwischen den Figuren basiert, stieg die Anzahl Zeilen enorm. Unser Konzept basiert sich auf das regelmässige Testen des Jars innerhalb der Gruppe um alle Aspekte, welche nicht durch Unittests abgedeckt sind im Auge zu behandeln.
Durch den implementierten Logger konnten wir mehrere Bugs auffinden, welche wir zuvor nie bemerkt hatten. Möglicherweise Loggen wir zu viel zu oft, jedoch wollen wir auf Nummer sicher gehen und alles im Auge behalten. Wir Loggen mit Log4j, führen die Unittest mit Junit aus und messen mit Jacoco und MetrixReloaded. Basierend auf unsere Messungen haben wir im Verlauf der Zeit immer wieder Code umstrukturiert und somit Schwachstellen entfernt.

2. Wie wird geprüft

Statistisch geprüft werden hauptsächlich die Komplexität der einzelnen Klassen und Funktionen, sowie die Anzahl Zeilen pro Klasse. Dies wird durch MetricsReloaded tabellarisch visualisiert. Dabei haben wir nicht auf genaue Zahlen geschaut, sondern eher die Klassen und Funktionen relativ zu einander verglichen basierend auf ihre Aufgaben. Uns ging es dabei hauptsächlich darum Code so unkompliziert wie möglich zu schreiben damit es weniger anfällig für Bugs ist.

Durch Jacoco konnten wir die Tests so breit wie möglich gestalten um alle notwendigen Aspekte abzudecken. Dabei wollen wir nicht jede if Bedingung jeder Methode durchlaufen lassen sondern die Klassen insgesamt auf ihre Zusammenarbeit überprüfen.

3. Durchführung

Es sollte, bevor die Plots berücksichtig werden, einige Aspekte unserer Tests noch erklärt werden. Wir haben, um die Tests schneller laufen zu lassen, einige Methoden, welche mit einer hohen Wahrscheinlichkeit nicht mehr verändert werden, für die Tests dupliziert um es Test freundlicher zu machen. Ein konkretes Beispiel dazu ist unsere Move Funktion im Data File. Da in unserem Schach nicht der gleiche Spieler mehrere Züge nacheinander ausführen kann, haben wir in den Unittests diese Bedingung entfernt. Die Vorteile überwiegen die Nachteile enorm weshalb uns diese, vielleicht gewagte Entscheidung, als angemessen erschien. Dadurch konnten wir in der Vergangenheit viel einfacher Bugs ausmisten und die Funktionalität des Codes überprüfen. Der einzig wichtige Nachteil liegt darin, dass es keine echte Schachpartie ist. Dennoch sind wir davon überzeugt, dass dies die richtige Entscheidung war, da unser Schach, soweit uns bekannt, Fehlerfrei nach unseren Vorstellungen läuft.

Die Funktionalität der Unittests wird jedes Meeting von uns überprüft. Jedes Mal, wenn Code verändert wird, testen wir alle Funktionalitäten durch die Unittests, indem wir das Jar erstellen durch den Gradle Build Befehl. In den Unittests werden alle wichtigen Funktionalitäten der Server Client Eigenschaften und des Schachs separat getestet. Für das Schach kann alles lokal getestet werden, da unser Schach im Data File, sobald es erstellt wird, spielbar ist, d.h. wir müssen nicht den langen Weg gehen indem Server und Clients erstellt werden, sondern können direkt im Data File testen. Für die Lobbies brauchen wir offensichtlich einen Server und einen (oder bis zu drei) Clients. Durch geschicktes manipulieren der Zeit, werden Server und Clients als Threads gestartet und jeweils 10-100 Millisekunden Zeit gegeben um bestimmte Nachrichten zu empfangen, welche Prozesse wie z.B das Eintreten einer Lobby berichten. Dabei haben wir das Beispiel vom «Hello World» Test adaptiert.

4. Resultate

Im ersten Plot ist unsere Test Coverage zu sehen. Im ersten Blick seht die Grafik schrecklich aus, da nur knapp ein drittel geprüft wird, aber der Schein trügt. Field, GFX und alles was „GUI“ im Titel hat ist für die graphische Repräsentation zuständig. Dies wird nicht getestet, da es einerseits mühsam ist und andererseits unnötig. Ihre Hauptaufgaben ist die Darstellung der Funktion nicht das Ausführen an sich. Wir haben GUI immer Manuel getestet, weil es sich als effizienter erwies.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Field.new JPanel() {...}		0%		0%	125	125	179	179	3	3	1	1
GFX		0%		0%	51	51	149	149	12	12	1	1
Data		58%		38%	168	236	161	410	2	18	0	1
Field		0%		0%	25	25	111	111	9	9	1	1
ClientThread		22%		18%	48	56	107	149	6	11	0	1
LobbyGUI		0%		0%	10	10	105	105	4	4	1	1
Net		50%		44%	34	59	90	178	9	20	0	1
Lounge		0%		0%	5	5	47	47	4	4	1	1
EncodeNet		22%		16%	26	30	54	70	1	3	0	1
ManualGUI		0%		0%	11	11	41	41	5	5	1	1
Client		52%		43%	12	23	45	92	2	9	0	1
Tools		37%		29%	21	30	49	77	3	6	0	1
Main		0%		0%	10	10	31	31	3	3	1	1
Lobby		64%		44%	10	17	22	56	4	8	0	1
HighScore		51%		37%	7	14	24	53	1	6	0	1
Chat		53%		0%	2	4	16	37	1	3	0	1
ServerThread		73%		61%	6	12	12	38	0	3	0	1
ToolsGUI.new WindowAdapter() {...}		27%		0%	3	4	10	11	1	2	0	1
StartMenu		93%		0%	2	5	4	57	1	4	0	1
CallClient		0%		n/a	2	2	4	4	2	2	1	1
CallServer		0%		n/a	2	2	4	4	2	2	1	1
Server		88%		50%	1	3	5	21	0	2	0	1
ToolsGUI		87%		n/a	1	6	3	15	1	6	0	1
ToolsGUI.new JPanel() {...}		92%		n/a	0	3	2	11	0	3	0	1
Point		100%		n/a	0	3	0	10	0	3	0	1
Total	7'529 of 11'097	32%	879 of 1'145	23%	582	746	1'274	1'953	76	151	9	25

Abbildung 1: Jacoco Test Report alle Klassen

In der zweiten Abbildung sieht es jedoch besser aus. Hier wird unser Test Konzept, wie oben beschrieben, deutlicher. Die Data Klasse ist unser Schach mit allen Funktionen und Figuren. Die move Funktion wurde für das Testen ersetzt mit moveTest und ist identisch abgesehen von einer kleinen if-Bedingung, welche überprüft ob die Spieler nacheinander spielen. Da dies die Hauptfunktion ist haben wir auch grossen Wert daraufgelegt, dass dies genügend Abgedeckt wird.

Data

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
move(int, int, int, int, char, String)		15%		6%	76	77	73	87	0	1
moveTest(int, int, int, int, char, String)		59%		38%	61	76	38	81	0	1
checkPawnCapturing(double, double, int, int, int, int, Point, Point, String, char)		0%		0%	5	5	14	14	1	1
main(String[])		0%		0%	3	3	19	19	1	1
checkCheck(String, char)		81%		73%	8	18	10	42	0	1
changeColor()		83%		55%	6	10	4	21	0	1
checkLimitedMove(double, double, int, int, int, int, Point, Point, String, char)		87%		50%	3	4	2	12	0	1
checkPawnMove(double, int, int, int, int, String, char)		90%		50%	1	2	1	9	0	1
checkArc30(double)		96%		83%	1	4	0	2	0	1
Data(String, Lobby)		100%		100%	0	11	0	61	0	1
checkMove(double, double, int, int, int, int, Point, Point, String, char)		100%		92%	1	8	0	25	0	1
getFields(Point)		100%		100%	0	2	0	8	0	1
getCoordinates(int, int)		100%		100%	0	2	0	6	0	1
sendField(String, int, int, int, int)		100%		100%	0	3	0	9	0	1
clearBoard(char)		100%		100%	0	4	0	5	0	1
checkArc60(double)		100%		62%	3	5	0	2	0	1
feedback(double, String, String)		100%		n/a	0	1	0	4	0	1
static {...}		100%		n/a	0	1	0	3	0	1
Total	1'179 of 2'858	58%	260 of 421	38%	168	236	161	410	2	18

Abbildung 2: Jacoco Test Data

Mit MetricsReloaded haben sich folgende Daten gesammelt. Alle GUI Klassen sehen natürlich toll aus, da ihre Komplexität von den eingefügten Komponenten abhängt. Im Durchschnitt seht es am schlechtesten für EncodeNet aus. Das liegt wahrscheinlich da dran, dass wir hier das Netzwerkprotokoll befolgen und sehr viele switch Cases haben, welche wiederum Funktionen aus der Net Klasse aufrufen.

class	OCavg	OCmax	WMC
ch.unibas.dmi.dbis.cs108.project.EncodeNet	28.00	28	28
ch.unibas.dmi.dbis.cs108.project.Data	8.06	34	137
ch.unibas.dmi.dbis.cs108.project.Tools	7.25	19	29
ch.unibas.dmi.dbis.cs108.project.Main	7.00	7	7
ch.unibas.dmi.dbis.cs108.project.Field	6.78	38	61
ch.unibas.dmi.dbis.cs108.project.ClientThread	5.71	21	40
ch.unibas.dmi.dbis.cs108.project.GFX	4.50	19	45
ch.unibas.dmi.dbis.cs108.project.ServerThreac	4.00	7	8
ch.unibas.dmi.dbis.cs108.project.Client	3.29	8	23
ch.unibas.dmi.dbis.cs108.project.Net	3.17	7	57
ch.unibas.dmi.dbis.cs108.project.Server	3.00	3	3
ch.unibas.dmi.dbis.cs108.project.HighScore	2.60	5	13
ch.unibas.dmi.dbis.cs108.project.LobbyGUI	2.50	6	10
ch.unibas.dmi.dbis.cs108.project.Lobby	2.29	7	16
ch.unibas.dmi.dbis.cs108.project.ManualGUI	2.25	4	9
ch.unibas.dmi.dbis.cs108.project.Chat	1.33	2	4
ch.unibas.dmi.dbis.cs108.project.Lounge	1.25	2	5
ch.unibas.dmi.dbis.cs108.project.StartMenu	1.25	2	5
ch.unibas.dmi.dbis.cs108.project.ToolsGUI	1.25	3	10
ch.unibas.dmi.dbis.cs108.tests.NetTest	1.12	2	9
ch.unibas.dmi.dbis.cs108.project.CallClient	1.00	1	1
ch.unibas.dmi.dbis.cs108.project.CallServer	1.00	1	1
ch.unibas.dmi.dbis.cs108.project.Point	1.00	1	3
ch.unibas.dmi.dbis.cs108.tests.DataTest	1.00	1	7
Total			**531**
Average	3.90	9.50	22.12

class	CLOC	JLOC	LOC
ch.unibas.dmi.dbis.cs108.project.Data	184	159	836
ch.unibas.dmi.dbis.cs108.project.Field	80	48	467
ch.unibas.dmi.dbis.cs108.project.GFX	127	107	358
ch.unibas.dmi.dbis.cs108.project.Net	123	89	324
ch.unibas.dmi.dbis.cs108.project.ClientThread	95	35	245
ch.unibas.dmi.dbis.cs108.project.Client	85	48	193
ch.unibas.dmi.dbis.cs108.project.LobbyGUI	60	25	169
ch.unibas.dmi.dbis.cs108.project.Tools	47	36	155
ch.unibas.dmi.dbis.cs108.project.EncodeNet	45	11	150
ch.unibas.dmi.dbis.cs108.project.Lobby	37	27	109
ch.unibas.dmi.dbis.cs108.project.HighScore	49	27	108
ch.unibas.dmi.dbis.cs108.project.StartMenu	42	25	102
ch.unibas.dmi.dbis.cs108.project.Lounge	40	25	96
ch.unibas.dmi.dbis.cs108.project.ToolsGUI	41	38	96
ch.unibas.dmi.dbis.cs108.project.Chat	32	19	73
ch.unibas.dmi.dbis.cs108.project.ServerThreac	21	14	67
ch.unibas.dmi.dbis.cs108.project.ManualGUI	17	12	65
ch.unibas.dmi.dbis.cs108.project.Main	16	9	51
ch.unibas.dmi.dbis.cs108.project.Point	22	22	42
ch.unibas.dmi.dbis.cs108.project.Server	16	14	42
ch.unibas.dmi.dbis.cs108.project.CallClient	3	3	9
ch.unibas.dmi.dbis.cs108.project.CallServer	3	3	9
Total	**1'185**	**796**	**3'766**
Average	53.86	36.18	171.18

Abbildung 3 (links): MetricsReloaded Analyse der Komplexität der Klassen

Abbildung 4 (rechts): MetricsReloaded Analyse der Lines of Code und Javadoc

Unsere Logger wird in 11 Files 44-mal eingesetzt. Im Durchschnitt also genau 4-mal pro File

Logger

Klassen	Logging Statements	Lines of Code
Client.java	5	193
ClientThread.java	2	245
Data.java	6	836
EncodeNet.java	2	150
Field.java	2	467
HighScore.java	4	108
Lobby.java	2	109
Main.java	7	51
Net.java	7	324
Server.java	2	42
ServerThread.java	5	67
Summe	**44**	**2592**

5. Diskussion

Die Resultate widerspiegeln unser Konzept durchaus. Unser Ziel war es von Anfang an funktionierenden Code zu schreiben, welcher wiederverwendbar und ausbaubar ist. Dabei haben wir die genauen Zahlen der Tests maximal als Warnung aufgefasst. Falls es einen Statistischen Ausreisser gab, wie z.B. die Klasse Data welche bis zu 1200 Zeilen hatte und uns erst durch die Messungen auffiel, wurde die Klasse um knapp ein Viertel verkürzt.

Wir sind mit den Unittests grundsätzlich zufrieden. Das Grundgerüst unseres Codes wird auf das nötigste getestet. Statistisch hätte es nicht geschadet mehr zu testen, aber die Tests erfüllen ihren Zweck.

Einige Files sind lesbarer als andere. Pro 10 Zeilen Code haben wir im Durchschnitt 3 Zeilen als Kommentar hinzugefügt. Auch hier hätte es nicht geschadet einiges mehr zu Kommentieren damit die Lesbarkeit des Codes garantiert werden kann. Es gibt Bereiche in unserem Code, die, für einen Fremden, möglicherweise schwer lesbar sein könnten. Das Data File ist dafür ein sehr gutes Beispiel. Es gibt sehr viele

Winkelberechnungen, welche mühsam zu verdauen sind. Gleichzeitig haben wir, nach konstruktiver Kritik von unserem Tutor die Client Klasse um ein Vielfaches lesbarer gemacht. Uns ist die Wichtigkeit dieser Veränderung erst später aufgefallen, als wir bemerkten wie einfach nun neue Funktionalitäten eingebaut werden können. Wir konnten uns dadurch viel einfacher in unserem Code wiederfinden.

Unser grösstes Problem, weshalb die Qualität des Codes leiden musste, war der Zeitdruck. Wir haben nicht grossen und schlecht kommentierten Code mit Absicht geschrieben und es lag auch nicht daran, dass wir nicht fähig sind es besser zu machen, aber es war schlicht nicht unsere höchste Priorität auf der Liste. Wir haben primär darauf geschaut, dass die Meilensteine erfüllt sind und wir am Schluss ein Bug freies Spiel erschaffen. Ist Zeit übriggeblieben wurde es genutzt um die Qualität zu verbessern, jedoch geschah dies selten.

Für die Zukunft werden wir sicherlich nicht die gleichen Fehler machen. Wir werden uns von Anfang an das Ziel setzten Code so lesbar wie möglich zu gestalten, damit keine Zeit verloren geht beim Weiterarbeiten. Auch werden wir versuchen Code zu fragmentieren, um riesige Klassen wie Data zu vermeiden. In unserem Fall hatten wir Glück, dass wir wenige Bugs mit dem Schach an sich hatten, aber es hätte sehr schnell schief gehen können.