Software-Qualitätssicherung im Projekt “Rat um Rad”

Einführung

Dieses Dokument ist ein Leitfaden und Nachschlagewerk für das Qualitätsmanagement und die Sicherungsmaßnahmen im Projekt “Rat und Rad”, die durchgeführt wird von der Gruppe ProgRadler, bestehend aus Jonah Sebright, Yaowen Rui, Rahel Kempf und Emanuele Tirendi. Qualitätssicherung ist nicht nur für kurzfristige Aspekte wichtig, wie z.B. um den Code lesbarer zu machen und die Fehlerfindung zu vereinfachen. Es hat auch wichtige langfristige Gesichtspunkte. Code muss Wartungs-Geeignet und die Klassen- und Methoden-Struktur muss zielorientiert und einfach veränderbar geschrieben werden, wobei für letzteres z.B. auch die lines of code per method wichtige Informationen liefern kann. Vor allem ist aber wichtig, dass Qualitätssicherung auch messbar definiert ist, um ihre korrekte Ausführung zu überprüfen und zu gewährleisten.

Merge-Requests sind ein zentraler Punkt in der Qualitätssicherung

Wir sehen das Mergen und den Merge-Request als einen zentralen Baustein beim Qualitätsmanagement und als geeigneten Zeitpunkt, eine umfassende Überprüfung durchzuführen, da mit dem Mergen ein neu geschaffenes Produkt (neuer Code, Zusammensetzung mehrerer Veränderungen) in einem bereits bestehenden Produkt eingebettet wird. Dafür haben wir eine Checkliste erstellt.

- [x] New classes and methods have Javadoc.
- [x] Testable Methods and Classes are tested.
- [x] Project builds (Pipeline).
- [x] All Tests pass (Pipeline).
- [x] Approved and reviewed by a team member.
- [x] Updated documentation documents if needed.
- [x] Code is readable and has no unused code or commented out code.
- [x] Code is properly formatted (Ctrl+Alt+L).

Dabei befolgt die Dokumentation mittels Javadoc übliche Konventionen. Auch das Schreiben des Codes befolgt einige von uns und einige von Google Java Style Guide festgelegten Konventionen. Letztere lassen sich hier finden: https://google.github.io/styleguide/javaguide.html. Bei den Code-Reviews orientieren wir uns an der Checkliste aus dem Dokument de06-cs108-FS23-SW_Qualitaetssicherung auf der Seite 36.

Für alle TeammitgliederInnen hat sich dieses Konzept sehr bewährt und damit wird es auch in künftigen Projekten, an denen wir mitwirken werden, Anwendung finden.

Testing

Es wird angestrebt, alle testbaren Bereiche des Spiels zu testen. Während der Entwicklung werden dafür für viele Bereiche Unit-Tests (mit JUnit) geschrieben. Nach Abschluss einer Teilaufgabe wird die Funktionalität zudem manuell durch die Entwickler getestet (Nach Black- und Whitebox Verfahren).

Wir legen großen Wert auf Unit-Tests. Dies kommt daher, dass Unit-Tests automatisiertes Testen ermöglichen und damit beliebig oft, gezielt und schnell testen können. Darüber hinaus ist das Schreiben von Unit-Tests eine Art des Reviews des eigenen Codes, da man die zu testenden Methoden oft umschreiben, also Testbar-machen muss. Und andererseits, dem Test-driven Development folgend, zwingt das Schreiben eines Unit-Tests dem/der EntwicklerIn, sich Gedanken zur Architektur der zu schreibenden Software-Komponente zu machen.

Messbar wird das Schreiben von Unit-Tests mit dem Jacoco-Test-Report, welches im Thema Messungen weiter behandelt wird und wo auch die Diskussion dazu stattfinden wird.

Ein Hauptunterschied des manuellen Testens zum Unit-Testing ist einerseits, dass es ohne größeren Aufwand schwer möglich ist, gezielt und beliebig oft manuell zu testen. Und dies wird andererseits verstärkt, dass unsere Software zweiteilig aus Client und Server besteht. Wenn man also beispielsweise den Server testen will, gibt es zwei Möglichkeiten:

- Man braucht einen Client, weil man erst anhand des Verhaltens von Server und Client in gemeinsamer Interaktion sichergehen kann, dass der Server funktioniert, wie er sollte. Dies bedingt eine gute Kommunikation und Koordination im Team und eine relativ strikt parallele Entwicklung an der gleichen Komponente der Software auf der Client- und Serverseite.
- Der/die EntwicklerIn simuliert selbst den Client und baut im Server eine Funktionalität ein, durch die der/die EntwicklerIn mit dem Server interagieren kann und so den Client simulieren kann (z.B. über die Konsole). Dies benötigt wiederum viel Zeit.

Damit zeigt sich, dass manuelles Testen, welches unabdingbar für die Softwareentwicklung ist, schnell mit viel Zeit oder aber mit einer gut koordinierten Kommunikation einhergehen kann bzw. muss.

Nach einem sehr guten Start mussten wir feststellen, dass es bei uns beim dritten Meilenstein nicht an der Zeit, aber an der internen Kommunikation mangelte. Wir haben bei uns eine sehr strikte Arbeitsteilung in Client und Server vorgenommen, bei denen die Kommunikation innerhalb der Komponenten sehr gut war, aber zwischen den Komponenten gefehlt hat. Zusammen hat dies dazu geführt, dass das Zusammenspiel des Servers und Clients bei der Abgabe des Meilensteins teilweise nicht funktioniert hat, wie es sollte, weil sie nie in ausreichendem Maß zusammen getestet wurden, sondern nur jede Komponente individuell. Dies konnten wir ab dem vierten Meilenstein dann folgendermaßen lösen:

- Aufgaben wurden dann nicht mehr in Server und Client unterteilt, sondern nach Funktionalität der gesamten Software. Dabei haben dann immer eine für die Serverseite und eine für die Clientseite zuständige Person zusammen an derselben Funktionalität gearbeitet und sie gemeinsam getestet.
- Die Formulierung der Aufgaben haben wir konkreter gefasst. Dies bedeutet konkret, dass die Aufgabe nicht so formuliert wurde, dass eine bestimmte Komponente programmiert werden soll, sondern dass die Komponente am Ende funktioniert mit Bezug auf ihre Umgebung. Dies verhindert eine Arbeitshaltung, nach der man zufrieden ist, wenn man die Komponente programmiert und isoliert getestet hat (und damit die Umgebung nicht miteinbezogen hat). Die Aufgabe ist dann fertig, wenn die Komponente in der Umgebung, in der sie funktionieren sollte, auch funktioniert. Erklärt wird das hier am Beispiel der Funktionalität “Updaten einer Game-Instanz durch einem Spielzug und anschließendem Informieren der User”. Diese Funktionalität funktioniert also nich nur, wenn intern im Server das Game korrekt geupdated wird, sondern wenn wir bestätigen können, dass auf der Client-Seite (also die Umgebung) die entsprechenden User das Packet mit dem richtigen (geupdateten) Game erhalten.
- Wir haben die Verantwortung der einzelnen EntwicklerInnen gestärkt. Alle oben genannten Probleme können in einem EntwicklerInnen-Team umgangen werden, wenn sich die einzelnen EntwicklerInnen genug verantwortlich fühlen, einerseits in der Gruppe anzusprechen, dass diese Probleme da sind und es Veränderung bedarf und andererseits dann auch die Initiative ergreifen, Veränderungen in Gang zu bringen.

Ein besonderes Augenmerk haben wir im Zuge des im Meilenstein 4 durchzuführenden Testens einer Komponente unseres Spieles auf den Game-Service gelegt, da dieser als zentraler Baustein des Spiels einen Grossteil der Programmlogik mitbestimmt und prägt und dieser sowohl Teil des Clients (in Form der Klasse GameService) als auch des Servers (in Form der Klasse GameService und GameServiceUtils) ist. Die Coverage wird im Themenbereich Messungen visualisiert, da dieser auch mit Jacoco ausgewertet wurde.

Logging

Für das Logging verwenden wir Log4j. Wir haben das Loggen für die Übersicht in zwei Dokumente unterteilt: Der Ping-Pong-Prozess in einer Log-File und die anderen Logs in einer anderen Log-File. Zudem

unterscheiden wir zwischen den unterschiedlichen Log4J-Levels. Dies hilft uns, die geloggten Informationen nach Wichtigkeit zu filtern. So soll eine gezielte Fehlersuche in unterschiedlichen Bereichen und von unterschiedlichen Personen (Anwender*innen, Entwickler*innen) ermöglicht werden. Error-Logs geben wir zusätzlich in der Commandline aus. Das Loggen hat uns die Arbeit um ein Vielfaches erleichtert bei der Fehlersuche. Diese werden wir auch in Zukunft weiterhin als zentrales Hilfsmittel bei der Entwicklung anwenden.

Teamkultur

Kommunikation mit Respekt, Fehlertoleranz und gemeinsamer Verantwortung sind zentrale Punkte in unserer Teamkultur. Ebenfalls fördern wir die Teamkultur durch Pair Programming (mit IntelliJ “Code with Me”).

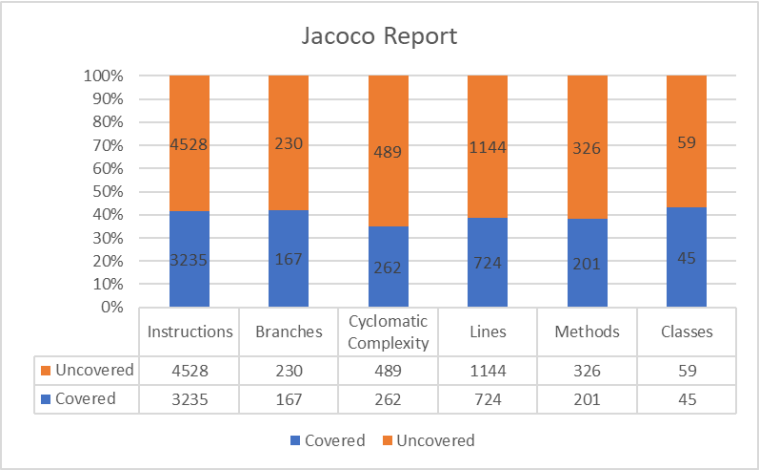
Da Projektmanagement, Gruppenmanagement und Kommunikation zentrale Bestandteile des Erfolges eines Projektes sind, haben wir viel Wert darauf gelegt. Die konkrete Umsetzung erfolgte folgendermaßen:

- Auf Gantt-Project haben wir die gröbere Planung der Meilensteine vorgenommen, damit wir einen Rahmen haben, an dem wir uns bei der Durchführung des Projektes orientieren können und damit uns grosse Herausforderungen schon früh bewusst werden und wir uns darauf vorbereiten können.
- In regelmässigen Sitzungen haben wir dann die konkreten To-Dos verteilt, wichtige Fragestellungen und Entscheidungen besprochen und Ideen gesammelt. Dabei haben wir die To-Dos so konkret wie möglich formuliert. Dabei hat sich eine sehr konkrete Formulierung in Zahlen und Fakten der To-Dos bewährt. Z.B. lässt sich immer ein Datum angeben, bis wann die Aufgabe fertig sein sollte.
- To-Dos, Entscheidungen, Herausforderungen und wichtige Informationen haben wir in unserem Projekttagebuch gesammelt. Neben den vielen Vorteilen, die solch ein Projekttagebuch für das Projekt selbst hat (jeder weiss, wer was wann tut, man kann gefällte Entscheidungen nachschlagen, es gibt jeder Sitzung eine Struktur, etc.), hat es auch langfristige Vorteile. Aus unserem Projekttagebuch können wir für künftige Projekte lernen, da es unser Projekt dokumentiert und somit eine Auswertung ermöglicht.
- Falls Bugs während des Code Review oder während des Ausprobierens auffielen, wurden sie reported.

Zusammenfassend sind wir sehr zufrieden mit diesem Teil des Qualitätsmanagements und werden es, wie hier beschrieben, weiterhin verwenden. Es gab einige Herausforderungen in der Kommunikation (siehe Thema Testen) und im Nachhinein hätten wir die Planung auf Gantt-Project etwas ausführlicher gemacht.

Messungen

Messungen zum Meilenstein 3: 16.04.2023, 19:00, Änderungen am Code wurden Vorgenommen bis zur Abgabe



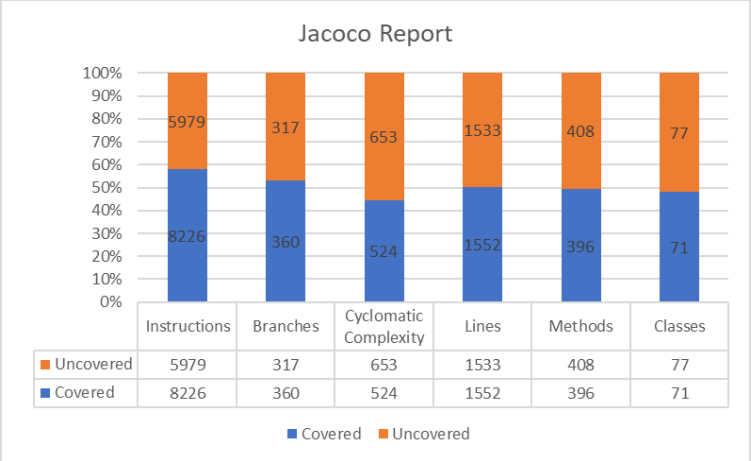
Verhältnis Logging-Statements zu Lines of Code:

Log-Statements	Lines of Code	Verhältnis
15	5’766	0.26%

Metriken:

	Lines of Code per Class	Lines of Code per Method	Lines of Javadoc per Class
Maximum	284	58	41
Minimum	5	1	0
Average	45.40	8.22	5.38
Total Lines	5’766	4’825	683
Total Classes/Methods	127	587	127

Messungen zum Meilenstein 4: 30.04.2023, 15:00, Änderungen am Code wurden Vorgenommen bis zur Abgabe



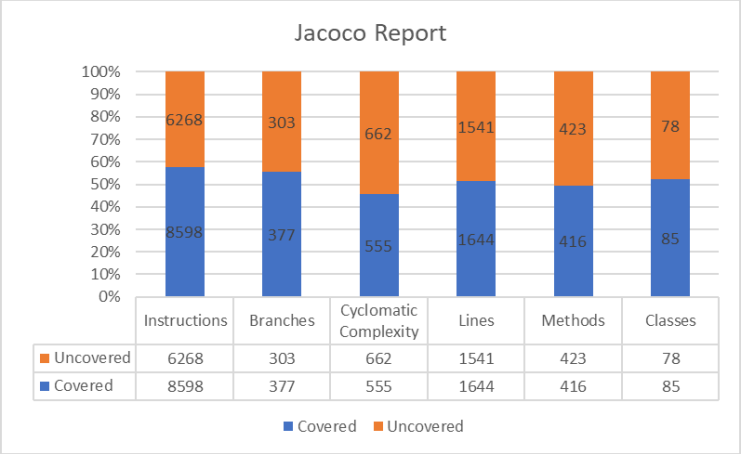
Verhältnis Logging-Statements zu Lines of Code:

Log-Statements	Lines of Code	Verhältnis
36	9’167	0.39%

Metriken:

	Lines of Code per Class	Lines of Code per Method	Lines of Javadoc per Class
Maximum	333	72	41
Minimum	8	1	0
Average	55.22	9.07	5.19
Total Lines	9’167	7’719	861
Total Classes/Methods	166	851	166

Messungen zum Meilenstein 5: 11.05.2023, 10:00, Änderungen am Code wurden Vorgenommen bis zur Abgabe



Verhältnis Logging-Statements zu Lines of Code

Log-Statements	Lines of Code	Verhältnis
50	9’703	0.51%

Metriken:

	Lines of Code per Class	Lines of Code per Method	Lines of Javadoc per Class
Maximum	401	90	40
Minimum	8	1	0
Average	53.91	9.12	4.81
Total Lines	9’703	8’123	865
Total Classes/Methods	180	891	180

Im Folgenden werden die erhobenen Daten detailliert und über die Zeit hinweg betrachtet und diskutiert.

Test-Coverage:

Nehmen wir die “Instructions” als Referenz für die allgemeine Entwicklung unserer Coverage. Wir können einen stetigen Anstieg über die Zeit der relativen Coverage beobachten, vor Allem vom Meilenstein 3 zum Meilenstein 4, welche sich von knapp über 40% beim dritten Meilenstein bis knapp unter 60% beim vierten und fünften Meilenstein entwickelt hat. Wir erklären uns die hohe (absolute) Anzahl an Tests, welche wir bereits beim dritten Meilenstein hatten, damit, dass wir seit Beginn damit begonnen haben, Uni-Tests zu schreiben. Warum sich diese Zahl vom dritten zum vierten Meilenstein (relativ) erhöht hat, erklären wir uns damit, dass wir nach den Herausforderungen beim dritten Meilenstein weitere Tests hinzugefügt haben, um die Fehler, von denen wir nach dem dritten Meilenstein hatten, zu finden. Mit 3’867 Zeilen im Test-Modul von 12’066 gesamten Zeilen (Stand: 12.05, nur Lines innerhalb der Methoden berücksichtigt), besteht unser Source-Code aus 32% Unit-Tests, 210 an der Zahl. Mit einer solch hohen Zahl an Unit-Tests ging für uns natürlich auch eine große Zeitinvestition einher, welche für einige EntwicklerInnen sehr herausfordernd war. Dies war aber in unseren Augen eine lohnenswerte Investition, da wir uns dadurch viele Vorteile verschafft haben. (Siehe Themengebiet Testing). Damit sind wir mit dieser Zahl an Unit-Tests sehr zufrieden und werden dies auch für künftige Projekte beibehalten.

Insgesamt sieht man, dass alle anderen Gesichtspunkte des Reports, also “Branches”, “Cyclomatic Complexity”, “Lines”, “Methods” und “Classes” linear zu “Instructions” wachsen. Die Fluktuationen sind da eher gering und somit nicht wirklich erwähnenswert. Eine Eigenheit ist, dass der niedrigste Gesichtspunkt die

zyklomatische Komplexität ist. Dies kommt daher, dass das GUI bei unserem Projekt eine hohe Dichte an zyklomatischer Komplexität hat. Generell ist es aber für das GUI nicht selten, dass es weniger stark getestet wird als andere Komponenten einer Software. So erklärt sich, warum die zyklomatische Komplexität bei uns das am wenigsten getestete Gesichtspunkt ist.

Logging:

Im dritten Meilenstein hatten wir mit 0.26% sehr wenige Logging-Statements. Dies hat damit zusammengehangen, dass wir in diesem Meilenstein erst mit dem Logging begonnen haben und es damit nicht so ausgebaut war. Bis zum vierten Meilenstein hat sich die absolute Anzahl der Logging-Statements mehr als verdoppelt, da wir in diesem Meilenstein einige Probleme, welche beim dritten Meilenstein aufgekommen sind, lösen mussten (siehe Thema Testing). Am Ende haben wir genug Logging-Statements, sodass eine effiziente Ursachensuche für aufkommende Fehler gewährleistet ist.

Lines of Code per Class und Lines of Code per Method:

Über das ganze Projekt haben wir eine konstant bleibende, niedrige Anzahl an Lines of Code per Method und per Class gehabt. Damit sind wir sehr zufrieden und wünschen uns keineswegs eine höhere Anzahl. Es zeigt uns, dass die Aufgaben, die eine Methode oder eine Klasse übernehmen, überschaubar sind. Dies ist für die Lesbarkeit des Codes, für die Wartung des Codes, für das Testen des Codes und für das Refactoring sehr wichtig. Die Klassen und Methoden sind überschaubar und nicht zu komplex.

Lines of Javadoc per Class:

Wir haben durch das ganze Projekt (Tendenz sogar etwas absteigend) ein Verhältnis von etwa 5 Javadoc-Lines pro Klasse gehabt. Von der Zahl her hört sich das nach wenig an. Wir rechtfertigen das aber damit, dass dadurch, dass wir in unserem Programm eine sehr strukturierte Arbeitsteilung in Klassen und Methoden haben und jede Methode und Klasse überschaubar viele Arbeiten übernimmt, der Code oft selbsterklärend ist. Denn mit Methoden, die nur eine Aufgabe erledigen, ist auch die intuitive Benennung der Methoden sehr einfach und damit auch das Verständnis, was die Methoden schließlich machen. Und zusätzlich kommt noch, dass unsere Tests, von denen wir viele haben, selbst wenig Javadoc haben, da diese sehr lange Namen haben, die genau erklären, wofür die Tests da sind. Zusätzlich sind natürlich noch im ganzen Projekt weitere Kommentare enthalten.

Test der wichtigen Kernkomponente: Game-Service:

Dies ist eine Zusammenfassung der drei Klassen: GameService im Client, GameService im Server und GameServiceUtil im Server. Wir sind sehr zufrieden mit dem vollständigen Testen dieser wichtigen Komponente des Spiels: