Qualitätssicherungskonzept JavaTiles

Organisatorische Massnahmen

Um eine effektive Zusammenarbeit zu ermöglichen, haben wir ein Netzwerkprotokoll implementiert, das klare Befehle zwischen Server und Client ermöglicht. Dadurch können wir flexibler arbeiten, indem zwei Teammitglieder am Server und zwei am Client arbeiten.
Darüber hinaus haben wir mithilfe von Ganttproject die Aufgaben klar verteilt und festgelegt, wer welche Aufgabe bis wann erledigen muss.
Ausserdem haben wir eine WhatsApp-Gruppe gebildet, um uns gegenseitig immer auf dem neuesten Stand zu halten, sei es über neue Probleme, die aufgetaucht sind, sei es, um spontan ein Meeting zu organisieren oder auch um neue Ideen für das Spiel einzubringen.
Durch die Commits in unser Git-Repository konnten wir immer sehen, was genau geändert wurde. Da die Commit-Nachrichten Anfangs nicht aufeinander abgestimmt waren, haben wir uns dafür entschieden, eine Datei mit von uns besprochenen Commit-Konventionen zu erstellen.

Technische Massnahmen

Die Integration von Checkstyle in IntelliJ ist ein wichtiger Bestandteil unserer QA-Massnahmen, um die Einhaltung von Coding-Standards sicherzustellen. Durch regelmäßige Überprüfungen während des Entwicklungsprozesses gewährleisten wir eine konsistente Codequalität. Dies fördert eine verbesserte Zusammenarbeit im Team und ermöglicht eine frühzeitige Identifizierung potenzieller Probleme.

Zusätzlich zum Checkstyle haben wir auch das Plug-in Tabnine installiert, welches ein KI-gesteuertes Codevervollständigungstool ist. Tabnine soll uns helfen, unseren Code schneller zu schreiben und auch weniger Fehler zu machen.

Um mögliche Fehler schneller zu finden und zu beheben, benutzen wir die Library Apache Log4j 2. Diese ermöglicht es uns, den Zustand unserer Anwendung detailliert zu überwachen und zu protokollieren. Durch präzises und konfigurierbares Logging auf verschiedenen Ebenen (DEBUG, INFO, WARN, ERROR) können wir die Ursachen von Problemen schnell identifizieren und beheben, ohne dabei auf invasive Debugging-Methoden angewiesen zu sein. Mit Log4j 2 haben wir die Möglichkeit, unsere Log-Nachrichten flexibel zu gestalten, was von unschätzbarem Wert ist, wenn es darum geht, den Überblick über den Betriebszustand unserer Anwendung in Echtzeit zu behalten. Die Konfiguration von Log4j 2 lässt sich nahtlos an unsere Bedürfnisse anpassen, und durch die Nutzung von asynchronen Loggern minimieren wir Leistungseinbußen, die durch das Logging verursacht werden könnten.

Analytische Massnahmen

Problematische Codeabschnitte wurden im Rahmen eines gemeinsamen Teamdurchlaufs überprüft. Neue Änderungen oder Implementierungen wurden stets mit dem gesamten Team kommuniziert, um sicherzustellen, dass jedes Mitglied auf dem neuesten Stand ist und vor allem den Code versteht. Es liegt uns sehr am Herzen, dass jedes Teammitglied gleichermaßen in das Projekt eingebunden ist, damit das Beste aus allen Einzelbeiträgen einfließen kann.
Eigene Codeabschnitte der Teammitglieder wurden kritisch geprüft und diskutiert, um die Qualität und Kohärenz des Gesamtprojekts sicherzustellen. Immer wieder haben wir Simulationen an unserem Code durchgeführt, um mögliche Schwachstellen ausfindig zu machen. Daraufhin konnten wir entsprechende Änderungen durchführen, um unser Programm stetig zu verbessern. Wenn mal jemand für sich selbst den Code getestet hat und einen Fehler entdeckt hat, hat diese Person dann auch einen Bug Report geschrieben. So konnten wir uns gegenseitig auf Fehler aufmerksam machen und diese auch schnell beheben.

Das Checkstyle Plug-in hilft uns auch den Code zu analysieren, indem es beispielsweise anzeigt, falls Libraries nicht gebraucht werden oder wenn eine Zeile zu lang ist, um auch den Code möglichst lesbar zu halten oder auch auf fehlende Javadoc-Kommentare aufmerksam zu machen.

Unit Tests

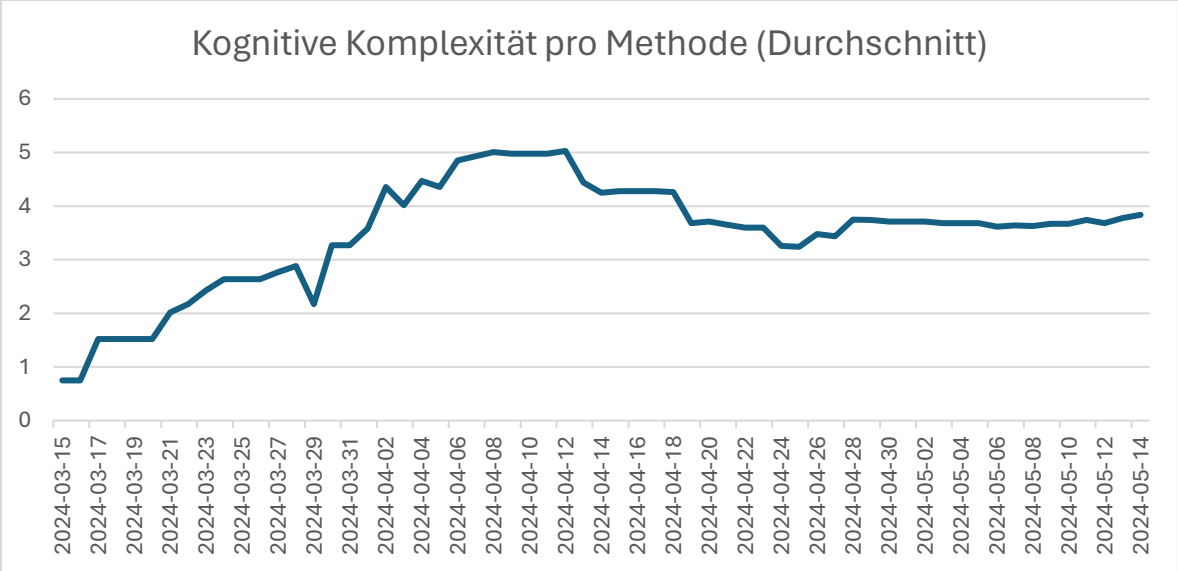
Als es um die Implementierung der Unit Tests ging, haben wir uns dafür entschieden, die Spiellogik zu testen, da wir unbedingt wollten, dass dieser Aspekt des Spiels fehlerfrei funktioniert. Wir haben während des Schreibens der Unit Tests tatsächlich auch noch Fehler in der Erkennung der Gewinnkonfiguration finden können, da wir nun nicht mehr das ganze Spiel durchspielen mussten, um eine Gewinnkonfiguration zu erhalten. Dadurch konnten wir einfach selbst Gewinnkonfigurationen in den Testmethoden herstellen. Dies ermöglichte es uns schneller mehr Randfälle zu testen und wir dachten während des Tests Schreibens auch viel mehr an die Randfälle, die eintreten könnten.

Unsere ausgesuchte Metrik (Durchschnitt der Kognitiven Komplexität pro Methode) hat uns bereits dazu gebracht unseren Code umzustrukturieren in kleinere Methoden, was uns geholfen hat die Tests effektiver zu schreiben und auch um einiges verständlicher zu gestalten.

Eine grosse Hilfe und definitiv empfehlenswert beim Schreiben der Tests war die Library Mockito. Mit Mockito kann man unter anderem Klassen nachahmen, was uns besonders bei der ClientThread Klasse geholfen hat, da wir sonst eine Verbindung zwischen Server und Client herstellen müssten, was auch nicht im Sinne unserer Tests war. Ausserdem konnten so Verbindungsfehler ausgeschlossen werden. Auch kann man mit Mockito verifizieren, wie oft eine Methode aufgerufen wird und auch die Reihenfolge der Methodenaufrufe überprüfen. Mockito unterstützt auch einen BDD (Behavior-Driven Development) Ansatz für Tests, bei dem Tests in einer natürlichen Sprache geschrieben werden können, die die Absichten hinter dem Code verdeutlicht und die Lesbarkeit verbessert.

Nach dem erstellen der Tests halfen sie uns weiter, in dem man sofort erfuhr, ob Fehler im Code enthalten sind. Von Fehlern, die von den Unittests aufgegriffen wurden, konnten wir mit git bisect ganz einfach den Ursprung finden.

Metrik



Für unser Projekt haben wir als Metrik den Durchschnitt der kognitiven Komplexitäten der Methoden von metricsReloaded (dort genannt cognitive complexity) von unserer ganzen Codebase gewählt.

Bis zum Meilenstein 3 (Mitte des Graphes) stieg diese Metrik ziemlich konstant an. Von dann an haben wir uns vorgenommen, diesen Wert zu senken, in dem wir grosse Methoden in kleinere, sinnvolle Methoden unterteilen. Dabei hatten wir aber einige Schwierigkeiten, da es oft gute Gründe dafür gab, dass diese Methoden so gross waren.

Trotzdem ist dies uns, wie im Graph zu sehen ist, recht gut gelungen; unsere durchschnittliche kognitive Komplexität ist von 5 auf 3.7 gesunken.

Diskussion der Resultate der Verschiedenen QA-Aspekte:

Die organisatorischen Maßnahmen, die wir eingeführt haben, ermöglichten eine optimale Aufteilung der Arbeit. Dadurch wurde sichergestellt, dass niemandem langweilig wurde, während gleichzeitig Überarbeitung vermieden wurde. Diese Strukturen halfen auch dabei, jedes Teammitglied stets auf dem Laufenden zu halten. Die Einführung und Einhaltung der Commit-Konventionen verbesserte die Lesbarkeit und das Verständnis der Commits erheblich, was die Zusammenarbeit erleichterte.

Durch die Implementierung unserer technischen Maßnahmen konnten wir unseren Code effizienter schreiben und das Debugging vereinfachen.

Die analytischen Maßnahmen spielten eine entscheidende Rolle bei der Identifizierung von Schwachstellen im Spiel, besonders als der Code umfangreicher und anfälliger für Fehler wurde. Diese Maßnahmen sorgten auch dafür, dass alle Teammitglieder über die neuesten Entwicklungen informiert blieben und die Qualität des Codes durchgängig hochgehalten wurde.

Die Unit Tests ermöglichten problemloses Refactoring des betreffenden Codes. Sie halfen uns, viele Randfälle zu testen und dadurch die Qualität unseres Spiels kontinuierlich zu verbessern.