Code Reviews

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de

Version: 0.4.1.2

Folien: [HTML] https://delors.github.io/lab-codereviews/folien.de.md.html
[PDF] https://delors.github.io/lab-codereviews/folien.de.md.html.pdf

Fehler melden: https://github.com/Delors/delors.github.io/issues

KI Verwendung: Bei der Erstellung der Unterlagen wurden KI Assistenten (insbesondere Claude aber ggf. auch ChatGPT, Ollama mit Gemma/LLama/Qwen oder OpenCode mit Kimi/Qwen/Deepseek...) unterstützend eingesetzt. Dies erfolgte insbesondere zur Unterstützung bei der Generierung von Grafiken (d. h. SVG Dateien), oder um sich Übersichtstabellen generieren zu lassen. Weiterhin wurde KI zur allgemeinen Qualitätssicherung eingesetzt. Inhalte, die ggf. von der KI vorgeschlagen wurden, wurden im Falle der Übernahme explizit validiert und angepasst.

Code Reviews und Software Qualität

? Frage

Sie bekommen morgen ein Projekt mit 15.000 Zeilen Code übergeben, das Sie weiterentwickeln sollen. Es ist keine Dokumentation vorhanden oder die Dokumentation ist veraltet oder die Dokumentation wurde KI generiert und die Originalautoren sind nicht mehr erreichbar.

- Wie fangen Sie an?
- Wovon hängt Ihr Aufwand ab?
- Was hätte geholfen?

> [...] Through a case study [...], we find that **code review coverage, participation, and expertise share a significant link with software quality.**[1] Hence, our results empirically confirm the intuition that poorly-reviewed code has a negative impact on software quality in large systems [...].
>
> —McIntosh, S., Kamei, Y., Adams, B. et al. An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering 21, 2146–2189 (2016).

[1] Markup nachträglich hinzugefügt.

Einführung - Was sind Code Reviews?

Definition: Code Review

Die systematische Untersuchung von Quellcode durch Dritte — Menschen, KI-Tools oder beide in Kombination — mit dem primären Ziel, Probleme zu identifizieren, die der Autor - Mensch oder KI - übersehen hat.

Typischer Ablauf

1. *Bereitstellung des Codes* - dies umfasst eine Beschreibung des Kontexts und ggf. des konkreten Ziels des Reviews.

   In professionellen Projekten erfolgen Code Reviews für gewöhnlich im Kontext so genannter Git(Hub) Pull Requests. Dies ist unabhängig davon, ob es sich um Open-Source oder kommerzielle/closed-source Software handelt. Sie können diese gerne im Rahmen des durchzuführenden Code Reviews nutzen. Die Dokumentation ist dann „einfach“ die Zusammenstellung der Pull Requests.

2. *Analyse des Codes* - dies erfolgt entweder manuell, werkzeuggestützt (dann insbesondere KI-gestützt) oder durch eine Kombination von beidem.
3. *Dokumentation relevanter Probleme/Aspekte* - fragwürdige oder problematische Codestellen werden mit Hilfe von einzelnen, nachvollziehbaren Anmerkungen dokumentiert.

Im Nachgang an das eigentliche Code Review erfolgt die *Umsetzung der Änderungen* durch die Originalautoren, wobei es ggf. zu Nachfragen/Diskussionen kommen kann.

Primäre Ziele:

- (insbesondere fachliche) Fehler finden, die dem Autor/den Autoren nicht aufgefallen sind
- Verständlichkeit und Wartbarkeit des Codes sicherstellen

Sekundäre Ziele:

- Wissen über die Codebasis im Team verteilen
- Gemeinsames Verständnis über Qualitätsstandards und Konventionen entwickeln
- Fähigkeiten weiterentwickeln durch das bewusste Lesen von fremdem Code

Menschliches Review vs. Tool/KI-gestütztes Review

Ziele werkzeuggestützter Reviews

- *Klassische Tools (Linter, Formatter, Analyzer):*
  - Syntaxfehler erkennen
  - Konsistente Formatierung erzwingen
  - Einhaltung konfigurierbarer Regeln (z. B. max. Verschachtelungstiefe)
  - Leichtgewichtige statische Analysen

  Die Art der benötigten klassischen Tools ist immer stark von der/den verwendeten Programmiersprachen abhängig. Zum Beispiel ist bei Java-Projekten eine explizite Syntaxprüfung nicht notwendig, da eine Standardvoraussetzung eines Code Reviews ist, dass der Code zumindest kompiliert. Bei Skriptsprachen wie Python oder JavaScript ist dies jedoch ggf. sinnvoll.

- *KI-gestützte Tools:*
  - Fehlendes Error-Handling aufdecken
  - Erkennung komplexerer Bugs, Anti-Patterns sowie Sicherheitslücken
  - Vereinfachungen und Refactorings

  KI-gestützte Tools können weiterhin bei Zusammenfassungen und Erklärungen von Code helfen, um den weiteren Code-Review-Prozess durch Menschen zu unterstützen.

Ziele menschlicher Reviews

- *Korrektheit:*
  - Passt die Logik zur fachlichen Anforderung?
  - Werden Sonderfälle und Randbedingungen korrekt behandelt?
- *Verständlichkeit und Wartbarkeit:*
  - Sind Bezeichner treffend?
  - Ist der Datenfluss (über Modulgrenzen hinweg) nachvollziehbar?
  - Sind Abstraktionen angemessen?
  - Ist der Code konform mit der bestehenden Architektur?

KI-Tools wie GitHub Copilot Code Review ergänzen menschliche Reviews, haben jedoch keinen oder nur begrenzten Zugang zum fachlichen Kontext, zu den Architekturentscheidungen des Teams oder zum impliziten Wissen über die Anwendungsdomäne. Gerade bei (größeren) Web-Anwendungen, bei denen Logik über Client, Server und Stylesheets verteilt ist, fehlt KI-Tools häufig das Verständnis für das Zusammenspiel der Schichten.

Voraussetzung für Code Reviews

Checkliste

✓der Code ist konsistent formatiert

(*Code Prettifier* wurden ausgeführt)

✓triviale Fehler wurden bereits behoben

(*Linting* wurde durchgeführt)

✓der Code kompiliert bzw. ist frei von Syntaxfehlern

✓der Code wurde getestet

(Wenn vorhanden, dann wurden Unit-/Integrations-Tests erfolgreich ausgeführt; falls nicht, dann wurde durch die Autoren zumindest ein manueller Test durchgeführt. In professionellen Projekten sind Tests unverzichtbar.)

✓Architektur und zentrale Designentscheidungen sind dem Reviewer bekannt

(Im Idealfall sind zentrale Aspekte zum Beispiel durch C4-Diagramme oder UML-Sequenzdiagramme bereits dokumentiert.)

Gutes Feedback formulieren

Starkes Feedback

★„Der Funktionsname `process()` sagt nicht, *was* verarbeitet wird — wäre z. B. `validateFormInput()` treffender?"

★„Hier wird `catch(e) {}` verwendet, aber der Fehler wird nicht geloggt. Falls die Validierung fehlschlägt, wäre es schwer, den Grund zu finden."

★„Die Berechnung in Zeile 42–58 könnte in eine eigene Funktion extrahiert werden, um die `handleSubmit()`-Methode kürzer und lesbarer zu machen."

★„Mir ist aufgefallen, dass `rem` und `px` gemischt werden — gibt es dafür einen Grund, oder sollten wir uns auf `rem` einigen?"

★„Die Validierungslogik liegt sowohl in `form.js` als auch `api.js` — wäre es sinnvoll, sie an einer Stelle zu bündeln, um Inkonsistenzen zu vermeiden?"

▲ Achtung!

Ein Issue sollte:

1. *eine konkrete Stelle* im Code benennen,
2. das *Problem* beschreiben und
3. idealerweise einen *Verbesserungsvorschlag* machen.

Schwaches, nicht-konstruktives, herabwertendes Feedback

✗„Das muss anders."

✗„Falsch."

✗„Das sollte man besser mit einer Map lösen."

(Es muss immer das Warum erkenntlich werden und auch der Codeabschnitt.)

✗„Warum hast du das so gemacht?"

(Falls eine negative Konnotation mitschwingt; es kann auch eine ehrliche Frage sein.)

Vorgehen beim Review

Für Reviewer

- Das Projekt/Produkt kurz vorführen lassen.
- Zuerst die Gesamtstruktur verstehen, bevor man in Details einsteigt.
  (Zum Beispiel im C4-Modell sich von oben nach unten durcharbeiten.)
- Immer die Frage stellen: „Würde ich diesen Code in drei Monaten noch verstehen?"
- Auch positives Feedback geben, wenn etwas gut gelöst ist.
  (Es kann für einen Einstieg ggf. hilfreich sein mit positivem Feedback zu starten.)
- Keine Annahmen treffen! Nachfragen.
- Notieren, was man an der eigenen Codebasis verbessern/überprüfen könnte.

Für Autoren

- Sollte Architekturdokumentation verfügbar sein, diese vor dem Review zur Verfügung stellen.
- Vor dem Review den Reviewern eine Einführung in Struktur und zentrale Entscheidungen geben.
- Nachfragen als Signal verstehen: Wenn der Reviewer etwas nicht versteht, werden es andere auch nicht.
- Nicht jedes Issue erfordert eine Codeänderung — manchmal reicht ein erklärender Kommentar im Code oder eine Ergänzung in der Architekturdokumentation.

◆ Bemerkung

Sie können als gesamtes Team alle Teile gemeinsam begutachten oder sich ggf. in themenspezifische (Frontend-CSS/HTML, Frontend JavaScript, Backend, ...) Subteams aufteilen — je nach Teamaufteilung und Fertigkeiten der Teammitglieder.

Sollten Sie im Nachgang Code Reviews als Praxis innerhalb Ihres Teams etablieren, dann sollten Sie darauf achten, dass Sie (kleine) fokussierte Reviews durchführen und ggf. auch das Ziel klar ist: Zum Beispiel allgemeine Code-Qualität, Sicherheitsprobleme, Architekturkonformität. Es kann ggf. helfen, auf Checklisten zurückzugreifen, um sicherzustellen, dass die Qualität der Code Reviews über alle Teammitglieder hinweg ein konsistentes Niveau erreicht.

Im Idealfall etablieren Sie die Praxis für jedes einzelne Feature einen Pull-Request zu stellen.

1. Technologiespezifische Reviews

Code Review des Build-Prozesses und der Projektstruktur

Exemplarische Kriterien

- *Einstieg und Reproduzierbarkeit*
  - Gibt es eine README.md mit klaren Anweisungen zum Bauen und Starten des Projekts?
  - Lässt sich das Projekt mit wenigen Schritten lokal aufsetzen und starten?
  - Sind alle Abhängigkeiten fixiert?
    (Im Falle von Node-/JavaScript-basierten Projekten zum Beispiel über `package-lock.json`.)
- *Konfiguration und Tooling*
  - Sind Linter und Formatter konfiguriert und über ein Skript aufrufbar?
    (Z. B. `npm run lint`.)
  - Gibt es eine klare Trennung von Entwicklungs- und Produktionsabhängigkeiten?
  - Werden Umgebungsvariablen sauber verwaltet (z. B. über `.env`-Dateien mit zugehörigem `.env.example`)?
- *Projektstruktur*
  - Ist die Verzeichnisstruktur nachvollziehbar und konsistent?
  - Spiegelt die Ordnerstruktur die Architektur wider?
  - Sind keine generierten Dateien oder Abhängigkeiten im Repository?
    (Zum Beispiel: `node_modules`, `dist`, `target`.)

Code Review von CSS

Exemplarische Kriterien

- *Namensgebung und Konsistenz*
  - Konsistente Schreibweise von CSS-Custom-Properties, Klassennamen und IDs
  - Sind die Namen verständlich und sprechend?

    (Korrektes Deutsch oder Englisch; konsistente Sprachnutzung.)
  - Keine Redundanzen, keine unnötigen Style-Definitionen
- *Selektoren und Struktur*
  - Einfachheit und Korrektheit von Selektoren (z. B. keine Verwendung von `!important`)
  - Sind komplizierte Selektoren — falls notwendig — dokumentiert und nachvollziehbar?
  - Wird CSS-Nesting verwendet, um die Struktur der Styles zu verdeutlichen?
  - Werden CSS-Layer (`@layer`) verwendet, um die Styles zu organisieren, die Wartbarkeit zu erhöhen und Spezifizitäts-probleme zu vermeiden?
- *Layout und Responsiveness*
  - Einfachheit des Layouts (dies muss ggf. im Zusammenspiel mit dem HTML-Code geprüft werden)
  - Gibt es ein klares Vorgehen (Mobile-first oder Desktop-first)?
  - Konsistente Verwendung von Größeneinheiten (`em`, `rem`, …) und Farbwerten
- *Modernes CSS*
  - Werden CSS-Custom-Properties sinnvoll verwendet, um ein leicht anpassbares Design zu ermöglichen?
  - *Container Queries* statt reiner Viewport-basierter Media Queries, wenn Komponenten in unterschiedlichen Kontexten wiederverwendet werden
  - `:has()` als Eltern-Selektor anstelle von JavaScript-basierten Workarounds oder zusätzlichen Klassen
  - *Logical Properties* (`margin-inline`, `padding-block` etc.) für bessere Internationalisierbarkeit
  - `color-mix()` und `oklch()` für konsistente Farbschemata

Code Reviews von HTML

Exemplarische Kriterien

- *Struktur und Semantik*
  - Ist die Struktur einfach nachvollziehbar oder übermäßig komplex (z. B. tief verschachtelte DIVs)?
  - Wird primär semantisches HTML genutzt (d. h. beschränkt sich die Nutzung von `<span>` und `<div>` auf die optische Gestaltung)?
  - Sinnvolle Strukturierung der Überschriftenhierarchie (`<h1>` bis `<h6>`)
  - Werden `meta`-Tags für die Spezifikation des Viewports sowie weiterer Metainformationen verwendet?
- *Trennung von Inhalt, Darstellung und Verhalten*
  - Keine Inline-Styles
  - Kein Inline-JavaScript
  - Keine Verwendung von `<br>` für Layoutzwecke
- *Accessibility*

  Führen Sie eine Bewertung der Barrierefreiheit nur durch, wenn es im Kontext des Projekts gelehrt wurde bzw. relevant/gefordert ist.

  - Haben Bilder ein `alt`-Attribut?
  - Sind Formularfelder mit `<label>` verknüpft?
  - Ist die Seite vollständig per Tastatur bedienbar?
  - Sind Farbkontraste ausreichend (WCAG AA)?
  - Werden ARIA-Attribute sinnvoll verwendet — und nur dort, wo semantisches HTML nicht ausreicht?
  - Sind Links inhaltlich nachvollziehbar (kein „hier klicken")?
- *IDs und Eingabevalidierung*
  - Sind HTML-only IDs (d. h. solche, die nicht für CSS-Styling verwendet werden) verständlich und konsistent vergeben?
  - Werden Eingaben clientseitig geprüft — über HTML-Validierungsattribute (`required`, `pattern`, `type`) oder, wenn nötig, per JavaScript?

Code Reviews von JavaScript

Exemplarische Kriterien

- *Korrektheit und Fehlerbehandlung*
  - Macht der Code, was er vorgibt zu tun (d. h. sind Bezeichner korrekt und sinnvoll)?
  - Werden Sonderfälle, Fehlerzustände und `null`/`undefined` korrekt behandelt?
  - Ist die Fehlerbehandlung konsistent — werden keine Fehler verschluckt und ist das Error-Logging aussagekräftig?
  - Werden Promises korrekt behandelt (keine fehlenden `.catch()` oder `try`/`catch` bei `async`/`await`)?
  - Werden Eingaben serverseitig validiert (zusätzlich zur clientseitigen Validierung)?
- *Struktur und Wartbarkeit*
  - Ist der Kontrollfluss verständlich? (Nachfragen deuten auf Probleme hin.)
  - Ist der Datenfluss nachvollziehbar — insbesondere bei globalem Zustand oder über Modulgrenzen hinweg?
  - Wird die Architektur eingehalten?
  - Ist der Code modularisiert und werden Funktionen klein und fokussiert gehalten (*Single Responsibility*)?
  - Wird auf tief verschachtelte Logik verzichtet?
  - Werden teure DOM-Manipulationen auf das notwendige Minimum beschränkt?
  - Werden Event-Listener korrekt registriert und wieder entfernt (Memory Leaks)?
- *Modernes JavaScript*
  - `const` und `let` statt `var`
  - Destrukturierung, Spread- und Rest-Operator wo sinnvoll
  - Klassen und ECMAScript Module statt globaler Funktionen und Skripte
  - Asynchrone Funktionen (`async`/`await`) statt verschachtelter Callbacks
  - Werden Browser-Kompatibilitätsanforderungen erfüllt?
- *Abhängigkeiten, Tests und Dokumentation*
  - Ist die Verwendung von Bibliotheken/Dependencies gerechtfertigt und aktuell?
  - Gibt es Testfälle und sind diese ausreichend?
    (Codeabdeckung ist *nur* ein erster Indikator.)
  - Ist die Inline-Dokumentation ausreichend, sinnvoll und korrekt?
  - Sind TODOs und FIXMEs verständlich und umsetzbar?
  - Bei verteilten Anwendungen: ist die Aufteilung der Logik nachvollziehbar und sinnvoll (zum Beispiel auch aus Sicht von Cheating-Möglichkeiten)?

Code Reviews von Java (Backend)

Exemplarische Kriterien

- *Korrektheit und Fehlerbehandlung*
  - Werden Eingaben serverseitig validiert (zusätzlich zur clientseitigen Validierung)?
  - Sind Bezeichner korrekt und sinnvoll gewählt?
  - Werden Sonderfälle und Fehlerzustände behandelt?
  - Ist die Fehlerbehandlung konsistent — werden Exceptions nicht verschluckt und ist das Logging aussagekräftig?
    (Werden checked und unchecked Exceptions bewusst und nachvollziehbar eingesetzt?)
- *Struktur und Wartbarkeit*
  - Ist der Code modularisiert und werden Klassen und Methoden klein und fokussiert gehalten (Single Responsibility)?
  - Wird die Architektur eingehalten?
  - Ist der Datenfluss nachvollziehbar — insbesondere über Schichtgrenzen hinweg?
  - Wird auf tief verschachtelte Logik verzichtet?
  - Werden Ressourcen korrekt geschlossen (try-with-resources)?
- *Modernes Java*
  - Werden aktuelle Sprachfeatures genutzt (*Records*, *Sealed Classes*, *Pattern Matching*, *Switch Expressions*)?
  - Werden Streams und Optionals sinnvoll eingesetzt?
  - Werden `var`-Deklarationen dort verwendet, wo sie die Lesbarkeit verbessern?
- *Abhängigkeiten, Tests und Dokumentation*
  - Ist die Verwendung der eingesetzten Bibliotheken gerechtfertigt?
  - Sind alle Abhängigkeiten aktuell?
  - Gibt es Testfälle und sind diese ausreichend?
    (Codeabdeckung ist *nur* ein erster Indikator.)
  - Ist die Javadoc-Dokumentation an öffentlichen Schnittstellen ausreichend?
  - Sind TODOs und FIXMEs verständlich und umsetzbar?

▤ Empfohlenes Vorgehen

1. Einführung durch die Autoren erhalten (Projekt vorführen lassen, Architektur und zentrale Entscheidungen verstehen.)
2. Build-Prozess und Projektstruktur prüfen (Lässt sich das Projekt bauen und starten? Ist die Struktur nachvollziehbar?)
3. Gesamtstruktur des Codes überblicken (Datenfluss, Modulgrenzen, Architekturkonformität.)
4. Technologiespezifisch in die Tiefe gehen (HTML → CSS → JavaScript — oder aufgeteilt in Subteams.)
5. Issues dokumentieren (Stelle → Problem → Verbesserungsvorschlag.)
6. Notizen für das eigene Projekt machen.