Steigerung der Codequalität

Unter Codequalität versteht man die Beurteilung von Code hinsichtlich bestimmter Kriterien wie zum Beispiel Lesbarkeit, Verständlichkeit oder Wartbarkeit. Die Erfüllung dieser Kriterien verringert das Einsatzrisiko der Software, minimiert die Einschulungszeit neuer Projektmitarbeiter und hält die Kosten für Wartung und Weiterentwicklung so gering wie möglich.

Auf den ersten Blick bedeutet das Anpeilen von hoher Codequalität erst einmal eine Verbesserung der Arbeitsprozesse in der Softwareentwicklung. Bei näherer Betrachtung wird klar, welch hohen Einfluss die Codequalität auf die Wirtschaftlichkeit von Softwareprojekten hat. Es empfiehlt sich daher, die von einem gegebenen Programmiererteam erzielte Qualität von Zeit zu Zeit durch einen Code Review zu überprüfen. Mit diesem Dokument geben wir einen Überblick der wichtigsten Kriterien von Codequalität und zeigen auf, wie eine Steigerung der Qualität erreicht werden kann.

1. Was sind die Kriterien von Codequalität?

Codequalität ist heutzutage in der EDV-Branche ein weit verbreitetes Schlagwort. Um den Begriff greifbarer zu machen, geben wir hier einen Überblick der Eigenschaften, die hochqualitativer Code unter anderem aufweisen muss:

- Lesbarkeit
- Verständlichkeit
- Wartbarkeit
- Fehlerfreiheit
- Inline-Dokumentation
- Portierbarkeit

Wenn diese Eigenschaften vorhanden sind, kann man von hoher Codequalität sprechen. Welche Bedingungen im Detail zutreffen müssen, damit die einzelnen Eigenschaften tatsächlich gegeben sind, behandeln wir auf den folgenden Seiten.

Ergänzend weisen wir bei jedem Abschnitt darauf hin, was wir selbst bei einem Code Review in der Praxis überprüfen. Das gibt Ihnen konkrete Hinweise, wie an der Steigerung der Codequalität gearbeitet werden kann.

2. Lesbarkeit

Sourcecode sollte unter allen Umständen leserlich sein. Das ist nur möglich, wenn in allen Modulen eindeutig nachvollziehbare Richtlinien eingehalten werden. Im Besonderen gilt das für die Verwendung (oder Auslassung) von Zeilenumbrüchen und Whitespaces.

Überprüfung bei einem Code Review:
Was die Lesbarkeit betrifft, beurteilen wir aus der Position eines Außenstehenden, was wir gut lesen können. Wenn etwas besonders unleserlich erscheint, kommentieren wir die Stellen mit entsprechenden Begründungen.

3. Verständlichkeit

Jeder Sourcecode sollte soweit als möglich selbst-dokumentierend sein. Unter "selbst"-dokumentierend verstehen wir folgendes: Die Namen der verwendeten Funktionen, Variablen und Klassen (bzw. Fields, Methods und Klassen, um im Java Jargon zu sprechen) müssen als erste Dokumentation ausreichend sein. Das heißt, auch ohne weitergehende Inline-Dokumentation muss der Leser eine grobe Idee von den Funktionsprinzipien des Codes bekommen sowie alle auftretenden Konzepte den modellierten Realweltkonzepten zuordnen können. Die Namen sollten dabei immer die Semantik wiedergeben und im Fall von Variablen beschreiben, was die Variable beinhaltet (und nicht etwa welchen Typ sie hat). Ebenso sollte ein Klassenname immer wiedergeben, welches Realweltkonzept die Klasse modelliert.

Zum Thema Verständlichkeit gehört ferner die Verwendung von gängigen Konstrukten (oder Idiomen) um gängige Probleme zu lösen. So hat zum Beispiel ein Schleifenkonstrukt, das eine bestimmte Anzahl von Malen durchlaufen wird, eine typische Form, die in der Industrie üblich ist und gut verstanden wird, obwohl alternative Formen denkbar wären. Naturgemäß ist die Definition von "üblich" eine weiche - dennoch kann gerade im Bereich von Codekonstrukten viel an Lesbarkeit erreicht oder vergeben werden. So ist etwa zu beachten, ob eine "for-loop" durch eine "while-loop" implementiert wurde oder umgekehrt.

Überprüfung bei einem Code Review:
In Hinblick auf Verständlichkeit überprüfen wir Code absichtlich mit einer gewissen Unkenntnis. Unter "Unkenntnis" verstehen wir hier die Unkenntnis der Funktionsprinzipien des jeweiligen Codes – dies hilft uns, die schwer verständlichen Teile zu identifizieren.
Als wichtigen Aspekt eines Reviews überprüfen wir auch, ob für das jeweilige Problem hinreichend von Abstraktionsschichten Gebrauch gemacht wurde und ob eine saubere Trennung zwischen den Abstraktionsschichten vorliegt. Wenn Konstrukte einer tiefer liegenden Abstraktionsschicht in eine höhere Schicht hineinragen, so vermerken wir das.

4. Wartbarkeit

Code sollte generell immer in Hinblick auf spätere leichte Änderbarkeit strukturiert werden. Dieses Grundprinzip hat eine Fülle von abgeleiteten Richtlinien zur Folge, wie zum Beispiel:

- Verwendung von Konstanten statt Literalen
- Ausfaktorisierung von mehrfach vorkommender Codelogik in Funktionen oder Methoden
- Zerlegung von umfangreichen Funktionen/Methoden in überschaubare Teilblöcke
- Verwendung von Interfaces (oder rein virtuellen Basisklassen in C++) zur sauberen Trennung von Codemodulen
- Verwendung von Abstraktionsschichten bzw. "Abstraktionslevels"

Überprüfung bei einem Code Review:
Im Rahmen eines Code Reviews überprüfen wir den gegebenen Sourcecode auf Kriterien wie die oben genannten.

5. Fehlerfreiheit

Dieser Punkt versteht sich eigentlich von selbst. An dieser Stelle weisen wir darauf hin, welche Fehler sich besonders gerne einschleichen:

- *Domainfehler:* Variablen haben einen falschen Typ (zu groß oder zu klein) oder der Code ist nicht entsprechend an den Typ angepasst. Ein typisches Beispiel ist etwa die Verwendung einer "unsigned short" Variablen, in die dann Werte > 0xFFFF "gesteckt" werden. Oder noch subtiler: die Verwendung einer „signed integer“ Variablen in einer Schleife, die beim Überschreiten ihres zulässigen Höchstwertes zu einer negativen Zahl wird, mit der die Schleife dann eventuell falsch umgeht (z.B. Indizierung in ein Array mit dem nunmehr negativen Index).
  Weitere prominente Kandidaten für Domainfehler sind Character Arrays (C-Style Strings) und deren Größe und die daraus oft resultierenden Buffer Overflow Errors. Diese machen in der Presse immer wieder Schlagzeilen im Zusammenhang mit Security Leaks. Besonderer Wert sollte auf das korrekte Abfangen solcher Overflows gelegt werden (z.B. durch Verwendung von _snprintf anstatt sprintf), selbst wenn das im Widerspruch zur Portierbarkeit stehen mag.

- *Multithreading-Fehler:* Fehler in Multithreading Code führen zu Deadlocks, Race Conditions und anderen Laufzeitfehlerzuständen. Sie sind oft durch lesen und "im-Geiste-Durchspielen" des Sourcecodes besser (oder früher) erkennbar als durch Testen, da sie unter Umständen nur sporadisch auftreten. Hier einige praktische Beispiele für solche Fehler:
  - Volatile bei Variablen vergessen, die zwischen Threads "geshared" werden
  - unzureichende Synchronisierung, die zur Zerstörung von Variableninhalten führt
  - Race Conditions (z.B. Thread A ist zu schnell und kann deswegen mit Thread B an der dafür vorgesehenen Stelle nicht kommunizieren - Stichwort "notify()" in Java)
  - Falsche Annahmen über die Atomizität von Schreiboperationen - diese Annahmen können oft auf Singleprozessor-Computern zutreffen, auf Multiprozessor-Computern hingegen nicht.

  Im selben Zuge sollte dann auch gleich überprüft werden, ob Annahmen über das Datenalignment gemacht werden und ob diese Annahmen haltbar sind (soweit die Zielhardware bekannt ist).

- *Resource Leaks:* Unter Resource Leaks versteht man, dass der Code während seiner Abarbeitung immer mehr Resources des Computers (wie etwa Hauptspeicher, Festplattenspeicher oder Backend-Prozesse von Datenbanken) belegt und so anderen Programmen entzieht. Resources müssen also sobald als möglich wieder freigegeben werden - dies muss vor allem im Zusammenhang mit Exceptions überprüft werden - ebenfalls ein Gebiet, das sich schwer testen lässt, da das planmäßige Auslösen von Exceptions sehr aufwändig ist. Hier kann eine Codeanalyse sehr viel Vorarbeit leisten.

- *Fehler im Exception Handling:* Unter Exception Safety versteht man das ordnungsgemäße Behandeln von Exceptions in dem Sinn, dass das Programm dann:
  - weiterläuft ohne abzustürzen oder unerwartet zu terminieren
  - keine Resource Leaks produziert
  - gegebenenfalls Protokolleinträge generiert, die es später ermöglichen, die Ausnahmesituation von einem Menschen untersuchen zu lassen

  Da es sehr aufwendig ist, Exceptions im Testbetrieb zu generieren (eine 100%-ige Testabdeckung findet hier praktisch nie statt), ist eine Codeanalyse der einzige Weg, alle Exception Handling Pfade zumindest einmal im Geiste abzudecken.

Überprüfung bei einem Code Review:

Bei einem Code Review achten wir natürlich auf alle Fehler, die sich eingeschlichen haben können. Domainfehler, Multithreading-Fehler, Fehler im Exception Handling und Ressource Leaks lassen sich dabei gut abfangen. Vor allem die letztgenannten würden in einer klassischen Testphase unter Umständen nicht oder nur selten erkannt (abhängig von den Testdaten und der Testhardware).

6. Inline-Dokumentation

Für die Inline-Dokumentation gelten u.a. folgende einfache Richtlinien:

- Sie sollte vorhanden sein
- Sie sollte dort vorhanden sein, wo es sinnvoll ist (weil Erklärungsbedarf besteht)
- Sie sollte sinnvolle Aussagen machen, die nicht direkt aus den Codestrukturen hervorgehen und die Semantik des Codes (und nicht die Syntax) betreffen. Dazu ein Beispiel: Statt "// Loop over index" sollte etwas in der Art wie: "// Check account balance for all found accounts" stehen.
- Sie sollte nicht zu umfangreich sein
- Auf keinen Fall sollte vor jeder Funktion oder Methode ein Kommentarblock stehen, der alle Parameter und Rückgabewerte erwähnt, aber keine Aussage zu deren Bedeutung macht. Das kommt besonders dann vor, wenn automatisierte Tools oder Editoren zur Erstellung von Kommentaren verwendet werden - hiervon raten wir ganz besonders ab. Der Effekt solcher Ansätze ist, dass irgendein (mehr oder minder sinnloser) Kommentar generiert wird und dann nicht mehr erkennbar ist, ob die Methode/Funktion/Klasse bereits dokumentiert wurde oder nicht - mit dem Resultat, dass eine wirklich sinnvolle Dokumentation nicht mehr erfolgt.

Überprüfung bei einem Code Review:
Im Rahmen eines Code Reviews überprüfen wir die Inline-Dokumentation auf die oben genannten Kriterien.

7. Portierbarkeit

Die wichtigsten Aspekte der Portierbarkeit sind jene von Compiler zu Compiler und von Betriebssystem zu Betriebssystem:

- *Compiler-Portierbarkeit:* Die Portierbarkeit auf andere Compiler ist vor allem für C bzw. C++ Code ein Thema. Für Java ist sie kaum von Bedeutung, da hier vor allem nur ein Compiler verwendet wird. Was die Compiler-Portierbarkeit betrifft, ist auf die Verwendung von standardisierten Konstrukten zu achten (sofern dies nicht mit dem Anspruch auf Fehlerfreiheit kollidiert).
- *Betriebssystem-Portierbarkeit:* Die Portierbarkeit von Betriebssystem zu Betriebssystem ist zwar keine unbedingte Voraussetzung für guten Code, aber dennoch zu empfehlen. Man bedenke, dass eines Tages das Betriebssystem, auf dem der Code läuft, durch eine Nachfolgeversion ersetzt werden wird - dann möchte man natürlich einen möglichst glatten und reibungsfreien Übergang haben. Als Hauptmittel, um die Betriebssystem-Portierbarkeit sicherzustellen, sei hier die Abstraktion genannt. Durch Einführung von Abstraktionsschichten zwischen der Applikation und dem Betriebssystem wird es leicht möglich, Änderungen im Betriebssystem abzufangen. Das gleiche gilt für die Verwendung von Standardbibliotheken der Compiler-Hersteller.

Überprüfung bei einem Code Review:
Im Rahmen eines Code Reviews überprüfen wir die Portierbarkeit im Rahmen der oben genannten Kriterien.

8. Sonstige Optimierungen

Bei Code Reviews haben wir natürlich die Gelegenheit, auch Optimierungsaspekte zu beleuchten, die nicht unbedingt unter den Begriff der Codequalität fallen. Wo immer uns Optimierungspotential beim Durchsehen eines Sourcecodes ins Auge sticht, vermerken wir das. Dabei kann es um Themen wie Schleifeninvarianten ebenso gehen wie um Pass-By-Value Konstrukte mit großen Klassen/Strukturen oder nicht-lokalen Datenzugriff (der zu Cache Misses führt). Mit anderen Worten, wir bemühen uns bei Code Reviews, in unseren Beurteilungen soweit als möglich state-of-the-art Informationen einfließen zu lassen. Wenn durch neue Forschungen im EDV-Sektor fortschrittlichere Techniken bekannt sind, wie man Standardprobleme bewältigen kann, weisen wir im Sinn einer umfassenden Codequalität ebenfalls darauf hin.

Ergänzend zu diesem Text steht Ihnen auf der InfraSoft Website ein Glossar zur Verfügung, in dem Sie die meisten der hier verwendeten Begriffe finden. Über das aktuelle Angebot an weiteren, kostenlosen Fachbeiträgen zur Softwareentwicklung informieren Sie sich bitte unter www.infrasoft.at/service.

Harald Nowak
Wien, im März 2004

Der Autor ist Mitarbeiter der InfraSoft, einem Unternehmen, das auf komplexe Softwareentwicklungen spezialisiert ist. Die Experten der InfraSoft haben langjährige Erfahrungen in der Entwicklung und verfügen über fundierte Kenntnisse in Design, Analyse, Realisierung, Test und Projektmanagement.
Für **individuelle Beratungen** zur Entwicklung von Softwarelösungen und die Bereitstellung von **Realisierungsteams** wenden Sie sich bitte an info@infrasoft.at.