Guter Code

Vorwort

Es gibt guten Code und es gibt schlechten Code, und leider gibt es mehr schlechten als guten Code. Zum Programmieren gehört nun einmal mehr als das reine Beherrschen der Syntax. Man muss die Eigenheiten der Sprache verstehen, und man muss *wissen*, was genau man tut.

Zum Wissen gehört auch das Lernen. Als Programmierer in einem Team muss man zwangsläufig den Code anderer Leute lesen und diesen verstehen, und als Einzelkämpfer muss man das genauso – schließlich sind Standardbibliotheken auch von anderen Menschen entworfen und geschrieben worden.

Wenn man allerdings die Kenntnis dessen, was in den Standardbibliotheken möglich ist, als Wissen bezeichnet, so ist das ein Trugschluss. Zum Wissen gehört auch, dass man die Implementierung versteht, und welche Effekte verschiedene Vorgehensweisen mit sich bringen.

In diesem Dokument versuche ich, sprachunabhängige Tipps zu sammeln. Gelegentlich wird mir das vielleicht nicht ganz gelingen, aber ich versuche, mein Bestes zu geben.

1 Lerne zu sprechen

Eine Sprache lernt man nur, indem man sie spricht. Die Grammatik, Syntax und Semantik einer Sprache zu beherrschen ist nützlich, aber nutzlos, wenn es darum geht, Inhalte auszudrücken. Der eigentliche Stein des Anstoßes, wenn jemand ein Programmierproblem hat, ist der, dass er zwar die Syntax beherrscht, jedoch seine Absichten nicht in die Sprache seiner Wahl übersetzen kann.

Als einfaches Beispiel sei hier einmal der Versuch erwähnt, »auf Nummer Sicher gehen« ins Englische zu übersetzen. Beherrscht man die Grammatik des Englischen und hat ein Wörterbuch zur Hand, so wird man mit Sicherheit auf etwas Ähnliches wie »to go on a safe number« kommen – was aber völlig sinnfrei ist, da die Bedeutung des deutschen Sprichworts völlig losgelöst von der wörtlichen Bedeutung ist. Die korrekte Übersetzung wäre »better safe than sorry«, was wiederum wörtlich ins Deutsche übersetzt etwa »besser sicher als Verzeihung« heißt. Noch interessanter wird dieses Thema im Französischen, insbesondere bei politischen Themen; dort werden in Zeitungsartikeln Parteien mithilfe der Straßen-, Orts-, Marktplatz- und Forumsnamen identifiziert, und ohne hinreichende

Kenntnisse der französischen Sprache käme man vermutlich noch nicht einmal auf die Idee, dass man gerade einen Artikel mit politischem Inhalt liest.

Wenn Du kein richtiges Gefühl für Deine Programmiersprache hast, wird es Dir schwerfallen, Deine Ideen in Programme umzusetzen. Du musst Deine Programmiersprache sprechen, und ab einem gewissen Punkt solltest Du Dich auch von einem sehr heiklen Thema loslösen, nämlich dem gleichgültigen Mitschwimmen in der Meute. Es gibt zwar die verschiedensten verbreiteten Paradigmen und Vorgehensweisen, dennoch sind bei Weitem nicht alle davon richtig. Als Beispiel sei der Straßenverkehr genannt: ich schätze, dass 80% aller Autofahrer innerorts zu schnell fahren, doch das macht es auch nicht richtiger, oder?

2 Erfinde das Rad nicht neu

Wie im Vorwort erwähnt, muss man den Code anderer Programmierer lesen. Dies ist aufgrund der Wartbarkeit von Software zwingend notwendig, und in großen Softwareprojekten mit vielen Modulen unumgänglich. Wenn da jeder sein eigenes Ding drehen würde, gäbe es keine gemeinsame Schnittstelle, die das Zusammenspiel der Module ermöglichen würde.

Aber nehmen wir an, die Schnittstelle steht und funktioniert, und Du bist auf Dich allein gestellt. Du fängst an, Deine eigenen Container zu schreiben, weil die Standardbibliothek zu langsam ist. Du gehst weiter und schreibst Deine eigenen Algorithmen, weil die Standardbibliothek zu langsam ist. Du hast Deine eigene kleine Standardbibliothek geschrieben, die erstens wahrscheinlich inkompatibel zur »echten« Standardbibliothek ist, und zweitens wahrscheinlich (wenn überhaupt) nur unwesentlich schneller.

Du hast das Rad neu erfunden. Glückwunsch! Was aber hieraus folgt, sind zwei Punkte, die mehr oder minder offensichtlich bereits erwähnt wurden.

2.1 Triff keine Annahmen

Du hast offensichtlich der Standardbibliothek nicht getraut. Du bist offensichtlich davon ausgegangen, dass die Standardbibliothek von dummen Theoretikern entworfen wurde, die keine Ahnung von »echten Programmen« haben. Bist Du Dir sicher? Ich habe von noch keiner Standardbibliothek gehört, die nicht von Gott weiß wie vielen guten Programmierern entworfen, begutachtet, korrigiert, weiterentwickelt, implementiert, verbessert, überprüft, ... wurde.

2.2 Kenne Dein Spielfeld

Bevor Du etwas neu schreibst, suche nach bereits vorhandenen Implementierungen, und evaluiere die Verwendbarkeit in Deinem Projekt. Es ist höchst unwahrscheinlich, dass noch kein anderer Programmierer auf ein zumindest ähnliches Problem gestoßen ist. Selbst in absoluten Randbereichen der Programmierung wird sich wahrscheinlich irgendwo eine Lösung finden lassen, die ein ähnliches Problem behandelt. Handelt es sich nicht

um einen Randbereich, wird sich die Lösung höchstwahrscheinlich irgendwo in der Standardbibliothek verbergen.

3 Triff keine Annahmen

Ist das nicht dasselbe wie oben? Nein, ist es nicht. Oben ging es um das Vertrauen in die Fähigkeiten der Standardbibliothek, hier geht es um das Vertrauen in Eingaben.

Aber mach Dir bitte klar, dass mit Eingaben nicht nur die paar Tasten auf der Tastatur oder das Drehen des Mausrades gemeint sind. Eingaben können auch in Form von Netzwerkstreams, Dateien, Audio, Video, etc. vorliegen. Du darfst *niemals* davon ausgehen, dass diese Daten in einem korrekten Format vorliegen – Programme, die das tun, sind für Pufferüberlaufangriffe vorbestimmt und können in kritischen Systemen sogar Menschenleben gefährden. Im besten Fall stürzt Dein Programm einfach ab. Wohlgemerkt, im besten Fall – im normalen Fall wird Dein Programm fröhlich weiterarbeiten und dabei Daten vernichten.

4 Behandle Fehler richtig

Kommen wir nun zu einem wichtigen Punkt: Fehlerbehandlung. Dabei ist das Problem eher weniger die Behandlung an sich, als die Behandlung des Codes zwischen Auftreten und Behandeln des Fehlers. In objektorientierten Sprachen ist dies weniger ein Problem, insbesondere wenn die Sprache einen Garbage Collector hat. Dennoch kann es hier zu Problemen führen, wenn bspw. eine Systemressource belegt wurde, dann eine Exception auftritt, und dann das Freigeben der Ressource übersprungen wird. Die üblichen Symptome solcher Fehler sind Ressourcenmangel, Deadlocks und Systemabstürze.

In der Regel wird Fehlerbehandlung unter der Annahme geschrieben, dass die Daten konsistent bleiben – vielfach ist diese Annahme aber *lokal beschränkt*. In nicht lokal gehaltenen Modulen wird sich ein Fehler außerhalb des aktuellen Moduls fortpflanzen.

5 Halte alles so lokal wie möglich

Der Sinn objektorientierter Programmierung ist es, Daten zu kapseln und die Behandlung und Interpretation der Daten hinter einer abstrakten Schnittstelle zu verbergen. Ein globales Objekt ist da fehl am Platz, denn ein globales Objekt ist von überall aus global zugreifbar und damit frei von Datenkapselung und man wird auf einfachste Weise dumme Sachen damit anstellen können – es fehlt der Kontext.

Ein globales Objekt entspricht übrigens ziemlich genau dem *Singleton Design Pattern*. Was spricht also dagegen, das globale Objekt designkonform statisch-lokal an eine Klasse zu binden? Insbesondere wird es dann leichter, die Singleton-Funktion durch eine Quasi-Factory-Funktion zu ersetzen, wenn sich bspw. herausstellt, dass ein Programm auf mehrere Threads aufgeteilt werden soll und nun jeder einzelne Thread seine eigene Objektinstanz benötigt. Wäre das Objekt als globale Variable deklariert, würde das Refactoring einen Rattenschwanz von Änderungen nach sich ziehen.

Diese Ansicht lässt sich weiter lokalisieren: In Funktionen sollten Variablen in den tiefsten Gültigkeitsbereich gezogen werden, wo sie gebraucht werden, und dann soweit nach unten, bis sie gebraucht werden. Stell Dir vor, du definierst eine Instanz einer gigantischen Klasse mit einem extrem ressourcenfressenden Konstruktor am Anfang einer Funktion, nur um danach in einem »if« aus der Funktion herauszuspringen, ohne die Variable verwendet zu haben, obwohl sie weiter unten gebraucht wird. Selbst der intelligenteste Compiler wird den Konstruktoraufruf nicht wegoptimieren, da die objektorientierte Programmierung voraussetzt, dass der Aufrufer einer Funktion nichts über die Implementierung der Funktion zu wissen braucht – der Compiler muss also davon ausgehen, dass der Konstruktor der instanziierten Klasse irgendwelche Seiteneffekte außerhalb der Klassenlogik an sich auslöst.

6 Optimiere später

Was denkst Du ist einfacher: lesbaren Code zu optimieren oder optimierten Code lesbarer zu machen?

Zweifellos ist Optimierung nützlich, aber man sollte es nicht übertreiben. Eine Regel besagt, dass 20% des Codes 80% der Ressourcen beanspruchen. Das Hauptproblem bei der Optimierung ist es also, diese 20% zu finden. Und diese 20% kann man erst finden, wenn das Programm nahezu fertig ist, denn erst dann kann man sich sicher sein, dass es nicht mehr unter »Laborbedingungen« läuft, sondern unter Praxisbedingungen – und diese beiden Szenarien können sich mitunter erheblich voneinander unterscheiden.

Des Weiteren verschlechtert Optimierung die Wartbarkeit, denn lesbarer Code lässt sich leichter verändern als optimierter Code. Optimierter Code ist per Definition wesentlich enger an den speziellen Kontext gebunden, in dem er läuft, und muss dementsprechend stärker als nicht-optimierter Code geändert werden, wenn sich etwas am Kontext ändert. Und diese Änderungen können unter Umständen sogar dazu führen, dass die bereits optimierte Version nach den Änderungen ineffizienter läuft als die vormals lesbare Funktion.

7 Halte es allgemein

Wiederverwendbarer Code erhöht die Wartbarkeit eines Programms, erleichtert durch eine gemeinsame Schnittstelle das Programmieren, und verringert den Aufwand beim Programmieren. Ich möchte an dieser Stelle nicht genauer darauf eingehen, denn sämtliche Punkte sollten jetzt eigentlich klar sein – denke an die Standardbibliothek Deiner Programmiersprache, welchen allgemeinen Code bietet.

Ich möchte lieber einen weniger offensichtlichen Punkt ansprechen, nämlich das Vermeiden von zu langen Funktionen. Wird der Code einer Funktion so lang, dass er nicht mehr komplett auf den Bildschirm passt, lohnt sich eine Analyse der Logik. In nahezu allen Fällen wirst Du den einen oder anderen Logikpunkt finden, der sich in eine eigene Methode oder sogar eine eigene Klasse auslagern lässt und somit zu allgemeinem Code wird, der sich wiederverwenden lässt.

Findest Du keine solchen Punkte, solltest Du dennoch versuchen, die Funktion aufzusplitten, denn die Arbeitsweise der Funktion lässt sich sonst schwerer nachvollziehen. Wenn Du darauf angewiesen bist, Codestücken kurze, prägnante und aussagekräftige Funktionsnamen zu geben, wirst Du vielleicht Deinen eigenen Code besser verstehen und sogar Optimierungspotenzial finden.

8 Nenne das Kind beim Namen

Nun noch ein kleiner, dennoch wichtiger Punkt: Benenne Funktionen, Typen und Variablen nach ihrer Funktion oder Bedeutung, nicht nach ihrem Typ. Es ist wirklich schlechter Stil, eine Variable mit einer Zahl darin einfach »zahl« zu nennen anstatt sie ihrer Bedeutung nach bspw. »mehrwertsteuersatz« zu nennen. In ein- oder zweizeiligen Funktionen kann man eine Variable mal »tmp« nennen – wenn die Bedeutung der Variable direkt ersichtlich ist. Im Übrigen ist »temporär« auch eine Art Typ, denn ein Variablenname »temp« sagt nichts über die darin enthaltenen Daten aus.

Genauso ungünstig wie die Benennung nach Typ sind undurchsichtige Abkürzungen, denn sie verhindern, dass man Code schnell und flüssig durchlesen kann wenn man bei jeder Abkürzung erst einmal über ihre möglichen Bedeutungen nachgrübeln muss.