Wartungsfreundliche Softwareentwicklung

Andreas Möller, EDVDAM, Februar 2019

Jeder Softwareentwickler kennt es. Schon nach ein paar Wochen fällt es selbst bei eigenen Programmierarbeiten schwer, noch nachzuvollziehen, wie alles funktioniert. Es ist oft schwierig und aufwändig sich noch einmal hineinzuversetzen. Notwendige Änderungen oder die Suche nach Fehlern erweist sich schwieriger als erwartet. Die Schwierigkeit steigt, wenn es sich um fremden Code handelt, der mit einem anderen Wissensstand erstellt wurde oder einfach einen anderen Stil hat. Hilfreich ist es, wenn schon bei der Erstellung der Software die Wartbarkeit berücksichtigt wird. Ein paar systematische Herangehensweisen und Konventionen sind dabei hilfreich.

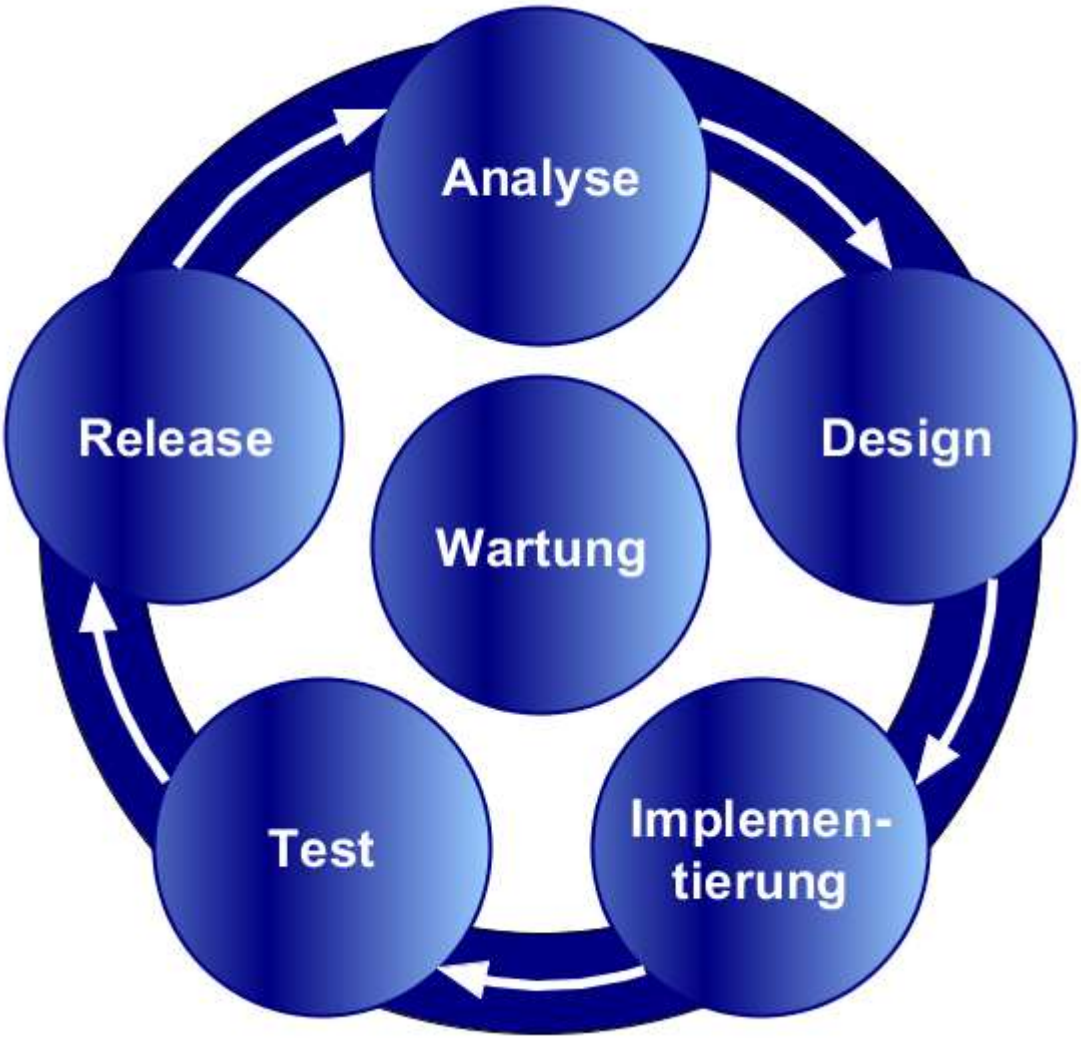

Bild 1 Wartung als zentraler Bestandteil des Entwicklungsprozesses. EDVDAM mit yEd

Inhalt

Gut konstruiert ... 3
- Architektur ... 3
- Datenmodell ... 3
- Coding Standards ... 3
- Format Standards ... 3

Testbar ... 3
- Anzahl der Parameter ... 4
- Komplexe Datentypen ... 4
- Globale Variablen ... 4
- Unittests ... 4

Debugbar ... 4
- Einzeln aufrufbar ... 4
- Globale Objekte ... 4
- Komplexe Datentypen ... 4

Verständlich ... 4
- Einheitliche Formatierung ... 4
- Formatierungswerkzeuge ... 5
- Inline Kommentare ... 5
- Header über Codeblöcken ... 5
- Stukturiert ... 5

Übersichtlich ... 5
- Namenskonventionen ... 5
- Größe der Objekte ... 5
- Verzweigungstiefe ... 6
- Tools ... 6

Versionierbar ... 6
- Repository ... 6
- Branchkonzept ... 6

Kontrolliert ... 6
- Getestet ... 6
- Reviewt ... 7

Dokumentiert ... 7
- Verfügbar ... 7
- Verknüpft ... 7
- Versioniert ... 7
- Pflegbar ... 7
- Sinnhaftigkeit ... 7

Grenzen der Gültigkeit ... 7

Gut konstruiert

Gut konstruierte Software erfüllt in der Regel nicht nur alle funktionellen Anforderungen, sondern sie hat auch eine logische, an die Anforderungen angepasste Architektur.

Architektur

Ein paar grundsätzliche Überlegungen zu Modularisierung und Aufbau der Software sollten anhand von Diagrammen diskutiert und abgestimmt worden sein, so dass alle wissen, an was sie arbeiten. Auf diesen grundsätzlichen Aufbau sollten sich im Folgenden alle Objekte beziehen. Wenn sie sich nicht eindeutig zuordnen lassen, dann droht die Übersicht verloren zu gehen. Entweder das Objekt, oder die Architektur, oder beides sollte dringend überarbeitet werden. Periodische Wartungsaufgaben, Verbesserungen und Fehlersuche können bei einer guten und dokumentierten Architektur viel zielgerichteter erfolgen.

Datenmodell

Ist die Applikation stark datenorientiert, ist das Datenmodell (unterstellt wird ein relationales Modell) essentiell für die Wartbarkeit. Formale Anforderungen sind:

- Technische Schlüssel
- Sinnvolle Normalisierung
- Aussagekräftige Bezeichnungen, Tabellen und Spaltenkommentare
- Konsequente Fremdschlüsselverwendung

Wichtig ist, dass sich in dem Datenmodell die Datensicht der Applikation möglichst einfach darstellen lässt. Gleichzeitig ist zu berücksichtigen, dass sich Daten, die von außen geladen werden, möglichst einfach integrieren lassen. Für die Konstruktion eines Datenmodells ist ein grafisches Tool deshalb sinnvoll, weil es mehr Beteiligten die Teilnahme an der Gestaltung ermöglicht und unübersichtliche Konstrukte leichter vermieden werden können. Optimal ist, wenn sich dieses grafische Modell immer synchron zur Datenbank mit verändert. So kann es jederzeit gezielt für die Wartung genutzt werden.

Coding Standards

Zu einer guten Konstruktion gehört auch die Verwendung von Coding Standards, die immer aktuell dokumentiert und zugänglich sind. In kleinen Teams einigt man sich oft formlos auf einen gemeinsamen Nenner. Das ist kurzfristig effizient, jedoch nicht, wenn ein Dritter später mit der Wartung beschäftigt ist. Hier sind formal festgehaltene Standards hilfreicher.

Format Standards

Die Frage, wie Quelltext formatiert werden soll wird von Team zu Team unterschiedlich beantwortet. Für das Zurechtfinden in dem Code ist es aber sinnvoll, dass die Formatierung möglichst gleichförmig erfolgt. Wenn man sich auf ein paar Gemeinsamkeiten verständigt hat, kann ein Code Beautifier diese sehr effizient umsetzen. Das enthebt die einzelnen Entwickler auch davon, schon beim Entwickeln die Formatierungsregeln genau zu befolgen. Vor der Code-Veröffentlichung wird der für alle einheitlich konfigurierte Beautifier einfach verwendet.

Testbar

Um kontrollieren zu können, ob die Software noch in vollem Umfang funktioniert und auch um effizient Fehler finden zu können, sollte Software gut testbar sein. Für die Wartbarkeit von Software ist die durchgängige Testbarkeit von Modulen und Einzelbausteinen von entscheidender Bedeutung. Sind diese Bausteine und Gruppen zu komplex oder schlecht abgrenzbar konstruiert, erhöht das den Aufwand für Wartung und Fehlersuche. Ein paar Regeln verringern das Risiko schlechter Testbarkeit.

Softwareobjekte werden oft mit Parametern aufgerufen und geben dann andere Parameter oder eine Reaktion zurück. Je komplizierter der Aufruf eines solchen Objektes ist, desto schwieriger ist es zu testen.

Anzahl der Parameter

Es macht Sinn, die Anzahl der Übergabeparameter gering zu halten. Außerdem sollten die Parameter mit überschaubarem Aufwand konstruierbar sein, damit sinnvolle Testfälle leicht erstellt werden können. Was erklärungsbedürftig ist, sollte in diesem Kontext im Inlinekommentar erklärt werden.

Komplexe Datentypen

Besonders kompliziert kann es werden, wenn komplexe Datentypen als Eingabe- oder Ausgabeparameter genutzt werden. Sollte es sich nicht vermeiden lassen, ist es hilfreich wenn die Erstellung des Datentyps in einem eigenen separaten Softwareobjekt erfolgt, das auch für den Test benutzt werden kann.

Globale Variablen

Die Verwendung von globalen Variablen kann das Testen ebenfalls komplizierter machen, wenn diese nicht als Parameter übergeben werden. Es ist daher überlegenswert, diese als Parameter in das Objekt einzuführen, auch wenn das im ersten Augenblick umständlich erscheint.

Unittests

Für einen kontinuierlichen und automatisierten Test der Funktionsfähigkeit der Software bieten sich auf der Quellcodeebene Unittests an. Diese überprüfen, ob das Softwareobjekt bei durchdacht definierten Eingaben immer das erwartete Ergebnis erzielt. Unittests sind ein wichtiger Indikator für die Stabilität der Software, wenn diese verändert wird, z.B. bei einem Refactoring oder einer Erweiterung. Sie liefern Fehler zurück, sobald etwas nicht so funktioniert, wie getestet.

Debugbar

Oft ist es unumgänglich, die Objekte zu debuggen, um zu verstehen, wie genau sie im Zusammenspiel funktionieren und um verborgene Fehler aufzuspüren. Daher ist es wichtig, die Softwareobjekte debugbar zu programmieren. Oft ist dieses Kriterium schon erfüllt, wenn das Objekt wie beschrieben testbar ist. Folgendes ist noch hinzuzufügen.

Einzeln aufrufbar

Beim Debuggen von Objekte ist es wichtig, dass diese als kompletter Workflow, aber auch einzeln aufrufbar debugbar sind. Das spart u.U. Zeit und ist effizienter.

Globale Objekte

Störend kann sich beim debuggen auswirken, wenn globale Objekte, die beim debuggen nicht einfach mit initialisiert werden können, für den Ablauf notwendig sind. Unschöne Effekte die daraus resultieren sind z.B. ein ungewollter Abbruch des Debuggens oder dass bestimmte Verzweigungen nicht mit debuggt werden können.

Komplexe Datentypen

Beim Debuggen ist es insbesondere von Interesse, den Zustand und die Werte von Variablen zur Laufzeit zu prüfen, etwas das über bloßes testen weit hinausgeht. Es ist aber ärgerlich, wenn das nicht möglich ist, weil das Objekt von einem zu komplexen Datentyp ist. Die Befüllung solcher Datentypen im Debugmodus muss einfach umsetzbar sein.

Verständlich

Durch das verwenden einfacher Regeln wird der gleiche Quelltext verständlicher und ist einfacher zu ändern.

Einheitliche Formatierung

Formatierungskonventionen sind sehr hilfreich dabei, Quelltext zu analysieren. Dabei gibt es im Rahmen gelebter Praxis immer noch eine Reihe von Spielarten, die alle ihr Für und Wider haben.

Der Punkt ist aber nicht, die einzig wahre Formatierung zu benutzen, sondern der Wert liegt darin, dass alle die gleiche verwenden. So fällt es jedem leichter, den Quelltext des anderen zu verstehen, weil Muster sich wiederholen und gleiches eben auf den ersten Blick gleich ist.

Formatierungswerkzeuge

Die Verwendung eines automatischen Beautifiers, der den Quelltext vor dem Deployment noch einmal auf die eingestellte Weise formatiert ein sehr effizienter Weg. Allerdings trägt auch eine einheitliche Struktur der Codeblöcke zur Lesbarkeit bei. Hier ist die Verwendung von Templates ein effizienter Schritt zur Vereinheitlichung.

Inline Kommentare

Auch im bestens formatierten Quelltext verbergen sich oft Tricks und komplizierte Lösungen, die sich oftmals nicht vermeiden lassen, die aber für einen anderen Entwickler schwierig zu verstehen sind. Quelltext, der nicht selbsterklärend oder Standard ist, sollte durch einen Inlinekommentar erläutert werden. Es ist aber wichtig, hier eine vernünftige Balance zu finden, da zu viele oder zu lange Kommentare den Quelltext wiederum unleserlich machen.
Kommentare werden auch oft benutzt, um Quelltext der bei einer Änderung entfernt werden muss, zunächst nur unausführbar zu machen. Der Sinn ist u.a., nicht zu vergessen, wie die ursprüngliche Lösung aussah, falls sich die aktuelle Entwicklung als Irrtum herausstellt. Diese Aufgabe, sich Lösungen zu merken, die nicht mehr aktuell sind, hat das Repository, das über eine Versionierung Zugriff auf jede frühere Version ermöglicht. Auskommentierter Quelltext sollte also spätestens vor dem Deployment komplett entfernt werden.

Header über Codeblöcken

Werden neue Codeblöcke erstellt (Klassen, Funktionen, Prozeduren), ist es üblich, einen Header voranzustellen, der einige Angaben über den folgenden Quelltext enthält. Hier gibt es über Art und Umfang des Headers unterschiedliche Ansätze. Am besten ist es, wieder einen Standard zu vereinbaren und diesen als Template zu hinterlegen. Es macht Sinn, hier zumindest den Autor, das Datum der Erstellung und den allgemeinen Zweck des Blocks zu hinterlegen. Es macht wenig Sinn Angaben, die aus dem Repository hervorgehen, wie eine Chronik hier noch einmal zu wiederholen.

Stukturiert

Quelltext sollte im größeren Zusammenhang inhaltlich und systematisch strukturiert erstellt werden. So werden z.B. bei PL/SQL Prozeduren und Funktionen in Packages zusammengefasst. Hier können auch komplexe Datentypen, Sessionvariablen und Konstanten hinterlegt werden. Es gilt dabei die Regel gleiches zu gleichem. Auch für solche Packages gilt es einen aussagekräftigen Header zu erstellen, an dem dann abzulesen ist, ob ein neuer Codeblock in den Kontext passen würde, oder nicht.

Übersichtlich

Ist der Quelltext im Detail verständlich, kann er einen Entwickler insgesamt trotzdem vor große Herausforderungen stellen. Gerade in großen Projekten ist es notwendig, das Augenmerk auch auf die Übersichtlichkeit des Gesamtproduktes zu lenken.

Namenskonventionen

Die Namensgebung von Objekten ist keine triviale Angelegenheit. Die Funktion des Objektes im Kontext des Projektes soll genauso ersichtlich werden, wie die Einordnung in bestimmte Objekttypen. Daher ist es sinnvoll, im Rahmen der Coding Standards auch Konventionen für die Benamung festzulegen. Darüber hinaus ist es sinnvoll, die Objektnamen hin und wieder im Team zu diskutieren, um ein gemeinsames Verständnis zu erzielen.

Größe der Objekte

Ein Dilemma. Zergliedert man konsequent nach Mikrofunktionalitäten wird die Anzahl der Objekte unüberschaubar. Lässt man alles beisammen entstehen Monsterobjekte. Was kann ein Entwickler noch gut überschauen? Wenn ein Codeobjekt in sich strukturiert werden muss, lässt es sich in der Regel auch sinnvoll zergliedern. Es mach Sinn, im Team oder im Rahmen eines Codereviews gemeinsam nach solchen Anzeichen zu suchen. Es ist aber auch gut, gemeinsam

Grenzen zu setzen, welche Größe noch im Rahmen ist. Der Entwickler hat so einen Anlass, sich über eine übersichtlichere Lösung Gedanken zu machen.

Verzweigungstiefe

Es ist wichtig, hier nicht zu übertreiben. Die Gesamtfunktionalität aus einer sehr verschachtelten Struktur zu erfassen ist schwierig. Eine klare übergreifende Hierarchie der Objekte ist von Vorteil. Gegenseitige Abhängigkeiten oder Schleifen können so vermieden werden.

Tools

Für die Entwicklung sind Werkzeuge mit folgenden Funktionalitäten sinnvoll:

- Versionierung
- Beautifying
- Templates
- Visualisierung
- Codevergleich
- Codemerge
- Reengineering
- Review

Die Verwendung der Tools muss ebenfalls dokumentiert werden und es sollte ein Konzept für die nachhaltige Bereitstellung dieser Tools geben, damit sie bei der Wartung auch noch zur Verfügung stehen.

Versionierbar

Die Frage, was wann und von wem an einem Codeobjekt geändert werden ist, kann im Rahmen der Entwicklung, aber auch bei der Wartung sehr wichtig sein.

Repository

Eine Versionierung sollte immer Zugriff auf alte Versionen der Codeobjekte bieten und die Unterschiede zwischen den Versionen darstellen können. Man kann nicht generell sagen, welche Repository Software für diese Aufgabe am besten geeignet ist. Die Bedienung sollte aber dokumentiert werden, damit es einheitlich bedient wird und damit sich jemand im Zuge einer Wartung schnell zurechtfindet.

Branchkonzept

Für die parallele ungestörte Arbeit am Quellcode kann man in vielen Repositories Branches bilden, die nach erfolgter Arbeit wieder mit dem Hauptzweig verbunden werden. Insbesondere diese Zusammenführung oder auch ein zwischenzeitlicher Abgleich der Veränderungen in verschiedenen Branches kann zu ungewollten Effekten bis hin zum Quellcodeverlust führen. Daher ist es besser, ein einheitliches Konzept zu verfolgen und auch zu dokumentieren.

Kontrolliert

Es ist das eine, sich etwas vorzunehmen, aber es ist etwas anderes, dies auch zu gewährleisten. In diesem Falle muss eine Qualitätskontrolle stattfinden. Dies kann obligatorisch oder als Stichprobe erfolgen.

Getestet

Softwaretests sind mittlerweile Standard und werden oft nach den Grundsätzen des ISTQB durchgeführt. Gut durchdacht ist hier angebrachter als gut gemeint und systematisch ist verlässlicher als intuitiv. Da die Ressourcen für Tests immer knapp sind, macht es Sinn, die Testüberdeckung dort zu priorisieren, wo man zuvor das größte Risiko erkannt hat. Die gleichen Funktionstests im Wartungsfall erneut durchzuführen ergibt ein hohes Maß an Sicherheit, dass nichts durch Seiteneffekte zerstört wurde, setzt aber voraus, das die Testergebnisse dokumentiert wurden und bereitgehalten werden. Unittests sollten ebenfalls erneut ausgeführt werden. Bei vielen

Umgebungen laufen sie täglich und müssen nicht extra reaktiviert werden. Im Falle von Programmergänzungen und -erweiterungen müssen Tests in der Regel ebenfalls ergänzt werden.

Reviewt

Das Review ist ein zentraler Akt der Qualitätskontrolle, aber auch des Findens gemeinsamer Sichtweisen auf Arbeitsweisen, die nicht spezifiziert sind. Eine gut reviewte Software erfüllt die Qualitätskriterien nicht nur formal, sondern auch dem Sinne nach. Reviews können durch spezielle Reviewtools vereinfacht werden, da diese viele Verstöße gegen Programmierkonventionen selbsttätig erkennen. Wenn der Entwickler solch ein Tool verwendet, um vor dem Review seinen Code anzupassen, kann man sich beim Review auf wesentlicheres konzentrieren.

Dokumentiert

Es ist unstrittig, dass für programmierte Software eine Dokumentation erstellt werden muss. Es macht Sinn, sich darüber zu verständigen, wie diese Dokumentation aussehen soll. Ich konzentriere mich hier auf die technische Dokumentation, die für die Wartung der Software entscheidend ist. Folgende Prinzipien und Elemente sollten bedacht werden.

Verfügbar

Eine Dokumentation sollte einfach und nachhaltig verfügbar sein. Sie sollte einfach aufzufinden sein und man sollte mit vorhandenen Mitteln auf die Dokumente zugreifen können. Auch noch nach mehreren Jahren und auch noch nach einem Systemwechsel. Daher sollten Dokumente in einem gängigen Standardformat erstellt werden und an einem sicheren Platz aufbewahrt werden. Es macht Sinn, die technische Dokumentation zusammen mit dem Quellcode aufzubewahren.

Verknüpft

Die Dokumentation besteht oft aus mehreren Dokumenten und zusätzlichen Rahmendokumenten, auf die in der Doku Bezug genommen wird. Es macht Sinn, die Bezüge dieser Dokumente untereinander aufzuzeigen. Als generelle Angabe in der Dokumentstruktur und konkret dort, wo ein Bezug besteht. Gegebenenfalls sogar aus dem Quellcode heraus, wenn sich Zusammenhänge anhand der Dokumentation dort besser verstehen lassen.

Versioniert

Wird der Quellcode in einem Repository mit automatischer Versionierung aufbewahrt, ist es wichtig, die Dokumentation ebenfalls zu versionieren. Der Bezug von unterschiedlichen Softwareversionen zu der jeweils passenden Dokumentationsversion muss sich einfach herstellen lassen. Entscheidend sind in den meisten Umgebungen aber die Dokumentationsstände, die zu den Quellcodeständen passen, die produktiv eingesetzt werden.

Pflegbar

Nach Möglichkeit sollte nichts zweimal dokumentiert oder erklärt werden. Es ist bekannt, dass Redundanzen zu Inkonsistenzen führen und eine inkonsistente Dokumentation verliert an Wert. Es ist auch besser, eine gut durchdachte, als eine umfangreiche Dokumentation zu machen. Eine zu umfangreiche Dokumentation ist schwierig zu pflegen. Beim Lesen der Dokumentation verliert man Zeit, die entscheidenden Informationen verstecken sich unter einer Flut von nutzlosen Infos und bei Änderungen ist zu viel Textarbeit zu leisten.

Sinnhaftigkeit

Bei jeder Information, die in die Dokumentation einfließt sollte klar ersichtlich sein, wozu diese Information später einmal benötigt wird. Ansonsten ist sie unnützer Ballast.

Grenzen der Gültigkeit

Diese Zusammenfassung aus dem, was mir in der Praxis als sinnvoll aufgefallen ist, kann im konkreten Projekt anders bewertet werden. Entscheidend ist aber, dass man diese Dinge im Blick behält, sich Gedanken macht und eine bewusste Entscheidung trifft. Wenn man die Dinge einfach nur geschehen lässt, gibt es eine hohe Wahrscheinlichkeit, dass dabei schlecht wartbarer Code herauskommt.