Handwerkliche Kunst im industriellen Softwareumfeld

embedded
Clean
code

Thomas Winz, Ostfildern 2017

Inhalt

Legende:

eCc konzentriert sich auf vier aufbauende Sichten, unterteilt in:

P: Beschreibt ein Prinzip dessen Einhaltung lohnt

A: Beschreibt eine Aktivität dessen Befolgung lohnt

Abstract

embedded Clean code (eCc) ist ein Fundament für ordentliche und vernünftige industrielle Softwareentwicklung

1. These: Der Entwickler trägt die alleinige Verantwortung für die Softwarequalität[1] und Wartbarkeit[2].

Die nicht funktionalen Anforderungen[3] heutiger Industriesoftware können ohne die Fähigkeiten professioneller Entwickler nicht erfüllt werden.

embedded Clean code handelt von einem respektvollen Umgang mit dem Quellcode. Dieses immaterielle Gut ist das wertvollste Kapital vieler Mittelständler. Deshalb zwingt die berufliche Verantwortung als Entwickler qualitativ hochwertige Software zu codieren.

2. These: Qualitätsabteilungen und IT-Administratoren, sowie das Projektmanagement stehen diesem Anspruch im Weg.

Minderwertige Codierung passiert nur dann,

- wenn wir nicht wollen,
- wenn wir nicht können,
- wenn wir nicht dürfen.

Aus diesem Grund finden sich hier allgemeine Praktiken und Aktivitäten, die keine spezifische Sprache erwarten, Werkzeuge voraussetzen oder von Prozessen blockiert werden.

Das *„Regelwerk embedded Clean code"* soll im gelebten Entwicklungsalltagsstress helfen, Entscheidungen unter Rücksichtnahme grundlegender Werte zu treffen.

"Wenn es hart auf hart kommt, vertrauen Sie auf ihre Disziplinen. Der Grund, warum Sie überhaupt diese Disziplinen haben, ist, dass sie ihnen in Zeiten mit hohem Druck Orientierung geben."[4]

Thomas Winz,

Ostfildern

---

1 https://de.wikipedia.org/wiki/Softwarequalität
2 https://de.wikipedia.org/wiki/Wartbarkeit
3 https://de.wikipedia.org/wiki/ISO/IEC_9126
4 Clean Coder, Verhaltensregeln für den professionellen Programmierer, Robert C. Martin

1. Die Funktion

Vernünftige Software ist sparsam und einfach

Definition Funktion:

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.[5]

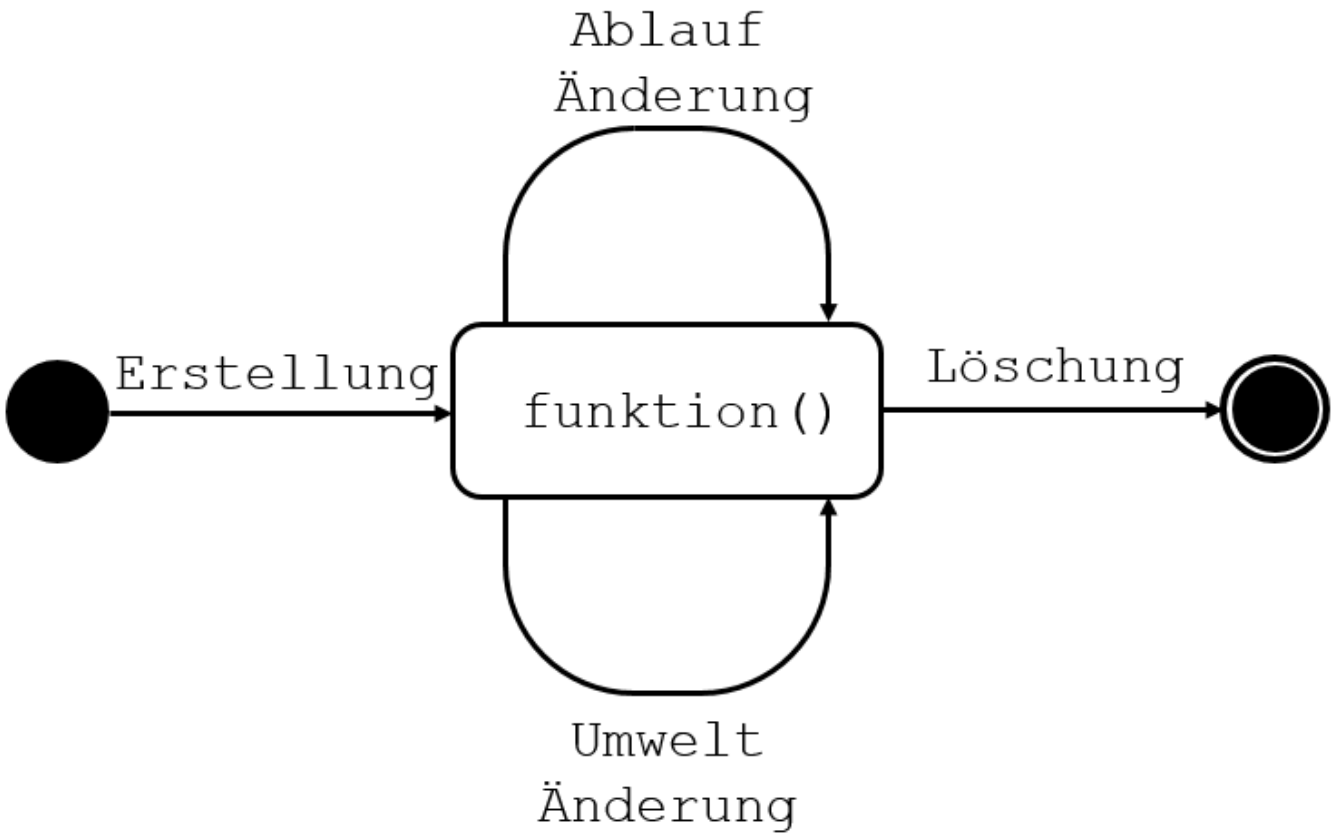
Die Kontrollstruktur einer Funktion[6] besteht aus drei fundamentalen Bedingungsanweisungen[7]:

- Sprung (goto)
- Wiederholung (while)
- Auswahl (if)

Daraus wird der Kontrollfluss[8] innerhalb der Funktion aufgebaut. Zuweisungen[9] [10] werden verwendet, um Daten zu manipulieren. Der Datenfluss[11] beschreibt Änderungen an Variablen durch eine Funktion (siehe P.6.).

Funktionen durchleben einen Lebenszyklus, wie Zeichnung 1 zeigt. Jeder Lebensabschnitt einer Funktion verlangt vom Entwickler ein angemessenes Verhalten:

[5] The c Programming Language, Brian W.Kernighan, Dennis M.Ritchie
[6] https://de.wikipedia.org/wiki/Funktion_(Programmierung)
[7] https://de.wikipedia.org/wiki/Anweisung_(Programmierung)
[8] https://de.wikipedia.org/wiki/Kontrollfluss
[9] https://de.wikipedia.org/wiki/Zuweisung
[10] https://de.wikipedia.org/wiki/Anweisung_(Programmierung)
[11] https://de.wikipedia.org/wiki/Datenflussdiagramm



10

Grafik 1: Lebenszyklus einer Funktion

A.1. Erstellung einer Funktion

Gib der einfachen Umsetzung den Vorrang

Der Basispfad[12] einer Funktionskontrollstruktur ist durchlaufen, wenn die Funktion das tut, was sie tun soll. Der Basispfad einer Funktion sollte eindeutig erkennbar und von kurzer Dauer sein.

Um diesen Basispfad wickeln sich weitere, weniger wichtige, unabhängige Nebenpfade. Die Aufgabe von Nebenpfaden ist meist die Überwachung oder Prüfung von Bedingungen, die den Basispfad gefährden:

Prüfung/ Überwachung	**Beschreibung**	**Beispiel**
Out of Range	Außerhalb des Wertebereichs	-1K
Sprungwert	Differenz zwischen zwei Wertepaaren ist zu hoch	5 K -> 200 K
SNA	Ungültiger Wert für Basispfad	0xFF 1/0

Diese Nebenpfade sind Quell vieler unbedachter Abhängigkeiten und sollten von der Anzahl geringgehalten werden.[13] [14]

Notiz:

[12] https://de.wikipedia.org/wiki/McCabe-Metrik
[13] https://de.wikipedia.org/wiki/Millersche_Zahl
[14] https://de.wikipedia.org/wiki/Äquivalenzklassentest

A.2. Ablauf Änderung einer Funktion

Verlasse eine Funktion verständlicher als vorgefunden

Prinzipiell führt die Ausprägung der Funktionskontrollstruktur zu einer Integrator-Funktion oder Operator-Funktion. Jede Kontrollstruktur liegt auf dem Spektrum zwischen diesen beiden Polen. Die folgende Tabelle zeigt Merkmale und Auswirkungen auf die Funktion:

Merkmal	**Operator**	**Integrator**
wird bestimmt vom	Datenfluss	Kontrollfluss
Aufgabe	Verarbeitung eines Datums	Bedingter Aufruf weiterer Funktionen
Parameter	Übergabe- und Rückgabe-Werte	void-Funktion

Durch weitere gewünschte Anpassungen der Funktion wird die ursprüngliche Kontrollstruktur erweitert. Es ist Wichtig, diesen Ursprung zu berücksichtigen, ansonsten verschiebt sich die Ausprägung der Kontrollstruktur vom gewünschten Pol. Damit gefährdet die gewachsene Kontrollstruktur die Verständlichkeit der Funktion. Es ist sinnvoll höher frequentierte Funktionen rigide zu reinigen.

Notiz:

A.3. Umwelt Änderung einer Funktion

Bei der Software-Optimierung ist Vorsicht geboten

Die Effizienz[15] einer Funktion wird vom Zeitverhalten, Speicherverhalten und der Konformität bestimmt. Je strikter diese Effizienzregeln festgelegt sind, umso größer deren Einfluss auf die Funktionskontrollstruktur.

Je weniger effizient die Funktion geschrieben wird, umso schwerer ist die Lesbarkeit. Die Funktionskontrollstruktur liegt auf dem Spektrum zwischen diesen beiden Polen.

Die Übererfüllung der Effizienzschwellen geht somit auf Kosten der Lesbarkeit und ist zu vermeiden.

Notiz:

[15] https://de.wikipedia.org/wiki/ISO/IEC_9126

A.4. Hintergrund: Technische Schuld

Messen der technischen Schuld mit Hilfe statischer Analysen

„Unter der technischen Schuld versteht man den zusätzlichen Aufwand, den man für Änderungen und Erweiterungen … einplanen muss" [16]

Der Grad der technischen Schuld ist maßgeblich von der Güte der Funktionskontrollstruktur und Datenkontrollstruktur der Funktionen abhängig. Technische Schulden müssen regelmäßig getilgt werden. Erfolgt dies nicht, kann es zu einen technischen Bankrott kommen, d.h. zusätzlicher Aufwand einer Änderung ist prohibitiv hoch.

Technische Schulden sind nicht-funktionale[17] Verletzungen und können nicht direkt gemessen werden. Mit Hilfe statischer Methoden[18] werden indirekt Faktoren[19] bestimmt um die technische Schuld zu ermitteln:

Faktor	**Beschreibung**	**Richtwert**
Funktionslänge	Anzahl der Zeilen	1 – 20 [max 50]
Instruktionsanzahl	Anzahl an Bedingungsanweisungen	1 – 5 [max 20]
Tiefe	Verschaltung von Auswahlanweisungen	0 - 3
Pfadanzahl	Unabhängige Pfade	1 - 5
Dichte	Verhältnis Kommentare zu Instruktionen	50 %
Funktionsaufruf (intern)	Wie viele Funktionen springt diese Funktion an?	0 – 5 [max 10]
Funktionsaufruf (extern)	Wie viele Funktionen rufen diese Funktion auf?	0 – 7 [max 10]
Rücksprung	Wie viele return hat die Funktion	1

[16] https://de.wikipedia.org/wiki/Technische_Schulden
[17] https://de.wikipedia.org/wiki/ISO/IEC_9126
[18] https://de.wikipedia.org/wiki/Statische_Code-Analyse
[19] z.B. HIS Metriken (http://www.itwissen.info/HIS-Herstellerinitiative-Software.html)

2. Die Komponente

Gutes Design ist immer der Aufgabe angemessen

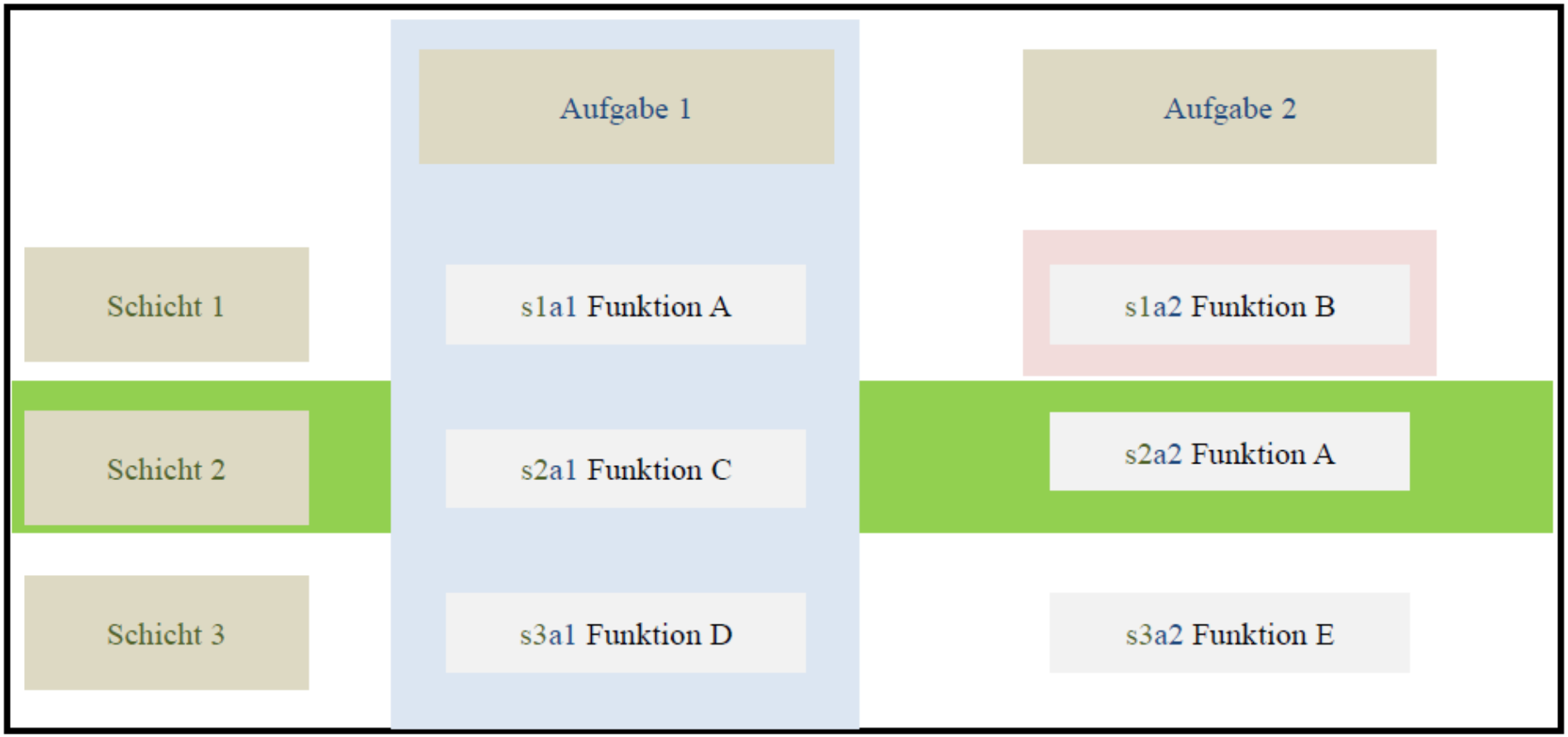
Definition Komponente:
Ein gekapselter Baustein mit vollständig definierten Schnittstellen, der innerhalb seiner Umgebung gegen einen Baustein mit identischen Schnittstellen ausgetauscht werden kann.[20]

Die Auswahlkriterien, welche Funktion in welche Komponente[21] gruppiert wird, ist eine systemrelevante Entscheidung und somit Teil der Architektur[22]. Eine wesentliche Architektur-Herausforderung ist die Strukturierung der Software in Komponenten.

Die Aufteilung der Funktionen innerhalb der Komponente erfolgt orthogonal zueinander. Horizontale Schichten trennen den technischen Aspekt[23] (Systemsichtweise), vertikale Schichten trennen den fachlichen Aspekt[24] (Kundensichtweise) (siehe Abbildung 4).

Somit stellt sich als Entwicklerfrage, wie Instruktionsketten in den einzelnen Funktionen innerhalb der Komponente verteilt werden. Mit den drei Prinzipien des angemessenen Designs werden kontinuierliche Änderungen an der Komponente handhabbar:

[20] Basiswissen für Softwarearchitekten, iSAQB
[21] https://de.wikipedia.org/wiki/Komponente_(Software)
[22] https://de.wikipedia.org/wiki/Softwarearchitektur
[23] https://de.wikipedia.org/wiki/Schichtenarchitektur
[24] https://de.wikipedia.org/wiki/Anforderung_(Informatik)



Grafik 2: Die graphische Interpretation der drei Prinzipien des angemessenen Designs

Legende:

P.1. – roter Kasten – Funktion kann durch Aufgabe in Matrix eindeutig gefunden werden

P.2. – blauer Kasten – horizontale Aufteilung der Komponente in Aufgaben

P.3. – grüner Kasten – vertikale Aufteilung der Komponente in technische Schichten

P.1. Eindeutige Verantwortung

Jede Funktion hat grundsätzlich nur eine Verantwortung

Für die Aufgabenbeschreibung der Funktion sollte nicht das Bindungswort[25] „und" verwendet werden (siehe Grafik 2, roter Kasten). Muss das Wort „und" verwendet werden, ist dies das Anzeichen, dass die Funktion überladen ist:

Code Beispiel:

```
//Die Funktion liest den Sensorwert
//UND bereitet das Signal auf.
SensorValue(){
 value = READ_SENSOR_PIN;
 value &= SENSOR_LEFT_MASK;
```

Um das Verhalten einer Komponente zu ändern, sollte gezielt nur die erwartete Funktion innerhalb der Komponente geändert werden. Es muss also gewährleistet werden, dass Instruktionsketten nachvollziehbar in einer eindeutigen Funktion innerhalb dieser Matrix eingeordnet sind.

Notiz:

[25] https://de.wikipedia.org/wiki/Konjunktion_(Wortart)

P.2. Aufgabenteilung

Für die Aufgabenbeschreibung der Komponente wird das Bindungswort [26] „und" verwendet (siehe Grafik 2, blauer Kasten). Jedes verwendete „und" trennt innerhalb der Komponente die Funktionen in verschiedene Aufgabenbereiche:

Code Beispiel:

```
// Die Komponente liest den Sensorwert
// UND bereitet das Signal auf
readSensorValue();
decodeSensorValue();
```

Diese Aufgabenbereiche umfassen eine Sammlung an Funktionen, wie es Komponenten tun. Diese haben aber keine Grenze zu anderen Aufgabenbereichen innerhalb der Komponente. Aus diesem Grund kann ein beliebiger Aufgabenbereich nicht durch einen anderen ausgetauscht werden.

Der erkaufte Vorteil ist, dass mehrere Aufgaben die gleiche Funktion nutzen. Damit entstehen technische Synergien. Innerhalb vom Aufgabenbereich sind die Funktionen wieder in bekannte technische Abstraktionsstufen gegliedert.

Notiz:

[26] https://de.wikipedia.org/wiki/Konjunktion_(Wortart)

P.3. Ordentliche Abstraktionsebenen

Die Instruktionen einer Funktion sollten einer Abstraktionsebene zugeordnet werden.

Welche Zahl passt nicht zu der Zahlenfolge: „2,4,6,7,8"? Dasselbe Prinzip wird angewendet, um die Abstraktionsebene innerhalb einer Instruktionskette zu ordnen (siehe Grafik 2, grüner Kasten):

Code Beispiel:

```
leseNachricht();
prüfeNachricht();
schickeAntwort();
RegisterCOM &= 0xFF; <- Instruktion mit falscher Abstraktion
```

Die Instruktionen einer Funktion sollten einer Abstraktionsebene zugeordnet werden. Die Einhaltung der Abstraktionsebene innerhalb der Funktion erhöht die Ordnung dieser Funktion. Im Beispiel gehört Funktion s2a1 zur Schicht 2 und sollte keine Instruktionen der Schicht 1 oder 3 ausführen. Das mentale Bild der Funktion hilft auch S2a2 zu verstehen, da diese Funktion sich in der gleichen Schicht befindet. Eine Verletzung am Wesen der Funktion kann somit eindeutig und zuverlässig erkannt werden.

Notiz:

A.5. Hintergrund: Modifizierbarkeit

Lebe eine konservative Codekonvention

"*Die Modifizierbarkeit von Software beschreibt, mit welchem Aufwand dieselbe an neue, zukünftige Anforderungen angepasst werden kann.*"[27]

Wie im Orchester die Instrumente muss jede Erweiterung der Funktion harmonisch zusammenwirken. Ohne diese Rücksicht hinterlässt der Entwickler eine persönliche Note und erhöht dadurch die technische Schuld (A.4.)

Hoher Verständnisaufwand ist eine direkte Folge der verschiedenen Entwicklungsstilmixe innerhalb der Funktion. Konstrukte, die sonst über die gesamte Projektsoftware einheitlich verstanden wurden, müssen im Sonderfall erneut erlernt werden.

Es gibt verschiedene Aspekte für Codekonventionen, die zusammengefasst den „Hausinternen Stil" beschreiben:

Aspekt	**Fachbegriff**	**Bsp.:**
Codier-Richtlinie	Defensiv	Misra
Formatierung	Stylecode	Googlesyle code[28]
Benamung	Naming Rules	

Wichtig ist, dass ein gemeinsamer Stil gelebt wird und nicht welcher Stil. Neue Projektmitglieder sollten diesen Stil im Quellcode erkennen und durch Schulungen, sowie weitere Maßnahmen, akzeptieren.

[27] https://de.wikipedia.org/wiki/Modifizierbarkeit
[28] https://google.github.io/styleguide/cppguide.html

3. Die Schnittstelle

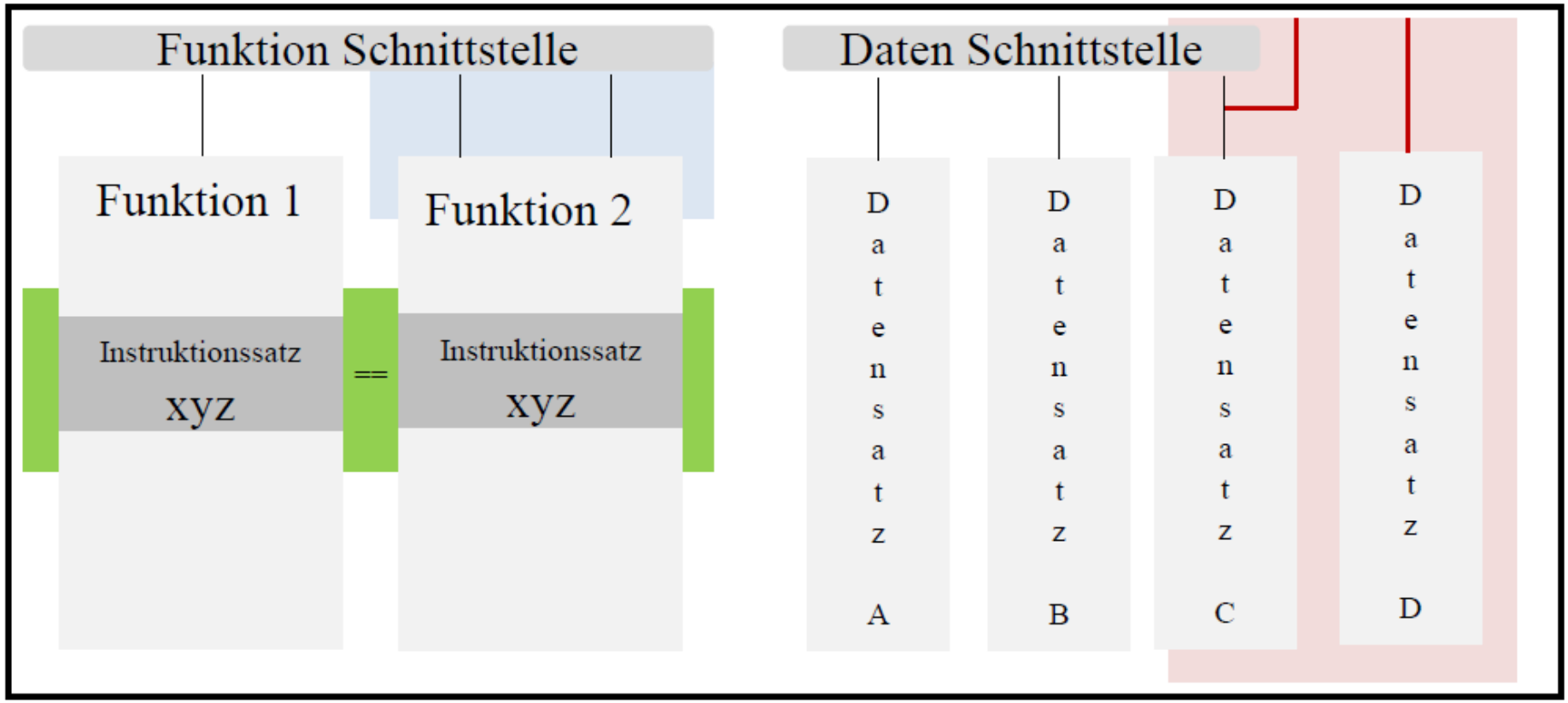
Rein symptomatische Codierung verspricht nur selten dauerhaften Erfolg

Definition Schnittstelle:
Eine Schnittstelle repräsentiert einen wohldefinierten Zugang zum System oder dessen Bausteinen [sic: Komponenten]. Dabei beschreibt eine Schnittstelle die Eigenschaften dieses Zugangspunkts…[29]

Architekturmuster[30] beschränken Zugriffe auf Funktionen und Daten über Schnittstellen[31]. Innerhalb einer Komponente sollte die Abhängigkeit der Funktionen untereinander höher sein, als zu Funktionen außerhalb der Komponente. Verletzungen durch Durchbrüche auf Informationen, die nicht von Schnittstellen freigegeben wurden, erhöhen die architektonische, technische Schuld.

Somit stellt sich als Entwicklerfrage, wie Schnittstellen ihrer Pförtnerpflicht nachkommen können. Mit den drei Prinzipen gegen symptomatische Codierung werden Durchbrüchen ein Riegel vorgeschoben:

[29] Basiswissen für Softwarearchitekten, iSAQB
[30] https://de.wikipedia.org/wiki/Architekturmuster
[31] https://de.wikipedia.org/wiki/Schnittstelle



Figur 3: Die graphische Interpretation der drei Prinzipien gegen symptomatische Codierung

Legende:

P.4. – blauer Kasten – unsaubere Schnittstelle

P.5. – roter Kasten – Verletzung der Datenkapslung

P.6. – grüner Kasten – Dopplung von Informationen

P.4. geringsten Überraschung

Es ist einfacher Code zu schreiben als zu verstehen

Der Name einer Schnittstelle weckt gewisse Erwartungen an deren Nutzung (siehe Figur 3, blauer Kasten). Dies gilt natürlich auch für alles, das in der Software durch einen Namen beschrieben wird. Diese Erwartung nicht zu erfüllen, führt zu hohen Aufwänden im Verstehen.

Code Beispiel:

```
void Funktion3(void) {
 // gelöscht, nicht ändern
 Funktion2();
}
```

Aus diesem Grund wird versucht, über Kommentare[32], schlecht benannte Dinge in weiterem Kontext zu beschreiben.

„Kommentare können komplizierte Sachverhalte klären aber asynchrone Kommentare können klare Sachverhalte komplizieren. [SIC]“ [33]

Deswegen sollte die Namensfindung einer Schnittstelle genauso lange dauern, wie der dazugeschriebene Kommentar. Und die Zeit der Kommentarpflege sollte besser in einen griffigen Namen investiert werden.

Notiz:

[32] https://de.wikipedia.org/wiki/Metadaten
[33] clean code, Robert Martin

P.5. Datenkapselung

Geregelter Zugriff auf Informationen verhindert Erosion

Datenschnittstellen vereinfachen die Nutzung von Informationen durch Abstraktion[34] und Kapslung[35] (siehe Figur 3, roter Kasten):

Code Beispiel:

```
zuKalt();
zuWarm();
```

Dies wird erreicht, indem der Datenzugriff auf die wesentliche Information beschränkt wird. Es werden keine Kenntnisse über die Beschaffung, Speicherung oder Aufbereitung der Dateninformation benötigt.

In Figur 3, erfolgt im roten Kasten ein Datenzugriffdurchbruch auf die Datensätze C & D. Dies ist eine technische Schuld. Daten C werden von einem nicht bekannten Zugriff manipuliert und D sollte überhaupt keine Abhängigkeiten nach außen haben. Es sollte überprüft werden wie diese Durchbrüche aufgelöst werden können.

Notiz:

[34] https://de.wikipedia.org/wiki/Abstraktion
[35] https://de.wikipedia.org/wiki/Datenkapselung_(Programmierung)

P.6. Redundanzvermeidung

Jede Duplizierung von Informationen bedeutet, dass Software-Konstrukte und Instruktionsketten in verschiedenen Teilen der Komponente im Überfluss geführt werden (siehe Figur 2, grüner Kasten).

Code Beispiel:

```
i++;      // kopiert aus Funktion
i=cos(y); // keine Ahnung warum
i=90;     // Test a2 war sonst Rot
```

Die Fehlerbehebung ist erschwert, da das fehlerhafte Verhalten unterschiedlich verdeckt wird. In Figur 2 ist der Instruktionssatz „xyz" sowohl in Funktion 1 als auch in Funktion 2 vorhanden. Der Wartungsaufwand für die gleiche Information ist höher als notwendig. So entstehen Dokumentationsaufwände, um die Synchronisierung zwischen den Redundanzen zu gewährleisten.

Notiz:

A.6. Hintergrund Wartung

Periodisches Refactoring der technischen Schuld

Wartungsarbeiten[36] am Projekt beginnen mit der ersten Auslieferung[37] und sind der Fixkostenbeitrag[38] der kontinuierlichen Entwicklung. Denn technische und fachliche Anforderungen verändern sich stetig. Ziel der Wartung ist es, die Lesbarkeit der Funktion zu erhöhen:

"... debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it? "[39]

Um produktiv zu codieren, muss die undisziplinierte Wartung strukturiert werden. Die Modifizierbarkeit (A.5.) der Funktionen wird als technische Schuld (A.4.) indirekt durch statische Methoden[40] gemessen. Um technische Schulden zu tilgen, werden gezielte Refactoring[41] Maßnahmen durchgeführt:

Faktor	**Beschreibung**	**Kürzel**
Namensanpassung	Aufgabe im Namen erkennbar	
Verschiebung	Instruktionsketten werden in andere / neue Funktionen verschoben	
Kommentarlöschung	Meta Information in Instruktionen sichtbar	

[36] https://de.wikipedia.org/wiki/Softwarewartung
[37] https://de.wikipedia.org/wiki/Entwicklungsstadium_(Software)
[38] https://de.wikipedia.org/wiki/Fixkosten
[39] "The Elements of Programming Style", 2nd edition, chapter 2, Brian Kernighan
[40] https://de.wikipedia.org/wiki/Statische_Code-Analyse
[41] https://de.wikipedia.org/wiki/Refactoring

4. Die Anforderung

Es muss mit Absicht gehandelt werden

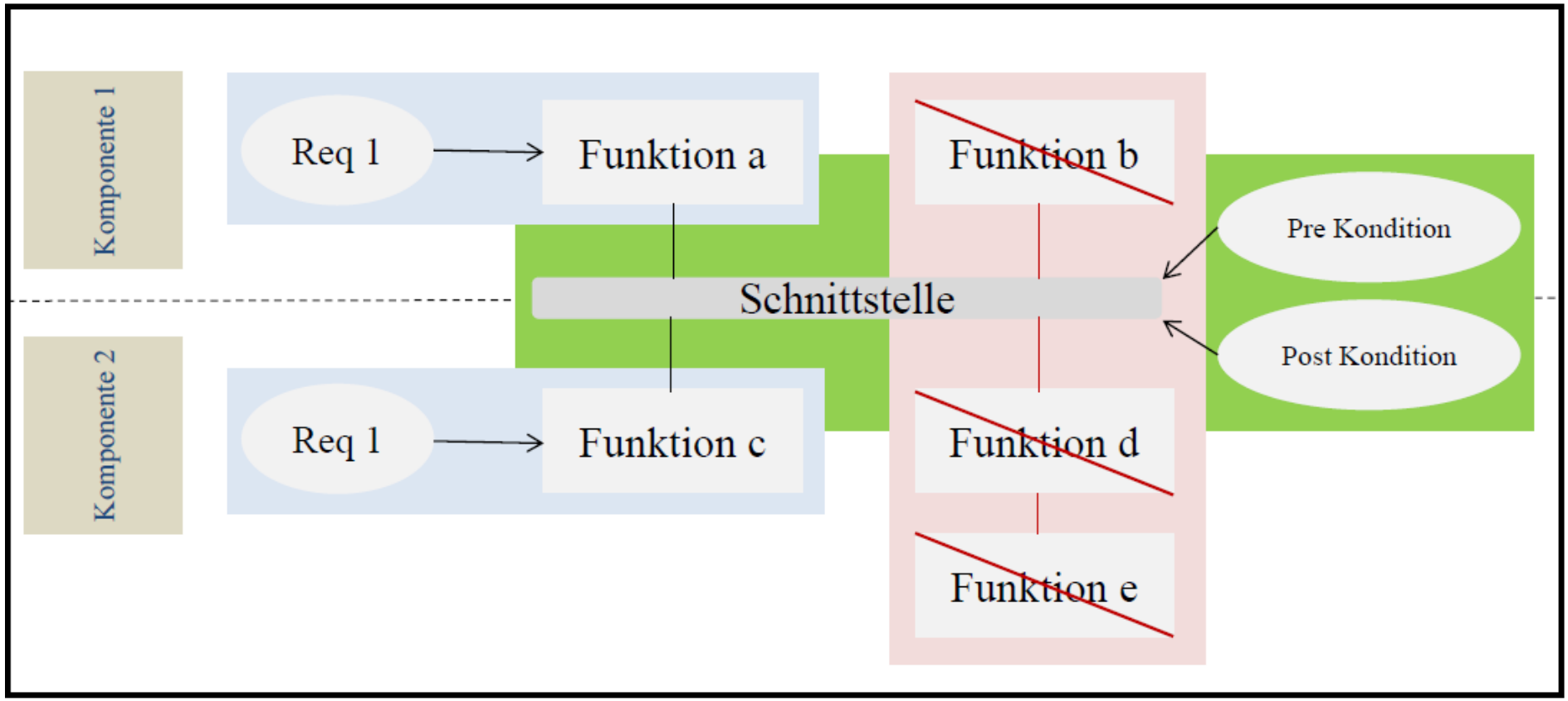
Definition Anforderungen:
Anforderungen beschreiben die Eigenschaften, die ein Softwaresystem besitzen muss, sowie Rahmenbedingungen, die für seinen Lebenszyklus (Entwicklung, Betrieb, Wartung) gelten.[42]

Software ist aufgrund ihrer aufwändigen Herstellung ein wertvolles Gut. Die Anforderungen[43] bestimmen, welche technischen und fachlichen (sowie qualitativen Kriterien) von der Softwarearchitektur umgesetzt werden müssen.

Damit stellt sich die Entwickler Frage, wie eine Forderung an Funktionalität mit erkennbaren Absicht umgesetzt werden kann. Diese zwei Prinzipien helfen dabei, nicht den Fokus zu verlieren:

[42] IEEE Std. 610.12-1990

[43] https://de.wikipedia.org/wiki/Anforderungsmanagement



Figur 3: Es sollte immer ein Anlass für die Handlung geben

Legende:

P.7. - blauer Kasten - Notwendigkeit

P.8. - grüner Kasten - Entwurfsvereinbarung

A.7. - roter Kasten - bedachte Löschung

P.7. Notwendigkeit

Notwendiges Verhalten der Software sind Funktionen in der Komponente, die sich auf mindestens eine Anforderung beziehen (siehe Figur 4, blauer Kasten).

Code Beispiel:

```
/** @req A_11 */ <- Verweis auf Anforderung
void zuKalt(void){
```

Mit Hilfe der „Abdeckungsgrad Messungen“[44] kann geprüft werden, welche Pfade einer Funktion sich auf welche Anforderung zurückgeführt. Wichtig ist, dass alle Anforderungen bedacht werden. Pfade, die in der Funktion vorhanden sind aber nicht von einer Anforderung stammen, sind technische Schulden und werden „nicht erwünschtes Verhalten“ genannt.

Notiz:

[44] https://de.wikipedia.org/wiki/Testabdeckung

P.8. Entwurfsvereinbarung

Um Komponenten beliebig austauschen zu können, muss das Verhalten der Schnittstelle exakt beschrieben werden.[45] Die Nutzung der Schnittstellenfunktion wird über preconditions und postconditions eingeschränkt (siehe Figur 4, grüner Kasten)

Code Beispiel:

```
/** @preCon Funktion nur nach TemperaturMessung() aufrufen */
void zuKalt(void);
```

Zeitliche Abhängigkeiten von Schnittstellenfunktionen sind ohne Entwurfsvereinbarungsinformationen von außen nicht erkennbar. Entwurfsvereinbarungsinformationen werden auch indirekt über die Benamung von Schnittstellen kommuniziert. Dies erschwert die Nutzung der Komponente.

Notiz:

[45] https://de.wikipedia.org/wiki/Design_by_contract

A.7. Löschung einer Funktion

Entferne niemals Software unvorsichtig

Das Alter einer Funktion dient als Indikator für dessen Qualität[46]. Das Verhalten der Funktion, auch Abweichungen von den Anforderungen, sind bekannt und erfolgreich umgesetzt worden.

Deswegen ist der Verständnismangel[47] über eine Funktion niemals als Grund für die Löschung der Funktion zu wählen. Denn es fehlt am notwendigen Wissen über die weiteren Zusammenhänge. Funktionen sollten nur gelöscht werden, wenn weder eine fachliche noch technische Anforderung vorhanden ist.

Ist der Entschluss gefallen eine unerwünschte Funktionalität zu entfernen, erfolgt eine Auswirkungsanalyse. Ziel ist die Aufdeckung des kompletten unnötigen Ausführungspfades (siehe roter Kasten). Ansonsten entstehen in der Software tote Ausführungspfade[48].

Notiz:

[46] https://de.wikipedia.org/wiki/ISO/IEC_9126
[47] https://de.wikipedia.org/wiki/Verstehen
[48] https://de.wikipedia.org/wiki/Toter_Code

A.8. Hintergrund: Review

Lebe eine freie Review Kultur

Die Auswahl der Wartungsarbeit Maßnahmen (A.6.) werden im Review getroffen. Dazu werden die technischen Schulden (A.4) unter dem Aspekt der Wartbarkeit (A.5.) bewertet.

Wichtig ist, dass eine automatisierte Aggregierung der technischen Schuldfaktoren vorgenommen wird. Denn erst im Zusammenhang und im Zeitverlauf können konkrete Trends erkannt werden:

„… dass Reviews Kosten Verursachen und als Ausgleich dafür bessere Qualität bekommt. … Richtig dagegen ist, dass man Qualität bekommt und zusätzlich Zeit einspart." [49]

Das Review ist nur solange eine wissenschaftlich neutrale Methode, wenn aggregierte und anonyme Trends bewertet werden. Direkte Prüfungen der Arbeitsprodukte erfordert ein Urvertrauen[50] untereinander. Hierzu muss der respektvolle[51] Umgang untereinander konsequent verteidigt werden. Nur durch diese Wertschätzung[52] kann eine fruchtbare Review Kultur gelebt werden.

10

Notiz:

[49] Populäre Irrtürmer und Fehleinschätzungen in der Reviewtechnik, Peter Rösler
[50] https://de.wikipedia.org/wiki/Urvertrauen
[51] https://de.wikipedia.org/wiki/Respekt
[52] https://de.wikipedia.org/wiki/Wertschätzung

www.embedded-clean-code.de