embedded Clean code

Der Softwerker als Zentrum der Industriesoftwareentwicklung

Thomas Winz, softwareimotion

"Nerdy hatte die Nase voll von diesem DinoPark Projekt. Obwohl man bereits der Terminplanung hinterherhinkte, verlangte InGen noch umfangreiche Modifikationen am System, war aber nicht bereit, dafür zu bezahlen; die Geschäftsleitung argumentierte, dies sei Teil des ursprünglichen Vertrages. Man drohte mit gerichtlichen Schritten." **[Ref1]**

Software ist ein wertvolles Gut, das im Umfeld größter Unsicherheit entsteht. Ohne eine ordentliche und vernünftige industrielle Softwareentwicklung sprengt dieser Spannungsbogen jewedige Kosten-/Nutzung-Rechnung; genau hier setzt eCc (embedded Clean code) an.

Einführung

Moderne Ansätze der Personalführung haben zur Folge, dass kritische Projektentscheidungen im Tagesgeschäft durch einzelne Entwickler fallen. Die Ausweitung der Entscheidungsträger auf das gesamte Projektteam ist mit Risiken verbunden. Die klaren Regeln von eCc helfen dem Entwickler sich bei der Codierung zu fokussieren und so dieses hohe Risiko bewusst zu tragen.

Hintergrund

embedded Clean code konnte 2014 auf dem Embedded Software Engineering Kongress seine Weltpremiere feiern. [Ref 2]. Das Regelwerk eCc dient als Fundament für ordentliche und vernünftige industrielle Softwareentwicklung (Siehe Abbildung1).
Die 2014 vorgestellte Theorie umfasst drei große Themengebiete:

- Überflüssige Codierung vermeiden
- Gutes Design ist immer angemessen
- Es ist immer einfacher Code zu schreiben, als zu verstehen

Dieser Bandbeitrag baut auf diesen "best practice" Grundlagen von eCc auf und geht dieses Jahr auf die Aktivitäten von eCc ein.

Abbildung 1: Übersicht über embedded Clean code

embedded Clean code

Vernünftige Software ist sparsam und einfach

A.4 Entferne niemals Software unvorsichtig
Nur mit einem Code unter Versionsverwaltung ist es möglich, mit Bedacht Funktionen vollständig zu löschen.

A.3 Verlasse eine Funktion verständlicher als vorgefunden
Minimalinvasive Restrukturierungen sollten zeitnah durchgeführt werden.

A.2 Bei Software-Optimierung ist Vorsicht geboten
Optimiere nur wenn der Engpass identifiziert wurde und die Verbesserung gemessen werden kann.

A.1 Gib der einfachen Umsetzung den Vorrang
Bevorzuge eine klare und verständliche Lösung, um die Anforderung umzusetzen.

Rein symptomatische Codierung verspricht nur selten dauerhaften Erfolg

1.1 Prinzip der Redundanz-Vermeidung
Dieses Design-Prinzip besagt, dass jede Information eine einzige, eindeutige, verbindliche Darstellung innerhalb der Software hat.

1.2 Prinzip der geringsten Überraschung
Dieses Design-Prinzip besagt, dass jede Funktion sich so Verhalten soll wie erwartet.

1.3 Prinzip der Datenkapslung
Dieses Design-Prinzip besagt, dass der direkte Zugriff auf die interne Datenstruktur unterbunden wird.

A.5 Lebe eine freie Review Kultur
Regelmäßige Reviews gewährleisten langfristige Zuverlässigkeit sowie eine ordentliche Software.

Gutes Design ist immer angemessen

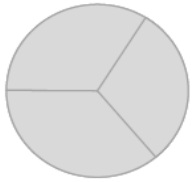
2.1 Prinzip der eindeutigen Abstraktionsstufe
Dieses Design-Prinzip besagt, dass jede Funktion auf einem erkennbaren Detaillierungsgrad liegt.

2.2 Prinzip der eindeutigen Verantwortung
Dieses Design-Prinzip besagt, dass jede Funktion eine einzige Verantwortung vollständig umsetzt.

2.3 Prinzip der eindeutigen Aufgabenteilung
Dieses Design-Prinzip besagt, dass jede Funktion einem eindeutigen Aufgabenbereich zugeordert ist.

A.6 Lebe konservative Codekonventionen
Uniformität im Code kann nur durch regelmäßige Pflege bewahrt werden.

Es ist immer einfacher Code zu schreiben als zu verstehen



3.1 Prinzip der Entwurfsvereinbarung
Dieses Design-Prinzip besagt, dass jede Schnittstelle eindeutig im Entwurf definiert ist.

3.2 Prinzip der Notwendigkeit
Dieses Design-Prinzip besagt, dass nur klar kommunizierte Anforderungen umgesetzt werden.

3.3 Prinzip der Code-Reinigung
Dieses Design-Prinzip besagt, dass tote Codepfade vollständig unter Berücksichtigung der Seiteneffekte zu entfernen sind.

A.7 Periodisches Refactoring mit Hilfe statischer Analysen
Statische Analysen dienen der Legimitation jeder geplanten Softwareverbesserung.

Werte für Gruppenarbeit

Ohne Vertrauen und eine moralische Wertebasis können Menschen nicht zusammenarbeiten. Dies ist auch die fundamentale Überzeugung aller agiler Methoden. Die in eCc beschriebenen Aktivitäten werden gelebt um Fehler zu vermeiden. Aber das bedeutet, dass der einzelne Entwickler seine Schwächen der Gruppe aufzeigen muss. Dazu muss der einzelne Entwickler Vertrauen in seine Kollegen aufbauen.

Das heißt, dass nur in einer gesunden Gruppenkultur die eCc Methoden überhaupt gelebt werden können.

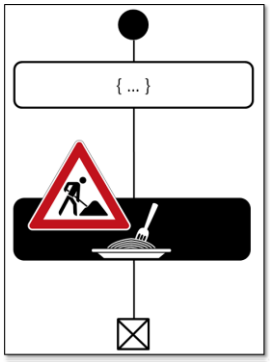
Aber diese grundlegenden Werte würden die Umfänge von eCc sprengen. Deswegen wird sich darauf beschränkt, die Werte von "Extreme Programming" aufzuführen. Diese Werte zeigen seit 2004 in tausenden Projekten ihre Wirkung und sind somit eine gute Grundlage für eCc.

Wert	Beschreibung
Einfachheit	Einfachheit ist ein Zustand, der sich dadurch auszeichnet, dass nur wenige Faktoren zu seinem Entstehen beitragen.
Kommunikation	Alle Projektmitglieder sollen intensiv miteinander kommunizieren. Durch persönliche Gespräche lassen sich Missverständnisse schneller ausräumen.
Feedback	Um eine hohe Qualität zu erzielen, nämlich das zu erreichen, was der Kunde benötigt, werden sehr kurze Feedback-Schleifen verwendet, um den Kunden kontinuierlich die Entwicklung zu zeigen.
Mut	Diese Werte einzusetzen und dabei offen zu Kommunizieren erfordert sehr viel Mut, besonders für die Projektmitglieder, die das Handeln nach diesen Werten nicht gewohnt sind.
Respekt	Das Fundament dieser Werte ist der Respekt; Respekt bezeichnet eine Form der Wertschätzung, Aufmerksamkeit und Rücksicht gegenüber allen anderen Teammitgliedern.

Tabelle 1: Extreme Programming Werte [Ref 3]

Die grundsätzlichen Software Aktivitäten

"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." [Ref4]

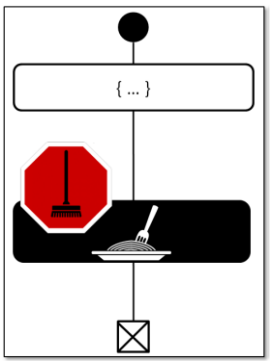


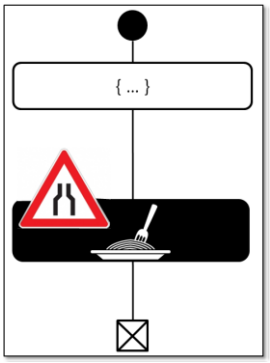
Gib der einfachen Umsetzung den Vorrang

Laut ISO-26262 ist ein Ziel vom Unit Test, ein Beweis, dass Software kein unerwünschtes Verhalten aufweist [REF 5]. Wenn gewünschtes Verhalten als "durch Anforderungen definiert" verstanden wird, bedeutet dies, dass jede Funktionalität ohne Anforderung die Software unnötig kompliziert und potentiell gefährdet. So muss davon ausgegangen werden, dass nicht alle Abhängigkeiten bedacht worden sind.

Verlasse eine Funktion verständlicher als vorgefunden

Wird eine Funktion nach ihrer Erstellung erneut geändert, ist davon Auszugehen, dass weitere Anpassungen stattfinden werden. Deswegen ist es sinnvoll diese höher frequentierten Codestellen rigide zu reinigen, um diesen Änderungsaufwand über die Projektlaufzeit insgesamt zu minimieren.





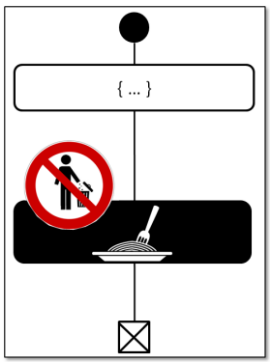
Bei der Software-Optimierung ist Vorsicht geboten

Jede Optimierung ohne Ziel ist Zeitverschwendung, da die Verbesserung nur subjektiv vom einzelnen Entwickler gemessen werden kann. Nicht bedachte Abhängigkeiten zu weiteren Teilsystemen können schwerwiegend gestört werden. Nur statische und architektonische Analysen können als Legitimation für Änderung herangezogen werden.

Entferne niemals Software unvorsichtig

Es ist absurd anzunehmen, dass eine Neuentwicklung besser als die vorhandene Funktion ist. Im Gegenteil, das Alter einer Funktion dient als Indikator für die Qualität der Funktion.

Das Verhalten der Funktion, auch Abweichungen von den Anforderungen, sind bekannt und schon mehrfach erfolgreich umgesetzt worden. Somit kann davon ausgegangen werden, dass die Neuentwicklung bekannte Fehler erneut enthält.



Lebe eine freie Review Kultur

„… dass Reviews Kosten Verursachen und als Ausgleich dafür bessere Qualität bekommt. … Richtig dagegen ist, dass man Qualität bekommt und zusätzlich Zeit einspart.[" Ref 6]

Viele Standards verlangen Reviews von Arbeitsprodukten, weswegen eine Vielzahl von Reviewarten existieren. Trotzdem kämpft diese Technik um Akzeptanz.
Wichtig ist, dass ein Review weitestgehend automatisiert wird. Dazu zählt zunächst einmal die Durchführung. Die Kommunikation, Planung und Verfolgbarkeit sollte ohne Werkzeugbrüche erfolgen. Dabei muss auch das Management akzeptieren, dass dieser Aufwand vom Entwickler verbucht werden darf. Wichtiger ist aber, dass das Arbeitsprodukt mit Werkzeugen analysiert wird, um einfache Routine-Fragen an das Arbeitsprodukt zu erfassen.
Dadurch wird das Review zu eine wissenschaftliche Methode. Persönliche Streits um Geschmäcker wird unterbunden. Anonymisierungen tragen dazu bei, dass Gruppenstreitereien nicht in Reviews ausgetragen werden.

Lebe eine konservative Codekonvention

"*Die Modifizierbarkeit von Software beschreibt, mit welchem Aufwand dieselbe an neue, zukünftige Anforderungen angepasst werden kann.*" [Ref 7]

Modifizierbarkeit ist Teil der nicht funktionalen Anforderung an die Wartbarkeit der Software. Je mehr der einzelne Entwickler seinen Abdruck in der Software hinterlässt, umso höher ist der Aufwand für andere Entwickler dessen Willen zu erkennen. Konstrukte, die sonst über die gesamte Projektsoftware einheitlich verstanden wurden, müssen somit erneut im Sonderfall erlernt werden. Damit ist jeglicher Mehrwert der Individuallösung hinfällig.
Dieses Verhaltensmuster ist aber mehr als eine willentlichen Irritation. Im Alltag mag es zu wesentlichen Einsparungen für den Einzelnen kommen. Die Fehlerfindung geschieht meist unter einem erhöhten Zeitdruck, somit kann im entscheidenden Moment diese schlampige Einstellung existenzbedrohend werden.

Periodisches Refactoring mit Hilfe statischer Analysen

"... 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? " [Ref 8]

Im Verlauf vom Projekt versandet die Software. Einst klare Schnittstellen wurden durchbrochen und "Workarounds" sind etabliert. Ein Grund hierfür ist die Konzentration auf die Umsetzungen von Anforderungen. Auch der "Bugfix" führt zur Softwareerosion.
Kosteneffiziente Umsetzungen erfordern Kompromisse zur bestehenden Architektur. Somit muss periodisch eine systematische Evaluierungen erfolgen, ob die Software noch der Architektur entspricht. Die "Workaround"-, "Bugfix"- und Durchbruch-Stellen werden identifiziert und in eine ordentliche Struktur gebracht.
Wichtig ist, dass solche Evaluierungen mechanisch durch Werkzeuge unterstützt werden. Damit können leichter Strukturen aufgedeckt werden und ein wissenschaftlicher Ansatz ersetzt die emotional Richtungsdebatten.

Zusammenfassung
Der Entwickler trägt die alleinige Verantwortung für die Softwarequalität und Wartbarkeit. Die nicht funktionalen Anforderungen wie Fehlertoleranz, Verständlichkeit und Portierbarkeit heutiger Industriesoftware können ohne die Fähigkeiten professioneller Entwickler nicht erfüllt werden.
Mithilfe von eCc soll dem Entwickler ein Leitfaden an die Hand gegeben werden, um diese Werte auch im Alltagsstress zu leben.

Über den Autor
2010 hat Thomas Winz an der HTWG Konstanz seinen Abschluss als Bachelor of Science „Technische Informatik“ erhalten. Seitdem ist Thomas Winz als Softwareentwickler bei softwareinmotion GmbH im Automotive-Umfeld tätig. In dieser Zeit konnte er in unterschiedlichen Projekten die Notwendigkeit der ordentlichen Softwareentwicklung erlernen. Seit 2013 arbeitet Thomas Winz am *embedded Clean code* – Programm, um so diese Erfahrungen in verschiedenen Vorträgen und Schulungen weiterzugeben.
Website von ecC: http://Embedded-clean-code.de
Schulungsangebot: http://www.softwareinmotion.de/academy

Rechtliche Situation zu embedded Clean code
Wir suchen Partner um das Regelwerk deutschlandweit bekannt zu machen.

Quellenverzeichnis

Ref	Titel	Autor/ Link
1	Dino Park	Michael Crichton
2	embedded Clean code; eseKongress 2014	http://2014.ese-kongress.de/paper_revised/list Embedded Clean Code
3		https://en.wikipedia.org/wiki/Extreme_programming
4	Clean Coder Verhaltensregeln für den professionelle Programmierer	Robert C. Martin
5	INTERNATIONAL STANDARD Road vehicles — Functional safety	ISO 26262-1 First edition 2011-11-15
6	Populäre Irrtürmer und Fehleinschätzungen in der Reviewtechnik	Peter Rösler
7		https://de.wikipedia.org/wiki/Modifizierbarkeit
8	"The Elements of Programming Style", 2nd edition, chapter 2	Brian Kernighan