

## Anforderungen im Software Engineering

Wenn ich heute zu unseren Kunden komme und für die Planung von neuen Projekten über die Gestaltung der Entwicklungsumgebung spreche, dann treffe ich immer wieder auf sehr ähnliche Anforderungen in Bezug auf das wünschenswerte Software Engineering.

Im Grunde sind es die gleichen Anforderungen wie ich sie schon vor 10 Jahren gehört habe, der einzige Unterschied: Heute sind sie noch akuter, die Probleme sind noch offensichtlicher, Zeitdruck und/oder Qualitätseinbußen sind noch einmal gestiegen.

Das Bewusstsein, dass am Vorgehen der Softwareentwicklung dringend etwas geändert werden muss ist offensichtlicher denn je. Im Wesentlichen können die Probleme und die daraus resultierenden Anforderungen wie folgt umschrieben werden:

Zu selten werden Software-Projekte auf Basis eines neuen Architektur-Designs entwickelt. Ein vor Jahren entwickeltes Design lebt bis heute und ist inzwischen an allen Ecken und Kanten erweitert, nachgebessert, umgeschrieben ...

Die ursprünglichen Strukturen sind nicht mehr zu erkennen. Evtl. vorhandene Dokumentation entspricht nicht dem Stand des Sourcecodes. Auf Basis solcher Sourcen kann nicht mehr effizient genug entwickelt werden. Änderungen an einer Funktion wirken sich ungewollt in anderen Bereichen der Applikation aus. Erweiterungen müssen auf Basis komplexer Strukturen durchgeführt werden, der Überblick fehlt und damit die Sicherheit vor ungewünschten Effekten. Hin und wieder kommt es sogar vor, dass Teile der Applikation vor langer Zeit von inzwischen ausgeschiedenen Mitarbeitern (oder extern) entwickelt wurden und das heutige Engineering-Team nicht mehr in der Lage ist in diesen Teilen überhaupt Änderungen vorzunehmen, weil eine Einarbeitung jeden zeitlichen Rahmen sprengen würde.

Um die gewünschte Qualität unter diesen Bedingungen aufrecht zu erhalten erhöht sich der Testaufwand. Und das unter permanent steigendem Zeitdruck.

Gleichzeitig wünschen Produkt-Management und/oder Vertrieb, dass neue Funktionen am Besten gestern implementiert wurden, um die geplanten Verkaufszahlen erreichen zu können.

Warum aber werden notwendige Schritte, um diesen Zustand zu verbessern, von einem Projekt zum nächsten immer wieder vor sich her geschoben?

Eine Zeit lang waren die besonderen Bedingungen von Embedded Projekten ein Grund. Das es inzwischen praktikable Software Engineering Lösungen auch für den Einsatz in Embedded Projekten gibt und wie diese aussehen, davon handelt dieser Artikel.

## Wie UML nicht funktioniert

Aber zuerst möchte ich damit anfangen wie es nicht funktioniert. Seit Jahren wird im Bereich der Embedded Software Entwicklung auf Basis der Notation ANSI-C programmiert. Gegenüber der davor üblichen Programmierung in Assembler hat diese sogenannte Hochsprache Vorteile. Ein wesentlicher sind Sprachkonstrukte, die die so genannte strukturierte Programmierung (Prozedurale Programmierung) als Engineering Methode unterstützt, welche die Notation Assembler nicht besitzt. Das ist der Hauptgrund, warum die Softwareentwicklung auf Basis einer Hochsprache wesentlich effizienter ist, als auf Basis von

Assembler. Die Unterschiede beider Techniken sind so groß, dass wir bei dem Übergang von einem Paradigmen-Wechsel sprechen.

Seit der Einführung der Hochsprache ANSI-C hat es keinen weiteren Paradigmenwechsel im Software Engineering gegeben und das ist heute überall zu spüren. Mit der Methode „Strukturierte Programmierung“ auf Basis der Notation „ANSI-C“ und den Tools „Compiler, IDE und HLL-Debugger“ sind in vielen Fällen die Anforderungen an das Software-Engineering nicht mehr zu erfüllen.

Es steht ein neuer Paradigmen-Wechsel an, hin zur Methode „Objekt Orientiertes Design“ auf Basis der Notation „UML“ (Unified Modelling Language) und mit Hilfe von so genannten „CASE“ (Computer Aided Software Engineering) Tools.

Sie sehen schon, andere Methode, andere Notation, andere Tools und Sie denken das klingt nach einem Haufen Aufwand, nach langen Einarbeitungszeiten, nach hohen Investitionen und letztendlich nach großem Risiko. Ja, Sie haben recht und genau das ist der Grund warum dieser Schritt so gerne vor sich her geschoben wird.

Aber es gibt eine Alternative. Wie wäre es, diesen gewaltigen Schritt in kleine verdauliche Häppchen mit entsprechend kurzer Einarbeitungszeit, geringen Investitionen und kleinem Risiko aufzuteilen? Genau so wird es sehr häufig gemacht.

Ein sehr beliebtes Vorgehen dieser Art ist es die UML auf Basis eines preiswerten grafischen UML Editors erst einmal zu Dokumentations-Zwecken einzusetzen. Die Frage, die Sie sich stellen sollten ist grundsätzlich: „Löst das mein Problem?“. Worin liegt denn das Problem genau genommen. Nicht unbedingt daran, dass keine Dokumentation existiert, sondern dass sie nicht dem aktuellen Stand der Software entspricht. Nachdokumentieren ist nicht einer der beliebtesten Tätigkeiten und unter Zeitdruck eine der ersten, die unter den Tisch fällt. Warum sollte sich das dadurch ändern, dass nun mit UML und einem grafischen Editor dokumentiert wird?

Ich kann Ihnen aus Erfahrung mit verschiedenen Projekten, die ich betreut habe sagen: Dieses Vorgehen löst das Problem nicht wirklich und ist ohne entsprechende Disziplin zur Dokumentation hinausgeworfene Zeit und Geldverschwendung. Disziplin hat wiederum nichts mit UML zu tun und kann auch mit anderen Mechanismen zu besserer Dokumentation führen. Die oft erlebte Praxis ist, dass Zeitdruck gegenüber der Disziplin überwiegt und das erwartete Ergebnis bleibt aus.

Nun höre ich häufig das Argument: „Aber auf diese Weise können wir erst einmal die UML erlernen und Erfahrungen mit ihr sammeln. Später kann dann ein weiterer Schritt folgen.“ Leider funktioniert auch das nicht. Ein Vergleich mit dem letzten Paradigmenwechsel macht es offensichtlich. Stellen Sie sich vor, man hätte Ihnen zum Erlernen der Hochsprache ANSI-C erst einmal ein ANSI-C Handbuch und einen Editor, aber noch keinen Compiler zur Verfügung gestellt. Nun hätten Sie ja fleißig in C programmieren können, bzw. in Form von Pseudo Code Ihre Assembler-Sourcen besser dokumentieren können. (Das wurde übrigens auch damals häufig versucht).

Aber wenn wir realistisch sind, Sie haben nicht die geringste Chance das Programmieren zu erlernen bzw. Erfahrungen zu machen ohne einen Compiler und der Möglichkeit das Programm auszuführen.

Warum sollte es in UML anders sein? Wer sagt Ihnen, dass Sie für die Assoziation das Symbol der Aggregation verwendet haben? Oder den Pfeil für einen Synchronen Event für einen Asynchronen eingesetzt haben. Die UML ist eine genau so exakte Notation, wie die Hochsprache ANSI-C und für eine erfolgreiche Einarbeitung benötigen Sie genau so Feedback wie für eine Hochsprache. Warum sollte es auch anders sein?

Eine Alternative zum obigen Vorgehen ist es, nur das Architektur-Design auf Basis der UML zu designen. Dieses kann dann automatisch in C-Rümpfe umgesetzt werden und diese werden denn herkömmlich in C implementiert.

Aber auch hier kann ich prophezeien, dass Ihnen nach einiger Zeit (insbesondere bei großem Zeitdruck) das Hin und Her zwischen UML Tool und IDE leid sein wird und Sie Änderungen, die eigentlich im UML Design durchgeführt werden müssten, nun direkt in C

gemacht werden. Nun haben Sie dasselbe Problem auf einer anderen Ebene, Design und Code sind nicht mehr kongruent. Vorher waren es Dokumentation und Code.

## Wie UML funktioniert

Wenn wir uns die realen Requirements im Softwareengineering ansehen, dann gilt es, Arbeiten, die auf Grund von Redundanzen im Verlauf der Entwicklung Änderungen an mehreren Stellen (Dokumenten) nach sich ziehen, so gering wie möglich zu halten. Die Erfahrung von weit mehr als 50 Projekten, in denen die Einführung der UML begleitet wurde zeigt, dass aus diesem Grund nur der konsequente Einsatz der UML zur Modellierung mit anschließender Codegenerierung langfristig erfolgreich ist. Gleichzeitig wird Reverse Engineering benötigt, um existierenden Code nicht wegschmeißen zu müssen, und sogenanntes Round Trip-Engineering, um bei bestimmten Gelegenheiten auch einmal schnell auf der C-Ebene Änderungen durchführen zu können. Automatisches Round-Tripping sorgt dafür, dass diese Änderungen auch auf Modellebene nachgezogen werden.

Um noch genauer verstehen zu können, dass nur dieser konsequente Ansatz zu Erfolg führen kann möchte ich exemplarisch auf einen kleinen Auszug aktueller Anforderungen und deren Lösung eingehen.

### Bessere Dokumentation.

Wie bereits oben angesprochen ist das eigentliche Problem oftmals nicht die Dokumentation an sich, sondern der veraltetete Stand, hervorgerufen durch Zeitmangel um Dokumentation parallel zum Code zu pflegen. Um diese Situation zu verbessern gibt es zwei prinzipielle Möglichkeiten:

1. Aus dem C-Code automatisch die Dokumentation erzeugen. Auf Basis von Tools wie z.B. DoxyGen ist so etwas möglich und wird häufig eingesetzt. Das hat grundsätzlich jedoch den Nachteil, dass nicht mehr Information erzeugt werden kann, als in den C-Quellen enthalten ist, und das Ergebnis wird immer unzureichend sein, wenn Architektur-Beschreibungen oder Experten-Informationen gewünscht werden.
2. Der andere Ansatz geht genau umgekehrt, die so genannte Vorwärts-Dokumentation. Erst wird die Dokumentation erstellt und daraus auf Basis eines Codegenerators ablauffähiger C-Code generiert.

Ein Haupt-Argument gegen dieses Vorgehen war lange Zeit die Effizienz der Codegeneratoren. Dieses Argument ist heute in der Regel unbegründet. Es gibt sehr effiziente Codegeneratoren. Dieser Weg ist jedoch nur auf Basis einer formalen Notation möglich, die sowohl Codierungs- als auch Informations-Konstrukte enthält. Das tut die UML.

### Wiederverwendung

Wenn von Wiederverwendung gesprochen wird, dann wird zuerst immer an den C-Code gedacht. Aber es gibt viele weitere Bereiche, in denen Wiederverwendung die Effizienz steigern kann.

Stellen Sie sich vor, Sie arbeiten auf Basis eines Pflichtenheftes (Requirements). Wenn nun dieses Pflichtenheft bereits auf Basis von UML erstellt wurde und in dem Repository (Zentrale Datenbank des CASE Tools) existiert, dann können geeignete Elemente daraus direkt zur Dokumentation von Design oder Implementation genutzt werden. Dieses geschieht natürlich nicht in Form von Copy & Past und schafft damit redundante Informationen, sondern mit einem direkten Link. Das hat noch weitere Vorteile. Werden z.B. vom Auftraggeber Änderungen gewünscht, können diese auf Basis des Pflichtenheftes durchgeführt werden. Auf Grund der Links ist sofort ersichtlich wo im Design sich evtl. Änderungen ergeben. Das hilft Pflichtenheft, Design, Code und Dokumentation in sich konsistent zu halten.

Aus diesem Vorgehen ergibt sich gleich noch die Lösung eines weiteren, häufig anzutreffenden Problems. Bei manchen unserer Kunden gibt es separate Testabteilungen.

Zu einem bestimmten Zeitpunkt des Projektes bekommen diese dann einen Prototypen incl. C-Source Code und dem ursprünglichen Pflichtenheft, um mit den Tests zu starten.

Wenn diese Aufgabe gewissenhaft durchgeführt wird, werden Zustände erkannt, in denen der Prototype nicht so reagiert wie im Pflichtenheft beschrieben. In vielen Fällen beginnt jetzt erst der aufwändigste Teil der Arbeit, nämlich nachträglich herauszufinden, ob das eine geänderte Eigenschaft des Produktes ist und das Pflichtenheft an dieser Stelle veraltet ist oder ein tatsächliches Fehlverhalten.

Hier ergibt sich ein weiterer Nutzen aus der UML. Sie besitzt Diagrammtypen, mit denen sich das prinzipielle Verhalten in Form von Abläufen beschreiben lässt. In vielen Fällen können auf Basis dieser Diagramme bereits festgelegte Abläufe im Pflichtenheft (Requirements Engineering) beschrieben werden. Zu diesen Diagrammen können dann im Verlauf des Designs die Interfaces spezifiziert werden, das ermöglicht dann diese Diagramme direkt als Testfälle wiederzuverwenden. Da wir ja mit der UML eine Formale Notation verwendet haben ist diese auch für Testtools verständlich. Die beste Voraussetzung Testabläufe zu automatisieren und das sogar auf Basis der aktuellen Requirements. *(Das ist übrigens eine der Forderungen in der Norm 61508 oder SPICE)*

Natürlich ermöglicht die UML auch eine sehr viel effizientere Wiederverwendung von implementierten Modellen und Code.

## Verstehbarkeit

Natürlich und offensichtlich erhöht die UML als Grafische Notation die Verstehbarkeit. Ein Bild sagt mehr als tausend Worte, aber was genau ist mit Verstehbarkeit denn eigentlich gemeint? Z.B., dass Sie besser verstehen was Ihr Kollege erschaffen hat und umgekehrt. Aber es geht noch weiter. Wie zum Beispiel kommunizieren Sie derzeit mit Kollegen aus anderen Ingenieurs-Disziplinen? Wahrscheinlich auf Basis von Handskizzen auf Flipcharts. Die UML ist eine Notation, die in großen Bereichen auch von NICHT Softwareentwicklern verstanden wird. Auf dieser Basis kann immer am aktuellen Stand des Modells kommuniziert werden.

Verstehbarkeit bedeutet auch, dass schnell ein tiefes Verständnis der Software erreicht wird. Dieses liegt sehr häufig im Bereich des so genannten Architektur-Designs. Was passiert an anderen Stellen meines Systems, wenn an dieser Stelle diese Änderung durchgeführt wird. Wie sind die Zusammenhänge? In C gibt es nur sehr wenig bis gar keine Konstrukte, die die Architektur einer Software beschreiben. Jegliches Wissen über die Architektur existiert oft nur in den Köpfen der Entwickler. Aus meiner persönlichen Sicht ist eine der grössten Vorteile der UML gegenüber herkömmlichen Hochsprachen, dass sie Konstrukte zur Modellierung der Laufzeit und Kommunikationsarchitektur besitzt und vor allem das Design von Interfaces unterstützt.

Es ließen sich noch sehr viel weitere Anforderungen aufführen, die aber den Rahmen dieses Artikels sprengen würden.

## Anforderungen an eine UML-Umgebung

Vielleicht haben Sie es erkannt. Viele Vorteile der UML ergeben sich durch den Einsatz von Informationen über den ganzen Entwicklungsprozess vom Requirements Engineering bis zum Test und automatisierten Arbeitsschritten auf Basis dieser Informationen. Die UML wird nicht umsonst als Modellierungs-Sprache bezeichnet. Sie besitzt gegenüber Hochsprachen einen wesentlich erweiterten Notationsumfang. Angefangen von Elementen zum Requirements Engineering über Elemente zur Modellierung der Architektur bis hin zu Elementen zur Modellierung von Ablaufverhalten. Das ist eine elementare Voraussetzung um Arbeitsschritte zu automatisieren.

Vor allem aber auch Mechanismen, die als Interface zwischen den verschiedenen Schichten, Arbeitsschritten oder Phasen dienen.

Im Zusammenspiel all dieser Elemente sind die meisten heutigen Probleme zu suchen. Nachdem auf Basis von Überstunden eine Erweiterung gerade noch halbwegs termingerecht zum Laufen gebracht werden konnte, nun die Möglichkeit zu haben, mit einem einzigen Knopfdruck dem Inbetriebnahme-Ingenieur noch die aktuelle Dokumentation mitzugeben, dass ist es, was die Effizienz steigert. Und Ihr Gewissen beruhigte es noch zusätzlich, weil Sie etwas Vollständiges abgeliefert haben.

Dazu benötigt es neben der Notation und Methoden eine Tool-Umgebung, die es Ihnen ermöglicht genau dort zu arbeiten, wo Sie gerade am effizientesten sind und womit Ihre Änderungen in alle anderen Ebenen automatisch überführt werden können. Manchmal ist es effizienter direkt im generierten Code zu ändern manchmal auf Modellebene, manchmal in der Dokumentation, manchmal in den Requirements. Die grosse Frage ist wie werden die Änderungen mit dem geringsten Aufwand zueinander konsistent gehalten. Dabei kommen Funktionalitäten wie Round-Tripping, oder Reverse-Engineering eine zentrale Rolle.

Als Abschluss ein paar Eigenschaften die ein Tool erfüllen sollte um wirklichen Produktivitätsgewinn zu erzielen:

- Grundsätzlich sollte ein Tool auf Basis eines so genannten „Repository“ arbeiten. Es ist die Grundvoraussetzung um Modellelemente ohne redundante Informationen wiederverwenden zu können
- Das Tool sollte so genannte Produktion-Code-Generierung unterstützen. Im Idealfall kann direkt aus dem Modell vollständiger Code generiert werden, der auf einer spezifischen Zielplattform ausführbar ist. Wenn das nicht der Fall ist und Sie parallel zum Model auch Codeelemente von Hand pflegen müssen wird es schnell ineffizient.
- Der generierte Code sollte gut verstehbar sein, denn häufig ist es notwendig auch darin zu arbeiten. Z.B. beim Debugging der Software auf der realen Hardware.
- Das Tool sollte Reverse-Engineering ermöglichen, denn mit großer Wahrscheinlichkeit haben Sie alte Sources, die Sie weiter verwenden möchten, oder zugekaufte Bibliotheken, die Sie ev. im Source verwalten möchten.
- Das Tool sollte die Möglichkeit besitzen jegliche Modellelementen (Requirements, Code, Dokumentation) miteinander zu verlinken.

Natürlich sollten Bedienbarkeit, Robustheit und viele Detailfeature ebenso vorhanden sein. All das aufzuführen sprengt den Rahmen dieses Artikels.

## Bewertung des Nutzen

Bereits Albert Einstein sagte einmal: "Probleme kann man niemals mit derselben Denkweise lösen, durch die sie entstanden sind."

Wenn wir aus unseren Erfahrungen in der Programmierung in ANSI-C nach einer Lösung unserer aktuellen Probleme suchen, dann werden wir mit großer Wahrscheinlichkeit nur die Symptome finden und behandeln, die aus dieser Vorgehensweise resultieren. Die große Frage ist, ob uns das langfristig weiter hilft.

Der Ansatz, sich an den hinter den Symptomen steckenden, übergeordneten Anforderungen zu orientieren beinhaltet ein um Potenzen größeres Potential zur Optimierung unseres Handelns und Schaffens. Dieses Vorgehen führt zum Paradigmenwechsel, hin zu UML, OOP und Modellierungs- Werkzeugen.

Wird also der Nutzen der UML gepaart mit OOP und geeigneter Toolunterstützung orientiert an den aktuellen Tagesproblemen betrachtet, dann scheint der Aufwand gegenüber dem Nutzen klein zu sein.

Betrachten wir jedoch den Nutzen in den darüber liegenden Ebenen, z.B. im globalen Entwicklungsprozess, dann ist er im Vergleich zu den Investitionen immens groß.

In der Praxis ist eine Effizienzsteigerung von 20% bis 30% in einigen Monaten zu erreichen. Die Kosten für alles was dazu notwendig ist (Schulung, Tools, entfallene Arbeitszeit für Einarbeitung ...) liegen bei etwa 30.000,- € pro Arbeitsplatz.

Werden diesen einmaligen Investitionen die Personalkosten gegenüber gestellt, dann ergibt sich eine Amortisation nach spätestens einem Jahr. In den Folgejahren bleibt der gewonnene Effizienzgewinn natürlich erhalten und kann entsprechend eingesetzt werden z.B. zur Verkürzung der Time-To-Market Zeit, oder zur Verbesserung der Qualität oder um den Standort Deutschland zu Hochleistung zu verhelfen. Hervorragende Ingenieure gibt es, geben wir ihnen mit ANSI-C auch das richtige Handwerkzeug zum Design der Applikationen?

#### Quellen:

„Wien wartet auf Dich!“ Der Faktor Mensch im DV-Management - Tom DeMarco / Timothy Lister

Systems Engineering mit SysML/UML - Tim Weilkiens

Willert Kundenumfrage 2004: Produktivitätssteigerung durch UML

Willert Kundenumfragen 1995/2000/2003/2007: Komplexitätsentwicklung und deren Folgen

„Artikel - Software im allgemeinen Wertewandel“ - Andreas Willert



#### **Textbox: Auswirkungen einer überalterten Softwarearchitektur.**

Änderungen, Weiterentwicklung, steigende Komplexität ... wirken sich über die Zeit in der Architektur von Software aus. Verstehbarkeit, Änderbarkeit und Qualität leiden. Ein eindeutiges Indiz in der Praxis des Software Engineering ist die Verschiebung des Aufwands für Implementation und Test. Passen Software Engineering Methode nicht zur Komplexität der Applikation, dann steigt der Aufwand für Test und Debugging im Verhältnis zum Aufwand für Codierung und Implementation wenn die Qualität stabil gehalten werden soll. Eine Umfrage unter den Kunden der Firma Willert Software Tools über die letzten 10 Jahre hat ergeben, dass sich das durchschnittliche Verhältnis kontinuierlich zu ungunsten der Implementations Zeiten verschoben hat. Ein eindeutiges Zeichen dafür, dass ein Paradigmenwechsel längst überfällig geworden ist.

Mit dem Einsatz einer neuen Vorgehensweise auf Basis neuer Werkzeuge kann dieses Verhältnis wieder in einen gesunden Bereich von 60% Implementation und 40% Test gebracht werden.

Quelle: Kundenbefragung 1995 bis 2007 Willert Software Tools GmbH