2. Zur Kluft zwischen Spezifikation und Implementation

Der Schlüssel zum Zusammenhalt eines komplexen Softwareproduktes bestehend aus unterschiedlichen Beschreibungen ist ein zentraler Begriffskatalog bzw. eine allumfassende Ontologie, in dem alle relevanten Begriffe einheitlich definiert sind. Dies bezieht sich nicht nur auf die Datenbegriffe sondern ebenso auf die Funktionsbegriffe und die Beziehungsbegriffe. Typische Datenbegriffe sind die Entitäten, Attribute, Tabellen, Objekte und Zustände. Typische Funktionsbegriffe sind Anwendungsfälle, Aktionsschritte, Aktivitäten Methoden und Operationen, Beziehungsbegriffe sind die Bezeichnungen für Systemelemente in denen Funktionen und Daten vereint werden, z.B. Klassen, Schnittstellen und Testfälle. Solche Elemente beinhalten Daten und Funktionen.

Ein Problem ergibt sich aus dem Bruch zwischen den semantischen Ebenen. Die gleichen Daten und Funktionen heißen anders auf der einen semantischen Ebene als auf der anderer. Auf der konzeptionellen Ebene bzw. auf der Ebene des Datenmodells ist eine datengruppe, eine Entität und die einzelnen Datenelemente, Attribute. Auf der Implementierungsebene sind die Datengruppen, Tabellen oder Sätze und die Einzeldatenelemente sind Spalten oder Folder. Seitens der Funktionen ist eine Gruppe zusammenhängender Funktion, die zeitlich aufeinander folgen, auf der konzeptionellen Ebene ein Anwendungsfall. Die einzelnen Funktionen werden als Schritte bezeichnet. Auf der Implementierungsebene heißen die gleichen Anwendungsfälle in der prozeduralen Programmierung Programme oder Transaktionen und in der objektorientierten Programmierung heißen sie Sequenzen bzw. Aktivitätsfolgen. Die Zuordnung der konzeptionellen Begriffe zu den Implementationsbegriffen ist keineswegs eindeutig. Es besteht eine *n:m* Beziehung zwischen Entitäten und Tabellen, d.h. die Attribute einer Entität können auf mehrere Tabellen verteilt werden, oder mehrere Entitäten können in einer Tabelle zusammengefasst werden. Idealerweise wäre eine Datenentität einer Tabelle bzw. einer Datei zugeordnet. Auf der Funktionsseite besteht ebenfalls eine *m:n* Beziehung zwischen Anwendungsfälle und Klassen. Ein Anwendungsfall benutzt *n* Klassen und eine Klasse wird von *m* Anwendungsfällen benutzt. Die Methoden einer Klasse sind gleichfalls die Knoten in einem Datenflussgraph bzw. ein einem Aktivitätsdiagramm und sie sind Zustandsübergänge in einem Zustandsdiagramm.

In der Objekttechnologie sind die Elementarfunktionen Methoden, die Klassen zugeordnet werden. In der prozeduralen Technologie sind die Elementarfunktionen Prozeduren, die Moduln zugeordnet werden. In prozeduralen Sprachen nehmen die Prozeduren unterschiedliche Formen an. In C sind sie Funktionen, on COBOL Paragraphen, in PL/I Prozeduren oder DO-Blöcke, in Natural und Fortran Subroutinen. Es gibt keine einheitliche Nomenklatur. Umso wichtiger ist es, dass die Namen der Funktionen etwas über den Inhalt aussagen. Die Entwickler hat die Möglichkeit die Bedeutung der Funktionen in Kommentaren zu beschreiben, dies ist aber nicht zwingend und dafür gibt es keiner normierten Syntax wie in JavaDoc. Jeder kann schreiben was er will. Auf Projektebene wäre es möglich einheitliche Kommentarformate durchzusetzen. Z.B. könnte der Kommentar auf die Anforderung

verweisen die diese Code-Funktion erfüllt, bzw. auf den Anwendungsfallschritt der hiermit implementiert wird. Der Projektleiter oder Produktbesitzer könnte dies durchsetzen, es wird aber selten gemacht. Also bleibt die Funktionsbezeichnung der einzige Anhaltspunkt

Es geht darum, den Bruch zwischen dem Code und der spezifizierten Anwendungsfällen zu überwinden. Wenn die Kommentare im Code fehlen, bleiben nur die Namen der Funktionen und Daten übrig um eine Verbindung herzustellen. Wenn die Namen im Code auch keine Ähnlichkeit mit den Begriffen der Anforderungsdokumentation haben, dann bleibt die Kluft zwischen der Spezifikation und der Implementation offen und die beiden Produktbeschreibungen werden immer mehr auseinander driften. Das Resultat wird sein, dass die Anforderungsspezifikation nicht mehr fortgeschrieben wird und die Anwender sich bei ihren Änderungsanträgen nicht länger darauf beziehen können.

Das Gleiche trifft für das Entwurfsmodell zu. Die UML Sprache bietet eine ganze Reihe Diagrammtypen an, um die Struktur einer Applikation darzustellen, Klassendiagramme beschreiben, die Struktur der Daten und Aktivitätendiagramme beschreiben, die Struktur der Funktionen, Sequenzdiagranne beschreiben, die Interaktionen zwischen Klassen und Zustandsdiagranne beschreiben – wie Objekte sich verändern. Die graphischen Darstellungen können sehr nützlich sein und sind es auch wenn es darum geht, die Rolle einer Funktion oder den Standort eines Datums herauszufinden. Sie können ebenfalls dazu dienen, die Auswirkung einer Änderung auf die Systemarchitektur zu erkennen. Sie sind jedoch nur so lange nützlich, als sie mit dem Code übereinstimmen. Wenn die Bezeichnungen in den Diagrammen nicht mit den Beziehungen im Code übereinstimmen, ist es nicht möglich die Entwurfselemente den Codebausteinen zuzuordnen. In diesen Fall ist das Entwurfsmodell von dem Code abgekoppelt.

Das dies passiert liegt daran, wie bei den Anforderungen, dass andere Menschen am Werk sind und jeder verwendet seine eigene Begriffe. Sogar in verschiedenen Diagrammtypen werden andere Begriffe für die gleichen Elemente benutzt. Die Klassen heißen in den Sequenzdiagrammen anders als in den Klassendiagrammen. Die meisten UML-Werkzeuge lassen dies leider zu. Die Folge ist, dass das UML Modell auch nicht in sich, geschweige denn mit dem Code Konsistent ist.

Falls der Code aus dem Entwurfsmodell generiert wird, werden Code und Entwurf miteinander übereinstimmen, aber nur so lange, als der Code nicht geändert wird. In dem Moment, wo Änderungen am Code vorgenommen werden, beginnt der Code sich von dem Entwurf abzusetzen und der Zusammenhang geht verloren. Um dies zu vermeiden, muss der Code bei jeder Änderung neu generiert werden. Das würde bedeuten, dass alles, was im Code steckt, auch in der UML Darstellung vorkommt. Dann wird UML zur Implementierungssprache. Das war mal der Traum von den UML-Vätern – „UML all the way from top to bottom“ aber dieser Traum hat sich bisher nicht verwirklicht. In der Praxis sind die UML Modelle selten vollendet und wenn ja, stimmen sie nur kurz mit dem Code überein. Das heißt, auch hier entsteht eine Kluft zwischen Spezifikation und Implementation.

3. Zur Kluft zwischen Implementation und Test

Der Test eines Softwaresystems stützt sich auf den Testfällen, die in der Testspezifikation festgehalten sind. Es ist Aufgabe der Tester Testfälle zu ermitteln und zu dokumentieren. Aus den Testfällen werden Testdaten generiert und an Hand der Testfälle werden Testergebnisse validiert. Die Testfälle beschreiben die Vor- und Nachzustände der Anwendungsfälle sowie die Reihenfolge der prozeduralen Schritte. Sie können in Tabellen, Texten oder Testskripten festgeschrieben sein. In den Testfällen wird auf die Schritte in den Anwendungsfällen, auf die Datenobjekte und auf die Schnittstellen Bezug genommen. Es ist unabdingbar, dass die Testfälle die real existierenden Datenstrukturen widerspiegeln. Wenn sie dies nicht tun, können sie ihren Zweck nicht erfüllen.

Ein Testfall muss einem bestimmten Anwendungsfall zugeordnet sein. Falls ein Anwendungsfall oder der dazu gehörende Code geändert wird, müssen die Testfälle für diesen Anwendungsfall in dem Regressionstest wieder ausgeführt werden. Sofern die Testfälle die Namen der Anwendungsfälle die sie testen und die Datenobjekte die sie generieren und validieren beinhalten, sind sie zuordnungsbar. Wenn aber die Be----------der Testfälle mit denen der Anforderungsspezifikation und des Datenmodells nicht oder nicht mehr übereinstimmen, sind die Testfälle nicht brauchbar. Sie sind von dem System, das sie testen sollten, abgehängt. Deshalb die Notwendigkeit deren Konsistenz sowohl mit der Dokumentation als auch mit dem Code.

Natürlich wird der Test zeigen, ob die Testfälle mit dem Code übereinstimmen. Wenn nicht, werden Abweichungen zwischen den Ist- und den Soll-Ergebnissen gemeldet. Dies ist letztendlich der Zweck des Testens.

Wer sichert aber, ob der Test die wahren Erwartungen der Anwender wiederspiegeln? Möglicherweise haben die Tester, die die Testfälle spezifizieren, die Anforderungen nicht verstanden oder falsch interpretiert. Solange als die Testfälle nicht automatisch aus der Anforderungsbeschreibung generiert werden – d.h. solange als die Testfälle und Testskripte noch manuell erstellt werden – werden sie von dem spezifizierten Verhalten der Software abweisen. Das ist ein Argument dafür, dass diejenigen, die die Anforderungen spezifizieren bzw. die Stories schreiben, auch die Testfälle spezifizieren sollen. Dafür sind sie jedoch nicht ausgebildet und wären sie es, dann hätten sie keine Zeit. Es dreht sich immer um die gleiche Frage, wie teilt man die Arbeit auf? Idealerweise gebe es nur eine Person, die alles macht - die Anforderungen spezifiziert, die Architektur entwirft, den Code schriebt und das System testet. Aber auch in diesem Fall wäre es nicht sicher, ob die Gedanken dieser Person bei der Festlegung der Anforderungen mit den Gedanken derselben Person bei der Konzeption des Tests übereinstimmen. Menschen ändern bekanntlich ihre Meinung und ihre Ansichten auf Probleme. Wenn sie die Anforderungen beschreiben, betrachten sie das System von der Ferne, sie erkennen nur grobe Konturen. Wenn sie Testfälle beschreiben, sind sie gezwungen das System von nächster Nähe zu betrachten. Es ist der Unterschied, ob jemand einen Berg von weiter Ferne betrachte, um seine Route zu erkennen oder ob jemand den Fels vor sich mustert um seinen nächsten Tritt auszuwählen. Die beiden Betrachtungen erfordern eine andere Sichtweise. Demzufolge können die Sichten nie ganz stimmig sein, auch wenn es sich um

eine und dieselbe Person handelt. Wenn auch noch mehrere Personen beteiligt sind, ist die Kluft zwischen den verschiedenen Sichten noch größer, vor allem wenn sie unterschiedliche Begriffe verwenden. Die Sicht des Testers auf ein bestimmtes Ereignis ist eine andere als die des Analytikers und des Entwicklers. Wenn die Beteiligten ihre Schichten dokumentieren, ist es sehr unwahrscheinlich, dass diese Dokumente Konsistent sind.

Aus der Inkonsistenz der Dokumente – sprich Sichten – entstammt ein Großteil der Softwarefehler. Wenn die Testfälle nicht den wahren Erwartungen des Anwenders entsprechen, kann der Test das erwartete verhalten eines Systems nicht nachweisen. dann ist der Test unvollständig, wenn nicht irreführend. Demzufolge muss die Konsistenz des Tests mit den Anforderungen geprüft werden.

4. Zur Sicherung der Produktkonsistenz

In den vorhergehenden Abschnitten wurde auch die Bedeutung der Konsistenz der Produktbeschreibungen, des Codes und des Tests hingewiesen. Für eine effektive und qualitätsbewahrende Fortschreibung des Produktes ist die Konsistenz der Beschreibungen unentbehrlich. Es ist nochmals zu betonen, dass der Verfall eines Softwareproduktes mit dem Verlust der „Traceability“ bzw. mit der Zuordnungsfähigkeit beginnt. Deshalb gibt es dies zu verhindern. Verhindern lässt sich die Zunahme der Inkonsistenz nur durch regelmäßige Kontrollen. Die Software-Artefakte müssen immer wieder auf ihre Konsistenz geprüft werden, zumindest nach jedem neuen Release.

Es gibt dazu zwei verschiedene Prüfarten. Die eine ist die Konsistenzprüfung innerhalb einer semantischen Ebene. Z.B. wird in der Anforderungsdokumentation geprüft, ob alle Datenentitäten auf die verwiesen wird, irgendwo definiert sind. Ferner wird geprüft, ob alle Anforderungen von einem Anwendungsfall erfüllt werden und dass alle Regel in einem Anwendungsfall implementiert sind. Auf der Entwurfsebene wird geprüft, ob die Klassen in einen Sequenzdiagramm und alle Methoden in einem Aktivitätsdiagramm vorkommen. Objekte in einem Zustandsdiagramm müssen in das Objektdiagramm definiert werden und Aktivitäten in einem Aktivitätsdiagramm müssen als Methoden in den Klassendiagramm erscheinen. Ein UML Modell ist nur wirklich brauchbar, wenn sie in sich konsistent ist.

Der Code muss in sich konsistent sein, sonst ist er nicht kompilierbar. Der Compiler sichert die Konsistenz des Codes. Es gibt aber nicht inne reib Compiler für die Testfälle. Testfälle werden nur kompiliert, wenn sie in einer formalen Skriptsprache beschrieben sind. Wenn sie in Tabellen oder Prosatexten verfasst sind, können sie nicht ohne weiteres geprüft werden. Es bedarf ein besonderer Testfallanalysator um ihre Konsistenz zu kontrollieren, z.B. ob die Daten, die sie als Eingabe zu einen Anwendungsfallschritt vorschreiben, ob den Benutzeroberflächendefinitionen wirklich vorkommen und ob die Ausgaben die sie validieren wollen, in den Datenbankschema definiert sind. Natürlich müssen die Namen der Daten übereinstimmen. Dies trifft für alle Systemelemente zu, auch für die Funktionsbezeichnungen. Es muss möglich sein, Testfälle über ihre Funktionsbezeichner einen ganz bestimmten

Anwendungsfall und sogar zu einen bestimmte Schritt – Elementarfunktion in jedem Anwendungsfall zuzuordnen.

Die andere Prüfungsart ist die Prüfung der Konsistenz zwischen semantischen Ebenen. Es gilt zu sichern, dass, was auf einer semantischen Ebenen behauptet wird – z.B. in dem Code – mit dem, was in der darüber liegenden semantischen Ebene sowie mit dem, was in der darunterliegenden semantischen Ebene übereinstimmt. Voraussetzung für diese Prüfung ist die Verwendung gleicher Begriffe bzw. fast gleiche Begriffe. Die Handlungen auf der Codeebene werden viel detaillierter beschrieben als die Handlungen auf der Entwurfsebene aber die Gegenstände der Handlungen sollten gleich oder ähnlich heißen. Mit „ähnlich“ ist gemeint, dass die Bezeichner der Objekte einen gleichen Wortstamm haben. Es können unterschiedliche Präfixen und Suffixen dazu kommen, aber der Wortstamm muss erhalten bleiben, z.B. in der Variabelnamen

Job_Kundenname_ein

Job ist der Präfix und ein der Suffix aber der Wortstamm „Kundenname“ bleibt. Die Bezeichnung

Lfd_Nummer

beschreibt eine Zahl, die in der Spezifikation als Laufende_Nummer bezeichnet wird. Der Wortstamm „Nummer“ verbindet die beiden Begriffe.

In dem Versuch, Code-Bausteine mit Anwendungsfällen in der Spezifikation nachträglich zu verbinden, werden die Objektnamen verglichen. Falls eine Methode oder eine Prozedur im Code viele des gleichen oder ähnlichen Begriffs verwendet wie der Anwendungsfall, wird angenommen, dass die Methode bzw. Prozedur zu diesem Anwendungsfall gehört. Es müssen mindestens drei der Objektnamen von dem Anwendungsfall mit den Operanden im Code übereinstimmen, damit dieser Codebaustein dem Anwendungsfall zugeordnet wird. Das gleiche gilt für die Testfälle wobei hier nur zwei der Begriffe gleich oder ähnlich sein müssen.

Es versteht sich, dass auf diese recht einfache Vergleichsart viele Duplikate und viele falsche Treffer vorkommen. Der zuständige Tester muss die Zuordnungsmatrix genau anschauen und die Zuordnung überprüfen. Es besteht aber kaum eine andere Möglichkeit die Software Artefakten miteinander nachträglich zu verbinden. Idealerweise müssten die Verbindungen über Verweise vom Anfang an in der Software eingebaut werden. Wenn das aber versäumt wurde, bleibt wie die Literatur zur Reverse Engineering bestätigt, nur der Weg über den Namensvergleich [99].

6. Zusammenfassung und weitere Forschung

In diesem Beitrag wurde ein Ansatz zur Prüfung- und Wiederherstellung der Konsistenz eines Softwareproduktes geschildert. Das Ziel ist es, Software Artefakte auf verschiedenen semantischen Ebenen – Anforderungen, Entwurf, Code und Test – miteinander zu verbinden.

Die Methode dies zu erreichen ist über den Abgleich der verwendeten Begriffe. Das Ergebnis ist eine Zuordnungsmatrix der Anforderungen, Anwendungsfalle, Datenentitäten, UML-Diagramme, Codebausteine und Testfälle. Hinzu kommt ein Bericht über die Konsistenz der semantischen Ebenen – Anforderungen, Code und Test.

Die hier geschilderte Arbeit kann nur als erster Schritt auf den Wege zur Sicherung der Software Produktkonsistenz betrachtet werden. Die nächsten Schritte sind den Namensvergleich zu verfeinern und die Nutzung der Daten in Betracht zu ziehen. Damit könnte die Zuordnungsquote weiter erhöht werden. Jedenfalls bleibt noch viel zu tun um das Ziel zu erreichen, aber das Ziel rechtfertigt die Forschungsarbeit. Die Inkonsistenz der Produktbeschreibungen bleibt als Hauptursache schwerer Softwarefehler und als Hauptgrund für die hohe Wartungsaufwende.