von martin rösch

mehr zum thema:
www.sustainablecomputing.org
www.generalobjects.com/begriffe/uml

VERIFIKATION VON UML-MODELLEN: MOTIVATION, GRUNDLAGEN UND VERFAHREN

Größere Softwareprojekte werden heute zunehmend auf einem Fundament aus UML gebaut. Doch wie fest ist dieses Fundament? Wie können Softwareentwickler sicherstellen, dass ihre UML-Modelle richtig sind? Und wie können sie bei späteren Änderungen das Einschleichen versehentlicher Fehler verhindern? Der Artikel beschreibt die Verifikation von UML-Modellen anhand eines Verfahrens, das sich seit einigen Jahren in der Praxis bewährt hat.

der autor

Martin Rösch (E-Mail: martin.roesch@generalobjects.com) ist Managing Director von General Objects LTD und Leiter der Object Academy. Sein besonderes Interesse gilt dem bewussten Umgang mit menschlichem Wissen und dessen gesicherter Behandlung im Entwicklungsprozess von Softwaresystemen.

Die UML ist inzwischen erwachsen und eine professionelle Softwareentwicklung ohne sie ist kaum noch vorstellbar. Als weltweit standardisierte und von vielen Tools unterstützte Planungssprache für Informationssysteme hat die UML deshalb einen festen Platz im Entwicklungsprozess von modernen Softwareprojekten.

So ist es nur natürlich, dass immer mehr UML-Modelle entstehen und dass sie einen immer größeren und wichtigeren Teil der Projektergebnisse bilden. Um so wichtiger wird daher die Frage nach ihrer Richtigkeit. Doch was ist hier „richtig"?

Salopp gesprochen ist ein Produkt richtig, wenn es macht, was es soll. Dieser einfache Satz zeigt uns schon die ersten drei Voraussetzungen für das Feststellen der *Richtigkeit von UML-Modellen*:

1) Wir müssen Erwartungen bzw. Anforderungen das Produkt haben (beschreiben).
2) Wir müssen die tatsächlichen Eigenschaften des Produkts feststellen können (messen).
3) Wir müssen die erwarteten mit den tatsächlichen Eigenschaften vergleichen können (vergleichen).

Genau dasselbe – nur etwas formaler – definiert die Norm ISO 9000 (vgl. [ISO]) als den Kern von Qualität (**Kasten 1**).

Für das Qualitätsmanagement in der Softwareentwicklung ergeben sich hieraus einige Denkanstöße:

1) Erst durch Anforderungen wird Qualität definiert.
2) Erst durch die Verifikation wird Qualität messbar.
3) Was man nicht messen kann, das kann man auch nicht managen.

Die Verifikation von UML-Modellen könnte aus diesem Grund auch für das Qualitätsmanagement in der Softwareentwicklung zu einem interessanten Thema werden, wobei die Norm den dritten Punkt als „Verifikation" bezeichnet und eine nicht erfüllte Anforderung als „Fehler".

Zusätzlich gibt es noch drei weitere Begriffe, die oft im Zusammenhang mit Verifikation genannt werden und die leider geeignet sind, Verwirrung zu stiften, da ihre umgangssprachliche Bedeutung nur teilweise mit ihren ISO-9000-Definitionen übereinstimmt: Fehlerfreiheit (**Kasten 2**), Validierung (**Kasten 3**) und Kundenzufriedenheit (**Kasten 4**). Damit sind die Grundlagen für die Verifikation von UML-Modellen schon beisammen. Mehr ist nicht erforderlich. Fangen wir also an.

Anwendungsbeispiel

Als Anwendungsbeispiel soll uns eine sehr einfache Auftragsverwaltung dienen. In ihr gibt es Kunden mit Name und Kreditlimit sowie Aufträge mit Beschreibung,

- **Verifikation:** Bestätigung durch Bereitstellung eines *objektiven Nachweises*, dass *festgelegte Anforderungen* erfüllt worden sind.
- **Objektiver Nachweis:** Daten, die die Existenz oder Wahrheit von Etwas bestätigen.
- **Anforderung:** Erfordernis oder Erwartung, das oder die festgelegt, üblicherweise vorausgesetzt oder verpflichtend ist.
- **Festgelegte Anforderung:** Eine festgelegte Anforderung ist eine *Anforderung*, die beispielweise in einem *Dokument* angegeben ist.
- **Dokument:** *Information* und ihr Trägermedium.
- **Information:** Daten mit Bedeutung.
- **Konformität:** Erfüllung einer *Anforderung*.
- **Fehler:** Nichterfüllung einer *Anforderung*.

Kasten 1: Definitionen aus ISO 9000:2000 zur Verifikation

Verifikation & Fehlerfreiheit

Nach den Buchstaben der deutschen Version der ISO 9000 könnte man ein verifiziertes UML-Modell eigentlich getrost und ohne rot zu werden als „fehlerfrei" bezeichnen, doch damit habe ich in der Vergangenheit schlechte Erfahrungen gemacht: Zu heftig sind die Emotionen, die an diesem Wörtchen hängen. Dabei ist Fehlerfreiheit gerade bei Software ein hochinteressantes Thema, zu dem es viel zu sagen gäbe. Doch das würde den Rahmen dieses Artikel sprengen.
Außerdem sind sowohl die englische als auch die französische Version des Normentextes deutlich vorsichtiger und nennen eine nicht erfüllte Anforderung lediglich eine „nonconformity" bzw. eine „non-conformité".
Doch bevor wir beginnen, mit Wort-Ungetümen wie „Nichtkonformität" (für Fehler) oder „nicht unkonform" (für fehlerfrei) zu hantieren, bleibe ich lieber bei „verifiziert" – da weiß man, was man hat, und es regt sich auch niemand darüber auf.

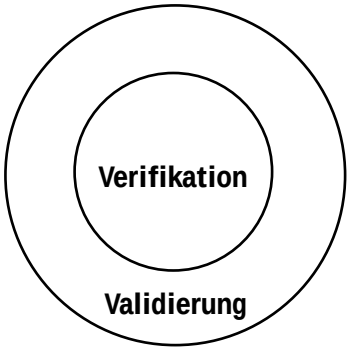
Kasten 2

Wert und Bezahlt-Status. Nehmen wir ferner an, dass die Anforderungssteller folgende *Anforderungen* haben:

1) Das System soll Kunden und Aufträge verwalten können. Ein Kunde kann *0* bis *n* Aufträge haben und jeder Auftrag soll zu genau einem Kunden gehören.
2) Für jeden Kunden sollen sein Name sowie sein Kreditlimit gespeichert werden.
3) Für jeden Auftrag sollen folgende Angaben gespeichert werden: seine Beschreibung, sein Wert und die Tatsache, ob er bereits bezahlt ist.
4) Das Kreditlimit eines Kunden beschränkt den Umfang seiner unbezahlten Aufträge. Neue Aufträge müssen in das freie Kreditlimit eines Kunden passen. Dies wird nur bei der Annahme neuer Aufträge geprüft.
5) Eine Verringerung des Kreditlimits hat keine Auswirkungen auf bestehende Aufträge.
6) Das freie Kreditlimit eines Kunden verringert sich bei jeder Erteilung eines Auftrags. Bei der Bezahlung eines Auftrags erhöht es sich wieder.
7) Wenn ein Kunde keine bzw. keine unbezahlten Aufträge hat, ist sein freies Kreditlimit identisch mit seinem eingetragenen Kreditlimit.

Verifikation & Validierung

Eine Validierung prüft das, was der Volksmund als „Qualität" bezeichnet.
Sie unterscheidet sich von der Verifikation in zwei wichtigen Punkten: Zum einen muss ein Produkt für eine Validierung auch „üblicherweise vorausgesetzte" Anforderungen erfüllen, ohne dass diese schriftlich festgelegt sein müssten. Zum anderen muss es auch „für einen beabsichtigten Gebrauch geeignet" sein. Sogar nicht genannte und daher auch gar nicht erfüllbare Erwartungen reichen also aus, um bei der Validierung einen Mangel (*Defect*) zu konstatieren.

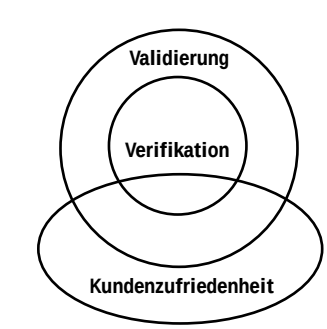


Deshalb warnt die Norm ganz ausdrücklich (und zum Teil sogar fett gedruckt) vor dem Begriff „Mangel" und damit indirekt auch vor der Validierung: „Die Unterscheidung zwischen den Benennungen Mangel und Fehler ist wegen ihrer rechtlichen Bedeutung, insbesondere in Fragen der Produkthaftung, wichtig. Die Benennung „Mangel" sollte daher mit äußerster Vorsicht verwendet werden" (O-Ton ISO 9000, Seite 26).

Kasten 3

Verifikation & Kundenzufriedenheit

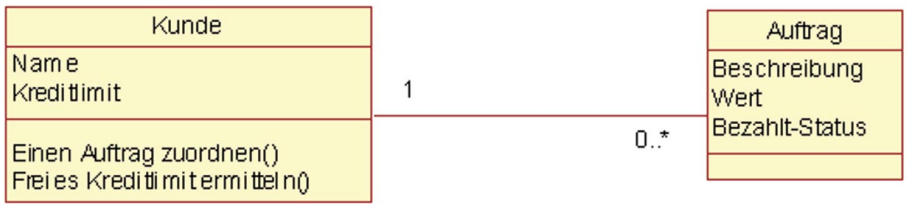
Zusätzlich zu Verifikation und Validierung, die beide durch Messungen und „objektive Nachweise" bestätigt werden, führt ISO 9000 den Begriff der Kundenzufriedenheit ein, als „Wahrnehmung des Kunden zu dem Grad, in dem



die Anforderungen des Kunden erfüllt sind". Sie wird rein subjektiv ermittelt.
Deshalb erläutert die ISO 9000 in einer Anmerkung zu dieser Definition: „Selbst, wenn Kundenanforderungen mit dem Kunden vereinbart und erfüllt worden sind, bedeutet dies nicht notwendigerweise, dass die Kundenzufriedenheit damit sichergestellt ist" ([ISO], S. 19).
Umgekehrt kann ein Produkt auch dann hohe Werte bei der Kundenzufriedenheit erreichen, wenn es weder verifiziert noch validiert ist. Viele populäre Programme belegen dies.

Kasten 4

Hierzu wurde das in **Abbildung 1** gezeigte UML-Modell erstellt, mit dem wir jetzt eine Beschreibung der späteren Software vorliegen haben. Doch wie können wir messen, ob es alle gestellten Anforderungen erfüllt? Hier kommt uns die ISO 9000 zur Hilfe, denn sie verlangt, dass die Erfüllung der Anforderungen „objektiv nachgewiesen" werden muss, z. B. durch Tests. Dies geht natürlich am besten, wenn man sich von Vornherein darauf einigt, wie die Erfüllung der Anforderungen getestet werden soll. Das erspart langwierige Missverständnis-Diskussionen im Nachhinein.



Einen Auftrag zuordnen: Ein Auftrag kann einem Kunden zugeordnet werden, wenn der Auftragswert nicht größer ist als das freie Kreditlimit des Kunden.
Freies Kreditlimit ermitteln: Vom Kreditlimit des Kunden werden die Werte aller seiner bislang unbezahlten Aufträge subtrahiert. Das Ergebnis ist das freie Kreditlimit.

Abb. 1: UML-Modell für das Kunde-Auftrag-Beispiel

In dem Verfahren, das wir bisher zur Verifikation von UML-Modellen benutzt haben[1]), hat jede Anforderung ihre eigenen Tests (meist zwischen drei und fünf); sie gilt als erfüllt, wenn sie alle ihre vereinbarten Tests bestanden hat.

Diese direkte und unmittelbare Zuordnung von Anforderungen und Tests geht über das hinaus, was ISO 9000 fordert, doch die Erfahrung hat gezeigt, dass sie die Handhabbarkeit der Anforderungssammlung deutlich verbessert – besonders dann, wenn Anforderungen geändert werden müssen. Dann helfen die zu einer Anforderung gehörenden Tests, sie schneller und sicherer zu verstehen. Und wenn eine Anforderung geändert oder entfernt werden muss, grenzt diese Zuordnung die Menge der potenziell betroffenen Tests sehr wirksam ein.

Für die folgenden Beispiele wählen wir die Anforderung 4, die in **Kasten 5** zusammen mit ihren Tests gezeigt wird.

In Trainings vermitteln wir die Verifikation von Mini-UML-Modellen durch Rollenspiele: Beim Nachspielen von Objekt-Interaktionen wie den in **Kasten 5** genannten Tests 4a bis 4c lernen Menschen sehr schnell, warum und wie die Verifikation funktioniert. Doch für den Praxis-Einsatz ist dieses nicht-automatisierte Verfahren natürlich zu langsam und daher letztlich nicht bezahlbar.

[1]) *Objects 9000 ist ein Verfahren für die Verifikation von UML-Modellen. Es ist selbst auch nach ISO 9000 zertifiziert.*

Anforderung 4 mit Tests

Das Kreditlimit eines Kunden beschränkt den Umfang seiner unbezahlten Aufträge. Neue Aufträge müssen in das freie Kreditlimit eines Kunden passen. Dies wird nur bei der Annahme neuer Aufträge geprüft.

- *a. (Normal):* Kunde Meier hat ein Kreditlimit von 500 Euro und zwei offene Aufträge *a1* und *a2* von 100 und 150 Euro. Ein dritter Auftrag von 200 Euro kann noch angenommen werden. Danach hat Kunde Meier drei Aufträge und einen freien Kreditrahmen von 50 Euro.
- *b. (Grenzfall positiv):* Kunde Meier hat ein Kreditlimit von 450 Euro und zwei offene Aufträge *a1* und *a2* von 100 und 150 Euro. Ein dritter Auftrag von 200 Euro kann (gerade noch) angenommen werden. Danach hat Kunde Meier drei Aufträge und einen freien Kreditrahmen von 0 Euro.
- *c. (Grenzfall negativ):* Kunde Meier hat ein Kreditlimit von 449 Euro und zwei offene Aufträge *a1* und *a2* von 100 und 150 Euro. Ein dritter Auftrag von 200 Euro kann nicht mehr angenommen werden. Dies wird durch eine Fehlermeldung signalisiert. Danach hat Kunde Meier zwei Aufträge und einen freien Kreditrahmen von 199 Euro.

Kasten 5

Praxistauglich wird die Verifikation von UML-Modellen deshalb erst durch ihre vollständige Automatisierung, wenn nach jeder Änderung an den Anforderungen oder am Modell eine erneute und zuverlässige Verifikationsaussage schon nach wenigen Minuten vorliegen kann.

Für die Automatisierung benötigen wir sowohl das UML-Modell als auch die Tests als ausführbaren Programmcode. Für die Beispiele in diesem Artikel wurde als Programmiersprache Java gewählt[2]). Eine einfache Codierung für das Modell zeigt der Codeausschnitt in **Listing 1**.

Wie man sieht, wurden die Angaben des UML-Modells gradlinig in Java umgesetzt. Dieser Vorgang kann z. B. mit Hilfe der Skriptsprachen der gängigen UML-Tools leicht automatisiert werden. Als nächstes müssen die Tests codiert werden. Das Java-Programm in **Listing 2** zeigt den Code für den Normalfall-Test[4]). Wenn man das Programm laufen lässt, erscheint nur eine kleine unscheinbare Anzeige:

```
Test_4a ok
```

Interessanter werden die Ausgaben erst, wenn wir absichtlich (oder unfreiwillig) ins Modell oder in die Tests Fehler einbauen. Dann bekommen wir auch Ausgaben wie z. B. die folgende zu sehen:

[2]) *In Projekten wurden bisher Smalltalk, Java, C++ und COBOL (mit CORBA-Schnittstellen) verwendet*

[4]) *Die Oberklasse „Test.java" sowie der vollständige Quellcode der Beispiele aus diesem Artikel können von* www.generalobjects.com/downloads/OBJEKTspektrum *herunter geladen werden.*

```
import java.util.Vector;

public class Kunde {

  public Kunde(String s) {
    super();
    name = s;
    limit = 0;
    auftraege = new Vector();
  }

  private String name;
  public void setName(String s) { name = s; }
  public String getName() { return name; }

  private int limit;
  public void setLimit(int i) { limit = i; }
  public int getLimit() { return limit; }

  Vector auftraege;
  public void addToAuftraege(Auftrag pAuftrag)
      throws MyException {
    if ( this.getFreiesLimit() < pAuftrag.getWert() ) {
      throw new MyException("Freies Limit "
        +this.getFreiesLimit()
        +" reicht nicht aus");
    }
    auftraege.add(pAuftrag);
  }

  public Auftrag[] getAuftraege() {
    return (Auftrag[])auftraege.toArray(new Auftrag[0]);
  }

  public int getFreiesLimit() {
    int freiesLimit = this.getLimit();
    Auftrag[] vorhandeneAuftraege =
      (Auftrag[])auftraege.toArray(new Auftrag[0]);
    for (int i=0; i<vorhandeneAuftraege.length; i++) {
      freiesLimit -= vorhandeneAuftraege[i].getWert();
    }
    return freiesLimit;
  }

}
```

Listing 1: Die Java-Klasse „Kunde.java" für die UML-Klasse „Kunde"

```
public class Test_4a extends Test {

  Kunde k1;
  Auftrag a1, a2, a3;

  public void buildSituation() throws MyException {
    k1 = new Kunde("Meier");
    k1.setLimit(500);
    a1 = new Auftrag(k1,"a1",100);
    a2 = new Auftrag(k1,"a2",150);
  }

  public void performAction() /* Exception wird
      nicht weitergereicht */ {
    try {
      a3 = new Auftrag(k1,"a3",200);
    } catch (MyException e) {
      fehlermeldung = e.getMessage();
    }
  }

  public void checkResult() throws MyException {
    // Kunde Meier muss 3 Aufträge haben
    if (k1.getAuftraege().length != 3) {
      reportNonconformity("Kunde hat nicht
          3 Aufträge, sondern "
        +k1.getAuftraege().length);
    }
    // Es darf keine Fehlermeldung aufgetreten sein
    if ( ! fehlermeldung.equals("") ) {
      reportNonconformity("Falsche Fehlermeldung:\n"
        +" Soll: \"\"\n"
        +" Ist : \""+fehlermeldung+"\"");
    }
    // Das freie Limit des Kunden Meier muss 50 sein
    if ( k1.getFreiesLimit() != 50 ) {
      reportNonconformity("Das freie Limit des Kunden ist "
        +"nicht 50, sondern "+k1.getFreiesLimit() );
    }
  }

  public static void main(String[] args) {
    Test_4a pb = new Test_4a();
    String pbName = pb.getClass().getName();
    if ( pb.verify() ) System.out.println(pbName+" ok");
    else System.err.println(pbName+" Verifikationsfehler");
  }
}
```

Listing 2: Die Java-Klasse „Test_4a" für den Test des Normalfalls

```
import java.util.Vector;

public class Anforderung {

  public Anforderung() {
    super();
    tests = new Vector();
  }

  Vector tests;
  public void addToTests(Test pTest) {
    tests.add(pTest);
  }
  public Test[] getTests() {
    return (Test[])tests.toArray(new Test[0]);
  }

  public boolean verify() {
    boolean status = true;
    Test[] alleTests = this.getTests();
    for (int i=0; i<alleTests.length && status; i++) {
      status &= alleTests[i].verify();
    }
    return status;
  }

  public static void main(String[] args) {
    Anforderung anf4 = new Anforderung();
    anf4.addToTests(new Test_4a());
    anf4.addToTests(new Test_4b());
    anf4.addToTests(new Test_4c());
    boolean b1 = anf4.verify();
    System.out.println( "Verifikationsstatus von
        Anforderung 4: "
      +(b1?"ok":"Fehler") );
  }

}
```

Listing 3: Das Java-Programm „Anforderung.java" zur Verifikation kompletter Anforderungen

```
Verifikationsfehler in Test_4a:
  "Das freie Limit des Kunden ist nicht 51, sondern 50"
Test_4a Verifikationsfehler
```

Hierbei handelt es sich um die Ausgabe von *Test_4a* nach Änderung des erwarteten Ergebnisses von *50* auf *51*: Das Modell meldet nach wie vor *50*, doch *Test_4a* erwartet (fälschlicherweise) *51* und meldet deshalb einen Verifikationsfehler.

Als nächstes verknüpfen wir die Anforderung 4 mit ihren Tests, damit diese nicht mehr einzeln gestartet werden müssen (**siehe Listing 3**). Die Ausgabe des Programms ist auch wieder wenig spektakulär, doch sie bedeutet, dass alle Tests der Anforderung 4 erfolgreich gelaufen sind, d.h. dass die Anforderung 4 erfolgreich verifiziert werden konnte:

```
Verifikationsstatus von Anforderung 4: ok
```

Nachdem im nächsten Schritt die Verifikation aller Anforderungen zusammen- ▶

gefasst wurde, informiert uns die folgende knappe Meldung über die Verifikation des gesamten Modells:

```
Verifikationsstatus von "Auftrags-System": ok
(7 Anforderungen, 26 Tests)
```

Das größte Modell, das auf ähnliche Weise in einem Kundenprojekt verifiziert wurde, bestand aus 20 Teilmodellen, die insgesamt 4.000 Anforderungen und 11.000 Tests besaßen.

Zusammenfassung

Wie man sieht, ist die Verifikation von UML-Modellen weder Hexenwerk noch Zauberei: In ihrem Kern beruht sie auf einfachen Testverfahren, von denen keines wirklich neu ist.

Die eigentliche Herausforderung in richtigen Projekten liegt deshalb auch weniger in der hier gezeigten Verifikationstechnik (sie wird seit 1994 nahezu unverändert benutzt), als vielmehr in der Organisation, Strukturierung und Verarbeitung der großen Anforderungsmenge, die in der Praxis entsteht.

Ausblick

Die Integration von UML-Werkzeugen mit Tools zur Anforderungsaufnahme ermöglicht es, die Kosten für die Verifikation von UML-Modellen deutlich zu senken: Während in früheren Projekten die Anforderungsaufnahme mit selbst erstellten und daher nicht ganz billigen Werkzeug erfolgen musste, steht jetzt durch die Verbindung von „Rational Rose" (UML-Modellierung) mit „RequisitePro" (Anforderungsaufnahme) erstmalig Standardsoftware zur Verfügung, die Anforderungen und UML-Modelle unter einem Dach verwalten kann.

Ein Folgeartikel wird deshalb anhand eines Beispiels zeigen, wie RequisitePro und Rational Rose zusammen mit geringem Anpassungsaufwand benutzt werden können, um UML-Modelle zu verifizieren. ■

Literatur & Links

[ISO] ISO 9000: „Qualitätsmanagementsysteme – Grundlagen und Begriffe - ISO 9000:2000", Ref-Nr. DIN EN ISO 9000:2000-12, Beuth-Verlag, Berlin (die internationale Norm für Qualitätsmanagementsysteme wurde im Jahr 2000 neu gefasst)