Daniel Speicher
(dsp@iai.uni-bonn.de)
forscht und unterrichtet an der Universität Bonn zur Softwareentwicklung mit einem Schwerpunkt auf Softwarequalität, wenn er nicht gerade die XP-Praktika, die er gemeinsam mit seinen Kollegen der Universität am B-IT in Bonn entwickelt hat, den Chinesischen Universitäten nahebringt.

Was meint Ihr Programm? Ein gelassenes Plädoyer für Verständlichkeit in der Programmierung

Eine Aufgabe der Architektur ist es, sicherzustellen, dass Programme verständlich und nachvollziehbar sind. Aber bis zu welcher Ebene ist das möglich? Meinen Sie, dass objektorientierte Programme die Wirklichkeit repräsentieren oder zumindest repräsentieren sollten? Vertreten Sie leidenschaftlich ein „Ja" oder ein „Nein"? Oder schauen Sie vielmehr mit Resignation oder einer gewissen „Herablassung" auf diese Frage? Wenn wir Programme als Mittel der Kommunikation zwischen Mensch und Maschine und vor allem zwischen Mensch und Mensch betrachten, können wir dieser Frage mit einer gewissen Gelassenheit begegnen. So ist für das gegenseitige Verständnis in der menschlichen Kommunikation erst an zweiter Stelle von Bedeutung, was die ausgetauschten „Zeichen" repräsentieren. An erster Stelle stehen die sich kontinuierlich entwickelnden Regeln ihrer Verwendung. Neben der Frage nach Verwendungsregeln ist die Frage nach der Repräsentation hilfreich, wenn man sich ihr pragmatisch nähert. Sofern wir uns auf die Lösungsdomäne beziehen, lässt sich zusätzlich Klarheit dadurch gewinnen, indem wir unterscheiden, ob wir unsere Software als Simulation oder Spiegel der Wirklichkeit verstehen oder aber als Fenster zur Wirklichkeit.

Bekanntermaßen haben Objekte in objektorientierten Programmen Identität, Zustand und Verhalten. Ganz so, wie die materiellen Dinge, mit denen wir alltäglich umgehen. Und es geht noch weiter: Objekte bestehen aus anderen Objekten, werden zu Klassen zusammengefasst und diese Klassen können in verschiedenen Beziehungen stehen. Auch diese drei Aspekte sind uns aus unserem alltäglichen intellektuellen Umgang mit der Welt vertraut. Aufgrund dieser Einheitlichkeit von Objekten und Dingen ist es in hohem Maße möglich, die gleiche Modellierungssprache sowohl für die Modellierung von Softwaresystemen als auch für die Modellierung der Wirklichkeit zu verwenden.

Da könnte man hoffen, dass Softwareentwicklung ein sehr flüssiger Prozess ohne Hindernisse sein könnte: Man modelliere einfach den relevanten Teilausschnitt der tatsächlichen (Geschäfts-)Wirklichkeit und der gewünschten Interaktionsmöglichkeit zwischen Mensch und Maschine in genügend präzisen Diagrammen und interpretiere diese Diagramme dann als Entwurf des zu entwickelnden Softwaresystems. Wäre das nicht schön?

Es wäre sogar nützlich: Denn dann wäre es leicht möglich, zu einer Änderung der Anforderungen die in unseren Diagrammen zu ändernden Stellen zu identifizieren und diese Änderungen wären zugleich die Vorlage für die geänderte Implementierung.

Repräsentieren Objekte die Realität?

Warum scheint uns diese Vorstellung zu schön, um wahr zu sein? Vertrauen wir nicht [mehr] der Magie des Versprechens, dass mit objektorientierter Programmierung (OOP) „alles ein Objekt ist"?

Die erste Sprache, in der sich wesentliche Gedanken der Objektorientierung fanden hieß "Simula" und verrät damit ihren Ursprung in der Simulation. Und solange unsere Software Simulation der Wirklichkeit ist, besteht tatsächlich kaum ein Grund, Schwierigkeiten zu erwarten. Für die Dinge der Wirklichkeit erzeugen wir Objekte, die diese Dinge repräsentieren und wenn ein Ding Einfluss auf ein anderes Ding hat, simulieren wir diesen Einfluss durch Methoden oder Ereignisse. Sollten die Einflüsse in der Wirklichkeit nicht übermäßig analog sein, besteht berechtigter Grund zu der Hoffnung, dass sich die Objekte in unserer Maschine genauso verhalten, wie die simulierte Wirklichkeit.

Ein solches Programm lässt sich leicht warten. Das liegt in erster Linie daran, dass sich jeder, der mit der simulierten Wirklichkeit vertraut ist, leicht im Quelltext des Programms orientieren kann. Die Namen der Konzepte, unter die wir die Dinge fassen, werden sich als Klassennamen, die Name ihrer Eigenschaften als Felder und ihr Verhalten als Namen von Methoden/Ereignissen/Nachrichten finden lassen. Eine Simulation enthält keine Komplexität, die sich nicht auch in der simulierten Wirklichkeit findet.

Wenn wir aber auf bestehende Systeme schauen, finden wir reichlich Klassennamen, die nicht darauf hindeuten, dass Objekte dieser Klasse ein wirkliches Ding simulierend repräsentieren. Ebenso ist es mit den Namen der Methoden und Felder. Wer sich mit der Architektur eines solchen Systems beschäftigt, wird sich daher auch nicht auf die Komplexität der simulierten Realität beschränken können.

Das ist für jeden Entwickler oder Architekten eine vertraute Tatsache. Aber angesichts der großen ursprünglich mit der OOP verbundenen Hoffnungen, sollte sie auch ein wenig verwundern. Könnte es sein, dass wir vielleicht OOP schlichtweg falsch betreiben? So offensichtlich wie der Widerspruch zwischen tatsächlicher und im beschriebenen Sinne idealer Software ist, so unklar scheint, welche Frage zu stellen ist, um diesen Widerspruch aufzulösen.

Aber immer, wenn eine Frage unklar ist, lohnt es sich, ein bisschen zu philosophieren. Nicht, um direkt eine Antwort zu bekommen, wohl aber, um die Frage – einige Gedanken später – klarer stellen zu können. Als Ansatzpunkt nehmen wir im Folgenden die Frage nach der Bedeutung der Bezeichner in Software und aller anderen Elemente des Quellcodes, die für Mensch und Maschine eine Bedeutung haben könnten.

Verständlichkeit vor Repräsentation

Die Disziplin der Philosophie oder Linguistik, die sich mit der Bedeutung von „Bedeutungsträgern“ beschäftigt, heißt Semiotik. Als Fachbegriff für „Bedeutungsträger“ – vom Straßenschild bis zum absichtlichen Gähnen – ist dort das „Zeichen“ fest etabliert. „Zeichen“ wird also in einem viel weiteren Sinn als nur für Schriftzeichen, für Daten vom Typ *char* oder für vom Compiler oder Interpreter erkannte Programmelemente verwendet.

Wie Rudi Keller in [Kel95] ausführt, kann man die Frage der Bedeutung eines Zeichens auf zwei verschiedene, einander ergänzende Weisen stellen. Zum einen kann man (mit Aristoteles und Frege) fragen, was denn ein Zeichen repräsentiert. So haben wir die Frage hier bisher ungefähr betrachtet.

Zum anderen kann man aber auch (mit Platon und Wittgenstein) fragen, wie denn Zeichen zum Instrument werden, damit wir einander verstehen. Bereits in Platons “Kratylos Dialog” wird die Frage aufgeworfen: „Wenn ich dieses Wort ausspreche und dabei jenes denke, wie ist es dann überhaupt möglich, dass du erkennst, dass ich jenes denke“ (zitiert nach [Kel95]). Oder, um diesen Satz auf unseren Anwendungsfall hin umzumünzen: „Wenn ich dieses Programmelement oder diesen Bezeichner schreibe und dabei jenes denke, wie ist es dann überhaupt möglich, dass du erkennst, dass ich jenes denke?”

Auf den ersten Blick ist die Antwort für uns sehr einfach: Unsere Programmiersprachen sind zur Kommunikation mit einer Maschine gemacht und schreibende und lesende Programmierer wissen, wie die Maschine die Programme interpretiert. Da versteht man sich. Genügt das nicht?

Wenn man eine ausreichend große Anzahl von Entwicklern am selben Programm hat arbeiten sehen, stellt man oft fest, dass es nicht genügt. Die Maschine als Referenz genügt zum einen nicht, weil sie nicht alles interpretiert. Zum anderen gibt jede verhaltenserhaltende Umgestaltung des Programmes die Möglichkeit, mit einer alternativen Struktur etwas auszudrücken. Martin Fowler erläutert in [Fow99] basierend auf vorhergehender Forschung viele solcher Möglichkeiten. Z. B. können wir unter gewissen Voraussetzungen eine Methode von einer Klasse zur anderen schieben.

Regelhaftigkeit ermöglicht Verständlichkeit

Wenn Entwickler in einer Software Klassennamen lesen, die auf „Adapter“, „Builder”, “Command”, “Composite“, „Decorator“, „Façade”, “Factory”, “Iterator”, “Listener”, “Observer”, “Proxy” oder “Visitor“ enden, dann wissen sie, was damit gemeint ist. Vermutlich könnten sie auch mit einer gewissen Zuversicht entscheiden, wann sie diesen Teil des Klassennamens für korrekt verwendet halten. Sie kennen die Regeln der Verwendung dieser Namen und sie wissen, wie z. B. ein Object der Klasse *XyzListener* zu benutzen ist.

Dieses Wissen wird ihnen im Allgemeinen genügen. Vermutlich werden sie bei diesem Namen außerdem über die Jahre gar nicht mehr daran denken, was es heißt, wenn ein Mensch auf etwas hört. Als sie das Muster kennenlernten, war sicher für einen Augenblick dieses Bild eines „realen“ Zuhörers hilfreich, aber inzwischen „wissen“ sie einfach, wann eine Klasse ein „Listener“ ist. Eine solche Entwicklung eines Zeichens von etwas, das durch Analogieschluss verstanden wird, zu etwas, das regelbasiert verstanden wird, ist typisch für die Entwicklung von Zeichenverständnis (vgl. hierzu Kapitel 13 in [Kel95]).

Schauen wir noch auf ein zweites Beispiel: Entwickler und Architekten wissen sicher, was sie von Methoden namens „open“ und „close“ zu erwarten haben und wie sie sie verwenden würden. Sie würden erwarten, dass die Klasse, die diese Methoden enthält, gewisse Interaktionen zulässt (z. B. „read“ und „write“), die nur nach „open“ und vor „close“ zulässig sind. Vermutlich haben sie auch ein gewisses Verständnis dafür, welcher Art diese Interaktionen sein können. Sie kennen die Regeln genügend gut, um zu verstehen.

Sie werden noch weitere Regeln dieser Art entdecken und sich auch vielleicht erinnern, wie diese sich über die Zeit entwickelt haben. Wie Guy Deutscher und Martin Pfeiffer in [DuP08] ausführen, ist das Bedürfnis nach Regelhaftigkeit eine der wesentlichen Kräfte, die zur Entstehung der Grammatik natürlicher Sprachen führt. Bestimmte grammatische Konstrukte lassen sich am besten als fruchtbares Missverständnis einer Generation über die Regeln der Sprache der vorhergehenden Generation interpretieren. Die Anzahl der Programmierergenerationen ist jedoch noch überschaubar, sodass wir solche Verfeinerung noch nicht erwarten können.

Bedeutung selbst in Leerzeilen

Um noch ein weiteres Beispiel für die Entstehung von Bedeutung zu geben, schauen wir auf ein Programmelement, das die Maschine in aller Regel nicht interpretiert: Leerzeilen. Raymond P. L. Buse und Westley R. Weimer haben 120 Personen 100 Codefragmente auf ihre Lesbarkeit hin bewerten lassen und stellten dabei fest, dass die Verwendung von Leerzeilen wichtiger ist als die Verwendung von Kommentaren. Außerdem konnten sie nachweisen, dass Code der – im Hinblick auf die von ihnen gewonnene Metrik – lesbarer ist,

Was im Programmcode stand:

```
public void isCaching(boolean value) {
  // Caching einschalten
}

public Field containsField(String name) {
  // Suche nach einem Feld mit dem an-
  // gegebenen Namen und gib es zurück
}
```

Was der Entwickler meinte:

```
public void setCaching(boolean value) {
  // Caching einschalten
}

public Field findField(String name) {
  // Suche nach einem Feld mit dem an-
  // gegebenen Namen und gib es zurück
}
```

Abb. 1: Selbst eine semantische Analyse auf rein statistischer Basis kann bei genug vorliegendem Programmcode herausfinden, welche Version „richtig“ ist.

weniger oft zur Beseitigung von Fehlern geändert wurde [BuW10].

Viele Entwickler trennen semantische Blöcke durch Leerzeichen voneinander, also zum Beispiel die Deklarationen von der Verarbeitung und diese wiederum von der Fehlerbehandlung. Besonders nützlich sind sie auch in Code, der für Schulung entsteht. Dieser wird in der Regel linearer präsentiert, als man es von gut strukturiertem Code erwartet. So werden Leerzeilen beispielsweise in einer Dialogkonfiguration genutzt, um deutlich zu machen „dies ist Erzeugung eines Elements einschließlich dessen Konfiguration“ im Gegensatz zu „dies ist die zusätzliche Verbindung der Elemente“.

Die konsistente Verwendung von Leerzeilen nach diesen Mustern führt alleine beim Lesen der Programme anderer Entwickler oder wenn im Unterrichtskontext Beispielcode durchgearbeitet wird, zu einem impliziten Verständnis der Semantik von Leerzeilen – und die Wahrscheinlichkeit ist groß, dass die Semantik der Leerzeilen über den Lernkontext hinaus beibehalten wird.

Xiaoran Wang, Lori Pollock und K. Vijay-Shanker stellen in [WPV11] ein Regelwerk vor, das Vorschläge zur Platzierung von Leerzeilen macht. Das Ergebnis wird von Entwicklern überwiegend als mindestens so gut oder besser als manuelle Platzierungen bewertet. Dieses Regelwerk hat eine nicht unerhebliche Komplexität und dennoch ist es nachvollziehbar, d. h. Entwickler verstehen diese Regelhaftigkeit in der Verwendung von Leerzeilen.

Ein anderes Beispiel für die Semantik von “leeren Stellen” – sogenanntem “Whitespace” – sind die Einrücktiefen für Kontrollblöcke –, eine Regel, die ursprünglich nur als Vorschlag Verbreitung fand und semantisch aus Maschinensicht durch andere Kontrollelemente wie geschweifte Klammern oder Schlüsselworte, z. B. *begin* und *end*, abgebildet wurde.

Mittlerweile allerdings gibt es Sprachen, die diese Elemente ganz bewusst nutzen, um damit entweder Schabernack zu treiben, wie die esoterische Programmiersprache Whitespace [BuM03], oder aber, um überflüssige Redundanzen auszumerzen und Strukturinformationen tatsächlich aus der optischen Struktur abzuleiten, wie in der Programmiersprache Python, die vor allem im Bereich des High Performance Computing immer mehr an Verbreitung gewinnt.

Implizite Namenskonventionen in der Regelhaftigkeit

In jedem Projekt, in jeder Organisation, zu jeder Programmiersprache und zu jedem Framework gibt es mit hoher Wahrscheinlichkeit Namenskonventionen – und teilweise alles andere als gelassene Diskussionen darüber, was denn in diesen Namenskonventionen alles festzuhalten sei. Interessant ist hier eine Betrachtung der verwendeten Bezeichner „in freier Wildbahn“, wie sie E.W. Høst und B. M. Østvold im Jahre 2009 durchgeführt haben.

Sie fanden, dass die Regelmäßigkeit der Namenswahl groß genug ist, um einige Regeln durch statistische Untersuchung zu gewinnen. Hierbei ist es nicht etwa nötig, Namenskonventionen vorzugeben und deren Nichteinhaltung aufzudecken. Allein eine Untersuchung der Regelmäßigkeiten in der Namenswahl in einem großen Korpus von Software führt dazu, eine Methode, die *isCaching* heißt, aber kein Ergebnis zurückliefert und zudem noch einen Boolschen Parameter verwendet, als “falsch benannt” zu zu erkennen; genauso, wie eine Methode, die *containsField* heißt, aber keinen Boolschen Wert, sondern das Feld selber zurückgibt. „Falsch benannt“ bedeutet hier natürlich keine absolute Aussage, wie sie bei der Syntax eines Programms möglich ist, sondern eine relative Aussage über die implizit etablierte Semantik. Wenn das Schema nicht den in der restlichen Codebasis erkannten Regeln entspricht, ist es wahrscheinlich, dass der Programmierer, der diesen Code liest, die “falsche” Assoziation bildet – nämlich die, die für den Rest der Software richtig wäre. Bessere Namen für diese beiden Beispiele wären z. B. *setCaching* bzw. *findField*.

Zwei Regeln der Repräsentation: Spiegel und Fenster

Wir hatten uns eine Auszeit von der Frage genommen, was denn die Elemente unserer Programme repräsentieren. Nachdem wir uns nun vergewissert haben, dass große Chancen bestehen, dass auch ohne Kenntnis darüber, was genau im Programm abgebildet ist, ein Verständnis zwischen Programmierern über ihren Quellcode möglich ist, können wir uns dieser Frage nun mit größerer Gelassenheit stellen.

Schon lange gilt unter Entwicklern die Maxime, dass gute Namensgebung entscheidend ist (“Naming is essential”, Adele Goldberg) – vor allem, wenn es darum geht, Bestandteile einer Anwendung auch in anderen Anwendungen zu verwenden. Da in der objektorientierten Programmierung ja eigentlich nie Objekte programmiert werden, sondern die Klassen, aus denen die Objekte dann zur Laufzeit erstellt werden, haben wir – namenstechnisch – immer mit Vertretern für alle Objekte der gleichen Klasse zu tun.

Im Falle einer Simulation ist zwar ein gut gewählter Klassenname sicher auch ein guter Name für ein Konzept, unter dem wir die simulierten Dinge zusammenfassen würden. Was aber, wenn es gar kein Ding

gibt, das durch die Klasse repräsentiert wird? Oder wenn dieses Ding nicht nur möglich ist, sondern wirklich existiert?

Gerade, wenn es um Geschäftsanwendungen geht, repräsentieren Klassen oft Dinge, die es so in der physischen Welt nicht gibt, z. B. ein Konto, zu dem es zwar Kontobücher, Kontoauszüge und ähnliches gibt, das aber im eigentlichen Sinne eine "reine" Idee ist – schließlich ist das Konto selber nicht physisch greifbar. Bei Simulationen auf der anderen Seite gibt es die physische Entsprechung oft durchaus, so zum Beispiel bei der Klasse "Flugzeug" in einem Luftverkehrskontrollsystem, ohne das die Objekte der Klasse "Flugzeug" wirklich mit ihren physischen Repräsentationen verbunden wären.

Bei Systemen, die tatsächlich messen und regeln, finden sich schließlich Objekte, die mit ihren Gegenstücken in der physischen Welt mehr oder weniger direkt verbunden sind – Sensoren, Schalter, Magnetventile, Einspritzpumpen und dergleichen mehr. Gerade diese sind zwar oft nicht objektorientiert modelliert, aber das ist eher ein Thema für "Embedded Systems".

In diesen "Grenzgebieten" findet sich viel Potenzial, wenn es um die Zuordnung von Bedeutungen zu Elementen der Softwaresysteme geht. Schließlich sind gerade Konzepte wie Schalter, Sensor oder Anzeige in heutigen Systemen vielfach vorhanden – vom "Schalter" der Feststelltaste Großschreibung über den Sensor für die Mausbewegung bis hin zu den unterschiedlichsten Möglichkeiten für ein informationsverarbeitendes System –, um etwas anzuzeigen, in Fenstern auf dem Bildschirm, durch Leuchtdioden oder auch durch die Ausgabe eines Stücks Papier. Wobei letzteres für einen Webshop zwar ein Problem des Anwenders ist, für die Software eines Fahrkartenautomaten aber einen zentralen Bestandteil ihres Daseinszwecks ausmacht.

Um all diese unterschiedlichen Sichtweisen leichter handhabbar zu machen, helfen die Konzepte *Fenster* und *Spiegel*, wie sie in [SNM11] vorgeschlagen wurden. Die Fenster sind dabei in diesem Sinne genauso wenig die *Fenster* der Benutzungsoberflächen, wie die *Spiegel*, sondern vielmehr eine Betrachtungsweise für die Elemente der Systeme, die wir entwerfen, warten und qualitätssichern wollen.

Wann immer eine Änderung am Zustand des Objektes sich auf die physische Realität auswirkt – wenn zum Beispiel der Aufruf von *aFile.delete()* eine Datei löscht oder ein *aDisk.eject()* einen USB-Stick trennt –, lassen sich die Objekte als *Fenster* in die Realität betrachten. Wenn andererseits Ereignisse "von außen" erkannt werden sollen und das System auf sie reagieren soll, so wird eine andere Denkweise benötigt – hier spiegelt das Objekt die Realität wider.

Wichtig ist dabei, zu unterschieden, welche Rolle der entsprechende Teil des Systems im aktuellen Kontext wahrnimmt – während für den Großteil des Systems ein *mouseClicked()* als Ereignis betrachtet werden kann und ein aus Systemsicht externes Ereignis *spiegelt*, muss natürlich ein kleiner Teil des Systems die Welt ganz anders betrachten – der Teil, der allerdings das Ereignis erzeugt, muss den "echten" Mausbutton durch ein *Fenster* betrachten und direkt mit ihm interagieren.

Aber nicht nur unterschiedliche Teile des selben Systems können unterschiedliche Sichten auf das gleiche Objekt entwickeln, sondern die komplette Modellierung eines Systemausschnitts kann von der Verwendung der Konzepte *Spiegel* und *Fenster* bestimmt sein – womit diese beiden Konzepte übrigens die Idee "Architektur durch Metaphern" aus dem Extreme Programming sehr greifbar umsetzen.

Verdeutlichen wir dies, [SNM11] folgend, durch ein plastisches Beispiel aus dem Bereich der Luft- und Raumfahrt. In einer Simulation kann die Fähigkeit einer Rakete zu starten durch eine Operation wie *Rocket.launch()* repräsentiert werden. Wird das System zur Steuerung einer physischen Rakete genutzt, dann kann die gleiche Signatur genutzt werden, solange die Metapher des *Fensters* verwendet wird.

Unglücklicherweise kann während des Starts einer physischen Rakete eine Menge schief gehen, sodass Spolskys-Gesetz zur Anwendung kommt: "Alle nicht-trivialen Abstraktionen sind zu einem gewissen Grad durchlässig". Die Abstraktion *Rocket.launch()* mag in vielen Fällen eine gute Wahl sein, z. B., wenn es um den Start einer Rakete in einem Computerspiel geht.

Aber, wenn Dinge schief gehen können, wie beim physischen Start, muss das System in der Lage sein, darauf zu reagieren. In diesem Fall erscheint es sinnvoll, die Illusion des direkten Zugriffs beispielsweise dadurch aufzulösen, dass andere Namen, wie *RocketProxy.initiateLaunch()*, verwendet oder Fehlerbehandlungen deklariert werden. Die verwendeten Bezeichner machen dann klar, dass der Start hier nur angestoßen wird und mit einem Fehlschlagen gerechnet werden muss.

Ist die Klasse *Rocket* ein reiner *Spiegel*, so gibt es wahrscheinlich gar keine Methode *launch()* mehr, da das Konzept des Starts von ganz anderen Klassen umgesetzt wird, deren Instanzen möglicherweise in *Rocket*-Objekten solche Informationen wie die Zeit des Beginns des Starts über Methoden wie *Rocket.setLaunchInitiated (Time)* ablegen. In einem *Spiegel*-Objekt sind die Verzerrungen oft groß, aber der Umfang der vermittelten Information sollte ähnlich sein wie bei *Fenster*-Objekten, auch wenn das Verhalten der physischen Objekte hier oft als Zustand repräsentiert wird und das Verhalten sich in mit dem *Fenster*-Objekt zusammenarbeitenden Objekten wiederfindet.

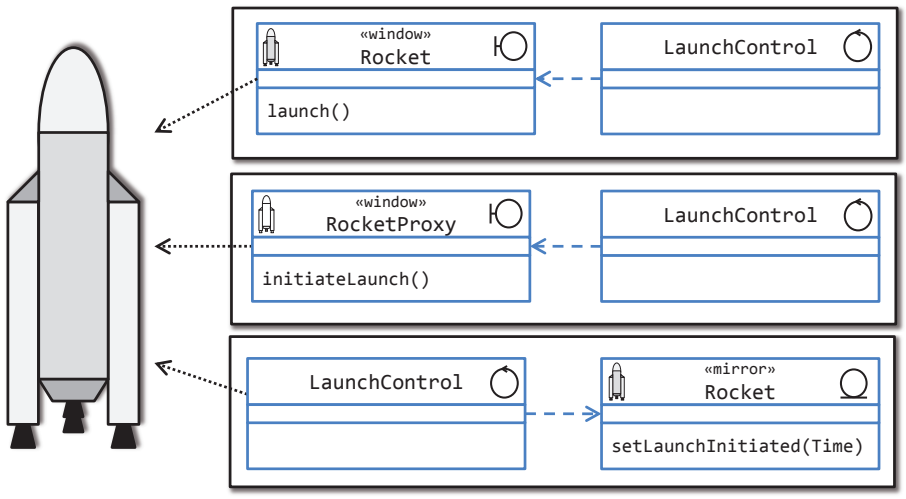

Abb. 2: Beispiel zur Raketenwissenschaft der Objektrepräsentation: Repräsentation als Fenster ohne und mit Berücksichtigung der Nichtidentität von Ding und Objekt sowie Repräsentation als Spiegel.

Fazit

Wenn sich objektorientierte Software als Model der Wirklichkeit entwickeln lässt, wie es bei Simulationen der Fall ist, fördert dies eine klare Architektur. Da, wo dies nicht möglich ist, ist eine solide Regelmäßigkeit der Software dem Verständnis förderlich. Insbesondere lässt sich unterscheiden, ob sich die Software zu realen Objekten – wenn sie sich denn auf diese bezieht – als *Spiegel* oder als *Fenster* verhält. Diese Unterscheidung kann wiederum genutzt werden, um eine konsequente regelhafte Strategie zur Bezeichnerwahl zu entwickeln. ■

Referenzen

[BuM03] Edwin Brady, Chris Morris, Whitespace, http://compsoc.dur.ac.uk/whitespace/, Erstveröffentlichung 01.04.2003.

[BuW10] Raymond P. L. Buse, Westley R. Weimer, Learning a Metric for Code Readability, IEEE Transactions on Software Engineering, vol. 36, pp. 546–558, 2010.

[DuP08] Guy Deutscher, Martin Pfeiffer, Du Jane, ich Goethe: Eine Geschichte der Sprache, C.H.Beck, München, 2008.

[Fow99] Martin Fowler et al., Refactoring, Improving the design of existing code, Addison-Wesley Professional, 1999.

[HuØ09] Einar W. Høst, Bjarte M. Østvold. Debugging Method Names. ECOOP, 2009, Genua.

[Kel95] Rudi Keller: Zeichentheorie: Zu einer Theorie semiotischen Wissens, UTB, Stuttgart, 1995.

[KuH12] William Kent, Steve Hoberman, Data and Reality: A Timeless Perspective on Perceiving and Managing Information in Our Imprecise World, Technics Publications LLC, Denville, 2012, Erstveröffentlichung in 1978.

[SNM11] Daniel Speicher, Jan Nonnen, Holger Mügge: How many realities fit into a program - Notes on the meaning of meaning for programs, SKY 2011, Paris.

[WPV11] Xiaoran Wang, Lori Pollock, K. Vijay-Shanker, Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability. WCRE 2011, Limerick.