J. Michael, J. Pfeiffer, A. Wortmann (Hrsg.): Modellierung 2022 Satellite Events,
Digital Library, Gesellschaft für Informatik e.V. 176

Modellierung in der Informatik-Hochschullehre

ein Trauerspiel

Wolfgang Reisig[1]

Der Modellierer:

Habe nun, ach!
Philosophie,
Juristerei und Medizin,
Und leider auch Informatik
Durchaus studiert, mit heißem Bemühn.
Da steh' ich nun, ich armer Tor,
Und bin so klug als wie zuvor!
Heiße Magister, heiße Doktor gar,
Und ziehe schon an die zehn Jahr'
Herauf, herab und quer und krumm
Meine Schüler an der Nase herum –

Der Programmierer:

Verzeiht! es ist ein groß Ergetzen,
Sich in den Geist der Zeiten zu versetzen;
Zu schauen, wie vor uns ein weiser Mann gedacht,
Und wie wir's dann zuletzt so herrlich weit gebracht.

frei nach J.W. v. Goethe

Vorbemerkung

Es ist das Risiko des Veranstalters und das Privileg des Sprechers eines eingeladenen Vortrags, dass kein Gutachter „helfend" eingreifen kann. Ich kann also beliebig pointiert formulieren, und das will ich nutzen.

1 Die Lage

Jeder Schreiner macht einen Plan, ein Modell, bevor er einen Stuhl baut.

[1] Humboldt-Universität zu Berlin, Berlin, Germany, reisig@informatik.hu-berlin.de

Jeder Elektriker macht einen Schaltplan.

Jeder Architekt plant ein Haus, bevor er es baut. Modelliert es, visualisiert es, Studenten bauen 3D-Modelle, manche Architekten kümmern sich gar nicht ums Bauen, sondern machen nur Pläne.

Jeder Ingenieur-Student lernt erst mal Differential-/Integralrechnung und Differentialgleichungen als Modellierungs-Handwerkszeug.

Kurz: Jeder Handwerker und jeder Ingenieur-Student lernt erst mal modellieren.

Ein Informatik-Student lernt erst mal programmieren.

Informatik-Kurrikulum HU Berlin, aber auch sonst wo:

1. Semester: Hacken mit Java (andernorts, edlere Variante: Hacken mit Python);
2. Semester: Algorithmen und Datenstrukturen (man beachte die Reihenfolge: 1. Semester: Hacken. Dann im 2. Semester ein wenig Systematik);
3. Semester: Software Engineering. Noch mehr Hacken und Fummeln.

2 Ein Beispiel

Meine Erfahrung als Beisitzer bei „endgültig durchgefallen“-Prüfungen: Prüfling soll *Sortieren* erklären. Um es einfach zu halten, braucht er kein Java-Code hinschreiben, sondern kann Sortieren „gern auch intuitiv“ schildern.

Wenn er es gut macht, malt er eine Sequenz aus Kästchen, schreibt was rein, skizziert mit Pfeilen, welcher Eintrag wann wohin wandert. Was ist das, was er da macht? Eine umgangssprachliche Skizze? Ein Modell des Algorithmus? Der Algorithmus selbst? Kann man das nicht systematischer machen?

Fragen wir das HPI: Was ist *Bubblesort*? Die Website zu Bubblesort enthält [HP22]:

- ein Beispiel: Eine Sequenz von acht Spielkarten wird in sechs Schritten sortiert.
- ein Pseudocode-Programm:

```
function BubbleSort(A) {
    newRound := TRUE;
    for i := n-1 downto 1 do {
      if newRound then {
         newRound := FALSE;
         for j := 0 to i-1 do {
             if A[j] > A[j+1] then {
```

```
                    temp := A[j];
                    A[j] := A[j+1];
                    A[j+1] := temp;
                    newRound := TRUE;
                }
            }
        }
        else break
    }
}
```

„Bubblesort ist die Verschachtelung von zwei for-Schleifen“

- Die Idee des Nachweises der korrekten Sortierung durch „Bubblesort“ („Korrektheitsbeweis“) wird an der Tafel erläutert:
  „Die äußere for-Schleife macht . . . “ ; „Elemente sind falsch rum“; „Vergleich stimmt / stimmt nicht“

Warum stellt man so was ins Netz? Was ist die Rolle des Pseudocode-Programms? Warum nicht gleich Java? Ist das Pseudocode-Programm eine Art Modell des Algorithmus? Kann man nicht ohne eine Programmier-Syntax den Bubblesort-Algorithmus hinschreiben? Warum redet man nicht präzise? Was wäre denn hier „präzises Reden“?

Vorschlag: Vernünftig über Bubblesort reden

Gegeben seien

1. einige Mengen:
   Eine Menge A mit einer Ordnung, $\leq$.
   Daraus abgeleitet:
   Tupel, geordnetes Tupel, Liste über A, geordnete Liste.
2. einige Funktionen:
   - für Tupel (a, b):

     „Tupel (a, b) ordnen”:
     ord : *Tupel* $\rightarrow$ *Tupel*
     $ord(a, b) := (a, b)$, falls (a, b) geordnet, sonst (b, a).

   - für Listen $(a_0, \ldots, a_n)$:

     „(a_{i-1}, a_i) *ordnen*“:

i-Schritt: *Listen* $\rightarrow$ *Listen*
i-Schritt$(a_0, \ldots, a_{i-1}, a_i, \ldots, a_n) \coloneqq (a_0, \ldots, ord(a_{i-1}, a_i), \ldots a_n)$

Wir schreiben „*i-Schritt*“ statt „*i-Schritt*$(a_0, \ldots, a_n)$“, wenn $(a_0, \ldots, a_n)$ im Kontext klar ist.

„Das größte Element ganz nach rechts spülen“:
max : *Listen* $\rightarrow$ *Listen*
max$(a_0, \ldots, a_n) \coloneqq$ *1-Schritt;* 2*-Schritt; . . . ; n-Schritt.*

„a_i an die richtigen Stelle setzen, wenn $a_{i+1}, \ldots, a_n$ schon an der richtigen Stelle stehen“:
i-ordnen : *Listen* $\rightarrow$ *Listen*
i-ordnen$(a_0, \ldots, a_{i-1}, a_i, \ldots, a_n) \coloneqq (max(a_0, \ldots, a_i), a_{i+1}, \ldots, a_n)$

Wir schreiben „*i-ordnen*“ statt „*i-ordnen*$(a_0, \ldots, a_n)$“, wenn $(a_0, \ldots, a_n)$ im Kontext klar ist.

„$(a_0, \ldots, a_n)$ ordnen“:
geordnet: *Listen* $\rightarrow$ *Listen*
geordnet$(a_0, \ldots, a_n) \coloneqq$ *n-ordnen*; *n*-1*-ordnen*; . . . ; 1*-ordnen*.

Das ist ein operationelles, formales Modell des Bubblesort-Algorithmus. Die intuitive Idee des Algorithmus wird präzise, mit den passenden formalen Ausdrucksmitteln, formuliert. Die Funktion *max* schildert formal und zugleich anschaulich, wie ein „*bubble* aufsteigt“. Die Funktion *i-ordnen* schildert, wie alle „*bubbles* aufsteigen“. Das versteht auch ein dummer Student. Ein Korrektheitsbeweis für den Algorithmus gehört auf diese Ebene. Von hier aus, wenn's denn sein muss, kann man Programmcode generieren.

Es ist hier nicht der Ort, allgemeine Prinzipien der Formulierung von Algorithmen zu diskutieren. Allerdings zeigt dieses Beispiel: An den vielen Studienabbrechern haben wir Schuld! Wie [Pf94] zur Informatik-Ausbildung sagt: *Fertigkeiten ersetzen Wissen, Machen erdrückt das Verstehen. Typischerweise lernen die meisten, Programme zu schreiben, ohne jemals welche gelesen, geschweige verstanden zu haben.* Das ist kein Wunder: ein Programm soll ja von einem Rechner verstanden werden, nicht unbedingt von Menschen. Für Menschen braucht man stattdessen Modelle.

3 Historische Entwicklung

In den 1950/60er Jahren waren zentrale Themen der Informatik das Suchen und Sortieren von Daten auf Magnetbändern, die numerische Mathematik und Rechnerarchitekturen (Betriebssysteme, Compiler). Dafür wurden immer neue, vermeintlich bequemere Programmiersprachen vorgeschlagen. Zwei wichtige Publikationen: *Assigning Meanings to*

Programs [Fl67]. Immerhin, nach 10 Jahren Programmieren: ein Programm soll etwas Eindeutiges bedeuten! Ein Jahr später: *The Art of Computer Programming* [Kn68]. Ein Programmierer ist ein Künstler, eine Art Genie. Knuth verwendet eine „synthetische Programmiersprache“. Was soll das? Fehlt ihm bei den damals gängigen Programmiersprachen die Präzision? Glaubt er, seine Sprache eigne sich besonders gut zur Formulierung der Algorithmen? Programmieren war damals schon eine fehleranfällige Angelegenheit, und es wurde eine „Software-Krise“ ausgerufen. Zur Lösung wurden weitere Programmiersprachen vorgeschlagen, darunter monströse wie ALGOL 68 und ADA (Wie ein Zyniker bemerkt: „Manche der früheren Sprachen waren ein deutlicher Fortschritt gegenüber einigen ihrer Nachfolger . . . “). Der Grundfehler all dieser Bemühungen war und ist der Versuch, mit Programmiersprachen zu modellieren. Die ganz unbegründete Vorstellung, ein Algorithmus soll von seiner Implementierung her verstanden werden, und nicht zunächst einmal von seinem beabsichtigten Sinn und Zweck her. Dijkstra hat diese Vorstellung befördert mit seinem wiederholt vorgetragenen Vorschlag, zwischen der formalen und der Anwender-Sicht auf ein System eine gedankliche Mauer zu errichten. Seine Begründung: Das „correctness problem“ des Informatikers verlange ganz andere Herangehensweisen als das „pleasantness problem“ des Anwenders [Di89].

Heute sind typische zentrale Themen der Informatik beispielsweise: Ware bestellen und Rechnung bezahlen, Werbefläche im Internet versteigern, autonome Fahrzeuge steuern. Allgemeiner formuliert: Ein gegebenes realweltliches Problem so präsentieren, dass Rechner Teile der Lösung übernehmen, zusammen mit Menschen oder mechanischen Maschinen. Beispiel: Ausschreibung und Besetzung einer Stelle in einer Behörde. Im konkreten Ablauf der Besetzung einer Stelle greift ein Algorithmus auf rechnergestützte Datenbanken zu, kommuniziert mit andern Verwaltungen, kontrolliert die Einhaltung vorgegebener Regeln, überwacht Fristen, etc. Zu dem Algorithmus gehören auch Menschen, die unter den Bewerbern den geeignetsten Kandidaten wählen und Urkunden unterschreiben. Solche Algorithmen müssen modelliert werden mit Modellen, die Dijkstra's Mauer geradezu einebnen.

4 Was hat die Modellierungs-Community zu bieten?

Vorschläge der Softwaretechnik: *Abstract State Machines (ASMs) / Actor model / Alloy / ANSI/ISO C Specification Language (ACSL) / Autonomic System Specification Language (ASSL) / B-Method / CADP / Common Algebraic Specification Language (CASL) / Esterel / FOCUS / Java Modeling Language (JML) / Knowledge Based Software Assistant (KBSA) / Lustre / mCRL2 / MSC-LSC / Perfect Developer / Petri nets / Predicative programming / Process calculi: CSP, LOTOS, π-calculus / RAISE / Rebeca Modeling Language / SPARK Ada / Specification and Description Language (SDL) / Statecharts / TLA+ / USL / VDM: VDM-SL, VDM++ / Z notation*.

Vorschläge der Wirtschaftsinformatik: *ADONIS / ARIS / BPMN / EPK / MEMO / St. Gallen Approach / UML-Varianten*.

Einige dieser Modelle wurden und werden in großen Softwareprojekten verwendet, in der Wirtschaftsinformatik mehr als in der allgemeinen Softwaretechnik, weil in der Wirtschaftsinformatik der Bedarf offensichtlicher ist: Wirtschaftsinformatik gestaltet nicht nur Software, sondern auch Prozesse, Organisationen und Geschäftsmodelle.

Ich selbst habe viele Jahre lang eine Vorlesung für Bachelors mit einem Strauß verschiedener Methoden bestückt: ASM, BPMN, CASL, FOCUS, MSC&LSC, Petrinetze, Prozessalgebren, Statecharts, TLA, Z. Die Resonanz war mittelmäßig. Immerhin sind jedes mal ein paar Studenten für Bachelor-/Masterarbeiten und Promotionen dabei geblieben.

Jeder bisher genannte Vorschlag hat Schwächen. Es gibt keine einheitliche konzeptionelle, theorie-basierte und allgemein akzeptierte Grundlage. Indem die sog. „Theoretische Informatik“ sich auf die Manipulation von Zeichenketten beschränkt, trägt sie hier nichts bei. Damit gibt es auch keine akzeptierte Systematik für Modellierungs-Frameworks und Modellierungs-Infrastrukturen, die man in der Lehre verwenden oder darstellen könnte.

Vielleicht könne man sich auf einige Anforderungen für eine solche Systematik einigen, beispielsweise:

- Ein System soll zunächst aus Sicht der Anwender modelliert werden, nicht aus Sicht der Implementierbarkeit.
- Ein Modellierer überführt informelle, intuitive Ideen zu einem gegebenen oder intendierten System in ein formales Modell.
- Ein Modell beschreibt lebensweltliche, organisatorische, von Menschen ausgeübte, von Maschinen durchgeführte, sowie digitale Prozesse in einer einheitlichen Weise.
- Die Modellierung von Systemen skaliert, ist also auch zur Beschreibung umfangreicher Systeme geeignet.

5 Schluss: Thesen zur Modellierung in der Informatik-Lehre

- Um Informatik an einer wissenschaftlichen Hochschule vernünftig zu lehren, müsste man Informatik erst mal vernünftig systematisch als Wissenschaft formulieren. Dann ergibt sich die zentrale Rolle der Modellierung von selbst [Re20]. Modellierung in der derzeitigen Informatik-Lehre kann man nicht verbessern. Man muss sie neu aufsetzen, zusammen mit der Gestaltung einer Wissenschaft der Informatik.
- Wer nur Zeichenketten manipulieren will, meint oft, er brauche keine Modelle. Allerdings meint [Pf94]: *Es kann keine Rede davon sein, dass sich aus der Technik des Formalen die Grenzen der informatischen Modellierung verstehen lassen.* Die Vulgär-Version der Church’schen These: „Was man mit einem Computer machen kann, das kann man auch mit einer Turingmaschine machen“ ist offensichtlich blanker Unsinn, wird aber von den Studenten so verstanden.

- Wir haben kein Erkenntnis-Problem, sondern ein Akzeptanz-Problem: an sich ahnen viele Kollegen, dass man Konzepte, Ideen, Algorithmen besser mit passenden Notationen unterrichten sollte, also mit passenden Modellen. Aber üblich sind nun mal Programm-Notationen.
- Der Informatik geht es zu gut: Mit Hacken kann man herrlich Geld verdienen. Warum soll man es da ordentlich machen (also modellieren)? „Do kannsch au em Ochs ins Horn pfetze!“

Danksagung

Peter Fettke hat mich zu einigen Änderungen in einer früheren Fassungen dieses Textes angeregt. Ich danke ihm.

Literaturverzeichnis

[Di89] Dijkstra, E. W.: Reply to comments. Commun. ACM, 32(12):1414, 1989.

[Fl67] Floyd, Robert W.: Assigning Meanings to Programs. Proceedings of Symposium on Applied Mathematics, 19:19–32, 1967.

[HP22] HPI: `https://hpi.de/friedrich/teaching/units/einfache-sortierverfahren.html`, 2022.

[Kn68] Knuth, Donald E.: The Art of Computer Programming. Addison-Wesley, 1968.

[Pf94] Pflüger, J.: Informatik auf der Mauer. Informatik-Spektrum, 17:251–257, 1994.

[Re20] Reisig, Wolfgang: Informatics as a Science. Enterprise Modelling and Information Systems Architecture – International Journal of Conceptual Modeling, 15(6):1–13, 2020.