Zur Berechnung der softwaretechnischen Komplexität von einfachen objektorientierten Programmen

Marc Roßner und Michael Fothe

Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik
Ernst-Abbe-Platz 2, 07743 Jena
{marc.rossner,michael.fothe}@uni-jena.de
https://www.fmi.uni-jena.de/

Abstract. Das Komplexitätsmaß nach Peter Rechenberg lässt sich auf einfache objektorientierte Programme übertragen. Solche Programme sind Gegenstand des Informatik-Unterrichts an Schulen. Die Softwaremetrik stellt eine Grundlage dafür dar, unterschiedliche Programme zu einer Aufgabe zu vergleichen.

Keywords: Softwaremetrik, Objektorientierung, Abituraufgaben

1 Einleitung

Im Thüringer Lehrplan für das Fach Informatik am Gymnasium ist für den Bereich der Selbst- und Sozialkompetenz unter anderem formuliert: "Der Schüler kann [...] Modelle und deren Implementierung beurteilen." [6, S. 20]. Für das Beurteilen zweier Implementierungen ein und desselben Modells wird eine Vergleichsgrundlage benötigt. In diesem Aufsatz wird dafür die Software-Metrik von Peter Rechenberg vorgeschlagen, und zwar in einer modifizierten Form, die sich auf einfache objektorientierte Programme bezieht. Rechenberg führte seine Metrik bereits 1986 mit dem Blick auf die Programmiersprachen Pascal (imperativ; strukturiertes Programmieren) und Modula-2 (modulares Programmieren) ein [5]. Die Gesamtkomplexität CC eines Programms ergibt sich demnach als Summe der Anweisungskomplexität SC, der Ausdruckskomplexität EC und der Datenkomplexität DC:

$$CC = SC + EC + DC \tag{1}$$

(siehe auch http://www.fmi.uni-jena.de/fmimedia/rossner_rechenberg.pdf). Es ist eine Forderung des Thüringer Lehrplans, "einfache Sachverhalte objektorientiert modellieren und implementieren" [6, S. 35] zu können. Sachverhalte, die Gegenstand der Abiturprüfung im Fach Informatik sind, dürften in diesem Sinne häufig als einfach anzusehen sein, sodass es lohnenswert erscheint, das modifizierte Verfahren auf solche Abituraufgaben anzuwenden. Exemplarisch wurden zwei Aufgaben aus Thüringen (Leistungsfach Informatik) auf verschiedenen Wegen in der Programmiersprache C++ gelöst, die Komplexität des

Quellcodes berechnet und die unterschiedlichen Implementierungen sowie deren Bewertungen verglichen. Ursprünglich waren die Abituraufgaben mit Turbo Pascal, Oberon-2 oder Java zu lösen.

Die Bewertungen der Programme können auch Ausgangspunkt für Diskussionen zu den Anforderungen in Informatikprüfungen sein.

2 Übertragen der Software-Metrik von Rechenberg auf einfache objektorientierte Programme und C++

In Konkretisierung von Festlegungen der EPA Informatik lassen sich die folgenden Kompetenzen angeben [3, S. 74]:

Die Schülerinnen und Schüler

a) erläutern Grundkonzepte der objektorientierten Modellierung (Objekt, Klasse, Vererbung, Polymorphie, Datenkapselung, Wiederverwendbarkeit),
b) modellieren Probleme, dokumentieren die Modelle und stellen die Modelle mit grafischen Mitteln dar,
c) analysieren, modifizieren und überprüfen eigene oder gegebene Modellierungen und
d) implementieren objektorientierte Modelle.

Unter einfachen objektorientierten Programmen sollen in diesem Aufsatz solche Programme verstanden werden, die sich ausschlielich auf die Grundkonzepte Objekt und Klasse beziehen. Diese Konzepte bieten die Möglichkeit des strukturierten Aufbaus von Softwareprojekten. Man kann Teillösungen in Klassen auslagern und diese in einem Hauptprogramm zur Gesamtlösung vereinen. Bei diesem Vorgehen kann die Implementierung der Methoden als quasi-prozedural angesehen werden; die Methoden können in der Software-Metrik wie gewöhnliche Prozeduren behandelt werden.

Für das Berechnen der Anweisungs- und der Ausdruckskomplexität können die Regelungen aus [5] unmittelbar übernommen werden. Das Berechnen der Datenkomplexität kann bei den Methoden so erfolgen, wie es bei Rechenberg für die modulare Programmierung beschrieben ist. Aus Sicht der Modul-Implementierung könnte die Schnittstelle als "äußerste Schicht" aufgefasst werden und alle Namen würden die Bewertung 3 erhalten. Andererseits muss man sich zur Implementierung des Moduls die Bedeutung der Namen ohnehin vergegenwärtigen, sodass man sie innerhalb des Moduls auch generell als lokal betrachten könnte. Es wird für C++ festgelegt:

Namen in Modulen werden nach der letzteren Variante mit 1 und erst bei deren Verwendung im Hauptprogramm mit 3 bewertet. Sollten Module ineinander geschachtelt sein, greift die Bewertung mit der Blocktiefendifferenz i.

Für Attribute soll gelten, dass den gekapselten Zustandsdaten, auf die alle oder fast alle Methoden der Klasse zugreifen, die Bewertung 3 zugewiesen wird unabhängig davon, ob innerhalb der Implementierung der Klasse oder im Hauptprogramm.

Alle Bezeichner des Namensraumes `std` sollen als Standardnamen gelten und somit keiner Bewertung unterliegen (z. B. `cout`, `cin`, Standarddatentypen).

Für die Verwendung von `cout` für die Kommandozeilenausgabe wird in Anlehnung an `printf(...)` festgelegt: Der gesamte als Ausgabe erzeugte Text wird als genau ein übergebener Parameter behandelt, unabhängig davon, aus wie vielen Teilen sich dieser zusammensetzt. Für die Bewertung gilt dann:

$$SC/EC(cout << '' \, Ergebnis :'' << ergebnis << endl;) = 1 + 1 \tag{2}$$

Die `return`-Anweisung besitzt in C++ einen Parameter. Daher wird festgelegt:

$$SC(return\, ergebnis;) = 1 + 1 \tag{3}$$

Der Operator der Ungleichheit `!=` besteht typografisch aus zwei Zeichen; es wird festgelegt:

$$EC(if(eingabe! = gesucht)) = 2 \tag{4}$$

3 Analysieren der Aufgabe "Plumpsack"

Das Spiel "Plumpsack" [2] ist mithilfe einer Ringliste zu realisieren. Die Ringliste wird durch ein Array implementiert. Es wurde eine prozedurale und eine objektorientierte Musterlösung erarbeitet. Die Tabellen 1 und 2 beinhalten für beide Lösungsvarianten die Bewertung des Teilprogramms zur Initialisierung einer Spielrunde.

Code	SC	EC	DC
`void spiel::init()`	nicht bewertet		
`{`			
`1 for (int i=0; i<anz_spieler; i++)`	3	2	4
`  {`			
`2  spielfeld[i]=i+1;`	1,5	4,5	4
`  }`			
`3 cout<<"Startaufstellung:"<<endl;`	2	2	0
`4 print();`	1	1	1
`}`			
Summe	7,5	9,5	9
CC		26	

Table 1. Bewertung der Methode zum Initialisieren des Spielfelds im objektorientierten Programm.

Code	SC	EC	DC
`void init(int spielfeld[], int anz_spieler)`	nicht bewertet		
`{`			
`1 for (int i=0; i<anz_spieler; i++)`	3	2	3
`{`			
`2 spielfeld[i]=i+1;`	1,5	4,5	3
`}`			
`3 cout<<"Startaufstellung:"<<endl;`	2	2	0
`4 print(spielfeld, anz_spieler);`	3	3	5
`}`			
Summe	9,5	11,5	11
CC		32	

Table 2. Bewertung der Prozedur zum Initialisieren des Spielfelds im prozeduralen Programm.

Beide Teilprogramme unterscheiden sich in der Zeile 4 beim Aufruf der print-Anweisung. Im prozeduralen Programm werden die Datenstrukturen von außen als Parameter übergeben. Im objektorientierten Programm wird direkt auf die in der Klasse gekapselten Datenstrukturen zugegriffen. Dem Bewertungs-Vorteil der fehlenden Parameter bei der Anweisungs- und Ausdruckskomplexität steht in der Objektorientierung die höhere Datenkomplexität gegenüber. Die Köpfe von Unterprogramm und Methoden sind unterschiedlich, werden jedoch nicht bewertet.
Die Erzeugung von Zufallszahlen wurde auf zwei Wegen realisiert; die Tabellen 3 und 4 geben Musterlösungen einschließlich Bewertung an. Beide Varianten unterscheiden sich in der Gesamtkomplexität nur wenig. Auch an diesem Beispiel zeigt sich somit, dass verschiedene Arten der Realisierung zu vergleichbaren Bewertungen führen können.

Code	SC	EC	DC
`bool gefangen()`	nicht bewertet		
`{`			
`time_t zeit;`	nicht bewertet		
`1 time(&zeit);`	2	2	4
`2 srand((unsigned int)zeit);`	2	2	4
`3 if (rand()%2==0)`	4	5	3
`4 return false;`	3	0	0
`5 else`			
`6 return true;`	3	0	0
`}`			
Summe	14	9	11
CC		34	

Table 3. Bewertung der ersten Variante zum Erzeugen einer Zufallszahl.

Code	SC	EC	DC
`bool gefangen()`	nicht bewertet		
`{`			
`  1 srand(static_cast<int>(time(NULL)));`	4	5	9
`  2 if (rand()%2==0)`	4	5	3
`  3  return false;`	3	0	0
`  4 else`			
`  5  return true;`	3	0	0
`}`			
Summe	14	10	12
CC		36	

Table 4. Bewertung der zweiten Variante zum Erzeugen einer Zufallszahl.

In der Tabelle 5 sind die Bewertungen der vollständigen prozeduralen und objektorientierten Programme zum Spiel "Plumpsack" angegeben.

	prozedural	objektorientiert
Anweisungskomplexität	121,75	112,75
Ausdruckskomplexität	118	110
Datenkomplexität	122	136
Gesamtkomplexität	361,75	358,75

Table 5. Bewertung von Programmen zur Aufgabe "Plumpsack".

Die Übertragung der prozeduralen in die objektorientierte Lösung erfolgte lediglich durch einige redaktionelle Umgruppierungen im Quelltext und dessen Anpassung an die Syntax der Objektorientierung. Beide Programme besitzen die gleiche Anzahl bewerteter Quellcodezeilen; nur sieben der bewerteten Codezeilen mussten geändert werden. Die Gesamtkomplexitäten der beiden Programme sind nahezu gleich; der Unterschied macht nur rund 1 % aus. Aus dieser Beobachtung lässt sich ableiten, dass man es in der Abiturprüfung dem Prüfungsteilnehmer überlassen kann, ob er eine Aufgabe prozedural oder objektorientiert löst (bei Beschränkung auf die Grundkonzepte Klasse und Objekt). Die Gesamtkomplexitäten der erarbeiteten Programme sind praktisch gleich.

4 Analysieren der Aufgabe "Liste"

Eine einfach verkettete Liste ist mit Hilfe eines Arrays zu realisieren. Je Listenelement ist ein Buchstabe als Inhalt und ein Integer-Wert als Verweis auf das nächste Element zu speichern (Genaueres siehe [1]). Die folgenden drei objektorientierten Varianten wurden implementiert und bewertet:

– **Q**uick **a**nd **D**irty: zwei eindimensionale Arrays des jeweiligen Datentyps

– **2D**-Array: ein zweidimensionales string-Array mit Typumwandlung (da negative Werte möglich sind)
– **S**low **a**nd **C**lean: ein eindimensionales struct-Array

Die Unterschiede in den drei Musterlösungen sollen am Beispiel der `search`-Methode deutlich gemacht werden, welche die Position des gesuchten Elements als Rückgabewert liefert bzw. -1, wenn das Element in der Liste nicht enthalten ist; siehe die Tabellen 6, 7 und 8.

Code	SC	EC	DC
`int liste::search(string element)`	nicht bewertet		
`{`			
`1 int finished=anfang;`	1	0	4
`2 while (finished!=1)-`	3	2	1
`{`			
`3 if (inhalt[finished]==element) return finished;`	9	3	7
`4 else finished=zeiger[finished];`	2,25	2	4
`}`			
`5 return -1;`	2	0	0
`}`			
Summe	17,25	7	16
CC	40,25		

Table 6. Lösungsvariante QaD.

Code	SC	EC	DC
`int liste::search(string element)`	nicht bewertet		
`{`			
`1 int finished=anfang;`	1	0	4
`2 while (finished!=1)-`	3	2	1
`{`			
`3 if (zweidfeld[finished][0]==element) return finished;`	9	5	7
`4 else finished=inInt(zweidfeld[finished][1]);`	9	6	5
`}`			
`5 return -1;`	2	0	0
`}`			
Summe	24	13	17
CC	54		

Table 7. Lösungsvariante 2D.

Unterschiedlich sind jeweils die Zeilen 3 und 4. Die SaC-Lösung hat gegenüber der QaD-Lösung wegen der Verwendung des Punktoperators eine höhere Ausdruckskomplexität. Die zusätzliche Kapselung bei QaD macht sich bei der Datenkomplexität bemerkbar. Bei der 2D-Realisierung erfolgt eine doppelte Indizierung,

Code	SC	EC	DC
`int liste::search(string element)`	nicht bewertet		
`{`			
`1 int finished=anfang;`	1	0	4
`2 while (finished!=1)-`	3	2	1
`{`			
`3 if (eindfeld[finished].value==element) return finished;`	9	4	10
`4 else finished=eindfeld[finished].next;`	2,25	3	7
`}`			
`5 return -1;`	2	0	0
`}`			
Summe	17,25	9	22
CC	48,25		

Table 8. Lösungsvariante SaC.

was eine Verdopplung der Bewertung zur Folge hat. Zusätzlich erfolgt ein Funktionsaufruf bzw. eine Schachtelung zur Datentypkonvertierung, woraus eine höhere Bewertung folgt. Wegen der höheren Anzahl an Fehlerquellen und der damit erhöhten Anforderung an die Konzentration beim Implementieren wird auch diese Unterscheidung als gerechtfertigt angesehen. Die Bewertungen der vollständigen Musterlösungen sind der Tabelle 9 zu entnehmen.

	QaD	2D	SaC
Anweisungskomplexität	339,5	377	338,5
Ausdruckskomplexität	200	258	221,5
Datenkomplexität	308	312	361
Gesamtkomplexität	847,5	947	921

Table 9. Bewertung von drei Musterlösungen zur Aufgabe "Liste".

Die drei Musterlösungen unterscheiden sich im Quellcode in nur 19 Zeilen. Die meisten Zeilen sind gleich, die drei Hauptprogramme sind sogar vollständig gleich. Im Hauptprogramm waren das Wort "Informatik" buchstabenweise in der Datenstruktur zu speichern und alle Funktionen daran zu testen. Der insert-Aufruf erfolgt also zehnmal hintereinander. Man könnte dies im Sinne von copy-and-paste abschwächend in der Komplexität berücksichtigen. Rechenberg trifft dazu keine Aussage und die Codezeilen sind deshalb in der Bewertung auch voll berücksichtigt. Im Endergebnis unterscheiden sich die günstigste und die ungünstigste Variante in der Gesamtkomplexität um 11,7 %. Wenn man es dem Prüfungsteilnehmer überlässt, mit welcher Datenstruktur er die Aufgabe löst, muss man damit rechnen, dass sich die Gesamtkomplexitäten des entwickelten Quellcodes in der Größenordnung 10% voneinander unterscheiden. Diese Größenordnung erscheint auch in einer Abiturprüfung vertretbar.

5 Ausblick

Mit dem vorliegenden Aufsatz wurde ein Verfahren vorgestellt, mit dem sich die softwaretechnische Komplexität von einfachen objektorientierten Programmen berechnen lässt. Die Verwendung von weiteren Grundkonzepten über Klasse und Objekt hinaus würde der Komplexität eines objektorientierten Programms gegenüber eines prozeduralen eine neue Qualität verleihen. Die Rechenbergschen Festlegungen sind in dem Fall nicht unmittelbar anwendbar. Dies könnte Gegenstand weiterer Untersuchungen sein.

Die für diesen Aufsatz erarbeiteten Musterlösungen und deren Komplexitäten liefern Hinweise darauf, dass es Prüfungsteilnehmern freigestellt werden kann, ob sie prozedural oder objektorientiert programmieren und welche Datenstruktur von ihnen eingesetzt wird (sofern die Datenstruktur als prinzipiell geeignet anzusehen ist). Die Komplexitäten der jeweiligen Programme unterscheiden sich nur in einer akzeptablen Größenordnung.

Nachdenklich stimmt der Unterschied in den Gesamtkomplexitäten von Programmen zu den beiden Abituraufgaben. Die Programme zum "Plumpsack" besitzen eine Gesamtkomplexität von ungefähr 360, die Programme zur Liste eine Gesamtkomplexität von rund 850-950, obwohl mit beiden Aufgaben die gleiche Anzahl von Bewertungseinheiten erreichbar war (jeweils 30 BE von 60 BE für die gesamte Abiturpüfung). Vor vorschnellen Urteilen ("Die eine Aufgabe war viel schwerer und verlangte auch viel mehr Bearbeitungsaufwand als die andere.") ist jedoch zu warnen, denn bei der Berechnung der Gesamtkomplexität wird nur das Endprodukt eingeschätzt. Schwierigkeiten und Aufwand in der Phase der Modellierung werden nicht bewertet. Selbstverständlich können die unterschiedlichen Komplexitäten Anlass zur Kommunikation zur Schwierigkeit und dem Programmieraufwand von Abituraufgaben sein.

References

1. Freistaat Thüringen: Abiturprüfung 2007 Leistungsfach Informatik (Haupttermin).
2. Freistaat Thüringen: Abiturprüfung 2008 Leistungsfach Informatik (Haupttermin).
3. Fothe, Michael: Kunterbunte Schulinformatik - Ideen für einen kompetenzorientierten Unterricht in den Sekundarstufen I und II. LOG IN Verlag, Berlin 2010.
4. Gloeckner, Christian: Entwurf und Implementierung elementarer Algorithmen im Informatik-Unterricht der Sekundarstufen I und II. Wissenschaftliche Hausarbeit zum ersten Staatsexamen an Gymnasien an der Friedrich-Schiller-Universität Jena (2016).
5. Rechenberg, Peter: Ein neues Maß für die softwaretechnische Komplexität von Programmen. In: Informatik Forschung und Entwicklung (1986) 1: 26-37.
6. Thüringer Ministerium für Bildung, Wissenschaft und Kultur: Lehrplan für den Erwerb der allgemeinen Hochschulreife Informatik (2012).