Ein Beitrag zum Seminar
Didaktische Grundfragen der Informatik

Software Engineering

Jens R. Calamé
calame@cs.uni-potsdam.de

6. Februar 2003

Thematik: Didaktische Grundfragen der Informatik, Software Engineering, Software Design Patterns

Schlüsselworte: Software Engineering, Definition, Einordnung, Entstehung, Software Design Patterns

Version: Revision: 1.6 – Date: 2003/02/06 15:07:20

Inhaltsverzeichnis

1. Einleitung **1**

2. Entstehung, Definition und Einordnung **2**

2.1. Die Entstehung der Disziplin 2

2.2. Definition der Disziplin 3

2.2.1. Der Aufbau des Software Engineering Book of Knowledge 3

2.2.2. Die Wissensgebiete des Software Engineering 5

2.2.3. Beziehungen des Software Engineering zu anderen Wissenschaften 6

3. Die Grundproblematik und eine Lösung **7**

3.1. Die Grundproblematik des Software Engineering 7

3.2. Eine Lösung: Pattern 8

3.3. Die Arbeit mit Pattern 9

3.4. Ein Beispiel für die Arbeit mit Pattern 9

3.4.1. Naive Lösung 9

3.4.2. Lösung mit dem MVC-Pattern 10

3.5. Bewertung 11

A. Quelltexte **15**

A.1. Beispiele zu: Software Engineering als „Textverarbeitung” 15

A.2. Beispiele zu: Die verschiedenen Arten, das „Rad zu erfinden” 16

A.3. Beispiele zu Pattern 16

A.3.1. Keine Anwendung von Pattern 16

A.3.2. Anwendung des MVC-Pattern 18

Abbildungsverzeichnis **22**

Literaturverzeichnis **24**

1. Einleitung

Ein Programm zu schreiben bedeutet,
einer Welt Gesetze zu geben, die man zunächst in seiner Phantasie erschaffen muss.
(J. Weizenbaum in: *Die Macht der Computer und die Ohnmacht der Vernunft*)

Die vorliegende Arbeit befasst sich im Rahmen des Seminars *Didaktische Grundfragen der Informatik* mit der Informatikdisziplin *Software Engineering*. Zunächst erfolgt eine Definition dieser Disziplin sowie die Einordnung in verschiedene Wissenschaften. Im Anschluss daran wird auf die Entstehung des *Software Engineering* eingegangen, wobei zugleich die Grundprobleme herausgearbeitet werden. Für diese Probleme soll dann eine Lösung angeboten werden, wobei hier auf die besondere Rolle von *Software Pattern* eingegangen wird.

Im Literaturverzeichnis ist die für diese Arbeit verwendete Literatur aufgeführt. Bezüglich der Einordnung der Fachdisziplin sei hier insbesondere auf [Abr00] und [BR02] verwiesen, [NR69] und [BR70] sind die Proceedings der Konferenzen, die die Geburtsstunde des *Software Engineering* nachzeichnen. Die grundlegenden Probleme der Disziplin sind [Nau92], aber auch den vorgenannten Proceedings zu entnehmen. Schlussendlich stellen [Ale77], [Gam95] und [Vli99] die grundlegende Literatur bezüglich der Lösung dieser Probleme durch Pattern dar. Zusätzlich zur gedruckten Literatur sind mit [Cet] und [Hil] einige weiterführende Web-Ressourcen angegeben.

Die in der Arbeit vorgestellten Quelltexte sind mithilfe des *GNU C++ Compilers, Version 2.95.3* (C++), des *SUN Java SDK/SE, Version 1.3.1* bzw. mit *SWI-Prolog, Version 4.0.9* übersetzbar.

2. Entstehung, Definition und Einordnung

2.1. Die Entstehung der Disziplin

Die Wissenschaftsdisziplin *Software Engineering* hat ihre Wurzeln in der Softwarekrise der 1960er Jahre. Es bildete sich damals ein freier Markt für Hard- und Software, die bis dahin auf den militärischen und universitären Bereich beschränkt war. Dieser Markt bildete sich, da Hardware zum damaligen Zeitpunkt immer leistungsfähiger und preiswerter wurde. Durch die steigende Leistungsfähigkeit wuchsen jedoch die Möglichkeiten, immer komplexere Probleme mit Software zu lösen, deren Komplexität infolgedessen ebenfalls zunahm [Dij72].

Zusätzlich zum Problem der komplexer werdenden Software vergrößerte sich der Anwenderkreis. Um diesen bedienen zu können, waren jedoch in zunehmendem Maße Spezialisten gefragt, deren Ausbildung jedoch – mangels bisheriger Erfahrung – noch keine gesicherten Grundlagen besaß. Dies führte in Verbindung mit der zu bewältigenden Menge an Aufträgen und der bislang kaum vorhandenen Erfahrung mit der Architektur und Entwicklung moderner, großer Softwaresysteme zu Ad-hoc-Entwicklungsmethoden, die einerseits fehleranfällig waren, andererseits aber auch die Wartung und Weiterentwicklung der Software erschwerte. Zum anderen war aber auch das Design der Rechner ab der Mitte der 1960er Jahre dergestalt vom Preis bestimmt, dass bereits hierdurch die Programmierung von Software zusätzlich erschwert wurde (ebd.).

Um diesen Missständen entgegenzutreten wurde im Oktober 1968 die erste Konferenz über Software Engineering des *NATO Science Committee* in Garmisch-Partenkirchen einberufen [NR69]. Auf dieser ersten Konferenz wurde unter anderem die Beziehung zwischen Software und Hardware von Computern diskutiert. Ein weiteres Thema war das Design von Software, das bis zum Anfang der 1960er Jahre aufgrund der geringen Komplexität der Programme (gemeint ist hierbei die Komplexität der Programmstruktur, nicht diejenige der verwendeten Algorithmen) eine nur äußerst untergeordnete Rolle gespielt hatte, nun jedoch zu essenzieller Bedeutung heranwuchs. Dies zeigt sich im übrigen nicht nur an der „Programmierung im Großen", also der Entwicklung von Software aus vorgefertigten und wiederverwendbaren Teilsystemen, sondern bereits in der Programmierung der einzelnen Algorithmen. So waren die bis dato zur Anwendung gebrachten, relativ unstrukturierten Programmierstile ebenfalls zu überdenken, worauf Dijkstra in seinem Artikel *Goto Considered Harmful* (CACM 11(2), S. 147f., nachgedruckt in [BD02]) 1968 hinwies.

Auch die Produktion von Software wurde in der Konferenz thematisiert, wobei bereits zur damaligen Zeit deutlich wurde, dass die fehlende Trennung von *Konstruktion* und *Produktion* von Software zu dem Problem führt, dass bezüglich der notwendigen Konstruktions- und Produktionsprozesse keine Anleihen an anderen Ingenieurwissenschaften genommen werden konnten. Software ist – im Gegensatz zu bspw. Maschinen – ein immaterieller Gegenstand, der ausschließlich ein komplexes Verhalten definiert. Die Schwierigkeit im Verständnis von Software liegt in ihrer Abstraktheit und zeigt sich besonders deutlich im Verständnistransfer von Quelltext auf das beschriebene Verhalten und vice versa (siehe Abschnitt 3.1 sowie [Nau92]).

Nicht zuletzt wurden auch wirtschaftliche Aspekte der Softwareindustrie diskutiert, die sich auf den Vertrieb von Software und notwendige Serviceleistungen bezogen. Hier wird besonders deutlich, dass die Einbeziehung der späteren Benutzer von Software in den Softwarekonstruktionspro-

zess unbedingt erforderlich ist, jedoch zum damaligen Zeitpunkt nicht stattfand [NR69, S. 22/alt; S. 11/neu][1].

Auf einer zweiten Konferenz ein Jahr später in Rom kamen dann Themen wie Spezifikation von Software, Portabilität, Adaptibilität sowie Software-Qualitätssicherung zur Sprache [BR70]. Auf diesen beiden Konferenzen entstand die Wissenschaftsdisziplin *Software Engineering*.

2.2. Definition der Disziplin

Die Frage nach der Definition der Disziplin *Software Engineering* wird unterschiedlich beantwortet. In [Abr00] wird zu diesem Zweck die IEEE-CS zitiert: „(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)." [Abr00, S. 1-1]. Dies bedeutet, dass zum Software Engineering sowohl die *Anwendung* von systematischen, bewährten Methoden zur Entwicklung und zum Betrieb von Software, als auch die *Lehre* dieser Methoden gezählt werden. Die Begründung dazu findet sich in den Erfahrungen aus der im vorangegangenen Abschnitt beschriebenen *Softwarekrise*. Zugleich ist jedoch anzumerken, dass es sich bei der hier gegebenen Definition um nur eine von vielen handelt. Es ist ein Problem dieser Disziplin, dass verschiedene Ansichten – insbesondere Seitens der Industrie – von den einzelnen Teilgebieten des Software Engineering existieren. Aus diesem Grund versuchen IEEE-CS und ACM innerhalb des *Software Engineering Coordination Committee* (SWECC), mit Unterstützung der Industrie eine definitive Beschreibung der Disziplin und vor allem auch des zugehörigen Lehr-Curriculums zu schaffen. Ein Ergebnis dieser Arbeit wird der *Software Engineering Body of Knowledge* (SWEBOK) sein, dessen Hauptdokument *Guide to the SWEBOK* [Abr00] die derzeit aktuelle Vorversion 0.7 erreicht hat, die oben bereits zitiert wurde und im weiteren Kapitel vorgestellt werden soll.

2.2.1. Der Aufbau des Software Engineering Book of Knowledge

Der *Software Engineering Body of Knowledge* besteht aus vier verschiedenen Produkten. Aufgabe des *Guide to the SWEBOK* ist, Wissensgebiete innerhalb des Software Engineering herauszuarbeiten, sie mit verwandten Gebieten anderer Wissenschaften zu verknüpfen sowie Themen für diese Gebiete zu benennen. Parallel dazu wird der Berufsethos im *Code of Ethics* [SEC98] festgelegt. Dieser ist eine Kollektion von Prinzipien für die Praxis, die garantieren sollen, dass in den vielfältigen Verflechtungen der Disziplin mit anderen Disziplinen sowie der Gesellschaft keine Benachteiligung für einzelne auftritt. Dieser Kodex wird der weiteren Entwicklung des Software Engineering kontinuierlich angepasst[2] und um Erfahrungen aus der Praxis angereichert werden.

Diese beiden Dokumente bilden nun die Basis für weitere Arbeiten, die sich zum einen mit Richtlinien für Softwareingenieure, den sog. *Performance Norms*, befassen, zum anderen mit der Ausarbeitung eines modellhaften Curriculums für die Basisausbildung, Richtlinien für die Akkreditierung [SEA98] sowie die Einrichtung einer zentralen Website für die Ausbildung.

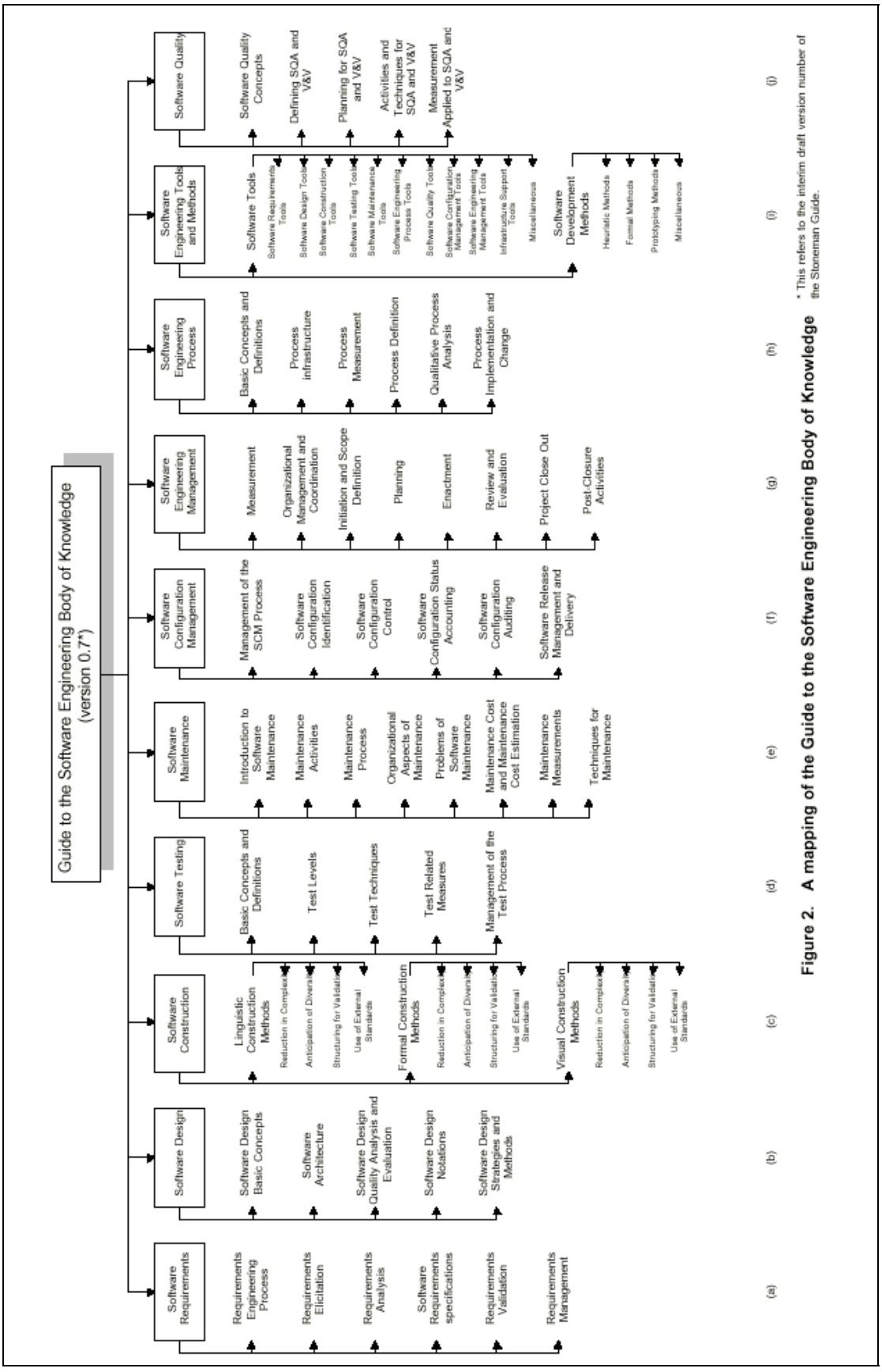

Figure 2. A mapping of the Guide to the Software Engineering Body of Knowledge

Abbildung 2.1.: Die Wissensgebiete des SWEBOK (aus dem *Guide to the SWEBOK*)

2.2.2. Die Wissensgebiete des Software Engineering

Der *Guide to the SWEBOK* ist eine Sammlung von Dokumenten, die jeweils eine Einführung in eines der durch das SWECC erarbeiteten Wissensgebiete des Software Engineering darstellen. Diese Wissensgebiete sind in Abbildung 2.1 dargestellt sowie im folgenden aufgeführt:

Software Requirements: Dieses Gebiet befasst sich mit der Erarbeitung, Analyse, Spezifikation und der Verwaltung von Anforderungen an Software.

Software Design: Das Design von Software ist der Prozess in der Software-Entwicklung, bei dem die Anforderungen an die Software – also das *Was* – in eine Beschreibung der Problemlösungsstrategie – das *Wie* – transformiert wird.

Software Construction: Die Konstruktion von Software ist derjenige Prozess, bei dem die vorliegende Problemlösungsstrategie tatsächlich durchgeführt wird. Ergebnis dieses Prozesses ist dann die lauffähige Software.

Software Testing: Beim Softwaretest wird verifiziert, ob die entwickelte Software tatsächlich das gegebene Problem korrekt und vollständig löst. Dabei wird versucht, die Fehlerfreiheit der Software zu widerlegen, wobei die Fehlerfreiheit angenommen wird, wenn diese Widerlegung fehlschlägt.

Software Maintenance: Dieses Wissensgebiet befasst sich mit der Aufrechterhaltung des laufenden Betriebs von Software.

Software Configuration Management: Unter einer Softwarekonfiguration wird die Menge aktuell installierter und sich im laufenden Betrieb befindlicher Software- und Systemkomponenten verstanden. Beim Konfigurationsmanagement geht es im Speziellen darum, die Integrität eines Softwaresystems über seinen gesamten Lebenszyklus hinweg zu wahren.

Software Engineering Management: Dieses Gebiet befasst sich mit der Planung, Durchführung und Bewertung von Software-Entwicklungsprozessen.

Software Engineering Process: Hierbei wird die Definition, Implementation, Bewertung, das Management und die Verbesserung von Softwareprozessen betrachtet.

Software Engineering Tools and Methods: Dieses Gebiet vereinigt eigentlich zwei Wissensgebiete auf sich: *Werkzeuge* und *Methoden*. Ersteres befasst sich mit dem Design und Einsatz von Werkzeugen zur Unterstützung von Softwareentwicklungsprozessen, letzteres mit der Vorgehensweise in solchen Prozessen.

Software Quality: Dieses scheinbar mit dem *Software Testing* kollidierende Wissensgebiet umfasst nicht die beim Softwaretest angewandten Techniken (die ihrerseits tatsächlich dem Gebiet *Software Testing* angegliedert sind), sondern vielmehr den Prozess des Testens im Speziellen und der Softwarequalitätssicherung im Allgemeinen.

[1] Die Begriffe *alt* bzw. *neu* beziehen sich auf die Seitennummerierung in den *Original*-Proceedings bzw. in der Neuauflage von 2001.

[2] Zumindest *soll* er kontinuierlich dem sich ändernden Umfeld angepasst werden, allerdings impliziert dann das Veröffentlichungsdatum der bislang aktuellsten Version (1998), die Disziplin habe sich seit nunmehr fünf Jahren nicht weiterentwickelt – was mit Sicherheit ein Trugschluss ist.

2.2.3. Beziehungen des Software Engineering zu anderen Wissenschaften

Es existieren internationale Bestrebungen, das Software Engineering den Ingenieurswissenschaften anzugliedern. In [BR02] wird es daher auf eine Ebene mit anderen Wissenschaften wie Elektrotechnik oder Maschinenbau gehoben. Ähnlich wie diese basiert das Software Engineering auf verschiedenen Grundlagenwissenschaften, wobei insbesondere die Informatik, Betriebswirtschaft und die Psychologie genannt werden. Die Mathematik, als eine Grundlagenwissenschaft für die Informatik, wird dort übrigens nicht etwa als „indirekte Grundlagenwissenschaft", sondern – etwas abwertend – als „Hilfswissenschaft" aufgeführt.

Die Informatik sollte dabei die größte Rolle spielen, da das Produkt des Software Engineering gerade Software ist, mit der sich weite Teile der Informatik ebenfalls grundlegend beschäftigen. Die Betriebswirtschaft hingegen ist bereits eine Beziehung zu den nicht-technischen (und hier auch nicht-*militärischen*) Anwendern dieser Software. Außerdem sind betriebswirtschaftliche Gegebenheiten und Prozesse bei der Entwicklung von Software nicht nur für den Auftraggeber abzubilden, sondern auch praktisch anzuwenden (bspw. bei der Preisfindung für die entwickelte Software). Die drittgenannte Grundlagenwissenschaft, die Psychologie, spielt eine herausragende Rolle bei der Entwicklung von Benutzerschnittstellen für Software. An dieser Stelle sei jedoch statt weiterer Ausführung auf Fachliteratur wie [Wan93] und [Bal00] verwiesen.

Auch der *Guide to the SWEBOK* führt einige mit dem Software Engineering in Beziehung stehende Wissenschaften auf, jedoch etwas pointierter und ohne Hierarchisierung (was der Mathematik hier die Degradierung erspart). Die aufgeführten Wissenschaften sind dabei:

- Informatik,
- Mathematik,
- Projektmanagement,
- Computer- und
- System-Ingenieurswesen,
- Management sowie
- Kognitionswissenschaften/Psychologie.

Dabei fällt auf, dass die Betriebswirtschaft als solche nicht genannt wird, sondern sich vielmehr auf zwei Teilbereiche, Projektmanagement und allgemeines Management, beschränkt, die für die Durchführung von Softwareentwicklungsprojekten ohne Betrachtung bspw. des Verkaufs von Software wesentlich sind. Die Analyse und Abbildung von betriebswirtschaftlichen Prozessen fällt hier in den Bereich des System-Ingenieurswesens.

3. Die Grundproblematik und eine Lösung

3.1. Die Grundproblematik des Software Engineering

In [Nau92] diskutiert Naur die Frage, ob die Entwicklung von Software reiner Textverarbeitung gleichkäme. Diese Diskussion sei an dieser Stelle exemplarisch verdeutlicht. Das erste Beispiel ist sinnleerer Text, wie er durch reine Textverarbeitung entsteht:

Blah blah blah blah blah Blah Blah bla bla Bla blah blah Blah Blah bla Bla
Blah blah blah blah blah Blah Blah bla bla Bla blah blah Blah Blah bla Bla
Blah blah blah blah blah Blah Blah bla bla Bla blah blah Blah Blah bla Bla
Blah blah blah blah blah Blah Blah bla bla Bla blah blah Blah Blah bla Bla
Blah blah blah blah blah Blah Blah bla bla Bla blah blah Blah Blah bla Bla
Blah blah blah blah blah Blah Blah bla bla Bla blah blah Blah Blah bla Bla
Blah blah blah blah blah Blah Blah bla bla Bla blah blah Blah Blah bla Bla
Blah blah blah blah blah Blah Blah bla bla Bla blah blah Blah Blah bla Bla
Blah blah blah blah blah Blah Blah bla bla Bla blah blah Blah Blah bla Bla

Betrachtet man nun das folgende Beispiel, so ist – bis auf die Auswahl des Fonts – zunächst scheinbar kein Unterschied erkennbar:

```
...
Blah BLah blah __blah Blah BLah _blah blah__ __Blah
   _Blah __blah
     _blah BLa BLAh bLA
        blah __blah _blah bLa BLAh blah__ blA _blah BlA
        BLAh
   blah__ BLA
Blah__

BLah _BLAH __blah blah__ __Blah
   _BLah Bla blah __blah BLAH blah__ Bla _BLAh BLA
   _Blah bla BLA
Blah__
```

Dies ist jedoch ein Trugschluss. Vielmehr ist der zweite Text ein Computerprogramm[1], das einen Algorithmus (die Fakultätsfunktion) beschreibt. Das Beispiel ist bewusst so gewählt, dass es unmöglich ist, einen Rückschluss auf seine Funktionalität zu ziehen, um Naurs These aus [Nau92] demonstrieren zu können: *Software ist ein Gedankengebäude ihres Entwicklers.*

Diese Erkenntnis hat weitreichende Folgen: Zunächst kennt *nur* der Entwickler einer Programmkomponente selbst, wie sie sich im Programmablauf verhält. *Andere* Entwickler müssen sich zur

[1]Dieses Programm ist – um einige Definitionen ergänzt – tatsächlich compilierbar und lauffähig. Es ist vollständig in Listing A.1 abgedruckt, die Auflösung findet sich in Listing A.2.

Pflege dieses Programmteils zunächst einarbeiten, um dieselbe Kenntnis über das Programmverhalten zu erlangen wie der ursprüngliche Programmierer selbst. Dieser Prozess des – wie Naur es nennt – *Theory Building* ist jedoch aufgrund der Abstraktheit von Software sehr schwierig und fehlerträchtig. Besonders erschwerend wirkt dabei die Tatsache, dass dasselbe Problem auf unterschiedliche Weisen gelöst werden kann. Für eine beispielhafte Darstellung sei dabei auf die „Erfindung des Rades" in den Listings A.3 und A.4 verwiesen, bei der es darum geht, ein „Rad" zu definieren und damit zu arbeiten[2]. Wie leicht zu erkennen ist, unterscheiden sich beide Lösungen in ihrer Umsetzung beträchtlich.

3.2. Eine Lösung: Pattern

Aus dem Bereich der Architektur stammt eine Lösung dieser Grundproblematik: der Einsatz von *Pattern*, englisch für *Muster*. Die Idee geht auf den Architekten Christopher Alexander zurück, der in den 1970er Jahren einen ersten Katalog von Lösungsvorschlägen für den architektonischen Bereich erstellte [Ale77]. Ein Pattern ist dabei eine generalisierte Beschreibung von Problemen und einer möglichen Lösung. Es vereinheitlicht einerseits die Lösungsansätze für Klassen von Problemen, andererseits das Vokabular bei der Beschreibung einer bereits existierenden Lösung. In [NB02] wird dazu folgendes Beispiel aufgeführt[3]: Statt eine Lösung mit den Worten „eine Tür unter der Treppe im Erdgeschoss" zu umschreiben, kann man – dank Alexanders Patternkatalog – die wesentlich präzisere Beschreibung „ein *Cupboard Under The Stairs*" liefern, die direkt ein bestimmtes Pattern referenziert, was seinerseits wieder Rückschlüsse auf das eigentliche Problem zur Lösung zulässt.

Ein *Pattern* besteht mindestens aus einer Problembeschreibung und der Beschreibung einer Problemlösungsstrategie in Form von Handlungsanweisungen. In diesem Punkt unterscheidet es sich vom Algorithmus, der zwar auch eine Lösungsstrategie in Form von Handlungsanweisungen beschreibt, dabei jedoch nicht das Problem als solches beschreibt. Ein Pattern besteht ferner noch aus der Beschreibung möglicher Anwendungskontexte sowie aus Verknüpfungen zu über- und untergeordneten Pattern, d.h. zu Pattern, die auf diesem Muster bestehen, und zu solchen, auf denen das beschriebene Muster selbst basiert. Auf diese Weise können Netzwerke von Pattern entstehen, wie bspw. in [Gam95] dargestellt. Für den einzelnen Anwendungsfall werden Pattern zu sog. *Patternsprachen* zusammengefasst werden.

Jedes Pattern eines Patternkatalogs folgt einem standardisierten Aufbau. In *A Pattern Language* [Ale77] besteht eine Musterbeschreibung aus folgenden Abschnitten:

1. konkretes Beispiel (mit Abbildung)
2. einführender Text
   - Anwendungskontext
   - weiterführende Pattern
3. Beschreibung des Problems
4. Beschreibung eines Lösungsweges
5. ein Lösungsbeispiel (mit Abbildung oder Diagramm)

[2] Dabei soll als nebensächlich angesehen werden, dass die Unterschiede hier auf den verschiedenen Programmierparadigmen der verwendeten Programmiersprachen zurückzuführen sind.

[3] Man sollte sich bei der Lektüre des Beispiels tunlichst nicht davon irritieren lassen, dass an dieser Stelle – sehr „wissenschaftlich" – J. Rowlings „Harry Potter und der Stein der Weisen" zitiert wird...

6. Basispattern

Dieser Aufbau kann je nach Patternkatalog variieren. So wird bspw. in [Gam95] ein Abschnitt mit alternativen Bezeichnungen eines Musters bei Bedarf eingeführt. Auf diese Weise begegnet man dem Problem, dass Pattern zur Lösung desselben Problems durchaus unter verschiedenen Namen bekannt sein können (zum Beispiel das *Observer*-Pattern auch unter dem Bezeichner *Publish-Subscribe*) oder mehrere Pattern sehr ähnliche Probleme lösen, so dass die Entscheidung für ein bestimmtes erschwert wird. Mit dieser Problematik befassen sich ausführlich [NB02] und [MD01].

3.3. Die Arbeit mit Pattern

Die Arbeit mit Pattern lässt sich in drei Teile gliedern: die Definition von Pattern, ihre Auswahl für einen bestimmten Anwendungsfall sowie schlussendlich ihre Anwendung.

Die *Definition* eines Pattern geschieht während der Lösung eines bis dato noch nicht gelösten Problems oder eines Problems, für das bislang noch kein Lösungsweg dokumentiert wurde. Zunächst wird dieses Problem beschrieben und eine exemplarische Lösung dargestellt. Daran schließt sich die Dokumentation des beschrittenen Lösungsweges und möglicher Lösungsalternativen an. Anschließend müssen die Beziehungen zu anderen Mustern hergestellt werden, wobei dies bei den verwendeten Basispattern sofort und bei den weiterführenden Pattern nach dem Sammeln von genügend Erfahrung mit dem neu entwickelten Muster geschieht. Zum Schluss wird das neu beschriebene Pattern in eine geeignete Kategorie eines Patternkatalogs eingeordnet.

Zur *Auswahl* von Pattern für einen Anwendungsfall, d.h. für die Entwicklung einer geeigneten Patternsprache, ist zunächst das Gesamtproblem aufzubereiten. Nun kann man die für die Lösung dieses Problems wesentlichen Muster und rekursiv ihre Basispattern herausarbeiten und der Patternsprache hinzufügen. Ist im Anschluss daran die Abhängigkeitsstruktur dieser Muster aufgebaut, so kann man für jedes Teilproblem geeignete Pattern dem Katalog entnehmen und *anwenden*.

3.4. Ein Beispiel für die Arbeit mit Pattern

Es sei die Aufgabe gestellt, ein Computerprogramm für die Erfassung von Personendatensätzen zu entwickeln. Die Anwendung soll über eine Benutzeroberfläche verfügen, in der Vorname, Nachname und E-Mail-Adresse einer Person eingegeben und geeignet weiterbearbeitet werden können. Eine um die Möglichkeit zur Weiterverarbeitung der eingegebenen Daten und zum Beenden des Programms angereicherte graphische Benutzeroberfläche ist in Abbildung 3.1 dargestellt.

3.4.1. Naive Lösung

Die intuitive objektorientierte Herangehensweise an diese Problemstellung führt zu einer Anwendung, die aus einer Klasse besteht (siehe Abbildung 3.2 und Listing A.5). Diese Anwendung sieht in ihrer Architektur sehr einfach aus, da sie die drei Aufgaben Datenhaltung, Präsentation der Daten und Steuerung der Applikation auf eine Klasse vereinigt. Diese Einfachheit scheint wegen ihrer Übersichtlichkeit zunächst sehr vorteilhaft zu sein, jedoch hat diese Lösung entscheidende Nachteile. So ist der Quelltext des Programms sehr komplex und vermengt die einzelnen Teillösungen für jede der genannten Aufgaben. Dies hat zum einen eine erschwerte Wartung zur Folge, zum anderen verhindert dieser Umstand aber auch die Nachnutzung von Teilen der Lösung für eine abgewandelte Anwendung mit bspw. einer Web-Oberfläche statt des Desktop-Client.

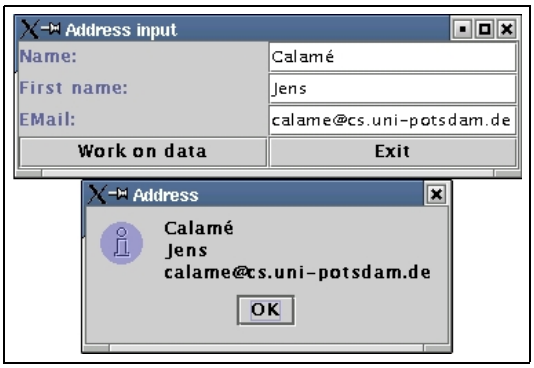

Abbildung 3.1.: Benutzeroberfläche der Applikation

Abbildung 3.2.: Klassendiagramm der naiven Lösung

3.4.2. Lösung mit dem MVC-Pattern

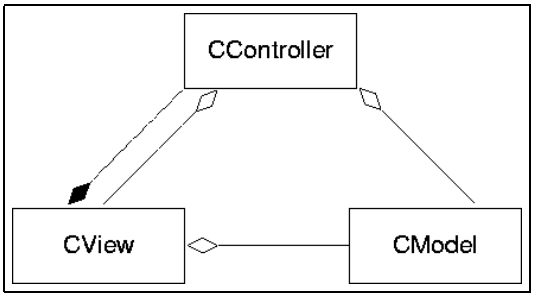

Abbildung 3.3.: Klassendiagramm der Lösung mit MVC-Pattern (nach [HR02])

Diese Nachteile sind so schwerwiegend, dass eine Neuimplementation – nun unter Verwendung des sog. *Model-View-Controller*-Pattern [KP88] – notwendig wird. Das Model-View-Controller-Pattern teilt die o.g. drei Aufgaben auf drei Teilsysteme der Applikation auf: das *Model* (Datenhaltung), die *View* (Präsentation der Daten) und den *Controller* (Anwendungssteuerung). Somit sieht die Architektur der Anwendung wie in Abbildung 3.3 aus. Die Lösung hat den Vorteil, dass die Implementation der Anwendung entzerrt und somit einerseits der erleichterten Wartung und andererseits der Nachnutzung von Teilsystemen zugänglich gemacht wird. Somit ist es nun auch möglich, das Programm um eine Web-Oberfläche zu ergänzen, ohne Controller oder Model reimplementieren zu müssen.

Die Quelltexte sind den Listings A.6, A.7 und A.8 zu entnehmen.

3.5. Bewertung

Bevor nun der Eindruck entsteht, Pattern seien ein Allheilmittel für alle Probleme des Software Engineering, so sollen an dieser Stelle einige Einschränkungen gemacht werden. Es ist nicht sinnvoll, sich bei der Entwicklung neuer Software ausschließlich an vorhandenen Mustern zu orientieren und sich bei der Implementation sklavisch an diese zu halten. So wird in [MIT01] ganz gegenteilig angeraten, eine Implementation zunächst ohne die Verwendung von Pattern durchzuführen, um sie anschließend mit ihrer Hilfe zu optimieren.

Auch Vlissides warnt in [Vli99] mit einem recht drastischen Beispiel vor der falschen Anwendung von Pattern. Er führt ein hypothetisches Muster an für den Anwendungskontext, dass bei der Ziehung der Lottozahlen der eigene Lottoschein einen Hauptgewinn zugeschlagen bekam, und das Problem, dass der eigene Hund eben diesen Schein gefressen hat. Wer nun dem Lösungsweg folgt, das Tier aufzuschneiden, um den Schein entnehmen zu können, erweist sich nicht als Tierfreund.

Nicht zuletzt ist die Auswahl an Pattern für einen Anwendungsfall auch von der Unterstützung dieses Musters durch die verwendeten Entwicklungstools abhängig. Das oben angeführte Beispiel ließ sich zwar mit dem SUN JDK und einem Texteditor recht leicht mit dem MVC-Pattern entwickeln, bei der Nutzung einer Microsoft-Entwicklungsumgebung wie dem Visual Studio wäre allerdings die Wahl auf das *Document-View*-Pattern, bei dem *Controller* und *View* verschmelzen, gefallen. Diese Auswahl wäre damit begründet gewesen, dass die Code-Generatoren und -Frameworks von Microsoft genau dieses und nicht das MVC-Pattern unterstützen. Deshalb lässt sich zusammenfassend mit der Erkenntnis schließen, dass Pattern nur mit Bedacht und nach guter Recherche effektiv einsetzbar sind.

Anhang

A. Quelltexte

A.1. Beispiele zu: Software Engineering als „Textverarbeitung”

Listing A.1: „Verschlüsselter” Quelltext

```
#include <iostream.h>
#define BLAH 4
#define BLAh 1
#define bla 0
#define Blah unsigned
#define BLah int
#define _blah i
#define _Blah return
#define _BLah cout
#define _BLAh endl
#define _BLAH main
#define __blah (
#define blah__ )
#define __Blah {
#define Blah__ }
#define Bla <<
#define BLa >
#define BLA ;
#define bLA ?
#define blA *
#define bLa -
#define BlA :
Blah BLah blah __blah Blah BLah _blah blah__ __Blah
   _Blah __blah
     _blah BLa BLAh bLA
        blah __blah _blah bLa BLAh blah__ blA _blah BlA
        BLAh
   blah__ BLA
Blah__

BLah _BLAH __blah blah__ __Blah
   _BLah Bla blah __blah BLAH blah__ Bla _BLAh BLA
   _Blah bla BLA
Blah__
```

Listing A.2: „Verschlüsselter” Quelltext – Auflösung

```
#include <iostream.h>

unsigned int fak(unsigned int i) {
   return (i>1?fak(i-1)*i:1);
}
```

```
int main() {
   cout << fak(4) << endl;
   return 0;
}
```

A.2. Beispiele zu: Die verschiedenen Arten, das „Rad zu erfinden”

Listing A.3: Die Erfindung des Java-Rades

```
public class Rad {

   public static void main(String[] args) {
      Rad meinRad = new Rad();
   }
}
```

Listing A.4: Die Erfindung des Prolog-Rades

```
rad.

?- rad.
```

A.3. Beispiele zu Pattern

A.3.1. Keine Anwendung von Pattern

Listing A.5: Die Klasse `CIchKannAlles`

```
package src1;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CIchKannAlles extends JFrame implements ActionListener {

   // data
   private String Name;
   private String FirstName;
   private String EMail;

   // view
   private JTextField TName;
   private JTextField TFirstName;
   private JTextField TEMail;

   private JButton ExitApp;
   private JButton DoSomething;

   // Initialization
   public CIchKannAlles() {
      getContentPane().setLayout(new GridLayout(4,2));
      setTitle("Address␣input");
```

```
      // Name
      getContentPane().add(new JLabel("Name:"));

      TName = new JTextField();
      getContentPane().add(TName);

      // FirstName
      getContentPane().add(new JLabel("First name:"));

      TFirstName = new JTextField();
      getContentPane().add(TFirstName);

      // EMail
      getContentPane().add(new JLabel("EMail:"));

      TEMail = new JTextField();
      getContentPane().add(TEMail);

      // Buttons
      DoSomething = new JButton("Work on data");
      DoSomething.setName("Work");
      DoSomething.addActionListener(this);
      getContentPane().add(DoSomething);

      ExitApp = new JButton("Exit");
      ExitApp.setName("Exit");
      ExitApp.addActionListener(this);
      getContentPane().add(ExitApp);

      pack();
   }

   // Actions
   public void actionPerformed(ActionEvent e) {
      Object obj = e.getSource();

      if(obj instanceof JButton) {
         if(((JButton)obj).getName().equals("Work"))
            doSomethingOnData();
         else {
            if(((JButton)obj).getName().equals("Exit"))
               System.exit(0);
         }
      }
   }

   public void doSomethingOnData() {
      setName(TName.getText());
      setFirstName(TFirstName.getText());
      setEMail(TEMail.getText());

      String msg = getName() + "\n" + getFirstName() + "\n" + getEMail();
      JOptionPane.showMessageDialog(this,msg,"Address",JOptionPane.
         INFORMATION_MESSAGE);
   }
```

```
   // Access data
   public void setName(String name) {
      Name = name;
   }

   public String getName() {
      return Name;
   }

   public void setFirstName(String fname) {
      FirstName = fname;
   }

   public String getFirstName() {
      return FirstName;
   }

   public void setEMail(String email) {
      EMail = email;
   }

   public String getEMail() {
      return EMail;
   }

   // main method
   public static void main(String [] args) {
      new CIchKannAlles().show();
   }
}
```

A.3.2. Anwendung des MVC-Pattern

Listing A.6: Das Model

```
package src2;

public class CModel {

   // data
   private String Name;
   private String FirstName;
   private String EMail;

   // Access data
   public void setName(String name) {
      Name = name;
   }

   public String getName() {
      return Name;
   }

   public void setFirstName(String fname) {
```

```
20       FirstName = fname;
   }

   public String getFirstName() {
      return FirstName;
25 }

   public void setEMail(String email) {
      EMail = email;
   }
30
   public String getEMail() {
      return EMail;
   }
}
```

Listing A.7: Die View

```
package src2;

import java.awt.*;
import java.awt.event.*;
5 import javax.swing.*;

public class CView extends JFrame {

   // data
10 private CModel Data;

   // view
   private JTextField TName;
   private JTextField TFirstName;
15 private JTextField TEMail;

   // controller instance linked within the following objects
   private JButton ExitApp;
   private JButton DoSomething;
20
   // Initialization
   public CView(CController controller, CModel data) {
      Data = data;

25    getContentPane().setLayout(new GridLayout(4,2));
      setTitle("Address input");

      // Name
      getContentPane().add(new JLabel("Name:"));
30
      TName = new JTextField();
      getContentPane().add(TName);

      // FirstName
35    getContentPane().add(new JLabel("First name:"));

      TFirstName = new JTextField();
      getContentPane().add(TFirstName);
```

```
40        // EMail
          getContentPane().add(new JLabel("EMail:"));

          TEMail = new JTextField();
          getContentPane().add(TEMail);
45
          // Buttons
          DoSomething = new JButton("Work on data");
          DoSomething.setName("Work");
          DoSomething.addActionListener(controller);
50        getContentPane().add(DoSomething);

          ExitApp = new JButton("Exit");
          ExitApp.setName("Exit");
          ExitApp.addActionListener(controller);
55        getContentPane().add(ExitApp);

          pack();
       }

60     public void updateModel() {
          Data.setName(TName.getText());
          Data.setFirstName(TFirstName.getText());
          Data.setEMail(TEMail.getText());
       }
65  }
```

Listing A.8: Der Controller

```
   package src2;

   import java.awt.*;
   import java.awt.event.*;
5  import javax.swing.*;

   public class CController implements ActionListener {

      CModel Data;
10    CView Presentation;

      public CController() {
         Data      = new CModel();
         Presentation = new CView(this,Data);
15    }

      public void run() {
         Presentation.show();
      }
20
      // Actions
      public void actionPerformed(ActionEvent e) {
         Object obj = e.getSource();

25       if(obj instanceof JButton) {
            if(((JButton)obj).getName().equals("Work"))
               doSomethingOnData();
            else {
```

```
            if(((JButton)obj).getName().equals("Exit"))
30             System.exit(0);
         }
      }
   }

35 public void doSomethingOnData() {
      Presentation.updateModel();

      String msg = Data.getName() + "\n" +
                Data.getFirstName() + "\n" + Data.getEMail();
40    JOptionPane.showMessageDialog(
         Presentation,
         msg,
         "Address",
         JOptionPane.INFORMATION_MESSAGE
45       );
   }


   // main method
50 public static void main(String [] args) {
      new CController().run();
   }
}
```

Abbildungsverzeichnis

2.1. Die Wissensgebiete des SWEBOK (aus dem *Guide to the SWEBOK*) 4

3.1. Benutzeroberfläche der Applikation 10
3.2. Klassendiagramm der naiven Lösung 10
3.3. Klassendiagramm der Lösung mit MVC-Pattern (nach [HR02]) 10

Listings

A.1. „Verschlüsselter" Quelltext 15
A.2. „Verschlüsselter" Quelltext – Auflösung 15
A.3. Die Erfindung des Java-Rades 16
A.4. Die Erfindung des Prolog-Rades 16
A.5. Die Klasse `CIchKannAlles` 16
A.6. Das Model 18
A.7. Die View 19
A.8. Der Controller 20

Literaturverzeichnis

[Abr00] ABRAN, ALAIN ET AL. (HERAUSGEBER): *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Technischer Bericht, Software Engineering Coordinating Committee (IEEE CS/ACM), 2000. Version 0.7, `http://www.swebok.org`.

[Ale77] ALEXANDER, CHRISTOPHER ET AL.: *A Pattern Language*. Oxford University Press, New York, 1977.

[Bal00] BALZERT, HELMUT: *Lehrbuch der Software-Technik – Software-Entwicklung*, Kapitel 16–20, Seiten 483–638. Nummer 2. Spektrum Akademischer Verlag, Heidelberg, Berlin, zweite Auflage, 2000.

[BD02] BROY, MANFRED und DENERT, ERNST (Herausgeber): *Software Pioneers – Contributions to Software Engineering*. Springer, Berlin, Heidelberg, 2002.

[BR70] BUXTON, J.N. und RANDELL, B. (Herausgeber): *Software Engineering Techniques – Report on a Conference sponsored by the NATO SCIENCE COMMITTEE*. North Atlantic Territory Organization, 1970. `http://www.cs.ncl.ac.uk/old/people/brian.randell/home.formal/NATO/`.

[BR02] BROY, MANFRED und ROMBACH, DIETER: *Software Engineering – Wurzeln, Stand und Perspektiven*. Informatik Spektrum, 25(6):438–451, 2002.

[Cet] *Cetus-Links: Architecture & Design: Patterns*. `http://www.cetus-links.org/oo_patterns.html`.

[Dij72] DIJKSTRA, EDSGER W.: *The humble programmer*. Communications of the ACM, 15(10):859–866, 1972.

[Flo95] FLOYD, CHRISTIANE: *Software Engineering: Kritik und Perspektiven*. In: FRIEDRICH, JÜRGEN (Herausgeber): *Informatik und Gesellschaft*, Seiten 238–256. Spektrum Akademischer Verlag, 1995.

[Fri00] FRIEDRICH, MARIO ET AL.: *Efficient Object-Oriented Software with Design Patterns*. In: CZARNECKI, K. und EISENECKER, U.W. (Herausgeber): *GCSE'99*, Nummer 1799 in *LNCS*, Seiten 79–90, Berlin, Heidelberg, 2000. Springer.

[Gam95] GAMMA, ERICH ET AL.: *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[Hil] *Hillside.net*. `http://hillside.net/patterns/patterns.htm`.

[HPI] *Ziele und Aufgaben des Hasso-Plattner-Instituts für Softwaresystemtechnik*. `http://www.hpi.uni-potsdam.de/deu/hpi/index.php?file=ziele.html`.

[HR02] HORN, ERIKA und REINKE, THOMAS: *Softwarearchitektur und Softwarebauelemente*. Hanser, München, Wien, 2002.

[KP88] KRASNER, GLENN E. und POPE, STEPHEN T.: *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*, 1988. `http://www.math.rsu.ru/smalltalk/gui/mvc_krasner_and_pope.pdf`.

[MD01] MCPHAIL, JONATHAN C. und DEUGO, DWIGHT: *Deciding on a Pattern*. In: MONOSTORI, L. ET AL. (Herausgeber): *IEA/AIE 2001*, Nummer 2070 in *LNAI*, Seiten 901–910, Berlin, Heidelberg, 2001. Springer.

[MIT01] *Design Patterns – 6.170 Lectures 12–14*, Oktober 2001. Kursunterlagen des MIT, `http://ocw.mit.edu/6/6.170/f01/lecture-notes/index.html`.

[Nau92] NAUR, PETER: *Programming as Theory Building*. In: *Computing: A Human Activity – Selected Writings From 1951 To 1990*. ACM Press/Addison-Wesley, New York, 1992.

[NB02] NOBLE, JAMES und BIDDLE, ROBERT: *Patterns as Signs*. In: MAGNUSSON, B. (Herausgeber): *ECOOP 2002*, Nummer 2374 in *LNCS*, Seiten 368–391, Berlin, Heidelberg, 2002. Springer.

[NR69] NAUR, PETER und RANDELL, BRIAN (Herausgeber): *Software Engineering – Report on a Conference sponsored by the NATO SCIENCE COMMITTEE*. North Atlantic Territory Organization, 1969. `http://www.cs.ncl.ac.uk/old/people/brian.randell/home.formal/NATO/`.

[SEA98] *Accredition Criteria for Software Engineering*, 1998. `http://www.acm.org/serving/se/Accred.htm`.

[SEC98] *Software Engineering Code of Ethics and Professional Practice, Version 5.2*, 1998. `http://www.acm.org/serving/se/code.htm`.

[Vli99] VLISSIDES, JOHN: *Entwurfsmuster anwenden*. Addison-Wesley, 1999.

[Wan93] WANDMACHER, JENS: *Software-Ergonomie*. Nummer 2 in *Mensch-Computer-Kommunikation: Grundwissen*. Walter de Gruyter, Berlin, New York, 1993.

[Wei78] WEIZENBAUM, JOSEPH: *Die Macht der Computer und die Ohnmacht der Vernunft*. Suhrkamp, 1978.

[YA01] YACOUB, SHERIF M. und AMMAR, HANY H.: *UML Support for Designing Software Systems as a Composition of Design Patterns*. In: GOGOLLA, M. und C. KOBRYN (Herausgeber): *UML 2001*, Nummer 2185 in *LNCS*, Seiten 149–165, Berlin, Heidelberg, 2001. Springer.

Zu den Angaben von Web-Adressen ist anzumerken, dass sich diese täglich ändern können. Die Angaben sind Stand Ende Januar 2003 und es kann keine Garantie gegeben werden, dass sie noch auf die hier referenzierten Dokumente verweisen.