Seminar „Ursachen und Vermeidung von Fehlern in der Softwaretechnik"´

Wintersemester 2005/06

Programmiertips

Barbara Haupt
haupt@inf.fu-berlin.de

31. März 2006

Zusammenfassung

Diese Arbeit stellt eine Reihe von praktischen Leitsätzen (Programmiertips) zur Fehlervermeidung vor. Die hier aufgezählten Tips erfüllen drei Kriterien: Sie führen zu fehlerfreierem Code, sie stellen Prinzipien oder Konzepte dar und sie lassen sich ohne großen Aufwand in die eigene Programmerpraxis übernehmen. Es wurde versucht, jeden Leitsatz in eine möglichst einprägsame Formulierung zu fassen.

Inhaltsverzeichnis

1 Einleitung

Die hier aufgeführten Programmiertips dienen zur Vermeidung von Fehlern bei der Entwicklung von Software und entstammen zum Großteil dem Buch "The Pragmatic Programmer" von Andrew Hunt und David Thomas. Ihre Befolgung hat sich in der Praxis als sinnvoll erwiesen. Es gibt jedoch auch Studien, die belegen, dass einige der genannten Konzepte zu fehlerfreierem Code führen - auf diese wird an entsprechender Stelle verwiesen.

Es wurden die Tips ausgewählt, deren Befolgung fehlerfreieren Code und nicht in etwa schnelleren, optimierteren Code verspricht. Weitere Auswahlkriterien waren:

- Es sollte sich um Prinzipien oder Konzepte handeln. Werkzeuge, die es vereinfachen, diese umzusetzen, werden erwähnt, jedoch nicht genauer beschrieben.
- Die gewonnenen Erkenntnisse sollten sich ohne großen Aufwand in die eigene Programmierpraxis übernehmen lassen.

Die Tips beschränken sich nicht nur auf den Prozess des Programmierens, sondern schließen auch die Einstellung des Programmierers zu seiner Arbeit mit ein. Auf quantitative Richtlinien wird nicht eingegangen. Der Leser sollte daher keine Tips wie "eine Klasse sollte höchstens xxx Zeilen haben" erwarten. Zum einen nennen Hunt und Thomas keine solche Richtlinien, zum anderen bin ich der Meinung, dass es nicht sinnvoll ist, sich an solche Vorgaben zu halten. Besser ist es, sich an Prinzipien zu orientieren, wie z.B. die *Kopplung* zwischen zwei Modulen zu minimieren. Mit eben solchen Prinzipien beschäftigt sich der folgende Text.

Es versteht sich von selbst, dass bei jedem Tip geprüft werden sollte, ob dieser für die eigene Arbeit einen Vorteil bringt oder nicht. Dennoch sei dem Leser empfohlen, die eigenen Denk- und Programmiergewohnheiten kritisch zu hinterfragen.

Es wurde versucht, jeden Tip in eine möglichst bildhafte, einprägsame Formulierung zu fassen. Dies soll dem Leser erleichtern, sich an den jeweiligen Kontext zu erinnnern. Eine Zusammenfassung der Tips findet sich daher am Ende.

2 Programmiertips

2.1 Denke kritisch

Vermeidung von Fehlern beim Programmieren beginnt schon vor dem Schreiben der ersten Zeile, schon bevor der erste Entwurf getätigt oder die erste Anforderung analysiert wurde - sie beginnt bereits mit der eigenen Einstellung. Gehe ich im Grunde schon davon aus, dass es nicht klappen wird? Sehe ich nur Probleme und keine Lösungen? Bemühe ich mich, auf dem neusten Stand des Wissens zu bleiben? Denke ich über meine Arbeit nach, oder bin ich froh, sie möglichst schnell hinter mich gebracht zu haben - egal wie? Signiere ich sie gern mit meinem Namen? Fühle ich mich ständig unter Druck oder meistere ich die Situation? Kommuniziere ich gut mit meinem Team? Traue ich mich, Fragen zu stellen? Denke ich in erster Linie kritisch? Kann ich Situationen realistisch einschätzen? Auch den zeitlichen Aspekt eines Projekts? Einige wichtige Leitsätze, die sich daraus ergeben, seien im folgenden genannt.

THINK!

"Think!" ist das langjährige IBM Firmenmotto. Es zum eigenen Motto zu machen, bedeutet kritisch über die eigene Arbeit nachzudenken und die eigenen Annahmen und deren Herkunft ständig zu hinterfragen. Denn wenn ich mir über meine Annahmen nicht im klaren bin, so werde ich womöglich nicht feststellen, wenn Situationen eintreten, in denen diese nicht mehr gelten. Es ist ferner auch die Aufforderung, pragmatisch zu denken und das zu tun, was am weitesten voran bringt.

ÜBE DICH IM SCHÄTZEN: PI MAL DAUMEN

Die Realität einschätzen zu können, ist eine Fähigkeit, die der Übung bedarf. Sie läßt sich durch nichts anderes erlernen, als durch Erfahrung (Schätzungen beruhen auf Modellbildungen, also Vereinfachungen der Wirklichkeit. Es ist eine Frage, wie angemessen das Modell ausgewählt wird, so dass es zwar vereinfacht, aber noch genau genug ist). Versuchen Sie einmal zu schätzen, wie lange Sie für eine bestimmte Aufgabe brauchen oder wie lange ein Projekt dauern wird, wie lange ein Algorithmus brafür eine bestimmte Eingabe braucht etc. Es ist fast hinfällig zu erwähnen, dass durch Fehlschätzungen Fehler entstehen können (man denke nur daran, wenn Entscheidungen gefällt werden müssen, z.B. ob die eine oder andere CPU für ein bestimmtes Projekt in einer großen Menge eingekauft werden soll).

KÜMMERE DICH UM DEIN HANDWERK

Programmieren kann man als modernes Handwerk betrachten. Nicht ohne Grund gibt es unzählige Programmierbücher, deren Titel mit "The Art of..."beginnen. Ein Handwerker signiert seine Arbeit, die ein Unikat darstellt, weil er etwas von sich selbst hineingelegt hat. Dieser Tip ist daher als Aufforderung zu verstehen, die eigene Arbeit so

zu erledigen, dass man sie abschließend mit einem Gefühl von Stolz signiert, sowie Verantwortung zu übernehmen, das heißt z.B. regelmäßige Sicherungskopien anzufertigen, nach Lösungen für Probleme und nicht nach Ausreden zu suchen und die eigene Arbeit ausgiebig zu testen.

KOMMUNIZIERE

Kommunikation ist ein wichtiges Element in einem Team. Wenn der eine nicht weiß, was der andere tut, so wird das Team nicht nur ineffizient, sondern es können auch Mißverständnisse entstehen, und somit Defekte im Projekt, die eventuell erst zu einem viel späteren Zeitpunkt des Projekts auftreten und dann mit größerem Aufwand behoben werden müssen.

Zusammengefasst: Ein guter Programmierer ist ein realistischer, kritischer Denker, der Verantwortung übernimmt und gut kommuniziert.

Die folgenden Abschnitte befassen sich im Gegensatz zum vorherigen Abschnitt mit Tips, die das Programmieren selbst betreffen. Wie bereits in Abschnitt 1 erwähnt und begründet, wurde mehr Wert auf Prinzipien und Konzepte als auf quantitative Richtlinien gelegt. Ebenso sollten die Tips leicht in die eigene Programmierpraxis zu übernehmen sein.

2.2 Keine Wiederholungen

Wiederholungen sind eine Fehlerquelle. Wenn sich dieselbe Information an zwei verschiedenen Stellen im Programm befindet, dann ist im Grunde nicht die Frage, wie lange Sie behalten werden, wo die jeweils andere Stelle war, sondern wann Sie es vergessen haben werden. Eine Änderung an der einen Stelle, ohne die andere Stelle zu ändern, wird zu Widersprüchen im Programm selbst führen. Wenn es jedoch gelingt, Wiederholungen zu eliminieren, liegt oftmals sogar ein besseres Design als Ergebnis vor.

VERMEIDE WIEDERHOLUNGEN

Beispiel: In Abb. 1 werden zwei Vorschläge für eine Klasse "Line" dargestellt. Im Fall links liegt eine verborgene Wiederholung vor, da die Länge der Linie von ihrem Start und ihrem Endpunkt abhängt. Angenommen, man müsste den Startpunkt der Linie ändern, so würde die Länge nicht automatisch angepasst. Es kann zu widersprüchlicher Information kommen. Im rechten Teil der Abbildung ist das Problem behoben. Allerdings mit dem Nachteil, dass jedesmal, wenn man die Länge der Linie abrufen will, diese neu berechnet wird. Wie ließe sich dies umgehen? [1]

[1] Eine Möglichkeit wäre, eine Boolsche Variable einzuführen, die angibt, ob sich ein Wert geändert hat. Die Länge würde nur dann neu berechnet, wenn die Variable es anzeigt.

```
class Line{
public:
   Point start;
   Point end;
   double length;
};
```

```
class Line{
public:
   Point start;
   Point end;
   double length(){
      return end.getDistanceTo(start);
   }
};
```

Abbildung 1: Wiederholung und Auflösung der Wiederholung

Beispiel: Eine Lieferfirma hat LKWs und Fahrer. Es werden gewisse Lieferrouten befahren. In Abb. 2 ist ein möglicher Entwurf dargestellt, der jedoch nicht normalisiert ist, da sowohl LKW als auch Route einen Fahrer zugeordnet bekommen. Es wird daher nicht von vornherein ausgeschlossen, dass es zu Widersprüchen kommen kann. Angenommen der Fahrer wechselt, weil der eigentliche Fahrer krank ist. Sollte dann nur der Fahrer der Route oder auch der Fahrer des LKWs geändert werden? Normalformen sind ein sehr nützliches Konzept, um Redundanzen zu eliminieren und die Integrität der Daten zu wahren.

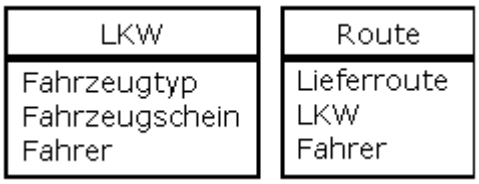

Abbildung 2: Nicht-normalisierter Entwurf

NORMALISIERE DEN ENTWURF

Allerdings verzichtet man selbst im Design von Datenbanken in manchen Fällen auf Normalisierung aus Performancegründen. Wie immer ist also bei jedem Tip abzuwägen, was für die eigene Anwendung dienlich ist, und was nicht.

Kommentare und Quellcode können ebenfalls Wiederholung beinhalten, wenn man nicht beachtet, dass es sich um verschiedene Ebenen der Informationsrepräsentation handelt (in Kommentaren sollte sich stets high-level, im Code low-level Wissen befinden).

Angenommen, Client und Server sind in verschiedenen Sprachen geschrieben, aber es

gibt gemeinsam verwendete Strukturen. Die Lösung, die auf der Hand liegt, wäre, jede Struktur einfach in beiden Sprachen zu repräsentieren. Dies bedeutete aber, dass der selbe Inhalt einfach zweimal dargestellt würde. Änderungen müssten also doppelt vorgenommen werden. Wie sähe eine Lösung des Problems aus, die Wiederholungen vermeidet?[2]

Manchmal wird Wiederholung sogar von der verwendeten Programmiersprache selbst gefordert, wie im Fall von C bzw. C++ - Headerdateien, in denen die Deklaration an einer zweiten, von der Implementierungsdatei getrennten Stelle erfolgt. Eine Möglichkeit unerwünschten Nebenwirkungen, die hierdurch entstehen in diesem schwierigen Fall zu umgehen, besteht z.B. darin, eine Entwicklungsumgebung zu verwenden, die Headerdateien automatisch generiert.

Wenn die Zeit drängt, scheint Wiederholung oftmals eine schnelle Lösung. Aber ein schlechter Entwurf zahlt sich auf Dauer nicht aus. Im Englischen gibt es ein Sprichwort, das den Ungeduldigen warnt: „short cuts make for long delays“. Frei übersetzt: Langsamer ist manchmal schneller!

Die vielleicht am schwierigsten aufzuspürende Art von Wiederholung ist die zwischen einzelnen Entwicklern. Hier hilft vor allem Eines, nämlich zu kommunizieren. Wenn zudem alle auf den Quellcode der anderen zugreifen können, besteht eine gute Basis für eine gemeinsame Entwicklung mit einem hohen Anteil an Wiederverwendung. Source Code Control Systeme wie CVS oder SVN sind Werkzeuge, die dies erleichtern.

MACHE WIEDERVERWENDBARKEIT LEICHT

Wiederholung vermeidet man vor allem dann, wenn es einfach ist, etwas wiederzuverwenden. Ein guter Entwurf, der Wiederverwendbarkeit leicht ermöglicht, ergibt sich, wenn man sich an die folgenden Prinzipien von geringer *Kopplung* (Coupling) und hoher *Bindungskraft* (Cohesion) hält. Diese Begriffe wurden in den 70er Jahren von Stevens, Myers, Constantine und Yourdon im Rahmen des *Structured Programming* geprägt. Das *Structured Programming* ist ein Programmierparadigma, das in erster Linie dafür bekannt geworden ist, dass das Verwenden von GOTO-Befehlen eingeschränkt oder abgeschafft wurde.

2.3 Bindungskraft (Cohesion)

Bindungskraft ist der Grad des inneren Zusammenhalts oder der Zusammengehörigkeit in einem Programm. Der Begriff hat seine Analogie in der Atom und Molekül-

[2]Eine Möglichkeit, dies zu umgehen, wäre, die gemeinsam verwendeten Strukturen auszulagern, und den Code in der jeweiligen Sprache automatisch zu generieren! Dies darf jedoch kein Prozess sein, der auf ein einmaliges Ausführen ausgelegt ist, vielmehr muss dies bei jeder Veränderung des gemeinsam verwendeten Teils geschehen. Am besten automatisch mit jedem Neukompilieren.

physik bei den Kernkräften, insbesondere bei der starken Wechselwirkung. Diese Kernkraft hat nur eine kurze Reichweite und ist diejenige Kraft, die den Zusammenhalt der Atomkerne bewirkt. Man unterscheidet verschiedene Arten von *Bindungskraft*: u.a. sequentielle (wenn bestimmte Operationen in einer Routine in einer ganz bestimmten Reihenfolge ausgeführt werden müssen), temporale (wenn bestimmte Operationen zur selben Zeit ausgeführt werden sollten) oder funktionale (wenn eine Routine eine einzige, ganz bestimmte Funktion hat). Funktionale Bindungskraft ist die beste Art von Bindungskraft. Sie führt zu mächtigen Funktionen.

ENTWERFE FUNKTIONEN,
DIE EINE EINZIGE AUFGABE HABEN

Kopplung ist ein Maß für die Abhängigkeit zwischen zwei Modulen A und B. Wenn man viel über A wissen muss, um B zu verstehen, dann ist die Kopplung zwischen beiden Modulen hoch. Meist geht damit einher, dass das einzelne Modul auch weniger Bindungskraft aufweist. Software, in der die einzelnen Module hohe Bindungskraft und niedrige Kopplung aufweisen ist erwiesenermaßen weniger fehlerträchtig, als solche bei der das Verhältnis von Bindungskraft und Kopplung geringer ist. Siehe dazu [SMC74] und [SB91].

Es lassen sich ebenfalls verschiedene Arten von Kopplung unterscheiden: Statische Kopplung, dynamische und zeitliche Kopplung.

2.4 Statische Kopplung

Statische Kopplung bedeutet, dass ein Stück Quelltext ein anderes Stück Quelltext benötigt, damit es übersetzt werden kann. Statische Kopplung läßt sich daher nicht vermeiden, denn selbst ein einfaches "Hallo Welt"-Programm benötigt in der Programmiersprache C die Datei `stdio.h`. Dennoch sollte man darauf achten, dass man nicht mehr Dateien einbindet, als zwingend notwendig. Vererbung bewirkt ebenfalls starke statische Kopplung. Ändert man eine Elternklasse, so verändert man sämtliche abgeleiteten Klassen. Das kann ein Vorteil sein, kann aber unter Umständen auch ein großer Nachteil sein. Eine Möglichkeit, statische Kopplung bei Vererbung zu umgehen, besteht darin das Entwurfsmuster *Delegation* zu verwenden (siehe [GHJ97]).

2.5 Dynamische Kopplung

Dynamische Kopplung liegt dann vor, wenn ein Stück Code (eine Funktion, eine Routine, eine Instanz, etc.) ein anderes Stück Code zur Laufzeit verwendet. Im Falle, dass dies Ausmaße annimmt, wie in Abb. 3 dargestellt, wird nicht nur der Quelltext unübersichtlich, sondern birgt auch die Gefahr, dass ein Fehler im Programm auftreten wird, wenn sich etwas in der Kette ändert.

```
getOrder().getCustomer().getAddress().getState()
```

Abbildung 3: Starke dynamische Kopplung

Ein Paradigma, an das man sich halten kann, um Kopplung zu minimieren ist *Schüchternheit*. Schüchterner Code gibt zum einen nicht viel von sich selbst Preis, zum anderen ist er aber auch nicht allzu neugierig, was die Interna anderer Module angeht.

Schreibe schüchternen Code

Konkrete Richtlinien für den Entwurf von besser entkoppelter Software, für "schüchternen" Coce, gibt das Law of Demeter (auch Law of Demeter für Funktionen) [Lie04]. In seiner objektorientierten Version lautet es:

Eine Methode M eines Objekts O sollte nur Methoden aufrufen, die

1. zu O selbst gehören
2. zu Parametern gehören, die der Methode M übergeben wurden
3. zu einem Objekt gehören, das M erzeugt hat
4. zu einem Objekt gehören, das in M lokal gehalten wird

Beispiel:

```
public void getPayment(Customer cust){
  Wallet walli = cust.getWallet();
  if (walli.getMoney(0.95)){
    myMoney+=0.95;
    return true;
  }else return false;
}
```

Hält sich die Methode aus dem obigen Beispiel an das Law of Demeter (LoD)? "walli" gehört der Methode **getPayment(...)** nicht. **getPayment(...)** verstößt somit gegen Punkt 4. Eine Lösung, die sich an das LoD hält, würde z.B. so aussehen:

```
public boolean getPayment(Customer cust){
  if (cust.getMoney(0.95)){
    myMoney+=0.95;
    return true;
  }else return false;
}
```

2.6 Zeitliche Kopplung

Die `strtok`-Routine aus der C Library ist ein Beispiel für eine Routine, bei deren Entwicklung Nebenläufigkeit nicht beachtet wurde. Diese Routine verwendet statische Daten um die aktuelle Position im Buffer zu speichern. `strtok` muss zunächst mit dem eigentlichen, zu parsenden Argument aufgerufen werden. Bei allen nachfolgenden Aufrufen muss als Argument NULL übergeben werden. Es ist also nicht möglich, zwei verschiedene Strings gleichzeitig zu parsen. Bevor `strtok` mit einem weiteren String aufgerufen werden kann, muss zunächst der alte String vollständig geparst sein, sonst geht Information verloren.

Die Beachtung von Nebenläufigkeit beim Entwurf von Software resultiert in besserem Design, mehr Flexibilität und weniger Fehleranfälligkeit. Denn kann ich wirklich mit Sicherheit sagen, dass niemals jemand den Status meines Objekts abfragen wird, wenn es sich in einem ungültigen Zustand befindet? Besser ist, wenn eine solche Abfrage gar nicht zugelassen wird.

Zudem kann es sein, dass sich die Anforderungen dahingehend ändern, dass auf einmal Nebenläufigkeit gefordert wird. Dies ist meist schwierig zu realisieren, wenn nicht im voraus beim Entwurf darauf geachtet wurde. Ein System, bei dessen Entwicklung auf Nebenläufigkeit geachtet wurde kann jedoch auch stets als stand-alone Produkt fungieren.

ENTWERFE FÜR NEBENLÄUFIGKEIT

Im den vorangegangenen Abschnitten sind Prinzipien erläutert worden, mit denen sich Fehler beim Programmieren vermeiden lassen. Wann es sich um einen Fehler handelt und wann nicht, muss jedoch festgelegt werden, um Überprüfbarkeit zu ermöglichen. Design by Contract (DBC) ist ein Konzept zu diesem Zweck, das von Bertrand Meyer [Mey97] (ursprünglich für die Programmiersprache Eiffel) entwickelt wurde.

2.7 Design by Contract

Ein Programm ist dann korrekt, wenn es genau das tut was es soll (nicht mehr und nicht weniger). Wenn man vertraglich festlegt, was Rechte und Pflichten des Programms sind, dann hält man mit einem solchen Vertrag ein Konzept in der Hand, das es ermöglicht, zu überprüfen, ob ein Programm korrekt ist oder nicht. Dieses Konzept lässt sich sowohl auf einzelne Funktionen, als auch auf die gesamte Anwendung ausdehen.

ENTWERFE EINE ROUTINE NIE OHNE IHREN VERTRAG

Ein Vertrag mit einem Programm hat drei Teile:

- *Vorbedingung (Precondition):* Die Vorbedingung muss erfüllt sein beim Aufruf der Routine. Es ist die Aufgabe des Aufrufers dies sicherzustellen!

- *Nachbedingung (Postcondition):* Der Status, nachdem die Routine durchlaufen wurde.
- *Invarianten:* Bedingungen, die während des Programms aus Sicht des Aufrufers stets erfüllt sind.

Es ist die Aufgabe des Aufrufers, sicherzustellen, dass die Vorbedingung erfüllt ist. Dennoch ist es sinnvoll innerhalb der Routine mit *Zusicherungen* (Assertions) sicherzustellen, dass dies der Fall ist. Zielsetzung dieses Seminars ist es, Ursachen von Fehlern aufzuspüren, und nicht die Fehler, nachdem sie bereits gemacht wurden. Daher die folgenden Bemerkungen über Zusicherungen der Vollständigkeit halber, jedoch nur am Rande:

Generell sollte man *Zusicherungen* immer dann benutzen, wenn man sicherstellen will, dass etwas, das niemals eintreten kann, auch nicht passiert.

FALLS DU DENKST: DAS KANN NIE PASSIEREN! BENUTZE EINE ZUSICHERUNG.

Da man beim Testen selten alle Fehler findet und außerdem in der wirklichen Welt Dinge passieren können, die man beim Testen einfach nicht bedacht hat, sollte man Zusicherungen nicht ausschalten.

LASSE ZUSICHERUNGEN STETS EINGESCHALTET

Dies vermeidet, dass das Programm mit falschen Daten laufen kann. Falls es aus Performancegründen besser wäre, Zusicherungen auszuschalten, so sollten doch nur diejenigen ausgeschaltet werden, die die größten Laufzeiteinbußen verursachen.

Fehler haben die Tendenz weitere Fehler nach sich zu ziehen. Weist eine Stelle im Programm eine unsaubere und somit fehleranfällige Implementierung auf, wird sich dies sehr wahrscheinlich auf weitere Programmteile auswirken. Vollständige Entkopplung gibt es nie, da ein Programmteil stets irgendwie benutzt werden muss. Besteht nun aber einmal solch ein fehlerhafter Teil, ist es notwendig, ihn möglichst umgehend auszubessern.

2.8 Ordnung pflegen

In diesem Abschnitt wird dies noch einmal bildhaft beschrieben, in dem eine Parallele zu einer in New York in den 80er Jahren entwickelten Theorie zur Eindämmung von Kriminalität gezogen wird, der sogenannten *Broken Window Theory*. Diese Theorie besagt, dass ein Haus, bei dem einige Fenster zerbrochen sind und über längere Zeit nicht repariert werden, leicht dem Vandalismus zum Opfer fallen kann, weil die Hemmschwelle, weitere Fenster einzuschlagen, oder die bereits schmutzigen Wände zu besprühen, niedriger liegt. Das gleiche Phänomen trifft man in Parks an, wenn auf

einem gepflegten Rasen irgendwann die ersten Bierdosen und Zigaretten liegen bleiben und längere Zeit nicht entfernt werden.

Dieses Phänomen lässt sich auf die Entwicklung von Software übertragen: Ein Fehler oder unsauberer Quelltext zieht weitere Fehler nach sich, weil

- sich Indifferenz breit machen kann
- einer allein nicht alles reparieren kann
- eine kaputte Tür (ein Programmteil, z.B. eine Klasse) schwieriger auszutauschen ist, wenn auch der Türrahmen (Programmteile, die den vorherigen verwenden) beschädigt ist.

Dulde keine eingeschlagenen Fenster

Im Gegenzug haben Gegenden, Wohnungen, Parks oder Programme, die einen gepflegten Eindruck machen, auch die Tendenz, weiterhin gepflegt zu werden.

3 Zusammenfassung

Es wurden eine Reihe von Programmiertips vorgestellt, die zu Code geringerer Fehlerdichte führen können. Die ersten vier Tips THINK! - ÜBE DICH IM SCHÄTZEN: PI MAL DAUMEN - KÜMMERE DICH UM DEIN HANDWERK - KOMMUNIZIERE bezogen sich auf persönliche Brennpunkte. Es handelt sich um Themen, die gewöhnlich in der Diskussion um Fehlervermeidung so nicht angesprochen werden, da sie eben in den persönlichen Bereich fallen. Die wissenschaftliche Diskussion darüber wird oftmals in den Bereich der Arbeitswissenschaft ausgegliedert. Ich bin jedoch der Meinung, dass es sich durchaus lohnt, auch im Bereich der Informatik hierüber nachzudenken. Einige der genannten Punkte mögen zwar selbstverständlich sein, da sie Charakteristika für Professionalität sind. Jedoch stellt sich z.B. die Frage, inwiefern man Fehler durch größere Aufmerksamkeit (z.B. gezielt durch geeignete Konzentrationsübungen) vermeiden könnte. Zumindest im Falle der Fehler, die durch Vertippen oder ähnliches geschehen, und die (meist) vom Compiler gemeldet werden, leuchtet dies ein. Von Interesse wäre auch, ob ein Programm, das eine geringere statische Fehlerdichte aufweist, auch insgesamt weniger Defekte hat.

Die folgenden Tips basieren zum Großteil auf Erfahrung. Viele Entwurfsmuster und Programmierparadigmen sind durch die Erkenntnis von Fehlerursachen und den daraus gezogenen Schlüssen zu deren Vermeidung entstanden. VERMEIDE WIEDERHOLUNGEN ist ein nützlicher Leitsatz. Wahrscheinlich hat der Leser selbst schon einmal die Erfahrung gemacht hat, dass Fehler, die durch Inkonsistenzen verursacht werden, schwierig aufzufinden sind und sehr schlechte Auswirkungen haben können. In manch einem Kinofilm wurde eine außer Kontrolle geratene Künstliche Intelligenz durch ein Gedicht oder ein Koan besiegt. Bevor also eine Textpassage per Kopieren und Einfügen wiederholt wird, sollte zunächst einmal eingehalten werden, denn meistens gibt es eine bessere, normalisierte Lösung. NORMALISIERE DEN ENTWURF ist daher ein positiv Formulierter Tip, der aufzeigt, wie Wiederholungen vermieden werden können. MACHE WIEDERVERWENDBARKEIT LEICHT fordert dazu auf, die Wiederholung zwischen einzelnen Entwicklern zu vermindern, in dem z.B. Source-Code-Control-System (SCCS) verwendet wird.

ENTWERFE FUNKTIONEN, DIE EINE EINZIGE AUFGABE HABEN - SCHREIBE SCHÜCHTERNEN CODE - ENTWERFE FÜR NEBENLÄUFIGKEIT sind Leitsätze, die sich aus den Konzepten von hoher Bindungskraft und geringer Kopplung ergeben. Hier sei auf einige Untersuchungen verwiesen, die belegen, dass genannte Konzepte tatsächlich zu fehlerfreierem Code führen. Im Falle von Kopplung und Bindungskraft (Coupling and Cohesion) ist dies die Untersuchung von Selby und Basili 1991 [SB91]. Die Studie von [BBM96] führt ein Maß, das sog. “Responsibility Set” ein, das mit Fehleranfälligkeit korreliert und das durch Code, der sich an das Law of Demeter hält, verringert wird.

Die folgenden Tips bewegen sich bereits am Rande der Zielsetzung des Seminars, bei

der es darum ging, Ursachen von Fehlern aufzufinden, und nicht in etwa Fehler, nachdem sie gemacht wurden. Da jedoch unklar ist, was ein Fehler ist, wenn man dies nicht definiert, wird in diesem Zusammenhang das Konzept *Design by Contract* dargestellt sowie auf die richtige Verwendung von Zusicherungen eingegangen.
ENTWERFE EINE ROUTINE NIE OHNE IHREN VERTRAG - FALLS DU DENKST: DAS KANN NIE PASSIEREN! BENUTZE EINE ASSERTION. - LASSE ASSERTIONS EINGESCHALTET - DULDE KEINE EINGESCHLAGENEN FENSTER ist ein abschließender Ratschlag, Ordnung zu halten, denn Unordnung verbreitet sich zwangsläufig, wenn man nicht eingreift. Das Gesetz der Thermodynamik, dass die Entropie nie abnimmt, gilt leider offensichlich auch für die Softwareentwicklung.

Literatur

[BBM96] V Basili, L Briand, and W L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engeneering*, 22(10):751–61, October 1996.

[GHJ97] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[HT00] A Hunt and D Thomas. *The Pragmatic Programmer*. Adison Wesley, 2000.

[Lie04] K Liebherr. http://www.ccs.neu.edu/research/demeter, 2004.

[Mey97] B Meyer. *Object Oriented Software Construction 2nd Ed.* Prentice Hall, 1997.

[SB91] R Selby and V Basili. Analysing error prone system structure. *IEEE Transactions on Software Engeneering*, SE-17(2):141–52, February 1991.

[SMC74] W Stevens, G Myers, and L Constantine. Structured design. *IBM Systems Journal*, 13(2):115–39, May 1974.