Seminar “Software aus Komponenten“

Sommersemester 2005

Entwurf von wiederverwendbaren Klassen

Olaf Hecht

hecht@inf.fu-berlin.de

Seminarleiter: Christopher Oezbek

7. August 2005

Inhaltsverzeichnis

1 Einleitung

Im Jahr 1988 haben die Wissenschaftler Ralph E. Johnson und Brian Foote der University of Illinois eine Arbeit über das Entwickeln von wiederverwendbaren Klassen mit objektorientierten Programmiersprachen vorgestellt. Der zugehörige Artikel mit dem Titel "Designing Reusable Classes"[JF88] wurde daraufhin im Journal of Object-Oriented Programming veröffentlicht. In dem Artikel beschreiben sie eine Reihe von Techniken, die Softwareentwickler bei der objektorientierten Programmierung verwenden sollten, um wiederverwendbaren Code zu erstellen.

Die Motivation für die Wiederverwendung von Software ergibt sich daraus, dass sie die Entwicklungszeit reduziert[JF88], weil auf existierende Problemlösungen zurückgegriffen wird, und das Rad nicht immer wieder neu erfunden werden muss. Dadurch verringern sich auch die Kosten für die Produktentwicklung.

Ein weiterer Aspekt ist, dass dadurch die Wartbarkeit der erstellten Software erhöht wird.[JF88] Vorteilhaft ist, dass wiederverwendete Komponenten meistens zuverlässiger als neue Komponenten sind. Sie wurden bereits in verschiedensten Umgebungen angewendet und sind in diesen auch schon getestet worden.[Som01] Dadurch ist es in vielen Fällen, wie z.B. bei der Wiederverwendung von Java-Bibliotheken aus dem JDK, möglich, sich auf die Korrektheit der verwendeten Softwarebausteine zu verlassen. Das hat zur Folge, dass auch das Testen eines mit Hilfe von Wiederverwendung entwickelten Systems weniger kostenintensiv wird.

Für die Erstellung von wiederverwendbarem Code ist die objektorientierte Programmierung besser geeignet als die nicht-objektorientierte Programmierung. Ein hoher Grad an Wiederverwendbarkeit ergibt sich durch den Einsatz von Objektorientierung aber nicht zwangsläufig. Vielmehr müssen hierfür spezielle Techniken angewandt werden. Einige von diesen werden im zweiten Kapitel dieser Arbeit vorgestellt.

Für die Wiederverwendung von Software im größeren Stil muss nicht nur darauf geachtet werden, dass die einzelnen Klassen für die Wiederverwendbarkeit geeignet entworfen werden. Sondern es bedarf auch einem wohlüberlegten Aufbau der Klassenhierarchie und der Wahl einer geeigneten Interaktion der Klassen untereinander. Das dritte Kapitel befasst sich hierzu mit einer speziellen Art von wiederverwendbaren Komponenten, den Rahmenwerken. Das vierte Kapitel beschreibt eine Reihe von Regeln, die Softwareentwicklern bei der Erstellung von wiederverwendbarer Software helfen sollen.

2 Objektorientierte Programmiersprachen und Wiederverwendung

Einer der Gründe, warum sich die objektorientierten Programmiersprachen gegenüber den nicht-objektorientierten durchgesetzt haben, ist, dass mit ihnen ein hoher Grad an Software-Wiederverwendung erreichbar ist. Von nicht-objektorientierten Sprachen wird dies nur wenig unterstützt. Nicht-objektorientiert entwickelter Code wird nur in geringem Maße wiederverwendet. Oftmals werden Programmabschnitte für bereits gelöste Probleme wieder neu geschrieben. Wiederverwendet werden dabei lediglich die beim Programmieren verwendeten Ideen für die Problemlösung und die von den Softwareentwicklern dabei gesammelte Erfahrung.[Hen04]

Der objektorientierte Ansatz bietet eine Reihe von Mitteln für die Erstellung von wiederverwendbarem Code. Einige wichtige werden in diesem Kapitel vorgestellt.

2.1 Polymorphie

Polymorphie lässt sich aus dem Griechischen mit Vielgestaltigkeit übersetzen. In der Informatik versteht man unter Inklusions-Polymorphie, dass Objekte aus verschiedenen Klassen einer Vererbungshierarchie auf eine gleichnamige Nachricht unterschiedlich reagieren können. Sie lässt sich in Laufzeit- und Kompilationszeit-Polymorphie unterscheiden.[CW85, Wikb]

Ein großer Vorteil von objektorientierten Programmiersprachen gegenüber nicht-objektorientierten im Bezug auf Wiederverwendbarkeit ist das Erlauben von Laufzeit-Polymorphie.[JF88] D.h., dass erst zur Laufzeit des Programms anhand der Klasse des Objektes bestimmt wird, welche Methode aufgerufen wird. Es findet eine sogenannte dynamische oder späte Bindung statt.

Im Gegensatz dazu ist Polymorphie in nicht-objektorientierten Programmiersprachen nur mit statischem Binden möglich. Schon beim Übersetzen des Codes wird festgelegt, welche Funktion unter welchen Bedingungen ausgeführt wird. Deshalb spricht man hier auch von Kompilationszeit-Polymorphie.

Um zu erläutern, warum damit die Wiederverwendung des Codes erschwert wird, betrachten wir folgendes Beispiel. Es soll ein Universitätsverwaltungsprogramm entwickelt werden, mit dem u.a. Daten über Studenten und Dozenten verwaltet werden können. Deren Datensätze seien zur Vereinfachung des Beispiels in globalen Feldern namens *StudentData* für die Studenten und *TeacherData* für die Dozenten gespeichert. Die Felder enthalten Datenstrukturen mit Informationen über die Personen, wie z.B. den Namen. Es soll nun eine Funktion *getName(int memberType, int index)* implementiert werden. In Abhängigkeit vom Typen (*memberType*) soll sie den Namen der Person zurückliefern, deren Datensatz an der Stelle *index* in dem jeweiligen Feld gespeichert wurde. Im nicht-objektorientierten Fall würde man hier z.B. wie in Abbildung 1 dargestellt vorgehen. Da es noch mehr Gemeinsamkeiten zwischen Studenten und Dozenten geben wird, würden switch-case-Anweisungen dieser Art in der Software wahrschein-

```
string getName(int memberType, int index) {
      switch (memberType){
            case STUDENT:
                  return StudentData[index].name;
            case TEACHER:
                  return TeacherData[index].name;
      }
      return ERROR;
}
```

Abbildung 1: Beispielhafter Pseudocode für die Funktion *getName*

lich an mehreren Stellen über den kompletten Code verteilt auftreten.

Das Problem in Bezug auf Wiederverwendung liegt hier in der schlechten Erweiterbarkeit. Wenn z.B. später auch noch die Daten über die Doktoranden mitverwalten werden sollen, so müsste jede der switch-case-Anweisungen um einen weiteren Fall erweitert werden. Dies kann nicht nur aufwendig sein. Sondern es kann auch schnell zu Fehlern führen, die nicht vom Übersetzer bemängelt werden, und zwar wenn man das Hinzufügen des neuen Falles an einer Stelle vergisst.

Durch das Konzept der Vererbung und der dynamischen Bindung würde man hingegen in objektorientierten Sprachen auf dieses Problem nicht treffen. Man könnte z.B. eine Basisklasse *Member* einführen, welche die Methode *getName* deklariert. Von dieser Klasse würden dann die Klassen *Teacher* und *Student* erben und gegebenenfalls die Methode *getName* überschreiben.

2.2 Schnittstellen und Methodennamen

Schnittstellen werden in objektorientierten Programmiersprachen verwendet, um eine Klasse von Objekten zu spezifizieren. Sie enthalten alle von den Objekten nach außen hin angebotenen Operationen, deren Methoden. Die dadurch entstehende Datenabstraktion hat den Vorteil, dass ein Softwareentwickler bei der (Wieder-)Verwendung einer Klasse nicht die Implementierung sondern lediglich das durch die Schnittstelle beschriebene Protokoll verstehen muss. [JF88]

Eine Methode ist im Gegensatz zu einer Prozedur immer von ihrem Kontext, also der Klasse, in der sie definiert wurde, abhängig. Anders als z.B. in einem Pascal-Programm, in dem alle Prozeduren eindeutig benannt sein müssen, ist es in objektorientierten Programmiersprachen deshalb erlaubt, die gleichen Methodennamen in verschiedenen Klassen mehrfach zu verwenden. Diese Wiederverwendung von Methodennamen führt zu einer besseren Verständlichkeit des Codes. [JF88] Ein Softwareentwickler, der eine ihm unbekannte Klasse zum ersten Mal benutzen will, kann die Bedeutung

der Methoden schneller erfassen, wenn er deren Namen bereits aus den ihm bekannten Klassen her kennt. Das setzt allerdings voraus, dass der Softwareentwickler der Klasse auch die mit dem Namen assoziierte Funktionalität implementiert hat. Ansonsten kann es schnell passieren, dass derjenige, der die Klasse später verwendet, nicht das von ihm erwartete Ergebnis erzielt.

Die Benutzung von Klassen wird also schneller erlernbar, und sie sind damit auch besser durch Dritte wiederverwendbar, wenn man bei der Benennung von Methoden darauf achtet, dass der Name am besten schon aus einem anderen Kontext bekannt ist und dass er auch das widerspiegelt, was die Methode leistet.

2.3 Vererbung

Der größte Vorteil von objektorientierten Programmiersprachen gegenüber nicht-objektorientierten in Bezug auf Wiederverwendbarkeit ist sicherlich das Konzept der Vererbung. Es ermöglicht einem Softwareentwickler in relativ kurzer Zeit, neue Klassen zu bauen, indem er diese von bereits vorhandenen ableitet und danach lediglich die gewünschten Modifikationen oder Erweiterungen vornimmt.

Durch die Einführung des Vererbungskonzepts hat sich der Programmierstil deutlich verändert. Ein Softwareentwickler erzeugt seinen Code nun meistens nach einer Methode, die von den Autoren Programmierung durch Differenzierung ("programming-by-difference") genannt wird.[JF88] Dabei sucht er sich zur Lösung seines Problems eine bereits vorhandene Klasse, die ein ähnliches Problem behandelt, heraus und schreibt dann eine neue Klasse, die von dieser erbt. Danach muss er nur noch die Unterschiede zwischen den beiden Problemen durch Veränderungen an der neuen Klasse umsetzen. Dieser Programmierstil fördert in hohen Maße die Wiederverwendung von Code. Es ist jedoch hierfür auch um so wichtiger, dass die Klassen so entworfen werden, dass sie für die Wiederverwendung geeignet sind.

Ein weiterer Vorteil des Vererbungskonzepts ist die Wiederverwendung von Code durch Bildung von Hierarchien, anstelle dies durch Duplizieren von Code zu erreichen. Wenn in mehreren Klassen identische Methoden oder Variablen spezifiziert sind, ist es oftmals sinnvoll, diese heraus zu extrahieren und in eine Basisklasse zu verlegen. Die Klassen erben dann die Methoden oder Variablen, indem sie von der Basisklasse ableiten. Dadurch wird der Code auch pflegeleichter, da Veränderungen nur noch an einer Stelle, und zwar in der Basisklasse, vorgenommen werden müssen.

2.4 Abstrakte Klassen

Von abstrakten Klassen können im Unterschied zu konkreten Klassen keine Objekte erzeugt werden. Zusätzlich zu bereits vollständig implementierten Methoden kann man in ihnen auch abstrakte Methoden deklarieren, die erst von ableitenden, konkreten Klassen implementiert werden müssen. Damit sind sie noch nicht auf ein konkretes Verhalten festgelegt, und sie können von der Datenrepräsentation abstrahieren.[JF88]

Durch das Vererbungskonzept unterstützen auch abstrakte Klassen die Wiederverwendbarkeit von Code. Wie bei der normalen Vererbung kann man nach Ableiten von abstrakten Klassen auf ererbte, bereits implementierte Methoden zurückgreifen. Das Ableiten von abstrakten Klassen gestaltet sich sogar um einiges unproblematischer als das von konkreten Klassen. Das liegt daran, dass eine konkrete Klasse schon Aussagen über die Datenrepräsentation trifft. Dadurch kann es zu Konflikten kommen, wenn man in einer ableitenden Klasse eine andere Repräsentation benötigt. Aus diesem Grund sollte man das Definieren von abstrakten Klassen immer vorziehen.[JF88]

Da es aber nicht einfach ist und viel Erfahrung benötigt, gleich am Anfang einer Problemlösung eine Abstraktion für das Problem zu finden, wird in der Design-Phase der Softwareentwicklung meistens erst einmal eine Hierarchie aus konkreten Klassen entworfen. Um den Grad der Wiederverwendbarkeit zu erhöhen, sollte aber in einer späteren Konsolidierungsphase darauf Acht gegeben werden, von dem konkreten Problem zu abstrahieren und so viele Klassen wie möglich abstrakt werden zu lassen.[JF88]

3 Rahmenwerke

Eine wichtige Technik für den Entwurf von wiederverwendbarer Software ist der Bau von Rahmenwerken (engl. *framework*). „Mit ihrer Hilfe erreichen objektorientierte Systeme den höchsten Grad an Wiederverwendung.“[GHJV01]

Ein Rahmenwerk ist ein objektorientiertes abstraktes Entwurfsmuster für die generische Lösung einer Problemfamilie aus einem bestimmten Kontext. Es besteht aus mehreren kollaborierenden Klassen und bildet ein nahezu vollständiges Programm. Zusammen stellen die Klassen einen wiederverwendbaren Entwurf für eine bestimmte Art von Software dar. [GHJV01]

Einem Softwareentwickler bietet ein Rahmenwerk die Möglichkeit, bestimmte Grundfunktionalitäten wiederzuverwenden. So stellt z.B. ein Applikationsrahmenwerk Funktionalitäten für die Entwicklung von Applikationen, wie GUI-Funktionalitäten, zur Verfügung.[Wika] Ein bekanntes Beispiel für ein Applikationsrahmenwerk ist Java Swing.

Eine weitere Ausprägung von Rahmenwerken bilden die domänenspezifischen Rahmenwerke.[Wika] Sie stellen Funktionalitäten und Strukturen für eine spezifische Anwendungsdomäne zur Verfügung. So kann man sich zum Beispiel ein Rahmenwerk vorstellen, dass Grundfunktionalitäten für Buchhaltungsprogramme bereitstellt.

Für die von ihm zu bauende Software muss ein Softwareentwickler dem Rahmenwerk nur noch den für den Anwendungsfall spezifischen Code hinzufügen. Dabei ruft er keine der zur Verfügung gestellte Funktionalität auf, sondern sein Code wird vom Rahmenwerk aufgerufen. Diese Methodik der Kontrollflussumkehrung ist auch bekannt als das Hollywood-Prinzip: “Don’t call us, we’ll call you.“Wenn man eine konventionelle Klassenbibliothek verwendet, schreibt man den Hauptteil der Anwendung selbst und ruft dabei den Code der Bibliothek auf, den man wiederverwenden will. Wenn man auf ein Rahmenwerk zurückgreift, verwendet man hingegen den Hauptteil wieder und schreibt den Code, der vom Rahmenwerk aufgerufen wird.[GHJV01]

Durch den Gebrauch von Rahmenwerken werden nicht einzelne Klassen wiederverwendet, sondern die gesamte Konstruktion. Es kommt also nicht nur zur Wiederverwendung des Codes, sondern auch von dem Design.

Man kann Rahmenwerke nach der Sichtbarkeit der inneren Strukturen und nach der Art, wie sie vom Softwareentwickler zu benutzen sind, in White-Box- und Black-Box-Rahmenwerke unterteilen.

3.1 White-Box-Rahmenwerke

White-Box-Rahmenwerke betreiben Wiederverwendung durch Unterklassenbildung. Die Bezeichnung White-Box kommt daher, dass durch die Vererbung die interne Struktur der Basisklasse in der Unterklasse sichtbar ist.[GHJV01] Für die Verwendung eines White-Box-Rahmenwerkes muss ein Softwareentwickler mit Hilfe von Vererbung bestimmte abstrakte Klassen implementieren bzw. Unterklassen von konkreten Klassen bilden. Dafür braucht er Kenntnisse über die Struktur und die Implementierung des Rahmenwerkes.[JF88]

3.2 Black-Box-Rahmenwerke

Wie der Name schon suggeriert, ist bei einem Black-Box-Rahmenwerk die Sichtbarkeit der internen Struktur nicht gegeben. Die für die Anpassung an das jeweilige Einsatzgebiet erforderlichen Erweiterung des Rahmenwerkes werden durch die Komposition von Objekten vorgenommen.[GHJV01] Es wird also Wiederverwendung durch Komposition betrieben.

Größere Black-Box-Rahmenwerke stellen nicht nur einzelne Klassen, sondern eine Menge von Komponenten bereit, die der Softwareentwickler für seine Anwendung geeignet zusammenstellen kann. Da ein Rahmenwerk immer nur ein Grundgerüst der Software darstellt, werden zusätzlich noch speziell auf den Anwendungsfall zugeschnittene Komponenten geschrieben und über die Schnittstellen des Rahmenwerkes mit eingebunden.

Ein Softwareentwickler muss bei der Benutzung eines Black-Box-Rahmenwerkes nur dessen Schnittstellen bedienen können und den Einsatzzweck der bereitgestellten Komponenten kennen. Er braucht nichts, über den inneren Aufbau des Rahmenwerkes zu wissen.[JF88]

3.3 White-Box vs. Black-Box

Das Verwenden von Black-Box-Rahmenwerken ist in der Regel schneller erlernbar. Das liegt daran, dass der Softwareentwickler, wie oben bereits erläutert, nur die Schnittstelle des Rahmenwerkes kennen muss. Bei White-Box-Rahmenwerken muss man, um zu lernen, wie es zu benutzen ist, erst einmal lernen, wie es konstruiert wurde. Außerdem entstehen bei dem Bau einer Applikation mit einem White-Box-Rahmenwerk normalerweise sehr viele Unterklassen. Diese sind zwar größtenteils relativ einfach, jedoch wird das Design des Programms dadurch sehr schnell unübersichtlich. Aus diesen Gründen sollte das Ziel beim Bau eines Rahmenwerkes möglichst immer ein Black-Box-Rahmenwerk sein. [JF88]

Die Autoren Gamma et al. haben diese Erkenntnis von Johnson und Foote später verallgemeinert und daraufhin folgendes Prinzip formuliert: “Ziehe Objektkomposition der Klassenvererbung vor“.[GHJV01]

4 Regeln zum Erstellen von wiederverwendbarem Code

Die Autoren haben in ihrer Arbeit 13 Regeln zusammengestellt, die Softwareentwickler beim Bau von wiederverwendbarer Software beachten sollten. Nachfolgend wird eine Auswahl dieser Regeln vorgestellt.

4.1 Regeln zum Finden von Standardprotokollen

Regel 2

Fallunterscheidungen, die Objekte auf ihre Klassenzugehörigkeit überprüfen, sollten aus dem Code entfernt werden.[JF88] Ein entsprechendes negatives Beispiel zeigt der Code in Abbildung 2.

```
...
BaseClass obj;

if (obj instanceof ClassX)
   doSomethingX();
else if (obj instanceof ClassY)
   doSomethingY();
...
```

Abbildung 2: Negatives Beispiel für Fallunterscheidungen

Anstelle der Fallunterscheidungen sollte in der Klasse *BaseClass* eine neue (am besten abstrakte) Methode, z.B. *doSomething*, eingeführt werden, die diesen speziellen Fall jeweils in den Unterklassen behandelt. Somit kann man den Code aus Abbildung 2 mit der Anweisung: *obj.doSomething();* ersetzen. Dadurch gestaltet sich eine spätere Erweiterung um einen neuen Fall einfacher, da die Methode, in der die Fallunterscheidung auftrat, nicht verändert werden müsste.

Regel 3

Je mehr Eingabeparameter eine Methode erwartet, desto schwieriger ist sie zu verwenden und desto schwieriger wird der Code zu lesen, in dem sie aufgerufen wird.[JF88] Während des Programmierens sollte deshalb darauf geachtet werden, die Anzahl der Eingabeparameter von Methoden so gering wie möglich zu halten. In einer Konsolidierungsphase sollten die Methoden noch einmal daraufhin überprüft werden. Eine Verringerung der Parameter kann hier auf zwei Wegen erreicht werden. Zum einen kann man die betreffende Methode in Bezug auf die Verwendung der Parameter in mehrere kleinere Methoden unterteilen. Zum anderen kann man mehrere Parameter zu einer Klasse zusammenfassen und anstelle der einzelnen Parameter ein Objekt dieser Klasse der Methode übergeben. [JF88]

Regel 4

Die vierte Regel schreibt vor, dass die Anzahl der Codezeilen pro Methode gering gehalten werden sollen, indem man große Methoden in mehrere kleinere unterteilt. [JF88] Das erhöht nicht nur die Wartbarkeit, sondern kann es auch einfacher machen, von der Klasse abzuleiten, da die Methoden dann weniger speziell arbeiten.

4.2 Regeln zum Finden von abstrakten Klassen

Regel 5

Klassenhierarchien sollten tief und schmal sein.[JF88] D.h., von einer Basisklasse sollten immer nur wenige Klassen direkt ableiten. Eine zu weitläufige Hierarchie sollte deshalb durch Einführen neuer abstrakter Basisklassen umstrukturiert werden. Sich ähnelnde Klassen der weitläufigen Hierarchieebenen erben jeweils von einer neu eingeführten abstrakten Klasse.

Regel 7

Konkrete Klassen unterscheiden sich vor allem durch das Vorhandensein einer Datenrepräsentation von abstrakten Klassen. Deshalb sollte in konkreten Klassen versucht werden, von der Datenrepräsentation zu abstrahieren, indem für Zugriffe auf Klassenvariablen jeweils eigene Getter- und Setter-Methoden definiert und verwendet werden.[JF88] Dadurch fällt es später leichter, diese Klassen als abstrakt zu deklarieren.

Regel 8

Unterklassen sollten Spezialisierungen der Basisklasse sein.[JF88] D.h., das Verhalten der Basisklasse sollte eine Teilmenge von dem Verhalten der Unterklasse bilden. Erreicht wird dies, indem in Unterklassen keine Methoden überschrieben werden, sondern lediglich neue Methoden hinzugefügt werden. Dadurch wird es einfacher, die Basisklasse als abstrakt zu deklarieren. Die zusätzlich in den Unterklassen definierten Methoden werden als abstrakte Methoden in der Basisklasse deklariert.

4.3 Regeln zum Finden von Rahmenwerken

Regel 9

Große Klassen mit vielen Methoden sollten daraufhin überprüft werden, ob sie sich nicht in mehrere kleinere Klassen unterteilen lassen. [JF88] Der Vorteil von kleineren Klassen ist, dass ihre Verwendung besser erlernbar ist. Das ergibt sich schon alleine aus der erhöhten Übersichtlichkeit, die kleinere Klassen bieten.

Regel 10

Falls einige Unterklassen einer Basisklasse eine Methode auf die gleiche Weise implementieren, bei anderen dies aber auf eine andere Weise geschieht, sollte man diese Methode aus der Basisklasse herausnehmen und in einer neuen Klassenhierarchie behandeln. [JF88]Der Grund hierfür ist, dass eine solche Erscheinung ein Indiz dafür sein kann, dass die Implementierung dieser Methode unabhängig von der Basisklasse ist und die Methode damit auch nicht in diese Klasse gehört.

Regel 11

Klassen mit vielen Methoden können unter Betrachtung des Zugriffs auf die Klassenvariablen aufgespaltet werden. [JF88] Wenn in einer Klasse ein Teil der globalen Variablen von mehreren Methoden benutzt wird, und ein anderer Teil nur von anderen Methoden verwendet wird, sollte man die Klasse meistens aufspalten. Dabei kommt der erste Teil der Klassenvariablen mit den darauf zugreifenden Methoden in die eine Klasse und der Rest in die andere Klasse.

5 Zusammenfassung und Ausblick

In ihrem Artikel haben die beiden Autoren Techniken und Regeln zusammengestellt, die ein Softwareentwickler verwenden sollte, wenn er wiederverwendbare Software schreiben will.

Wie William Opdyke, dessen Doktorvater der Autor Ralph Johnson war, in [Opd] beschreibt, fehlt dem Softwareentwickler aber oftmals die Motivation dazu, den von ihm erstellten Code in Bezug auf Wiederverwendbarkeit zu konsolidieren. Er erkennt den dadurch resultierenden Nutzen nicht immer sofort. Das liegt daran, dass der Effekt, den eine konsolidierte Software in Bezug auf ihre Funktionalität erzielt, gleich dem der ursprünglichen ist. Bei der Konsolidierung wird schließlich nicht die Funktionalität, sondern nur die Struktur der Software verändert.

Die vorgestellten Regeln bieten auch heute noch eine gute Grundlage für weitere Forschungen auf diesem Gebiet. So könnte man sich zum Beispiel ein in die Programmierumgebung integriertes Werkzeug vorstellen, das den Softwareentwickler darauf aufmerksam macht, falls er eine der Regeln nicht befolgt hat.

Wiederverwendung hat aber auch seine Grenzen. Ted Biggerstaff und Charles Richter bemängeln in [BR87], dass Programme, die nur aus wiederverwendeten Komponenten gebaut wurden, typischerweise unter Performanzproblemen leiden.

Ein, wie ich finde, wichtiger Aspekt in Bezug auf Wiederverwendbarkeit ist die gute Dokumentation. Dieser Aspekt wird jedoch weder in dem hier betrachteten Artikel noch in anderen von mir gelesenen Quellen intensiver untersucht. Meines Erachtens nach übt die Qualität der Code-Dokumentation einen großen Einfluss auf den Grad der Wiederverwendbarkeit aus. Je besser die Dokumentation von Klassen ist, desto leichter erlernbar ist deren Benutzung, und dementsprechend gestaltet sich auch deren Wiederverwendung einfacher.

Aufbauend auf den Artikel haben die Wissenschaftler Brian Foote und William Opdyke 1994 eine Arbeit über die Konsolidierung von Software verfasst. [FO94] Darin beschreiben sie eine Reihe von Mustern für die Konsolidierung unter Verwendung der oben stehenden 13 Entwurfsregeln.

1992 traf William Opdyke auf der OOPSLA Konferenz in Vancouver den Autor Martin Fowler. Letzterer bat ihn später ein Kapitel für sein neues Buch zu schreiben: “Refactoring: Improving The Design Of Existing Code“.[FBB+99] Dieses Buch ist heute eines der meist zitierten Bücher, wenn es um das Thema der Software-Konsolidierung geht. Es sei hier als weiterführende Lektüre empfohlen.

Literatur

[BR87] T.J. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41–49, März 1987.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[FO94] Brian Foote and William F. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. *PloP*, August 1994.

[GHJV01] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, Reading, Mass., 2001.

[Hen04] Michael Hendrix. Wiederverwendbarkeit von software. 2004. Online-Skript zur Veranstaltung Objektorientierte Programmierung mit C++, letzter Besuch August 2005.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Opd] William F. Opdyke. Object-oriented refactoring, legacy constraints and reuse.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2001.

[Wika] Wikipedia. Framework. letzter Besuch August 2005.

[Wikb] Wikipedia. Polymorphie(programmierung). letzter Besuch August 2005.