

Design Patterns für die Spielprogrammierung



Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)

Der Verlag räumt Ihnen mit dem Kauf des ebooks das Recht ein, die Inhalte im Rahmen des geltenden Urheberrechts zu nutzen. Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und Einspeicherung und Verarbeitung in elektronischen Systemen.

Der Verlag schützt seine ebooks vor Missbrauch des Urheberrechts durch ein digitales Rechtemanagement. Bei Kauf im Webshop des Verlages werden die ebooks mit einem nicht sichtbaren digitalen Wasserzeichen individuell pro Nutzer signiert.

Bei Kauf in anderen ebook-Webshops erfolgt die Signatur durch die Shopbetreiber. Angaben zu diesem DRM finden Sie auf den Seiten der jeweiligen Anbieter.

Robert Nystrom

Design Patterns für die Spieleprogrammierung

Übersetzung aus dem Amerikanischen
von Knut Lorenzen



Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <<http://dnb.d-nb.de>> abrufbar.

ISBN 978-3-95845-091-2

1. Auflage 2015

www.mitp.de

E-Mail: mitp-verlag@sigloch.de

Telefon: +49 7953 / 7189 - 079

Telefax: +49 7953 / 7189 - 082

© 2015 mitp Verlags GmbH & Co. KG

Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Authorized translation from the English language edition, entitled »Game Programming Patterns«, 978-0-9905829-0-8 by Nystrom, Robert, published by Genever Benning, Copyright © 2014 Robert Nystrom. All rights reserved.

Lektorat: Sabine Schulz

Sprachkorrektur: Maren Feilen

Coverbild: Robert Nystrom

Satz: III-satz, Husby

*Für Megan, für Vertrauen und Zeit,
den beiden wichtigsten Bestandteilen.*

Inhaltsverzeichnis

	Danksagungen	17
Teil I	Einführung	19
<hr/>		
I	Architektur, Performance und Spiele	27
I.1	Was ist Softwarearchitektur?	27
I.1.1	Was zeichnet eine <i>gute</i> Softwarearchitektur aus?	28
I.1.2	Wie nimmt man Änderungen vor?	28
I.1.3	Inwiefern hilft eine Entkopplung?	30
I.2	Zu welchem Preis?	30
I.3	Performance und Geschwindigkeit	32
I.4	Das Gute an schlechtem Code	33
I.5	Ein ausgewogenes Verhältnis finden	34
I.6	Einfachheit	35
I.7	Fang endlich an!	37
Teil II	Design Patterns neu überdacht	39
<hr/>		
2	Command (Befehl)	41
2.1	Eingabekonfiguration	42
2.2	Regieanweisungen	45
2.3	Rückgängig und Wiederholen	47
2.4	Klassen ohne Funktionen?	51
2.5	Siehe auch	53
3	Flyweight (Fliegengewicht)	55
3.1	Den Wald vor lauter Bäumen nicht sehen	55
3.2	Tausend Instanzen	58
3.3	Das Flyweight-Pattern	58
3.4	Ein Ort, um Wurzeln zu schlagen	59

3.5	Und die Performance?	64
3.6	Siehe auch	65
4	Observer (Beobachter)	67
4.1	Erzielte Leistungen	67
4.2	Funktionsweise	69
4.2.1	Der Observer	69
4.2.2	Das Subjekt	70
4.2.3	Beobachtung der Physik-Engine	72
4.3	»Das ist zu langsam!«	73
4.3.1	Oder ist es doch zu schnell?	74
4.4	»Zu viele dynamische Allokationen«	74
4.4.1	Verkettete Observer	75
4.4.2	Ein Pool von Listenknoten	79
4.5	Verbleibende Schwierigkeiten	79
4.5.1	Subjekte und Observer löschen	80
4.5.2	Keine Sorge, der Garbage Collector erledigt das schon	81
4.5.3	Was geht hier vor?	82
4.6	Heutige Observer	83
4.7	Zukünftige Observer	84
5	Prototype (Prototyp)	87
5.1	Das Design Pattern <i>Prototype</i>	87
5.1.1	Wie gut funktioniert es?	91
5.1.2	Spawn-Funktionen	91
5.1.3	Templates	92
5.1.4	First-Class-Typen	93
5.2	Eine auf Prototypen beruhende Sprache	93
5.2.1	Self	93
5.2.2	Wie ist es gelaufen?	96
5.2.3	Was ist mit JavaScript?	97
5.3	Prototypen zur Datenmodellierung	99
6	Singleton	103
6.1	Das Singleton-Pattern	103
6.1.1	Beschränkung einer Klasse auf eine Instanz	103
6.1.2	Bereitstellung eines globalen Zugriffspunkts	104
6.2	Gründe für die Verwendung	105
6.3	Gründe, die Verwendung zu bereuen	107
6.3.1	Singletons sind globale Variablen	108

6.3.2	Das Pattern löst zwei Probleme, selbst wenn es nur eins gibt	109
6.3.3	Die späte Initialisierung entzieht Ihnen die Kontrolle	110
6.4	Verzicht auf Singletons	112
6.4.1	Wird die Klasse überhaupt benötigt?	112
6.4.2	Nur eine Instanz einer Klasse	114
6.4.3	Bequemer Zugriff auf eine Instanz	115
6.5	Was bleibt dem Singleton?	118
7	State (Zustand)	119
7.1	Altbekanntes	119
7.2	Zustandsautomaten erledigen das	123
7.3	Enumerationen und Switch-Anweisungen	124
7.4	Das State-Pattern	127
7.4.1	Das Interface für den Zustand	128
7.4.2	Klassen für alle Zustände	128
7.4.3	An den Zustand delegieren	129
7.5	Wo sind die Zustandsobjekte?	130
7.5.1	Statische Zustände	130
7.5.2	Instanzierte Zustandsobjekte	131
7.6	Eintritts- und Austrittsaktionen	132
7.7	Wo ist der Haken?	134
7.8	Nebenläufige Zustandsautomaten	134
7.9	Hierarchische Zustandsautomaten	136
7.10	Kellerautomaten	138
7.11	Wie nützlich sind sie?	139

Teil III Sequenzierungsmuster (Sequencing Patterns) 141

8	Double Buffer (Doppelter Buffer)	143
8.1	Motivation	143
8.1.1	Computergrafik kurz und bündig	143
8.1.2	Erster Akt, erste Szene	145
8.1.3	Zurück zur Grafik	146
8.2	Das Pattern.	146
8.3	Anwendbarkeit.	147
8.4	Konsequenzen	147
8.4.1	Der Austausch selbst kostet Zeit.	147
8.4.2	Zwei Framebuffer belegen mehr Arbeitsspeicher	147

8.5	Beispielcode	148
8.5.1	Nicht nur Grafik	151
8.5.2	Künstliche Unintelligenz	151
8.5.3	Gebufferte Ohrfeigen	155
8.6	Designentscheidungen	156
8.6.1	Wie werden die Buffer ausgetauscht?	157
8.6.2	Wie fein ist der Buffer untergliedert?	158
8.7	Siehe auch	159
9	Game Loop (Hauptschleife)	161
9.1	Motivation.	161
9.1.1	Interview mit einer CPU	161
9.1.2	Ereignisschleifen	162
9.1.3	Eine aus dem Takt geratene Welt	163
9.1.4	Sekunden pro Sekunde	164
9.2	Das Pattern	164
9.3	Anwendbarkeit.	164
9.4	Konsequenzen	165
9.4.1	Abstimmung mit der Ereignisschleife des Betriebssystems	165
9.5	Beispielcode	166
9.5.1	Die Beine in die Hand nehmen.	166
9.5.2	Ein kleines Nickerchen.	166
9.5.3	Ein kleiner und ein großer Schritt.	167
9.5.4	Aufholjagd.	169
9.5.5	In der Mitte hängen geblieben.	171
9.6	Designentscheidungen	173
9.6.1	Stammt die Game Loop aus Ihrer Feder oder benutzen Sie die der Plattform?	173
9.6.2	Wie handhaben Sie die Leistungsaufnahme?	174
9.6.3	Wie steuern Sie die Spielgeschwindigkeit?	175
9.7	Siehe auch	176
10	Update Method (Aktualisierungsmethode)	177
10.1	Motivation.	177
10.2	Das Pattern	180
10.3	Anwendbarkeit.	180
10.4	Konsequenzen	181

10.4.1	Verkomplizierung durch Aufteilen des Codes in einzelne Frames	181
10.4.2	Der Zustand muss gespeichert werden, um im nächsten Frame fortfahren zu können	181
10.4.3	Alle Objekte simulieren jeden Frame, aber nicht wirklich exakt gleichzeitig	182
10.4.4	Obacht bei der Modifizierung der Objektliste während der Aktualisierung	182
10.5	Beispielcode	184
10.5.1	Entity-Unterklassen?	186
10.5.2	Entities definieren	186
10.5.3	Zeitablauf	189
10.6	Designentscheidungen	190
10.6.1	Zu welcher Klasse gehört die update()-Methode?	190
10.6.2	Wie werden inaktive Objekte gehandhabt?	191
10.7	Siehe auch	192

Teil IV Verhaltensmuster (Behavioral Patterns) 193

II	Bytecode	195
II.1	Motivation	195
II.1.1	Wettkampf der Zaubersprüche	196
II.1.2	Daten > Code	196
II.1.3	Das Interpreter-Pattern	196
II.1.4	Faktisch Maschinencode	200
II.2	Das Pattern.	201
II.3	Anwendbarkeit.	201
II.4	Konsequenzen	201
II.4.1	Befehlsformat	202
II.4.2	Fehlender Debugger.	203
II.5	Beispielcode	203
II.5.1	Eine zauberhafte API	203
II.5.2	Ein bezaubernder Befehlssatz	204
II.5.3	Eine Stackmaschine	206
II.5.4	Verhalten = Komposition.	209
II.5.5	Eine virtuelle Maschine	212
II.5.6	Hexerwerkzeuge.	213

II.6	Designentscheidungen	215
II.6.1	Wie greifen Befehle auf den Stack zu?	215
II.6.2	Welche Befehle gibt es?	216
II.6.3	Wie werden Werte repräsentiert?	217
II.6.4	Wie wird der Bytecode erzeugt?	220
II.7	Siehe auch	222
12	Subclass Sandbox (Unterklassen-Sandbox)	223
12.1	Motivation.	223
12.2	Das Pattern	225
12.3	Anwendbarkeit.	226
12.4	Konsequenzen	226
12.5	Beispielcode	226
12.6	Designentscheidungen	229
12.6.1	Welche Operationen sollen bereitgestellt werden?	229
12.6.2	Sollen Methoden direkt oder durch Objekte, die sie enthalten, bereitgestellt werden?	231
12.6.3	Wie gelangt die Basisklasse an die benötigten Zustände? ..	232
12.7	Siehe auch	236
13	Type Object (Typ-Objekt)	237
13.1	Motivation.	237
13.1.1	Die typische OOP-Lösung	237
13.1.2	Eine Klasse für eine Klasse	239
13.2	Das Pattern	241
13.3	Anwendbarkeit.	241
13.4	Konsequenzen	242
13.4.1	Typ-Objekte müssen manuell gehandhabt werden	242
13.4.2	Die Definition des Verhaltens der verschiedenen Typen ist schwieriger	242
13.5	Beispielcode	243
13.5.1	Typartiges Verhalten von Typ-Objekten: Konstruktoren ...	245
13.5.2	Gemeinsame Nutzung von Daten durch Vererbung	246
13.6	Designentscheidungen	250
13.6.1	Ist das Typ-Objekt gekapselt oder zugänglich?	250
13.6.2	Wie werden Typ-Objekte erzeugt?	251
13.6.3	Kann sich der Typ ändern?	252
13.6.4	Welche Formen der Vererbung werden unterstützt?	253
13.7	Siehe auch	254

Teil V	Entkopplungsmuster (Decoupling Patterns)	255
14	Component (Komponente)	257
14.1	Motivation	257
14.1.1	Der Gordische Knoten	258
14.1.2	Den Knoten durchschlagen	258
14.1.3	Unerledigtes	259
14.1.4	Wiederverwendung	259
14.2	Das Pattern	261
14.3	Anwendbarkeit	261
14.4	Konsequenzen	262
14.5	Beispielcode	262
14.5.1	Eine monolithische Klasse	263
14.5.2	Abspalten eines Bereichs	264
14.5.3	Abspalten der übrigen Bereiche	266
14.5.4	Robo-Bjørn	268
14.5.5	Ganz ohne Bjørn?	270
14.6	Designentscheidungen	272
14.6.1	Wie gelangt ein Objekt an seine Komponenten?	273
14.6.2	Wie kommunizieren die Komponenten untereinander?	273
14.7	Siehe auch ...	277
15	Event Queue (Ereigniswarteschlange)	279
15.1	Motivation	279
15.1.1	Ereignisschleife der grafischen Benutzeroberfläche	279
15.1.2	Zentrale Ereignissammlung	280
15.1.3	Wie bitte?	281
15.2	Das Pattern	284
15.3	Anwendbarkeit	284
15.4	Konsequenzen	285
15.4.1	Eine zentrale Ereigniswarteschlange ist eine globale Variable	285
15.4.2	Den Boden unter den Füßen verlieren	285
15.4.3	Steckenbleiben in Rückkopplungsschleifen	286
15.5	Beispielcode	286
15.5.1	Ein Ring-Buffer	289
15.5.2	Anfragen zusammenfassen	293
15.5.3	Threads	294

15.6	Designentscheidungen	295
15.6.1	Was soll in die Warteschlange aufgenommen werden?	295
15.6.2	Wer darf lesend auf die Warteschlange zugreifen?	296
15.6.3	Wer darf schreibend auf die Warteschlange zugreifen?	298
15.6.4	Wie lang ist die Lebensdauer der Objekte in der Warteschlange?	299
15.7	Siehe auch	300
16	Service Locator (Dienstlokalisierung)	301
16.1	Motivation.	301
16.2	Das Pattern	302
16.3	Anwendbarkeit.	302
16.4	Konsequenzen	303
16.4.1	Der Dienst muss auch tatsächlich lokalisiert werden können	303
16.4.2	Dem Dienst ist nicht bekannt, wer ihn nutzt	303
16.5	Beispielcode	304
16.5.1	Der Dienst	304
16.5.2	Der Dienstanbieter	304
16.5.3	Ein einfacher Service Locator	305
16.5.4	Ein leerer Dienst	306
16.5.5	Protokollierender Dekorierer	308
16.6	Designentscheidungen	310
16.6.1	Wie wird der Dienst lokalisiert?	310
16.6.2	Was geschieht, wenn die Lokalisierung des Dienstes scheitert?	312
16.6.3	Wer darf auf den Dienst zugreifen?	315
16.7	Siehe auch	316

Teil VI Optimierungsmuster (Optimization Patterns) 317

17	Data Locality (Datenlokalität)	319
17.1	Motivation.	319
17.1.1	Ein Datenlager	320
17.1.2	Eine Palette für die CPU	322
17.1.3	Daten = Performance?	323
17.2	Das Pattern	324
17.3	Anwendbarkeit	325

17.4	Konsequenzen	325
17.5	Beispielcode	326
17.5.1	Aneinandergereihte Arrays	326
17.5.2	Gebündelte Daten.	331
17.5.3	Hot/Cold Splitting	335
17.6	Designentscheidungen	337
17.6.1	Wie wird Polymorphismus gehandhabt?	338
17.6.2	Wie werden Spielobjekte definiert?	339
17.7	Siehe auch	342
18	Dirty Flag (Veraltet-Flag)	345
18.1	Motivation	345
18.1.1	Lokale Koordinaten und Weltkoordinaten	346
18.1.2	Gecachte Weltkoordinaten.	347
18.1.3	Verzögerte Berechnung	348
18.2	Das Pattern.	350
18.3	Anwendbarkeit.	350
18.4	Konsequenzen	351
18.4.1	Nicht zu lange verzögern	351
18.4.2	Das Flag bei jedem Zustandswechsel ändern	352
18.4.3	Vorherige abgeleitete Daten verbleiben im Speicher.	352
18.5	Beispielcode	353
18.5.1	Nicht-optimierte Traversierung	354
18.5.2	Let's Get Dirty	355
18.6	Designentscheidungen	358
18.6.1	Wann wird das Dirty Flag gelöscht?	358
18.6.2	Wie feingranular ist Ihr »Dirty-Tracking«?	359
18.7	Siehe auch	360
19	Object Pool (Objektpool)	361
19.1	Motivation	361
19.1.1	Der Fluch der Fragmentierung	361
19.1.2	Das Beste beider Welten	362
19.2	Das Pattern.	363
19.3	Anwendbarkeit.	363
19.4	Konsequenzen	363
19.4.1	Der Pool verschwendet möglicherweise Speicherplatz für ungenutzte Objekte	363
19.4.2	Es steht nur eine feste Anzahl von Objekten zur Verfügung	364

19.4.3	Die Objekte sind von fester Größe	365
19.4.4	Wiederverwendete Objekte werden nicht automatisch zurückgesetzt	365
19.4.5	Unbenutzte Objekte verbleiben im Arbeitsspeicher	366
19.5	Beispielcode	366
19.5.1	Eine kostenlose Liste	369
19.6	Designentscheidungen	372
19.6.1	Sind Objekte an den Pool gekoppelt?	372
19.6.2	Wer ist für die Initialisierung der wiederverwendeten Objekte verantwortlich?	374
19.7	Siehe auch	376
20	Spatial Partition (Räumliche Aufteilung)	377
20.1	Motivation.	377
20.1.1	Kampfeinheiten auf dem Schlachtfeld	377
20.1.2	Schlachtreihen zeichnen	378
20.2	Das Pattern.	379
20.3	Anwendbarkeit.	379
20.4	Konsequenzen	379
20.5	Beispielcode	380
20.5.1	Ein Bogen Millimeterpapier.	380
20.5.2	Ein Gitternetz verketteter Einheiten	381
20.5.3	Betreten des Schlachtfeldes	383
20.5.4	Klirrende Schwerter	384
20.5.5	Vormarschieren	385
20.5.6	Um Armeslänge	386
20.6	Designentscheidungen	390
20.6.1	Ist die Aufteilung hierarchisch oder gleichmäßig?	390
20.6.2	Hängt die Zellengröße von der Verteilung der Objekte ab?	391
20.6.3	Werden die Objekte nur in den Zellen gespeichert?	393
20.7	Siehe auch	394
	Stichwortverzeichnis	395

Danksagungen

Ich habe mir sagen lassen, dass nur andere Autoren einschätzen können, was zum Schreiben eines Buches alles dazugehört. Allerdings gibt es da noch ein anderes Völkchen, das genau weiß, was für eine Belastung das ist – nämlich die Unglücklichen, die in einer Beziehung zum Autor stehen. Ich habe mir die Zeit zum Schreiben mühsam zusammenkratzen müssen, denn der Alltag ist für meine Frau Megan und mich eigentlich schon anstrengend genug. Abzuwaschen und die Kinder zu baden, ist zwar nicht gerade das, was man unter »Schreiben« versteht, aber ohne ihre Hilfe gäbe es dieses Buch nicht.

Als ich mit diesem Projekt begann, war ich bei Electronic Arts als Programmierer tätig. Ich denke, das Unternehmen wusste nicht so recht, was davon zu halten war, und so bin ich Michael Malone, Olivier Nallet und Richard Wifall für ihre Unterstützung und ihr detailliertes, aufschlussreiches Feedback zu den ersten paar Kapiteln umso dankbarer.

Nachdem etwa die Hälfte des Buches erledigt war, entschloss ich mich, auf die Veröffentlichung durch einen traditionellen Verlag zu verzichten. Mir war durchaus bewusst, dass mir damit auch die fachliche Unterstützung eines Lektorats fehlen würde, andererseits hatten mir jedoch schon Dutzende Leser gemailt, die keinen Zweifel daran ließen, in welche Richtung das Buch ihrer Meinung nach gehen sollte. Auch auf das Korrektorat musste ich verzichten, dafür hatte ich aber bereits mehr als 250 Fehlerberichte erhalten, die mir dabei halfen, den Text auf Vordermann zu bringen. Ich hatte es schnell aufgegeben, mich beim Schreiben an einen Zeitplan zu halten – doch wenn einem die Leser nach der Fertigstellung der einzelnen Kapitel auf die Schulter klopfen, ist das Ansporn genug!

» Ich musste zwar auf ein verlagsseitiges Lektorat verzichten, nicht aber auf gute Korrekturleser. Lauren Briese war stets zur Stelle, wenn ich Hilfe brauchte, und hat ausgezeichnete Arbeit geleistet. «

» Mein besonderer Dank gilt Colm Sloan, der jedes einzelne Kapitel *zweimal* überprüft hat und mir haufenweise fabelhaftes Feedback lieferte – nur weil er so ein großzügiger Mensch ist. Du hast ein Bier (oder zwanzig) bei mir gut! «

Diese Art der Veröffentlichung wird als »Selbstverlag« oder »Self-Publishing« bezeichnet, obwohl der Begriff »Crowd-Publishing« der Wahrheit näher kommt. Schreiben kann man zwar auch alleine, ich selbst war allerdings niemals alleine.

Sogar als ich die Arbeit am Buch zwei Jahre ruhen ließ, wurde ich weiterhin ermutigt, es fertigzustellen. Ohne all die Leute (Dutzende!), die mich immer wieder daran erinnerten, dass sie auf weitere Kapitel warteten, hätte ich das Buch wohl nicht zu Ende gebracht.

Ich möchte allen, die E-Mails geschrieben, Kommentare gesendet, getwittert, Fehler berichtet, Freunden von diesem Buch erzählt oder irgendwie Verbindung mit mir aufgenommen haben, sagen: Ich bin euch wirklich von ganzem Herzen dankbar. Dieses Buch fertigzustellen, gehörte zu meinen größten Wünschen. Und ihr habt dafür gesorgt, dass ich es geschafft habe. Danke!

Bob Nystrom, 6. September 2014

Teil I

Einführung

In diesem Teil:

- **Kapitel 1**
Architektur, Performance und Spiele 27

Im fünften Schuljahr standen in unserem Klassenzimmer ein paar ziemlich ramponierte TRS-80-Computer herum, die meine Freunde und ich benutzen durften. Ein Lehrer, der unsere Begeisterung wecken wollte, gab uns einen Stapel Ausdrucke einiger einfacher BASIC-Programme, mit denen wir experimentieren konnten.

Die Kassettenlaufwerke der Rechner waren defekt, d.h., wenn wir Code ausführen wollten, mussten wir ihn immer sehr sorgfältig komplett neu eingeben. Das führte dazu, dass wir bevorzugt auf Programme zurückgriffen, die nur einige wenige Zeilen lang waren:

```
10 PRINT "BOBBY IST EIN RADIKALER!!!"  
20 GOTO 10
```

» Vielleicht wird es ja auf wundersame Weise wahr, wenn der Computer es nur oft genug ausgibt ... «

Trotzdem war das Ganze Glückssache. Wir hatten keine Ahnung von Programmierung, und schon der kleinste Syntaxfehler war für uns unergründlich. Wenn ein Programm nicht funktionierte, und das kam des Öfteren vor, fingen wir einfach von vorne an.

Am unteren Ende des Papierstapels mit Ausdrucken fanden wir schließlich ein richtiges Ungeheuer: ein Programm, das aus mehreren eng mit Code bedruckten Seiten bestand. Es dauerte ein Weilchen, bis wir überhaupt den Mut aufbrachten, es einzugeben, aber es war einfach unwiderstehlich, denn der Titel am Anfang des Listings lautete »Tunnels & Trolls«. Wir hatten keinen blassen Schimmer, worum es bei dem Programm ging, aber es klang nach einem Spiel, und was könnte cooler sein als ein selbstprogrammiertes Computerspiel?

Leider haben wir es nie zum Laufen bekommen und im darauffolgenden Schuljahr zogen wir in ein anderes Klassenzimmer um. (Jahre später, nachdem ich etwas BASIC gelernt hatte, wurde mir klar, dass es sich seinerzeit gar nicht um ein Spiel gehandelt hatte. Das Programm erzeugte bloß verschiedene Persönlichkeiten für das eigentliche Rollenspiel.) Dessen ungeachtet waren die Würfel gefallen: Ich hatte mir in den Kopf gesetzt, Spieleprogrammierer zu werden.

Als ich ein Teenager war, schaffte sich meine Familie einen Macintosh mit Quick-BASIC an, später kam dann THINK C dazu. Ich habe fast meine gesamten Sommerferien damit verbracht, Spiele zusammenzuhacken. Es war mühsam, auf mich allein gestellt zu lernen und es ging nur langsam vorwärts. Anfangs war es gar nicht so schwer und ich bekam schnell etwas zum Laufen – vielleicht eine Landkartendarstellung oder ein kleines Puzzle. Aber sobald die Programme etwas umfangreicher waren, wurde es immer schwieriger.

» Die übrige Zeit verbrachte ich damit, in den Sümpfen des südlichen Louisianas Schlangen und Schildkröten einzufangen. Wenn es draußen nicht so glühend heiß wäre, könnte sich dieses Buch sehr wohl auch um Amphibien- und Reptilienforschung drehen, statt um Programmierung. «

Zunächst bestand die Herausforderung nur darin, überhaupt etwas zum Funktionieren zu bekommen. Dann musste ich mir überlegen, wie ich Programme schreiben konnte, die aus mehr Zeilen bestanden, als ich mir gleichzeitig merken konnte. Anstatt Bücher wie »Programmierung in C« von Kernighan und Ritchie zu lesen, versuchte ich Fachliteratur zu finden, die sich mit dem *Organisieren* von Programmen befasste.

Zeitsprung. Mehrere Jahre später drückte mir ein Bekannter das Buch *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software* in die Hand. Endlich! Das war genau das Nachschlagewerk, nach dem ich seit meiner Teenagerzeit gesucht hatte. Ich las es in einem Rutsch von vorne bis hinten durch. Zwar hatte ich noch immer mit meinen eigenen Programmen zu kämpfen, aber mir fiel ein Stein vom Herzen, als ich begriff, dass andere Leute ähnliche Schwierigkeiten hatten und Lösungen dafür erarbeiteten. Ich hatte das Gefühl, dass ich statt »nur« meiner bloßen Hände nun endlich richtige *Werkzeuge* einsetzen konnte.

» Den Bekannten, der mir das Buch gab, lernte ich bei dieser Gelegenheit übrigens gerade erst kennen. Fünf Minuten nachdem wir einander vorgestellt wurden, saß ich auf seinem Sofa und verbrachte die nächsten Stunden mit Lesen, wobei ich ihn vollkommen ignorierte. Ich hoffe allerdings, dass sich meine Sozialkompetenz seitdem doch ein wenig verbessert hat.

2001 trat ich meinen Traumjob an: Programmierer bei Electronic Arts. Ich konnte es kaum erwarten, mir die *richtigen* Spiele anzusehen und herauszufinden, wie die Profis sie entwickeln. Wie sieht wohl die Architektur eines so gigantischen Spiels wie *Madden Football* aus? Wie arbeiten die verschiedenen Systemkomponenten zusammen? Wie schaffen sie es, eine einzige Codebasis auf mehreren Plattformen zum Laufen zu bringen?

Als ich den Quellcode sah, war das eine demutsvolle und überraschende Erfahrung. Der Code zur Grafikerzeugung, die künstliche Intelligenz (KI), die Animationen und visuellen Effekte – all das war einfach brilliant. Hier gab es Leute, die das letzte bisschen Leistung aus der CPU herauskitzeln und nutzen konnten. Sie erledigten Dinge, von denen ich gar nicht wusste, dass sie überhaupt *möglich* sind, schon vor der Mittagspause.

Aber der *Architektur*, in der sich dieser brillante Code befand, wurde kaum Beachtung geschenkt. Die Programmierer waren so sehr auf *Features* konzentriert, dass sie dabei die Organisation übersahen. Es wimmelte nur so von eng gekoppelten Modulen. Neue Features wurden der Codebasis da aufgepfropft, wo es gerade passte. Ernüchtert musste ich feststellen, dass es viele der Programmierer offenbar nicht weiter als bis zum *Singleton*-Pattern geschafft hatten – wenn sie das Buch *Design Patterns* denn überhaupt mal aufgeschlagen haben sollten.

Na ja, ganz so schlimm war es natürlich auch wieder nicht. Ich hatte mir allerdings ausgemalt, dass Spieleprogrammierer mithilfe von Wandtafeln wochenlang in aller Seelenruhe in einem Elfenbeinturm die architektonischen Einzelheiten ausdiskutierten – tatsächlich war der Code, den ich mir ansah, jedoch von Leuten

geschrieben worden, die sich mit engen Abgabefristen konfrontiert sahen. Sie gaben wirklich ihr Bestes und das war, so dämmerte mir allmählich, oft sehr gut. Je länger ich mich mit der Arbeit am Gamecode beschäftigte, desto mehr brillante Codeschnipsel entdeckte ich, die sich unter der Oberfläche versteckten.

Leider war »versteckt« häufig eine allzu treffende Beschreibung. Es waren echte Juwelen im Code verborgen, die viele völlig übersahen. Ich habe mehr als nur einmal erlebt, dass andere Programmierer sich damit abmühten, Dinge neu zu erfinden, obwohl in der Codebasis, mit der sie arbeiteten, bereits gute Lösungen für genau die betreffende Fragestellung vorhanden waren.

Dieses Problem möchte das vorliegende Buch lösen. Ich habe die besten in der Spieleprogrammierung vorkommenden Patterns ausgewählt, geordnet und zusammengestellt, damit wir unsere Zeit damit verbringen können, Dinge tatsächlich *neu* zu erfinden, anstatt sie *wieder* zu erfinden.

Darum geht es

Es gibt bereits Dutzende Bücher über Spieleprogrammierung – warum also ein weiteres schreiben? Die meisten mir bekannten Bücher zum Thema Spieleprogrammierung gehören einer der beiden folgenden Kategorien an:

- **Themenspezifische Bücher** Diese Bücher sind weitestgehend auf einen bestimmten Aspekt der Spieleprogrammierung fokussiert, wie etwa 3D-Grafik, Rendern in Echtzeit, Physiksimulation, künstliche Intelligenz oder Audio. Das sind die Gebiete, auf die sich viele Spieleprogrammierer im Lauf ihrer Karriere spezialisieren.
- **Bücher über komplette Spiel-Engines** Im Gegensatz zu der vorgenannten Kategorie versuchen diese Bücher, alle Bestandteile einer Engine abzuhandeln. Sie sind auf die Entwicklung einer kompletten Engine für ein bestimmtes Spielgenre ausgerichtet, meist 3D-Ego-Shooter.

Mir sagen die beiden Kategorien durchaus zu, aber ich denke, dass sie einige Lücken lassen. Themenspezifische Bücher erklären selten, wie der Code mit dem Rest des Spiels zusammenwirkt. Sie mögen vielleicht beim Rendern und der Physiksimulation ein alter Hase sein, aber wissen Sie auch, wie man beides vernünftig miteinander verbindet?

Dieser Bereich wird von der zweiten Kategorie zwar abgedeckt, ich finde die Bücher über komplette Spiel-Engines allerdings oft zu einseitig und genrebezogen. Mit dem Aufkommen von Smartphones und der damit einhergehenden Verbreitung mobiler Spiele und sogenannter *Casual Games* (einfache Spiele, die man bei Gelegenheit spielt) sind wir in einem Zeitalter angelangt, in dem viele verschiedene Spielegenres bedient werden. Wir klonen nicht mehr nur Quake. Bücher über Spiel-Engines helfen Ihnen nicht weiter, wenn *Ihr* Spiel nicht zur Engine passt.

Ich bemühe mich hier darum, die Themen eher wie auf einer Speisekarte zu präsentieren. Die einzelnen Kapitel im Buch beruhen auf jeweils unabhängigen Konzepten, die auf Ihren Code anwendbar sind. Auf diese Weise können Sie sie so zusammenstellen, wie es am besten zu dem Spiel passt, das *Sie* entwickeln möchten.

» Ein weiteres Beispiel für diesen »Speisekarten-Ansatz« ist die hochgelobte Buchreihe *Game Programming Gems*. «

Bezug zu Design Patterns

Jedes Buch über Programmierung, dessen Titel das Wort »Patterns« enthält, steht in einer gewissen Beziehung zu dem Klassiker *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides (die merkwürdigerweise als die »Gang of Four« bezeichnet werden).

» Das Buch *Design Patterns* wurde seinerseits von einem anderen Buch inspiriert. Die Idee, Patterns zur Beschreibung ergebnisoffener Problemlösungen zu verwenden, entstammt dem Buch *A Pattern Language* von Christopher Alexander (das er zusammen mit Sarah Ishikawa und Murray Silverstein verfasst hat).

Darin geht es zwar um Architektur (also *richtige* Architektur, Gebäude, Wände usw.), die Autoren hatten jedoch die Hoffnung geäußert, dass die von ihnen beschriebenen Lösungsstrukturen auch auf andere Bereiche angewendet würden – und *Design Patterns* ist der Versuch der Gang of Four, diese Methode auf Software zu übertragen. «

Mit dem Titel *Design Patterns für die Spieleprogrammierung* will ich keineswegs sagen, dass die Design Patterns der Gang of Four nicht auf Spiele anwendbar sind. Im Gegenteil: Das Kapitel *Design Patterns neu überdacht* stellt viele der klassischen Design Patterns vor, legt den Schwerpunkt dabei aber gezielt auf deren Anwendung in der Spieleprogrammierung.

Umgekehrt denke ich, dass dieses Buch auch auf Software anwendbar ist, bei der es sich nicht um Spiele handelt. Ich hätte ebenso gut einen Titel wie *Weitere Design Patterns* wählen können, ich finde aber, dass Spiele oft besonders ansprechende Beispiele ermöglichen. Sie wollen doch nicht wirklich schon wieder ein Buch über Angestelltendatensätze und Bankkonten lesen, oder?

Die hier vorgestellten Patterns sind nicht nur für andere Software nützlich, sie sind meiner Ansicht nach auch besonders gut für Probleme geeignet, die bei der Spieleprogrammierung häufig auftreten:

- Zeit und Abläufe sind oftmals das Kernstück einer Spielarchitektur. Vorgänge müssen in der richtigen Reihenfolge und zum richtigen Zeitpunkt stattfinden.
- Die Entwicklungszyklen folgen oft eng aufeinander und die Programmierer müssen in der Lage sein, schnell neue Versionen mit vielen verschiedenen

neuen Verhaltensweisen zu entwickeln, ohne sich dabei gegenseitig im Weg zu stehen oder überall in der Codebasis Spuren zu hinterlassen.

- Wenn die Verhaltensweisen festgelegt sind, folgen die Interaktionen. Monster beißen den Helden, Zaubertänke werden angerührt und Bomben sprengen Freund und Feind in die Luft. Solche Interaktionen müssen stattfinden können, ohne dass die Codebasis zu einem verworrenen Knäuel wird.
- Und zu guter Letzt kommt der Performance bei Spielen eine maßgebliche Bedeutung zu. Spieleentwickler befinden sich in einem dauerhaften Wettstreit, um herauszufinden, wer das meiste aus einer Plattform herausholen kann. Tricks und Kniffe, mit denen sich einige CPU-Takte einsparen lassen, können den Unterschied zwischen einem hervorragend bewerteten, millionenfach verkauften und einem von Bildaussetzern geprägten sowie genervten Spielern verrissenen Spiel bedeuten.

Hinweise zur Lektüre

Das vorliegende Buch ist in drei größere Teile gegliedert. Der erste (I) besteht aus der Einführung, die Sie gerade lesen, und dem folgenden Kapitel 1.

Der zweite Teil (II), *Design Patterns neu überdacht*, stellt einige der Entwurfsmuster aus dem Buch der Gang of Four vor. Ich lege in den einzelnen Kapiteln meine Ansichten zu den einzelnen Patterns dar und beschreibe, wie sie sich auf die Spieleprogrammierung anwenden lassen.

Danach folgt der Hauptteil des Buches (III bis VI), in dem 13 Design Patterns präsentiert werden, die sich als nützlich erwiesen haben. Sie sind in vier Kategorien unterteilt: Sequenzierungsmuster (III, Sequencing Patterns), Verhaltensmuster (IV, Behavioral Patterns), Entkopplungsmuster (V, Decoupling Patterns) und Optimierungsmuster (VI, Optimization Patterns). Die dazugehörigen Patterns werden jeweils auf einheitliche Art und Weise beschrieben, damit Sie das Buch als Referenz verwenden können und schnell finden, was Sie suchen:

- Der einleitende Abschnitt erläutert kurz und knapp die **Zielsetzung** des Design Patterns und beschreibt, für welche Problemstellungen es gedacht ist. Diese Informationen sollen es Ihnen ermöglichen, das Buch bei der Suche nach einem Pattern, das bei der Lösung eines Problems hilfreich wäre, schnell durchforsten zu können.
- Der Abschnitt **Motivation** beschreibt ein Anwendungsbeispiel des Patterns. Im Gegensatz zu konkreten Algorithmen ist ein Pattern mehr oder weniger formlos, solange es nicht auf ein bestimmtes Problem angewendet wird. Ein Pattern ohne ein Beispiel zu erklären, käme einem Backkurs gleich, in dem der Teig keine Erwähnung findet. Dieser Abschnitt liefert sozusagen den Teig, der in den nachfolgenden Abschnitten gebacken wird.

- Der Abschnitt **Das Pattern** fasst die wesentlichen Charakteristiken und Eigenschaften des im vorausgegangenen Beispiel dargestellten Entwurfsmusters zusammen. Wenn Sie eine trockene, lehrbuchhafte Beschreibung des Patterns suchen, werden Sie hier fündig. Dieser Abschnitt kann auch zur Wissensauffrischung dienen, wenn Ihnen der Gebrauch eines Patterns schon geläufig ist und Sie sich vergewissern möchten, dass Sie nichts vergessen haben.
- Bisher wurde das Pattern nur anhand eines einzigen Beispiels erläutert. Wie aber können Sie feststellen, ob es für *Ihr* Problem geeignet ist? Der Abschnitt **Anwendbarkeit** enthält einige Richtlinien dazu, wann die Verwendung des Patterns sinnvoll ist und wann man besser darauf verzichten sollte.
- Der Abschnitt **Konsequenzen** weist auf mögliche Risiken bei der Anwendung des Patterns hin.
- Wenn es Ihnen wie mir geht und Sie ein konkretes Beispiel brauchen, um etwas wirklich zu begreifen, sind Sie im Abschnitt **Beispielcode** richtig. Hier finden Sie eine schrittweise Implementierung des Patterns, damit Sie genau nachvollziehen können, wie es funktioniert.
- Patterns unterscheiden sich von Algorithmen dadurch, dass sie ergebnisoffen sind. Wenn Sie ein Pattern mehrmals verwenden, werden Sie es vermutlich jedes Mal anders implementieren. Der Abschnitt **Designentscheidungen** geht dem auf den Grund und zeigt verschiedene Möglichkeiten bei der Anwendung des Patterns auf.
- Und zum Abschluss wird in einem kurzen Abschnitt **Siehe auch ...** erklärt, in welcher Beziehung das Pattern zu anderen Patterns steht. Außerdem finden Sie hier Hinweise auf Open-Source-Code, der das Pattern in der Praxis einsetzt.

Über den Beispielcode

Die Codebeispiele in diesem Buch sind in C++ geschrieben, das heißt jedoch nicht, dass die Design Patterns nur in dieser Programmiersprache nützlich sind oder dass C++ besonders gut dafür geeignet wäre. Fast alle Programmiersprachen sind geeignet, allerdings setzen einige Patterns das Vorhandensein von Objekten und Klassen voraus.

Ich habe aus verschiedenen Gründen C++ gewählt. Zunächst einmal handelt es sich hierbei um die bei kommerziell vertriebenen Spielen verbreitetste Programmiersprache. C++ ist sozusagen die *Lingua franca* der Spielebranche. Außerdem bildet die C-Syntax, auf der C++ beruht, auch die Grundlage für Java, C#, JavaScript und viele andere Programmiersprachen. Selbst wenn Sie sich nicht mit C++ auskennen, stehen die Chancen daher nicht schlecht, dass Sie den Beispielcode mit ein klein wenig Mühe verstehen können.

Es ist *nicht* das Ziel dieses Buches, Sie C++ zu lehren. Die Beispiele sind so einfach wie möglich gehalten und sollen weder guten Programmierstil noch eine beson-

dere Verwendungsart darstellen. Denken Sie beim Betrachten des Codes an das dahinterstehende Konzept, nicht an den Code, der es formuliert.

Der Code wurde insbesondere nicht im »modernen« Stil (C++11 oder neuer) verfasst. Die Standardbibliothek wird nicht verwendet und Templates nur selten. Der C++-Code ist zwar nicht besonders elegant, aber er ist kompakt und ich hoffe, dass er auf diese Weise für Leute, die Erfahrung mit C, Objective-C, Java oder anderen Sprachen haben, leichter verständlich ist.

Um Platz zu sparen, werde ich in den Beispielen hin und wieder Code weglassen, wenn Sie ihn schon kennen oder er für ein Pattern nicht relevant ist. Diese Stellen sind durch Auslassungspunkte gekennzeichnet.

Betrachten Sie beispielsweise eine Funktion, die bestimmte Aufgaben erledigt und dann einen Rückgabewert liefert. Für das Pattern ist nur der Rückgabewert von Bedeutung, nicht die Erledigung der Aufgaben. Der Beispielcode sieht dann folgendermaßen aus:

```
bool update()
{
    // Aufgaben erledigen ...
    return isDone();
}
```

Wie geht es weiter?

Patterns sind ein Bestandteil der Softwareentwicklung, der einem ständigen Wandel und kontinuierlichen Erweiterungen unterliegt. Dieses Buch möchte den von der Gang of Four angestoßenen Prozess fortsetzen, die bislang entdeckten Patterns zu dokumentieren und mit anderen zu teilen. Und natürlich ist dieses Vorhaben mit dem vorliegenden Buch noch lange nicht abgeschlossen!

Sie selbst sind ebenfalls Teil dieses Prozesses. Wenn Sie Ihre eigenen Patterns entwickeln und die im Buch vorgestellten optimieren (oder sie ablehnen!), erweisen Sie der Community der Softwareentwickler ebenfalls einen Dienst. Sollten Sie Vorschläge haben, Fehler entdecken oder anderweitiges Feedback geben wollen, melden Sie sich bitte!

Architektur, Performance und Spiele

Bevor wir uns kopfüber in einen Haufen Patterns stürzen, ist es vermutlich hilfreich, wenn ich Ihnen an dieser Stelle zunächst kurz darstelle, wie ich persönlich über die Softwarearchitektur und deren Anwendung in der Spieleprogrammierung denke. Dadurch wird der Rest des Buches möglicherweise besser verständlich. Zumindest möchte ich Sie mit einigen schlagenden Argumenten bewaffnen, falls Sie demnächst in eine Diskussion hineingezogen werden, in der es darum geht, wie furchtbar (oder fabelhaft) Design Patterns sind.

» Beachten Sie bitte, dass ich keine Vermutungen darüber anstelle, auf welcher Seite Sie bei einer solchen Diskussion stehen würden. Wie jeder Waffenhändler, beliebere ich gern alle Beteiligten ...

1.1 Was ist Softwarearchitektur?

Wenn Sie dieses Buch von vorne bis hinten durchlesen, werden Sie anschließend weder die der 3D-Grafik zugrunde liegende lineare Algebra noch die für Physiksimulationen erforderliche Infinitesimalrechnung beherrschen. Es wird nicht erklärt, wie man einen binären Suchbaum rekursiv durchläuft oder wie man bei der Audiowiedergabe den Widerhall eines Raums simuliert.

» Meine Güte, der letzte Absatz wäre ein wirklich furchtbarer Werbetext für das Buch!

Dieses Buch thematisiert den Code, der all die genannten Dinge miteinander verknüpft. Es geht weniger darum, den Code zu schreiben, sondern vielmehr darum, ihn zu *organisieren*. Jedes Programm ist in *irgendeiner* Form organisiert – selbst wenn es dabei nur nach der Devise »Stopf den ganzen Krempel in main() und guck, was passiert« zugeht. Interessanter ist die Frage, was denn eigentlich eine *gute* Organisation auszeichnet. Wie unterscheidet sich eine gute Architektur von einer schlechten?

Über diese Frage grübele ich seit etwa fünf Jahren nach. Natürlich verfüge ich, ebenso wie Sie, über ein gewisses Gespür für gutes Design. Aber wir alle haben doch schon so schlechte Codebasen gesehen, dass es das Beste gewesen wäre, sie zu löschen, um sie von ihrem Leiden zu erlösen.

» Machen wir uns nichts vor: Die meisten von uns haben die eine oder andere davon sogar selbst fabriziert!

Einige wenige Glückliche haben aber auch die gegenteilige Erfahrung machen dürfen, mit wunderbar organisiertem Code zu arbeiten – der Sorte Codebasis, die sich anfühlt wie ein sorgfältig ausgewähltes Luxushotel voller Bediensteter, die eifrig darauf warten, den Gästen jeden Wunsch von den Augen abzulesen. Doch worin besteht dieser Qualitätsunterschied eigentlich?

1.1.1 Was zeichnet eine *gute* Softwarearchitektur aus?

Für mich bedeutet gutes Design, dass das gesamte Programm bestens für Änderungen, die ich gegebenenfalls daran vornehmen möchte, gerüstet ist. Ich kann eine Aufgabe durch einige wenige ausgewählte Funktionsaufrufe lösen, die sich problemlos einfügen lassen, ohne an der friedlich ruhenden Struktur des Codes zu kratzen.

Das klingt hübsch, ist aber tatsächlich kaum machbar. »Schreib den Code einfach so, dass Änderungen die ruhende Struktur des Codes unberührt lassen.« Jaja.

Lassen Sie mich das einmal etwas genauer aufschlüsseln. Entscheidend ist, dass es *bei der Architektur um Änderungen geht*. Irgendjemand muss die Codebasis ändern. Wenn der Code nicht geändert wird (sei es nun, weil er bereits perfekt ist, oder aber weil er so elendig ist, dass niemand seinen Texteditor damit besudeln möchte), ist das Design irrelevant. Ein Maßstab für die Qualität eines Designs ist die Einfachheit, mit der sich Änderungen vornehmen lassen. Eine Codebasis, an der keine Änderungen vorgenommen werden, ist wie ein Sprinter, der die Startlinie erst gar nicht verlässt.

1.1.2 Wie nimmt man Änderungen vor?

Bevor Sie den Code ändern, um eine neue Funktion hinzuzufügen, einen Fehler zu beheben oder warum auch immer Sie Ihren Texteditor sonst gestartet haben mögen, müssen Sie verstehen, wie der schon vorhandene Code funktioniert. Es ist natürlich nicht nötig, das gesamte Programm zu kennen, Sie müssen aber durchaus alle relevanten Bestandteile in Ihrem »Primatenhirn« hinterlegen.

» Es ist eine merkwürdige Vorstellung, dass es sich hierbei *buchstäblich* um eine Form der optischen Texterkennung handelt. «

Dieser Schritt wird gern unter den Teppich gekehrt, obwohl er zu den zeitraubendsten Aufgaben der Programmierung überhaupt gehört. Wenn Sie der Ansicht sind, dass das Paging einiger Daten von der Festplatte ins RAM zu langsam abläuft, dann versuchen Sie mal, das mit ganzen zwei Sehnerven und einem menschlichen Großhirn zu bewerkstelligen.

Sobald Sie Ihre kleinen grauen Zellen mit den richtigen Inhalten gefüttert haben, denken Sie ein wenig nach und gelangen schließlich zu einer Lösung. Sie werden vermutlich eine Weile Hin- und Herüberlegen, meistens funktioniert das aber

relativ unkompliziert – denn wenn Sie das Problem erst mal verstanden und die zugehörigen Stellen des Codes gefunden haben, ist die eigentliche Programmierung manchmal sogar trivial.

Also lassen Sie Ihre muskulösen Finger ein Weilchen über die Tastatur wandern, bis die richtigen »Lämpchen« auf dem Bildschirm angehen, und die Aufgabe ist erledigt, richtig? Nicht so hastig. Bevor Sie Tests für den Code programmieren und ihn zur Überprüfung einchecken können, müssen Sie nicht selten erst mal aufräumen.

» Ist da eben der Begriff »Tests« gefallen? Oh ja, tatsächlich. Das Programmieren von Unit-Tests kann bei manchen Spielen ziemlich schwierig sein, der Großteil der Codebasis lässt sich aber bestens testen.

Ich werde jetzt keine Predigt zu diesem Thema halten, möchte Sie aber an dieser Stelle bitten, mehr automatisierte Tests in Betracht zu ziehen (sofern Sie das nicht ohnehin schon tun). Oder haben Sie etwa nichts Besseres zu tun, als den Kram immer wieder von Hand zu überprüfen? «

Vielleicht haben Sie etwas zusätzlichen Code in Ihr Spiel eingebracht, möchten aber verhindern, dass der nächstbeste Programmierer über die Ecken und Kanten stolpert, die Sie im Code hinterlassen haben? Sofern es sich nicht gerade um eine sehr geringfügige Änderung handelt, ist oft eine gewisse Umstrukturierung nötig, damit sich Ihr neuer Code nahtlos in das übrige Programm einfügt – aber wenn Sie das richtig anstellen, wird der nächste Programmierer nicht feststellen können, zu welchem Zeitpunkt die einzelnen Codezeilen jeweils programmiert wurden.

Langer Rede, kurzer Sinn: Das Ablaufdiagramm der Programmierung sieht in etwa aus wie in Abbildung 1.1:

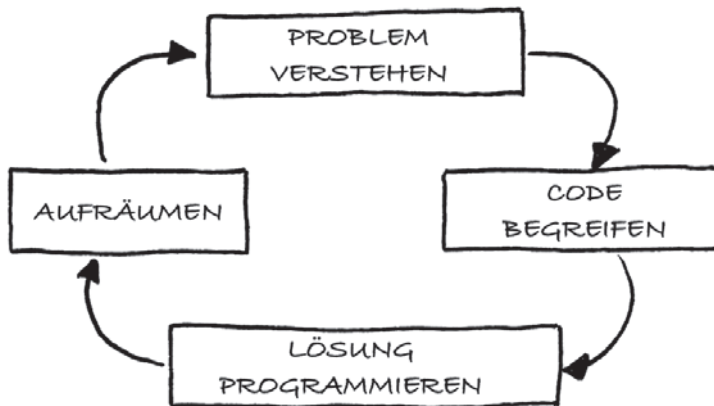


Abb. 1.1: Ihr Arbeitstag kurz zusammengefasst

» Wenn ich genauer darüber nachdenke, finde ich die Tatsache, dass es kein Entkommen aus dieser Schleife gibt, doch etwas beunruhigend. «

1.1.3 Inwiefern hilft eine Entkopplung?

Es ist nicht gerade offensichtlich, aber ich denke, dass ein Großteil der Softwarearchitektur von dieser Lernphase bestimmt wird. Den Code in die Hirnzellen aufzunehmen, geht so qualvoll langsam vonstatten, dass es sich lohnt, nach Strategien zur Verringerung seines Volumens zu suchen. Deshalb ist ein ganzer Teil dieses Buches den *Entkopplungsmustern* gewidmet, und darüber hinaus befasst sich auch ein guter Teil der *Design Patterns* mit diesem Konzept.

Der Begriff »Entkopplung« lässt sich auf verschiedene Weise definieren. Meiner Auffassung nach bedeutet die Entkopplung zweier Codeabschnitte, dass man den einen verstehen kann, ohne den anderen verstehen zu müssen. Wenn Sie den Code entkoppeln, können Sie über beide Teile unabhängig voneinander nachdenken. Das hat den enormen Vorteil, dass Sie nur denjenigen Code in Ihrem Gedächtnis aufnehmen müssen, der Ihr aktuelles Problem betrifft – und nicht auch noch die andere Hälfte.

Das ist meiner Ansicht nach das entscheidende Ziel der Softwarearchitektur: **Minimierung des Wissens, das man im Kopf haben muss, um Fortschritte machen zu können.**

Die nachfolgenden Schritte spielen natürlich auch eine Rolle. Eine andere Definition der Entkopplung besagt, dass man *Änderungen* an einem Teil des Codes vornehmen kann, ohne zugleich auch den anderen Teil ändern zu müssen. Es liegt auf der Hand, dass *irgendetwas* geändert werden muss, aber je geringer die Kopplung ist, desto weniger beeinflusst eine Änderung den Rest des Programms.

1.2 Zu welchem Preis?

Das klingt doch großartig, oder? Alles entkoppeln und schon kann man programmieren wie ein Wirbelwind. Jede Änderung betrifft nur ein oder zwei ausgewählte Methoden und Sie können auf der Struktur der Codebasis herumtanzen, ohne auch nur den Schatten einer Spur zu hinterlassen.

Das ist genau der Grund, warum die Leute sich für Abstrahierung, Modularität, Design Patterns und Softwarearchitektur begeistern. Es ist wirklich ein Vergnügen, mit einer guten Softwarearchitektur zu arbeiten – und produktiver zu sein, weiß jeder zu schätzen. Eine vernünftige Architektur bewirkt einen *immensen* Anstieg der Produktivität. Man kann gar nicht oft genug betonen, wie tiefgreifend die Auswirkungen sind.

Aber es gibt im Leben nun mal nichts umsonst. Eine gute Architektur erfordert Anstrengung und Disziplin. Bei jeder Änderung und jeder Ergänzung einer neuen Funktion sollten Sie stets um eine elegante Integration der entsprechenden Neuerung in den Rest des Programms bemüht sein. Sie müssen sorgfältig darauf