

Inhaltsverzeichnis

	Danksagungen	17
Teil I	Einführung	19
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
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	Instanziierte 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