Design Patterns

Design Patterns sind Vorlagen für die Softwareentwicklung, mit denen sich häufig auftretenden Probleme in ähnlicher Weise lösen lassen, unabhängig vom gegebenen Kontext. So macht es beispielsweise keinen Unterschied, ob man eine Website für einen kleinen Blumenladen schreibt oder ein Fernlenksystem für ballistische Raketen - Listen lassen sich in beiden Fällen mit dem Iterator-Pattern durchgehen. Design Patterns bieten Strukturen, die bereits erprobt sind und auf die man sich verlassen kann, wenn sie richtig umgesetzt werden. Außerdem sind sie auch für Entwickler, die neu zu einem Projekt stoßen eher verständlich als situationsspezifische bzw. individuelle Lösungen.

Design Strategies

Im Gegensatz zu Entwurfsmustern sind Entwurfsstrategien keine Lösungen für Probleme, sondern Ansätze, die zu diesen führen. Sie dienen dazu, Projekte zu strukturieren und Probleme, die es zu lösen gilt, zu identifizieren.

Teile und herrsche

Dieses Verfahren bildet die Grundlage von Entwurfsstrategien. Das Ziel ist es, das gesamte System in kleinere Probleme aufzuteilen, die dann unabhängig voneinander gelöst werden können. Dies hat den angenehmen Nebeneffekt, dass einzelne Module unter Umständen in anderen Projekten wiederverwertet werden können.

Function oriented vs. Object oriented

Beim funktionsorientierten Design wird das System in Teilsysteme aufgeteilt, die dann wieder unterteilt werden und so weiter. Am Ende liegt im Optimalfall eine Reihe voneinander unabhängiger Funktionen vor, die gemeinsam die Aufgaben des Systems erfüllen. Diese Vorgehensweise ist besonders geeignet, um Komponenten erneut verwenden zu können. Die Funktionen sollten dabei nicht vom Zustand des Systems abhängig sein, sondern idempotent. Die selbe Funktion soll also bei mehrfacher Ausführung mit den gleichen Parametern das gleiche Ergebnis liefern.
Das objektorientierte Design basiert dagegen nicht auf den Funktionalitäten des Systems, sondern auf Entitäten wie Kunden, Rechnungen oder Produkten sowie ihren Beziehungen untereinander. Das Ergebnis sind einige Objekte mit ihren jeweiligen Attributen und zugehörigen Funktionen.

Top-down vs. Bottom-up

Der Top-down-Ansatz beschäftigt sich damit, ein Gesamtsystem immer weiter aufzuteilen und dabei zu überlegen, welche Einzelkomponenten man aus der momentanen Ebene noch extrahieren kann. Diese Vorgehensweise ist bei neuen Systemen sinnvoll, wenn es keine Basis gibt, auf der man aufsetzen kann.
Das Gegenstück zu Top-down ist das Bottom-up-Schema, bei dem einzelne, gegebenenfalls bereits existente Komponenten zusammengesetzt und mit höheren Ebenen weiter

abstrahiert werden, bis die erwünschte Funktionalität des Gesamtsystems erfüllt ist.Das bietet sich an, wenn ein altes System neu gebaut wird und dabei Teile der alten Version verwendet werden sollen. In der Realität wird man vermutlich meistens eine Kombination aus beiden Ansätzen verwenden, um ein optimales Ergebnis zu erzielen.

Level of Implementation

Design Patterns können in Programmiersprachen auf drei Ebenen existieren: Invisible, Formal und Informal. Jede der Ebenen unterscheidet sich von den anderen durch den Aufwand der betrieben werden muss, um das Pattern zu verwenden.

Invisible

Ein Pattern ist in einer Sprache “unsichtbar” implementiert, wenn es so grundlegend in die Sprache eingebaut ist, dass man es nicht wahrnimmt. Ein gutes Beispiel hierfür ist die Verkapselung von Daten. In Sprachen wie C gibt es dafür ein Pattern, das “privat” und “öffentlich” in verschiedene Dateien trennt. Die Struktur sah dabei in etwa folgendermaßen aus:

```
project
|_main.c     (includes public.h)
|_public.h   (defines public fields and methods)
|_private.c  (includes public.h, defines private fields and methods,
             implements public and private fields and methods)
```

In vielen Sprachen wie C# ist das heutzutage überflüssig, da der Zugang mit den Keywords `public` und `private` (und `protected`, `internal`, …) geregelt werden kann, sodass keine mehreren Files nötig sind. (Code-Beispiel im Ordner code/encapsulation)
Ein weiteres Beispiel hierfür ist das Iterator-Pattern, das in C#-Listen implementiert ist und mit `foreach` automatisch aufgerufen wird. Mit diesem Muster lassen sich auch die anderen Levels anschaulich erklären, da es in C# mehr oder weniger auf allen Ebenen implementiert ist.

Formal

Eine formale Implementierung liegt dann vor, wenn ein Pattern im Sprachkern implementiert wurde, aber nicht durch etwas sprachenspezifisches wie `foreach` aufgerufen wird, sondern, z.B. im Falle des C#-Enumerators, mit `iterator.MoveNext()` und `iterator.Current`. Der Enumerator/Iterator ist dabei nicht von Grund auf selbst implementiert, sondern kann mit `list.GetEnumerator()` von einem Enumerable angefordert werden.

Informal

Eine informale Implementierung kann beispielsweise ein Interface sein, dass das Verhalten des Patterns definiert, die eigentliche Implementierung aber dem Entwickler überlässt. Das kann sinnvoll sein, um verschiedene Algorithmen, die das gleiche Pattern erfüllen, beispielsweise beim Sortieren oder Iterieren, einsetzen zu können und ähnelt dem

Bridge-Pattern (s.u.). In den meisten Fällen wird bereits eine Standard-Implementation vorhanden sein, anstatt derer man eine eigene verwenden kann.

Arten von Design Patterns

Structural

Strukturelle Design Patterns beschäftigen sich mit dem Aufbau der Entitäten und ihrer Relationen. Die wichtigsten Konzepte in diesem Bereich sind in objektorientierten Sprachen Vererbung und Interfaces.
Beispiele:

- **Adapter**: Verwendung von Legacy-Komponenten oder fremder Software
- **Bridge**: austauschbare Implementierung (z.B. verschiedene Sortieralgorithmen)
- **Composite**: Bestandteile werden als eigene Klassen dargestellt und können “zusammengebaut” werden

Creational

Creational Patterns erleichtern das Erstellen von Objekten oder sorgen dafür, dass weniger Ressourcen verbraucht werden. Sie ersetzen den Aufruf eines Konstruktors.
Beispiele:

- **Builder**: Vermeiden des Konstruktors beim Erstellen komplexer Objekte, Werte werden in der Build()-Funktion der Builder-Klasse gesetzt
- **Factory Method**: erstellt ein Objekt einer bestimmten abgeleiteten Klasse
- **Object Pool**: erstellt eine bestimmte Anzahl an Objekten zu Beginn der Laufzeit im Pool, nicht mehr benötigte Objekte werden recycled.

Behavioral

Verhaltensmuster beschreiben die Kommunikation zwischen Entitäten bei bestimmten Operationen und helfen Algorithmen strukturiert und wiederverwendbar zu gestalten.
Beispiele:

- **Interpreter**: Erkennung und Interpretation von Mustern und Strukturen z.B. in einem sprachlichen Ausdruck
- **Iterator**: Durchlaufen der Elemente von Datenstrukturen
- **Observer**: Callback-Methoden werden durch bestimmte Operationen aufgerufen

Sonstige Patterns

Daneben existieren noch Patterns, die nicht in eine dieser Kategorien passen, bzw. eine abstrahierte Form mehrerer Muster darstellen.

- **Dependency Injection**: kombiniert das Bridge-Pattern mit einem der Situation angemessenen Creational Pattern

- **MVC**: verwendet Observer- und Command-Pattern, um View und Model voneinander zu trennen.

Anti-Pattern

Neben den positiven Design Patterns existieren auch noch Muster, die als eine Art “How to not”-Anleitung fungieren sollen. Sie stellen verschiedene häufige “bad practices” aus den verschiedenen Bereichen der Projektarbeit dar. Während Design Patterns zeigen, wie man von einem Problem zu einer Lösung kommt, zeigen Anti-Patterns den Weg von einer *schlechten* Lösung zu einer *guten* Lösung auf.

Entwicklung

Anti-Patterns im Bereich der Entwicklung sollen zur Verbesserung der Code-Struktur anregen. Ein Beispiel ist der “Golden Hammer”, ein scheinbares Allheilmittel, das von Entwicklern für jede Situation verwendet wird, auch wenn es keinen Vorteil oder sogar Nachteile bringt. Aber auch Klassiker wie der Spaghetti-Code oder das Copy-Paste-Programming gehören dazu.

Architektur

Die Muster dieser Kategorie sollen helfen, die Architektur zu verbessern und damit die Struktur des Projektes zu vereinfachen oder Arbeit bzw. Zeit zu sparen. Häufige Probleme in diesem Bereich sind unter anderem das Swiss-Army-Knife, ein Programm (oder eine Klasse, Library, ...), das zwar alles kann, aber nichts richtig, und Reinvent the Wheel, der Versuch, ein bereits gelöstes Problem selbst zu lösen.

Management

Schlechte Angewohnheiten erstrecken sich nicht nur auf die eigentliche Software, sondern auch auf das ganze Drumherum. Projekte können früh an der Analyse-Paralyse oder dem “Tod durch Planung” zugrunde gehen, wenn die Projektleitung alles bis ins kleinste Detail analysieren bzw. alles von Anfang bis Ende durchplanen will. Und persönliche Fehden zwischen Abteilungsleitern, die auf dem Rücken ihrer Angestellten ausgetragen werden, können ganze Firmenbereiche lahmlegen. Management-Anti-Patterns skizzieren mögliche Probleme in der Projektleitung und Kommunikation.