# Zusammenfassung anti-patterns

Anti-patterns sind der „böse Gegenspieler“ der design patterns und beschreiben Muster und Vorgänge, die in dieser Form **nicht umgesetzt** werden sollten.

Im Folgenden sind neun anti-patterns kurz beschrieben und in die entsprechende Kategorie eingeordnet:

## Projektmanagement anti-patterns:

Viewgraph Engineering:

Entwickler sind mit dem Verfassen von Dokumenten für das Management beschäftigt und können ihre eigentlichen Fähigkeiten nicht einsetzen.

Lösung: Entwickler könnten Prototypen entwickeln, welche zur Validierung der Software eingesetzt werden. Diese können bestimmte Fragen u.U. besser beantworten als Dokumente.

Feature creep:

Der Projektplan wird nach der Festlegung nach und nach erweitert, z.B. durch den Kundeneinfluss.

Lösung: Der Projektplan sollte feststehen und bis auf einige kleine Anpassungen nicht geändert werden.

Brooks’sches Gesetz:

Zum aufholen verlorener Zeit werden neue Mitarbeiter eingestellt, welche das Projekt durch ihre Einarbeitungszeit noch weiter verspäten.

Lösung: Statt neuem Personal sollte über andere Lösungsansätze, wie z.B. Umverteilung der Aufgaben nachgedacht werden.

## Softwarearchitektur und Softwareentwurf anti-patterns:

Big ball of mud:

Die Software folgt keiner erkennbaren Softwarearchitektur.

Lösung: Beim design der Softwarearchitektur sollten kontinuierlich design patterns eingesetzt werden.

Sumo Marriage:

Sehr viel Logik wird „auf“ der Datenbank implementiert, damit sind Client und DB nicht entkoppelt.

Lösung: Es sollte auf eine generische Schnittstelle zwischen Client und DB geachtet werden, damit eine Migration oder ein DB Wechsel wenig/gar keinen Aufwand erfordern.

Spagetticode:

Der Kontrollfluss des Codes gleicht durch viele Sprungbefehle einem Topf mit Spagetti und ist somit kaum wartbar und nicht wiederverwendbar.

Lösung: Es sollte eine sinnvolle Architektur im Code erkennbar sein und Grundsätze der Kapselung und Aufgabenverteilung sollten eingehalten werden.

## Programmierung anti-patterns:

God class:

Es existiert eine Klasse, die zu viel „weiß“ und kontrolliert. Dies verletzt OO-Grundsätze und führt zu schwerer Wartbarkeit.

Lösung: OO-Grundsätze zur Kapselung und Aufgaben-/Verantwortungsverteilung sollten eingehalten werden, außerdem ist der Einsatz von design patterns empfohlen.

Lava flow:

Es existiert „toter“ Code, der unter keinen Umständen ausgeführt wird.

Lösung: Nach einer Änderung sollte überprüft werden, ob „toter“ Code entstanden ist und dieser gelöscht werden. Es sollte nicht weiter auf diesen Code aufgebaut werden.

Reinventing the wheel:

Eine bereits vorhandene Funktion wird erneut programmiert. Diese neue Version ist ggf. fehlerhaft und nicht performant.

Lösung: Vor der Implementierung einer neuen Funktion sollte geprüft werden, ob es nicht bereits einen entsprechende Funktion gibt, die genutzt werden kann.

## Weiterführende Referenzen

Unter den folgenden drei Links kann weiterführende Dokumentation zu anti-patterns und die Lösungen gefunden werden:

* <https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Architecture/Anti-Patterns>
* <https://sourcemaking.com/antipatterns>
* <http://wiki.c2.com/?AntiPattern>