What is an anti-pattern?

155

I am studying patterns and anti-patterns. I have a clear idea about patterns, but I don't get anti-patterns. Definitions from the web and Wikipedia confuse me a lot.

Can anybody explain to me in simple words what an anti-pattern is? What is the purpose? What do they do? Is it a bad thing or good thing?

60

design-patterns terminology anti-patterns ooad

edited Nov 5 '18 at 20:05
Machavity
25.1k 14 61 81

asked Jun 11 '09 at 11:33
g.revolution
5,045 18 70 98

3 sourcemaking.com/antipatterns – jaco0646 Jul 28 '16 at 16:38

2 martinfowler.com/bliki/AntiPattern.html – jaco0646 Jul 28 '16 at 16:43

12 Answers

199

Anti-patterns are certain patterns in software development that are considered bad programming practices.

As opposed to design patterns which are common approaches to common problems which have been formalized and are generally considered a good development practice, anti-patterns are the opposite and are undesirable.

For example, in object-oriented programming, the idea is to separate the software into small pieces called objects. An anti-pattern in object-oriented programming is a God object which performs a lot of functions which would be better separated into different objects.

For example:

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```
    function ReadFromFile() {}
    function WriteToFile() {}
    function DisplayToScreen() {}
    function PerformCalculation() {}
    function ValidateInput() {}
    // and so on... //
}
```

The example above has an object that does *everything*. In object-oriented programming, it would be preferable to have well-defined responsibilities for different objects to keep the code less coupled and ultimately more maintainable:

```
class FileInputOutput {
    function ReadFromFile() {}
    function WriteToFile() {}
}

class UserInputOutput {
    function DisplayToScreen() {}
    function ValidateInput() {}
}

class Logic {
    function PerformInitialization() {}
    function PerformCalculation() {}
}
```

The bottom line is there are good ways to develop software with commonly used patterns (design patterns), but there are also ways software is developed and implemented which can lead to problems. Patterns that are considered bad software development practices are anti-patterns.

edited Feb 24 '18 at 7:49

Randy Howard
2,065 10 26

answered Jun 11 '09 at 11:36

coobird
138k 31 199 222

7 any other examples of Anti-Patterns beside GodObject? – Tomasz Mularczyk Jul 9 '16 at 18:23

@Tomasz Programming Pasta serves is one such example. It's best generalized as poor encapsulation between many small objects. Consider it the opposite of the God object en.wikipedia.org/wiki/Spaghetti_code – AWrightIV Sep 6 '17 at 17:48

@Tomasz anything that is bad, but is done by some people, is an antipattern. E.g., `try: <do something>; except: pass` may be the Cardinal Sin antipattern in Python. See this: realpython.com/blog/python/… – neuronet Dec 14 '17 at 15:48

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

46

Whenever I hear about Anti-patterns, I recollect another term viz. Design smell.

"Design smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality". (From "Refactoring for Software Design Smells: Managing technical debt")

There are many design smells classified based on violating design principles:

Abstraction smells

Missing Abstraction: This smell arises when clumps of data or encoded strings are used instead of creating a class or an interface.

Imperative Abstraction: This smell arises when an operation is turned into a class.

Incomplete Abstraction: This smell arises when an abstraction does not support complementary or interrelated methods completely.

Multifaceted Abstraction: This smell arises when an abstraction has more than one responsibility assigned to it.

Unnecessary Abstraction: This smell occurs when an abstraction that is actually not needed (and thus could have been avoided) gets introduced in a software design.

Unutilized Abstraction: This smell arises when an abstraction is left unused (either not directly used or not reachable).

Duplicate Abstraction: This smell arises when two or more abstractions have identical names or identical implementation or both.

Encapsulation smells

Deficient Encapsulation: This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required.

Leaky Encapsulation: This smell arises when an abstraction "exposes" or "leaks" implementation details through its public interface.

Missing Encapsulation: This smell occurs when implementation variations are not encapsulated within an abstraction or hierarchy.

Unexploited Encapsulation: This smell arises when client code uses explicit type checks (using chained if-else or switch statements that check for the type of the object) instead of exploiting the variation in types already encapsulated within a hierarchy.

Modularization smells

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Insufficient Modularization: This smell arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both.

Cyclically-Dependent Modularization: This smell arises when two or more abstractions depend on each other directly or indirectly (creating a tight coupling between the abstractions).

Hub-Like Modularization: This smell arises when an abstraction has dependencies (both incoming and outgoing) with a large number of other abstractions.

Hierarchy smells

Missing Hierarchy: This smell arises when a code segment uses conditional logic (typically in conjunction with “tagged types”) to explicitly manage variation in behavior where a hierarchy could have been created and used to encapsulate those variations.

Unnecessary Hierarchy: This smell arises when the whole inheritance hierarchy is unnecessary, indicating that inheritance has been applied needlessly for the particular design context.

Unfactored Hierarchy: This smell arises when there is unnecessary duplication among types in a hierarchy.

Wide Hierarchy: This smell arises when an inheritance hierarchy is “too” wide indicating that intermediate types may be missing.

Speculative Hierarchy: This smell arises when one or more types in a hierarchy are provided speculatively (i.e., based on imagined needs rather than real requirements).

Deep Hierarchy: This smell arises when an inheritance hierarchy is “excessively” deep.

Rebellious Hierarchy: This smell arises when a subtype rejects the methods provided by its supertype(s).

Broken Hierarchy: This smell arises when a supertype and its subtype conceptually do not share an “IS- A” relationship resulting in broken substitutability.

Multipath Hierarchy: This smell arises when a subtype inherits both directly as well as indirectly from a supertype leading to unnecessary inheritance paths in the hierarchy.

Cyclic Hierarchy: This smell arises when a supertype in a hierarchy depends on any of its subtypes.

The above definition and classification is described in "Refactoring for software design smells: Managing technical debt". Some more relevant resources could be found here.

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

 **21.1k** 10 120 167  **592** 5 6

33

A pattern is an idea of how to solve a problem of some class. An anti-pattern is an idea of how not to solve it because implementing that idea would result in bad design.

An example: a "pattern" would be to use a function for code reuse, an "anti-pattern" would be to use copy-paste for the same. Both solve the same problem, but using a function usually leads to more readable and maintainable code than copy-paste.

answered Jun 11 '09 at 11:35

 sharptooth
123k 69 424 846

16

An anti-pattern is a way of not solving a problem. But there is more to it: it is also a way that can frequently be seen in attempts to solve the problem.

edited Feb 9 '18 at 12:01

 nbro
6,029 10 53 104

answered Jun 11 '09 at 11:38

 Ralph M. Rickenbach
10.7k 5 23 46

10

If you really wish to study AntiPatterns, get the book *AntiPatterns* (ISBN-13: 978-0471197133).

In it, they define "An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences."

So, if it's a bad programming practice but not a common one—limited to one application, one company or one programmer, it does not meet the "Pattern" part of the AntiPattern definition.

edited Oct 24 '12 at 16:53

 Keith Pinson
4,834 5 42 88

answered Jun 11 '09 at 12:26

 kmarsh
1,114 6 18

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

9

edited Oct 24 '12 at 16:32
Keith Pinson
4,834 5 42 88

answered Jun 11 '09 at 11:37
Robert Gould
42.2k 53 168 262

6

An *anti-pattern* is the complement of a *design pattern*. An anti-pattern is a template solution you should not use in a certain situation.

edited Feb 9 '18 at 11:59
nbro
6,029 10 53 104

answered Aug 29 '15 at 13:24
xxmajia
408 5 5

6

Just like with a *design pattern*, an *anti-pattern* is also a template and a repeatable way of solving a certain problem, but in a non-optimal and ineffective way.

edited Feb 9 '18 at 12:01
nbro
6,029 10 53 104

answered Jun 11 '09 at 11:40
Darnell
779 4 8

5

Interestingly a given way of solving a problem can be both a pattern and an anti-pattern. Singleton is the prime example of this. It will appear in both sets of literature.

answered Jun 11 '09 at 12:32
Ed Sykes
1,149 8 15

3

Today, software engineering researchers and practitioners often use the terms “anti-pattern” and “smell” interchangeably. However, they are conceptually not the same. The Wikipedia entry of anti-pattern states that an anti-pattern is different from a bad practice or a bad idea by at least two factors. An anti-pattern is

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

It clearly indicates that an anti-pattern is chosen in the belief that it is a good solution (as a pattern) to the presented problem; however, it brings more liabilities than benefits. On the other hand, a smell is simply a bad practice that negatively affects the quality of a software system. For example, Singleton is an anti-pattern and God class (or Insufficient Modularization) is a design smell.

answered May 2 '16 at 15:25

Tushar
458 4 9

2

Anti-patterns are common ways people tend to program the wrong way, or at least the not so good way.

answered Feb 9 '10 at 8:20

Roman A. Taycher
5,514 14 71 122

0

Any design pattern that is doing more harm than good to the given software development environment would be considered as anti-pattern.

Some anti-pattern are obvious but some are not. For example Singleton, even though many consider it good old design pattern but there are others who don't.

You can check question What is so bad about singletons? to better understand the different opinions on it.

answered Mar 13 '18 at 11:32

Himanshu Negi
1 1

Actually, anti-patterns generally are not obvious. Obviously bad design patterns are simply bad design patterns. A genuine anti-pattern looks tenable on the surface, but manifests problems later on. In fact, not being obviously bad is the distinction that makes them an anti-pattern in the first place. – hawkeyegold May 18 '18 at 16:54

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.