Anti-pattern

An **anti-pattern** is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.[1][2] The term, coined in 1995 by Andrew Koenig,[3] was inspired by a book, *Design Patterns*, which highlights a number of design patterns in software development that its authors considered to be highly reliable and effective.

The term was popularized three years later by the book *AntiPatterns*, which extended its use beyond the field of software design to refer informally to any commonly reinvented but bad solution to a problem. Examples include analysis paralysis, cargo cult programming, death march, groupthink and vendor lock-in.

Contents

Definition

According to the authors of *Design Patterns*, there must be at least two key elements present to formally distinguish an actual anti-pattern from a simple bad habit, bad practice, or bad idea:

1. A commonly used process, structure, or pattern of action that despite initially appearing to be an appropriate and effective response to a problem, has more bad consequences than good ones.
2. Another solution exists that is documented, repeatable, and proven to be effective.

Examples

Social and business operations

Organizational

- Analysis paralysis: A project stalled in the analysis phase, unable to achieve support for any of the potential plans of approach
- Bicycle shed: Giving disproportionate weight to trivial issues
- Bleeding edge: Operating with cutting-edge technologies that are still untested and/or unstable, leading to cost overruns, under-performance, and/or delayed delivery
- Bystander apathy: The phenomenon in which people are less likely to or do not offer help to a person in need when others are present

- Cash cow: A profitable legacy product that often leads to complacency about new products
- Design by committee: The result of having many contributors to a design, but no unifying vision
- Escalation of commitment: Failing to revoke a decision when it proves wrong
- Groupthink: A collective state where group members begin to (often unknowingly) think alike and reject differing viewpoints
- Management by objectives: Management by numbers, focus exclusively on quantitative management criteria, when these are non-essential or cost too much to acquire
- Micromanagement: Ineffectiveness from excessive observation, supervision, or other hands-on involvement from management
- Moral hazard: Insulating a decision-maker from the consequences of their decision
- Mushroom management: Keeping employees "in the dark and fed manure" (also "left to stew and finally canned")
- Peter principle: Continually promoting otherwise well-performing employees up to their level of incompetence, where they remain indefinitely
- Seagull management: Management in which managers only interact with employees when a problem arises, when they "fly in, make a lot of noise, dump on everyone, do not solve the problem, then fly out"
- Stovepipe or Silos: An organizational structure of isolated or semi-isolated teams, in which too many communications take place up and down the hierarchy, rather than directly with other teams across the organization
- Typecasting: Locking successful employees into overly safe, narrowly defined, predictable roles based on their past successes rather than their potential
- Vendor lock-in: Making a system excessively dependent on an externally supplied component

Project management

- Cart before the horse: Focusing too many resources on a stage of a project out of its sequence
- Death march: A project whose staff, while expecting it to fail, are compelled to continue, often with much overwork, by management which is in denial
- Ninety-ninety rule: Tendency to underestimate the amount of time to complete a project when it is "nearly done"
- Overengineering: Spending resources making a project more robust and complex than is needed
- Scope creep: Uncontrolled changes or continuous growth in a project's scope, or adding new features to the project after the original requirements have been drafted and accepted (also known as requirement creep and feature creep)
- Smoke and mirrors: Demonstrating unimplemented functions as if they were already implemented
- Brooks' law: Adding more resources to a project to increase velocity, when the project is already slowed down by coordination overhead.

Software engineering

Software design

- Abstraction inversion: Not exposing implemented functionality required by callers of a function/method/constructor, so that the calling code awkwardly re-implements the same functionality in terms of those calls
- Ambiguous viewpoint: Presenting a model (usually Object-oriented analysis and design (OOAD)) without specifying its viewpoint
- Big ball of mud: A system with no recognizable structure
- Database-as-IPC: Using a database as the message queue for routine interprocess communication where a much more lightweight mechanism would be suitable
- Gold plating: Continuing to work on a task or project well past the point at which extra effort is adding value
- Inner-platform effect: A system so customizable as to become a poor replica of the software development platform
- Input kludge: Failing to specify and implement the handling of possibly invalid input
- Interface bloat: Making an interface so powerful that it is extremely difficult to implement
- Magic pushbutton: A form with no dynamic validation or input assistance, such as dropdowns

- Race hazard: Failing to see the consequences of events that can sometimes interfere with each other
- Stovepipe system: A barely maintainable assemblage of ill-related components

Object-oriented programming

- Anemic domain model: The use of the domain model without any business logic. The domain model's objects cannot guarantee their correctness at any moment, because their validation and mutation logic is placed somewhere outside (most likely in multiple places). Martin Fowler considers this to be an anti-pattern, but some disagree that it is always an anti-pattern.[4]
- Call super: Requiring subclasses to call a superclass's overridden method
- Circle-ellipse problem: Subtyping variable-types on the basis of value-subtypes
- Circular dependency: Introducing unnecessary direct or indirect mutual dependencies between objects or software modules
- Constant interface: Using interfaces to define constants
- God object: Concentrating too many functions in a single part of the design (class)
- Object cesspool: Reusing objects whose state does not conform to the (possibly implicit) contract for re-use
- Object orgy: Failing to properly encapsulate objects permitting unrestricted access to their internals
- Poltergeists: Objects whose sole purpose is to pass information to another object
- Sequential coupling: A class that requires its methods to be called in a particular order
- Yo-yo problem: A structure (e.g., of inheritance) that is hard to understand due to excessive fragmentation

Programming

- Accidental complexity: Programming tasks which could be eliminated with better tools (as opposed to essential complexity inherent in the problem being solved)
- Action at a distance: Unexpected interaction between widely separated parts of a system
- Boat anchor: Retaining a part of a system that no longer has any use
- Busy waiting: Consuming CPU while waiting for something to happen, usually by repeated checking instead of messaging
- Caching failure: Forgetting to clear a cache that holds a negative result (error) after the error condition has been corrected
- Cargo cult programming: Using patterns and methods without understanding why
- Coding by exception: Adding new code to handle each special case as it is recognized
- Design pattern: The use of patterns has itself been called an anti-pattern, a sign that a system is not employing enough abstraction[5]
- Error hiding: Catching an error message before it can be shown to the user and either showing nothing or showing a meaningless message. This anti-pattern is also named *Diaper Pattern*. Also can refer to erasing the Stack trace during exception handling, which can hamper debugging.
- Hard code: Embedding assumptions about the environment of a system in its implementation
- Lasagna code: Programs whose structure consists of too many layers of inheritance
- Lava flow: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences[6][7]
- Loop-switch sequence: Encoding a set of sequential steps using a switch within a loop statement
- Magic numbers: Including unexplained numbers in algorithms
- Magic strings: Implementing presumably unlikely input scenarios, such as comparisons with very specific strings, to mask functionality.
- Repeating yourself: Writing code which contains repetitive patterns and substrings over again; avoid with once and only once (abstraction principle)
- Shotgun surgery: Developer adds features to an application codebase which span a multiplicity of implementors or implementations in a single change
- Soft code: Storing business logic in configuration files rather than source code[8]
- Spaghetti code: Programs whose structure is barely comprehensible, especially because of misuse of code structures

Methodological

- Copy and paste programming: Copying (and modifying) existing code rather than creating generic solutions
- Every Fool Their Own Tool: Failing to use proper software development principles when creating tools to facilitate the software development process itself.[9]
- Golden hammer: Assuming that a favorite solution is universally applicable (See: Silver bullet)
- Improbability factor: Assuming that it is improbable that a known error will occur
- Invented here: The tendency towards dismissing any innovation or less than trivial solution originating from inside the organization, usually because of lack of confidence in the staff
- Not Invented Here (NIH) syndrome: The tendency towards *reinventing the wheel* (failing to adopt an existing, adequate solution)
- Premature optimization: Coding early-on for perceived efficiency, sacrificing good design, maintainability, and sometimes even real-world efficiency
- Programming by permutation (or "programming by accident", or "programming by coincidence"): Trying to approach a solution by successively modifying the code to see if it works
- Reinventing the square wheel: Failing to adopt an existing solution and instead adopting a custom solution which performs much worse than the existing one
- Silver bullet: Assuming that a favorite technical solution can solve a larger process or problem
- Tester Driven Development: Software projects in which new requirements are specified in bug reports

Configuration management

- Dependency hell: Problems with versions of required products
- DLL hell: Inadequate management of dynamic-link libraries (DLLs), specifically on Microsoft Windows
- Extension conflict: Problems with different extensions to classic Mac OS attempting to patch the same parts of the operating system
- JAR hell: Overutilization of multiple JAR files, usually causing versioning and location problems because of misunderstanding of the Java class loading model

See also

- Code smell – symptom of unsound programming
- Design Smell
- List of software development philosophies – approaches, styles, maxims and philosophies for software development
- Software Peter principle
- Capability Immaturity Model
- ISO 29110: Software Life Cycle Profiles and Guidelines for Very Small Entities (VSEs)

References

1. Budgen, D. (2003). *Software design* (https://books.google.com/books?id=bnY3vb606bAC&pg=PA225&dq=%22anti-pattern%22+date:1990-2003). Harlow, Eng.: Addison-Wesley. p. 225. ISBN 0-201-72219-4. "As described in Long (2001), design anti-patterns are 'obvious, but wrong, solutions to recurring problems'."
2. Scott W. Ambler (1998). *Process patterns: building large-scale systems using object technology* (https://books.google.com/books?id=qJJk2yEeoZoC&pg=PA4&dq=%22anti-pattern%22+date:1990-2001). Cambridge, UK: Cambridge University Press. p. 4. ISBN 0-521-64568-9. "...common approaches to solving recurring problems that prove to be ineffective. These approaches are called antipatterns."
3. Koenig, Andrew (March–April 1995). "Patterns and Antipatterns". *Journal of Object-Oriented Programming*. **8** (1): 46–48.; was later re-printed in the: Rising, Linda (1998). *The patterns handbook: techniques, strategies, and applications* (https://books.google.com/books?id=HBAuixGMYWEC&pg=PT1&dq=0-521-64818-1). Cambridge, U.K.: Cambridge University Press. p. 387. ISBN 0-521-64818-1. "An antipattern is just like a pattern, except that instead of a solution it gives something that looks superficially like a solution, but isn't one."

4. "The Anaemic Domain Model is no Anti-Pattern, it's a SOLID design" (https://blog.inf.ed.ac.uk/sapm/2014/02/04/the-anaemic-domain-model-is-no-anti-pattern-its-a-solid-design/). *SAPM: Course blog*. 4 February 2014. Retrieved 3 January 2015.
5. "Revenge of the Nerds" (http://www.paulgraham.com/icad.html). "In the OO world you hear a good deal about "patterns". I wonder if these patterns are not sometimes evidence of case (c), the human compiler, at work. When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough— often that I'm generating by hand the expansions of some macro that I need to write."
6. Lava Flow (http://www.antipatterns.com/lavaflow.htm) at antipatterns.com
7. "Undocumented 'lava flow' antipatterns complicate process" (http://www.icmgworld.com/corp/news/Articles/RS/jan_0202.asp). Icmgworld.com. 14 January 2002. Retrieved 3 May 2010.
8. Papadimoulis, Alex (10 April 2007). "Soft Coding" (http://thedailywtf.com/Articles/Soft_Coding.aspx). thedailywtf.com. Retrieved 27 June 2011.
9. https://www.linkedin.com/pulse/every-fool-his-own-tool-marcel-heemskerk-msc-scea

10. Peter, Lawrence J. (1969), *The Peter Principle: Why Things Always Go Wrong*; 1969 Buccaneer Books, ISBN-13: 9781568491615

11. Yourdon, Edward (1997), *Death March*; ISBN-13: 978-0137483105

Further reading

1. Laplante, Phillip A.; Neill, Colin J. (2005). *Antipatterns: Identification, Refactoring and Management*. Auerbach Publications. ISBN 0-8493-2994-9.
2. Brown, William J.; Malveau, Raphael C.; McCormick, Hays W.; Thomas, Scott W. (2000). Hudson, Theresa Hudson, ed. *Anti-Patterns in Project Management*. John Wiley & Sons. ISBN 0-471-36366-9.

External links

- Anti-pattern at WikiWikiWeb
- Anti-patterns catalog
- AntiPatterns.com Web site for the AntiPatterns book
- Patterns of Toxic Behavior
- C Pointer Antipattern
- Email Anti-Patterns book
- Patterns of Social Domination