Anti-pattern

From Wikipedia, the free encyclopedia

In software engineering, an **anti-pattern** (or **antipattern**) is a pattern that may be commonly used but is ineffective and/or counterproductive in practice.[1][2]

The term was coined in 1995 by Andrew Koenig,[3] inspired by Gang of Four's book *Design Patterns*, which developed the concept of design patterns in the software field. The term was widely popularized three years later by the book *AntiPatterns*, which extended the use of the term beyond the field of software design and into general social interaction. According to the authors of the latter, 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:

- Some repeated pattern of action, process or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results, and
- A refactored solution exists that is clearly documented, proven in actual practice and repeatable.

Many anti-pattern ideas amount to little more than mistakes, rants, unsolvable problems, or bad practices to be avoided if possible. Sometimes called *pitfalls* or *dark patterns*, this informal use of the term has come to refer to classes of commonly reinvented bad solutions to problems. Thus, many candidate anti-patterns under debate would not be formally considered anti-patterns.

By formally describing repeated mistakes, one can recognize the forces that lead to their repetition and learn how others have refactored themselves out of these broken patterns.

Contents

Known anti-patterns

Organizational anti-patterns

- Analysis paralysis: Devoting disproportionate effort to the analysis phase of a project

- 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
- Management by perkele: Authoritarian style of management with no tolerance of dissent
- Management by objectives: Management by numbers, focus exclusively on quantitative management criteria, when these are non-essential or cost too much to acquire.
- Moral hazard: Insulating a decision-maker from the consequences of his or her decision
- Mushroom management: Keeping employees uninformed and misinformed; employees are described as being kept in the dark and fed manure, left to stew, and finally canned.
- Stovepipe or Silos: A structure that supports mostly up-down flow of data but inhibits cross organizational communication
- Vendor lock-in: Making a system excessively dependent on an externally supplied component[4]

Project management anti-patterns

- Avalanche: An inappropriate mashup of the Waterfall model and Agile Development techniques
- Death march: Everyone knows that the project is going to be a disaster – except the CEO – so the truth is hidden to prevent immediate cancellation of the project - (although the CEO often knows and does it anyway to maximize profit). However, the truth remains hidden and the project is artificially kept alive until the Day Zero finally comes ("Big Bang"). Alternative definition: Employees are pressured to work late nights and weekends on a project with an unreasonable deadline.
- Groupthink: During groupthink, members of the group avoid promoting viewpoints outside the comfort zone of consensus thinking
- Overengineering: Spending resources making a project more robust and complex than is needed
- Smoke and mirrors: Demonstrating unimplemented functions as if they were already implemented
- Software bloat: Allowing successive versions of a system to demand ever more resources
- Waterfall model: An older method of software development that inadequately deals with unanticipated change

Analysis anti-patterns

- Bystander apathy: When a requirement or design decision is wrong, but the people who notice this do nothing because it affects a larger number of people

Software design anti-patterns

- Abstraction inversion: Not exposing implemented functionality required by users, so that they re-implement it using higher level functions
- 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: Coding implementation logic directly within interface code, without using abstraction
- Race hazard: Failing to see the consequence of different orders of events
- Stovepipe system: A barely maintainable assemblage of ill-related components

Object-oriented design anti-patterns

- Anemic Domain Model: The use of 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).
- BaseBean: Inheriting functionality from a utility class rather than delegating to it
- 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 anti-patterns

- Accidental complexity: Introducing unnecessary complexity into a solution
- Action at a distance: Unexpected interaction between widely separated parts of a system
- Blind faith: Lack of checking of (a) the correctness of a bug fix or (b) the result of a subroutine
- 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 reset an error flag when an error 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
- Error hiding: Catching an error message before it can be shown to the user and either showing nothing or showing a meaningless message
- Hard code: Embedding assumptions about the environment of a system in its implementation
- Lava flow: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences[5][6]
- 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: Including literal strings in code, for comparisons, as event types etc.
- Repeating yourself: Writing code which contains repetitive patterns and substrings over again; avoid with once and only once (abstraction principle)
- Soft code: Storing business logic in configuration files rather than source code[7]
- Spaghetti code: Programs whose structure is barely comprehensible, especially because of misuse of code structures

- Shotgun surgery: Developer adds features to an application codebase which span a multiplicity of implementors or implementations in a single change.

Methodological anti-patterns

- Copy and paste programming: Copying (and modifying) existing code rather than creating generic solutions
- 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
- 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"): 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 anti-patterns

- 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 pre-Mac OS X versions of the Mac OS attempting to patch the same parts of the operating system
- JAR hell: Overutilization of the 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
- 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* (http://books.google.com/?id=bnY3vb606bAC&pg=PA225&dq=%22anti-pattern%22+date:1990-2003) . Harlow, Eng.: Addison-Wesley. p. 225. ISBN 0-201-72219-4. http://books.google.com/?id=bnY3vb606bAC&pg=PA225&dq=%22anti-pattern%22+date:1990-2003. "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* (http://books.google.com/?id=qJJk2yEeoZoC&pg=PA4&dq=%22anti-pattern%22+date:1990-2001) . Cambridge, UK: Cambridge University Press. p. 4. ISBN 0-521-64568-9. http://books.google.com/?

id=qJJk2yEeoZoC&pg=PA4&dq=%22anti-pattern%22+date:1990-2001. "...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* (http://books.google.com/?id=HBAuixGMYWEC&pg=PT1&dq=0-521-64818-1) . Cambridge, U.K.: Cambridge University Press. p. 387. ISBN 0-521-64818-1. http://books.google.com/?id=HBAuixGMYWEC&pg=PT1&dq=0-521-64818-1. "Anti-pattern is just like pattern, except that instead of solution it gives something thats looks superficially like a solution, but isn't one."
4. ^ Vendor Lock-In (http://www.antipatterns.com/orig/vendorlockin.htm) at antipatterns.com
5. ^ Lava Flow (http://www.antipatterns.com/orig/lavaflow.htm) at antipatterns.com
6. ^ "Undocumented 'lava flow' antipatterns complicate process" (http://www.icmgworld.com/corp/news/Articles/RS/jan_0202.asp) . Icmgworld.com. 2002-01-14. http://www.icmgworld.com/corp/news/Articles/RS/jan_0202.asp. Retrieved 2010-05-03.
7. ^ Papadimoulis, Alex (2007-04-10). "Soft Coding" (http://thedailywtf.com/Articles/Soft_Coding.aspx) . thedailywtf.com. http://thedailywtf.com/Articles/Soft_Coding.aspx. Retrieved 2011-06-27.

Further reading

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

External links

- Anti-pattern (http://c2.com/cgi/wiki?AntiPattern) at WikiWikiWeb
- Anti-patterns catalog (http://c2.com/cgi/wiki?AntiPatternsCatalog)
- AntiPatterns.com (http://www.antipatterns.com) Web site for the AntiPatterns book
- Patterns of Toxic Behavior (http://www.personal.psu.edu/cjn6/Personal/Antipatterns-%20Patterns%20of%20Toxic%20Behavior.htm)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Anti-pattern&oldid=474842892"
Categories: Anti-patterns | Software architecture | Design

- This page was last modified on 3 February 2012 at 20:08.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
  Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.