ANTI-PATTERNS VS PATTERNS: WHAT IS AN ANTI-PATTERN?

Jargon permeates the software development industry. Best practices. Artifacts. Scope Creep. Many of these terms are so common as to be called overused, and it is easy to assume we understand them because they seem so obvious. Still, we sometimes find new depth when we examine them closely. In this post, let us muse on the "Pattern," and its somewhat lesser known counterpart, the "Anti-Pattern."

Patterns

We all know what patterns generally are in common language, but to understand their importance in software engineering it's important to first discuss algorithms. An algorithm is simply a way of performing a common task, such as sorting a list of items, storing data for efficient retrieval, or counting occurrences of an item within a data set.

Algorithms are one of the oldest, most fundamental concepts in software engineering. Indeed, on this author's desk sits a copy of what is considered by many to be one of the most seminal works on the subject, "Fundamental Algorithms" by Donald Knuth. The First Edition of this small tome of just over 600 pages was first copyrighted in 1968, 50 years ago.

The text would be nearly unrecognizable to a modern programmer, as it mainly emphasizes Calculus-based proofs of its solutions and its only code examples are provided in obscure, outdated languages such as Algol or MIX Assembly. Despite this, much of what was covered is still used today: singly- and double-linked lists, trees, garbage collection, etc. The details are often buried in

convenient libraries, but the concepts are the same. These algorithms have remained valid solutions to common software engineering problems for more than 5 decades and are still going strong.

A "pattern" can be considered a more general form of an algorithm. Where an algorithm might focus on a specific programming task, a pattern might consider challenges beyond that realm and into areas such as reducing defect rates, increasing maintainability of code, or allowing large teams to work more effectively together. Some common patterns include:

- **Factories** - An evolution of early object-oriented programming concepts that eliminated the need for the creator of an object to know everything about it ahead of time. A flowchart application might support extensible stencil libraries by focusing on creating and organizing "shapes," allowing the stencils themselves to manage the details of creating a simple square vs. a complex network router icon.
- **Pub/Sub** - A mechanism for "decoupling" applications. Rather than having a sender directly send messages to a receiver, the sender "publishes" the messages to a topic or queue. One or more receivers can "subscribe" to receive those messages, and the message queue handles details such as transmission errors or resending messages. This simplifies both the sending and receiving applications.
- **Public-key Cryptography** - A mechanism by which two parties can communicate securely and without interception, yet without the need to pre-arrange an exchange of secret encryption keys. Each party maintains a pair of keys (public and private), and the public key can often be obtained as needed rather than exchanged in advance.
- **Agile** - A philosophy that encapsulates a set of guiding principles for software development that emphasize customer satisfaction, embrace the need for flexibility and collaboration, and promote the adoption of simple, sustainable development practices.

These are just four of the many common patterns in the industry, and even in this mix we can see how they range from highly technical to broader, more process-oriented points. Factories are a very code-oriented pattern, while pub/sub is more architectural in nature. And while public-key cryptography has broad implications, libraries to support its operations are available for nearly every programming language in common use today, making it generally straightforward to implement.

At the other end of the spectrum, "Agile" remains somewhat elusive: simultaneously a rallying point and an instrument of divisiveness among developers, project managers, and other stakeholders about exactly what it means and how it should be implemented. It is a great example of an overused yet poorly understood term. Seeing the terms "Waterfall" or "Stand ups" in the same sentence as "Agile" is almost always an example of misuse. Agile is a philosophy, not a software development methodology, so it cannot be directly compared to Waterfall, nor does it directly spell out process components such as stand ups. (Those are a component of Scrum, a methodology that *implements* Agile principles, but does not represent Agile itself.)

Narrow or broad, technical or process-oriented, a good working knowledge of these patterns is an essential component in a technologist's toolbox.

What is an Anti-Pattern?

If a "pattern" is simply a known-to-work solution to a common software engineering problem, wouldn't an "anti-pattern" simply be the opposite? A non-Agile development methodology, or a tightly-coupled application?

Actually, anti-patterns do not just incorporate the concept of failure to do the right thing, they also include a set of choices that seem right at face value, but lead to trouble in the long run. Wikipedia defines the term "Anti-pattern" as follows:

"An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive."

Note the reference to "a common response." Anti-patterns are not occasional mistakes, they are common ones, and are nearly always followed with good intentions. As with regular patterns, anti-patterns can be broad or very specific, and when in the realms of programming languages and frameworks, there may be literally hundreds to consider. Here are just a few of this author's high-level, personal favorites:

Whiteboard programming challenges in software interviews

David Hansson, creator of Ruby on Rails and the Founder and CTO of Basecamp, once tweeted "Hello, my name is David. I would fail to write bubble sort on a whiteboard. I look code up on the internet all the time. I don't do riddles." The anti-pattern here is evaluating the wrong metrics during an interview, such as where a typical task assignment will be "Add zip code lookup during registration" but interview questions sound like "Sort this array in pseudocode using functional programming concepts."

Remember the "good intentions" aspect of anti-patterns? It seems as if we are testing the candidate on a valuable principle: knowledge of fundamentals. However, programming is often a ruthlessly pragmatic practice, and this focus on theoretical knowledge over practical skills and experience might cause us to choose a candidate that meets our cultural ideals, but lacks the actual skills required to be successful in the position.

Put another way: if StackOverflow will be a regular resource used by the developer in the position, it should be available (and used) during the interview. Homework assignments and pair programming challenges may also be worth exploring.

All patterns and anti-patterns have valid exceptions. A developer whose job will be to make libraries of algorithms for others to use may very well need to know the Calculus behind a mechanism. The error here is applying this expectation universally, even to developers who will not be doing so.

Moral Hazard

In philosophical contexts, Moral Hazard is the separation of individuals from the consequences of their decisions. This sounds like an obvious behavior to avoid, but this anti-pattern is the root cause of many SDLC inefficiencies.

Consider the traditional QA process, in which "tickets" are addressed by developers, then passed to QA for review before being deployed. There are two problems here. First, staffing ratios are almost never "1 developer to 1 QA analyst," and even a handful of developers can easily exceed the capacity of the QA team. Second, this insulates developers from the consequences of their mistakes by making it another individual's responsibility to find them before they are released - a moral hazard.

The effects of this anti-pattern can be subtle: if the QA team is effective, it may not directly lead to lower quality output. It is more likely to show up in other areas such as complaints about estimation

accuracy and missed targets. Quality and estimation accuracy suffer because developers instinctively focus on "getting things through QA" rather than shipping high quality software. Even with a modest defect rate of 20-30% (a number which even might be optimistic in many organizations), the churn this produces can significantly impact team productivity.

Additional anti-patterns often arise in the attempt to solve the problem. In Scrum, it may be tempting to make sprints longer or hold them open. But a sprint is meant to be a measure of time, not a measure of output. This act reverses that nature, which destroys the value of other tools such as "velocity" metrics that are based upon it. It is also common to see longer sprint planning or pre-planning meetings to more deeply review tickets. But this attempts to convert an instinctive process into a scientific one, forgetting that the purpose for implementing a methodology like Scrum was to acknowledge this impossibility in the first place.

Two patterns that are often effective at resolving this issue include:

- Embracing a culture of continuous improvement: "ship it when it's better, not when it's right." (Also see "Polishing the Cannonball" below). Developers encouraged and empowered to do this can make better decisions about how they address their tasks, and also experience a more tangible sense of personal accomplishment.
- Make developers responsible for their work product all the way through to Production deployments. Facebook, Google, and other industry titans have all reported success with this approach.

Polishing the Cannonball

Sometimes also known as "gold plating" or "boiling the ocean," trying to ship perfect products often significantly increases project timelines and costs without actually increasing the value delivered. A closely related anti-pattern is the "zombie ticket," the plaque on the arterial walls of the Backlog. Zombie tickets are never a high enough priority to get cleaned out, but are never closed for fear of losing the documentary record of the task.

The problem with both habits is that the metrics that support them are phantoms. Unshipped features have zero value to customers, and tasks that do not cause enough pain to become priorities may never be worth addressing. It is almost always better to focus available resources on regularly delivering new, valuable features rather than on constantly looking backward on small issues that affect very few users.

The "pattern" counterpart here is the minimally viable product (MVP), which often ends up being a bit of a phantom itself. (MVPs are almost never as small as planned or hoped for.) However, the act of attempting to ship an MVP is itself often an antidote to the problems listed above, so even if some slippage does occur it is still worth the effort. Iterative development processes also address this by emphasizing regular, predictable delivery of incremental value, reinforced by feedback from actual end users.

Further Reading

There are enough patterns and anti-patterns in the industry to fill books, and indeed many have been written about them. In the end, it is usually not necessary to memorize lists of them, although developers specializing in certain languages or frameworks should be encouraged to research those specifically targeted at those areas.

If you are interested in learning about more patterns and anti-patterns, I have found these resources to be valuable in my own reading:

- Wikipedia's "Software Design Patterns" and "Anti-Patterns" pages provide good examples of high level topics.
- SourceMaking's "Design Patterns" and "AntiPatterns" contain useful specifics about general software engineering tasks.
- ***Enough Rope to Shoot Yourself in the Foot: Rules for C and C++ Programming***, Alan Holub, McGraw Hill, 1995. Although this book is focused on C and C++, most of the rules it covers still apply to nearly any modern language.

In addition, nearly every language or framework has dozens of resources available if you simply search for "NodeJS Patterns" or similar in any search engine. I would encourage every developer to do this for their particular fields: even when we think we know good vs. bad practices, it can sometimes be surprising when we find ourselves following an anti-pattern - always with the best of intentions, of course!