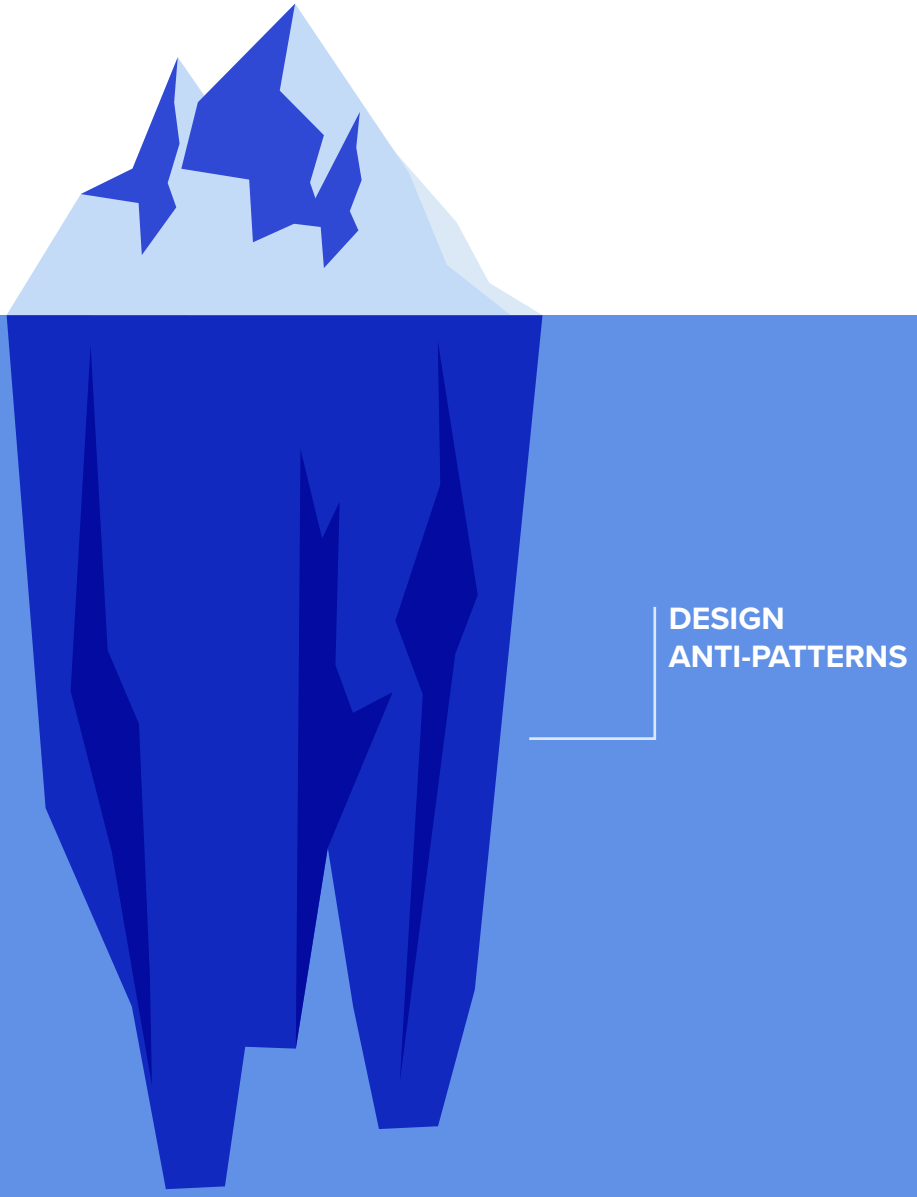



# Technical Debt That You Don't See

The unknown truth about technical debt in software engineering

Vivek Reghunath

embold

# Technical debt - why should you care?

Technical debt is the price a software engineering team has to pay for all of the quick fixes and shortcuts taken over a while. Technical debt can be accumulated due to business pressure or incomplete requirements definition which invariably results in suboptimal code design.

Most engineers are not sufficiently aware of the implications of accumulating technical debt. Well, what are the implications anyway?

Technical debt slows down any engineering team as the understandability and maintainability of the code base degrades with mounting debt. Teams will spend more time fixing bugs and less time developing new features. Due to poor understandability, onboarding of new engineers will take considerably more time slowing down the team even further.

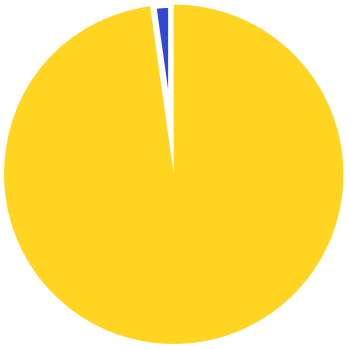
If technical debt is left unattended, repositories (and related projects) will eventually reach the tipping point of sustainability. This tipping point will force the team to reengineer everything from the start, and force the team into building with improved practices for sustainable code design.

Teams spend an average of

**42%** [1]

time servicing technical debt
(17.3 hours/week/developer)

- Reduced team efficiency
- Increased onboarding time
- Service disruption
- Increased risk of bugs
- Unmaintainable repositories
- Depleted team morale



**99%** [3]

of total technical debt comes from design anti-patterns

# Sources of technical debt

Technical debt can come from localized issues such as vulnerabilities(including “security issues”), code issues and design anti-patterns. The first two (vulnerabilities and code issues) are usually located on a given line of code and are generally easy to find and fix.

Software developers generally focus on easy to fix issues such as code issues (code smells) and vulnerabilities. While there is nothing wrong with this approach, this is a less effective approach to address technical debt and reduce maintenance costs over time.

Design anti-patterns are manyfold more problematic and should be targeted for appreciable technical debt control. We will evaluate and quantify the negative effects of design anti-patterns and their contribution to the technical debt of software projects.

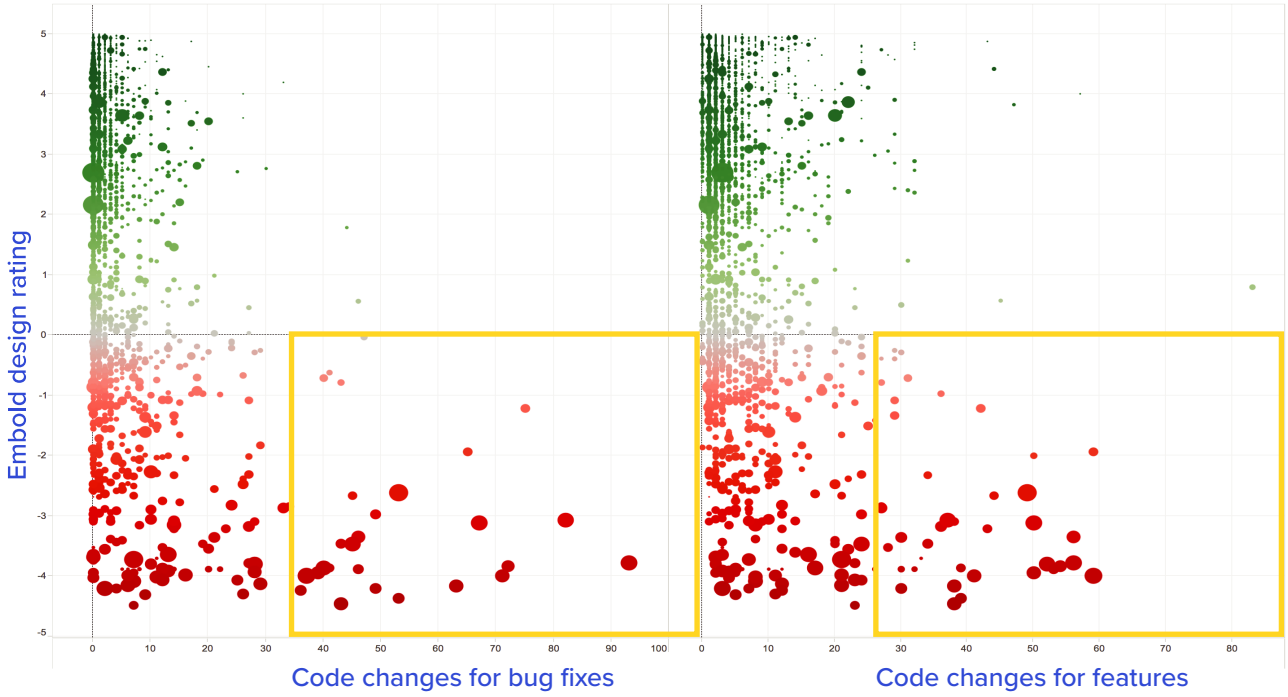
# What are design anti-patterns ?

In software engineering, an anti-pattern [7] is a pattern that seems to work but is counter-productive and far from optimal in practice. An anti-pattern can easily result in unmaintainable and error-prone code. Usually, anti-patterns emerge when new functionality is added incrementally without focus on continuous refactoring.

*Read more at : https://docs.embold.io/anti-patterns/*

# Negative effects of design anti-patterns

Code changes vs Embold design rating [2]

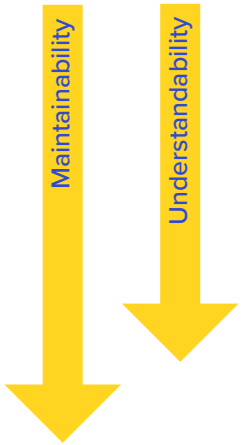



*Read more about embold rating: https://docs.embold.io/embold-score/#embold-rating-system*

- Code components with a low Embold design rating are frequently involved in bugs and features (tasks)
- This means they go through frequent changes and are difficult to maintain. If not refactored, they can lead to an increased risk of bugs and maintainability issues.

Let's consider an Embold scan of the Apache CloudStack project [5]. Design anti-patterns bring down almost every KPI [6]. Maintainability gets significantly affected. Significant compromise in robustness and accuracy KPIs [6] is also observed. Another important degradation is in analyzability and understandability. This means, adding new features becomes slow as more and more anti-patterns are added.

Accelerated deterioration of engineering efficiency can be observed with increased design anti-pattern density. With a high number of anti-patterns, adding new features or fixing bugs becomes slow and results in high code churn.

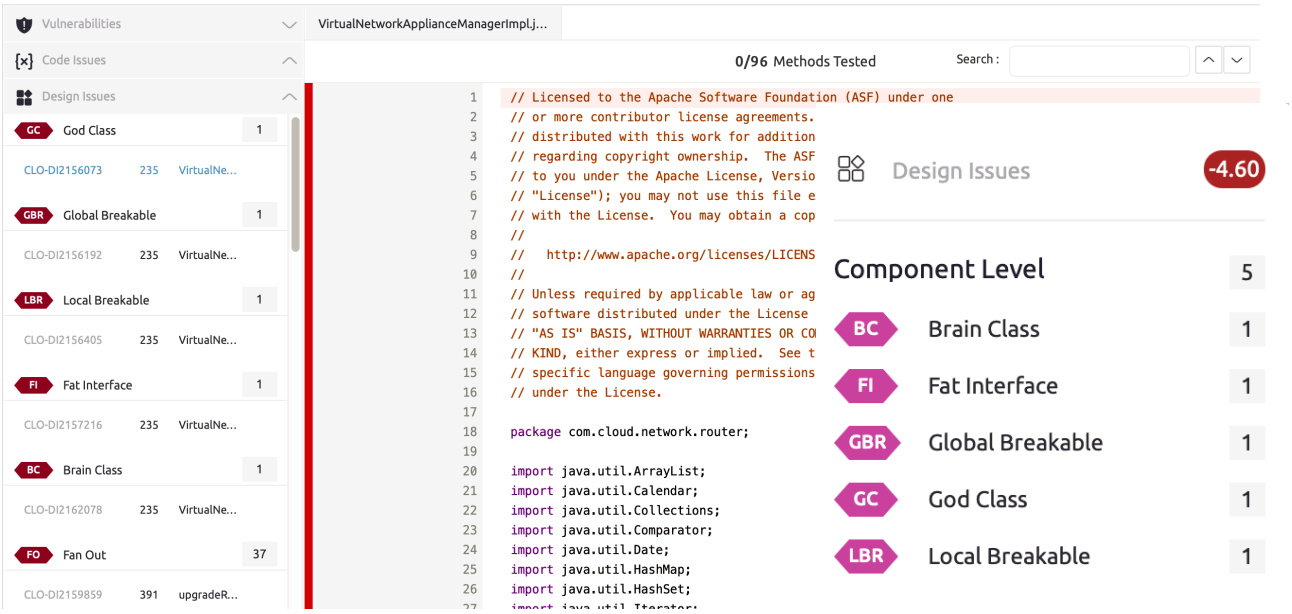



KPI Dashboard

3.0.1.prerelease5
Aug 11, 2021 2:10 pm

| Hotspot ratio in modules | Accuracy | Analyzability | Changeability | Efficiency | Functionality | Maintainability | Portability | Resource Utilization | Robustness | Security | Test KPI | Understandability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| core | 121 | 638 | 27 | 139 | 45 | 676 | 21 | 33 | 686 | 218 | | 322 |
| utils | 44 | 283 | 60 | 74 | 42 | 384 | 22 | 53 | 189 | 83 | | 91 |
| server | 753 | 1.8K | 41 | 607 | 41 | 2.7K | 13 | 42 | 822 | 141 | 1 | 907 |

# Calculating technical debt




Explore Embold scan results [5] to understand the context of the below calculations.

Class **VirtualNetworkAppliancemanagerImpl** is a hotspot from project Apache CloudStack with **3413** lines of code and a whopping **96** methods! This class has **5** class-level design anti-patterns [7] and **110** method-level design anti-patterns [7].

Based on the code refactoring case study by Embold [3], effort estimation for fixing various kinds of issues is as given below.

| Issue Type | Time required to fix |
|---|---|
| Class level anti-pattern | 5 days [3] |
| Method/function level anti-pattern | 1 day [3] |
| Code issue / vulnerability | 2 minutes [3] |

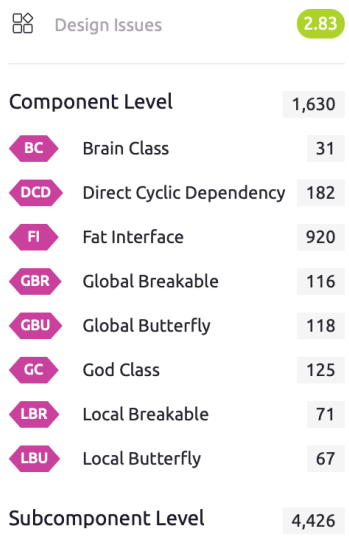
Total debt from Class level anti-pattern = 5 * 5 = 25 days
Total debt from Method level anti-pattern = 1 * 110 = 110 days
Total technical debt due to anti-patterns for the class = 25 + 110 = **135** days

This class has only 4 code issues that amount to a technical debt of **8** minutes!

Considering just code issues and vulnerabilities for technical debt calculation gives a false sense of maintainability as we considerably underestimate the debt.

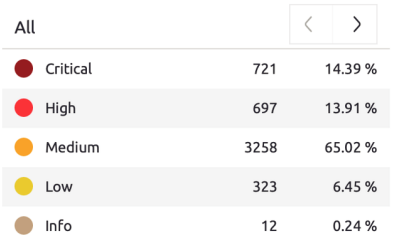
# The big picture

Now let's calculate technical debt from different sources for the project Apache CloudStack.



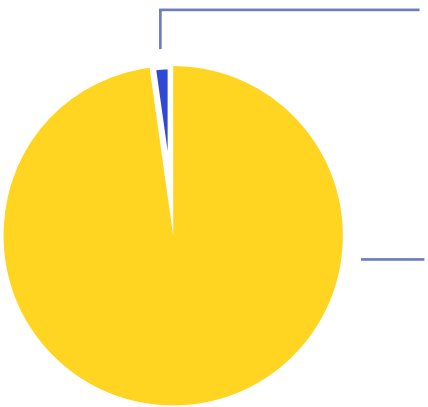

Total debt from Class level anti-pattern =
5 * 1630 = 8,150 days
Total debt from Method level anti-pattern =
1 * 4426 = 4,426 days

**Total debt from all design anti-patterns** =
8150 + 4426 = **12,576** days.




Total code issues = 5,011
Total debt from code issues = 5011 x 2 = 10,022 minutes.

**Total debt from code issues** = 167 Hrs = **21** days.




As we can see, to reduce technical debt and bring down its ill effects in a meaningful way, it is essential to address design anti-patterns.

# References


1. Striple's 2018 study 'The Developer Coefficient: a $300B opportunity for businesses':
   https://stripe.com/en-gb-de/reports/developer-coefficient-2018

2. Design refactoring with Embold partitioning tool - a case study:
   https://embold.io/wp-content/uploads/2021/11/Gamma_Partitioning_CaseStudy.pdf

3. Code refactoring case study by Embold:
   https://embold.io/wp-content/uploads/2021/11/Embold_Refactoring_CaseStudy.pdf

4. Embold rating system :
   https://docs.embold.io/embold-score/#embold-rating-system

5. Explore Embold scan results:
   https://explore.embold.io/
   Username : explorer@embold.io
   Password : explorer

6. Embold software KPIs:
   https://docs.embold.io/key-performance-indicator/

7. Embold design anti-patterns:
   https://docs.embold.io/anti-patterns/