Wyobraź sobie, że masz na imię Mirek i kupujesz

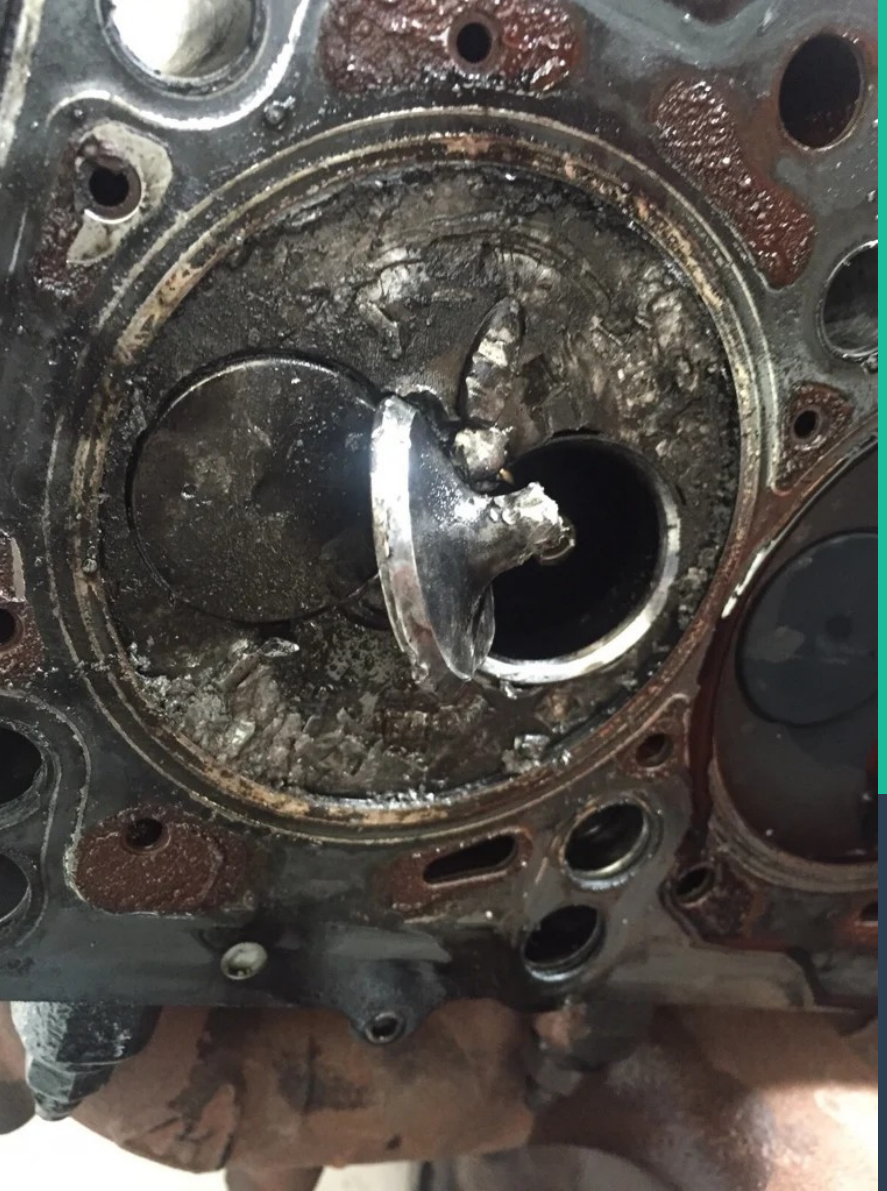
pięknego Passata B5

Prestiż, blichtr i splendor.


Diesel musi dymić!
Jadęęę!

Turbo i wtryski do wymiany, ale może pojeździ.

Potem się będę **martwić**



Mirek nagle musi spłacić dług, który zaciągneli poprzedni właściciele jego samochodu.

Cześć!

Ja jestem **Max**

i Opowiem wam dzisiaj o:

długu technicznym

Refactoring.

Jak pozostać przy zdrowych zmysłach, redukując dług?

Max Małecki - PHPers Summit 2022

Szybkie

3 pytania

Gotowi?

Czy do grzebania w serwisie napisanym przez Grześka zakładasz latexowe rękawiczki?

Ile razy P.O. obciął ci czas na funkcjonalność bez redukcji zakresu?

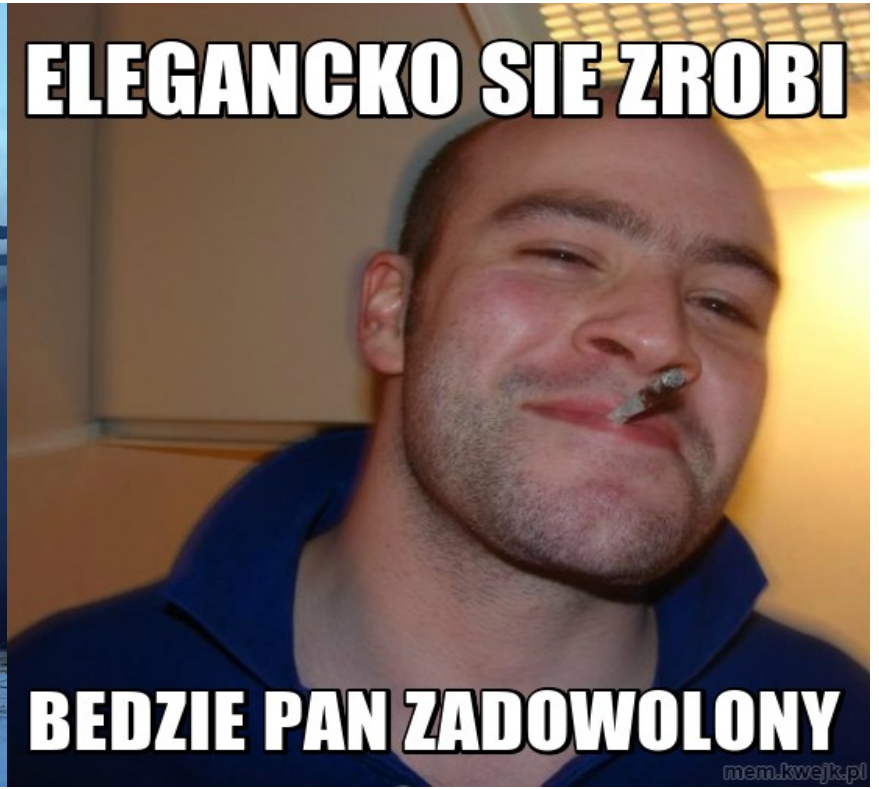
Czy twoje estymacje zawsze są zbyt optymistyczne?

To znak, że twój projekt może być

Zadłużony.

Zadłużony technicznie.

Czym jest dług techniczny?



Czym jest ten dług?

Dla Mirka?	Kosztem remontu silnika.
Dla programisty?	Stratą czasu, frustracją.
Dla projektu?	Kolejnymi opóźnieniami.
Dla firmy?	Stratą pieniędzy.

Dług zawsze **rośnie** wraz z rozrostem złożoności

code-base projektu.

Skąd pochodzi dług?

To wynik odwiecznego **kompromisu** między

jakością a **delivery**

Wynika to z praw

Prawa ewolucji oprogramowania

M. Lehmana

Continuing Change

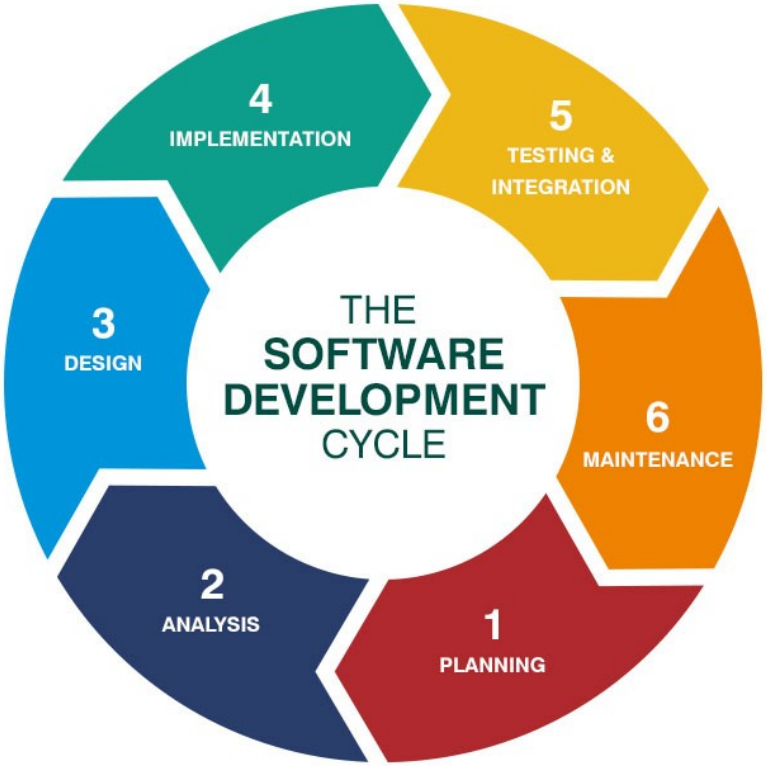
A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful.

Increasing Complexity

As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.

M. Lehman: „Programs, Life Cycles, and Laws of Software Evolution". PROCEEDINGS OF THE IEEE, VOL. 68, NO. 9 , SEPTEMBER 1980

Proces powstawania ~~oprogramowania~~ długu.

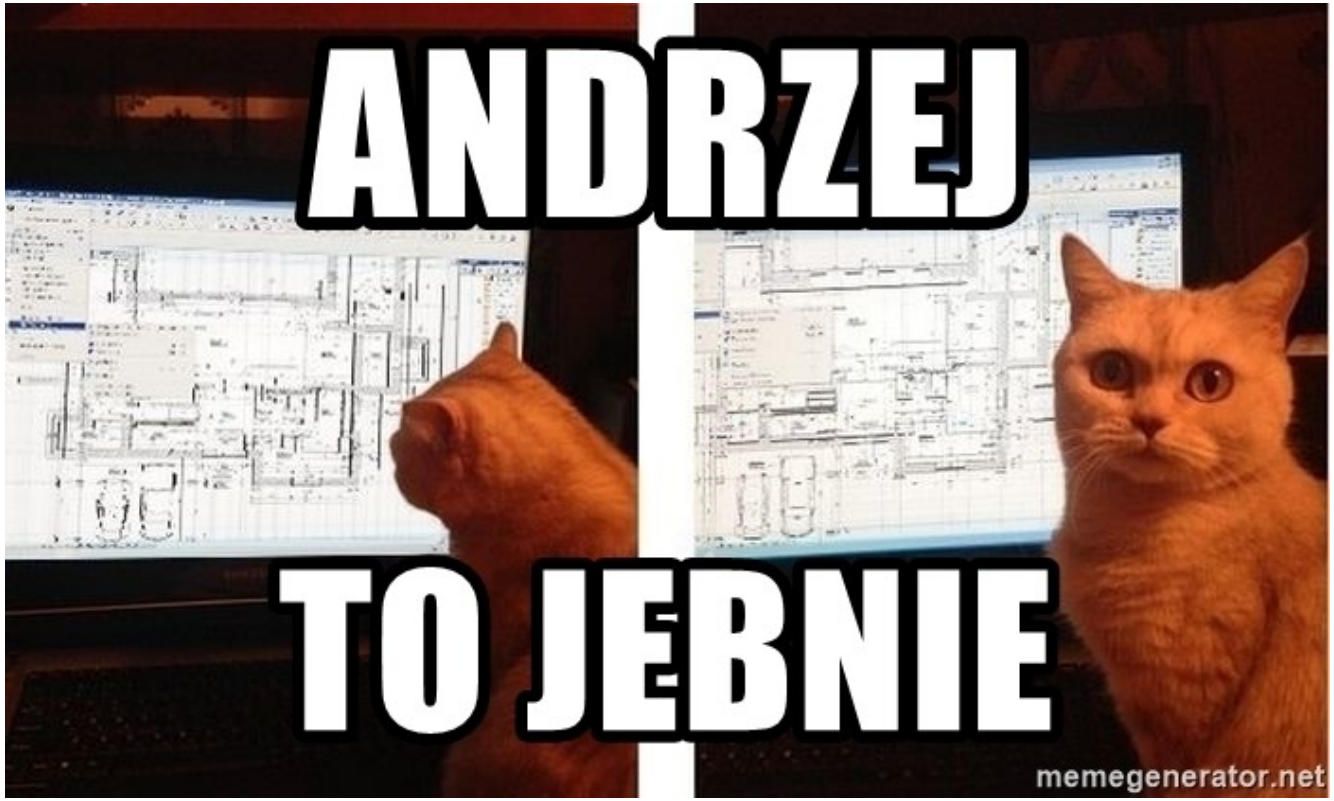


- Dług można zaciągnąć na każdym z sześciu etapów,
- Od złego planowania,
- Poprzez kiepską analizę,
- Z której to wynika zły design,
- Źle zaprojektowane rozwiązanie, nie może być dobrze zakodowane
- Testy możemy zrobić później,
- A utrzymując system łatamy tylko objawy nie szukając źródła problemu.

Czym grozi niepanowanie nad długiem?

- Więcej bugów – rosną koszty utrzymania aplikacji.
- Trudniej dodać nowe funkcjonalności, żeby nie rozsypać produkcji – rosną koszty wdrożenia nowych funkcjonalności.
- Estymacje nie pokrywają się z rzeczywistością.
- Spada uptime aplikacji – klient nie zarabia, a Ty po nocach usuwasz awarie bo SLA i kary umowne!
- Osiągniecie Masy Krytycznej Długu. „Projekt zaorać. Przepisujemy go w nowej technologii x”

Aż w końcu:



Czy dług jest

policzalny?

Tak.

Można przedstawić skalę długu jako

liczbę godzin/lat potrzebną na zlikwidowanie go.

Ale bez kontekstu fakt, że masz 1000 godzin długu do spłaty nie znaczy zupełnie

Nic.

Jak walczyć z długiem?

Refaktoryzować!

Ale co właściwie

refaktoryzować?!

O tym za chwilę.

Do **refaktoringu** należy zaplanować

fragmenty **kodu** które:

Low Cohesion

Zajmują się wieloma rzeczami naraz.

KISS

Są niezrozumiałe lub nazbyt zawiłe.

Risky Code

Często pojawiaja się w nich bugi

CODE SMELLS

np. naruszające **SOLID**

Code Smells

The Bloaters	The Object-Orientation Abusers	The Change Preventers	The Dispensables	The Couplers
Large method	**Refused bequest**	**Code scattering**	**Duplicated code**	**Inappropriate intimacy**
Large class		Parallel Hierarchy	Lazy class	Feature envy
Long parameters list			Dead code	

Cyclomatic Complexity

Najtrudniejsze w zrozumieniu, gdzie dług kosztuje najwięcej

High Coupling

Jeżeli nie da się napisać testu do fragmentu kodu.
Wiedz że coś się dzieje.

Co jeszcze śmierdzi długiem?

Ekstra lista potencjalnych dłużników:

- Braki w dokumentacji
- Braki w testach
- Nieaktualne vendory
- Trzymanie się kurczowo niewspieranych już vendorów
- Hotfixy późną nocą – bez planu na zrobienie tego dobrze

Nie da się tego jakoś

zautomatyzować?

Pewnie, że się da.

- SonarQube
- PHPMD - PHP Mess Detector
- j6s/phparch
- PHP_CodeSniffer
- PHP CS Fixer
- PHPStan
- Psalm
- Scrutinizer
- PHPUnit – CRAP mode

Ale biznes nie da mi budżetu na ogarnięcie całego długu!


wat

Ogarnąć cały dług?

O’RLY?

Walcz z długiem tam, gdzie on

kosztuje najwięcej.

Ale jak policzyć co

kosztuje

najwięcej?

Znajdź

hotspoty

w kodzie

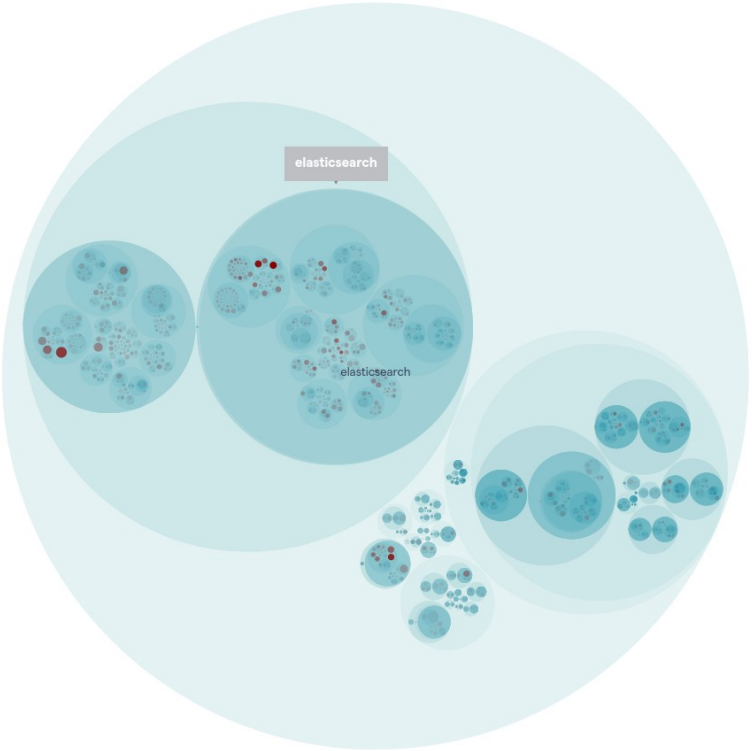
Hotspot to fragment kodu, który:

- jest najczęściej zmieniany (zwłaszcza przez wielu autorów)
- oraz ma najwyższą złożoność cyklometryczną.

Adam Thornhill: „Prioritizing Technical Debt as if Time and Money Matters” Goto; conference 2020

Narzędzia do lokalizacji hotspotów

- **codescene.io**
  - Płatny Kombajn do analizy repozytoriów



Narzędzia do lokalizacji hotspotów

- https://github.com/adamtornhill/code-maat

- Można odpalić prosto z .jar
- Analizuje commit log z repozytorium
- Wyznacza najczęściej edytowane pliki
- Generuje raport
- Licencja GNU

Odważna teza:

„Dług w stabilnym kodzie jest dopuszczalny, gdyż nie wpływa on bezpośrednio na podniesienie kosztu wprowadzania nowych funkcjolanlości w projekcie.”

Work **smart**, not hard.

Kod niezmieniany od roku, można spokojnie traktować jako kod stabilny. Dług zawarty w nim nie wpływu na koszt utrzymania projektu

Adam Tornhill „Software Design X-Rays - Fix Technical Debt with Behavioral Code Analysis”

Wygaszanie jedynie hotspotów,

realnie wpływa na obniżenie kosztu utrzymania projektu.

Refaktoryzować?!

Ja się **boje!**


DEVELOPER
REFACTORING
NEW FEATURE
imgflip.com

Boisz?

Czy nie chcesz?


Ale mam
dług techniczy
Uwaga
Refaktor
Kuuuuu
imgflip.com

Ale najpierw jedno zajebiście ważne pytanie:

Jak wygląda u Ciebie pokrycie testami?

To pytanie jest

Kluczowe.

Użyjmy Code coverage

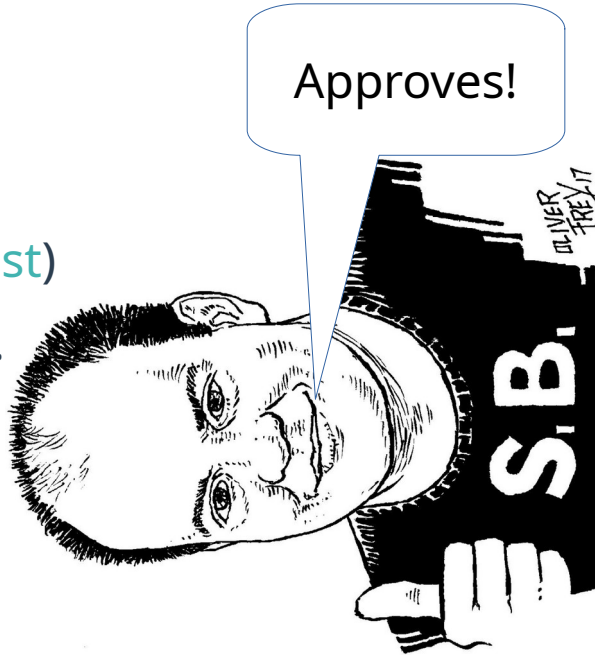
jako punktu odniesienia do podjęcia decyzji, którą strategię testową wybrać

- Sprawdzamy pokrycie testami całego systemu
- Sprawdzamy pokrycie funkcjonalności, którą będziemy refaktoryzować

- Na poziomie:
  - Unit Tests
  - Integration Tests
  - Functional Tests
  - End2End Tests
  - Contract Tests

Scenariusz 1. Nie ma testów!

- Opcja Minimum Effort Maximum Value:
  1) Wybieramy najgorętszy hotspot,
  2) Tworzymy Test Charakteryzujący (Characterization Test)
  3) Refaktoryzujemy wyłącznie pokryty testem fragment.
  4) Testy przechodzą?
  5) Commit & Push
  6) Następny hotspot



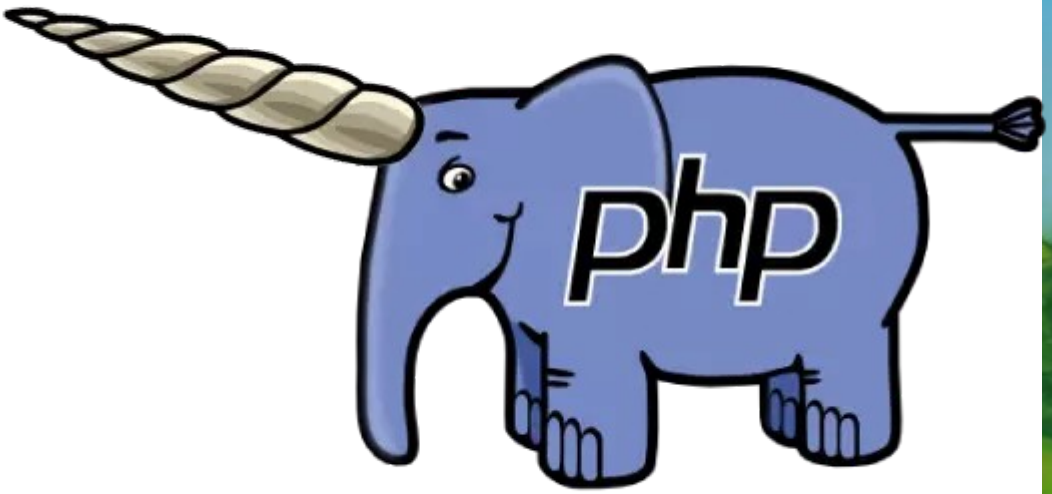
Scenariusz 2: Jakieś tam testy ale im nie ufam

- **Patrz scenariusz: Nie ma testów.**

Scenariusz 3: Są jakieś testy CC <50%

- Wybierz hotspot,
- Zrób analizę testów, czy pokrywają ścieżki krytyczne.
- Zepsuj coś celowo, zobacz czy testy są wiarygodne.
- Jak już masz jakieś tam zaufanie do testów.
- Napisz własny test do funkcjonalności, którą zamierzasz refaktoryzować. Jeżeli już są testy napisz je od nowa.
- Refaktoryzuj

Scenariusz 4: Ufam testom pokrycie >75%



Mogę już

refaktoryzować?

Nie! Stój!

Zapomniałeś o **benchmarku**.

Masz **test charakteryzujący**.

Odpal go z włączonym **profilerem**.

Benchmark

- Punkt odniesienia wydajności fragmentu kodu.
- Sprawdzenie czy nasz refactor nie zepsuje wydajności aplikacji.
- Pozwoli na wykrycie przedwczesnej optymalizacji.
- Gwarantuje lepsze zrozumienie kodu i celowości jego wcześniejszej optymalizacji.

Reasumując.

1.	2.	3.	4.	5.	6.
Test	**Benchmark**	**Refactor**	**Re-Test**	**Benchmark**	**Success!**

Czy to już?

Tak **Śmiało!**

Refaktoryzuj!

Jak utrzymać dług pod kontrolą?

Zmień proces Krzysiu.

Dodaj do procesu CI/CD

analizę statyczną

Dodaj do pocesu CI/CD

benchmark wydajności

Dodaj krok z analizą ich wyników do

Code Review

Cyklicznie analizuj **hotspoty**. Twórz **tickety**.

Priorytetyzuj.

Zaplanuj refactor.

Konsekwentnie go realizuj.

Refaktoryzuj tylko tam gdzie to

konieczne i **praktyczne**

Zakładaj **tickety** jak napotkasz miejsca w kodzie, które proszą się o **refactor** i **eskaluj** na planowaniu.

Dziel zadania

na max

jeden dzień

roboczy

Zrozum kod!

Testuj jednosktowo **zawsze**,

integracyjnie i e2e przy releasie

Jeżeli robisz **hotfix** na produkcji nocą.

Dodaj ticket by naprawić **przyczynę**.

Skup się przy **Code Review**, nie akceptuj

wszystkiego jak leci

Skup się na jednym. **Skończ**, a potem

wydaj

Nie ma negocjacji przy estymacji

czasowej („biznesy” chcą szybciej to muszą dostać mniej).

A co powiedzieć mojemu P.M / P.O?

Dodaj Epic w Jirze żeby śledzić postępy walki z długiem.

Zadania z tego Epic’a traktuj jak normalne zadania projektowe

(analiza, dekompozycja, estymacja)

Dodaj **timebox**

w cyklu by systematycznie walczyć z długiem.

Zmieniaj jego rozmiar w zależności od potrzeb i możliwości

Dług jest w **każdym** projekcie, **panuj** nad nim,

a **Cię nie ugryzie**.

A **twoi developerzy** nie będą myśleli,

że pracują przy

wywozie szamba.

Pytania?


VIA 9GAG.COM
T.HANKS
T.hanks a lot