24. Дизайн

??.01.2025

На днешната дата (миналата година - 11.01):

- През 532 в Константинопол започват бунтовете “Ника”
- Причината са... Ултраси...
- На “отбори” по надбягвания с колесници

Също така...

- Реално нищо от това нямаше значение, но вече имате история, с която да заговаряте мацки / пичове по дискотеките
- Ако ви трябват още любопитни факти за Византийската империя - из останките на Константинопол са открити надписи върху порутени стени, на които императрица Ирина проклинала боговете, че създали жената само с 3 дупки
- Ползвайте това знание отговорно
- Честит имен ден на Богдан, Богдана, Богомил, Досьо, Теодоси, Теодосии, Теодосия
- Най-вече на Досьо

Да си поговорим за дизайн

- DRY
- YAGNI
- KISS
- Premature Optimization
- SOLID
- Broken Window
- Boy scouting
- Fail Fast
- Pythonic Python
- Design Patterns
- Видове DP
- Design Anti-Patterns... Anti-Design Patterns?
- Testing

Момент!

- Защо трябва да ви пука?
- Крайната цел е чист код.
- Принципите, които ще обсъдим днес, сами по себе си няма еднолично да гарантират чист код, но са добър guideline.
- Имайте предвид, че тези парадигми са създадени във време, в което статичното типизиране е царувало - не всичко може да се чете буквално, когато става въпрос за Python.

А какво е чист код?

- Лесен за разбиране
  - Разбираем поток (flow) на приложението
  - Разбираеми взаимовръзки между различните обекти
  - Roles and responsibilities на различните класове
  - Разбира се какво прави всеки метод
  - Ясно е предназначението на всяка променлива или израз
- Лесен за промяна
  - Класовете и методите са кратки и еднозначни
  - Приложението има ясно клиентско API
  - Класовете и методите са предсказуеми и работят както се очаква
  - Кодът е лесно тестваем (и е изтестван)
  - Тестовете са разбираеми и лесни за промяна

Да започнем с принципите - DRY

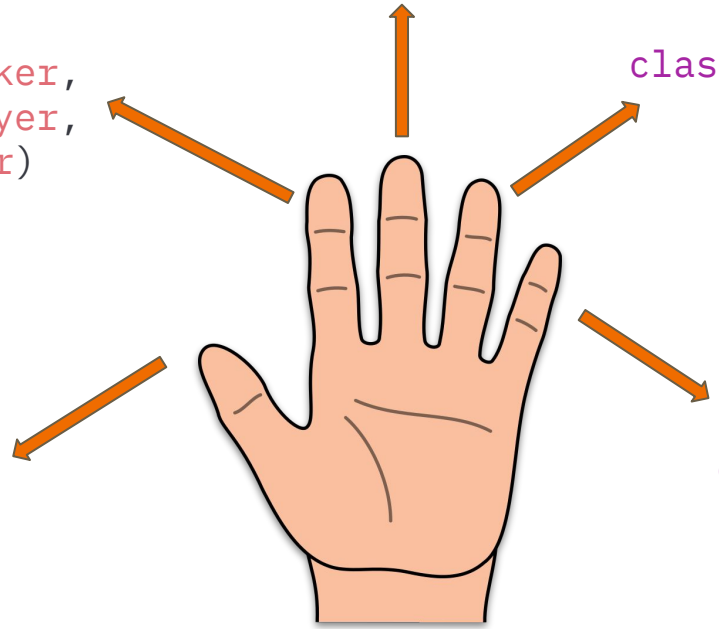
- DRY = Don’t Repeat Yourself
- Когато пишем нещо обемно, сложността се покачва. Хората не се справяме особено добре с високата сложност и многото взаимовръзки.
- Крайната цел е достигане до single responsibility парчета от системата.
- Идеята на DRY е тези парчета да съществуват само веднъж в цялата инфраструктура на проекта ни.
- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Помните ли това?

```
class MiddleFinger(DriverAssistant,
                   ForePlayer,
                   Scrather)
```

```
class IndexFinger(NosePicker,
                  ForePlayer,
                  Scrather)
```

```
class RingFinger(ForePlayer,
                 Scrather,
                 RingHolder)
```



```
class Thumb(RingHolder)
```

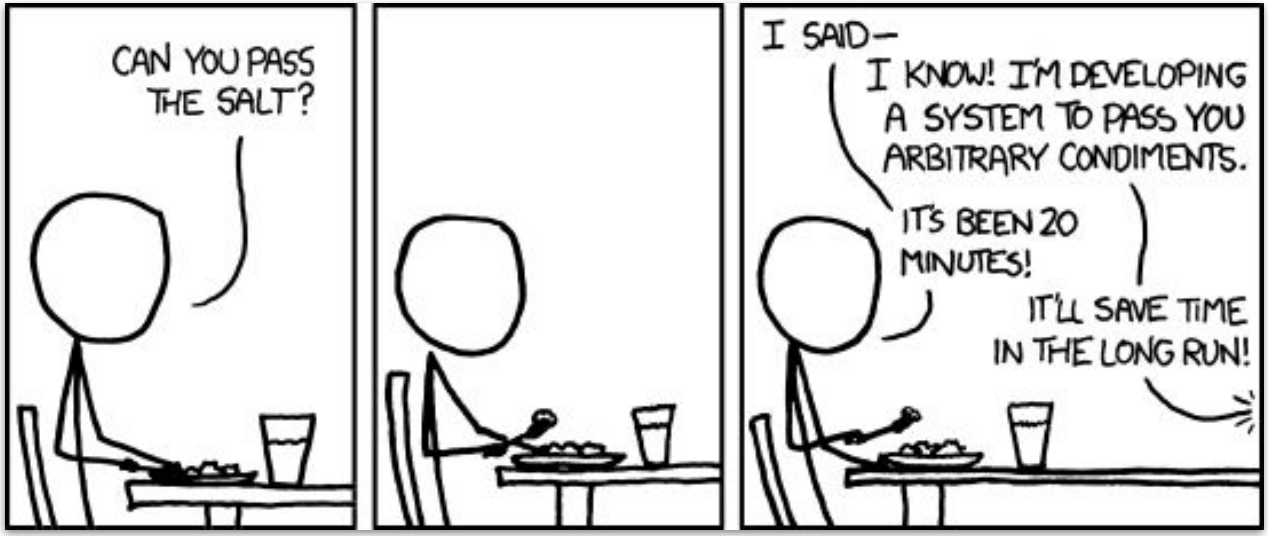
```
class Pinkie(NosePicker)
```

Как постигаме сухота?

- Визуална репрезентация
- Разбивайте на дребно по време на дизайна
- Възползвайте се от абстракцията!
- Динамично типизираните класове са перфектни за сух код

YAGNI

- На северозападен диалект - “ягне”
- На древен тракийски - “агни”
- Реално - **Y**ou **A**in’t **G**onna **N**eed **I**t

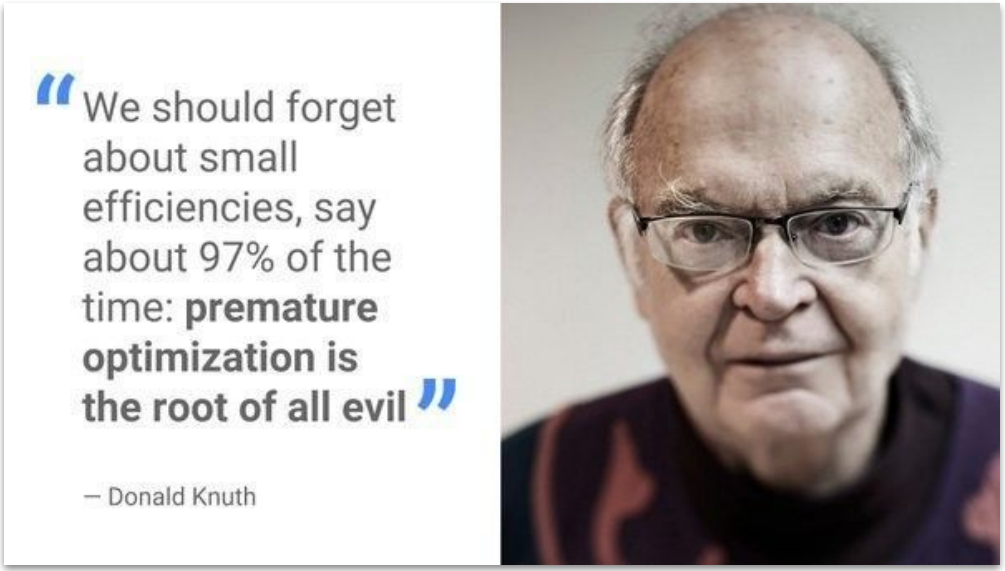


KISS

- Тук можем да сложим много мемета, но е прекалено low-hanging fruit.
- А и самият принцип не е много сложен - **K**eep **I**t **S**imple, **S**tupid.
- С други думи - стремим се към простота на кода и на дизайна.
- Нерядко нарушаването на KISS и нарушаването на YAGNI вървят ръка за ръка.
- “Code is like humor. When you *have* to explain it, it’s bad”

Premature Optimization

- Както в секса, така и в програмирането - premature is not okay.
- Нерядко получаваме въпроси *(не, не такива)* относно това дали Python не е бавен, дали това няма да е бавна операция и прочие.
- Вместо да вярвате на нас, послушайте Доналд Кнут:



SOLID

- Реално картинката няма връзка със SOLID принципите
- Но пък не намерих по-добро място от това
- SRP - **S**ingle **R**esponsibility **P**rinciple
- OCP - **O**pen / **C**losed **P**rinciple
- LSP - **L**isko(v) **S**ubstitution **P**rinciple
- ISP - **I**nterface **S**egregation **P**rinciple
- DIP - **D**ependency **I**nversion **P**rinciple

Single Responsibility

- Малко или много вече го обсъдихме, говрейки за DRY.
- Искаме всеки метод / клас да е едно парче функционалност *(погледнато като логически свързани единици, не търсим клас, който да има само един метод, например)*.
- Например, не искаме метод, който да прави следното:
  - Създава и изпълнява HTTPS заявка;
  - Записва полученият отговор в база данни;
  - Parse-ва полученият отговор, търсейки определени ключови думи;
  - Създава извадка за тежестта на ключовите думи;
  - Рисува диаграма на горната извадка;

Open / Closed

- “Software entities (classes, modules, functions, etc.) should be **open** for extension, but **closed** for modification.”
- Не искаме да пишем код, който ще претърпява големи промени при всяка малка промяна в изискванията.
- Най честата грешка е използване на конкретни класове / интерфейси, вместо абстракция. Това пречи на interface scalability.
- If / else?
- За щастие в Python имаме duck typing, което решава проблема на 75%.

Liskov Substitution

- “If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction.”
- Идеята на този принцип е, че наследниците на даден клас, трябва да могат да се използват на всички места, на които може да се използва родителят.
- Композицията помага да се реши този проблем.

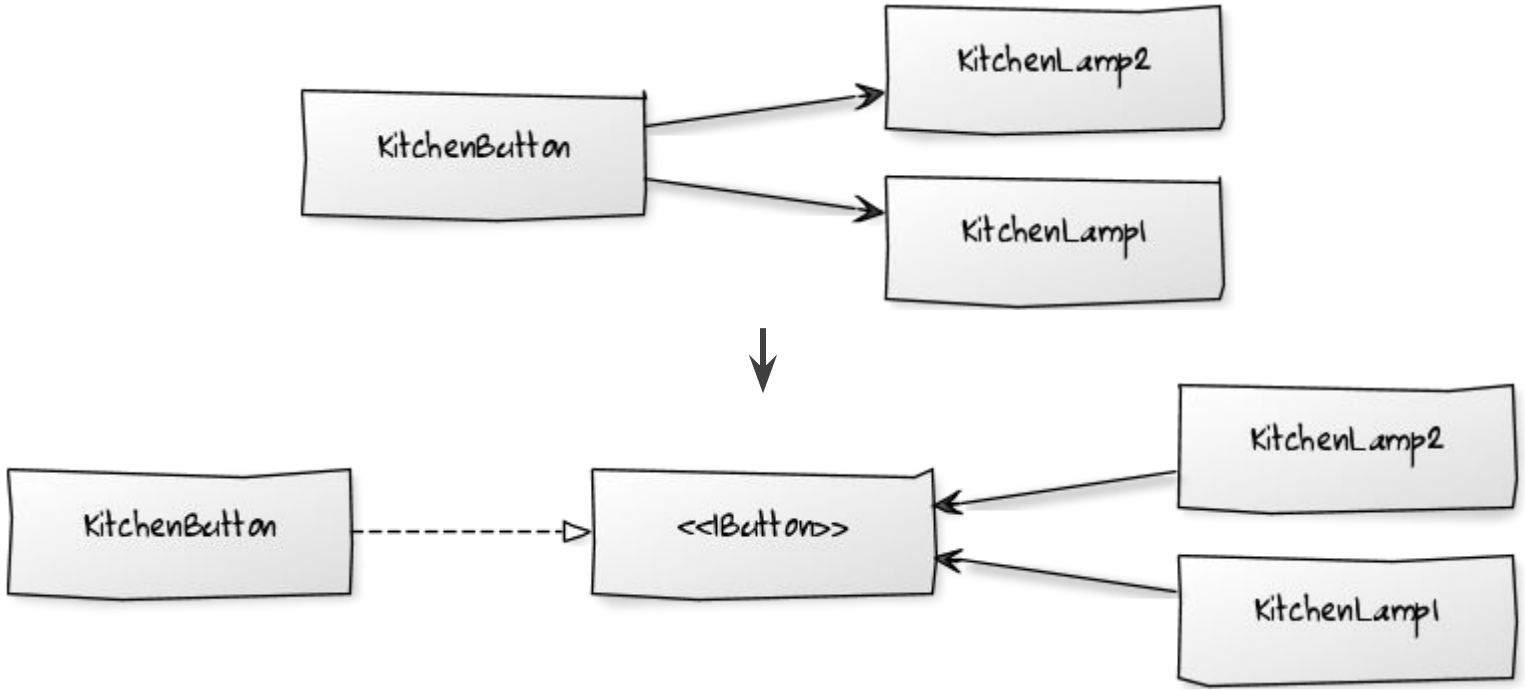
Interface Segregation

- ISP гласи, че наследниците не трябва да бъдат “карани” да имплементират интерфейси, които няма да използват.
- Пример?
- Потенциално решение?
- Mixins.

Dependency Inversion

- Това е един от по-важните принципи, тъй като той е крайно облагодетелстван от Python.
- Идеята му е, че не трябва да създаваме зависимости към конкретни обекти, а да работим с абстракция.
- Или както формално се обяснява принципът:
  - A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - B. Abstractions should not depend upon details. Details should depend upon abstractions.

Ако преподавахме друг език:



Разликата между OCP и DIP?

- Да, двете доста си приличат.
- Duck typing отново играе важна роля.
- Обикновено нарушеният DIP води до нарушен OCP.
- Разликата е, че **Dependency Inversion** принципът е в контекста на взаимовръзки, а **Open / Closed** се фокусира върху отделните части (модул, клас, интерфейс).

Оф, много се обърках

- Пишейки Python не е нужно да мислите за всичко това.
- И ние не го правим.
- Но ако трябва да го сведем до няколко практически съвета:
  - Пишете логически свързани / дефинирани парчета код (SRP)
  - Пишете EAFP код, а не LBYL код (OCP)
  - Гледайте да правите разлика между необходимостта от наследяване и композиция (LSP и ISP)
  - Разчитайте на абстракция, а не на проверки (DIP)

Теорията за счупеният прозорец

- През 60-те, 70-те и 80-те след наблюдения на психолози и полицейски управления се заражда **broken window** теорията.
- **1 счупен прозорец + време = много счупени прозорци** и др.
- Същото важи и за кода ви!
- “All the rest of this code is crap, I’ll just follow suit...”
- С други думи, оставяйки лош дизайн, кофти парчета код и прочие, полагате основа за още хаос и замърсяване.
- Обаче важи и обратното!

Boy Scout Rule

- Бойскаутите имат правилото, че трябва да оставят къмпинга по-чист отколкото са го заварили.
- И... Познайте, същото важи и за кода!
- Не е необходимо да хванете едно парче стар код и да го излижете от до - винаги можем да бъдем бойскаути на дребно!
- Това не е загубено време.

Boy Scout Rule

Fail Fast

- Когато пишем софтуер, той винаги има проблеми.
- За решаването на тези проблеми имаме 2 подхода:
  - Вариант 1 - Пишем системата, така че да се опитва да заобиколи или поправи всеки възникнал проблем.
  - Вариант 2 - **Fail Fast**.
- Идеята е да позволим на грешките в кода да бъдат видими колкото се може по-скоро.
- Има разлика между това да оправим проблем и това да го скрием.
- Пример - ако клиент използва публичен интерфейс с неподходящ набор от данни, не е добра идея да се опитаме да ги “оправим”. Това води до риск да създадем труден за дебъгване проблем по-нататък.

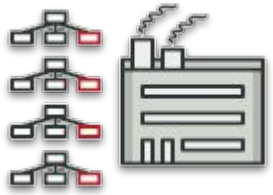
Нека поговорим малко за design patterns

- **Design patterns are all about communication!**
- Сериозно, design pattern-ите не са нещо “открито” през 21 век, в един момент просто се дефинира конкретна номенклатура.
- Рядко ще стоите и ще се чудите:
  “Хм, кой ли дизайн патърн да използвам днес...?”
- Терминът **Factory** придава смисъл, който в противен случай би бил обяснен в детайл.
- Освен това, клас, който се казва `AnimalFactory` очевидно имплементира съответният pattern.
- Друг бонус - design patterns помагат за добра структура.

Видове DP (*wink, wink*)

- Преди това, **disclaimer**:
  - Design patterns ≠ cargo cult
  - Не всичко е пирон
- **Creational** (Factory, Singleton):
  - Механизми за създаване на различни обекти;
  - Не са критични за Python;
- **Structural** (Decorator, Facade):
  - Спомагат за изграждането на прости и ефикасни йерархии и взаимовръзки;
- **Behavioural** (Iterator, Observer):
  - Целят ефикасно и сигурно дефиниране на поведение и комуникация между обектите;

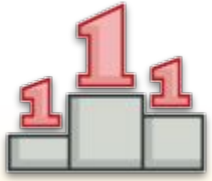
Factory (или Abstract Factory)



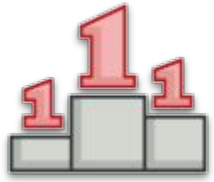
- Цели създаването на обекти от едно семейство, без да се интересува от конкретният клас.
- Познахте, **невероятно** лесно е да се направи нещо такова в Python:

```
class A:
    pass
class B:
    pass
...

LETTERS = [A, B, ...]

def letter_factory(number_of_letters):
    return [LETTERS[letter]() for letter in range(number_of_letters)]
```

Singleton



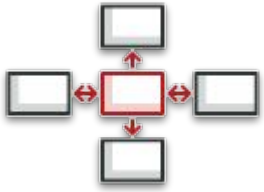
Singleton



- Това е клас, от който може да съществува само една инстанция. Опит за създаване на нова инстанция, връща вече съществуващата **единствена** инстанция.
- Има два лагера, единият от които страстно мрази **Singleton** шаблона.
- Все пак - може да е полезен за класове, които пазят глобален state (безкрайно лесно е да се подведете, че е добра идея, а да не ви е необходимо), за драйвери (ако не искате множество инстанции да карат хардуера да прави различни неща едновременно).

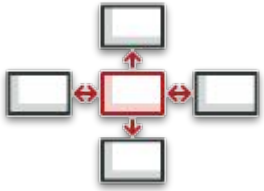
```
class TOAA:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance
```

Pool

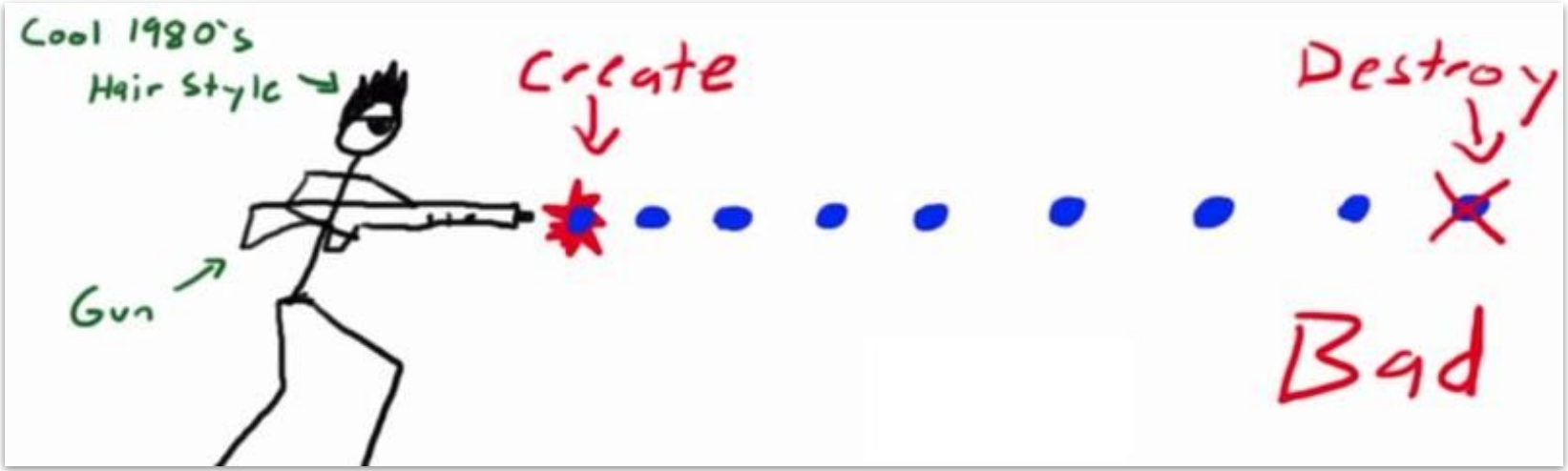


- Представлява басейн от обекти, които да се преизползват при нужда.
- Изключително полезен в случай на **обемни** или **трудни за инстанциране** обекти.
- Какво се случва, когато трябва администраторите да сетъпнат работно място за нов колега?
  - Ако има налични компютри/монитори ползват тях.
  - Ако няма - купуват нови.
- Все пак сравнително полезен, дори и обектите да не са “скъпи”.

Pool

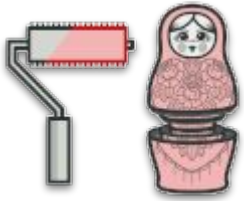


- Трябва да имате доста интензивна откъм обекти игра, за да ви се наложи, но ето ви един пример, който може да ви изкара от някой bottleneck.



- Вместо поведението на картинката, можем да запазваме вече инстанцираните обекти и да ги преизползваме.
- *“Малките” числа в Python са имплементирани по сходен начин.*

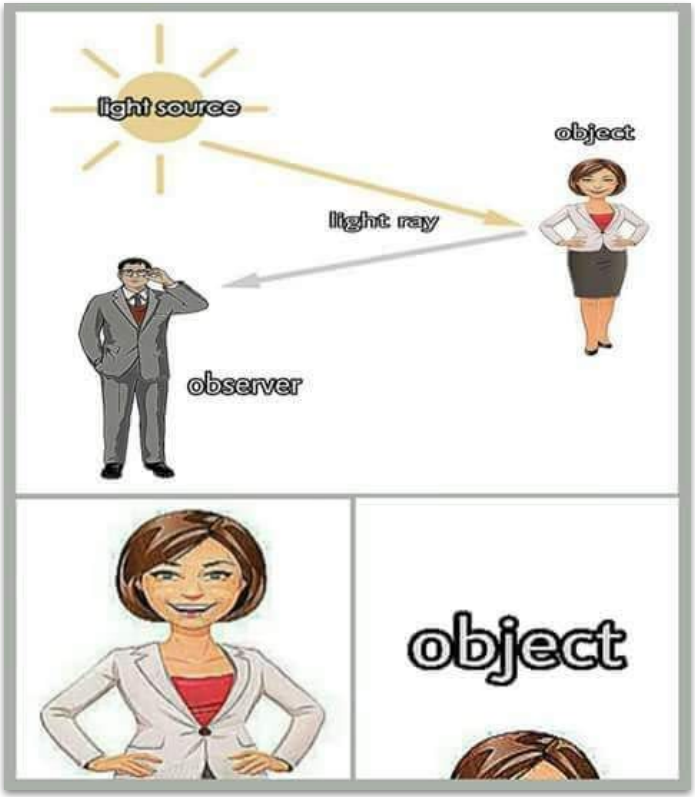
Decorator



- Декораторите динамично променят поведението на един метод, без да променят интерфейсът, чрез който той се извиква.
- Променят ще рече - **добавят** допълнителна функционалност.
- Това помага за преизползване на код, който иначе би присъствал в много на брой методи/функции.
- В Python вече имаме този патърн имплементиран.
- Enter the `@декоратор`.

Observer

- Създава един-към-много връзка между дадени обекти.
- При този pattern се дефинират два типа обекти:
  - Subject / Publisher
  - Observer / Subscriber
- Когато настъпи промяна в **subject**-а, той известява всичките си **observer**-и.
- Ще ви дам пример с най-добрата картинка по темата, която намерих в нета.
- Ние, **не се** присъединяваме към това твърдение.
- Still funny.



Защо?

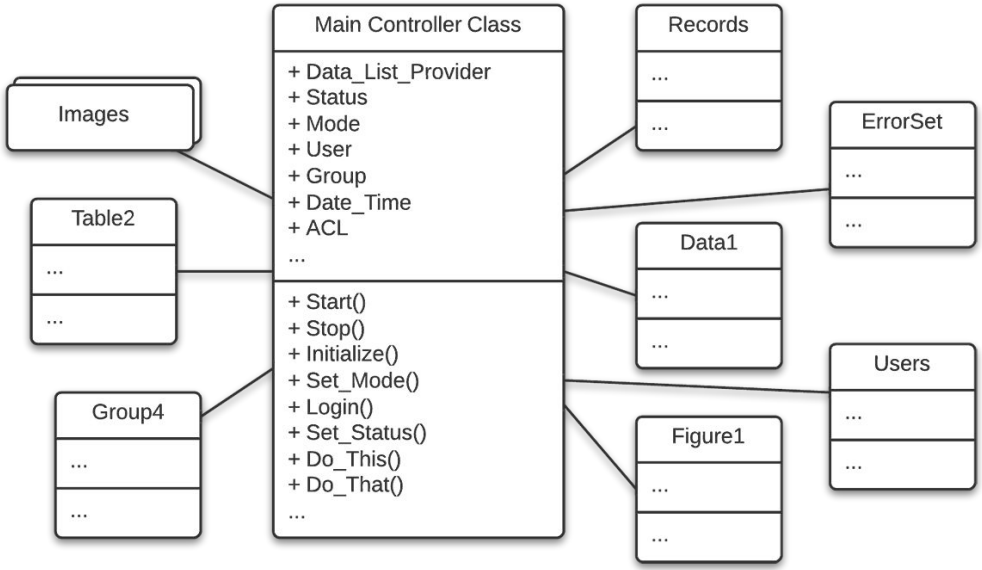
- Добре де, но защо изобщо ги намесваме и защо са само някои?
- Намесваме ги, защото имате да пишете проекти, а проектите ви е хубаво да имат добър дизайн. Да, в момента, в който довършвам лекцията имате малко по-малко от 2 седмици да ги довършите, но пък когато ви обещавах, че ще я допълня и ще я кача - имаше над месец. *Упс.*
- А защо само някои - просто да ви вкараме малко във филма, а и за да ви покажем, че Python е достатъчно гъвкав за да направи голям процент от широко-използваните design pattern-и излишни...
- Пример - `@декоратор`
- Друг пример - замислете се колко лесна беше имплементацията на **Factory**.

Антипатърни в дизайна

- И-и-и така и така сме на темата, нека поговорим малко и за anti-patterns.
- Анти...шаблоните? в дизайна не са очевидни на пръв поглед.
- За сметка на това последствията са неприятни.
- Не стават видими в самото начало, но с нарастването на проекта, започват да вредят все повече.
- Да си поговорим за някои по-важни.

The Blob

- Клас, който съдържа огромно количество функционалност.
- Обикновено върви ръка за ръка с много по-малки “data” класове.
- Много често аргументацията е, че класът е *“controller”*.



The Blob

Симптоми:

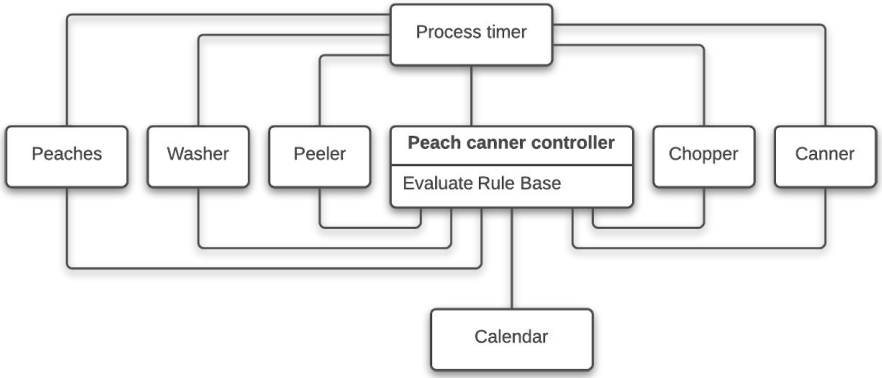
- Клас с много атрибути и методи (представете си цифра над 60)
- Наличие на несвързани данни и функционалност
- Контролер клас + много малки дата класове
- Отсъствие на обектно ориентиран дизайн

Проблеми:

- Нарушава **SRP**
- Пречи да се насладим на ползите от ОО дизайна
- Прави модификациите трудни
- Кодът става сложен за преизползване
- И сложен за тестване

Poltergeist

- Вариация на “**The Blob**”.
- Класове, които имат кратък живот и ограничената функционалност да извикат чужда функционалност.
- Резултатът е класове, които нямат никакъв state и пазят връзки към всички останали класове.

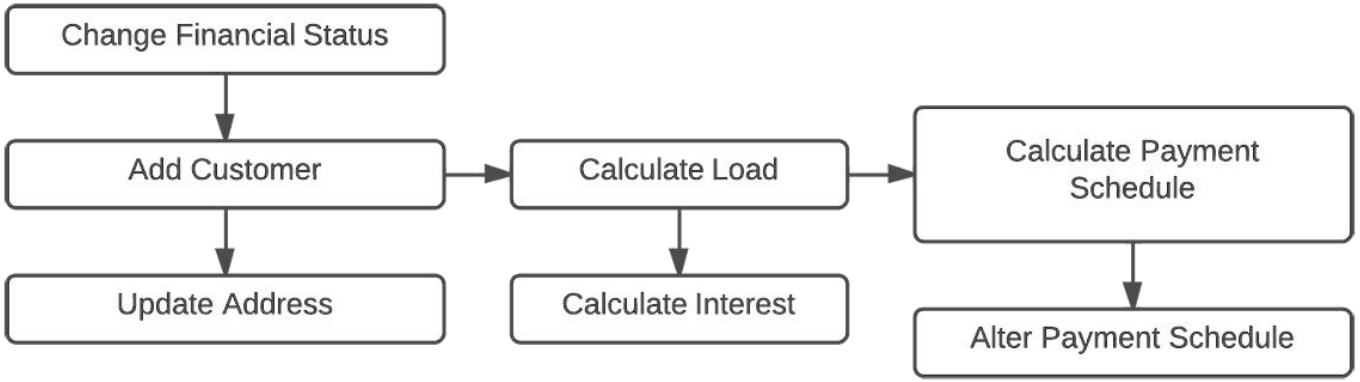


The Swiss Army Knife

- Може да бъде вариация на “**The Blob**”, но не задължително.
- Причината е, че обикновено има 1 blob, но може да има много ножове.
- **Причината** е стремеж в един клас да се вкара колкото се може повече функционалност.
- **Последствията** са сходни.

Functional Decomposition

- Алтернативно име:
  Писане на FORTRAN код в Python
- Характеризира се с използването функционално-ориентиран код, маскиран като ООП.



Functional Decomposition

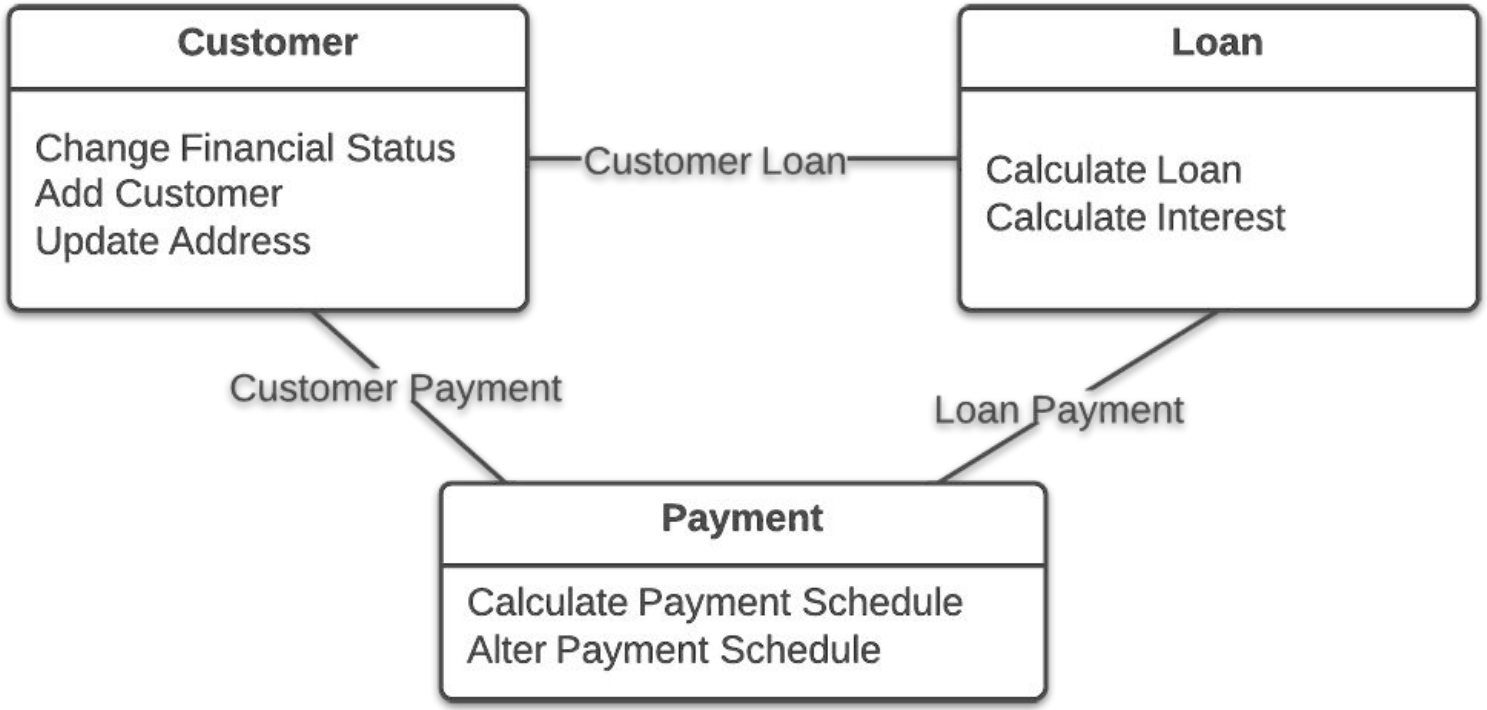
Симптоми:

- Класове с имена от типа на:
  - `CalculateInterest`
  - `DisplayTable`
- Класовете имат атрибути, които се използват само за вътрешни цели
- Класовете изпълняват само 1 действие

Проблеми:

- Никаква употреба на ООП
- Трудност да се опише как точно работи системата
- Кодът е труден за преизползване
- За тестване - зависи, но е лесно постижимо
- Налага използването на едни и същи данни като аргументи и return стойности

Functional Decomposition Fix



Изключение

Важна забележка:

Не винаги е нужно да използваме ООП подход за решаването на даден проблем. Разликата е, че тогава е осъзнато и не използваме артефактите на ООП самоцелно!

Dead Code

- “Абе нямам идея за какво е тоя метод, написан е преди аз да дойда във фирмата..."
- Това са парчета код, които нямат *(или имат, но на 10% от съдържанието си)* употреба в приложението.
- **Възможни причини:**
  - R&D / prototype code
  - Лош контрол на дистрибуцията на версиите на кода
  - Lone wolf code
  - Недобро менажиране на промените
  - **Синдромът “Абе, то някой ден може да е полезно..."**

Мъртъв Код

Симптоми:

- Странни променливи и функции/фрагменти из кода, които не изглежда да бъдат употребявани
- Недокументирани парчета код, които нямат очевидна връзка с цялостната архитектура
- Огромни секции с закоментиран код, без обяснение защо

Проблеми:

- Замърсява кода, затруднявайки четимостта и разбираемостта му
- Прави **архитектурата** (не само кода) трудно разбираема
- Експоненциален растеж при големи проекти

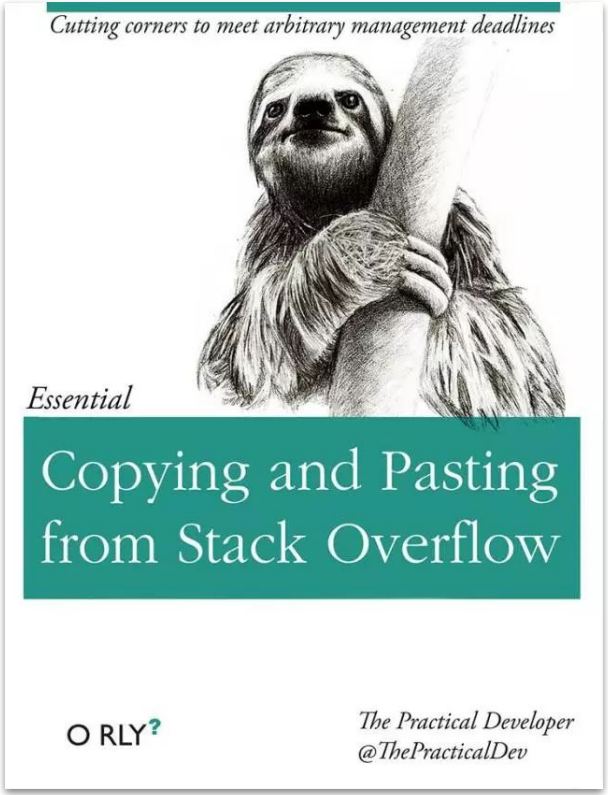
Reinventing the Wheel (или топлата вода)

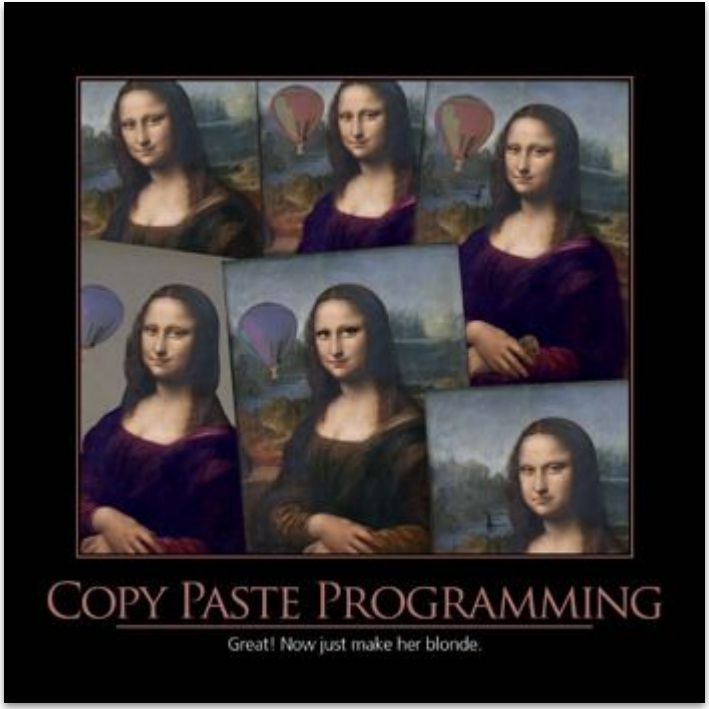
- Стремежа към **самоцелно** “откриване” на нови подходи води до забавяне на цялостният процес.
- Преизползването не спира само до кода, който сте написали, всъщност това е по-трудната за преизползване част.
- По-важното е преизползване на концепции (**дизайн**, **архитектура**) и технологии (от прости **модули** до **готови решения**).
- С други думи, това да си напишете сами парсър за xml е готино упражнение, но не и добра практика за професионален проект.

Златният Чук

- От друга страна, не е добра идея да имате единствено готово решение за **всеки** проблем.
- Това може да доведе до неоптимални решения.
- Последствията могат да бъдат лоши performance, scalability, etc.
- **Къде е балансът?**
  Запознавайте се с нови подходи, технологии, парадигми. Бъдете критични към тях в даден контекст, употребата на дадено нещо, трябва да бъде аргументирана.

Copy-paste Programming





Антипатърни в питонския код

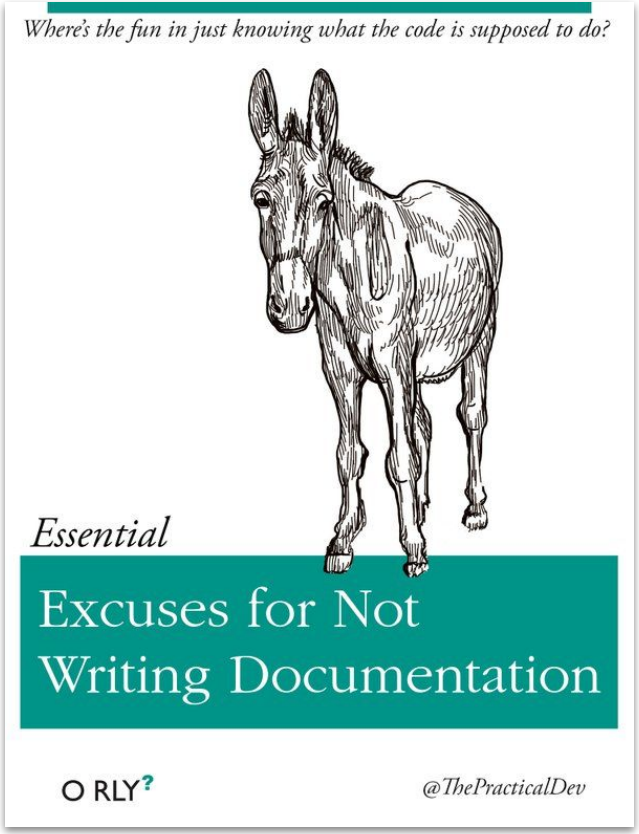
- Говорили сме много по темата, но пак ще ви припомним някои от нещата...
- Непитонски `for`
- Различен тип на връщаните стойности на една и съща функция
- Проверка вместо употреба на `dict.get()`
- Неизползване на unpacking функционалността
- Работа с файлове без `with`
- Конкатенация на стрингове
- Java-style гетъри и сетъри
- Скриване на built-ins
- Кофти ред на прихващане на грешките
- Проблеми с mutable аргументи
- Грешно сравнение (`==`, `is`, `<`, `>`, etc.)

Още!

- Сигурно изпускаме немалко... Разгледайте си обратната връзка на домашните, за да се подсигурите, че вие няма да изпуснете нещо из проектите си.
- Останалите неща са стилистични:
  - Индентация
  - Именоване
  - Импорти
  - Документация

И да ви напомним защо пишем документация

- Коментарите са за програмистите (какво прави този код, как го прави)
- Докстринговете са за потребителите (какво прави този интерфейс, как да го използваме)
- А защо изобщо да документираме?
  - Някой друг ще чете кода ви
  - Някой друг ще използва кода ви
  - Вие самите ще четете/използвате кода си след 6 месеца
  - Автоматично-генерирани уикита/документи
- Все пак, не прекалявайте, документацията не е заместител на четимият код (камо ли на добрия дизайн)...



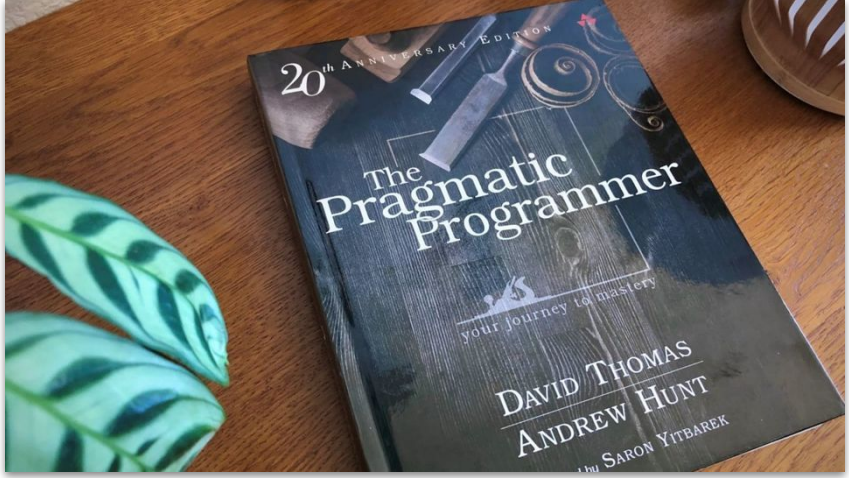
Коментарите са важни

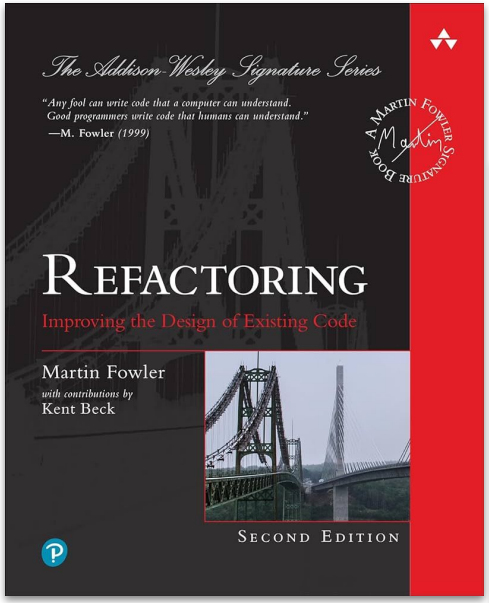
```
// I dedicate all this code, all my work, to my wife, Darlene, who will
// have to support me and our three children and the dog once it gets
// released into the public.


// somedev1 -  6/7/02 Adding temporary tracking of Login screen
// somedev2 -  5/22/07 Temporary my ass


// drunk, fix later


// Magic. Do not touch.


// I'm sorry.
```

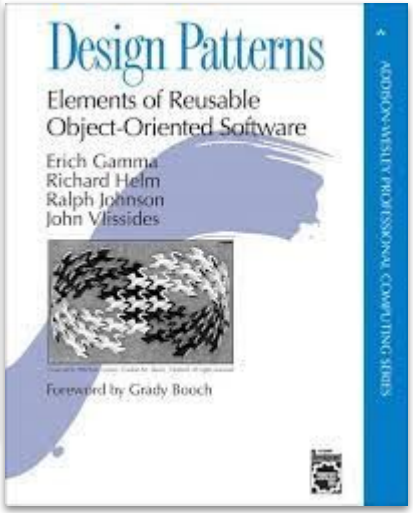
Коментарите са важни

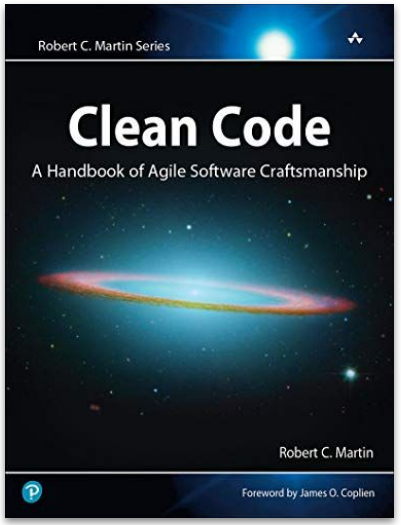
```
return 1; # returns 1


////////////////////////////////////// this is a well commented line


// I don't know why I need this, but it stops the people being upside-down
x = -x;


// I am not responsible of this code.
// They made me write it, against my will.


options.BatchSize = 300; //Madness? THIS IS SPARTA!
```

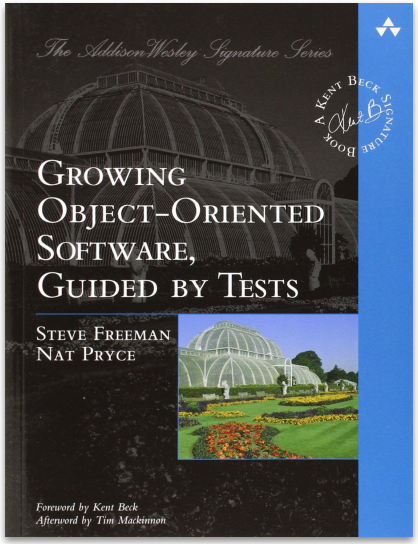
Четива по темата











Въпроси?