Object Calisthenics

9 steps to better OO code

About me

Paweł Lewtak

Senior Developer at Xstream

@pawel_lewtak

Agenda

Learn how to make our code more:

- readable
- reusable
- testable
- maintainable

Things worth knowing

- DRY
- KISS
- SOLID
- YAGNI
- GRASP

Calisthenics

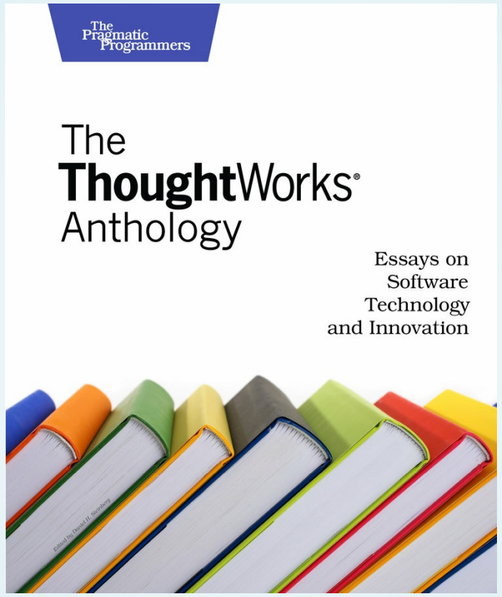
Cal • is • then • ics

/ˌkaləsˈTHeniks/

"Calisthenics are exercises consisting of a variety of gross motor movements; often rhythmical and generally without equipment or apparatus."

Wikipedia

Object Calisthenics



Jeff Bay

Written for Java

Why bother?

Code is read more than it is written

Author unknown

You need to write code that minimises the time it would take someone else to understand it - even if that someone else is you

Art of Readable Code by Dustin Boswell, Trevor Foucher

Rule #1

Only one level of indentation per method

```
class Board(object):
    def __init__(self, data):
        # Level 0
        self.buf = ""
        for i in range(10):
            # Level 1
            for j in range(10):
                # Level 2
                self.buf += data[i][j]
```

```
class Board(object):
    def __init__(self, data):
        self.buf = ""
        self.collect_rows(data)

    def collect_rows(self, data):
        for i in range(10):
            self.collect_row(data[i])

    def collect_row(self, row):
        for j in range(10):
            self.buf += row[j]
```

```
class UserService(object):
    def register(self, username, email, promo_code = False):
        user = self.create_user(username)
        if email:
            send_email(user, email)
            if promo_code:
                send_promo_code(user, promo_code)
```

```
products = self.get_products_by_user(...)
if products is None:
    products = self.get_products_by_media(...)
    if products is None:
        products = self.get_products_by_domain(...)
        if products is None:
            products = self.get_any_products(...):
            if products is None:
                raise Exception('Access denied')
            else:
                ...
        else:
            ...
    else:
        ...
else:
    ...
```

```
products = self.get_products_by_user(...)
if products is None:
    products = self.get_products_by_media(...)
if products is None:
    products = self.get_products_by_domain(...)
if products is None:
    products = self.get_any_products(...):
if products is None:
    raise Exception('Access denied')

# else...
```

Chain of command

```python
class Command(object):
    next_command = None

    def add(self, next_command):
        if self.next_command is None:
            self.next_command = next_command
        else:
            self.next_command.add(next_command)

    def _process(self, *args, **kwargs):
        """ computations """
        pass

    def process(self, *args, **kwargs):
        result = self._process(*args, **kwargs)
        if result is None:
            if self.next_command is not None:
                return self.next_command.process(*args, **kwargs)
        else:
            return result

        return None
```

```
class GetProductsForUser(Command): pass
class GetProductsByMedia(Command): pass
class GetProductsByDomain(Command): pass
class GetAnyProducts(Command): pass

commands = GetProductsForUser()
commands.add(GetProductsByMedia)
commands.add(GetProductsByDomain)
commands.add(GetAnyProducts)

products = commands.process(...)
```

Benefits

- Single responsibility
- Better naming
- Shorter methods
- Reusable methods

Rule #2

Do not use *else* keyword

```
def login (self, request):
    if request.user.is_authenticated():
        return redirect("homepage")
    else:
        messages.add_message(request,
                             messages.INFO,
                             'Bad credentials')
        return redirect("login")
```

```
def login (self, request):
    if request.user.is_authenticated():
        return redirect("homepage")

    messages.add_message(request,
                         messages.INFO,
                         'Bad credentials')
    return redirect("login")
```

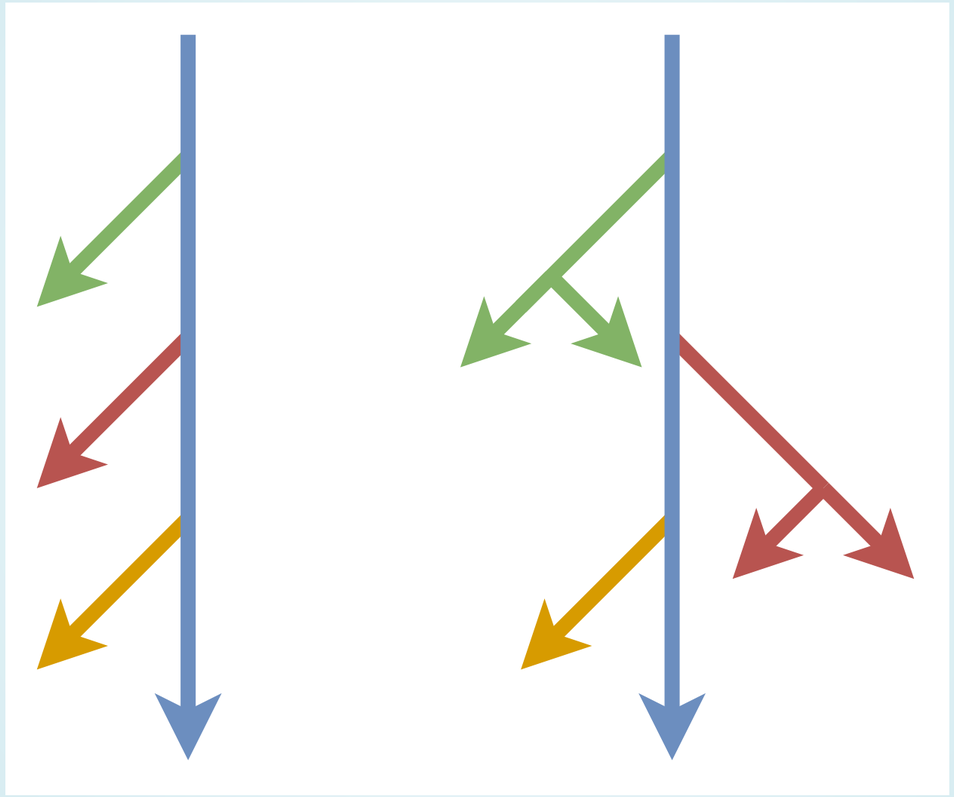
```
def function(param):
    if param is not None:
        value = param
    else:
        value = "default"

    return value
```

```
def function(param):
    value = "default"
    if param is not None:
        value = param

    return value
```

```
def function(var_a, var_b, var_c, var_d):
    if var_a:
        if var_b:
            # some code
        else:
            # some code
    elif var_b and var_c:
        if not var_d:
            # some code
        else:
            # some code
    elif var_b and not var_c:
            # some code
    else:
            # some code
```



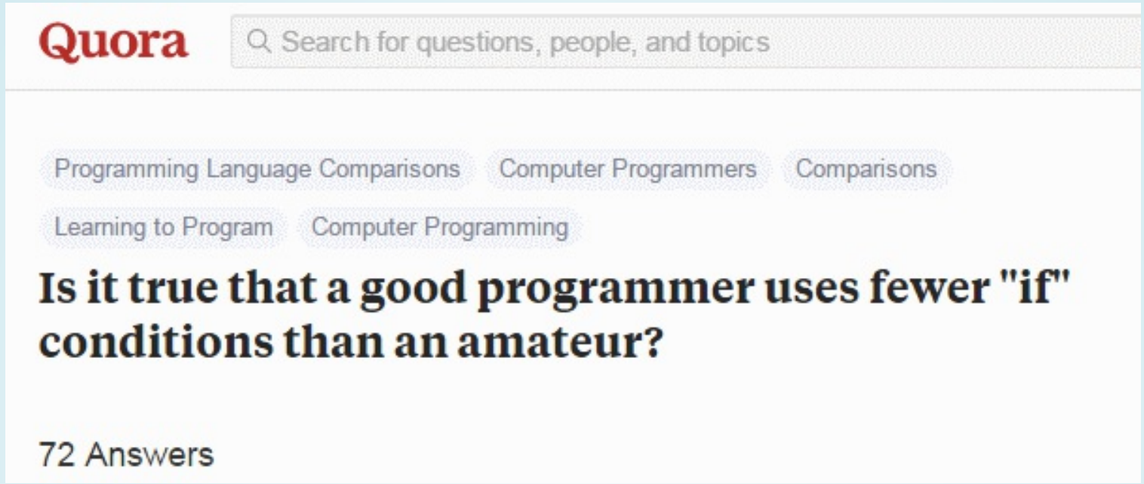
Extract code

Default value

Polymorphism

Strategy pattern

State pattern



https://www.quora.com/Is-it-true-that-a-good-programmer-uses-fewer-if-conditions-than-an-amateur

Benefits

- Avoids code duplication
- Lower complexity
- Readability

Rule #3

Wrap primitive types if it has behaviour

Value Object in DDD

```python
class Validator(object):
    def check_date(self, year, month, day):
        pass

# 10th of December or 12th of October?
validator = Validator()
validator.check_date(2016, 10, 12)
```

```
class Validator(object):
    def check_date(self, year: Year, month: Month, day: Day) -> bool:
        pass

# Function call leaves no doubt.
validator.check_date(Year(2016), Month(10), Day(12))
```

```
def calculate_distance(source_x, source_y, target_x, target_y):
    pass

calculate_distance(1, 2, 3, 4)
```

```python
from collections import namedtuple

class Point2D(namedtuple("Point2D", "x y")):
    pass

def calculate_distance(source_point, target_point):
    pass

calculate_distance(Point2D(1, 2), Point2D(3, 4))
```

Benefits

- Encapsulation
- Type hinting
- Attracts similar behaviour

Rule #4

Only one dot per line

OK: Fluent interface

```
class Poem(object):
    def __init__(self, content):
        self.content = content

    def indent(self, spaces):
        self.content = " " * spaces + self.content

        return self

    def suffix(self, content):
        self.content = self.content + " - " + content

        return self

Poem("Road Not Travelled").indent(4)\
    .suffix("Robert Frost")\
    .content
```

Not OK: getter chain

```
class CartService(object):
    def get_token(self):
        token = self.get_service('auth')\
            .auth_user('user', 'password')\
            .get_result()\
            .get_token()

        return token

# 1. What if None is returned instead of object?
# 2. How about exceptions handling?
```

```
class Field(object):
    def __init__(self):
        self.current = Piece()

class Piece(object):
    def __init__(self):
        self.representation = " "

class Board(object):
    def board_representation(self, board):
        buf = ''
        for field in board:
            buf += field.current.representation

        return buf
```

```
class Field(object):
    def __init__(self):
        self.current = Piece()

    def add_to(self, buffer):
        return self.current.add_to(buffer)

class Piece(object):
    def __init__(self):
        self.representation = " "

    def add_to(self, buffer):
        return buffer + self.representation

class Board(object):
    def board_representation(self, board):
        buf = ''
        for field in board:
            buf = field.add_to(buf)

        return buf
```

Benefits

- Encapsulation
- Demeter's law
- Open/Closed Principle

Rule #5

Do not abbreviate

Why abbreviate?

Too many responsibilities

Name too long

```python
def register_user_send_welcome_email_and_add_to_default_groups():
    pass


# vs


def handle_user_registration():
    user = create_user()
    send_welcome_email(user)
    add_to_default_groups()
```

Avoid confusion

```
acc = 0
// accumulator? accuracy?

pos = 100
// position? point of sale? positive?

auth = None
// authentication? authorization? both?
```

Duplicated code

```python
class Order(object):
    def ship_order(self):
        pass

order = Order()
order.ship_order()


// vs

class Order(object):
    def ship(self):
        pass

order = Order()
order.ship()
```

Split & extract

Refactor!

Think about proper naming

Benefits

- Clear intentions
- Indicate underlying problems

Rule #6

Keep your classes small

What is small class?

- 15-20 lines per method
- 50 lines per class
- 10 classes per module

Benefits

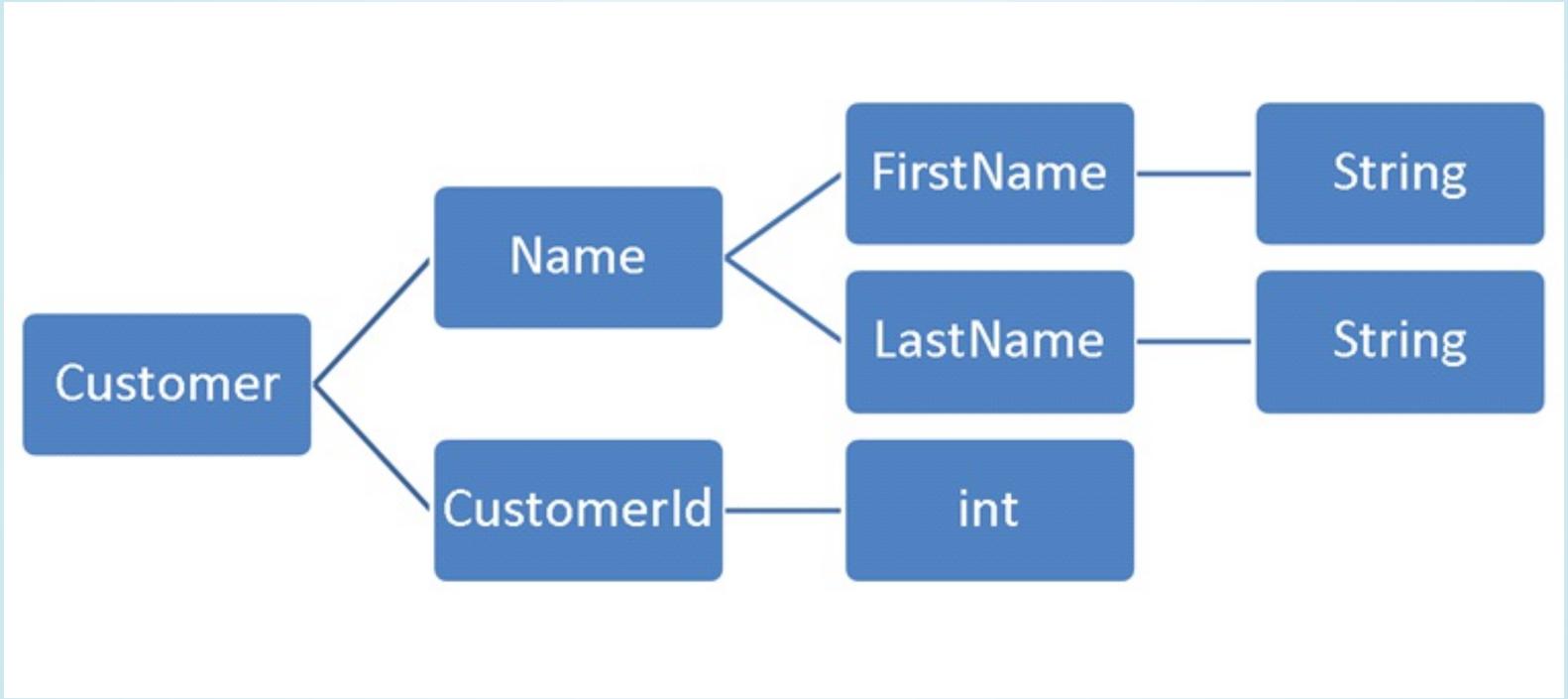
- Single Responsibility
- Smaller modules
- Coherent code

Rule #7

No more than 2 instance variable per class

Class should handle single variable state

In some cases it might be two variables


Customer
Name
CustomerId
FirstName
LastName
String
String
int

```
class CartService(object):
    def __init__(self):
        self.logger = Logger()
        self.cart = CartCollection()
        self.translationService = TranslationService()
        self.auth_service = AuthService()
        self.user_service = UserService()
```

```
class CartService(object):
    def __init__(self):
        self.logger = Logger()
        self.cart = CartCollection()
```

Benefits

- High cohesion
- Encapsulation
- Fewer dependencies

Rule #8

First class collections

Python's *collections* module

Doctrine's *ArrayCollection*

Benefits

- Single Responsibility

Rule #9

Do not use setters/getters

Accessors are fine

Don't make decisions outside of class

Let class do it's job

Tell, don't ask

```
class Game(object):
    def __init__(self):
        self.score = 0

    def set_score(self, score):
        self.score = score

    def get_score(self):
        return self.score

# Usage
ENEMY_DESTROYED_SCORE = 10
game = Game()
game.set_score(game.get_score() + ENEMY_DESTROYED_SCORE)
```

```
class Game(object):
    def __init__(self):
        self.score = 0

    def add_score(self, score):
        self.score += score

# Usage
ENEMY_DESTROYED_SCORE = 10
game = Game()
game.add_score(ENEMY_DESTROYED_SCORE)
```

Benefits

- Open/Closed Principle

Recap

1. Only one level of indentation per method,
2. Do not use else keyword,
3. Wrap primitive types if it has behavior,
4. Only one dot per line,
5. Don’t abbreviate,
6. Keep your entities small,
7. No more than two instance variable per class,
8. First Class Collections,
9. Do not use accessors

Homework

Create new project up to 1000 lines long

Apply presented rules as strictly as possible

Draw your own conculsions

Customize these rules

Make them your own

Final thoughts

These are not best practices

These are just guidelines

Use with care!

Questions?

Links

https://www.cs.helsinki.fi/u/luontola/tdd-2009/ext/ObjectCalisthenics.pdf

https://pragprog.com/book/twa/thoughtworks-anthology

https://en.wikipedia.org/wiki/Law_of_Demeter

https://www.quora.com/Is-it-true-that-a-good-programmer-uses-fewer-if-conditions-than-an-amateur

Thank you!

 @pawel_lewtak