

# Czysty kod, część II

## Zaawansowane metody programowania

---

mgr inż. Krzysztof Rewak

24 marca 2019

Wydział Nauk Technicznych i Ekonomicznych

Państwowa Wyższa Szkoła Zawodowa im. Witelona w Legnicy

# Plan prezentacji

1. Funkcje
2. Formatowanie kodu
3. Podsumowanie

*Dlaczego dobry kod tak szybko się psuje?*

*Wyobraźmy sobie następującą sytuację: przed operacją pacjent zdecydowanie żąda, aby lekarz przestał wreszcie myć ręce, ponieważ zabiera to zbyt wiele czasu*

*Wyobraźmy sobie następującą sytuację: przed operacją pacjent zdecydowanie żąda, aby lekarz przestał wreszcie myć ręce, ponieważ zabiera to zbyt wiele czasu.*

*Skauci amerykańscy mieli prostą zasadę, jaką można zastosować w naszym zawodzie:*

*Pozostaw obóz czystszy, niż go zastałeś.*

# Funkcje

---

Funkcja czy metoda to podstawowa jednostka operacyjna w naszych programach.



Zatem warto przede wszystkim zachować porządek w samej budowie funkcji.

Niektóre języki są dosyć liberalne pod względem definiowania argumentów czy zwracanych typów, ale nie powinniśmy nadużywać tej możliwości.



```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class Friendly implements Rule
{
    public function passes($attribute, $value)
    {
        return !$this->service->checkForBadWords($value);
    }
}
```

(kod studentów, pisownia oryginalna)

Martin pisze w swojej książce, że funkcje powinny być krótkie oraz krótsze niż są.

Im krótsza funkcja, tym łatwiej ją przeczytać i zrozumieć. Łatwiej też nie złamać SRP.

```

<?php

namespace App\Http\Controllers;

use App\Guest;
use App\Http\Requests\TaskRequest;
use App\Http\Resources\TaskResource;
use App\Task;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Log;

class TaskController extends Controller
{
    /** (...) */

    public function update(Request $request, Guest $guest, Task $task)
    {
        $current_user = Auth::user();
        $current_user_id = Auth::user()->id;
        $weddings = $current_user->weddings()->wherePivot('user_id', '=', $current_user_id)->first();
        if (($weddings->pivot->wedding_id == $guest->wedding_id) && ($weddings->pivot->wedding_id ==
$task->wedding_id) && ($task->guest_id == $guest->id)) {
            $task->update($request->only(['task_name', 'note', 'for_everyone', 'date', 'is_finished']));
            Log::channel('weddings_logs')->info('User with id: '.$current_user_id.' updated task');
            return new TaskResource($task);
        }else{
            return response()->json('error',403);
        }
    }
}

```

(kod studentów, pisownia oryginalna)

Wcięcia i bloki bardzo ułatwiają czytanie funkcji. Ale ile poziomów wcięć jesteśmy w stanie zaakceptować w naszym kodzie?

```
package Post;

import java.util.ArrayList;
import java.util.Random;
import java.util.Scanner;

final public class Post
{
    public static ArrayList<String> post = new ArrayList<>();

    public static ArrayList<String> PostList()
    {
        int listpoint;
        for(listpoint=0 ; listpoint<post.size(); listpoint++)
            System.out.println(post.get(listpoint));
        return post;
    }
}
```

(kod studentów, pisownia oryginalna)

I wracając jeszcze do SRP:

*Funkcje powinny wykonywać jedną operację.*

*Powinny to robić dobrze.*

*Powinny robić tylko to.*

```

package ParkingLotManager;

final public class ParkingLot {

    /** (...) */

    public boolean checkIfCanEnter(EntityInterface entity) {
        fillBlackList();
        if(isParkingFull(entity)) {
            entity.setEntry(false);
            if(entity instanceof PrivilegedVehicles || entity instanceof Courier) {
                entity.setEntry(true);
            }
        }
        if(isOnBlackList(entity)) {
            Log.info("You're under arrest, you have the right to maintain your silence...");
            entity.setEntry(false);
        }
        if(!isMoneyGoodEnough(entity.getMoney(),entity)) {
            entity.setEntry(false);
        }
        if(entity instanceof Tank) {
            Log.info("You're too fat!");
        }

        return entity.canEnter();
    }
}

```

(kod studentów, pisownia oryginalna)



Skomplikowane *switch-casy* czy zestawy *ifów* bardzo zagrażają podstawowym zasadom tworzenia dobrego kodu. Czy istnieje sposób *ulepszenia* poniższego kodu?

```
public class ZooEmployee {  
  
    private String name;  
    private static boolean canFeed = false;  
  
    public ZooEmployee() {  
  
    }  
  
    public ZooEmployee(String name) {  
        this.name=name;  
        Log.info(name + " has been recruited!");  
    }  
  
    public Food feedingPlan(int hour) {  
        Food food = null;  
        if(hour==8) {  
            food = new Meat();  
        }  
        else if(hour==9) {  
            food = new Milk();  
        }  
        else if(hour == 12) {  
            food = new Water();  
        }  
        else if(hour == 15) {  
            food = new Fruits();  
        }  
        else if(hour == 13) {  
            food = new Vegetables();  
        }  
        return food;  
    }  
}
```

(kod studentów, pisownia oryginalna)

*Idealną liczbą argumentów dla funkcji jest zero (funkcja bezargumentowa). Następnie mamy jeden (jednoargumentowa) i dwa (dwuargumentowa).*

```
/**
 * @param BookProductRepository $bookProductRepository
 * @param CartContextInterface $cartContext
 * @param OrderProcessorInterface $orderProcessor
 * @param ObjectManager $manager
 * @param TaxonRepository $taxonRepository
 * @param ProductRepositoryInterface $productRepository
 * @param RejectedBookService $rejectedBookService
 * @param BookRecordRepository $bookRecordRepository
 * @param ChannelContextInterface $channelContext
 * @param LocaleContextInterface $localeContext
 * @param BookCookieHelper $bookCookieHelper
 */
public function __construct(
    BookProductRepository $bookProductRepository,
    CartContextInterface $cartContext,
    OrderProcessorInterface $orderProcessor,
    ObjectManager $manager,
    TaxonRepository $taxonRepository,
    ProductRepositoryInterface $productRepository,
    BookRecordRepository $bookRecordRepository,
    RejectedBookService $rejectedBookService,
    ChannelContextInterface $channelContext,
    LocaleContextInterface $localeContext,
    BookCookieHelper $bookCookieHelper
)
{
    $this->bookProductRepository = $bookProductRepository;
    $this->cartContext = $cartContext;
    $this->orderProcessor = $orderProcessor;
    $this->manager = $manager;
    $this->taxonRepository = $taxonRepository;
    $this->productRepository = $productRepository;
    $this->bookRecordRepository = $bookRecordRepository;
    $this->rejectedBookService = $rejectedBookService;
    $this->channelContext = $channelContext;
    $this->localeContext = $localeContext;
    $this->bookCookieHelper = $bookCookieHelper;
}
```

Czy *setter* powinien zwracać wynik? A może warto zastanowić się nad separacją zapytań od poleceń?

```
<?php

namespace Map\Helpers;

use Map\Exceptions\InvalidCoordinateValue;

/**
 * Class Coordinates
 * @package Map\Helpers
 */
class Coordinates {


    public function setLatitude(float $latitude): void {
        $this->validateLatitude();
        $this->latitude = $latitude;
    }

    public function setLatitude(float $latitude): bool {
        $this->validateLatitude();
        $this->latitude = $latitude;
        return true;
    }

    public function setLatitude(float $latitude): string {
        $this->validateLatitude();
        $this->latitude = $latitude;
        return $this->getFormattedCoordinate($this->latitude);
    }

    protected function validateLatitude(float $latitude): void {
        if(abs($latitude) > 90) {
            throw new InvalidCoordinateValue("Invalid value. Latitude cannot be larger than ±90deg.");
        }
    }
}
```

Jak powinny być obsługiwane błędy, których się spodziewamy? Czy funkcje powinny zwracać tylko to, czego oczekujemy? Jeżeli tak to w jakim formacie?



```
public function oczekiwanieNaGracza(Request $request)
{
    $list = DB::table('game')
        ->where("id_game", "=", $request->game_id)
        ->whereNotNull("gamer2_id")
        ->select("*")
        ->first();

    if ($list)
        return response()->json(true);
    else
        return response()->json(false);
}
```

(kod studentów, pisownia oryginalna)



Kod poniżej pochodzi z zeszłosemestralnego studenckiego projektu. Co można byłoby w nim poprawić?

<https://pastebin.com/raw/a0B72eik>

# Formatowanie kodu

---

*Formatowanie kodu jest ważne.*

*Jest zbyt ważne, aby je ignorować, i zbyt ważne, aby traktować je dogmatycznie. Formatowanie kodu ma zapewnić komunikację, a dobra komunikacja jest pierwszą zasadą biznesu zawodowego programisty.*

Warto wykorzystać pojęcie *gęstości pionowej* i zewrzeć ze sobą wiersze mające ścisłe związki ze sobą.

```

/**
 * "Delete" user
 *
 * @param [integer] id
 * @return [string] message
 */
public function deleteUser(Request $request)
{
    $request->validate([
        'id' => 'required|integer|exists:users'
    ]);
    $user = $request->user()->where('id',$request->id)->first();
    if ($user->is_admin == 1) {
        return response()->json([
            'message' => 'Nie możesz usunąć admina'
        ], 401);
    }
    if ($user->deleted_at !== null)
    {
        return response()->json([
            'message' => 'Użytkownik już został usunięty'
        ], 200);
    }
    $user->deleted_at = Carbon::now();
    $user->save();
    return response()->json([
        'message' => 'Usunięto użytkownika!'
    ], 200);
}

```

(kod studentów, pisownia oryginalna)

```
/**
 * "Delete" user
 *
 * @param [integer] id
 * @return [string] message
 */
public function deleteUser(Request $request)
{
    $request->validate([
        'id' => 'required|integer|exists:users',
    ]);
    $user = $request->user()->where('id', $request->id)->first();
    if ($user->is_admin == 1) {
        return response()->json([
            'message' => 'Nie możesz usunąć admina',
        ], 401);
    }
    if ($user->deleted_at !== null) {
        return response()->json([
            'message' => 'Użytkownik już został usunięty',
        ], 200);
    }
    $user->deleted_at = Carbon::now();
    $user->save();
    return response()->json([
        'message' => 'Usunięto użytkownika!',
    ], 200);
}
```

(kod studentów, po autoformacie)

```

/**
 * Soft delete user
 *
 * @param Request $request
 * @return Response
 */
public function deleteUser(Request $request): Response
{
    $request->validate([
        'id' => 'required|integer|exists:users',
    ]);
    $user = $request->user()->where('id', $request->id)->first();
    if ($user->is_admin == 1) {
        return response()->json([
            'message' => 'Nie możesz usunąć admina',
        ], 401);
    }
    if ($user->deleted_at !== null) {
        return response()->json([
            'message' => 'Użytkownik już został usunięty',
        ], 200);
    }
    $user->deleted_at = Carbon::now();
    $user->save();
    return response()->json([
        'message' => 'Usunięto użytkownika!',
    ], 200);
}

```

(kod studentów, po zmianie opisu funkcji)

```
/**
 * Soft delete user
 *
 * @param DeleteUserRequest $request
 * @return Response
 */
public function deleteUser(DeleteUserRequest $request): Response
{
    $user = User::withoutAdmin()->findOrFail($request->get("id"));
    $user->deleted_at = Carbon::now();
    $user->save();
    return response()->json([
        'message' => 'Usunięto użytkownika!',
    ], 200);
}
```

(kod studentów, po oczyszczeniu funkcji)





```
/**
 * Soft delete user
 *
 * @param DeleteUserRequest $request
 * @return Response
 */
public function deleteUser(DeleteUserRequest $request): Response
{
    $user = User::withoutAdmin()->findOrFail($request->get("id"));

    $user->deleted_at = Carbon::now();
    $user->save();

    return response()->json([
        'message' => 'Usunięto użytkownika!',
    ], 200);
}
```

(kod studentów, po zwiększeniu pionowych odstępów)

Dobrym zwyczajem może być układanie metod w klasie pod względem ich znaczenia.

*Wywoływana funkcja powinna znajdować się poniżej funkcji wywołującej.*

```
<?php

namespace App\Http\Controllers\API;

class AnswerController extends APIController
{
    private function getAnswerResponse(Game $game, UserAnswer $user_answer)
    {
        return $this->response
            ->setMessage($this->message)
            ->setData([/** (...) */])
            ->setSuccessStatus()
            ->getResponse();
    }

    public function skipCurrentPlayerTurn(Request $request)
    {
        /** (...) */
        return $this->getAnswerResponse($game, $user_answer);
    }
}
```

(kod studentów, pisownia oryginalna)

Wiersze nie powinny być zbyt długie, ale nigdy nie powinien być to kontrargument dla pisania kodu bez odstępów.

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class BaseController extends Controller
{
    public function Success($data=null)
    {
        $output['ok']=true;
        $output['data']=$data;
        return response()->json($output);
    }
    public function Failure($code=0,$messages=null)
    {
        $output['ok']=false;
        $output['code']=$code;
        $output['messages']=$messages;
        return response()->json($output);
    }
}
```

(kod studentów, pisownia oryginalna)

Nie warto przejmować się natomiast udziwnieniami formatowania kodu, które z różnych przyczyn są popularne w niektórych środowiskach. Najważniejsza jest czytelność.

```
class Response
{
    const HTTP_CONTINUE = 100;
    const HTTP_SWITCHING_PROTOCOLS = 101;
    const HTTP_PROCESSING = 102; // RFC2518
    const HTTP_EARLY_HINTS = 103; // RFC8297
    const HTTP_OK = 200;
    const HTTP_CREATED = 201;
    const HTTP_ACCEPTED = 202;
    const HTTP_NON_AUTHORITATIVE_INFORMATION = 203;
    const HTTP_NO_CONTENT = 204;
    const HTTP_RESET_CONTENT = 205;
    const HTTP_PARTIAL_CONTENT = 206;
    const HTTP_MULTI_STATUS = 207; // RFC4918
    const HTTP_ALREADY_REPORTED = 208; // RFC5842
    const HTTP_IM_USED = 226; // RFC3229
    const HTTP_MULTIPLE_CHOICES = 300;
    const HTTP_MOVED_PERMANENTLY = 301;
    const HTTP_FOUND = 302;
    const HTTP_SEE_OTHER = 303;
    const HTTP_NOT_MODIFIED = 304;
    const HTTP_USE_PROXY = 305;
    const HTTP_RESERVED = 306;
    const HTTP_TEMPORARY_REDIRECT = 307;
    const HTTP_PERMANENTLY_REDIRECT = 308; // RFC7238
    const HTTP_BAD_REQUEST = 400;
    const HTTP_UNAUTHORIZED = 401;
    const HTTP_PAYMENT_REQUIRED = 402;
    const HTTP_FORBIDDEN = 403;
    const HTTP_NOT_FOUND = 404;
    const HTTP_METHOD_NOT_ALLOWED = 405;

    /** (...) */
}
```

Wcięcia to podstawa.

*Nasze oko może szybko określić strukturę wcięć w pliku. Można niemal natychmiast odszukać zmienne, konstruktory, akcesory i metody.*



```
class Game(object):
    champions = []
    scoreboard = Scoreboard()
    turn_counter = 0

    def add_champion(self, champion):
        self.champions.append(champion)

    def process_turn(self):
        self.turn_counter = self.turn_counter + 1
        attacker = random.choice(self.champions)
        target = attacker

        while attacker == target:
            target = random.choice(self.champions)

        attacker.attack(target)

        if target.life_points <= 0:
            self.champion_death(target)

# (...)
```

(kod studentów, pisownia oryginalna)

# Podsumowanie

---

**Pytania?**

Kod prezentacji dostępny jest w repozytorium git pod adresem  
<https://bitbucket.org/krewak/pwsz-zmp>



Wszystkie informacje dot. kursu dostępne są pod adresem  
<http://pwsz.rewak.pl/kursy/10>

