Warsztat Inżyniera Oprogramowania (bardzo mały wstęp)

Sztuka Wytwarzania Oprogramowania, w. 2

Konrad Grochowski

Instytut Informatyki, Politechnika Warszawska, 2024 ©

Sztuka Wytwarzania Oprogramowania

- *Software craftsmanship*
- Osobiście wolę tłumaczenie „rzemiosło”
- Talent + *warsztat* (pełen narzędzi)
- Dadaizm, prymitywizm, czy ogólnie rozumiana sztuka abstrakcyjna *nie powinna* trafiać do oprogramowania
- Mini dygresja: ten *warsztat* **nigdy** nie będzie kompletny
  - bądź *zamrożony*
- Standardy kodowania są jednym z narzędzi

Standardy kodowania

- Cel: usystematyzowanie procesu wytwarzania kodu i samego kodu
- Ujednolicenie – usprawnienie utrzymywalności (*maintainability*) i pracy w zespole
- Zbiór reguł ma pomagać osiągnąć zakładaną jakość
- *Coding conventions, coding standards, coding style* – stosowane często zamiennie, ale czasem ktoś rozróżnia *style* od *standard*
- Bo *coding standard* to może być więcej niż głębokość wcięć...
  - Czy *tabs vs spaces*
- Mini dygresja:
  - dostosowanie się do obowiązującego *coding standardu* stanowi test „dojrzałości”
  - usprawnianie standardów to cecha lidera i eksperta, ale konsensus jest kluczowy

Standard kodowania - Styl

- Czyli: wcięcia, pozycje *klamerek*, białe znaki, białe linie etc.

```
int* a          vs  int *a
while( a==b )   vs  while (a == b)
if (x) {        vs  if (x)
                     {
```

- Niektóre języki *proponują* style opracowane przez twórców np. Python (PEP8), C# (C# Coding Conventions)
- I na szczęście coraz częściej raczej się go *wybiera*, niż *wymyśla*
- Wbrew pozorom *nie musi* być składnikiem standardu

Standard kodowania - Nazewnictwo

› Zarówno *notacja* jak i *semantyka*

```
Nazwa rzeczownik;          vs  LPSTR str1;
rzeczownik->czasownik();   vs  str1->m_len;
bool isManager;            vs  bool manager;

DługaNazwa                 vs  dluga_nazwa
```

Standard kodowania – co więcej?

- Reguły komentowania kodu
- Metryki kodu (długość funkcji, liczba parametrów etc.)
- Wykorzystywane paradygmaty
  (np. czy programujemy obiektowo w C)
- Reguły bezpiecznego kodu
  (np. nie używamy *gołych* wskaźników w C++,
  wszystkie parametry funkcji powinny być *const*)
- Reguły wynikające ze specyfiki przemysłu
  (np. zakaz używania dynamicznej alokacji)
- Wiele innych...

Standard kodowania – podsumowanie

- Nie ma *jedynego słusznego poprawnego zawsze*
- W idealnym świecie jest osiągany przez konsensus zespołu
- Zespoły mogą używać różnych, a nawet ten sam zespół może używać wielu w różnych projektach
- Rewolucje w inżynierii rzadko są lepsze od ewolucji (przestarzały standard trzeba powoli i konsekwentnie *przepychać* ku jasnej stronie mocy)
- Istnieją *formalne* standardy przemysłowe, czasem wymagane w konkretnej branży (np. MISRA C), wtedy mniejszy wybór...
- Istnieją standardy (reguły) przemysłowe *de-facto*, które można stosować do wielu zestawów: KISS, DRY, GRASP czy SOLID

SOLID

- Reguły dobrego projektowania kodu obiektowego
- Zlepek innych skrótów:
  - **S**RP – Single Responsibility Principle
  - **O**CP – Open-Closed Principle
  - **L**SP – Liskov Substitution Principle
  - **I**SP – Interface Segregation Principle
  - **D**IP – Dependency Inversion Principle
- To są zalecenia i, jak wszystko w inżynierii, trzeba podchodzić do nich z rozsądkiem (płynącym niestety z doświadczenia)

SRP - Single Responsibility Principle

- Klasa/funkcja powinna robić tylko jedną rzecz
- Prosta reguła, a potrafi radykalnie wpłynąć na jakość kodu
- Robert C. Martin –
  klasę powinno się zmieniać tylko z jednego powodu

- Jeśli opisując klasę muszę powiedzieć „i”, to zapala mi się lampka
- Np.: jak w klasie `Circle` są metody
  `saveTo(File) i draw(Screen)`

OCP – Open-Closed Principle

- *Moduły powinny być otwarte na rozszerzenie,*
  ale zamknięte na modyfikacje
- Czyli – moduł powinien pozwalać dodawać sobie zachowanie, ale zmiana zachowania innego modułu, bądź dodanie nowego, nie powinna wpływać na mój
- A także – nie powinniśmy musieć dotykać kodu modułu, by wykorzystać jego funkcje w inny sposób
- Np.: `saveTo(File)`
  vs `saveTo(Stream)`
  vs `saveUsing(`*`Serializer`*`)`

LSP – Liskov Substitution Principle

- *Funkcje które używają referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów*

Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

- Czyli – każdy obiekt pochodny po obiekcie bazowym powinien być jego *dobrym zastępnikiem*
- Kluczowa reguła dla dobrego zrozumienia *kontraktów klas*
- Np.: implementacja `saveUsing(`*`Serializer`*`)` powinna móc traktować każdą implementację interfejsu *`Serializer`* w ten sam sposób, a zapisany obiekt powinien się dać poprawnie odczytać

LSP – Liskov Substitution Principle

- Jeśli `Rectangle` ma metody:
  - `setWidth`
  - `setHeight`
- To czy Square może po nim dziedziczyć?
- *Bonus: a może po prostu* `set*` *są już „bez sensu”?*

ISP – Interface Segregation Principle

- *Wiele dedykowanych interfejsów jest lepsze niż jeden ogólny*
- Klient klasy nie powinien być zmuszany by zależał od interfejsu, którego nie potrzebuje
- To trochę jak SRP ale „od zewnątrz” – jeśli potrzebuję metody `draw` to nie powinienem „widzieć” też metody `saveUsing` i w efekcie zależeć od *`Serializer`*
- Ale klasy zgodne z SRP też mogą skorzystać na ISP: np.: klasa implementująca mapowanie obiektu do bazy danych i udostępniająca interfejs *`Serializer`* mogła by też udostępniać *`Deserializer`*

ISP – Interface Segregation Principle

- *Ale żeby pokazać, że te dyskusje można odbywać na wielu poziomach abstrakcji:*
- Weźmy

```
auto area(std::shared_ptr<Rectangle> r) {
      return r->width() * r->height();
}
```

- Tutaj do metody przekazane są tak naprawdę dwa interfejsy:
  `Rectangle` i `shared_ptr`
- A użyty tylko jeden
- To może jednak

```
auto area(const Rectangle& r) {
      return r.width() * r.height();
}
```

- *Bonus: jak spojrzeć na to od strony wywołania to ta metoda łamie też OCP. Większość „złego” kodu łamie więcej niż jedną zasadę, co też oznacza, że można go „podejść” wychodząc od różnych zasad (tutaj nawet KISS mógłby się zapalić).*
  Nie jest ważne którą regułkę się rozpoznało, ale żeby kod był dobry.

DIP – Dependency Inversion Principle

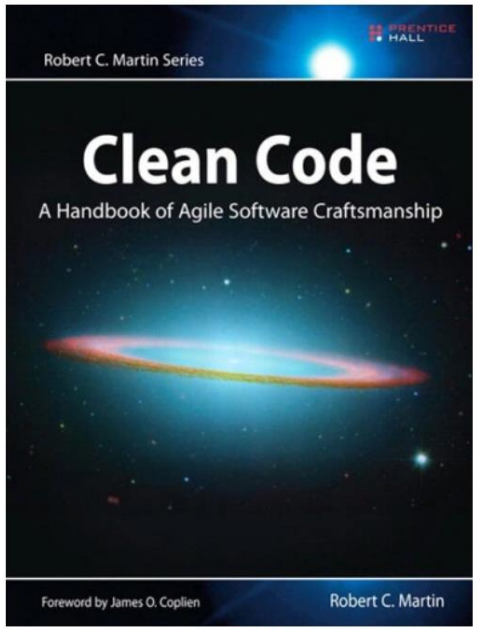
- *Abstrakcje nie powinny zależeć od konkretów, ale na odwrót*
- Inaczej: obiekty wysokiego poziomu i obiekty niskiego poziomu, zamiast zależeć jakoś od siebie nawzajem, zależą od wspólnej abstrakcji

Clean Code

- Współtwórca i propagator SOLID – Robert C. Martin (*Uncle Bob*) opublikował w 2009 książkę zbierającą wiele porad i definiującą podejście *Clean Code* (potem *Clean Coder,* też warto)
- Moim zdaniem – jedna z obowiązkowych pozycji każdego, kto chce tworzyć oprogramowanie (ale jak wszystko – nie jest to bezwzględna wykładnia Prawdy przed duże P)
- Główne przesłanie: dobry kod czyta się jak prozę

Clean Code – hasła, które pobudzają dyskusję

- *Komentarze kłamią*
- *Jeśli czuję, że muszę skomentować kod, to zastanawiam się, co zrobiłem źle*
- *Funkcje mogą mieć po max. 3 - 4 linijki*
- *TDD wszędzie*



Dziękuję za uwagę

Konrad.Grochowski@pw.edu.pl