Clean code & Dobre praktyki OOP

Marcin Chrost

Agenda

- Kilka słów o mnie
- Dobry kod a zły kod
- Clean code – zasady ogólne
- Clean code – zasady szczegółowe
- Dobre praktyki OOP – Zasady SOLID
- Pytania & dyskusja

Who am I ?

sages

BOTTEGA
IT minds

JSYSTEMS
INSPIRACJA - WIEDZA - REALIZACJA

Marcin Chrost

https://chrost.eu

What I teach ?

Reactive

Project Reactor

Clean Code
Novelties

Who I teach?


motorola


Lufthansa


ING


BNP
PARIBAS


HUUUGE


SŁUŻBA
WIĘZIENNA

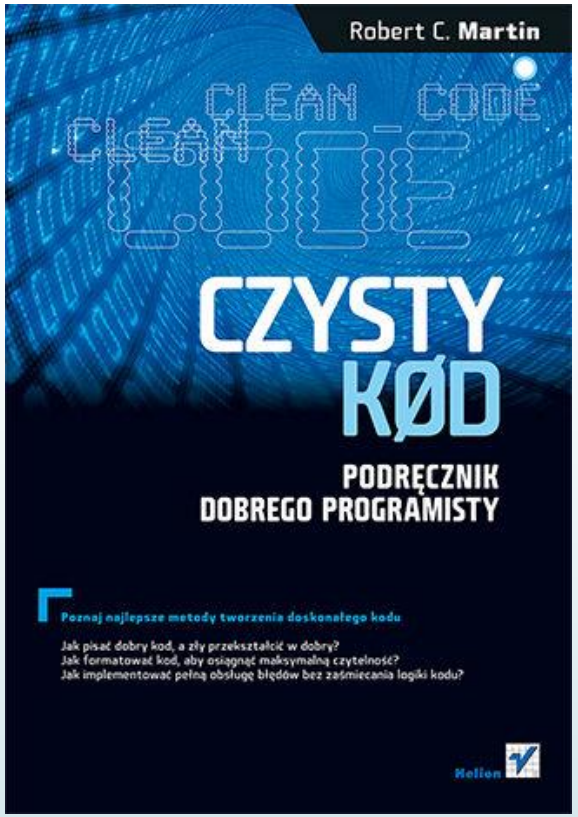

ZETO
KATOWICE


BSB
up to excellence

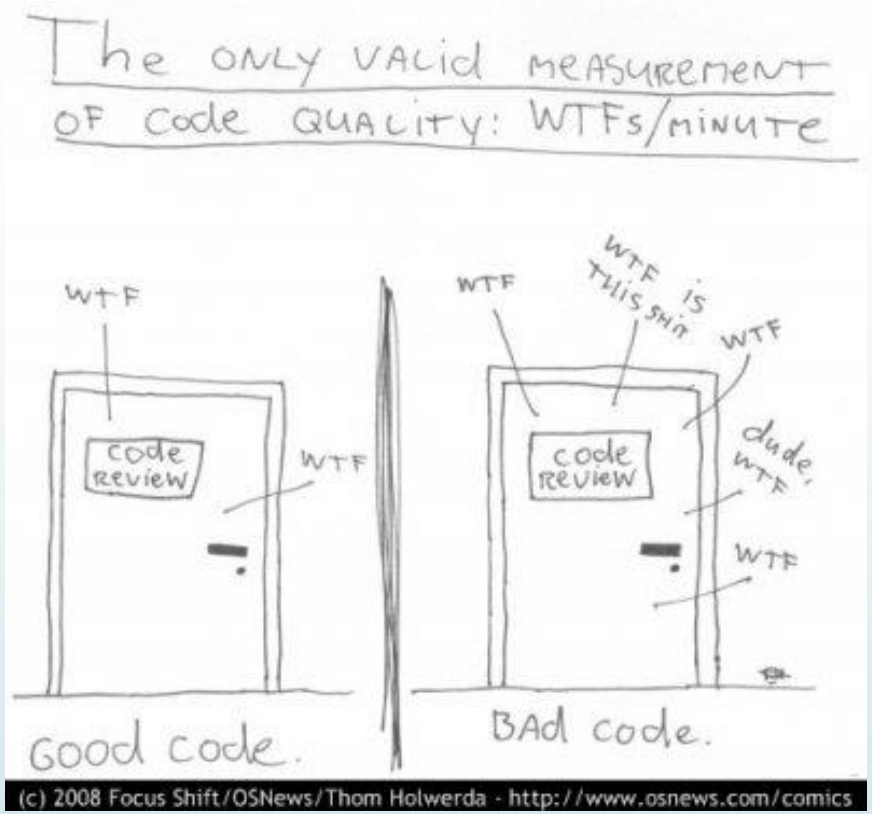

ocado


JCommerce

Dobry kod a zły kod



Lektura obowiązkowa



Jedyna prawdziwa miara jakości kodu – WTF/min

Kod słabej jakości - przyczyny

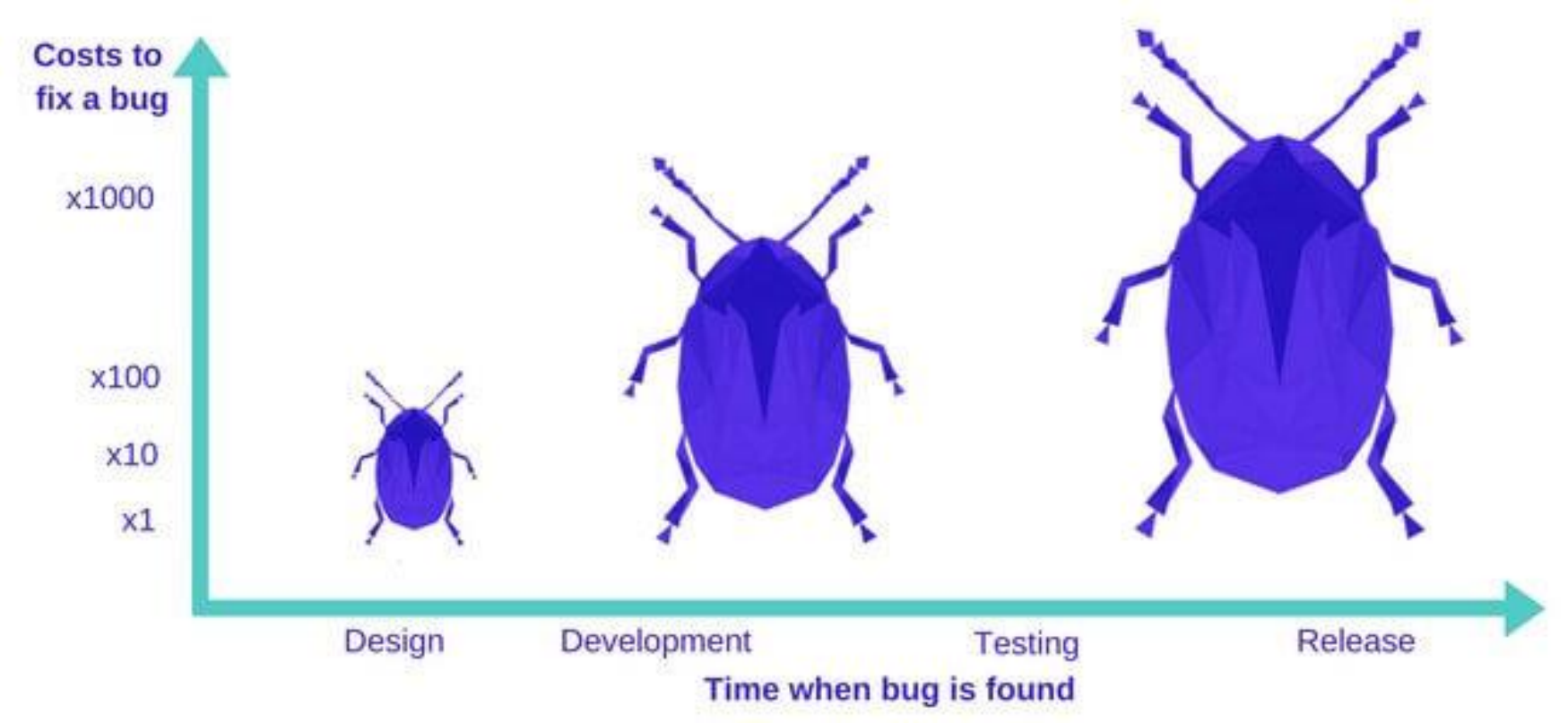
- Pośpiech !!!
- Złe zarządzanie zasobami (czas, ludzie, pieniądze)
- Nieprzestrzeganie dobrych praktyk
- Brak tak zwanej odwagi cywilnej i profesjonalizmu zawodowego
- *Później znaczy nigdy*

Kod słabej jakości – objawy kliniczne

- Skostniałość - system jest trudny w modyfikacji, drobna z pozoru zmiana, pociąga za sobą dziesięć innych
- Kruchość - zmiana w jednym miejscu systemu powoduje, że nie działa coś w innym miejscu
- Zbitość - trudno podzielić system na fragmenty, które mogą być wielokrotnie używane
- Nowotworowość - łatwiej jest zrobić „hacka" i obejść fragment rozwiązania niż go zmodyfikować, efekt – zły kod rozsiewa się tak szybko jak nowotwór
- Złożoność - jest dużo skomplikowanego, często niepotrzebnego kodu
- Powtórzenia - kod wygląda jak pisany metodą ctrl+C / ctrl+V
- Nieprzejrzystość - projekt jest tak zawiły, że nikt nie wie o co w nim chodzi

Kod słabej jakości - konsekwencje

- Duża szansa na pojawienie się błędów (i to w niespodziewanych miejscach)
- Niska produktywność (zarówno podczas rozwoju jak i utrzymania)
- Niemożność oszacowania ile zajmie zmiana w kodzie / poprawa błędu
- Wypalenie zawodowe
- Rotacja programistów
- Nadgodziny
- Bardzo napięte stosunki z biznesem



Koszt naprawy błędu – kula śniegowa

Czysty kod

Nie ma jednej dobrej definicji tego czym jest czysty kod. Można jednak spróbować go scharakteryzować jako taki, który:

- Wygląda jakby był napisany przez kogoś komu na tym zależało
- Jest prosty i bezpośredni
- Dobrze się go czyta
- Nie zawiera rzeczy, które natychmiast proszą się o poprawienie
- Wszystkie mechanizmy ma ulokowane tam gdzie się ich spodziewamy
- Podświadomie zniechęca nas do próby jego zepsucia

Clean code – zasady ogólne

Zasada skautów

„*Zawsze zostawiaj obóz czystszym, niż go zastałeś*”

- W przełożeniu na język tworzenia oprogramowania – jeżeli zmieniasz coś w danym obszarze kodu, zrób tak aby kod wynikowy był lepszy niż wejściowy.
- Nie musi to być wielka zmiana, wystarczy np. drobna poprawa nazewnictwa
- Lekceważenie takiej postawy, kończy się tzw. syndromem rozbitych okien

DRY – Don't Repeat Yourself

Każda porcja wiedzy / logiki biznesowej w programie musi być zawarta w dokładnie jednym, nie budzącym wątpliwości miejscu

- Złamanie zasady DRY może doprowadzić do „rozjechania" się logiki biznesowej programu w sytuacji gdy podczas wprowadzania zmian pominiemy któreś z miejsc, w których zawarta jest duplikująca się wiedza.
- Powtórzenia mogą przyjmować różne formy:
  - copy-paste – najłatwiejsze do wykrycia i naprawienia
  - te same ciągi instrukcji if-else / switch – do zastąpienia przez polimorfizm
  - identyczne algorytmy – do podmiany przez metodę szablonową / strategię

Zasada najmniejszego zaskoczenia

Każda metoda / klasa powinna być zaimplementowana tak, jakby inna osoba tego oczekiwała

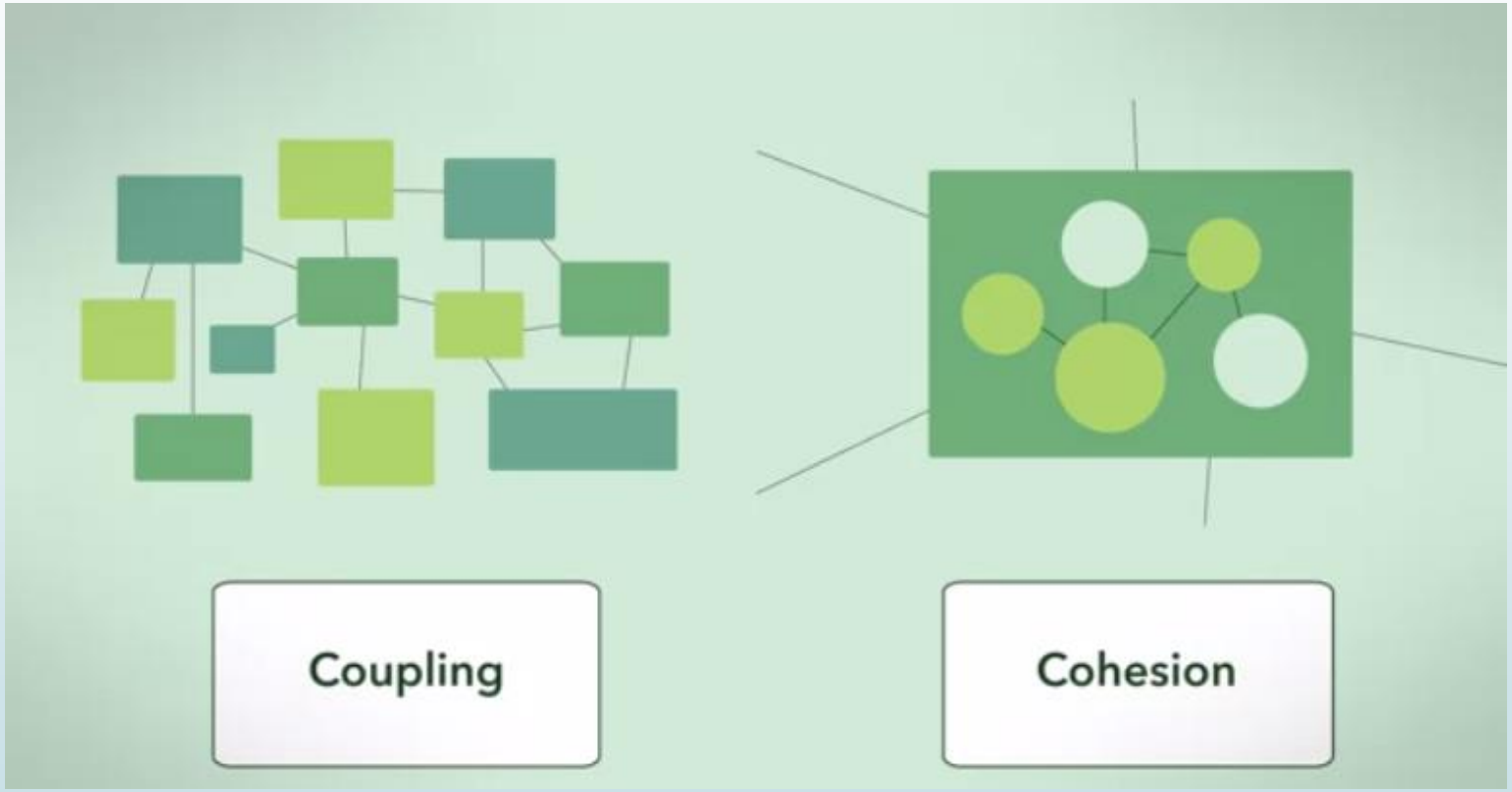
- Przykłady złamania zasady:
  - metoda `toString()` nie zwraca tekstowej reprezentacji obiektu
  - metoda o nazwie `getXYZ()` zmienia stan obiektu
- Dodatkowo jeżeli przyjęliśmy jakąś konwencję (np. nazewniczą) to należy trzymać się jej w całym kodzie
- Złamanie zasady bardzo negatywnie wpływa na szybkość pracy z kodem zarówno w zakresie jego rozwijania, jak też i poprawiania błędów (wynika to m.in ze sposobu w jaki działa ludzki mózg – nie mamy możliwości koncentracji na wielu rzeczach naraz)

High cohesion, low coupling

- *Cohesion* (spójność) jest miarą tego jak bardzo związany jest ze sobą kod w danej jednostce (klasie, module, pakiecie)
- Niska spójność oznacza iż w obrębie takiej jednostki znajdują się co najmniej dwie grupy niezwiązanego ze sobą kodu – co utrudnia na przykład reużycie jednej z nich
- *Coupling* (zależność) jest z kolei metryką opisującą jak związany jest ze sobą kod pomiędzy dwoma różnymi jednostkami
- Wysoka zależność doprowadza do tego, iż nie da się w prosty sposób zmienić jednej jednostki tak aby nie była konieczna zmiana w drugiej – w efekcie kod staje się wrażliwy nawet na drobne zmiany (kruchość)

Docelowo kod powinien posiadać wysoką spójność i niską zależność

High cohesion, low coupling



KISS / YAGNI

- Zasada *KISS* (*Keep It Simple Stupid*) oznacza iż należy tworzyć kod, który jest w stanie rozwiązać dany problem w najprostszy możliwy sposób bez zbędnych udziwnień.
- Zasada *YAGNI* (*You Ain't Gonna Need It*) z kolei mówi iż nie należy tworzyć funkcjonalności „na zapas" – z przeznaczeniem iż będzie „kiedyś potrzebna".
- Obie zasady wynikają z faktu iż każda fragment kodu == potencjalny problem w przyszłości

Zewnętrzne biblioteki / frameworki

- ZAWSZE należy rozważyć skorzystanie z zewnętrznej biblioteki / frameworka, jeżeli tylko jest taka możliwość – zamiast tworzyć własne rozwiązania
- Dzięki temu:
  - Redukujemy rozmiar stworzonego kodu
  - Nie popełniamy drugi raz tych samych błędów ☺
- Rzeczy do rozważenia przy wyborze biblioteki / frameworka
  - Żywotność
  - Licencja

Iteracyjne tworzenie kodu

Czysty kod nigdy nie powstaje od razu

- Do tworzenia kodu trzeba podejść podobnie jak do wypracowania:
  - Tworzymy najpierw szkic / brudnopis. W przypadku kodu jest to wersja, która działa (ale wygląda źle). **NIE ODDAJEMY TAKIEJ WERSJI !**
  - Poprawiamy / czyścimy fragmenty tekstu (refactor kodu – o tym na kolejnym slajdzie). Czynność powtarzamy wiele razy
  - Proces kończymy w momencie gdy uznamy że można oddać wypracowanie / wystawić PR ☺

Refactor istniejącego kodu

- Chaotyczne próby poprawiania istniejącego kodu (szczególnie złożonego) rzadko kiedy kończą się sukcesem.
- Podstawą dobrego procesu refaktoryzacji jest:
  - Podejście do kodu z szacunkiem ☺
  - Zrozumienie co dany kod ma właściwie robić
  - Stworzenie siatki bezpieczeństwa złożonej z testów
- Następnie można wprowadzać poprawki krok po kroku, cały czas kontrolując za pomocą testów czy coś się nie zepsuło.

Clean code – zasady szczegółowe

Nazewnictwo (1)

- Nazwa (identyfikator) zmiennej, metody, klasy itd. powinna
  - wyjaśniać cel istnienia, pełnioną funkcję, sposób użycia, intencje autora
  - dać się wymówić
  - dać się wyszukać
  - być spójna
  - być zrozumiała
- Przykład dobrej nazwy zmiennej : `int elapsedTimeInDays;`
- Stosowanie przemyślanych nazw ułatwia czytanie i zmienianie kodu
- Jeśli dana metoda ma jakieś efekty uboczne, powinny one znaleźć odzwierciedlenie w nazwie

Nazewnictwo (2)

Nazwy zmiennych, które nie powinny być używane:

- Niezgodne z rzeczywistością (`Set<Integer> accountList`)
- Z dodaną liczbą (`String name2`)
- Przekręcone (`int numbarOfMonths`)
- Jednoliterowe (`int d; //number of days`)
- Nieznane skróty (`double mwpr; //monthly worker paing rate`)
- Z oznaczeniem typu (`String phoneString`)
- Zbyt ogólne (`List<Integer> list`)

Formatowanie kodu

- Jest obowiązkowe (znacząco poprawia czytelność i komunikację)
- Powinno być spójne na poziomie całego projektu (**style guide**)
- Dotyczy między innymi:
  - Stosowania wcięć i separatorów w postaci pustej linii
  - Odpowiedniego łamania instrukcji
  - Długości linii
  - Kolejności deklaracji składników klasy
  - Konwencji nazewniczych

Funkcje / metody

- Reguły dotyczące dobrze napisanej funkcji / metody:
  - Przyjmuje jak najmniej parametrów wejściowych (idealnie 0, akceptowalnie 1 – 3, powyżej oznacza potencjalne problemy)
  - Brak parametrów wyjściowych
  - Brak parametru znacznikowego („tryb”)
  - Brak efektów ubocznych
  - Jest krótka (kilka instrukcji)
  - Realizuje jedno zadanie
- Martwe metody powinny zostać usunięte jak najszybciej
- Nazwa metody powinna zawsze zawierać czasownik (pamiętajmy też o zasadach JavaBean – `set, get, is`)
- Jeśli dana metoda ma jakieś efekty uboczne, powinny one znaleźć odzwierciedlenie w nazwie

Klasy

- Dobrze napisana klasa:
  - Realizuje tylko jedno dobrze zdefiniowane zadanie
  - Zachowuje się jak czarna skrzynka (jedynym sposobem interakcji jest publiczne API)
  - Jest konfigurowana zewnętrznymi zależnościami
- Nazwa klasy powinna być rzeczownikiem
- Zdecydowanie należy unikać nazw w stylu `Manager`, `Processor`, `Info`, `Data`
- Martwe klasy również usuwamy najszybciej jak się da (uwaga: refleksja !)

„Zazdrość” o składniki klasy. Prawo Demeter

- „Zazdrość” o składniki klasy występuje wtedy gdy jedna z klas korzysta zbyt często i bezpośrednio z obiektów zawartych w drugiej klasie.
- W efekcie jedna klasa zaczyna być za bardzo związana z wewnętrzną strukturą drugiej, co łamie zasadę niskiej zależności
- Dobrą regułą ograniczającą takie niepotrzebne sprzężenia jest prawo Demeter. Określa ono iż każda metoda danego obiektu może mieć dostęp wyłącznie do:
  - tego obiektu
  - pól i metod tego obiektu
  - argumentów przekazanych do tej metody
  - obiektów stworzonych przez tę metodę
- **Prawo Demeter istotnie ogranicza niepotrzebne powiązania między klasami, jednakże należy je stosować z rozsądkiem, gdyż w skrajnej postaci prowadzi do tworzenia sporej liczby metod - pośredników**

Obsługa błędów / wyjątków

- Preferujemy wyjątki zamiast kodów błędu
- Tam gdzie to możliwe stosujemy wyjątki typu `RuntimeException`
- Klasa wyjątku i komunikat błędu powinny dostarczać pełną informację o tym co się stało
- Nadmierna liczba typów wyjątków nie jest dobra (można pomyśleć o wrapperze)
- Staramy się unikać zwracania i przekazywania wartości null (w tym celu używamy `Optional` / *Null Object*)

Wartości `null`

- Jeżeli to możliwe to nie zwracajmy wartości `null` z metod
  - Szczególnie dotyczy to kolekcji
  - Kod sprawdzający potencjalne wartości null – zaciemnia obraz
  - Można zastosować tzw. special case pattern
- Przekazywanie wartości `null` jest jeszcze gorsze niż jej zwracanie
  - Należy się odgrodzić od takich wartości na poziomie publicznego API
  - I nie przekazywać ich do metod prywatnych
  - `@NonNull` może się tu przydać ☺

Komentarze

- Najlepiej nie powinno być ich wcale – kod powinien opisywać się sam (nie dotyczy to oczywiście publicznego API)
- Komentarze nie powinny opisywać tego, do czego nie są przeznaczone (np. historii zmian w pliku)
- Nieaktualny komentarz jest 100 razy gorszy niż jego brak
- Jeżeli komentarz musi zostać faktycznie napisany, to powinien być stworzony w sposób przemyślany
- **Zdecydowanie unikamy zakomentowanego kodu. Należy go jak najszybciej usunąć !**

Testy jednostkowe

- Są po prostu koniecznością
- Powinny być szybkie, niezależne od siebie, dające jednoznaczny i powtarzalny wynik
- Muszą dać się uruchomić w prosty sposób (jednym kliknięciem / poleceniem)
- Powinny pokrywać możliwie największy zakres danych wejściowych (przypadki graniczne)
- Oprócz samego wyniku, ważna jest też miara pokrycia kodu
- Niedziałające testy należy naprawiać jak najszybciej

Pozostałe zasady (1)

- Bezpieczniki mają swój sens i nie wolno ich wyłączać (np. ostrzeżenia kompilatora, ignorowane testy)
- Nie wolno zakładać tzw. pomyślnej ścieżki. Wartości graniczne i niepoprawne również muszą zostać przetestowane
- Należy udostępniać na zewnątrz tylko to co jest niezbędnie konieczne. Reszta rzeczy powinna być ukryta
- Zmienne globalne są złem i należy ich unikać
- Magiczne numery w kodzie należy zastąpić stałymi

Pozostałe zasady (2)

- Złożone warunki logiczne należy enkapsulować za pomocą metod z opisową nazwą. Zamiast:

```
if (timer.hasExpired() && !timer.isRecurrent())
```

lepiej napisać

```
if (shouldBeDeleted(timer))
```

- Jeśli to tylko możliwe to dobrze jest unikać zanegowanych warunków logicznych – są słabiej czytelne.

Pozostałe zasady (3)

- Należy stosować typy wyliczeniowe (enum), wszędzie tam gdzie jest to możliwe
  - Update: Sealed classes są jeszcze lepsze
- Poza klasami stricte utilowymi, należy używać niestatycznych metod, gdzie to tylko możliwe
  - Jeżeli statyczna metoda przyjmuje jakieś obiekty jako argumenty ➔ pomyśleć o konwersji do metody instancyjnej argumentu

I na sam koniec…

Najpierw pomyśl co właściwie chcesz osiągnąć

A dopiero potem pisz kod ☺

Antywzorce

Antywzorce ➔ nauka na błędach popełnionych przez innych

Antywzorce w programowaniu (1)

Wyjątki / błędy:

- Sterowanie przez wyjątki
- Ukrywanie błędów

Brak elastyczności:

- Hard-coding
- Magiczne liczby / łańcuchy
- Accumulate & fire

Antywzorce w programowaniu (2)

- Accidental complexity
  - „po co komu gotowy DI framework, zróbmy sobie własny”
- Blind faith (ślepa wiara)
  - „szkoda czasu tego sprawdzać, na pewno działa”
- Busy spin
  - „na pewno operacja się już skończyła, sprawdzisz ?”
- Tester-Driven-Development
  - „nasz QA sprawdzi jak to ma faktycznie działać”

Antywzorce w programowaniu (3)

- Blob
  - „ta klasa jest sercem naszego systemu”
- Lava flow
  - „radosna twórczość wczesnego dzieciństwa”
- Poltergeist
  - „pojawiam się i zaraz znikam”
- Boat anchor
  - „pozyskajmy to, na pewno się przyda”

Antywzorce w programowaniu (4)

- Golden hammer
  - „całe życie prałem na tarce, po co mi pralka ?”
- Dead end
  - „ten framework ma błędy, „poprawmy” go”
- Input kludge
  - „3latek zniszczy ten system w 30 sekund”
- Mushroom management
  - „Pisz ten kod i nie zadawaj pytań !”

Dobre praktyki OOP

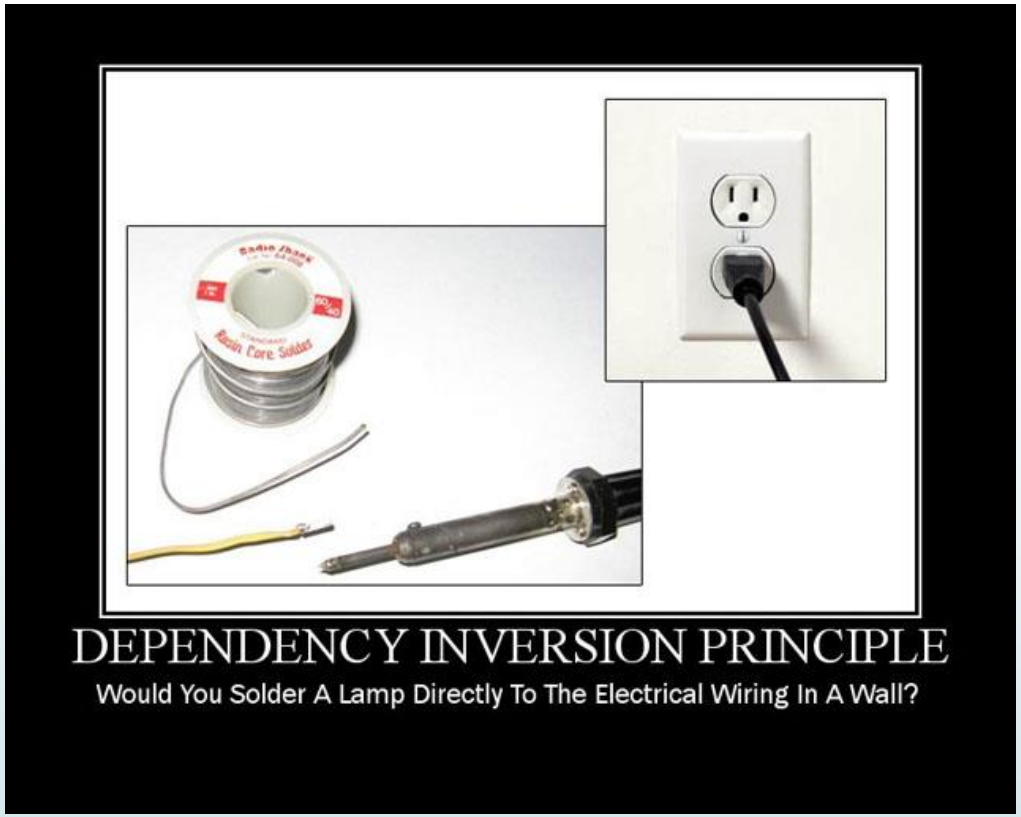
SOLID principles

SOLID principles

5 podstawowych reguł opisujących dobre praktyki programowania obiektowego:

- **S** – *Single responsibility principle*
- **O** – *Open / Closed principle*
- **L** – *Liskov substitution principle*
- **I** – *Interface segregation principle*
- **D** – *Dependency inversion principle*

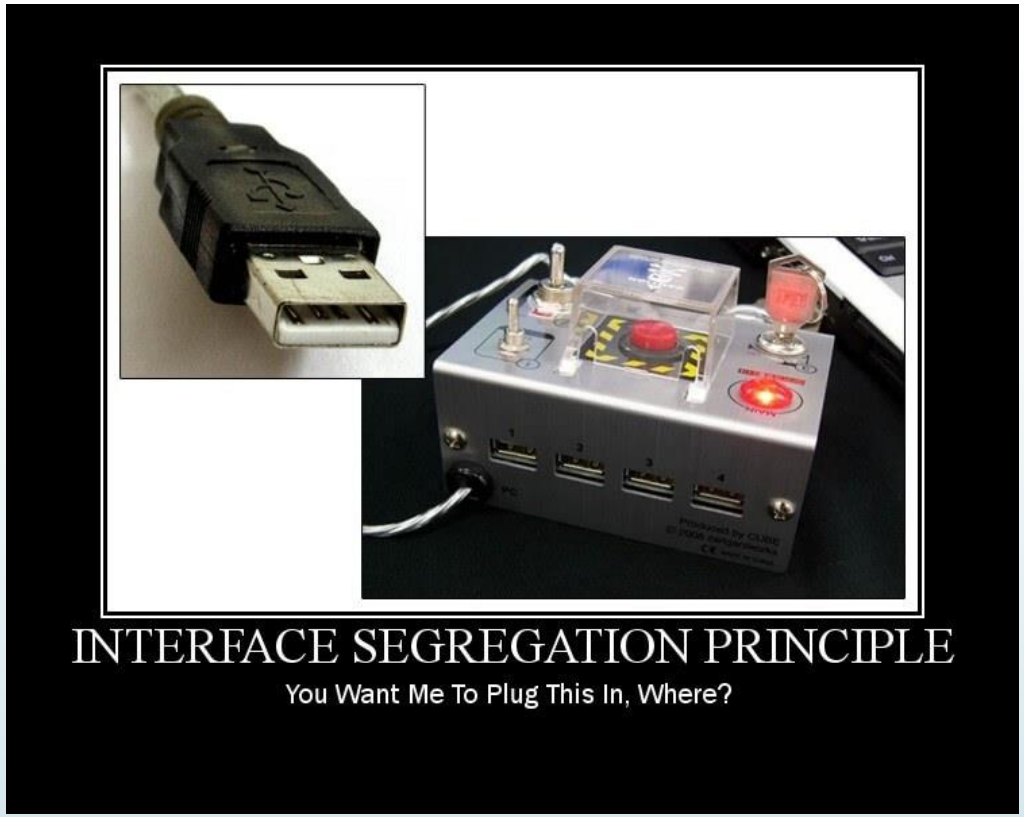
Po raz pierwszy zostały przedstawione przez Roberta Martina w 2000 r. – jako podzbiór promowanych przez niego reguł tworzenia oprogramowania



Dependency inversion principle

Dependency inversion principle (Zasada odwrócenia zależności)

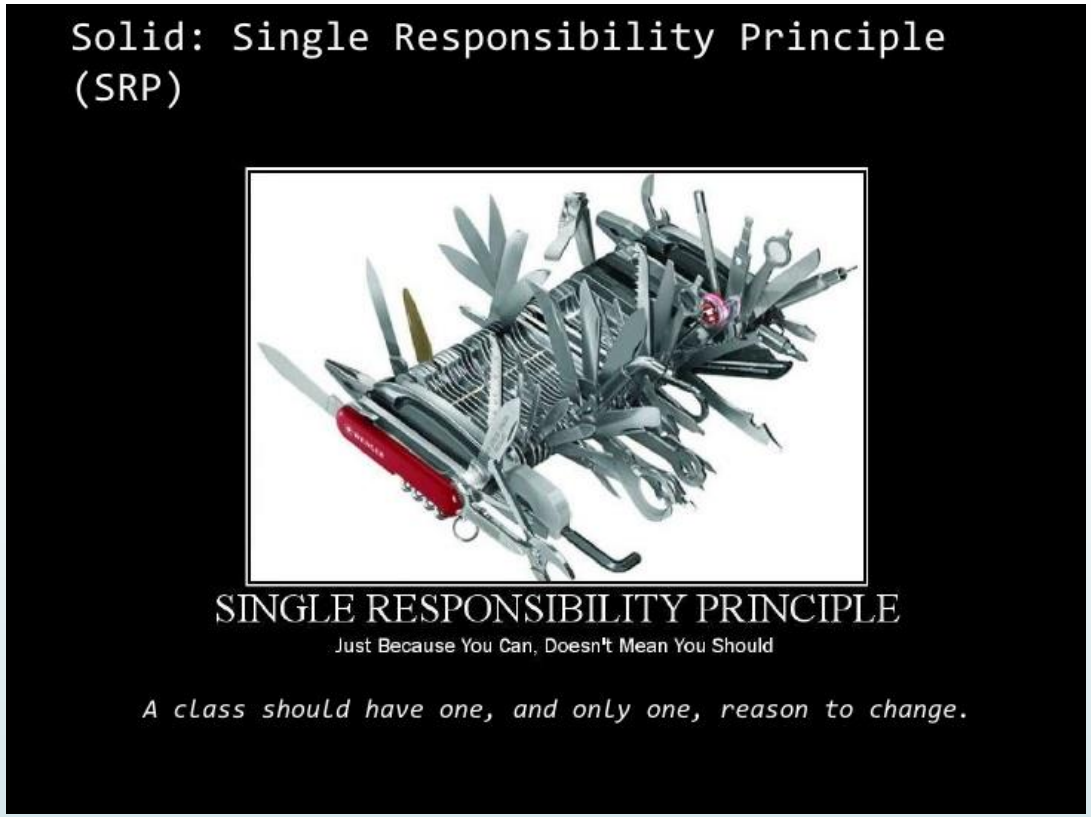
- Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych. Zależności między nimi powinny wyrażone wyłącznie przez abstrakcje (interfejsy)
- Interfejs nie powinien być nigdy zależny od implementacji. To implementacja powinna być zależna od interfejsu
- **Ta zasada zazwyczaj jest najprostsza do spełnienia. Wystarczy w tym celu użyć zgodnie ze sztuką jakiegokolwiek frameworka oferującego wstrzykiwanie zależności i zarządzanie obiektami poprzez kontener (np. Spring, Guice etc.)**
- Rozszerzeniem tej zasady jest tzw. zasada Hollywood (*don't call us, we'll call you*). W tym przypadku oznacza ona iż moduł niskopoziomowy „czeka cierpliwie" aż moduł wysokopoziomowy poprosi go o wykonanie jakiejś akcji



Interface segregation principle

Interface segregation principle (Zasada segregacji interfejsów)

- Wiele dedykowanych interfejsów jest lepsze niż jeden ogólny.
- Klient nie powinien być zależny od metod, których w ogóle nie używa
- **Klinicznym objawem złamania tej reguły jest tzw. *boski interfejs* zawierający mnóstwo metod nie powiązanych ze sobą logicznie oraz wielu klientów tego interfejsu, z których każdy używa tylko jakiegoś jego wycinka.**



Single responsibility principle

Single responsibility principle (Zasada jednej odpowiedzialności

- Klasa powinna mieć tylko jedną odpowiedzialność (nigdy nie powinien istnieć więcej niż jeden powód do modyfikacji klasy).
- Jeżeli klasa łamie taką zasadę (zwykle jest to proste do zauważenia) to oznacza że ma niską spójność i należy ją podzielić na mniejsze podklasy
- **Zasada ta nie powinna być stosowana „na oślep" – może ona doprowadzić do sytuacji, kiedy nadmierna złożoność systemu spowodowana zbyt wielką ilością klas przeważy nad zyskami wynikającymi z silnej spójności tych klas.**
- Zwykle wystarczy doprowadzić do tego aby klasa nie mieszała różnych poziomów abstrakcji

Open / closed principle

Open / Closed principle (Zasada otwarte / zamknięte)

- Klasy powinny być otwarte na rozszerzenia i zamknięte na modyfikacje.
- Zasada brzmi tylko pozornie sprzecznie. Klasa powinien być tak stworzona, aby dało się rozszerzać jej zachowanie **BEZ** zmiany jej kodu. Oznacza to w praktyce że jak najwięcej funkcjonalności musi być wyciągnięte do:
  - Abstrakcyjnych metod, które implementują klasy podrzędne
  - Zewnętrznych współpracowników, których można wstrzyknąć podmieniając zachowanie
- **Zwykle pisząc klasę od zera nie da się dotrzymać tej zasady, gdyż często nie wiadomo jeszcze gdzie dana klasa w przyszłości będzie wymagała modyfikacji. Dopiero po zdobyciu takiej wiedzy można zrefaktorować kod pod kątem tej reguły.**

Liskov substitution principle

Liskov substitution principle (Zasada podstawienia Liskov) (1)

- Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów
- **Zasada wbrew pozorom jest trudna do poprawnej implementacji. O wiele prościej jest sprawdzić czy jest złamana – typowym objawem klinicznym jest wtedy bezpośrednie sprawdzanie w kodzie czy otrzymana referencja do typu nadrzędnego jest typu podrzędnego**
- Dziedziczenie zgodne z zasadą LSP jest zwykle nietrywialne a czasem wręcz niemożliwe. Z tego względu warto rozważyć zastąpienie dziedziczenia kompozycją (favor composition over inheritance)

Liskov substitution principle (Zasada podstawienia Liskov) – reguły dziedziczenia

- Typ zwracany przez metodę w podklasie musi być taki sam jak w klasie bazowej albo być jego podtypem
- Typy argumentów przekazywane do metody w podklasie muszą być takie same jak w klasie bazowej – albo być ich nadtypem
- Metoda podklasy może rzucać wyjątek tylko wtedy jeśli czyni to metoda w klasie bazowej. Typ wyjątku musi być taki sam jak w klasie bazowej albo musi być jego podtypem
- Warunki wejściowe metody w podklasie nie mogą być silniejsze niż warunki wejściowe w klasie bazowej
- Warunki wyjściowe metody w podklasie nie mogą być słabsze niż warunki wyjściowe w klasie bazowej
- Metoda w podklasie nie może niszczyć niezmienników klasy bazowej

GRASP principles

GRASP principles

Zestaw 9 reguł dotyczących programowania obiektowego oraz przypisywania odpowiedzialności

- Information expert
- Creator
- Controller
- Low coupling
- High cohesion
- Indirection
- Polymorphism
- Pure Fabrication
- Protected Variations

Przedstawione przez Craiga Larmana w 1997 w książce “Applying UML and patterns”

Information expert

Przypisz odpowiedzialność za konkretne działanie do tej klasy, która posiada niezbędne informacje do tego by to działanie wykonać

- Jeżeli żadna klasa nie posiada wszystkich niezbędnych informacji ➔ wybierz tę, która posiada największy podzbiór

Creator

Przypisz odpowiedzialność za tworzenie obiektu X klasie Y, która spełnia co najmniej jeden z poniższych warunków (oczywiście im więcej warunków jest spełnionych tym lepiej):

- **Y zawiera X (poprzez dziedziczenie lub kompozycję)** (w przypadku konfliktu ta opcja wygrywa)
- Y "śledzi" instancje X
- Y "blisko używa" X
- Y posiada wszystkie dane niezbędne do stworzenia X

Wzorzec może być częściowo sprzeczny z wzorcem dependency injection

Controller

Przypisz odpowiedzialność za obsługę zdarzenia pochodzącego z zewnątrz (w tym reakcja na UI) jednej z dwóch klas:

- Klasie reprezentującej system / podsystem (facade controller)
- Klasie reprezentującej dany przypadek użycia (use case controller)

Uwagi dodatkowe:

- Typ kontrolera zależy w dużej mierze od ilości różnych zdarzeń, jakie musimy obsłużyć (przy małej ilości sprawdzi się fasada, przy dużej – przypadki użycia ➔ zwykle powiązane z high cohesion)
- Kontroler powinien delegować wszystko co możliwe do konkretnych klas serwisowych a sam tylko zarządzać całą pracą

Indirection

Stwórz obiekt pośredniczący by pozbyć się zbyt mocnego sprzężenia między obiektami

- Bardzo często realizowany jest tutaj wzorzec mediatora (ale nie zawsze musi to być regułą - można też zastosować adapter)
- Należy wyważyć plusy i minusy stosowania tej reguły – w skrajnym przypadku może ona doprowadzić do wielu dziesiątek klas zawierających jedną metodę oraz bardzo utrudnić czytanie kodu

Polymorphism

Przypisz odpowiedzialność za zachowanie do poszczególnych typów, w których ma się ono zmieniać (przy użyciu metod polimorficznych)

- Innymi słowy testowanie wprost jakiego typu jest dany obiekt ➔ to prawie zawsze błąd projektowy
- Często wspólne zachowanie jest definiowane w klasie abstrakcyjnej, a konkretne klasy pochodne są odpowiedzialne za to co się różni

Pure fabrication

W przypadku problemu ze znalezieniem miejsca na przypisanie jakiejś odpowiedzialności ze względu na wysokie sprzężenie i/lub niską spójność – stwórz nową sztuczną klasę nie reprezentującą żadnego obiektu domenowego i przypisz tę odpowiedzialność tej klasie

- Bardzo często w ten sposób tworzone są tzw. klasy utilowe i / lub serwisy

Protected variation

Przypisz odpowiedzialności do stabilnych interfejsów odpornych na zmiany w przyszłości

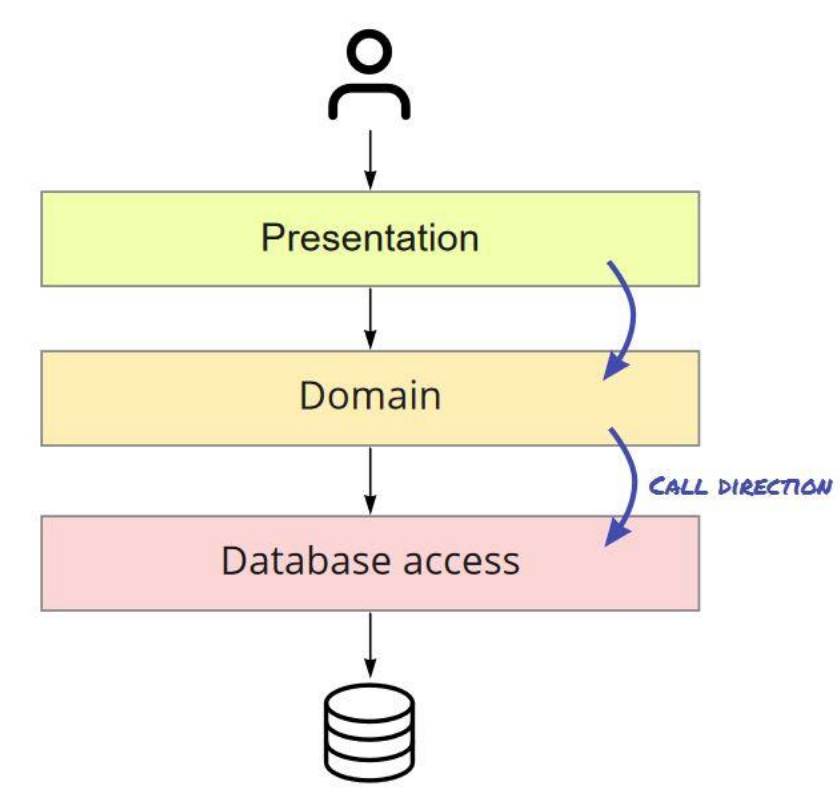
- Zasada ta jest uznawana za najważniejszą ze wszystkich zasad GRASP
- Jedyną stałą rzeczą w systemie jest jego zmienność ➔ należy być na to gotowym
- Zmiany należy przewidywać z wyprzedzeniem ➔ tam gdzie takowa może nastąpić należy jak najszybciej wydzielić stabilny interfejs
- Inne określenie ➔ "znajdź to co się zmienia i zamknij (zenkapsuluj) to"

Czysta architektura

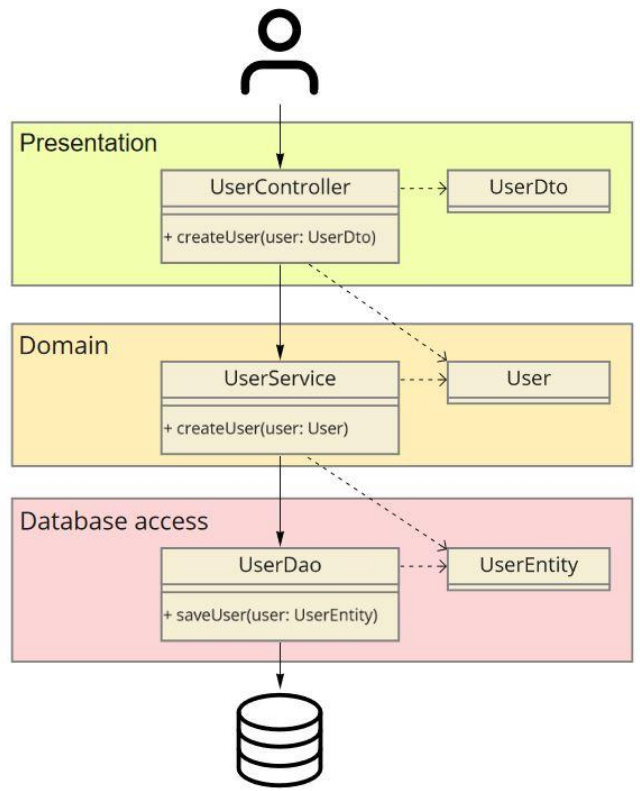
Czym jest architektura / czym zajmuje się architekt ?

- Główne zadanie - uczynić system zamieszkiwalnym (habituable) zarówno dla użytkowników jak i dla developerów
- Realizacja ➔ głównie poprzez wyznaczanie tzw. szerokich kresek (zostawiających sporo wolności)
- Architekt powinien mieć także prawo weta (stosowane oczywiście w skrajnych przypadkach)
- Wieża z kości słoniowej kontra okopy ➔ czyli jak NIE należy tworzyć architektury

Architektura warstwowa (1)



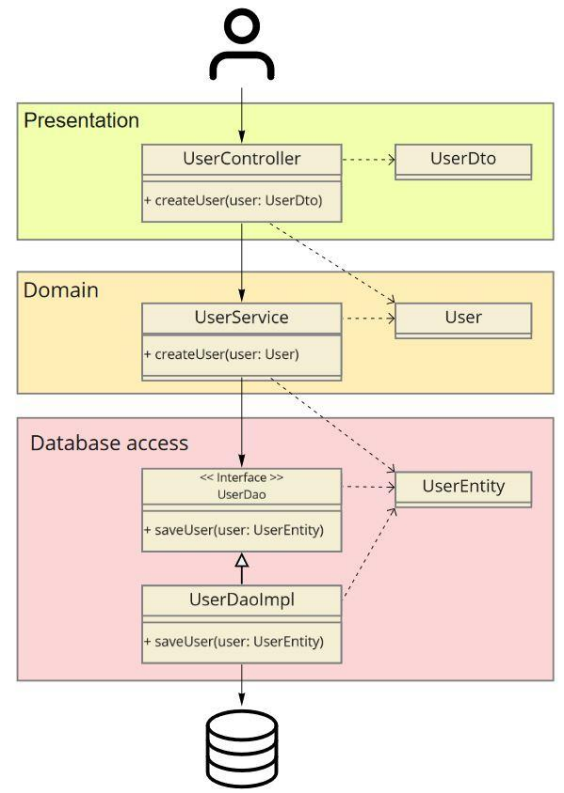
Architektura warstwowa (2)



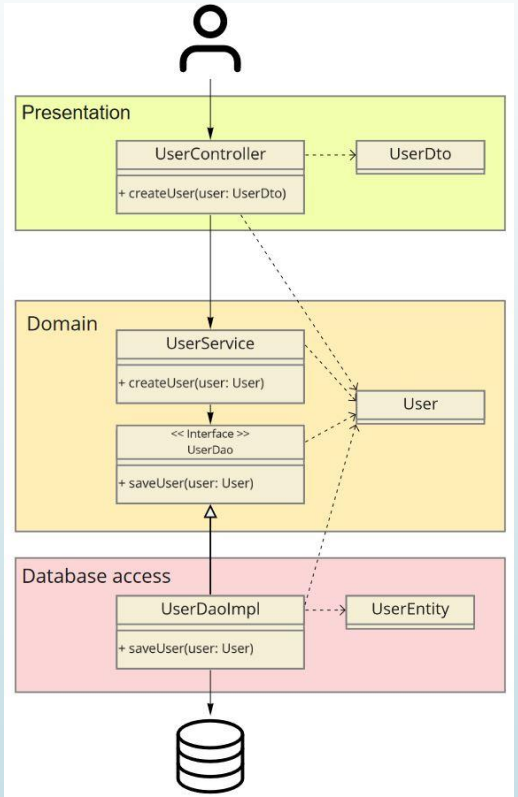
Architektura warstwowa - wady

- Bazocentryczność – baza danych staje się najważniejszą cześcią systemu - od niej zaczynamy design całości
- Warstwa domenowa jest zanieczyszczona kodem związanym typowo z bazami danych (encje, konwertery etc.)
- Ciężko jest podmienić jedną bazę danych na inną, lub wręcz na coś innego (np. NoSQL)

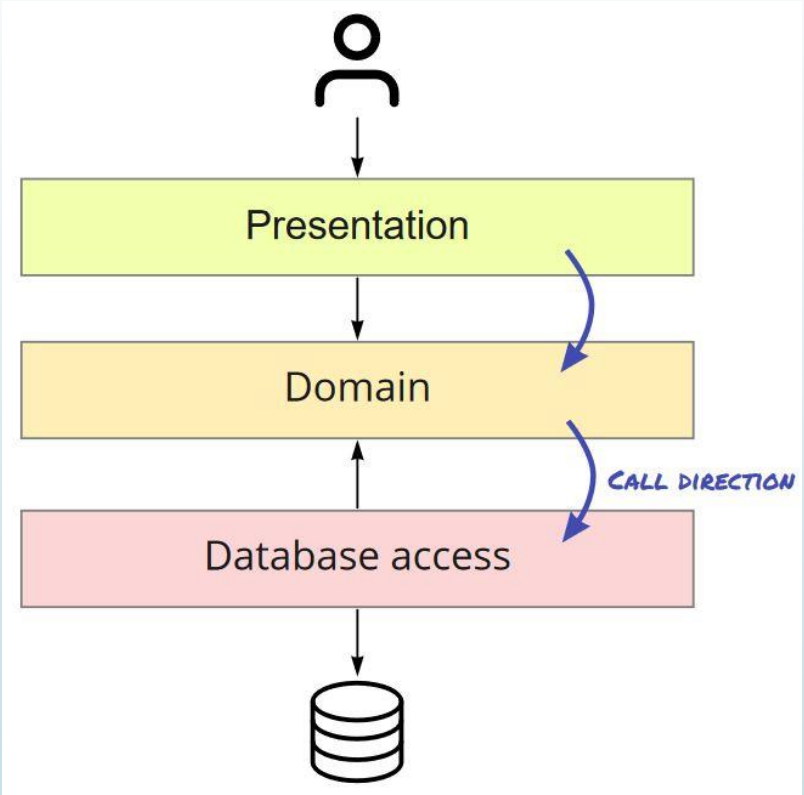
Od architektury warstwowej do cebulowej (1) – ekstrakcja interfejsu



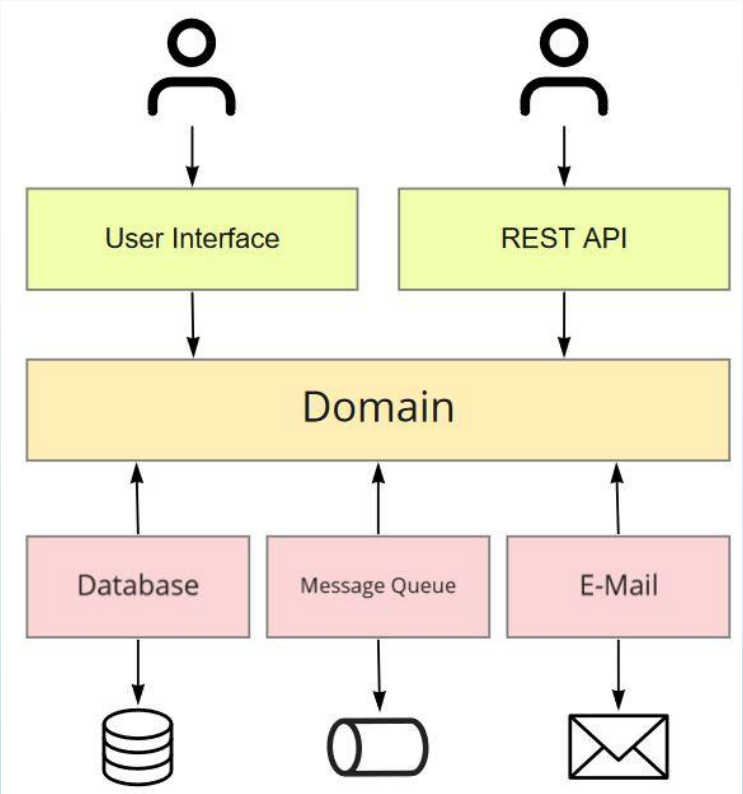
Od architektury warstwowej do cebulowej (2) – model domenowy



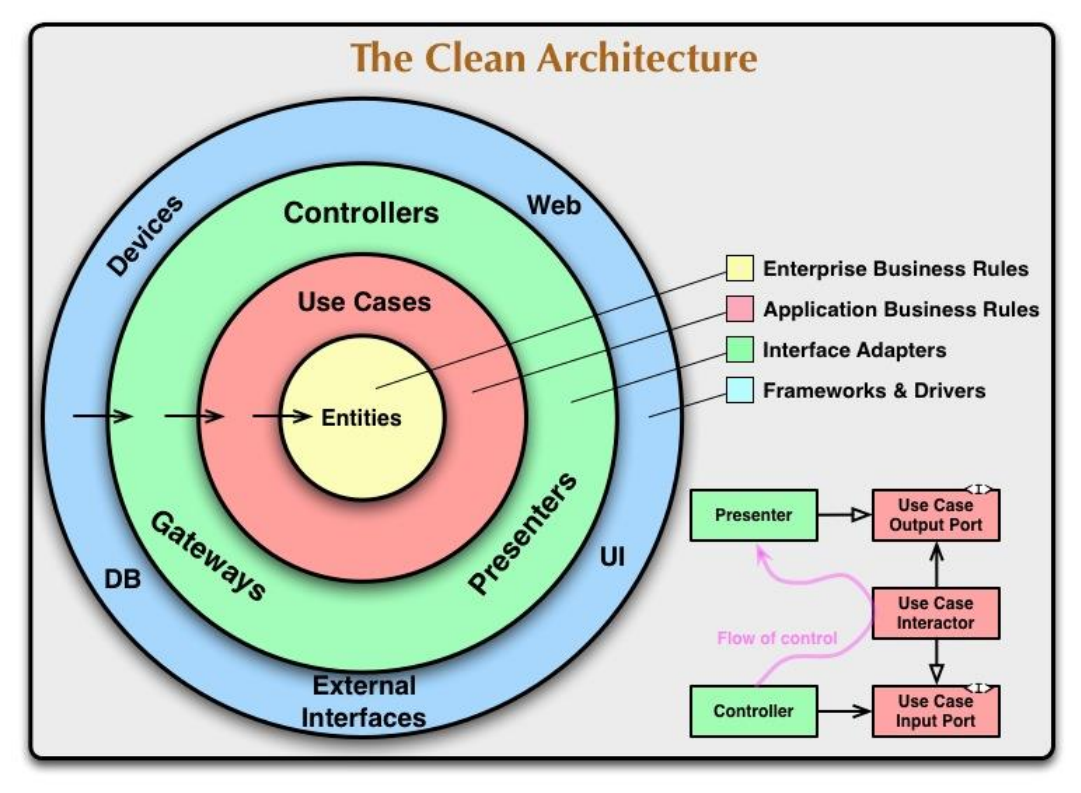
Od architektury warstwowej do cebulowej (3) – odwrócenie zależności



Od architektury warstwowej do cebulowej (4) – adaptery



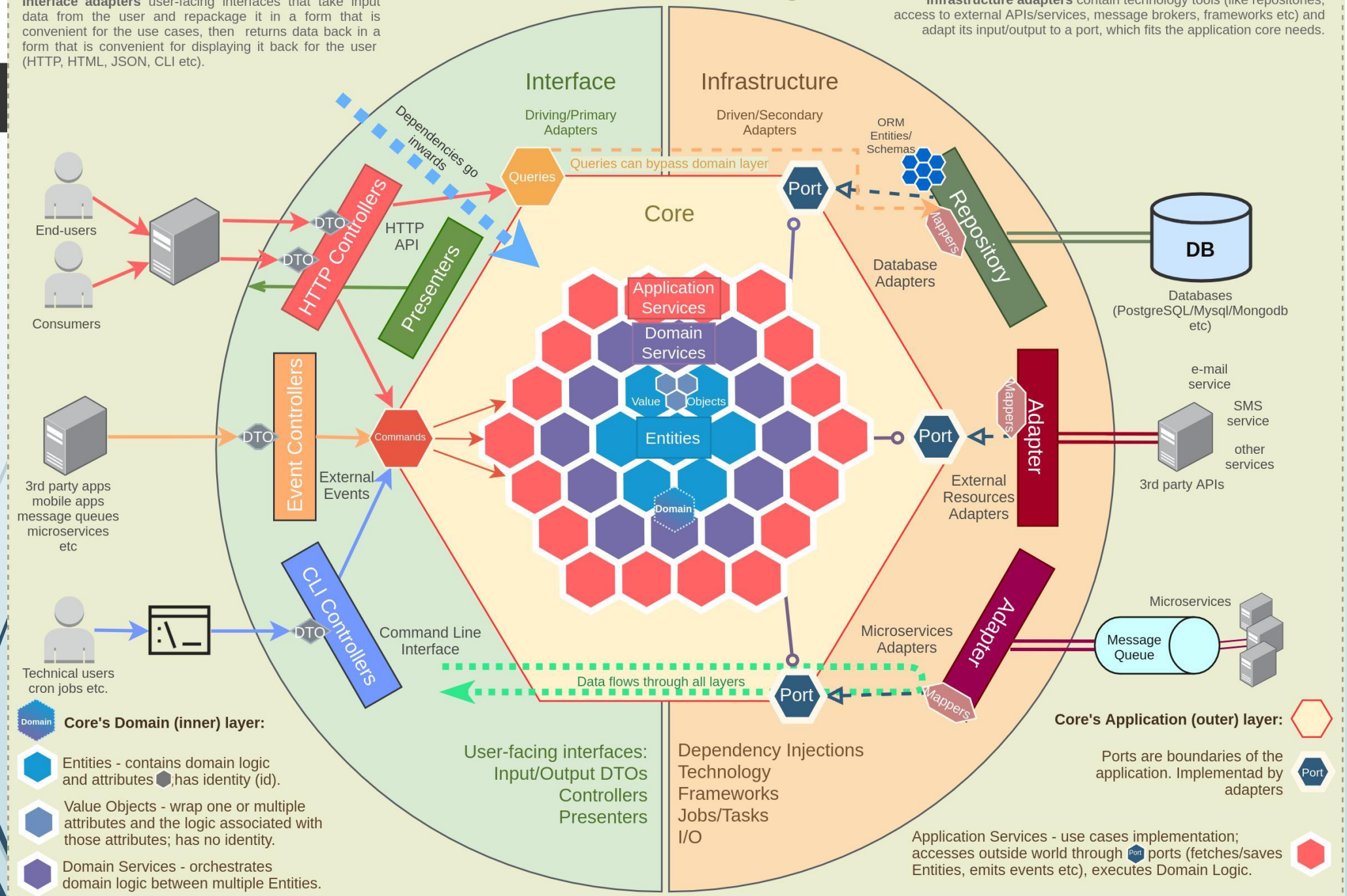
Architektura cebulowa / czysta architektura



Domain-Driven Hexagon

Interface adapters user-facing interfaces that take input data from the user and repackage it in a form that is convenient for the use cases, then returns data back in a form that is convenient for displaying it back for the user (HTTP, HTML, JSON, CLI etc).

Infrastructure adapters contain technology tools (like repositories, access to external APIs/services, message brokers, frameworks etc) and adapt its input/output to a port, which fits the application core needs.



Core's Domain (inner) layer:

Entities - contains domain logic and attributes ⬢ ,has identity (id).

Value Objects - wrap one or multiple attributes and the logic associated with those attributes; has no identity.

Domain Services - orchestrates domain logic between multiple Entities.

Core's Application (outer) layer:

Ports are boundaries of the application. Implementad by adapters

Application Services - use cases implementation; accesses outside world through Port ports (fetches/saves Entities, emits events etc), executes Domain Logic.

Przydatne linki

- https://www.mscharhag.com/architecture/layer-onion-hexagonal-architecture
- https://bulldogjob.pl/news/732-clean-architecture-z-java-11
- https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- https://jeffreypalermo.com/tag/onion-architecture/

Pytania i dyskusja

Dziękuję za uwagę !