

CLEAN CODE & SOLID

AGENDA

- Clean code
- Dług technologiczny
- Refaktoring
- Dobre praktyki projektowe i architektoniczne oprogramowania
 - KISS
 - YAGNI
 - **SOLID**

ROZGRZEWKĄ

- Po co mamy tworzyć Clean Code?
 - Aby szybko i łatwo dało się go przeczytać i zrozumieć!
 - Nawet tak banalna rzecz jak formatowanie ma duży wpływ na czytelność.

Technical debt is an eologistic metaphor referring to the eventual consequences of poor system design, software architecture or software development within a code base. The debt can be thought of as work that needs to be done before a particular job can be considered complete or proper. If the debt is not repaid, then it will keep on accumulating interest, making it hard to implement changes later on. Unaddressed technical debt increases software entropy. As a change is started on a code base, there is often the need to make other coordinated changes at the same time in other parts of the code base or documentation. The other required, but uncompleted changes, are considered debt that must be paid at some point in the future. Just like financial debt, these uncompleted changes incur interest on top of interest, making it cumbersome to build a project. Although the term is used in software development primarily, it can also be applied to other professions.

CLEAN CODE

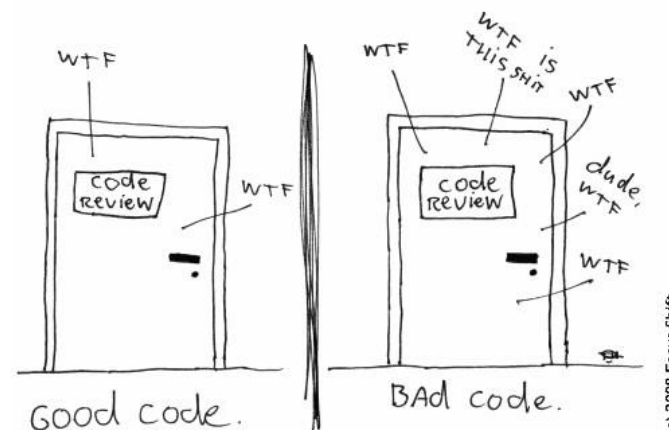
Kod źródłowy jest i będzie podstawowym **BUDULCEM** projektów informatycznych.

Czysty kod jest łatwy do czytania, zrozumienia i rozwijania. Czysty kod powoduje, że rozwijanie oprogramowania jest przewidywalne i zwiększa jakość tworzonego produktu.

Czysty kod to kod:

- Czytelny (*samoopisujący się, zgodny ze wzorcami*)
- Oczywisty dla innych programistów
- Nie zawiera duplikacji
- Nie zawiera zbędnych komentarzy kodu
- Ma odpowiednią strukturę
- Testowalny (i zawiera testy jednostkowe)
- Bardziej niezawodny

The ONLY valid measurement
of code quality: WTFs/minute



(c) 2008 Focus Shift

CZYTELNY KOD

- Programista 90% czasu spędza na czytaniu kodu !
- Dobry kod umożliwia pracę zespołową.
- Każdy z nas chce pracować z czytelnym kodem.

```
double getPrice()
{
    double basePrice = quality * itemPrice;
    double discountFactor;
    if (basePrice > 1000)
        discountFactor = 0.8;
    else
        discountFactor = 1.0;
    double tax;
    if (specialItem)
        tax = 0.08;
    else
        tax = 0.23;
    return basePrice * discountFactor * tax;
}
```

```
double getPrice()
{
    return getItemPrice() * getQuantity()
        * getTax() * getDiscount();
}
```

KOSZTY W PROJEKCIE

- Clean Code ma duży wpływ na koszty projektu! Dlatego obie strony (IT i Biznes) powinny być nim zainteresowane.
- $\text{Koszt projektu} = \text{Koszt wytworzenia} + \text{Koszt utrzymania}$
- $\text{Koszt utrzymania} = \text{Koszt zrozumienia} + \text{Koszt modyfikacji} + \text{Koszt testowania} + \text{Koszt wdrażania}$



PODEJŚCIA DO REALIZACJI PROJEKTU

- Zakładając, że zakres jest stały często manipuluje się kosztami (wielkość zespołu) i czasem, co ma duży wpływ na jakość kodu i przekłada się na jedno z podejść do realizacji projektu:
 - Jakkolwiek, byle szybko i tanio
 -
 - Porządnie, ale w odpowiednim czasie i kosztach



DŁUG TECHNOLOGICZNY

- Analogia do długu finansowego:
 - W krótkim okresie czasu dług może być korzystny – szybki dostęp do dodatkowych środków finansowych
 - Do momentu spłaty naliczane są odsetki
 - Szybsza spłata zmniejsza koszt kredytu
- Trzeba mieć świadomość istnienia długu technologicznego (uświadamiać także managerów 😊) i ciągle go spłacać, gdyż im większy dług tym wyższe koszty wprowadzania zmian, większa awaryjność oprogramowania czas ich usuwania jak również niższe morale zespołu

DŁUG TECHNOLOGICZNY W PROJEKTACH INFORMATYCZNYCH

- Niska jakość kodu źródłowego:
 - Nazewnictwo
 - Nieczytelny kod (także duża ilość komentarzy)
 - „Hardcodowanie”
 - Duplikacje
 - Nieprzestrzeganie „dobrych praktyk”
- Brak testów automatycznych
- Nieodpowiednia architektura
 - boskie klasy („ośmiotysięczniki”)
 - nieużywanie wzorców projektowych



REFACTORING

to zamiana nieczytelnego kodu
w clean code bez zmiany
funkcjonalności

KOD NIE MUSI BYĆ DŁUGI, ABY BYĆ NIECZYTELNY

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
  
    return list1;  
}
```

przykład zapożyczony z książki „Clean Code”, Robert C. Martin, 2008

- Co biznesowo robi poniższa metoda?
- Co zawiera lista theList?
- Czym wyróżnia się pierwszy element w tablicy?
- Dlaczego wartość 4 jest kluczowa?
- Co właściwie jest zwracane z metody?

REFACTORING — WPŁYW ZMIANY NAZW

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<>();  
    for (int[] cell : gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

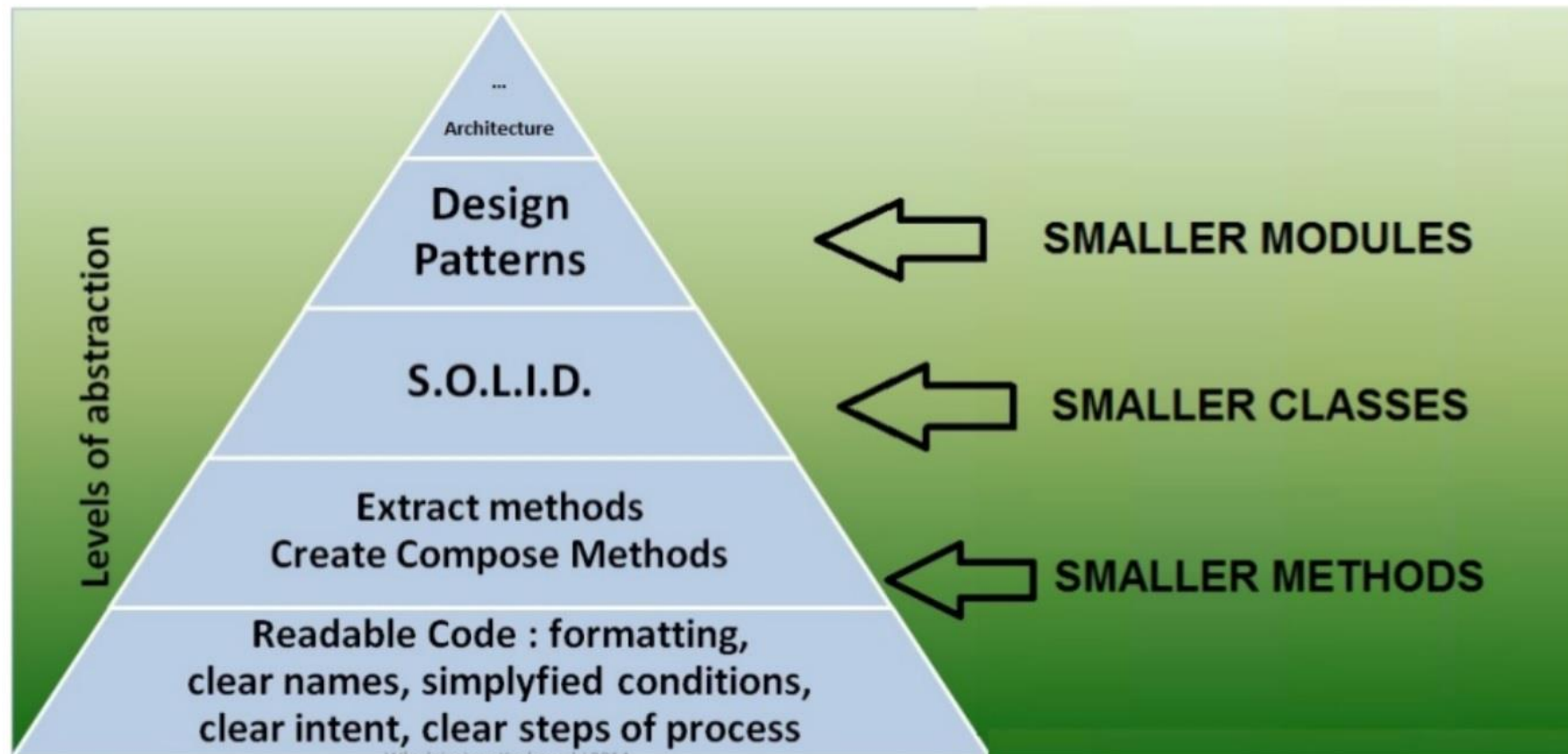
REFACTORING — JAWNE NAZYWANIE STRUKTUR RÓWNIEŻ ZWIĘKSZA CZYTELNOŚĆ

```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new ArrayList<>();  
    for (Cell cell : gameBoard)  
        if (cell.isFlagged())  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

REFACTORING — PODEJŚCIE FUNKCYJNE

```
public List<Cell> getFlaggedCells() {  
    return gameBoard.stream()  
        .filter(Cell::isFlagged)  
        .collect(Collectors.toList());  
}
```

Pyramid of Refactoring



ZASADY CLEAN CODE

NAZEWNICTWO

- Nazwy oddające znaczenie (najlepiej biznesowe) - jeżeli potrzebny komentarz to nazwa nie jest najlepsza
- Jedna spójna nazwa w całym kodzie (łatwiej znaleźć konkretną nazwę)
- Długa czytelna nazwa jest lepsza od krótkiej enigmatycznej
- Opisowa długa nazwa jest lepsza od opisowego długiego komentarza
- `int s; //elapsed time in seconds` ☹️
- `int seconds;`
- `int duration;`
- `int durationInSeconds;` 😊

NAZWY KLAS - RZECZOWNIKI

- Rzeczownik lub grupa rzeczowników. Stosuj tylko łatwo wyszukiwane nazwy.
- Pojedyncza odpowiedzialność.
- Przykłady:
 - Employee
 - WavPlayer
 - PostalAddressDecoder
- Do unikania:
 - Manager
 - Processor
 - Data
 - Info

NAZWY METOD - CZASOWNIKI

- Akcesory i mutatory zgodnie z konwencją **java bean**:
 - setDataSource
 - getName
 - isNegative
- Czasownik lub fraza z czasownikiem
 - save
 - removePage
 - sendEmail

CZYTELNOŚĆ METOD — PODSTAWOWE ZASADY

- Możliwie mała
- Realizowanie tylko jednej rzeczy
- Dobra nazwa wyjaśniająca tę jedną funkcjonalność
- Wyrażenia warunkowe przeniesione do metod
- Krótkie bloki w if, else, for
- Bez dalszych zagnieżdżeń w blokach kodu
- Usuwanie nieużywanych fragmentów kodu

```
public static String renderPageWithSetupsAndTeardowns(PageData pageData, boolean isSuite) throws Exception {  
    if (isTestPage(pageData)) {  
        includeSetupAndTeardownPages(pageData, isSuite);  
    }  
    return pageData.getHtml();  
}
```

ARGUMENTY METOD

- Funkcje bezargumentowe są świetne: `record.remove()`;
- Funkcje z jednym argumentem są bardzo dobre: `file.save(„output”)`;
- Funkcje z dwoma argumentami są dobre: `file.open(„input”, „r”)`;
- Funkcje z trzema są akceptowalne: `assertEquals(message, expected, actual)`;
- Funkcje z więcej niż trzema argumentami są wstrętne i należy ich unikać.
 - Konieczność dodatkowej analizy wywołania
 - Utrudnione pisanie testów – konieczność pokrycia wszystkich kombinacji
 - Często lepiej przekazać obiekt
- Funkcje z argumentami flagowymi powinny być unikane. Łamią zasadę SPR.
 - Metoda powinna robić to, o czym mówi jej nazwa. Zawsze można je rozbić:
`printFile(true) = printFileWithHeader() + printFileWithoutHeader()`

KLAUZULE DOZOROWANE VS. SINGLE RETURN POINT

UWAGA – Single Return Point jest bardzo starą zasadą odnoszącą się do języków typu assembler, COBOL, etc. W JAVIE ZASADY TEJ NIE STOSUJEMY!

Zamiast tego stosujemy tzw. klauzule dozorowane (Guard clause) czyli zamiast zagnieżdżać się coraz głębiej w ify, jeśli to możliwe (szczególnie kiedy else jest pusty) powinno się szybciej opuszczać metodę.

„ZŁY” PRZYKŁAD (Single return point, brak klauzul dozorowanych):

```
public void processUserParameters(Status status, Option option) {  
    if (status != Status.DO_NOTHING) {  
        doSth(status);  
        if (option != null) {  
            doSthElse(option, status, Session.getUser());  
        }  
    }  
    return;  
}
```

DOBRY PRZYKŁAD (zastosowane klauzule dozorowane):

```
public void processUserParameters(Status status, Option option) {  
    if (status == Status.DO_NOTHING) return;  
    doSth(status);  
    if (option == null) return;  
    doSthElse(option, status, Session.getUser());  
}
```

SEPARACJA LOGIKI OD OBSŁUGI BŁĘDÓW

Połączona logika operacji i obsługa błędów:

```
if (deleteRegistryValue(value) == STATUS_OK) {
    if (deleteRegistryKey(subkey) == STATUS_OK) {
        if (deleteRegistryKey(key) == STATUS_OK) {
            log.info("Keys and value deleted from registry");
        } else {
            log.error("Unable to delete key from registry");
        }
    } else {
        log.error("Unable to delete subkey from registry");
    }
} else {
    log.error("Unable to delete value from registry");
}
```

Krótszy i bardziej przejrzysty zapis bez mieszania logiki z obsługą błędów:

```
try {
    deleteRegistryValue(value);
    deleteRegistryKey(subkey);
    deleteRegistryKey(key);
} catch (RegistryException e) {
    log.error(e, e);
}
```

NIGDY NIE ZWRACAJ NULL I ZABEZPIECZAJ KOD PRZED NULLAMI

- **Lepiej zwrócić pustą listę, Null Object lub Optional (Java 8)** - zwracanie null'i powoduje dużo błędów w oprogramowaniu i ma duży wpływ na zrozumiałość kodu
- Stosuj `Objects.nonNull(foo);`

```
public void setFoo(Foo foo){  
    Objects.nonNull(foo);  
    foo.print();  
}
```


OBSŁUGA BŁĘDÓW — PODSTAWOWA ZASADA

Co jest nie tak z tym kodem?

```
public void deleteRegistryValueWithKeys() {  
    try {  
        internalDeleteRegistryValueWithKeys();  
    } catch (RegistryException e) {  
        //should not happen  
    }  
}
```

OBSŁUGA BŁĘDÓW — PODSTAWOWA ZASADA

```
public void deleteRegistryValueWithKeys() {  
    try {  
        internalDeleteRegistryValueWithKeys();  
    } catch (RegistryException e) {  
        //should not happen  
    }  
}
```

- Brak informacji w logach, że coś poszło źle
- Znacznie utrudnione diagnozowanie błędów w działaniu
- Trzeba przerzucić albo zalogować

OBSŁUGA BŁĘDÓW — PODSTAWOWA ZASADA

Co jest nie tak z tym kodem?

```
public void deleteRegistryValueWithKeys() {  
    try {  
        internalDeleteRegistryValueWithKeys();  
    } catch (RegistryException e) {  
        log.error("Problem with deleting registry key");  
    }  
}
```

OBSŁUGA BŁĘDÓW — PODSTAWOWA ZASADA

```
public void deleteRegistryValueWithKeys() {  
    try {  
        internalDeleteRegistryValueWithKeys();  
    } catch(RegistryException e) {  
        log.error("Problem with deleting registry key", e);  
    }  
}
```

- Wyjątek jako drugi argument, aby zalogować również stos

KOMENTARZE

Źle:

```
//Check if payment can be moved to archive  
if ((payment.isPaid() &&  
    payment.payDate() + 30 < currentDate &&  
    !payment.isArchiveForbidden()))
```

```
// Method send message to file  
public void print(String s) {...}
```

Dobrze:

```
if (payment.canBeMovedToArchive())
```

```
public void sendMessageToFile(String message) {...}
```

„Don't comment bad code – rewrite it”

B. Kernighan i P. Plaugher

KOMENTARZE

Ale czasami się przydają:

- Wyjaśnienie specyficznego zachowania

// format matched kk:mm:ss EEE, MMM dd, yyyy

```
Pattern timeMatcher = Pattern.compile(„\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*”);
```

- //TODO – w sytuacjach, gdy w danej chwili nie można czegoś poprawić (ale z umiarem! i trzeba cyklicznie przeglądać i poprawiać)

DOBRE PRAKTYKI PROJEKTOWE I ARCHITEKTONICZNE OPROGRAMOWANIA

PRZYDATNE ZASADY PROJEKTOWE

- **YAGNI** (You aren't gonna need it) - Implementacja tylko rzeczy które są niezbędne
- **KISS** (Keep it simple stupid) - Prostota projektu i realizacji
- **DRY** (Don't repeat yourself) - Unikaj duplikacji kodu
- **SOLID**

SOLID

5 zasad architektury systemów zaproponowanych przez Roberta. C. Martina, Uncle Boba.

S - Single Responsibility Principle

O - Open-Closed Principle

L - Liskov Substitution Principle

I - Interface Segregation Principle

D - Dependency Inversion Principle

Wg Uncle Boba – program jest dobrze napisany jeśli łatwo daje się go zmieniać!



Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

SPR - ZASADA POJEDYNCZEJ ODPOWIEDZIALNOŚCI

- Klasa powinna zmieniać się tylko z jednego powodu czyli powinna być odpowiedzialna tylko za jedną funkcjonalność
- Rezultat: Klasy będą mniejsze i ich cel jasno zdefiniowany, co poprawi architekturę systemu i ułatwi jego zmianę w przyszłości. Zapobiega powstawaniu Boskich Klas (https://en.wikipedia.org/wiki/God_object)
- Ile odpowiedzialności ma ta klasa?

```
class Employee {  
    public Pay calculatePay() {...}  
    public void sendMessage(String type, String message) {...}  
    public String getReport(int format) {...}  
    public void save() {...}  
}
```



SPR - ZASADA POJEDYNCZEJ ODPOWIEDZIALNOŚCI

Zły przykład – funkcja robi dwie rzeczy – dwa poziomy abstrakcji:

```
public List<ResultDto> buildResult(Set<ResultEntity> resultSet) {  
    List<ResultDto> result = new ArrayList<>();  
    for (ResultEntity entity : resultSet) {  
        ResultDto dto = new ResultDto();  
        dto.setShoeSize(entity.getShoeSize());  
        dto.setNumberOfEarthWorms(entity.getNumberOfEarthWorms());  
        dto.setAge(computeAge(entity.getBirthDay()));  
        result.add(dto);  
    }  
    return result;  
}
```

Dobry przykład:

```
public List<ResultDto> buildResult(Set<ResultEntity> resultSet) {  
    List<ResultDto> result = new ArrayList<>();  
    for (ResultEntity entity : resultSet) {  
        result.add(toDto(entity));  
    }  
    return result;  
}  
  
private ResultDto toDto(ResultEntity) {  
    ResultDto dto = new ResultDto();  
    dto.setShoeSize(entity.getShoeSize());  
    dto.setNumberOfEarthWorms(entity.getNumberOfEarthWorms());  
    dto.setAge(computeAge(entity.getBirthDay()));  
    return dto;  
}
```



Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.

OCP - ZASADA OTWARTE-ZAMKNIĘTE

- Klasa powinna być otwarta na rozszerzenie i zamknięta na modyfikacje.
- Oznacza to, że nowe funkcjonalności powinny być realizowane przez dziedziczenie (kompozycję, dekorację etc.) a nie przez zmianę już istniejących klas (oczywiście za wyłączeniem bug fixingu 😊)
- Rezultat:
 - Trudniej zepsuć coś, co już działało
 - Powstaje obiektowa architektura kodu

OCP - ZASADA OTWARTE-ZAMKNIĘTE

Źle

```
class GraphicEditor {  
    public void drawShape(Shape shape) {  
        if (shape.isCircle()) { ... }  
        else if (shape.isRectangle()) { ... }  
        ...  
    }  
}
```

Dobrze

```
class GraphicEditor {  
    public void drawShape(Shape shape) {  
        shape.draw();  
    }  
}  
  
interface Shape {  
    void draw();  
}  
  
class Circle implements Shape {  
    void draw() { ... }  
}
```




Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

LSP - ZASADA PODSTAWIENIA LISKOV

Barbara Liskov: *"Kod który używa klasy typu A musi być w stanie użyć podklasy A bez wiedzy o tym"*.

Wszystkie podklasy muszą działać w ten sam sposób jak klasa bazowa. Oczywiście specyficzne zachowania w podklasie mogą być inne niż w klasie bazowej ale muszą być zgodne z oczekiwanym zachowaniem w klasie bazowej. Zasada zapewnia, że podklasa spełnia kontrakt zdefiniowany przez klasę rodzica.

```
public class Rectangle {
    protected double a;
    protected double b;
    public void setA(double a) {
        this.a = a;
    }
    public void setB(double b) {
        this.b = b;
    }
    public double getArea() {
        return a * b;
    }
}
```

```
public class Square extends Rectangle {

    @Override
    public void setA(double a) {
        this.b = a;
        this.a = a;
    }

    // Co z setB() ?
}
```

```
public static void main(String[] args) {
    Rectangle rectangle = new Rectangle();
    rectangle.setA(5.0);
    rectangle.setB(10.0);
    System.out.println(rectangle.getArea());

    Rectangle square = new Square();
    square.setA(5.0);
    square.setB(10.0); // Przecież wolno! ☹
    System.out.println(square.getArea());
}
```



Interface Segregation Principle
You want me to plug this in *where?*

ISP - ZASADA SEGREGACJI INTERFEJSÓW

- Klasa musi być zależna tylko od metod których używa
- Dzielenie dużych interfejsów na mniejsze

ŹŁE:

```
public interface Messenger {  
    askForCard();  
    tellInvalidCard();  
    askForPin();  
    tellInvalidPin();  
    tellCardWasSiezed();  
    askForAccount();  
    tellNotEnoughMoneyInAccount();  
    tellAmountDeposited(); tellBalance();  
}
```

Dobrze:

```
public interface LoginMessenger {  
    askForCard();  
    tellInvalidCard();  
    askForPin();  
    tellInvalidPin();  
}  
  
public interface WithdrawalMessenger {  
    tellNotEnoughMoneyInAccount();  
    askForFeeConfirmation();  
}  
  
public class EnglishMessenger implements LoginMessenger,  
WithdrawalMessenger { ... }
```



Dependency Inversion Principle

Would you solder a lamp directly to the electrical wiring in a wall?

DIP - ZASADA ODWRÓCENIA ZALEŻNOŚCI

- Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych – zależności między nimi powinny wynikać z abstrakcji. Czyli innym słowy w klasach i metodach nie powinniśmy używać żadnych nazw konkretnych klas, mogą nazwy tylko interfejsów i klas abstrakcyjnych.
- Również klasy nie powinny dziedziczyć po konkretnych klasach, jedynie po klasach abstrakcyjnych i interfejsach.

ŹLE:

```
class CharCopier {  
    void copy() throws IOException {  
        InputStream reader = System.in;  
        OutputStream writer = System.out;  
        int c;  
        while ((c = reader.read()) != -1) {  
            writer.write(c);  
        }  
    }  
}
```

DOBRE:

```
public interface Reader { char getChar(); }  
public interface Writer { void putchar(char c)}  
  
class CharCopier {  
    void copy(Reader reader, Writer writer) {  
        int c;  
        while ((c = reader.getChar()) != EOF) {  
            writer.putchar(c);  
        }  
    }  
}  
  
public Keyboard implements Reader {...}  
public Printer implements Writer {...}
```

- Inny przykład: Klasa realizująca powiadomienia nie powinna obsługiwać emaili, sms-ów itd – powinna otrzymać obiekt implementujący interfejs z metodą send, a obiekt powinien zawierać logikę realizacji powiadomienia.

PODSUMOWANIE - BAD CODE SMELLS

- **Long Method** – każda metoda dłuższa niż 10 linii powinna budzić niepokój
- **Large Class** - klasa zawierająca dużo pól, metod, linii kodu
- **Primitive Obsession** – używanie prymitywnych typów zamiast małych obiektów (takich jak waluta, zakres, specjalne ciągi dla numerów telefonów, etc.) i używanie stałych do kodowania informacji (np.: `USER_ADMIN_ROLE = 1`)
- **Long Parameter List** – dłuższe niż 4 parametry
- **Duplicated Code**
- **Comments**
- Dużo **if**, **instance of** lub **switch** statements – [Object-Orientation Abusers]
- **Dead code** lub **kod zostawiony do użycia w przyszłości** (reguła YAGNI – You Ain't Gonna Need It)
- Duża ilość zmian kiedy chcesz zrobić małą zmianę (np. kiedy chcesz dodać nowy produkt musisz zmienić metody do wyszukiwania, wyświetlania) [Divergent Change]
- **Magic numbers**