



... .. Czyli profesjonalizm w programowaniu

Czysty kod

Nauka „pisarstwa” (w języku naturalnym)



- Najpierw uczymy się **czytać**
 - Krótkie teksty **wzorcowe**
 - **Dobłą** literaturę w oryginale
 - Moby-Dick, Robinson Crusoe, Pride and Prejudice, War and Peace, The War of the Worlds, The Lord of the Rings
 - Artykuły (naukowe, dziennikarskie)
- Potem uczymy się samodzielnie pisać
 - Krótkie teksty wzorcowe (listy, życiorysy, sprawozdania, podania)
 - ...

Nauka „pisarstwa” zwanego programowaniem



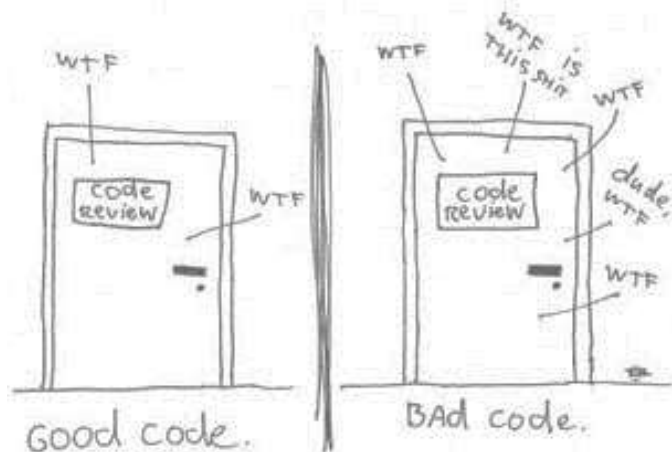
- Najpierw uczymy się **pisać** (sic!)
- Ktoś słyszał kiedyś o kursie (albo książce): „Czytanie programów w Java”?

Nauka „pisarstwa” zwanego programowaniem



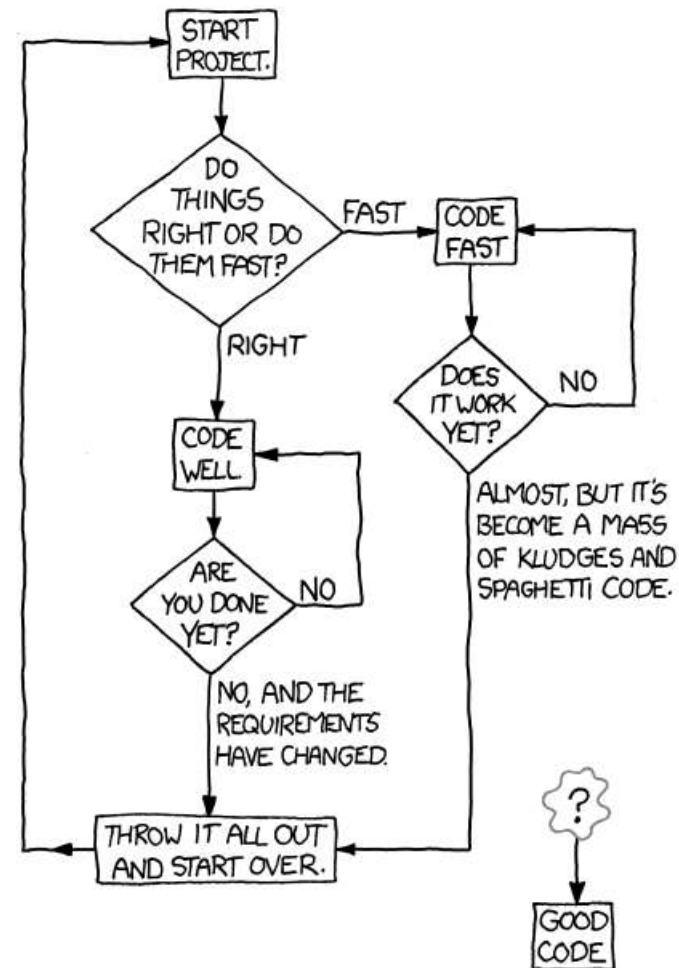
- Ale żeby uczyć czytać, trzeba znaleźć przykłady dobrego (i złego) kodu
 - Tylko jak wygląda dobry i zły kod?

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>

HOW TO WRITE GOOD CODE:



Przykłady dobrego kodu, gdzie szukać?



“Good” Java code examples? [closed]



Can anyone point out some java code which is considered "good"?

26



14

I have started programming recently, about two years ago. I mostly program using java. I write bad code. I think the reason behind this, is that I have never actually seen "good" code. I have read a couple of books on programming, but all of them just have some toy examples which merely explain the concept. But this is not helpful in complex situations. I have also read books/ articles / [SO questions](#) on what is "good" code, but none of them has a complex enough example.

So, can anyone point me to some java code which is considered "good"? (I know that my coding skills will improve as I practice, but perhaps looking at some examples will help me.)

The best option for you to study good code is to look at some popular open source projects. I think 2 years is good enough time to understand code in these projects. Some of the projects you could look at:

- [openjdk](#)
- [apache tomcat](#)
- [spring framework](#)
- [apache commons](#) (very useful)
- [Google collections](#)

Enough for you study and understand a variety of concepts. I frequently study code in JDK catalina(tomcat) and spring, jboss, etc.

To me, one of the best books about the subject is [Clean Code](#) by Robert C. Martin.

Przykłady dobrego kodu, gdzie szukać?



“Good” Java code examples? [closed]



Can anyone point out some java code which is considered "good"?

26



14

I have started programming recently, about two years ago. I mostly program using java. I write bad code. I think the reason behind this, is that I have never actually seen "good" code. I have read a couple of books on programming, but all of them just have some toy examples which merely explain the concept. But this is not helpful in complex situations. I have also read books/ articles / [SO questions](#) on what is "good" code, but none of them has a complex enough example.

So, can anyone point me to some java code which is considered "good"? (I know that my coding skills will improve as I practice, but perhaps looking at some examples will help me.)

The best option for you to study good code is to look at some popular open source projects. I think **2 years** is good enough time to understand code in these projects. Some of the projects you could look at:

- [openjdk](#)
- [apache tomcat](#)
- [spring framework](#)
- [apache commons](#) (very useful)
- [Google collections](#)

Enough for you study and understand a variety of concepts. I frequently study code in JDK catalina(tomcat) and spring, jboss, etc.

To me, one of the best books about the subject is [Clean Code](#) by Robert C. Martin.

Percepcja jakości

- Umiemy rozpoznawać jakość



Ecce Homo de Elías García Martínez.



To może czysty kod oznacza:



Debugging is twice as hard as writing
the code in the first place.
Therefore, if you write the code as
cleverly as possible, you are, by
definition, not smart enough to
debug it.

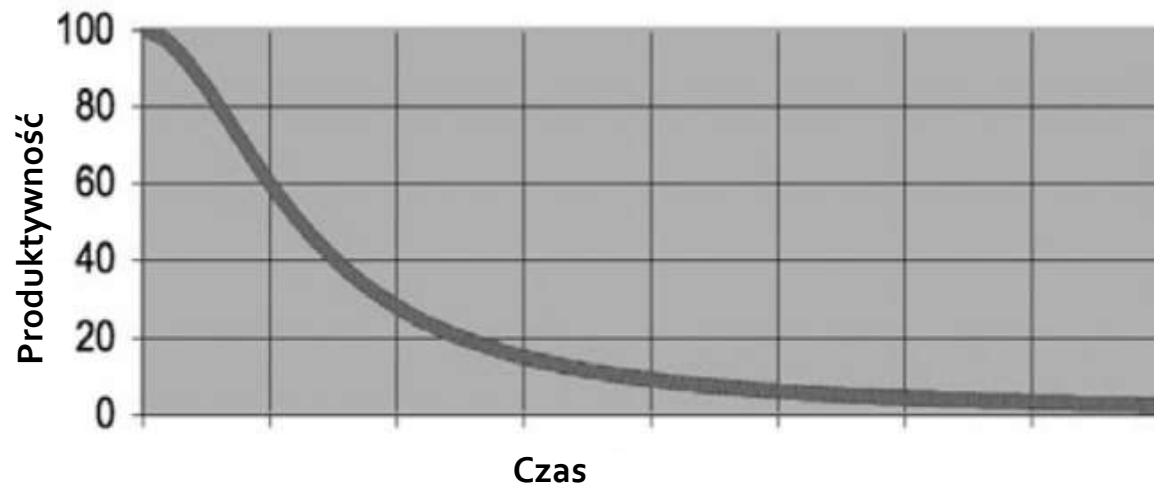
— *Brian Kernighan* —

AZ QUOTES

Dług techniczny (ang. technical debt)



- Dług techniczny to miara poziomu trudności jaki napotykają programiści podejmując próbę wprowadzenia wymaganych zmian do istniejącego kodu. Jeżeli dług wzrasta to rosną również:
 - Wartości oszacowań, nieprzewidywalność, liczba efektów ubocznych, poziom lęku



Dług techniczny bardziej wymiennie



- Jakość kodu
 - Złożoność
 - Np.: Złożoność cyklomatyczna
 - Powtórzenia
 - DRY
 - Naruszanie reguł
 - Pokrycie testem
 - Dokumentacja

Kiedy się zadłużamy technicznie?



- Obniżając jakość kodu, powody:
 - Brak wiedzy
 - Brak profesjonalizmu
 - Pośpiech związany z deadline'm
 - Entropia

Wymiary jakości kodu

- Czytelność

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

- Zarządzalność
(zdadność do
pielęgnacji)

- Wydajność

```
for (Person p : persons) {  
    s += ", " + p.getName();  
}  
s = s.substring(2);
```

```
StringBuilder sb = new StringBuilder(persons.size() * 16);  
for (Person p : persons) {  
    if (sb.length() > 0) sb.append(", ");  
    sb.append(p.getName());  
}
```

```
public class a  
{  
  
    public a(String s, String s1, String s2)  
    {  
        c = s;  
        b = s1;  
        a = s2;  
    }  
  
    public String a()  
    {  
        return c;  
    }  
  
    public String b()  
    {  
        return b;  
    }  
  
    public String toString()  
    {  
        return "Name: " + c + ", Email: " + b + ", Phone: " + a;  
    }  
  
    private String c;  
    private String b;  
    private String a;  
}
```

Jaki powinien być czysty kod?



1. Elegancki i efektywny
2. Czyta się go jak dobrą prozę (!)
3. Łatwy w ulepszaniu przez innych
4. Cechuje go dbałość o szczegóły
5. Nie zawiera powtórzeń
6. Jest konsekwentny, prosty, urzekający

Czysty kod - porady



1. Nazewnictwo
2. Konstrukcja funkcji
3. Komentarze
4. Klasy i obiekty
 - Zasady SOLID

Nazewnictwo

■ Nazwy wyjaśniające intencje

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

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

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

```
String[] l1 = getArr(str1);
```

```
String[] fieldValues = parseCsvRow(csvRow);
```

Nazewnictwo

- Nazwy jednoznaczne, wymowne

```
class DtaRcrd102 {  
    private Date genymdhms;  
    private Date modymdhms;  
    private final String pszqint = "102";  
    /* ... */  
};
```



```
class Customer {  
    private Date generationTimestamp;  
    private Date modificationTimestamp;  
    private final String recordId = "102";  
    /* ... */  
};
```

```
for (int j=0; j<34; j++) {  
    s += (t[j]*4)/5;  
}
```



```
int realDaysPerIdealDay = 4;  
const int WORK_DAYS_PER_WEEK = 5;  
int sum = 0;  
for (int j=0; j < NUMBER_OF_TASKS; j++) {  
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;  
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);  
    sum += realTaskWeeks;  
}
```

Komentarze



- Wstawiamy komentarz ponieważ:
 - Chcemy opisać bałagan w kodzie
 - Chcemy wyjaśnić intencje
 - Chcemy coś doprecyzować, podkreślić
 - Bo przecież trzeba komentować

Komentarze – The good, the bad and the ugly



```
/**
 * Returns TRUE if now is the work day.
 * @return boolean
 */
public static boolean isWorkDay() {

    * @param parameters AppLoginRequest
    * @param httpRequest HttpServletRequest
    * @return AppLoginResponse
    */
    AppLoginResponse login(AppLoginRequest parameters, HttpServletRequest httpRequest);

/**
 * Returns next day.
 * @return Date
 */
public static Date getNextDay() {

String listItemContent = match.group(3).trim();
// the trim is real important. It removes the starting
// spaces that could cause the item to be recognized
// as another list.

// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();

// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
    "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");

// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))

    if (employee.isEligibleForFullBenefits())
```

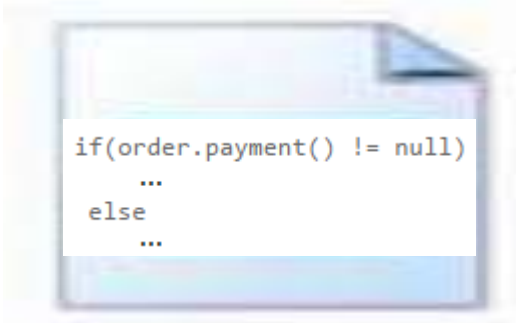

Nie powtarzaj się!



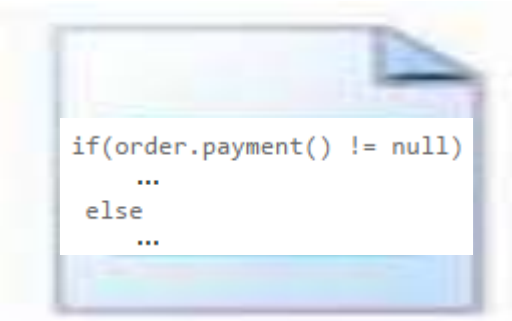
- Zasada DRY (ang. Don't Repeat Yourself)
 - Wielokrotnie wykorzystuj kod zamiast go powielać
 - Wybieraj odpowiednią lokalizację dla kodu
 - „single source of truth”/“Every piece of knowledge must have a single, unambiguous authoritative representation within a system” – pojedyncza zmiana, pojedynczy test
 - Stosuj zasady poprawnego nazewnictwa

Naruszanie DRY - przykłady

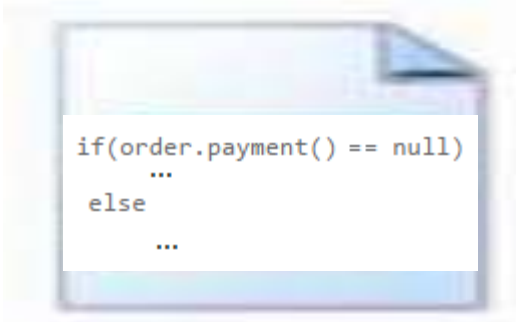
```
class Order {  
    Date payment = null;  
    Date payment() { return this.payment; }  
}
```



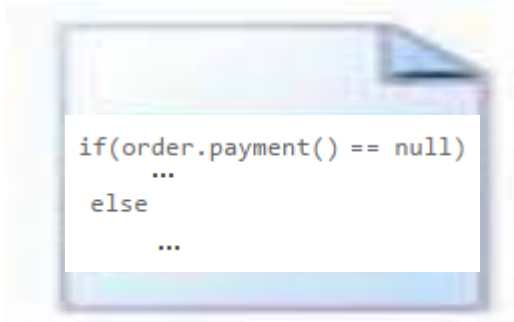
```
if(order.payment() != null)  
    ...  
else  
    ...
```



```
if(order.payment() != null)  
    ...  
else  
    ...
```



```
if(order.payment() == null)  
    ...  
else  
    ...
```



```
if(order.payment() == null)  
    ...  
else  
    ...
```

Naruszanie DRY - przykłady

```
class Order {  
    Date payment = null;  
    Date payment() { return this.payment; }  
}
```



```
class Order {  
    Date payment = null;  
    Date payment() { return this.payment; }  
    boolean isPaid { return this.payment != null; }  
}
```

```
if (order.isPaid())  
    ...  
else  
    ....
```

```
if (order.isPaid())  
    ...  
else  
    ....
```

```
if (order.isPaid())  
    ...  
else  
    ....
```

```
if (order.isPaid())  
    ...  
else  
    ....
```

Naruszanie DRY - przykłady



```
for (int i = 0; frames.size() > i; i++) {
    AnimationFrame frame = frames.get(i);
    if (i == 0) {
        fadeIn = (frame.getFadeIn() == -1 ? 0 : frame.getFadeIn());
        stay += (frame.getStay() == -1 ? 0 : frame.getStay());
        stay += (frame.getFadeOut() == -1 ? 0 : frame.getFadeOut());
    } else if (i + 1 == frames.size()) {
        stay += (frame.getFadeIn() == -1 ? 0 : frame.getFadeIn());
        stay += (frame.getStay() == -1 ? 0 : frame.getStay());
        fadeOut = (frame.getFadeOut() == -1 ? 0 : frame.getFadeOut());
    } else {
        stay += (frame.getFadeIn() == -1 ? 0 : frame.getFadeIn());
        stay += (frame.getStay() == -1 ? 0 : frame.getStay());
        stay += (frame.getFadeOut() == -1 ? 0 : frame.getFadeOut());
    }
    totalTime += frame.getTotalTime();
}
```

Funkcje i poziomy abstrakcji



- SLA(P) (ang. Single Layer of Abstraction Principle)
 - Wszystkie instrukcje w ramach funkcji/metody powinny działać na tym samym poziomie abstrakcji i być związane z pojedynczym zadaniem (pojedyncza odpowiedzialność)

SLAP - przykład



```
public void addorder(ShoppingCart cart, String userName,
                    Order order) throws SQLException {
    Connection c = null;
    PreparedStatement ps = null;
    Statement s = null;
    ResultSet rs = null;
    boolean transactionState = false;
    try {
        s = c.createStatement();
        transactionState = c.getAutoCommit();
        int userKey = getUserKey(userName, c, ps, rs);
        c.setAutoCommit(false);
        addSingleOrder(order, c, ps, userKey);
        int orderKey = getOrderKey(s, rs);
        addLineItems(cart, c, orderKey);
        c.commit();
        order.setOrderKeyFrom(orderKey);
    } catch (SQLException sqlx) {
        s = c.createStatement();
        c.rollback();
        throw sqlx;
    } finally {
        try {
            c.setAutoCommit(transactionState);
            dbPool.release(c);
            if (s != null)
                s.close();
            if (ps != null)
                ps.close();
            if (rs != null)
                rs.close();
        } catch (SQLException ignored) {
        }
    }
}
```

```
public void addorder(ShoppingCart cart, String userName,
                    Order order) throws SQLException {
    setupDataInfrastructure();
    try {
        add(order, userKeyBasedOn(userName));
        addLineItemsFrom(cart, order.getOrderKey());
        completeTransaction();
    } catch (SQLException sqlx) {
        rollbackTransaction();
        throw sqlx;
    } finally {
        cleanup();
    }
}
```

Zasady SOLID w obiektowości

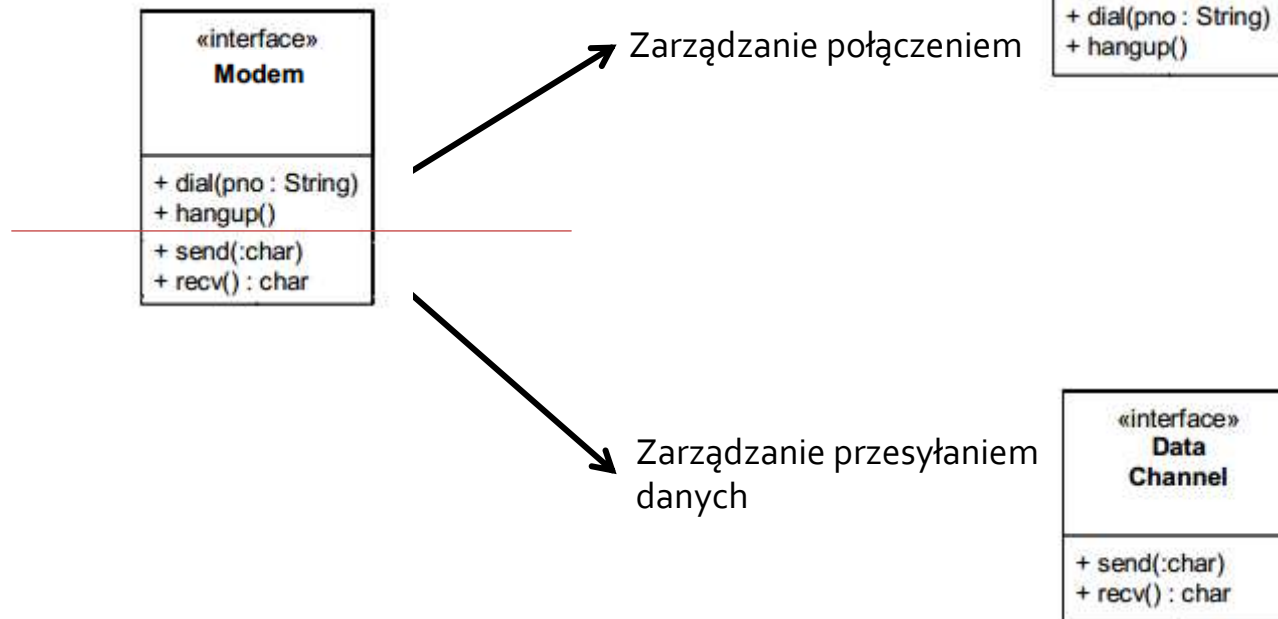


- SRP - Single Responsibility
 - OCP - Open/Closed
 - LSP - Liskov Substitution
 - ISP - Interface Segregation
 - DIP - Dependency Inversion
- } Principle

SRP - Zasada pojedynczej odpowiedzialności



- Odpowiedzialność
 - Powód do zmiany



Spójność i zależność



- **Spójność (ang. cohesion)**
 - Zmiana w A dopuszcza zmianę w B, tak że obie zyskują nową wartość
 - Jednostka (np. klasa) powinna być wewnętrznie możliwie wysoko spójna (ang. high cohesion) – wsparcie dla SRP
 - Klasy powinny mieć niewielką liczbę własności instancji. Każda metoda obiektu powinna używać jedną lub większą liczbą tych własności.
- **Zależność/sprzężenie (ang. coupling)**
 - Zmiana w B jest wymuszona zmianą w A
 - Jednostki (np. klasy) powinny być możliwie luźno powiązane (ang. loose coupling)
 - Interfejsy, minimalizacja zależności

Spójność



```
public class Stack {
    private int topOfStack = 0;
    List<Integer> elements = new LinkedList<Integer>();

    public int size() {
        return topOfStack;
    }

    public void push(int element) {
        topOfStack++;
        elements.add(element);
    }

    public int pop() throws PoppedWhenEmpty {
        if (topOfStack == 0)
            throw new PoppedWhenEmpty();
        int element = elements.get(--topOfStack);
        elements.remove(topOfStack);
        return element;
    }
}
```


Ukryte sprzężenia



```
public class UserValidator {  
    private Cryptographer cryptographer;  
  
    public boolean checkPassword(String userName, String password) {  
        User user = UserGateway.findByName(userName);  
        if (user != User.NULL) {  
            String codedPhrase = user.getPhraseEncodedByPassword();  
            String phrase = cryptographer.decrypt(codedPhrase, password);  
            if ("Valid Password".equals(phrase)) {  
                Session.initialize();  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

Jawne powiązanie

Ukryte powiązanie

Open/close principle



- Klasa powinna być otwarta dla rozszerzeń ale zamknięta dla modyfikacji
 - Idealnie wprowadzanie nowych cech nie powinno wymagać modyfikacji w istniejącym kodzie.
- Dziedziczenie
- Delegacja

Strukturalny/proceduralny kształt



```
class Square {
    public Point topLeft;
    public double side;
}

class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}

class Circle {
    public Point center;
    public double radius;
}

class Geometry {
    public final double PI = 3.141592653589793;

    public double area(Object shape) throws NoSuchShapeException {
        if (shape instanceof Square) {
            Square s = (Square) shape;
            return s.side * s.side;
        } else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.height * r.width;
        } else if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```

Struktury danych

Procedura

Obiektowy kształt



```
public interface Shape {  
    double area();  
}
```

```
class Square implements Shape {  
    private Point topLeft;  
    private double side;  
  
    public double area() {  
        return side * side;  
    }  
}  
  
class Rectangle implements Shape {  
    private Point topLeft;  
    private double height;  
    private double width;  
  
    public double area() {  
        return height * width;  
    }  
}  
  
class Circle implements Shape {  
    private Point center;  
    private double radius;  
    public final double PI = 3.141592653589793;  
  
    public double area() {  
        return PI * radius * radius;  
    }  
}
```

Liskov Substitution Principle

- Instrukcje, które manipulują obiektem wykorzystując referencje do klasy bazowej muszą być zdolne użyć obiektu klasy potomnej bez wiedzy o jego istnieniu.
 - W obiektowości relacja dziedziczenia (jest rodzajem) odnosi się do zachowania.



```
public class Square extends Rectangle {
    public void setWidth(int width) {
        m_width = width;
        m_height = width;
    }

    public void setHeight(int height) {
        m_width = height;
        m_height = height;
    }
}
```

Łamanie LSP – symptomy:

- Jeżeli dziedzicząc sprawiamy, że istniejący kod będzie musiał sprawdzać z jakim typem ma do czynienia.

```
public class HeatController {  
  
    private final List<Thermostat> thermostats;  
  
    public HeatController(List<Thermostat> thermostats) {  
        super();  
        this.thermostats = new ArrayList<Thermostat>(thermostats);  
    }  
  
    public void configure(int temp) {  
        for (Thermostat thermostat : thermostats) {  
            thermostat.setTemperature(temp);  
        }  
    }  
  
    public void tempChanged(int newTemperature) {  
        /**...*/  
        for (Thermostat thermostat : thermostats) {  
            if(thermostat.isOverHeated(newTemperature)){  
                //do something  
            }  
        }  
    }  
}
```

```
public interface Thermostat {  
  
    void setTemperature(int temp);  
  
    boolean isOverHeated(int currentTemp);  
}
```

```
public class StandardThermostat implements Thermostat {  
  
    private int temperature;  
  
    public void setTemperature(int temp) {  
        this.temperature = temp;  
    }  
  
    public boolean isOverHeated(int currentTemp) {  
        return currentTemp > temperature;  
    }  
}
```

Łamanie LSP – symptomy:

- Jeżeli dziedzicząc sprawiamy, że istniejący kod będzie musiał sprawdzać z jakim typem ma do czynienia.

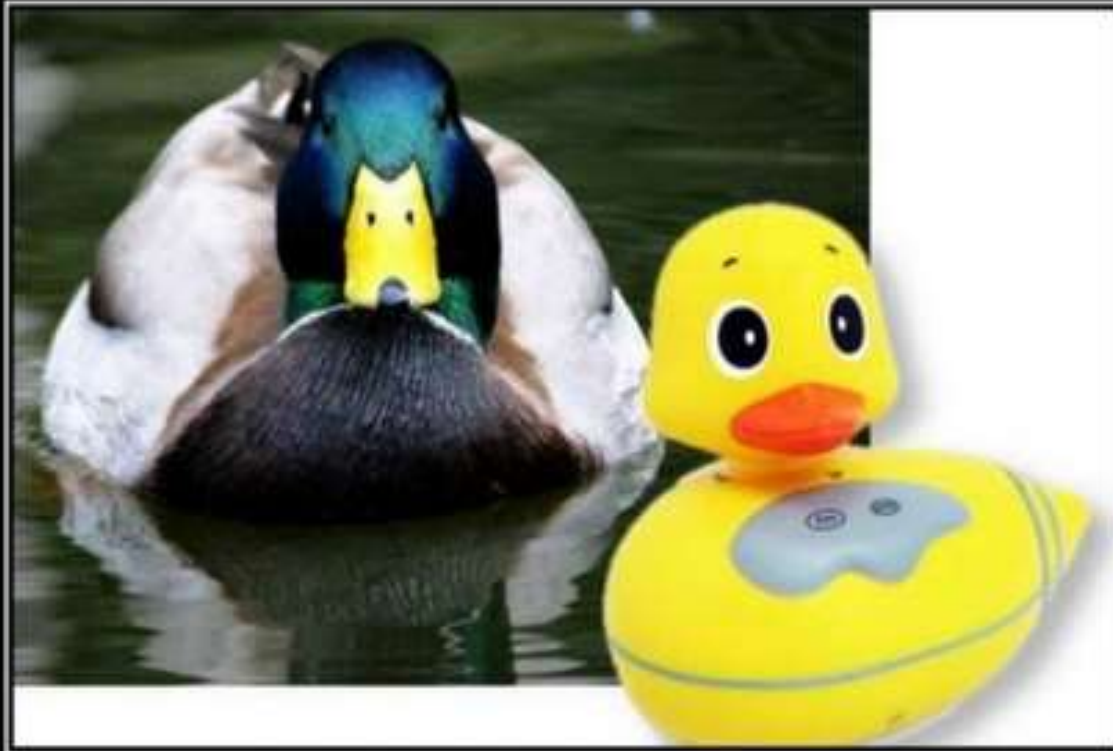
```
public class HeatController {  
  
    private final List<Thermostat> thermostats;  
  
    public HeatController(List<Thermostat> thermostats) {  
        super();  
        this.thermostats = new ArrayList<Thermostat>(thermostats);  
    }  
  
    public void configure(int temp) {  
        for (Thermostat thermostat : thermostats) {  
            thermostat.setTemperature(temp);  
        }  
    }  
  
    public void tempChanged(int newTemperature) {  
        /**...*/  
        for (Thermostat thermostat : thermostats) {  
            if(thermostat.isOverHeated(newTemperature)){  
                //do something  
            }  
        }  
    }  
}
```

```
public class StandardThermostat implements Thermostat {
```

```
public interface Thermostat {  
  
    void setTemperature(int temp);  
  
    boolean isOverHeated(int currentTemp);  
}
```

```
public class ClockThermostat extends StandardThermostat {  
  
    private Map<DayOfWeek, Integer> tempForDay  
    = new EnumMap<DayOfWeek, Integer>(DayOfWeek.class);  
  
    public void setTemperatureForDayOfWeek(DayOfWeek day, int temp) {  
        tempForDay.put(day, temp);  
        this.setTemperature(temp);  
    }  
  
    @Override  
    public boolean isOverHeated(int currentTemp) {  
  
        Integer temperatureForDay =  
            tempForDay.get(LocalDate.now().getDayOfWeek());  
        if (temperatureForDay == null) {  
            return super.isOverHeated(currentTemp);  
        }  
        return temperatureForDay > currentTemp;  
    }  
}
```


Demot

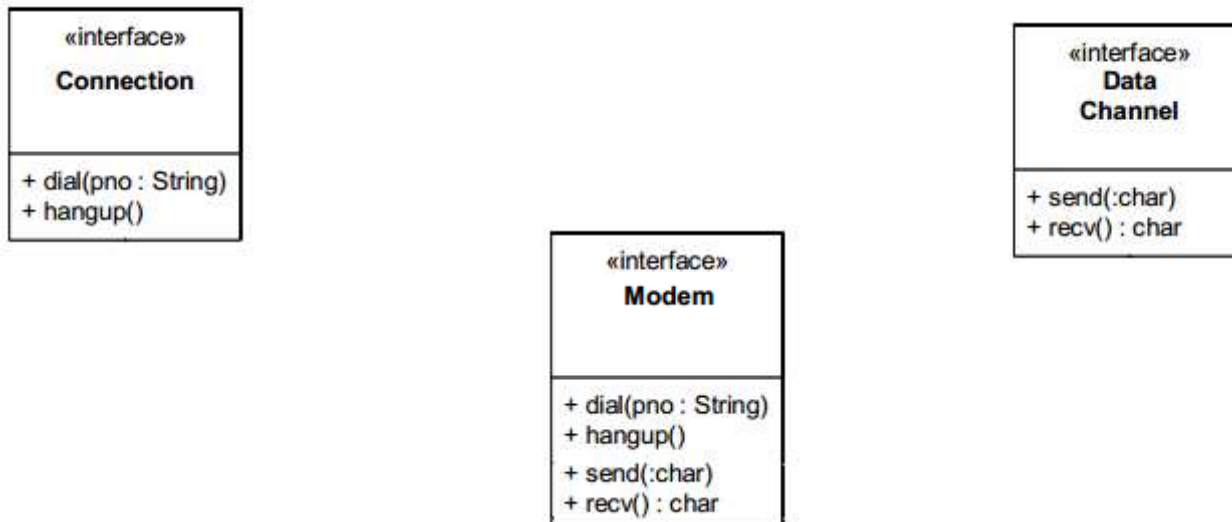


LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

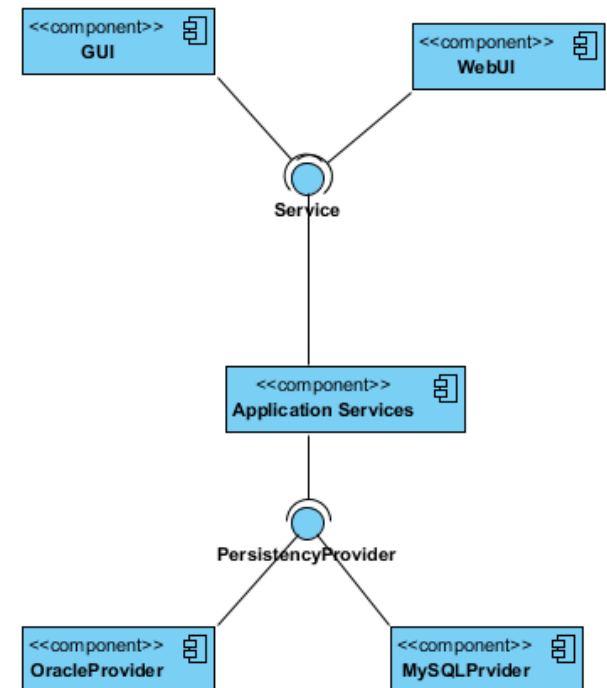
Interface Segregation Principle

- Klient nie powinien być zmuszany do zależności od interfejsów, których nie używa
- SRP dla interfejsów



Dependency Inversion Principle

- Moduły wyższego poziomu nie powinny być zależne od modułów niższego poziomu. Oba powinny zależeć od abstrakcji (interfejsu)



Inne reguły



1. Programuj względem interfejsów nie implementacji.
2. Hermetyzuj zmienność.
3. Przedkładaj kompozycję na dziedziczenie.
4. Stosuj wzorce projektowe kiedy tylko jest to możliwe.
5. Staraj się budować systemy o luźnych zależnościach.
6. KISS – prostota, prostota, ...

Ale jak pisać dobry kod

- Ćwiczyć, ćwiczyć, szukać wzorców.
- Ale przecież ćwiczę bo w pracy koduje...
- Ćwiczenia niekoniecznie muszą być odkrywczе – dążenie do perfekcji wymaga wielokrotnego powtarzania tych samych czynności w warunkach „laboratoryjnych”



Bibliografia



- hasschapman.blogspot.com/2011/11/visualising-your-technical-debt.html
- dearjunior.blogspot.com/2012/03/dry-and-duplicated-code.html
- c2.com/cgi/wiki?CouplingAndCohesion
- www.slideshare.net/skarpushin/solid-ood-dry
- www.oodesign.com/liskov-s-substitution-principle.html
- www.codeproject.com/Articles/567768/Object-Oriented-Design-Principles
- www.odi.ch/prog/design/newbies.php
- *The Pragmatic Programmer*, Andrew Hunt and David Thomas, 1999
- Robert C. Martin: *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008

A must read!



- Andy Hunt, **Pragmatic Thinking and Learning: Refactor Your Wetware** (The Pragmatic Programmer series)