

# Zwinne wprowadzenie do TDD, BDD cz. 2 Jak pisać testy (i kod)

Krzysztof Manuszewski



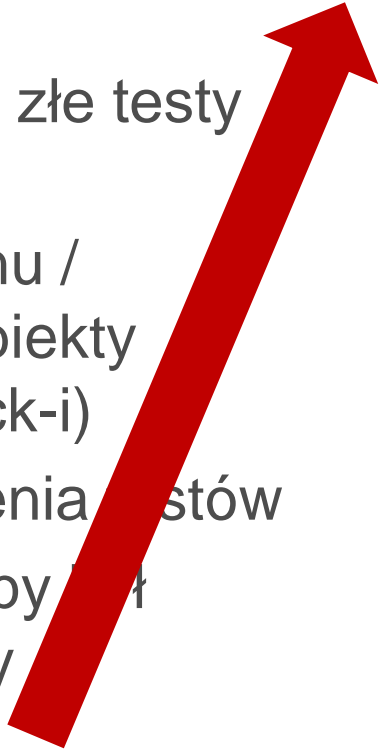
# Agenda

- Elementy testu jednostkowego
- Czym różnią się złe testy od dobrych
- Testowanie stanu / zachowania - obiekty Namiastek (Mock-i)
- Strategia tworzenia testów
- Jak pisać kod aby był łatwo testowalny
- Zasady SOLID



Trudno jest testować  
zły kod ...

# Nowa agenda

- Elementy testu jednostkowego
  - Czym różnią się złe testy od dobrych
  - Testowanie stanu / zachowania - obiekty Namiastek (Mock-i)
  - Strategia tworzenia testów
  - Jak pisać kod aby był łatwo testowalny
  - Zasady SOLID
- 
- Zasady SOLID czyli jak pisać dobry kod

# Nowa agenda

- Zasady SOLID czyli jak pisać dobry kod
- Elementy testu jednostkowego
- Czym różnią się złe testy od dobrych
- Testowanie stanu / zachowania - obiekty Namiastek (Mock-i)
- Strategia tworzenia testów



**ZŁY KOD WYGLĄDA TAK ...**

## ... jest „sztywny” i „delikatny”

- ❑ Nowe błędy pojawiają się w obszarach, które zdają się być niezwiązane ze zmienianą funkcjonalnością
- ❑ Pozornie drobne zmiany indukują poważne zmiany w wielu miejscach kodu i/lub skutkują trudnymi do przewidzenia błędami
- ❑ Trudno przewidzieć efekt nawet drobnych zmian
- ❑ Trudno przewidzieć czas oraz koszty rozwoju projektu/poprawek
- ❑ Zespół programistów traci wiarygodność
- ❑ Menedżerowie niechętnie godzą się na zmiany

## ... i trudny do „reuzycia”

- Potrzebne elementy zależą od niepotrzebnych
- Ryzyko ekstrakcji potrzebnego kodu jest duże a jej koszt większy niż napisanie potrzebnej funkcjonalności od podstaw



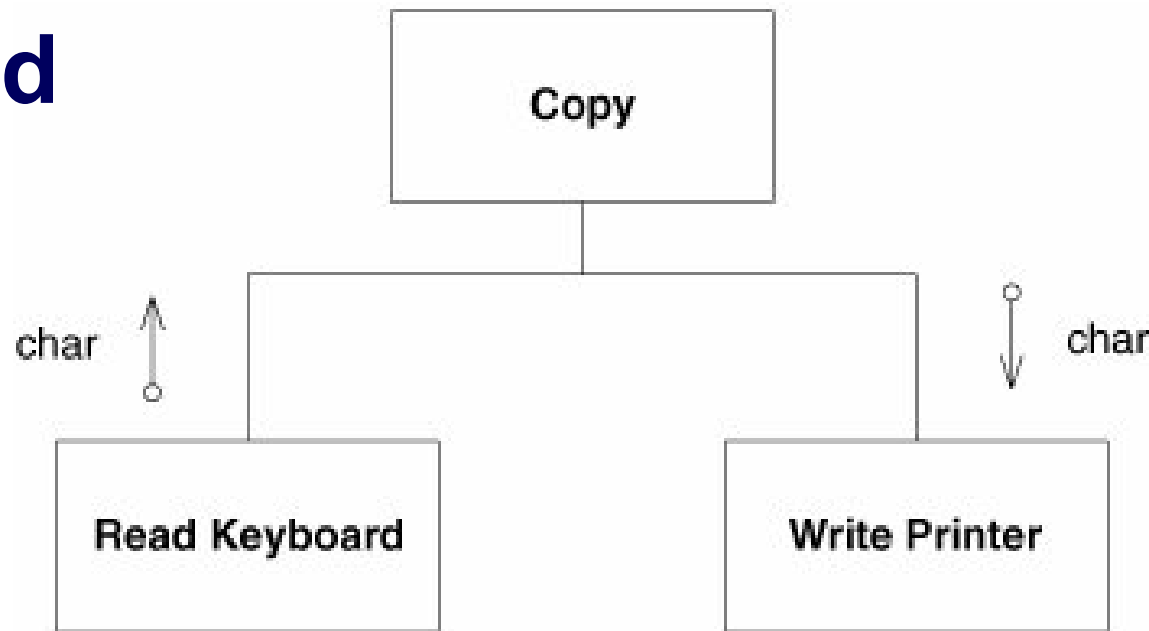
# Bezpośrednie źródła problemów

- Praca z cudzym kodem
- Pośpiech
- Zmiany, ciągłe zmiany
- Niedostateczna/niejasna specyfikacja

... a może przyczyną jest nienajlepsza struktura kodu

# Prosty przykład

- kopiowanie
- znaków
- z klawiatury
- na drukarkę



```
public class Copier {
    public static void Copy() {
        int c;
        while( (c=Keyboard.Read()) != -1)
            Printer.Write(c);
    }
}
```

## ... drobna zmiana ...

- Z klawiatury lub z czytnika taśmy

```
public class Copier {  
    public static bool rdFlag = false;  
    public static void Copy() {  
        int c;  
        while( (c= (rdFlag ? PaperTape.Read()  
                    : Keyboard.Read() ) ) != -1 )  
            Printer.Write(c) ;  
    }  
}
```

# ..i następna ...

## ■ Na drukarke lub ekran

```
public class Copier {  
    public static bool rdFlag = false;  
    public static bool ptFlag = false;  
    public static void Copy() {  
        int c;  
        while( (c= (rdFlag ? PaperTape.Read()  
                    : Keyboard.Read() ) ) != -1)  
            if (ptFlag) Screen.Write(c);  
            else Printer.Write(c);  
    }  
}
```

# ...i już nie jest prosty...

- Więcej źródeł i ujść danych
- Obsługa błędów I/O
- Przekodowywanie znaków
- Logowanie kopiowanych znaków do pliku
- Zmiana formatu tekstu w oparciu o kontekst (np. justowanie tekstu)



**Wymagania się zmieniają ...**

**Zawsze**

lub przynajmniej

**często**

zwłaszcza w kontekście  
iteracyjnej realizacji projektu

# ... być gotowym na zmiany

```
public class KeybrdReader : {
    public int Read() { return Keyboard.Read(); }
}

public class PrinterWriter : {
    public Write(int c) { return Printer.Write(c); }
}

public class Copier {
    public static KeybrdReader reader= new KeybrdReader();
    public static PrinterWriter writer= new PrinterWriter();
    public static void Copy() {
        int c;
        while( (c=(reader.Read())) != -1)
            writer.Write(c);
    }
}
```

# ... to podzielić odpowiedzialność

Wyraźna separcja kodu:

Twórcy klas Printer i Writer **nie muszą znać ani rozumieć logiki kopiowania**

Ale:

- Zmiany w sposobie obsługi drukarki/klawiatury:
  - ☐ wymuszają zmiany w klasie Copier (typy atrybutów)
  - ☐ wymuszają rekompilację klasy Copier
- Implementujący klasę Copier musi znać i umieć tworzyć obiekty klas Printer, KeybrdReader
- Klasa Screen musi dziedziczyć po Printer (choć nie ma nic wspólnego z drukarką)






**LEPSZY KOD WYGLĄDA TAK ...**

```
public interface IReader {
    public int Read() ;
}

public class KeybrdReader : IReader{
    public int Read() { return Keyboard.Read() ; }
}

public class Copier {
    public IReader reader = new KeybrdReader() ;
    public IWriter writer = new PrintWriter() ;
    public void Copy() {
        int c;
        while( (c=(reader.Read())) != -1)
            writer.Write(c) ;
    }
}
```



```
public class Copier {  
    IReader  reader;  
    IWriter  writer;  
    public Copier (IReader newReader,  
                   IWriter newWriter) {  
        reader = newReader;  
        writer = newWriter;  
    }  
  
    public static void Copy() {  
        ...  
    }  
}
```

# Dobra separacja kodu

- Klasa Copier **nie zależy** od Printer ani od Reader
- Klasy Printer ani Reader **nie zależą** od Copier
- Wszystkie klasy (usługobiorcy i usługodawcy zależą od interfejsu)
- Zmiany interfejs-u są jedynym powodem do zmian w większych obszarach kodu
- Interfejs stanowi specyfikację kontraktu pomiędzy 2 stronami: klientem i dostarczycielem pewnej funkcjonalności

# S.O.L.I.D. - ny kod

- **S**RP: The Single Responsibility Principle
- **O**CP: The Open/Close Principle
- **L**SP: The Liskov Substitution Principle
- **I**SP: The Interface Segregation Principle
- **D**IP: The Dependency Inversion Principle

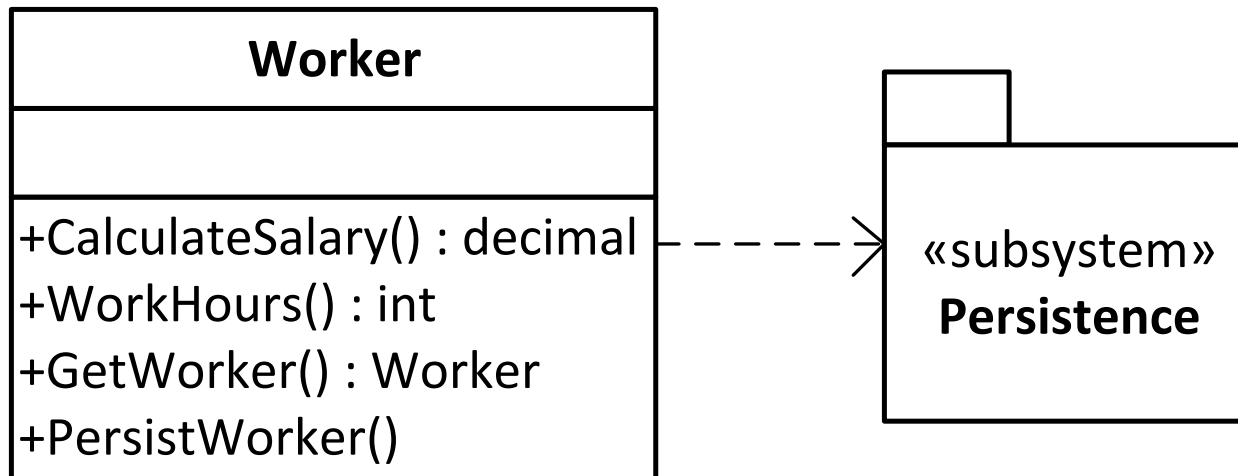
***Robert C.C Martin***

# (SRP) Single-Responsibility Principle

*Klasa powinna mieć pojedynczy powód do zmian*

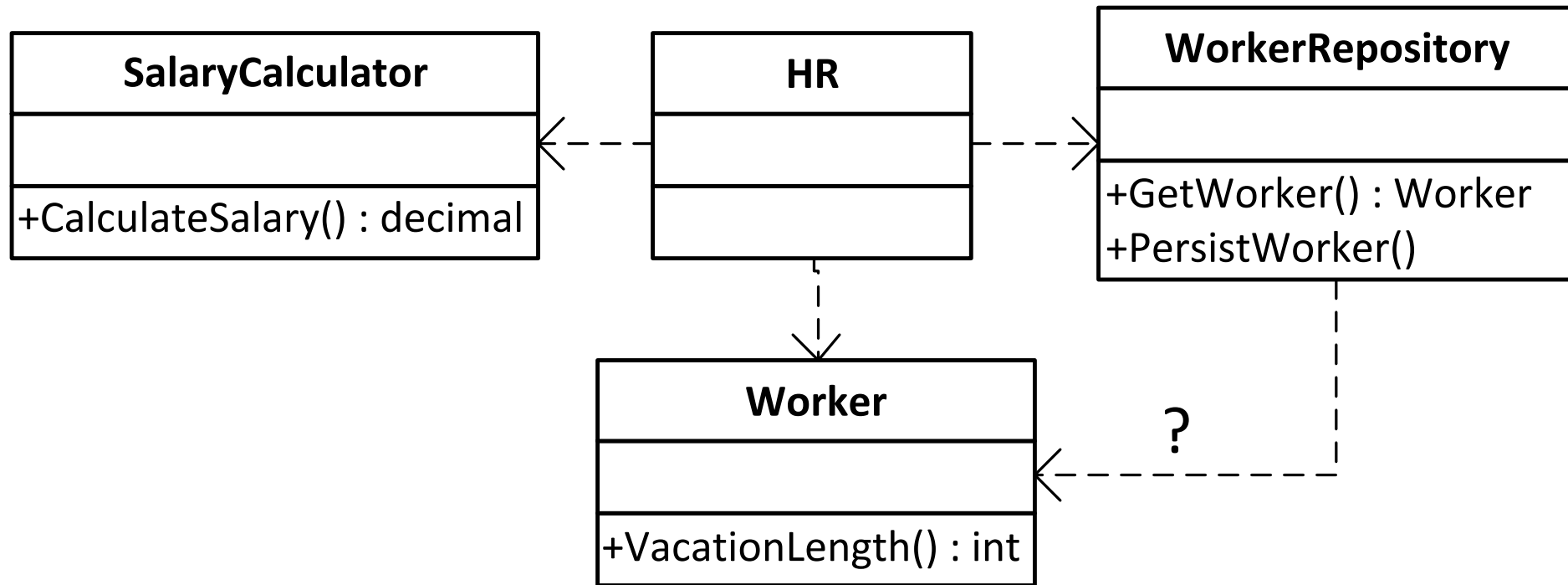
- Klasa Printer jest odpowiedzialna za pisanie na drukarkę
- Klasa Copier jest odpowiedzialna za proces kopiowania

# SRP - przykład problemów



- Zmiany dowolnego z aspektów oznaczają zmiany w klasie Worker.
- Łatwo wprowadzić błąd do pobocznej funkcjonalności.
- Testować trzeba całą klasę.

# SRP - Możliwe rozwiązanie





# Moment!

- Obiekty powinny hermetyzować swoją zawartość!
- Czy obiekty powinny mieć wiedzę:
  - ☐ Jak zapisać samego siebie?
  - ☐ Jak raportować swój stan?
- To nie jest tak istotne!
  - ☐ Filozofia, która kryje się za OO nie jest w tym wypadku tak istotna
  - ☐ Podstawowym celem jest ograniczenie propagacji zmian w systemie
  - ☐ System ma być łatwy w utrzymaniu i modularny!

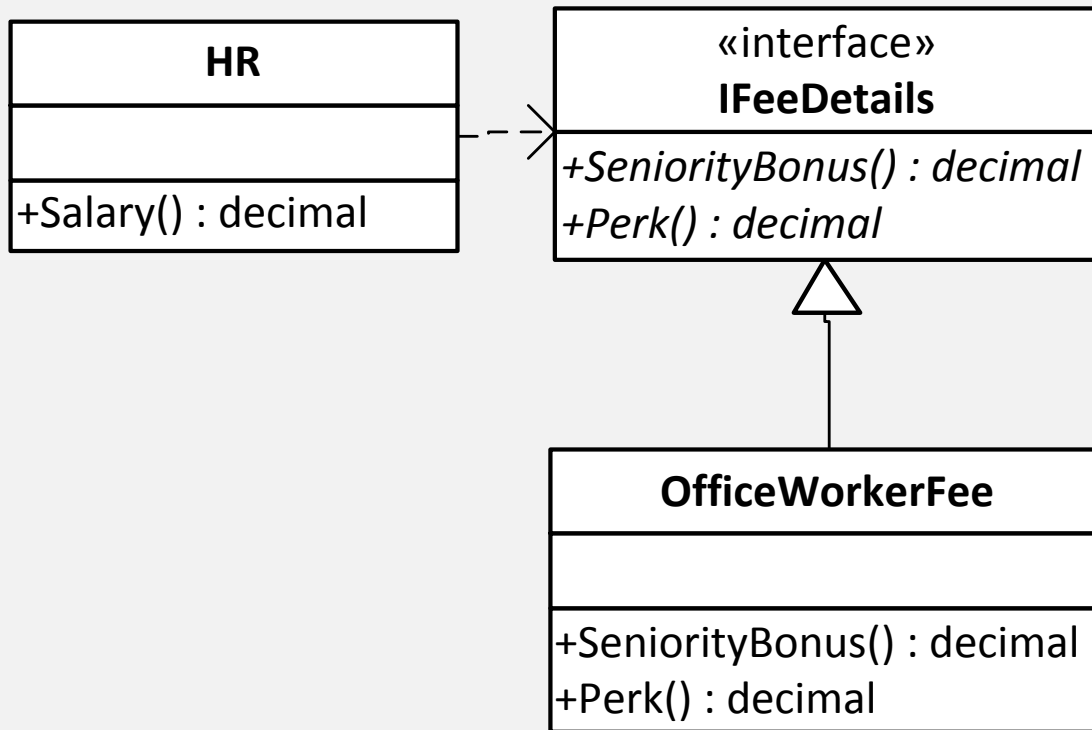
# **(OCP) Open/Close Principle**

**Jednostki programowe (klasy, moduły, funkcje, itd.) powinny być otwarte na modyfikacje a zamknięte na zmiany**

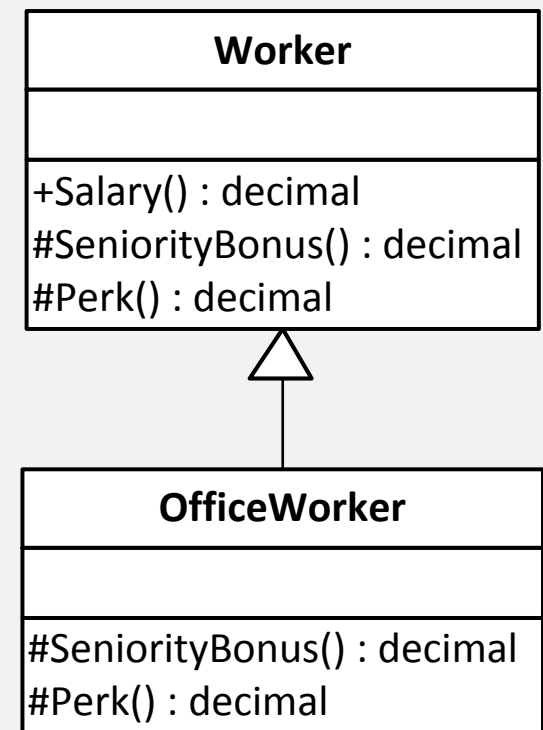
- Do klasy Copier można łatwo dodać nowe typy czytników/pisarzy (bez zmian w klasie Copier)
- „Bez zmian”? - klasa Copier nie powinna sama tworzyć obiektów, z których korzysta

# OCP - Otwartość na rozszerzanie

- Zapewniają np. wzorce projektowe:



strategia



metoda szablonowa

- Które rozwiązanie wybrać ?

# Które rozwiązanie wybrać?

- Dziedziczenie

- silniejsze związki między klasami

- Agregacja

- możliwość zmiany zachowania w czasie wykonania
  - możliwość niezależnego określania zachowania w różnych obszarach (niezależnych strategii)

- Dziedziczenie interfejsu jest naturalne

- Dziedziczenie implementacji niekoniecznie

- Jeśli nie ma dodatkowych wskazówek  
- lepszym rozwiązaniem agregacja może być!

# OCP – co ze zmianami?

- Nie można zapobiec wszystkim zmianom!
- Czym się kierować:
  - ☐ co może się zmieniać często
  - ☐ co będzie trudno zmienić w przyszłości
- Dodanie nowej funkcjonalności powinno być łatwe



# **(LCP) Liskov Substitution Principle**

**Podklasy muszą być logicznie zgodne z typami bazowymi.**

# LCP – Szkolny przykład (1)

```
public class Rectangle
{
    private Point topLeft;
    private double width;
    private double height;

    public double Width
    {
        get { return width; }
        set { width = value; }
    }

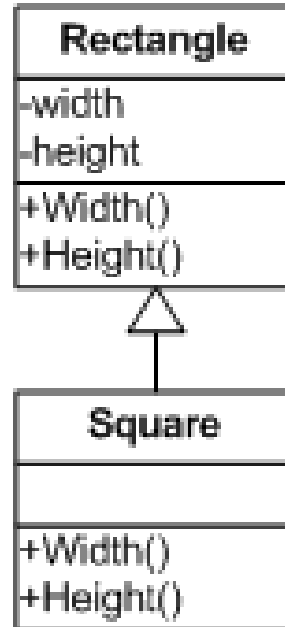
    public double Height
    {
        get { return height; }
        set { height = value; }
    }
}
```

# LCP – Szkolny przykład (2)

```
public class Rectangle
{
    private Point topLeft;
    private double width;
    private double height;
```

```
    public virtual double Width
    {
        get { return width; }
        set { width = value; }
    }
```

```
    public virtual double Height
    {
        get { return height; }
        set { height = value; }
    }
}
```



```
public class Square : Rectangle
{
    public override double Width
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
```

```
    public override double Height
    {
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```



## LCP – Szkolny przykład (3)

```
void foo (Rectangle r)
{
    r.Width = 32; // calls .SetWidth (32)
}
```

Co będzie gdy przekażemy obiekt typu Square do kodu poniżej ?

```
void goo (Rectangle r)
{
    r.Width = 5;
    r.Height = 4;
    if(r.Area() != 20)
        throw new Exception("Bad area!");
}
```

**Kwadrat nie zachowuje się jak szczególny przypadek prostokąta!**

# **(LCP) Liskov Substitution Principle**

**Podklasy muszą być logicznie zgodne z typami bazowymi.**

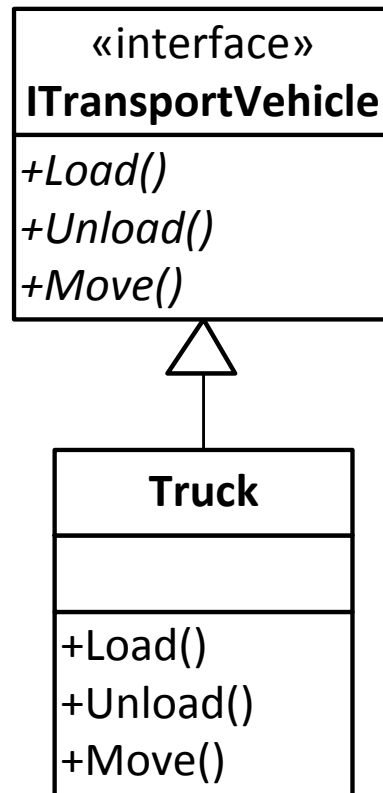
- **TapeReader : KeyboardReader ???**
- **Dziedziczenie oznacza „jest szczególnym przypadkiem”**

# (ISP) Interface Segregation Principle

*Klasa nie powinna zależeć od tego, czego nie używa*

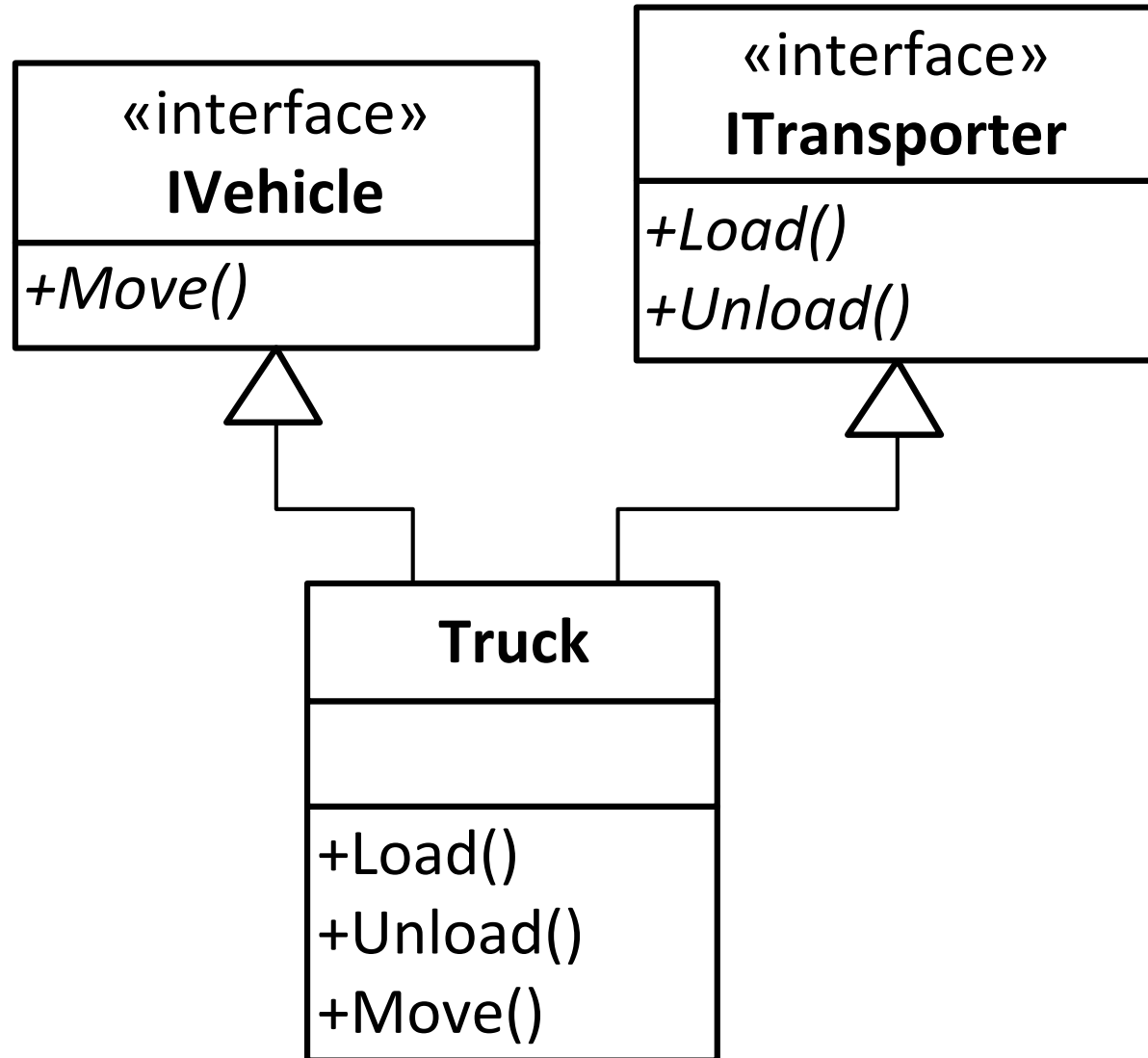
- **Interfejs IReader powinien być oddzielny od IWriter.**
- „Tłuste” klasy wprowadzają zwykle silne związki ze swoimi klientami
- Zmiana wymuszona przez jednego z klientów dotyka pozostałych

# (ISP) Interface Segregation Principle



- Kierowca niekoniecznie odpowiada za załadunek/rozładunek
- Jak reprezentować przyczepę?

# (ISP) Interface Segregation Principle



# (DIP) Dependency Inversion Principle

1. **Wysokopoziomowe moduły nie powinny zależeć od niskopoziomowych (ani odwrotnie). Jedne i drugie powinny zależeć od abstrakcji (kontraktów).**
  2. **Abstrakcje (kontrakty) nie powinny zależeć od szczegółów implementacyjnych. Implementacja powinna zależeć od abstrakcji (kontraktu).**
- Hollywood principle: „Don't call us – we will call you”
  - Copier i KeyboardReader zależą od interfejsu IReader, ale nie zależą od siebie
  - Interfejs jest bardziej przejrzysty niż klasa

# Problemy w kodzie

- Nadmiar abstrakcji
  - kod działa również bez interfejsów
- Zbyt duże rozdrobnienie logiki
  - klasy mogą mieć więcej niż kilka linii....




# Testy jednostkowe



# Części testu czyli AAA

- Arrange – utwórz SUT, zainicjuj środowisko
  - Act – wykonaj test
  - Assert – sprawdź wyniki
- 
- Przygotowanie może być realizowane
    - ☐ W samym teście
    - ☐ Wydzielonej części SetUp/Establish
- 
- Różne modele asercji



```
[TestFixture]
public class Calculator_Tests {

    [Test]
    public void TestOfASingeNumber()      {
        var sut = new StringCalculator();
        var result = sut.Add("2");
        Assert.AreEqual(2,result, "Result should\
                                be equal to a singleArgument");
    }
}
```

# Przypomnienie:

## Po co pisać testy jednostkowe?

- Testy weryfikują **na bieżąco** konkretne aspekty zachowania klas. Złamanie założeń powoduje załamanie konkretnych testów.
- Przy dodawaniu/zmianach funkcjonalności testy chronią przed zepsuciem już zaimplementowanych funkcji.
- Stanowią dokumentację i zarazem przykłady użycia
- Kod powinien być pisany prosto. Działający kod można i **należy** udoskonalać. Aby to było bezpieczne potrzebne są testy.

# Gdzie leży problem

- Kod produkcyjny powinien być dobrze przetestowany – nie powinna udać się żadna zmiana logiki bez “wysadzenia” jednego lub wielu testów
- Test powinien się jasno nazywać i powinien testować pojedynczy aspekt działania kodu to znaczy poszczególne zmiany w kodzie powinny "zapalać" jak najmniej testów
- Nie należy pisać niepotrzebnych testów
- Kod testów powinien zawierać jak najmniej powtórzeń
- Test powinien być zrozumiały

# Dlaczego

**Kod produkcyjny powinien być dobrze przetestowany ...**

- Jeśli testy mają "dziury" – nie można im ufać - cały wysiłek nie ma sensu.
- Jeśli znajdujemy problem w kodzie produkcyjnym – dodajemy nowy test/testy a potem poprawiamy kod.

# Co testować

- **Logikę.** Instrukcje warunkowe, pętle itd.  
Testowanie prostych właściwości/funkcji mija się z celem.
- **Publiczny interfejs.** Jeżeli metody prywatne zawierają nietrywialną logikę może to znak, że klasa powinna zostać zrefaktoryzowana.
  - Np. samochód vs. Silnik

## Dyskusyjne obszary:

- Konstruktory
- Gui
- Baza danych

# Dlaczego

## Test powinien się jasno nazywać

- Testy powinny wskazywać na konkretne problemy
- Dobre nazwy zwalniają ze szczegółowych komunikatów przy asercja
- Niejasna nazwa przy dużym zestawie testów powoduje, że musimy debugować/analizować kod
- **Jeśli mamy problem z nazwą testu** bardzo prawdopodobne, że próbujemy przetestować zbyt wiele rzeczy na raz

# Jasno czyli jak ?

## Konwencje

- ☐ LoginComponent\_InvalidUser\_ShuldThrowException
- ☐ WhenUserIsInvalid. IsLoginOk\_ShouldThrowException



# Dlaczego

**Test powinien testować pojedynczy aspekt działania kodu**

- **Poszczególne zmiany w kodzie powinny “zapalać” jak najmniej testów**
- **Test, który testuje wiele rzeczy**
  - ☐ będzie częściej "padać" przy zmianach kodu
  - ☐ jest zwykle skomplikowany
  - ☐ niewiele mówi w momencie upadku

# Dlaczego

## Nie należy pisać niepotrzebnych testów

- Pisanie testów kosztuje!
- Utrzymanie testów kosztuje ... jeszcze więcej!
- Jeśli 2 testy padają zawsze razem – jeden jest niepotrzebny.
- Jeśli test pada zawsze razem z innymi – jest niepotrzebny.
- Pokusa: *"jak przetestuję coś jeszcze raz\* będę mieć kod lepiej przetestowany...."*
  - Piszemy więcej kodu
  - Testy padają częściej

\* *Jeszcze raz = testuję coś co testują już inne testy*

# Dlaczego

## Kod testów powinien zawierać jak najmniej powtórzeń

- Duplikacja w kodzie to **ZŁO** ...
  - jest zbędna, utrudnia utrzymanie, zaciemnia kod
- Jak unikać
  - wspólny kod inicjalizujący
    - Setup (ale nie tak ze jest jedno setup dla wszystkich testów – wolne i niejasne)
    - Hierarchia klas
    - Buildery danych testowych
  - własne asserty
  - elastyczny framework np. MSpec

# Dlaczego

## Test powinien być zrozumiały...

- Test stanowi dokumentację
- Przed zmianą funkcjonalności należy zmienić test
- Nie lubię zmieniać tego czego nie rozumiem!!!

## ■ Test nie powinien zawierać logiki

- ☐ jak testować testy?
- ☐ Jeśli test zawiera logikę należy ją wydzielić (np. do funkcji). Takie funkcje można przetestować.

## ■ Testy można i należy refaktoryzować

# NUnit

```
[TestFixture]
```

```
public class When_calculator_is_tested {
```

```
    [Test]
```

```
    public void when_a_single_number_is_provided_add\  
                _should_return_this_number () {
```

```
        var sut = new StringCalculator();
```

```
        var result = sut.Add("2");
```

```
        result.ShouldEqual(2);
```

```
    }
```

```
    [Test]
```

```
    public void when_two_numbers_are_provided_add\  
                _should_return_sum_of_them () {
```

```
        var sut = new StringCalculator();
```

```
        var result = sut.Add("2,3");
```

```
        result.ShouldEqual(5);
```

```
    }
```

```
}
```



```
[TestFixture]
```

```
public class When_calculator_is_tested {
```

```
    StringCalculator sut;
```

```
    [SetUp]
```

```
    public void  SetUp() {
```

```
        sut = new StringCalculator();
```

```
    }
```

```
    ...
```

```
    [Test]
```

```
    public void when_two_numbers_are_provided_add\  
                _should_return_sum_of_them () {
```

```
        var sut = new StringCalculator();
```

```
        var result = sut.Add("2,3");
```

```
        result.ShouldEqual(5);
```

```
    }
```

```
}
```

# MSpec

```
class with_string_calculator
{
    Establish context = () =>
        sut = new StringCalculator();

    protected static StringCalculator sut;
}

class when_two_numbers_are_provided_for_addition :
    with_string_calculator
{
    Because of = () =>
        result = sut.Add("2,3");


    It should_return_their_sum = () =>
        result.ShouldEqual(5);

    static int result;
}
```

# Jak pisać testy - podsumowanie

- **Dobrej jakości** testy **nie wymagają** intensywnej pielęgnacji.
- Projekty (agile) częściej padają nie z powodu braku ale z powodu **złej jakości testów**
- **Kod nie może zawierać "hack-ów"** (if test ....)
- **Test, który zawsze działa – nic nie testuje.** Zawsze należy sprawdzić czy istnieją przypadki gdy test zawodzi
- **Typowy (tradycyjny) kod jest trudny do testowania.**
- **Testy dla istniejącego (i stabilnego kodu) mają umiarkowany sens (chyba że chcemy kod zmieniać)**





# Testowanie zachowania

## Izolacja klas

# Filozofia: co testować

- **Stan** obiektów

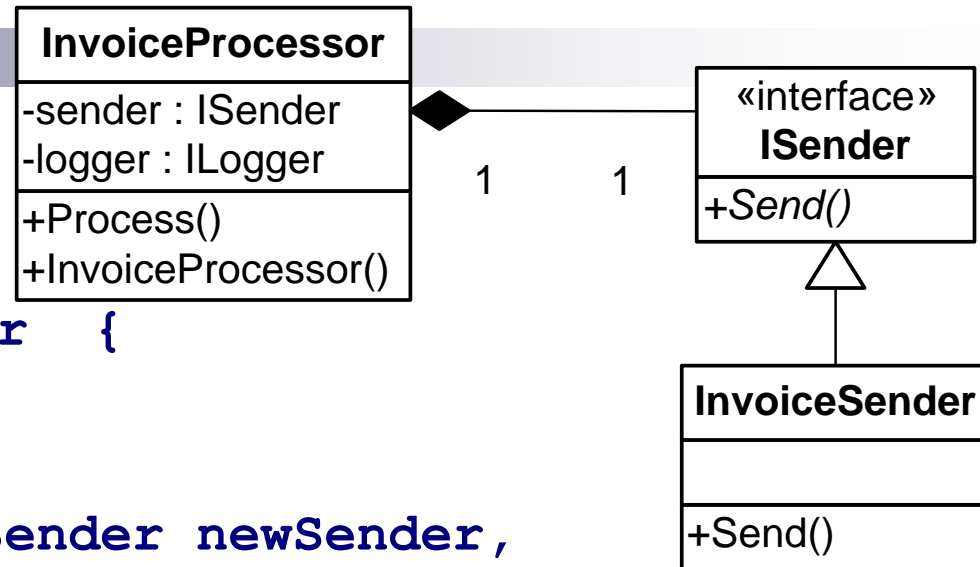
- **Zachowanie** obiektów:

- Testujemy wołania innych funkcji/obiektów
- Zasada proś [o przysługę] nie pytaj [o stan]

- **Jak trzeba mieszamy podejścia**

- Przede wszystkim należy testować to co **ma wartość** z punktu widzenia klienta

# Zachowanie ...



```
public class InvoiceProcessor {
    private ISender sender;
    private ILogger logger;

    public InvoiceProcessor(ISender newSender,
                           ILogger newLogger) {
        sender = newSender; logger = nLogger;
    }

    public bool Process(...) {
        logger.Log("start");
        if (...) {
            ...
            bool ret = sender.Send(invoice);
            ...
        }
    }
}
```

PRZETESTOWAĆ

```
var procesor = new InvoiceProcesor(new InvoiceSender(...),
    new Logger());
```



## ...to nie stan

Problem 1: ignorujemy zachowanie kodu `logger.Log()`

Problem 2: nie mamy skonfigurowanego sendera  
– czy `sender.Send()` zwrócił `true` czy `false`

Problem 3: czy sender został wywołany i z jakimi parametrami

# Wymagane zastępstwo

## Problem 1:

```
public class FakeLogger : ILogger {  
    public void Log(string msg) {}  
}
```

## Problem 2:

```
public class FakeSender : ISender {  
    public bool Ret = true;  
    public bool Send (object toSend) {  
        return Ret; }  
}
```

# Wymagane zastępstwo

## Problem 3:

```
public class FakeSenderValidator : ISender
{
    public bool Ret = true;
    public bool SendWasCalled = false;
    public object SendArgument;
    public bool Send (object toSend) {
        SendWasCalled = true;
        SendArgument = toSend;
        return Ret;
    }
}
```

# Bez nowych klas...

Stub:

- obiekt kreowany dynamicznie – akceptujący wołania i ew. zwracający zadane wartości

Mock:

- obiekt kreowany dynamicznie – z możliwością weryfikacji konkretnych zachowań

Mocking frameworks:

- Nmock, Moq – stosunkowo proste
- RhinoMock – bardzo zaawansowany
- TypeMock – jeszcze bardziej zaawansowany ale ... komercyjny

# Przykład 1, 2

[Test]

```
public void Process_whenSendingSuccessful_...() {  
    //Problem1:  
    var logger = MockRepository.GenerateStub<ILogger>();  
    //Problem2:  
    var sender = MockRepository.GenerateStub<ISender>();  
    sender.Stub(s => s.Send(null)).  
        IgnoreArguments().  
        Return(true);  
  
    InvoiceProcessor sut = new InvoiceProcessor(sender, logger);  
    var result = Sut.Process(...);  
    ...  
}
```



# Przykład 3

[Test]

```
public void Process_whenSendingSuccessful_...() {  
    var logger = MockRepository.GenerateStub<ILogger>();  
    var sender = MockRepository.GenerateStub<ISender>();  
    sender.Stub(s => s.Send(null))  
        .IgnoreArguments()  
        .Return(true);  
    Invoice invoice = ...;  
    InvoiceProcessor sut = new InvoiceProcessor(sender, logger);  
    var result = sut.Process(invoice);  
    ...  
    //Problem 3:  
    sender.AssertWasCalled( s => s.Send(invoice) );  
}
```

# Problemy przy testach

## Niejawne wejście - środowisko zewnętrzne np.:

- Pojawienie się pliku
- Brak pamięci
- Pojawienie się procesu
- Otrzymanie maila
- Przyciśnięcie przycisku w GUI
- Zmiana danych w bazie

## Niejawne wyjście – efekt działania kodu np.:

- Skasowanie pliku
- Zabicie procesu
- Wysłanie maila
- Wyświetlenie czegoś na ekranie, zmiana stanu elementów GUI
- Zapis danych do bazy

# Trudny test

SystemMonitor
+StartMonitoring()

```
public void StartMonitoring(...)  
{
```

```
    ...
```

```
        if (System.IO.File.Exists("myFile"))  
            //send email
```

```
}
```

Niejawne wejście



Niejawne wyjście



# Wydzielone trudne elementy

SystemMonitor
+StartMonitoring() #FileExists() : bool #SendEmail()

```
class SystemMonitor{
    public void StartMonitoring(...) {
        ...
        if (FileExists("myFile"))    SendEmail(...)
        ...
    }
    protected virtual bool FileExists(string fileName)
    {
        return System.IO.File.Exists(fileName);
    }
    protected virtual bool SendEmail (...) {
        //send email
    }
}
```

# Dedykowana Podklasa

```
class SystemMonitorTestSubclas : SystemMonitor {
```

```
    public bool fileExists = true;
```

```
    public bool emailSent = false;
```

```
    public override void SendEmail(...)  
{
```

```
        emailSent = true;
```

```
    }
```

```
    public override bool FileExists (...)  
{
```

```
        return fileExists;
```

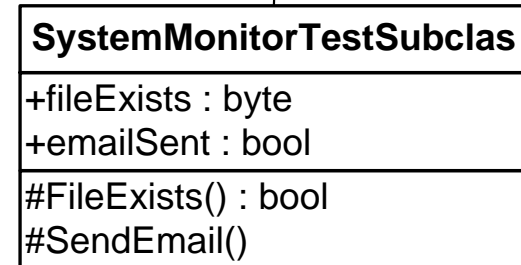
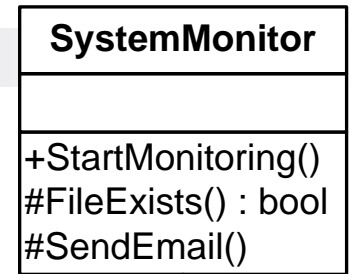
```
    }
```

```
}
```

```
var sut = new SystemMonitorTestSubclas ();
```

```
sut.StartMonitoring();
```

```
Assert.IsTrue(sut.emailSent);
```



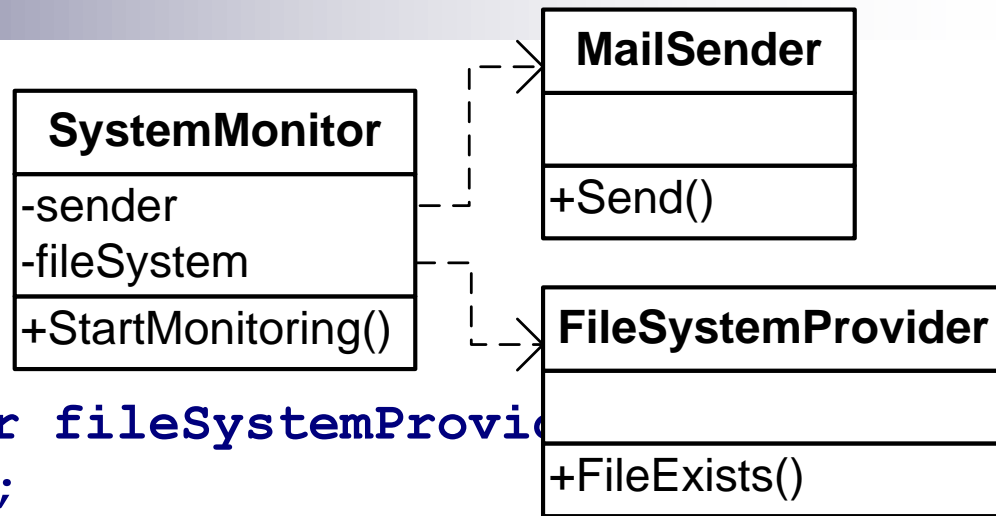
# Mock podklasy

SystemMonitor
+StartMonitoring() #FileExists() : bool #SendEmail()

```
var sut = MockRepository
    .GeneratePartialMock< SystemMonitor >();
sut.Stub(s => s.FileExist (null))
    .IgnoreArguments().Return(true);
sut.Stub(s => s.SendEmail(null))
    .IgnoreArguments();
....

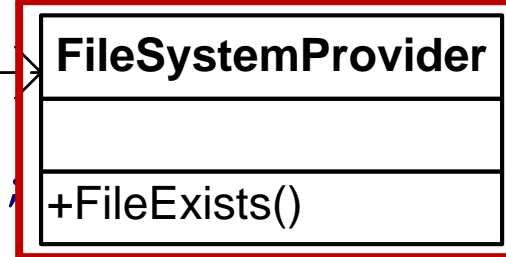
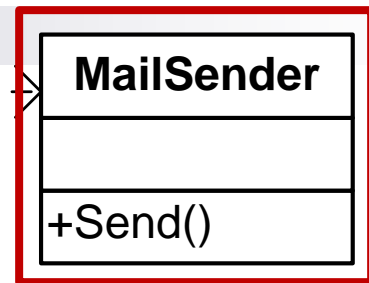
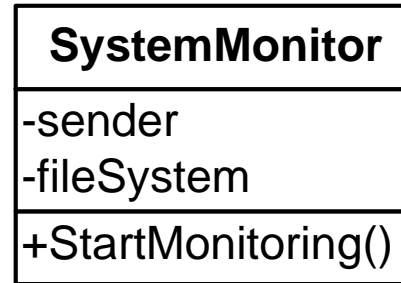
sut.StartMonitoring();
sut.AssertWasCalled( s => s.SendEmail(null), options =>
    IgnoreArguments());
```

# Obiekty izolujące



```
class SystemMonitor {
    private FileSystemProvider fileSystemProvider;
    private MailSender sender;
    public SystemMonitor(newSender, newFSProvider) { ...
    }
    public void StartMonitoring(...)
    {
        ...
        while(...) {
            ...
            if (fileSystem.FileExists("myFile"))
                sender.SendEmail(...)
        }
    }
}
```

# Mock - obiekty izolujące



```
var fakeMailer =  
    MockRepository.GenerateMock<MailSender>();  
var fakeFSProvider =  
    MockRepository.GenerateStub<FileSystemProvider>();  
fakeMailer.Stub(mailer => s.SendEmail(null))  
    .IgnoreArguments();  
fakeFSProvider.Stub(fsProvider => s.FileExist (null))  
    .IgnoreArguments().Return(true);  
var sut = new SystemMonitor(fakeMailer, fakeFSProvider);  
  
sut.StartMonitoring();  
  
fakeMailer.AssertWasCalled( s => s.SendEmail(null),  
    options => IgnoreArguments());
```





# Problemy

- Kod: Nadużycie mocków
- Kod: Nadmierna specyfikacja



# Testy sterowane danymi

# Testy sterowane danymi

- Pojedynczy kod testu (parametryzowany)
- Test jest uruchamiany wielokrotnie dla różnych zestawów danych
- Dane dla testu mogą być umieszczone w kodzie lub brane z zewnętrznych źródeł (txt, xml, csv, xls, mdb itd.)
- **UWAGA: to nie jest srebrna kula**
  - Słaba diagnostyka
  - Tendencja do testowania kombinatorycznego

# Testy sterowane danymi MSTest

```
[TestClass]
public class TestClass
{
    [TestMethod]
    [DeploymentItem("FPNWIND.MDB")]
    [DataSource("System.Data.OleDb",
        "Provider=Microsoft.Jet.OLEDB.4.0;\
Data Source=\"FPNWIND.MDB\"",
        "Employees", DataAccessMethod.Sequential)]
    public void TestMethodThatWritesOnlyTestCases()
    {
        Console.WriteLine("EmplID:{0} LastName: {1}",

            TestContext.DataRow["EmployeeID"],
            TestContext.DataRow["LastName"]);
    }
}
```

# Testy sterowane danymi NUnit

```
[TestCase(2.5d, 2d, Result=1.25d)]  
[TestCase(-2.5d, 1d, Result = -2.5d)]  
public double ValidateDivision(double numerator,  
                                double denominator)  
{  
    var myClass = new MyClass();  
    return myClass.Divide(numerator, denominator);  
}
```



# Jak się uczyć

1. Czytać, oglądać, słuchać
2. Pisać testy (TDD/BDD)
3. Eksperymentować
4. Kata
5. Coding dojo
6. Dyskusje nad kodem, code review, pair programming
7. Doskonalić się

# Narzędzia

## ■ Frameworki UT:

- NUnit, MSTest, Mspec

## ■ Mockowanie

- RhinoMock, Moq, FakeItEasy, NSubstitute

## ■ Code Kata

- <http://codekata.com/>
- <http://www.codekatas.org/>
- <http://www.codingdojo.org/cgi-bin/index.pl?KataCatalogue>

## ■ Coding Dojo

- <http://codingdojo.org/>