Tworzenie dobrego kodu dla niewtajemniczonych

Czyli jak pisać czytelniejszy, testowalny i łatwiejszy w utrzymaniu kod

Marcin Zajączkowski
m.zajaczkowski@gmail.com

Warszawa, 2011-06-11

Ostrzeżenie

Głębsze zapoznanie się z zasadami kryjącymi się pod pojęciem Software Craftsmanship, szczególnie lektura książki „Clean Code” może wywołać **nieodwracalne zmiany w mózgu**.

Jeżeli **nie jesteś** na to gotowy/a **nie oglądaj** tej prezentacji. Autor **nie bierze** odpowiedzialności za komplikacje zawodowe, jakie może ona za sobą pociągnąć.

Plan prezentacji

- Software Craftsmanship vs. Agile
- Dlaczego warto
- Clean Code – książka z zasadami
- Wpływ na wytwarzany kod

Dwa słowa o sobie
w świetle IT

- Z komputerami od dziecka
- 8+ lat zawodowego programowania
- 5+ lat z technologią Java
- Architekt Java
- Zwolennik metodyk zwinnych
- Pod wrażeniem Software Craftsmanship i TDD
- Miłośnik FOSS i Linuksa

Cel prezentacji

Spełnić oczekiwania
jak największej liczby osób
zgromadzonych na prezentacji

Cel prezentacji
Sposób realizacji

- Zapoznanie z koncepcjami stojącymi za Software Craftsmanship
- Przegląd wybranych aspektów opisanych w książce „Clean Code”, jako znana wszystkim podstawa do podnoszenia jakości wytwarzanego kodu
- Zachęcenie do przeczytania książki i stosowania na co dzień dobrych praktyk
  => **zachęcenie do tworzenia dobrego kodu**

Manifest Agile
Ogólnie

- Zima 2001
- Kilkanaście osób promujących nowe podejście do tworzenia oprogramowania
- Próba spisania zasad łączących różne lekkie metodyki (formalizujące się w latach 90-tych)
- 4 główne założenia
- 12 zasad uszczegóławiających

Manifest Agile
Główne założenia

- **Osoby i interakcje**
  nad procesy i narzędzia
- **Działające oprogramowanie**
  nad wyczerpującą dokumentacją
- **Współpraca z klientem**
  nad negocjacją kontraktu
- **Reagowanie na zmiany**
  nad podążanie według planu

Manifest Software Craftsmanship
Ogólnie

- Formalnie marzec 2009
- Wcześniej lata stosowania
- Próba odniesienia się do aspektów tworzenia oprogramowania, których Agile nie precyzuje
- Rozwinięcie czterech zasad z manifestu Agile

Manifest Software Craftsmanship
Główne założenia

- Nie tylko działające oprogramowanie, ale również **dobrze napisane oprogramowanie**
- Nie tylko reagowanie na zmiany, ale również **ciągłe zwiększanie wartości**
- Nie tylko osoby i interakcje, ale również **społeczność profesjonalistów**
- Nie tylko współpraca z klientem, ale również **produktywne partnerstwo**

Tworzenie kodu vs. tworzenie dobrego kodu

- Różnica w myśleniu
- Inny poziom wewnętrznego zadowolenia
- Potrzeba większego zaangażowania
- Inne wymagania i oczekiwania
- Dolny poziom poniżej którego nie można zejść - dobry specjalista nie robi bubli

Clean Code
A Handbook of Agile Software Craftsmanship

- Zbiór zasad, dzięki którym kod ma szansę być:
  - czytelniejszy
  - testowalny
  - łatwiejszy w utrzymywaniu
- Robert C. Martin (a.k.a. Uncle Bob) i inni
- Niezbędnik dla każdego kto chce pisać **dobry** kod

Informacja

Dalsza część prezentacji jest w dużej mierze oparta na zasadach przedstawionych w książce „Clean Code” autorstwa Robeta C. Martina i innych.

Jej celem jest szybkie zapoznanie nowych osób z z zasadami, które mogę diametralnie wpłynąć na jakość tworzonego kodu.

Słuchacze, którzy czytali tę książkę mogą odnieść wrażenie efektu déjà vu.

Dobre nazewnictwo

- Nazwy oddające znaczenie
- Jeżeli potrzebny komentarz to nazwa nie jest najlepsza
- Długa czytelna nazwa jest lepsza od krótkiej enigmatycznej
- Opisowa długa nazwa jest lepsza od opisowego długiego komentarza

Dobre nazewnictwo
Przykłady

```
int s;      //elapsed time in seconds

int seconds;

int durationInSeconds;
```

Nazewnictwo
Przykład[1]

- Kod nie musi być długi, aby być nieczytelny

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

- Co robi ta metoda?

[1] – przykład zapożyczony z książki „Clean Code”, Robert C. Martin, 2008

Nazewnictwo
Przykład – c.d.

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

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

Nazewnictwo
Przykład – wpływ zmiany nazw

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

- Co zawiera lista theList?
- Czym wyróżnia się pierwszy element w liście?
- Dlaczego wartość 4 jest kluczowa?
- Co właściwie jest zwracane z metody?

Nazewnictwo
Przykład – dalszy refaktoring

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

- Jawne nazywanie struktur również zwiększa czytelność

Nazwy klas

- Rzeczownik lub grupa rzeczowników
- Przykłady:
  - Employee, WavPlayer, PostalAddressDecoder
- Do unikania:
  - Manager, Processor, Data, Info
- Łatwo wyszukiwalne nazwy

Nazwy metod

- Czasownik lub fraza z czasownikiem
  - save, removePage, sendEmail
- Akcesory i mutatory[1] zgodnie z konwencją javabean:
  - setDataSource, getName, isNegative

[1] – po polsku gettery i settery

Spójność nazw
w całym kodzie

- Jedna spójna nazwa w całym kodzie
- Łatwiej znaleźć konkretną nazwę
- Zły przykład:
  - **get**AnimalsByCode
  - **fetch**AnimalsByName
  - **retrieve**AnimalsByType

Nazewnictwo

- Warto używać słownictwa biznesowego
- Gdy nazwa jest zła:
  - Nieintuicyjna
  - Niespójna
  - Posiada literówkę

  to trzeba ją zmienić (IDE -> zmień nazwę)
- Dobre nazwy zwiększają czytelność kodu

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

Czytelność metod
Przykład

```
@Override
public void importOrders() {
    DateTime date = new DateTime();

    List<SalesOrderExportDetail> orders = salesOrderWS.getSalesOrdersFromWebService();
    SalesOrderImport salesImport = new SalesOrderImport();
    for(SalesOrderExportDetail salesOrder : orders) {
        try {

            Order order = new Order();
            order.setValue(new BigDecimal(salesOrder.getValue(), new MathContext(2, RoundingMode.HALF_EVEN)));
            order.setAddress(convertToAddress(salesOrder.getShippingAddress()));
            order.setBuyerAddress(convertToAddress(salesOrder.getBuyerAddress()));

            double vat = calculator.vatRate(order.getAddress());
            order.setVatRate(vat);
            BigDecimal value = calculator.value(vat);
            order.setVatValue(value);

            Currency curr = order.getCurrency();
            if(curr != Currencies.PLN) {
                order.setPln(converter.convert(curr, order.getValue()));
            }

            orderRepository.save(order);
            salesImport.imported(order);
        } catch(Exception e) {
            salesImport.notImported(e);
        }
    }
    salesImport.setDate(date);
    orderRepository.save(salesImport);
}
```

[1] – przykład zaczerpnięty z bloga Jakuba Nabrdalika Solid Craft - http://blog.solidcraft.eu/

Czytelność metod
Przykład – po pierwszym spojrzeniu

- Pierwsze wrażenie
  - Zbyt duża metoda – 30+ linii
  - Podział na bloki kodu – prawdopodobnie powiązane funkcjonalnie operacji
- Dostrzeżone problemy:
  - Konieczność przeczytania całej metody przed określeniem zakresu jej funkcjonalności
  - Łamanie zasady pojedynczej odpowiedzialności

Czytelność metod
Przykład – pierwszy krok zmian

- Określenie i spisanie funkcjonalności poszczególnych bloków

```
@Override
public void importOrders() {
//create date before importing from web service
    DateTime date = new DateTime();

//import data transfer objects from web service
    List<SalesOrderExportDetail> orders =
                        salesOrderWS.getSalesOrdersFromWebService();

//create summary for importing orders
    SalesOrderImport salesImport = new SalesOrderImport();

//for each order try
    for(SalesOrderExportDetail salesOrder : orders) {
        try {
(...)
```

```
(...)
//convert it to domain object
            Order order = new Order();
            order.setValue(new BigDecimal(salesOrder.getValue(),
                    new MathContext(2, RoundingMode.HALF_EVEN)));
            order.setAddress(convertToAddress(salesOrder.getShippingAddress()));
            order.setBuyerAddress(convertToAddress(salesOrder.getBuyerAddress()));

//calculate vat
            double vat = calculator.vatRate(order.getAddress());
            order.setVatRate(vat);
            BigDecimal value = calculator.value(vat);
            order.setVatValue(value);

//calculate value in PLN
            Currency curr = order.getCurrency();
            if(curr != Currencies.PLN) {
                order.setPln(converter.convert(curr, order.getValue()));
            }

//save in repository
            orderRepository.save(order);

//prepare information whether it was succesfull
            salesImport.imported(order);
        } catch(Exception e) {
            salesImport.notImported(e);
        }
    }
//save the summary for importing orders
    salesImport.setDate(date);
    orderRepository.save(salesImport);
}
```

Czytelność metod
Przykład – drugi krok zmian

- Zamiana opisanych bloków na wydzielone metody, na przykład:

```
//calculate vat
    double vat = calculator.vatRate(order.getAddress());
    order.setVatRate(vat);
    BigDecimal value = calculator.value(vat);
    order.setVatValue(vat);
```

na metodę

```
private calculateVat() {
    double vat = calculator.vatRate(order.getAddress());
    order.setVatRate(vat);
    BigDecimal value = calculator.value(vat);
    order.setVatValue(vat);
}
```

z wywołaniem

```
order.calculateVat();
```

Czytelność metod
Przykład – drugi krok zmian

- Zamiana opisanych bloków na wydzielone metody (lub nowe klasy)
- Przeniesienie ciał pętli oraz bloku try..catch do osobnych metod
- Zapewnienie pojedynczej odpowiedzialności
- Poprawienie nazewnictwa

Czytelność metod
Przykład – rezultat – metody pomocnicze

```
private ImportSummary tryToImportEach(List<SalesOrderDto> salesOrderDtos,
                                      DateTime dateImportStartedAt) {
    ImportSummary importSummary = new ImportSummary(dateImportStartedAt);
    for(SalesOrderDto salesOrderDto : salesOrderDtos) {
        importSummary.add(tryToImport(salesOrderDto));
    }
    return importSummary;
}
private OrderImportOutcome tryToImport(SalesOrderDto salesOrderDto) {
    OrderImportOutcome orderImportOutcome = null;
    try {
        orderImportOutcome = importOne(salesOrderDto);
    } catch(Exception e) {
        orderImportOutcome = new OrderImportFailure(e);
    }
    return orderImportOutcome;
}
private OrderImportSuccess importOne(SalesOrderDto salesOrderDto) {
    Order order = orderConverter.toOrder(salesOrderDto);
    order.calculateVat();
    order.updatePlnValueIfNeeded();
    orderRepository.save(order);
    return new OrderImportSuccess(order);
}
```

Czytelność metod
Przykład – rezultat – główna metoda

- Tylko cztery linijki, które musimy przeczytać, aby dowiedzieć się, co robi metoda

```
@Override
public void importOrders() {
    DateTime dateImportStartedAt = new DateTime();
    List<SalesOrderDto> salesOrderDtos =
                salesOrderWebServiceClient.getSinceLastVisitOrderedByNumber();
    ImportSummary importSummary = tryToImportEach(
                salesOrderDtos, dateImportStartedAt);
    orderRepository.save(importSummary);
}
```

- W porównaniu do 30+ w oryginale

Czytelność metod
Przykład – oryginalna metoda

```
@Override
public void importOrders() {
    DateTime date = new DateTime();

    List<SalesOrderExportDetail> orders = salesOrderWS.getSalesOrdersFromWebService();
    SalesOrderImport salesImport = new SalesOrderImport();
    for(SalesOrderExportDetail salesOrder : orders) {
        try {

            Order order = new Order();
            order.setValue(new BigDecimal(salesOrder.getValue(), new MathContext(2, RoundingMode.HALF_EVEN)));
            order.setAddress(convertToAddress(salesOrder.getShippingAddress()));
            order.setBuyerAddress(convertToAddress(salesOrder.getBuyerAddress()));

            double vat = calculator.vatRate(order.getAddress());
            order.setVatRate(vat);
            BigDecimal value = calculator.value(vat);
            order.setVatValue(value);

            Currency curr = order.getCurrency();
            if(curr != Currencies.PLN) {
                order.setPln(converter.convert(curr, order.getValue()));
            }

            orderRepository.save(order);
            salesImport.imported(order);
        } catch(Exception e) {
            salesImport.notImported(e);
        }
    }
    salesImport.setDate(date);
    orderRepository.save(salesImport);
}
```

Argumenty metod

- Zero argumentów – największa czytelność
  - Metoda robi to, o czym mówi jej nazwa
- 1-2 argumenty – przydatne w określonych okolicznościach
- Trzy+ argumenty – mała czytelność
  - Konieczność dodatkowej analizy wywołania
  - Utrudnione pisanie testów – konieczność pokrycia wszystkich kombinacji
  - Często lepiej przekazać obiekt

Argumenty metod - c.d.

- Argumenty flagowe
  - ~~printFile(true)~~ – niejasne bez podglądu deklaracji metody
  - printFileWithHeader() i printFileWithoutHeader() plus wewnętrznie wydzielenie wspólnych części do osobnej metody

Efekty uboczne metod

- Źródło trudnych do wyśledzenia problemów
  - interrupted()
  - getFearFactor() - dokonujący w wyniku obliczeń zmiany pól danego obiektu (zamiast gettera)
- Parametry wyjściowe
  - ~~includeDetailsInProduct(Product product);~~
  - product.addDetails(getProductDetails());
  - Często lepiej wykonywać operacje na this
  - Utrudnia testowanie

Obsługa błędów

- Zwracanie pustej kolekcji zamiast null z metody
  - Upraszcza kod (bezpieczne dla „for each”)
- Nieprzekazywanie null metodzie
  - Wymusza dodatkową walidację
  - Trudne do kontroli
  - JSR 305 – dodatkowe adnotacje (np. @NotNull) mają zwiększyć skuteczność statycznej analizy kody

Obsługa błędów
Sprawdzanie wartości zwracanej

- 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");
}
```

Obsługa błędów
Korzystanie z wyjątków

- Wyjątki do obsługi sytuacji nietypowych

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

- Krótszy zapis
- Większa przejrzystość
- Lepsze oddzielenie logiki od obsługi błędów

Obsługa błędów
Dalsza separacja kodu do obsługi błędów

- try...catch zmniejsza czytelność – lepiej kod w środku wydzielić do osobnych metod

```
public void deleteRegistryValueWithKeys() {
  try {
    internalDeleteRegistryValueWithKeys();
  } catch(RegistryException e) {
    log.error(e, e);
  }
}

private void internalDeleteRegistryValueWithKeys()
      throws RegistryException {
  deleteRegistryValue(value);
  deleteRegistryKey(subkey);
  deleteRegistryKey(key);
}
```

Komentarze

„Don't comment bad code – rewrite it” [1]

- Często są wynikiem niedostatecznie czytelnie napisanego kodu
- Mogą nie być aktualne
  - Refaktoring zazwyczaj nie dostosowuje komentarzy
  - Rzadko ktoś poza autorem modyfikuje/usuwa komentarze
- Kod powinien sam się dokumentować

[1] - B. Kernighan i P. Plaugher

Komentarze
Przykład

- Zazwyczaj lepiej poprawić kod niż pisać komentarze

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

vs.

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

Komentarze
Czasem się przydają

- Wyjaśnienie zachowania

```
public String formatDateToYYYYMMDD(Date date) {
  //SimpleDateFormat is not thread safe,
  //new instance has to be created every time[1]
  SimpleDateFormat df = new SimpleDateFormat(YYYYMMDD);
  return df.format(date);
}
```

- Dokumentacja zewnętrznego API
- Informacje o licencji na jakiej jest kod
- TODO – w sytuacjach, gdy w danej chwili nie można czegoś poprawić (ale z umiarem!)

[1] – synchronizowanie metod jest innym podejściem

Komentarze – c.d.
Często warto unikać

- Nic nie wnoszące

```
i++; //increment i
```

- Wymuszone standardami

```
/**
* Returns logger.
*
* @return logger
*/
protected Logger getLogger() {
   return logger;
}
```

Komentarze – c.d.

- Błędne komentarze
  - Nieaktualne
  - Wprowadzające w błąd przez niewiedzę autora
- Wykomentowany fragment kodu
  - Zaciemnia czytanie kodu
  - Najlepiej usunąć
  - SCM zawsze pozwoli do niego wrócić

Klasy
Zalecenia

- Możliwie małe
- Pojedyncza odpowiedzialność (SRP[1])
  - Do przedstawienia w jednym prostym zdaniu w JavaDoc do klasy
  - „i”, „lub”, „jeżeli” w opisie są podejrzane
- Bez duplikacji
- Dobrze nazwane

[1] – Single Responsibility Principle – http://en.wikipedia.org/wiki/Single_responsibility_principle

Klasy
Niepokojące objawy

- Klasa bazowa wie o klasach potomnych
- Udostępnianie informacji o swoich strukturach wewnętrznych
- Korzystanie ze zbyt wielu klas
- Słaba spójność[1]
- Łamanie Prawa Demeter[2]
- Nadmiarowa statyczność metod

[1] – Cohesion – http://en.wikipedia.org/wiki/Cohesion_(computer_science)
[2] – Law of Demeter - http://en.wikipedia.org/wiki/Law_of_Demeter

Klasy
Zwiększenie elastyczności rozwiązania

- Praca na interfejsach
- Preferowanie wstrzykiwania wykorzystywanych obiektów (DI[1]) ponad ich samodzielne tworzenie w kodzie (poprzez operator new)
- Separacja logiki operacji od zadań dodatkowych – na przykład z AOP
- Zasada otwarte-zamknięte[2]

[1] - Dependency Injection - http://en.wikipedia.org/wiki/Dependency_Injection
[2] - Open/closed principle - http://en.wikipedia.org/wiki/Open/closed_principle

Refaktoring

- Dobry refaktoring nie jest zły :)
- Zasada skauta[1] -
  - Pozostaw kod, który czytasz w lepszej stanie niż był, gdy go zastałeś
- Gdy coś źle wygląda warto poprawić *
- Nie należy obawiać się zmian – silne testy jednostkowe pokażą, czy kod nadal działa

[1] - The boy scout rule - „Clean Code” Robert C. Martin, 2008

Test Driven Development[1]
Kluczowe do tworzenia dobrego kodu

- Nie o pisanie testów w tym naprawdę chodzi
- Testy tworzone najpierw wymuszają konstrukcje, które są łatwo testowalne
- Dodaje pewności przy dokonywaniu zmian
- Powinny być łatwo uruchamialne („jeden przycisk”)
- Powinny działać możliwie szybko (sekundy)
- Więcej w osobnej prezentacji

[1] – TDD - http://en.wikipedia.org/wiki/Test-driven_development

Podsumowanie

- Kod jest tworzony dla innych programistów[1]
- Kod jest czytany kilka razy częściej niż następuje jego modyfikacja – warto go raz, a dobrze napisać
- Niedbanie o kod to równia pochyła do nieutrzymywalnego projektu, którym każdy się brzydzi i wymaga przepisania od nowa

[1] - Oprogramowanie jest pisane dla klienta, ale większość klientów nie przejmuje się wyglądem kodu

Literatura

- „Clean Code: A Handbook of Agile Software Craftsmanship”, Robert C. Martin i inni, 2008
- „The Pragmatic Programmer”, Andrew Hunt i David Thomas, 1999
- „Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman”, Dave Hoover, Adewale Oshineye, 2009
- „Test Driven Development: By Example”, Kent Beck, 2002

Grupy lokalne

- Warszawa-DP – Software Craftsmanship, TDD, Coding Kata, OCR, wzorce projektowe

  http://groups.google.com/group/warszawa-dp/

- Agile Warsaw – Agile i tematy pokrewne

  http://agilewarsaw.com/

- Warszawa JUG – Java, JVM i nie tylko

  http://www.warszawa.jug.pl/

Blogi tematyczne

- http://blog.solidcraft.eu/
- http://agile.waw.pl/
- http://art-of-software.blogspot.com/
- http://studiopragmatists.blogspot.com/
- http://cleancoder.posterous.com/
- http://misko.hevery.com/
- http://solidsoft.wordpress.com/ - mój blog

Cel prezentacji
Czy został spełniony?

- Przedstawienie koncepcji stojących za Software Craftsmanship
- Przegląd wybranych aspektów opisanych w książce „Clean Code”, jako znana wszystkim podstawa do podnoszenia jakości wytwarzanego kodu
- Zachęcenie do przeczytania książki i stosowania na co dzień dobrych praktyk
  => **zachęcenie do tworzenia dobrego kodu**

[1] – Martin Golding – autor cytatu z następnego slajdu

Pytania?

„Always code as if the guy
who ends up maintaining your code
will be a violent psychopath
who knows where you live” [1]

Marcin Zajączkowski
m.zajaczkowski@gmail.com

http://solidsoft.wordpress.com/