Wykład III
Jakość kodu

Akademia ADB

Jak pisać kod?

Podstawowe zasady pisania kodu:

- **Kod piszemy raz, ale czytamy wiele razy**
- Należy pamiętać o tym, że widzą go też inni
- Należy stosować się do standardów kodowania języka/organizacji – przydatne będą narzędzia FxCop, JsLint itp.
- Istnieje wiele narzędzi wspomogających wykrywanie złych praktyk (wysokiej złożoności cyklomatycznej, copy-paste, martwych ścieżek itd.)
- Poznaj swoje IDE

Przegląd kodu

Zespoły programistyczne bardzo często stosują przegląd kodu:

- Dzięki niemu wykrywa się trochę bugów
- Sprawdzenie czy kod jest do utrzymania/czytelny
- Więcej osób zna tajniki każdej części (tzw. *bus effect*)
- Możemy się uczyć od siebie nawzajem
- Powinien on mieć miejsce często
- Należy pamiętać o czynniku afektywnym

Logiczne grupowanie kodu

Kod będzie czytelniejszy jeśli logicznie powiązane elementy będą blisko siebie:

- W przestrzeniach nazw
- W klasach
- W metodach/funkcjach

Zachowywanie wcięcia

Zachowujcie wcięcia!

```
if (is_array($users)) {
echo "<table border>";
foreach ($users as $user) {
$sr      = ldap_search($ds, $basedn, "uid=".$user['login'], array('o'));
$info    = ldap_get_entries($ds, $sr);
$country = strlen($info[0]['o'][0])>0 ? $info[0]['o'][0] : 'Poland';
echo $country;
}
echo "</table>";
}
```

Zachowywanie wcięcia

Zachowujcie wcięcia!

```
if (is_array($users)) {
   echo "<table border>";
   foreach ($users as $user) {
        $sr     = ldap_search($ds, $basedn, "uid=".$user['login'],
   array('o'));
        $info   = ldap_get_entries($ds, $sr);
        $country = strlen($info[0]['o'][0])>0 ? $info[0]['o'][0] :
   'Poland';
        echo $country;
    }
    echo "</table>";
}
```

Dobre komentarze

Zasady:

- Komentujcie to, co ma uzasadnienie
- Nie ma sensu komentować oczywistości

```
function computeFactor($f1, $f2) {
  return $f1 + $f2 // zwracamy sumę dwóch składników
}
```

Jasne oznaczanie bloków kodu

Zawsze używajcie nawiasów, aby oznaczyć blok kodu – nawet jeśli składa się z jednej instrukcji:

```
foreach($vals as $key=>$val)
        if(!in_array($key,$this->_entryObject->availableAttributes)) {
                unset($vals[$key]);
}
```

```
foreach($vals as $key=>$val) {
        if(!in_array($key,$this->_entryObject->availableAttributes)) {
                unset($vals[$key]);
        }
}
```

Instrukcje warunkowe

Przy stosowaniu instrukcji warunkowych warto pamiętać o:

- umieszczaniu typowych przypadków jako pierwszych

```
if ($sum > 5000 || $this->isVip()) {
```

- używaniu raczej pozytywnych warunków

```
if (!$this->notInAdminGroup()) {
```

- unikaniu skomplikowanych testów zastępując je wywołaniami funkcji

```
if ($balance > 3000 && $balance < 7000) {
if ($this->isInMiddleRange()) {
```

Instrukcje warunkowe

Przy stosowaniu instrukcji warunkowych warto pamiętać o:

- sprawdzaniu pokrycia wszystkich przypadków
- użyciu switch i uporządkowaniu przypadków od najmniejszego do największego lub alfabetycznie. Default powinien być ostatni.

```
switch ($a) {
    case 3: ...
    case 5: ...
    default: ...
    case 4: ...
}
```

Nazwa jest wszystkim

Prawidłowe nazywanie w kodzie jest podstawą czytelności

```
compute: function(i, v) {
            var s = 0;
            i.each(function(j) {
                var p = j.get('price');
                var q = j.get('quantity');
                s += p * q;
            });
            if (v) {
                s += parseInt(v) / 100 * s;
            }
        return s;
}
```

Nazwa jest wszystkim

Prawidłowe nazywanie w kodzie jest podstawą czytelności

```
computeGross: function(items, vat) {
            var sum = 0;
            items.each(function(i) {
                var p = i.get('price');
                var q = i.get('quantity');
                sum += p * q;
            });
              if (vat) {
                sum += parseInt(vat) / 100 * sum;
            }
      return sum;
}
```

Jak stworzyć dobrą klasę?

Zasady tworzenia dobrych klas :

- Klasy powinny być niewielkie i zajmować się tylko jedną rzeczą (*Single Responsibility Principle*)
- Należy chować jak najwięcej elementów składowych
- Nie należy upubliczniać pól
- Nazwy metod i klas powinny być czytelne
- Należy unikać „magicznej wiedzy" o kolejności wywołań metod
- Warto stosować abstrakcyjne klasy bazowe
- Nie można tworzyć „boskich klas"
- Metody statyczne powinny być używane z rozwagą
- SOLID

Relacje między klasami

W zasadzie występują dwie relacje między klasami – jest i ma:

- IS-A realiozowania przez dziedziczenie od ogółu do szczegółu
- HAS-A realizowane jako kompozycja, przy czym należy ją prywatyzować. Zasada substytucji Liskov określa, że nie wolno nadużywać tego narzędzia.
- Hierarchia nie powinna być zbyt głęboka
- Wspólne elementy powinny być umieszczone wysoko w hierarchii
- Wielokrotne użycie `switch` wskazuje na możliwość zastosowania polimorfizmu

Jak napisać dobrą metodę/funkcję?

Dobra funkcja:

- Jest prosta: nie może być długa i mieć zbyt wiele zagnieżdżeń
- Jeśli coś się powtarza w kilku miejscach, to może być to funkcja.
- Pamiętajcie o dobrych nazwach
- Maksymalna liczba parametrów to 7

Nazwy funkcji - antyprzykłady

Takich rzeczy nie róbcie:

- reportGeneration() – nie jest czasownikiem
- reportGenerationAndPriniting – długa. Wygląda na to, że to powinny być dwie metody.
- printing3() – zawiera cyfrę
- printNew() – co oznacza new?

Zmienne

Zasady użycia zmiennych:

- Zmienne powinny być dobrze nazwane
- Deklaracja powinna być blisko miejsca użycia i o jak najmniejszym zakresie
- Nie używajcie zmiennej do dwóch różnych celów
- Deklaracja i inicjalizacja powinny być w tym samym miejscu
- Blokujcie możliwość zmiany wartości jeśli język na to pozwala, a okoliczności - wymagają

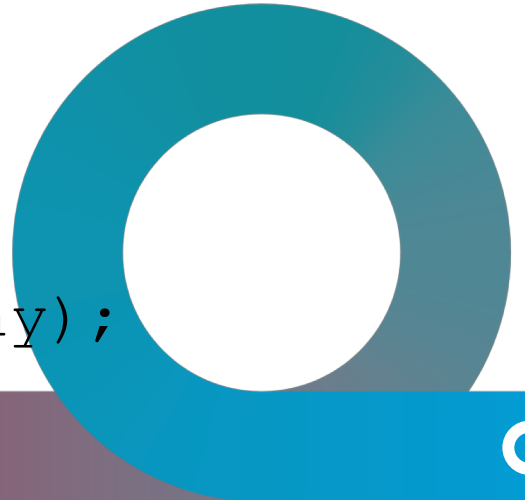
Testy jednostkowe

Obecnie każdy język posiada narzędzie do testów jednostkowych z rodziny xUnit, ponieważ testy automatyczne:

- są szybsze i dokładniejsze
- pozwalają na pewniejsze wprowadzanie zmian
- promują lepszą architekturę systemu
- mogą stanowić dokumentację systemu

Przykład testu jednostkowego

```
[TestFixture]
public class CalculatorTests
{
    [Test]
    public void TestFunctioning()
    {
        Calculator calculator = new
Calculator();
        calculator.Enter(2);
        calculator.PressPlus();
        calculator.Enter(2);
        calculator.PressEquals();
        Assert.AreEqual(4, calculator.Display);
    }
}
```



Refaktoryzacja

Refaktoryzacja to zmiana struktury kodu mająca na celu nie zmianę zachowania, ale polepszenie jego struktury.

- Zwykle dokonuje się tej operacji w następujących po sobie mikrokrokach – widać iteracyjność charakteryzującą agile
- Celem są zwiększenie czytelności i zmniejszenie skomplikowania – spłacamy dług techniczny
- Kompleksowa refaktoryzacja przynosi efekt tylko w połączeniu z testami jednostkowym
- Wiele IDE posiada komendy dokonujące takich zmian

Przykłady mikrorefaktoryzacji

- Zmień nazwę metody
- Zamiana warunków na polimorfizm
- Ekstrakcja klasy
- Ekstrakcja metody
- Podciągnij (metodę w hierarchii klas)
- Opuść (metodę w hierarchii klas)

Integracja ciągła *(continuous integration)*

Integracja ciągła to kolejna technika kojarzona z Agile:

- Sprowadza się do automatycznego budowania systemu w określonym momencie, zwykle po każdym commicie
- System integracji ciągłej (np. Atlassian Bamboo) uruchamia skrypt budujący
- Skrypt budujący zwykle uruchamia testy jednostkowe
- Wyniki budowania oraz artefkaty są zachowywane w systemie

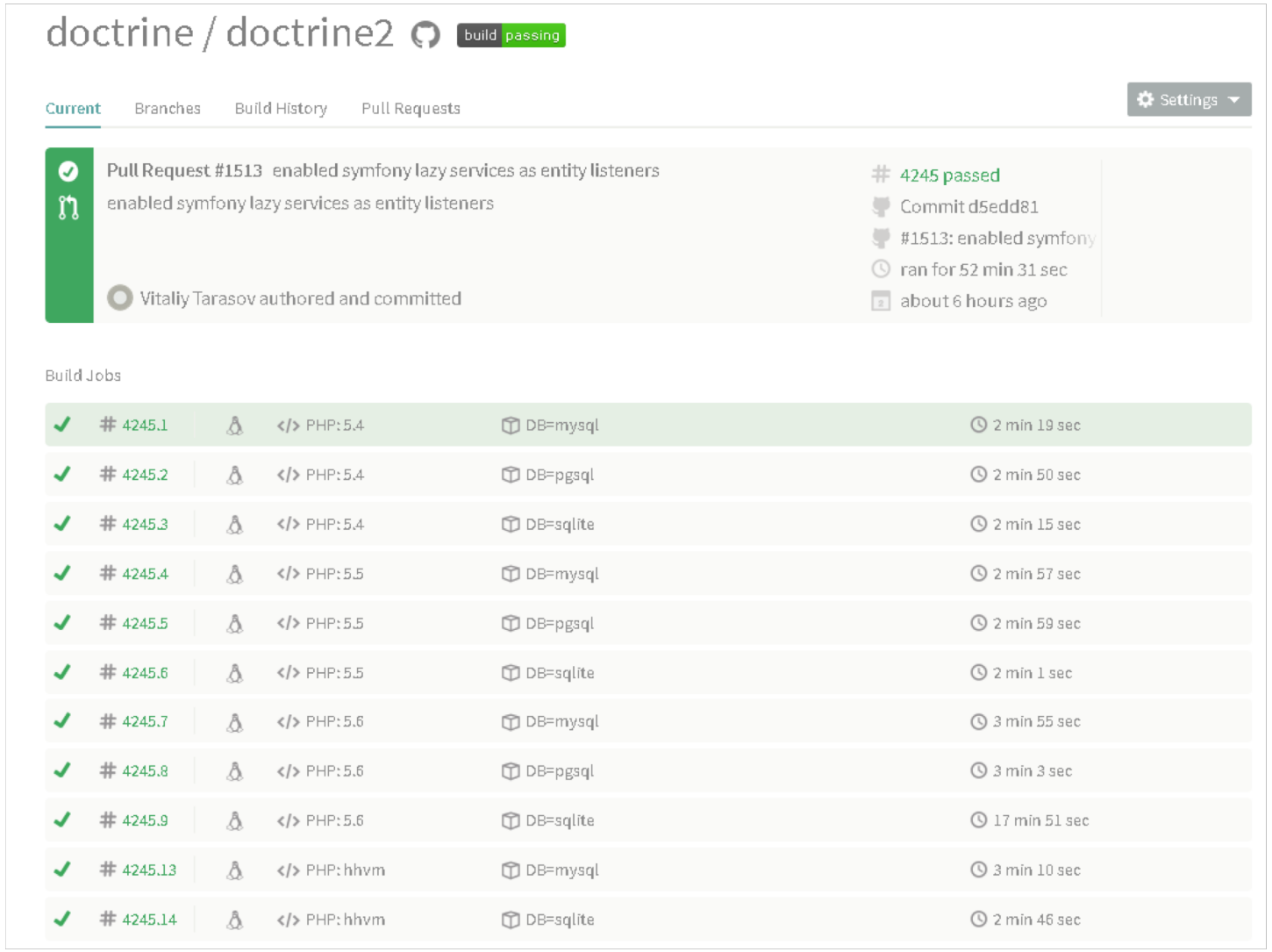
Integracja ciągła - korzyści

Korzyści wynikające ze stosowania CI:

- Błędy integracyjne są natychmiast widoczne
- Stale widoczny stan oprogramowania – bugi widoczne jak na dłoni
- Unikamy paniki przed terminem wydania
- Stale dostępne artefakty
- Programiści zwracają większą uwagę na jakość

Niestety, czas potrzebny do wdrożenia systemu jest znaczny.

Przykład CI


doctrine / doctrine2
build passing
Current
Branches
Build History
Pull Requests
Settings
Pull Request #1513 enabled symfony lazy services as entity listeners
enabled symfony lazy services as entity listeners
Vitaliy Tarasov authored and committed
4245 passed
Commit d5edd81
#1513: enabled symfony
ran for 52 min 31 sec
about 6 hours ago
Build Jobs
4245.1 PHP: 5.4 DB=mysql 2 min 19 sec
4245.2 PHP: 5.4 DB=pgsql 2 min 50 sec
4245.3 PHP: 5.4 DB=sqlite 2 min 15 sec
4245.4 PHP: 5.5 DB=mysql 2 min 57 sec
4245.5 PHP: 5.5 DB=pgsql 2 min 59 sec
4245.6 PHP: 5.5 DB=sqlite 2 min 1 sec
4245.7 PHP: 5.6 DB=mysql 3 min 55 sec
4245.8 PHP: 5.6 DB=pgsql 3 min 3 sec
4245.9 PHP: 5.6 DB=sqlite 17 min 51 sec
4245.13 PHP: hhvm DB=mysql 3 min 10 sec
4245.14 PHP: hhvm DB=sqlite 2 min 46 sec

Do poczytania

- R.C. Martin, *Czysty kod. Podręcznik dobrego programisty*
- S. McConnell, *Code Complete: A Practical Handbook of Software Construction, Second Edition*
- A. Hunt i D. Thomas, *JUnit. Pragmatyczne testy jednostkowe w Javie*
- M. Fowler i inni, *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*
- Martin Fowler o integracji ciągłej http://martinfowler.com/articles/continuousIntegration.html

Koniec

Dziękuję za uwagę

adbglobal.com