

Praktyczny wstęp do wytwarzania systemów oprogramowania, które zmieniają bieg historii

Sprawdź, jak robią to specjaliści

**NOKIA**



Z wielką przyjemnością przedstawiam Wam książkę „Praktyczny wstęp do wytwarzania systemów oprogramowania, które zmieniają bieg historii. Sprawdź, jak robią to specjaliści.” poświęconą tematyce najnowocześniejszych technologii telekomunikacyjnych.

Jest to książka o tyle dla nas ważna, iż jej autorami są nasi pracownicy – specjaliści z Europejskiego Centrum Oprogramowania i Inżynierii Nokia Networks we Wrocławiu. W tym miejscu pragnę im serdecznie podziękować nie tylko za merytoryczny wkład, ale również za ich entuzjazm, pasję oraz niezastąpione zaangażowanie w rozwój naszego wrocławskiego Centrum.

Książka, którą macie w rękach składa się z 22 artykułów poświęconych w całości tematyce rozwoju oprogramowania oraz innowacjom we wspomnianym obszarze. Co więcej, książka „Praktyczny wstęp do wytwarzania systemów oprogramowania, które zmieniają bieg historii. Sprawdź, jak robią to specjaliści.” stanowi swego rodzaju podróż pozwalającą czytelnikowi poznać arkana pracy naszych specjalistów. Problematyka najnowszych osiągnięć programistycznych widziana okiem praktyków oraz pragmatyczny styl artykułów to atuty tej publikacji, które niewątpliwie przyciągną wielu pasjonatów telekomunikacji.

Życzę Wam przyjemnej lektury,

**Bartosz Ciepluch**

Dyrektor Europejskiego Centrum Oprogramowania i Inżynierii  
Nokia Networks we Wrocławiu

# Programowanie w C++ i nie tylko

4

6	<b>1.1 — Sławomir Zborowski</b>
16	<b>1.2 — Michał Bartkowiak</b>
22	<b>1.3 — Artur Kochowski</b>
28	<b>1.4 — Robert Matusewicz</b>
32	<b>1.5 — Łukasz Grządko</b>

Praktyczny C++ 14  
Na terytorium kompilatora: libclang  
Wprowadzenie do biblioteki STL  
C++ Lambda  
Zapytania o ekstremum w przedziale

38

40	<b>2.1 — Wojciech Razik</b>
44	<b>2.2 — Kamil Pawłowski</b>
50	<b>2.3 — Marcin Zawadzki</b>
56	<b>2.4 — Marcin Załuski</b>
60	<b>2.5 — Tomasz Drwięga</b>

System kontroli wersji: Git czy SVN?  
Zastosowanie systemu operacyjnego Linux w firmie Nokia Networks  
Statyczna/dynamiczna analiza kodu  
Wyrażenia regularne  
Jak przeszliśmy od Javy do JavaScript?

64

66	<b>3.1 — Krzysztof Bulwiński</b>
72	<b>3.2 — Wojciech Pisarski</b>
78	<b>3.3 — Krzysztof Matuszek</b>
88	<b>3.4 — Michał Rudowicz</b>
94	<b>3.5 — Maciej Jaskot</b>
100	<b>3.6 — Andrzej Lipiński</b>
106	<b>3.7 — Marcin Gudajczyk</b>

Dobre praktyki inżynierii oprogramowania  
Testowanie jednostkowe oprogramowania  
„Clean Design”  
Utrzymanie starego kodu dla zabawy i zysku  
Programowanie a bezpieczeństwo  
Software Configuration Management – czym jest i do czego służy?  
Continuous Integration – integracja oprogramowania po „tuningu”

# Narzędzia programistyczne

# Praktyka inżynierii oprogramowania

114

116	<b>4.1 — Artur Orzechowski i Marcin Miernik</b>
122	<b>4.2 — Marcin Domański</b>
128	<b>4.3 — Łukasz Gostkowski</b>
136	<b>4.4 — Piotr Malatyński</b>
142	<b>4.5 — Agnieszka Szufarska</b>

Parametry sygnałów radiowych  
Pomiar parametrów radiowych stacji bazowych LTE (eNodeB) przy  
użyciu wektorowego generatora sygnałowego i analizatora widma  
Sieć mobilna – cybernetyczne pole walki  
Testowanie systemu LTE w środowisku end-to-end  
Czego spodziewać się po systemie 5G?

# Technologie radiowe

# Programowanie w C++ i nie tylko

1.1

**Sławomir Zborowski**  
Praktyczny C++14

6

1.2

**Michał Bartkowiak**  
Na terytorium kompilatora: libclang

16

1.3

**Artur Kochowski**  
Wprowadzenie do biblioteki STL

22

1.4

**Robert Matusewicz**  
C++ Lambda

28

1.5

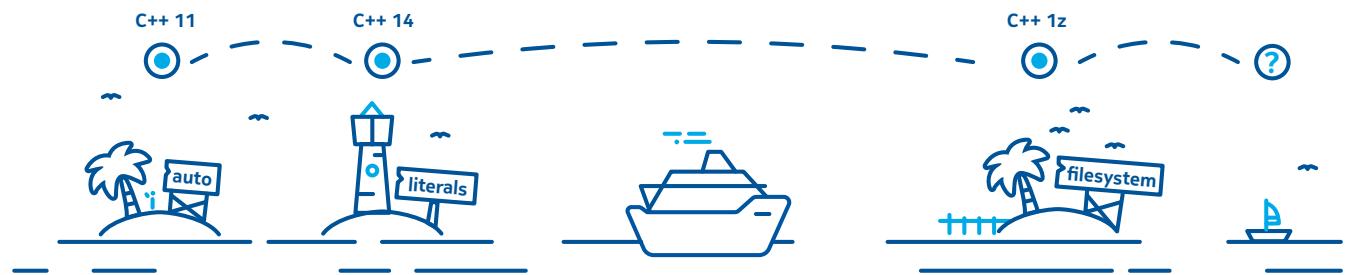
**Łukasz Grządko**  
Zapytania o ekstremum w przedziale

32

# Praktyczny C++14

Sławomir Zborowski  
Engineer, Software Development C++  
MBB SRAN OMCP CP 3

NOKIA



## Wstęp

Początki jednego z najbardziej rozpowszechnionych obecnie języków programowania, jakim jest język C++ sięgają lat 70. XX wieku [1]. W roku 1979 Bjarne Stroustrup rozpoczął prace nad wprowadzeniem obiektowości do królującego wówczas C. Inspiracja pochodziła z języka Simula 67. Nietrudno jednak zauważyć, że pierwszy standard języka C++ pochodzi z roku 1998. Dlaczego użytkownicy musieli czekać tak długo, skoro pierwszy komercyjny kompilator C++ został zaimplementowany już w roku 1985?

Odpowiedzią na to pytanie jest oczywiście proces standaryzacji. Można powiedzieć, że czas jego trwania stał się już legendą, po perturbacjach związanych z kolejną wersją standardu – C++0x. Cierpliwość się jednak opłaciła, ponieważ kolejne wersje standardu wprowadziły wiele oczekiwanych zmian oraz znaczco rozszerzyły bibliotekę standardową. Od tej chwili język C++ stał się innym językiem. Językiem, który pomimo zachowania wstępnej kompatybilności, wygląda znaczco inaczej oraz oferuje o wiele więcej w porównaniu z poprzednimi wersjami. Językiem, który w obecnych czasach może śmiało konkurować z pozostałymi językami dostępnymi na rynku.

“Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever[8].”

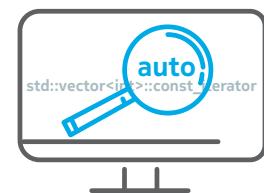
Bjarne Stroustrup

Podczas standaryzacji członkom komitetu przyświecało wiele celów, takich jak m.in. zwiększone bezpieczeństwo typów, zapewnienie wsparcia dla aplikacji wielowątkowych, ujednolicenie istniejących mechanizmów i ap. Jak się później okazało cele te zostały w pełni osiągnięte.

W niniejszym artykule przedstawione zostały te nowe elementy, które według autora znajdują najszerze zastosowanie w komercyjnych aplikacjach. Należy jednak pamiętać, że wszystkie wprowadzo-

ne konstrukcje i zmiany mogą okazać się praktyczne. Wszystko uzależnione jest od problemu, przed którym stoi programista.

W artykule autor opiera się na wersji standardu N3793 [0]. We wszystkich fragmentach kodu C++ zamieszczonych w niniejszym artykule, dla zwiększonej czytelności, pominięte zostały deklaracje dołączania (ang. include) standardowych plików nagłówkowych oraz założono użycie przestrzeni nazw std (**using namespace std**).



## auto

Zaletą (lub wadą, w zależności od zastosowania) języków dynamicznych jest łatwość posługiwania się wcześniej zadeklarowanymi zmennymi. Wynika to z faktu, że programista nie musi znać typu danej zmiennej. Przykładowo, aby wypisać wszystkie elementy z listy na ekran, w języku Python można się posłużyć kodem przedstawionym na [Listingu 1](#).

**Listing 1** Wypisywanie elementów listy na ekran – wersja w języku Python

```
elements = [1, 2, 3, 4, 5, 6] # Could be range(7) too
for element in elements:
    print str(element) + ',',
print
```

Jak kod wykonujący podobną czynność zazwyczaj wygląda w C++98? Ilustruje to [Listing 2](#).

#### [Listing 2](#) Wypisywanie elementów listy na ekran – wersja w języku C++

```
vector<int> elements = getElements(); // 1, 2, 3, 4, 5, 6

for (vector<int>::const_iterator it = elements.begin();
     it != elements.end(); ++it) {
    cout << *it << ",";
}

cout << endl;
```

W przypadku przedstawionego przykładu język dynamiczny zdecydowanie wygrywa. Kod jest bardziej przejrzysty oraz zwięzły. Jednym z czynników powodujących rozwlekłość kodu C++ jest konieczność podawania kompletnego typu zmiennej podczas deklaracji. W tym wypadku jest to deklaracja zmiennej kontrolnej pętli. W wielu przypadkach ma to znaczenie marginalne (np. dla typów `int`), jednak jeżeli typ jest „długi” to może to przyprawić programistę o ból głowy. Jest to problematyczne zwłaszcza, gdy pojawiają się szablony. W niektórych przypadkach (np. w niektórych implementacjach generycznych funkcji) typ wynikowy pewnych wyrażeń jest wręcz nieznany.

Problem ten został zauważony przez komitet standardizacyjny, który postanowił zmienić znaczenie słowa kluczowego `auto`. Wcześniej oznaczało ono zmienną automatyczną, czyli taką, której czas „życia” wyznaczany jest przez blok, w którym została zadeklarowana. Obecnie zleca ono dedukcję typu wyrażenia po prawej stronie kompilatorowi. Na [Listingu 3](#) zamieszczony został fragment kodu realizujący tą samą czynność co poprzednio z użyciem słowa kluczowego `auto`.

#### [Listing 3](#) Wypisywanie elementów listy na ekran – wersja w języku C++ z użyciem `auto`

```
auto elements = getElements(); // 1, 2, 3, 4, 5, 6

for (auto it = elements.begin(); it != elements.end(); ++it) {
    cout << *it << ",";
}

cout << endl;
```

Być może kod nie skrócił się znacząco, jednak jego przejrzystość z pewnością wzrosła. Zarówno osoba pisząca jak i czytająca taki kod zdają sobie sprawę, że wartością zwracaną przez metody `begin()` i `end()` jest jakiś iterator. Nikt z nich jednak nie potrzebuje znać do-

kładnego typu tegoż iteratora. Inaczej mówiąc słowo kluczowe `auto` może być pomocne w wielu sytuacjach, kiedy informacja o typie wyrażenia niekoniecznie jest istotna dla programisty. Należy jednak pamiętać, że `auto` dedukuje tylko wartości. Referencje oraz wskaźniki muszą być jawnie użyte z `auto` przez programistę przez zastosowanie odpowiedniego znaku `-&` lub `*`. [Listing 4](#) prezentuje potencjalnie niebezpieczną i często napotykana sytuację, do której może dojść w przypadku nieuwagi programisty.

#### [Listing 4](#) Przykład potencjalnie błędного wykorzystania słowa kluczowego `auto`

```
auto elements = getElements(); // 1, 2, 3, 4, 5, 6

// Getting first element by value!
auto firstElement = elements[0];

// elements vector is not updated!
firstElement = 0;
```

#### Pętle „po kolekcji”

Kolejną długo oczekiwana przez społeczność programistów C++ konstrukcją wprowadzoną wraz z standardem C++11 do języka jest pętla „po kolekcji” (ang. range-based for loop). Występuje ona od lat w innych językach – zarówno dynamicznych jak i statycznych. Problem, który adresuje to iteracja po całej kolekcji, która została przekazana jako parametr wejściowy. Nie wymaga zatem żadnych dodatkowych zmiennych kontrolnych reprezentujących indeks w tablicy itp. [Listing 5](#) ilustruje fragment kodu z [Listingu 3](#) z zastosowaniem pętli „po kolekcji”.

#### [Listing 5](#) Wypisywanie elementów listy na ekran z użyciem `auto` i pętli „po kolekcji”

```
auto elements = getElements(); // 1, 2, 3, 4, 5, 6

for (auto const element: elements) {
    cout << element << ",";
}
```

Z wykorzystaniem pętli „po kolekcji” kod staje się jeszcze bardziej zwięzły. Co również istotne, zwiększa się jego przejrzystość, ponieważ zlikwidowane zostają elementy implementujące iterację. Dodatkowym plusem jest automatyczna kompatybilność kontenerów biblioteki STL z nowymi pętlami.



#### Wyrażenia lambda

Wyrażenia lambda (czasami nazywane też funkcjami anonimowymi) to kolejna po słowie kluczowym `auto` i pętlach „po kolekcji” konstrukcja, która była długo wyczekiwana przez programistów języka C++. Pozwalają one na tworzenie funkcji/funktorów „w locie” oraz ich wołanie. Mają one wiele zastosowań. Zwłaszcza tam, gdzie wcześniej programiści zmuszeni byli podawać wskaźniki do funkcji wolnych lub ręcznie pisanych funkторów. Na [Listingu 6](#) zamieszczony został fragment kodu w C++98 odpowiadający za posortowanie kolekcji zawierającej instancje klasy `Employee` po polu `displayName`.

#### [Listing 6](#) Przykład sortowania kontenera, za pomocą wolnej funkcji jako komparatora

```
struct Employee {
    unsigned long id;
    string firstName;
    string lastName;
    string displayName;
};

bool compareByDisplayName(Employee const& lhs,
                         Employee const& rhs) {
    return lhs.displayName < rhs.displayName;
}

using Employees = vector<Employee>

Employees sortByDisplayName(Employees const& employees) {
    Employees result = employees;
    sort(result.begin(), result.end(), compareByDisplayName);
    return result;
}
```

Dzięki wyrażeniom lambda programiści mogą pominąć definicję funkcji `compareByDisplayName`, podając jej ciało bezpośrednio w wywołaniu funkcji `std::sort`. Jest to o wiele wygodniejsze oraz czytelniejsze od funkcji, funkторów bądź kombinacji `std::bind` z `std::less`. Zwłaszcza, jeżeli taki komparator występuje w aplikacji tylko jeden raz. [Listing 7](#) prezentuje funkcję `sortByDisplayName` z zastosowaniem generycznych wyrażeń lambda.

#### [Listing 7](#) Przykład sortowania kontenera z pomocą funkcji lambda

```
Employees sortByDisplayName(Employees const& employees) {
    auto result = employees;
    sort(result.begin(), result.end(),
          [] (auto const& lhs, auto const& rhs) {
              return lhs.displayName < rhs.displayName
          });
    return result;
}
```

Temat funkcji anonimowych w C++ jest bardzo obszerny. Z tego powodu szczegółowe informacje takie jak ich składnia oraz tajniki działania wraz z większą liczbą przykładów znajdują się w artykule "C++ Lambda" niniejszej publikacji.

#### Listy inicjalizacyjne

W zamieszczonych dotąd listingach przedstawiających wypisywanie elementów kolekcji na ekran wykorzystywana była funkcja `getElements()`, która w założeniu konstruuje i wypełnia wektor liczbami. Wraz z pojawiением się standardu C++11 programiści otrzymali jednak nowe narzędzie ułatwiające pracę – tzw. listy inicjalizacyjne (ang. initializer list), które rozwiązują problem tworzenia kolekcji „w miejscu”. Ilustruje to [Listing 8](#).

### **Listing 8** Wykorzystanie listy inicjalizacyjnej

```
vector<int> getElements() { // C++98 version
    vector<int> elements(6);
    elements.push_back(1);
    elements.push_back(2);
    elements.push_back(3);
    elements.push_back(4);
    elements.push_back(5);
    elements.push_back(6);
    return elements;
}

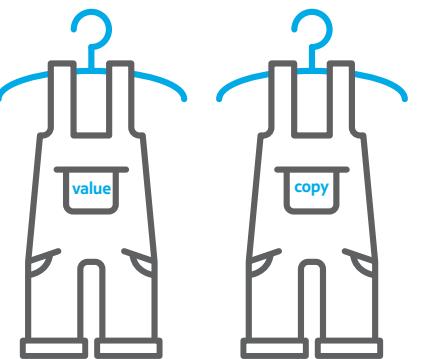
auto getElements() { // C++14 version
    return vector<int>{ 1, 2, 3, 4, 5, 6 };
}

void printElements() {
    vector<int> elements = getElements();
    // ... or directly:
    // vector<int> elements = { 1, 2, 3, 4, 5, 6 };

    for (auto const& element: elements) {
        cout << element << ",";
    }

    cout << endl;
}
```

Kontenery dostępne w bibliotece STL współpracują z listami inicjalizacyjnymi dla każdego typu, łącznie z typami zdefiniowanymi przez użytkownika. Również programiści mogą wykorzystywać listy inicjalizacyjne w konstruktach (i nie tylko konstruktorach) swoich klas. W tym celu użyć należy szablonu `std::initializer_list`. Przykładowe jego wykorzystanie ilustruje [Listing 9](#).



### **Listing 9** Przykładowe wykorzystanie szablonu `initializer_list`

```
struct Numbers {
    void feed(initializer_list<int> const& newNumbers) {
        size_t const newSize = numbers_.size() +
            distance(begin(newNumbers), end(newNumbers));

        numbers_.reserve(newSize);

        numbers_.insert(
            end(numbers_), begin(newNumbers), end(newNumbers));
    }

private:
    vector<int> numbers_;
};
```

#### Ujednoliciona inicjalizacja

Narastającym problemem podczas wprowadzania nowych funkcjonalności do języka C++ okazała się także inicjalizacja. Do istniejących już sposobów inicjalizacji dołączyła bowiem opisana w poprzedniej sekcji inicjalizacja za pomocą listy inicjalizacyjnej. Doprowadziło to zatem do wielu możliwych sposobów tworzenia obiektów:

- () – tworzenie obiektu za pomocą konstruktora klasy
- {} – inicjalizacja tablicy, klasy agregującej lub za pomocą listy inicjalizacyjnej
- brak – domyślny konstruktor

Mnogość rozwiązań nie była jednak jedynym problemem. Doskonale na to przykładem jest możliwość potraktowania przez kompilator inicjalizacji obiektu jako deklaracji funkcji. Ilustruje to [Listing 10](#).

### **Listing 10** Przykład niepoprawnej definicji zmiennej automatycznej

```
struct Field {
    Field() = default;
};

struct FieldWrapper {
    FieldWrapper(Field const& field);
};

int main() {
    FieldWrapper fieldWrapper(Field()); // Function declaration!
```

Intencją programisty w we wspomnianym przykładzie jest oczywiście stworzenie instancji klasy `FieldWrapper` za pomocą tymczasowego obiektu `Field`. Niestety, ponieważ podczas parsowania kodu źródłowego C++ kompilatory stosują metodę zachłanną (ang. most vexing parse)[0]/6.8, linia zostanie zinterpretowana jako definicja funkcji zwracającej obiekt `FieldWrapper` i przyjmującej jako argument bezargumentową funkcję zwracającą obiekt typu `Field`.

Te i inne problemy rozwiązane zostały z pomocą ujednoliconej inicjalizacji. Na [Listingu 11](#) znajdują się przykłady wykorzystania tej praktycznej metody.

### **Listing 11** Przykłady wykorzystania ujednoliconej inicjalizacji

```
// Normal variables and objects
int number { 5 };
string s { "Hello, world!" };

// Objects w/o c-tor
struct MyStruct {
    int number;
    string description;
};

MyStruct ms { 42, "Life, the universe and everything's" };

// Arrays
int numbers[] { 1, 2, 3, 4, 5, 6 };

// Containers
vector<float> floats { 1.f, 2.f, 3.f };

// Associative containers
enum class Color { White, Red, Blue, Black };
map<Color, std::string> mapping {
    {Color::White, "White"}, {Color::Red, "Red"}, {Color::Blue, "Blue"}, {Color::Black, "Black"}}
```

#### Bezpieczeństwo typów

C++ jest i zawsze był językiem, który obok wydajności stawał na bezpieczeństwo typów. Dzięki temu programiście bardzo cięźko jest popełnić błąd wynikający np. ze złego użycia dowolnego typu. Jak jednak wiadomo nie ma nic doskonałego na świecie. C++ nie jest wyjątkiem. W wydaniu C++98 było kilka nieszczelności, które zostały załatwione w C++11. Dwie z nich są bardzo praktyczne i cieszą się dużą popularnością: klasy typu wyliczeniowego i `nullptr`.

Wszędzie tam, gdzie daną cechę pewnego obiektu zakodować można za pomocą niedużej liczby wartości programiści wykorzystują typ wyliczeniowy (ang. enum). Niestety, w C++98 typy wyliczeniowe nie posiadały swojego własnego zakresu. Dodatkowo można było je bez większego problemu porównywać z innymi typami wyliczeniowymi ze względu na niejawną konwersję do typu `int`. Na [Listingu 12](#) przedstawione zostały wymienione problemy.

### **Listing 12** Przykłady problemów związanych z typem wyliczeniowym w C++98

```
enum Color { Red, Black, Orange };
enum Size { Small, Medium, Large, Huge };

enum Fruit { Apple, Pineapple, Orange }; // Error!
// Orange already used in Color!
// Unfortunately everything inside these enums
// is in outer scope.

void setCursorColor(int color);

int main() {
    setCursorColor(Huge); // Doesn't make sense,
                          // but okay in C++98
    bool const weird = (Red == Apple);
}
```

Standard C++11 rozwiązuje wszystkie wymienione problemy wprowadzając tzw. klasę typu wyliczeniowego (ang. enum class). Posiada ona swój własny zakres oraz jest silnie typowana. Dodatkowo można ją „zapowiadać” (ang. forward declare). [Listing 13](#) ilustruje jej wykorzystanie.

### **Listing 13** Przykład stosowania klas typów wyliczeniowych

```

enum class Color {
    Red,
    Black,
    Orange
};

enum class Fruit {
    Apple,
    Pineapple,
    Orange // Ok, different scope.
};

void setCursorColor(int color);

int main() {
    // Addressing enum value
    Color c = Color::Orange;

    // Must be explicit!
    setCursorColor(
        static_cast
            <typename underlying_type<Color>::type>
                (Color::Red)
    );

    // Following is not possible
    bool const weird = Color::Red == Fruit::Apple; // Error
}

```

Drugą luką w systemie typów języka C++ była wartość reprezentująca nieustawiony wskaźnik (ang. null pointer). Programiści stosowali w tym celu wartość liczbową ukrytą pod aliasem **NULL**. Zazwyczaj **NULL** implementowano jako makro. Jednym z poważniejszych problemów związanych z **NULL** było to, że w niektórych przypadkach mógł być interpretowany jednocześnie jako liczba i wskaźnik. Ilustruje to **Listing 14**.

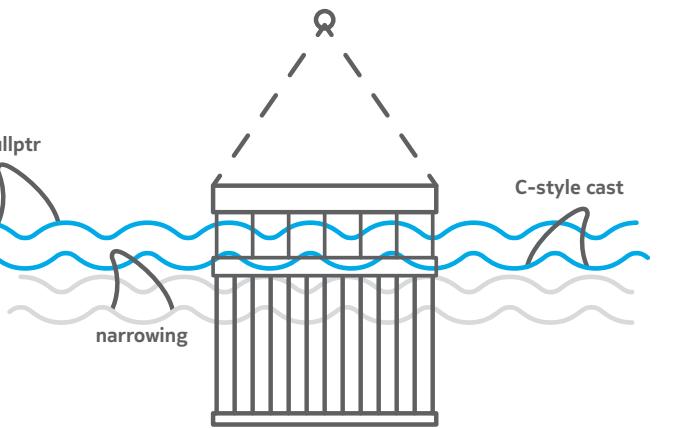
### **Listing 14** Przykład niejednoznaczności przy zastosowaniu NULL

```

void someFunction(int number);
void someFunction(char * text);

someFunction(NULL); // int or char* ?

```



Nowe słowo kluczowe **nullptr** rozwiązuje ten i wiele innych problemów związanych ze swoim poprzednikiem. Typ **nullptr\_t** jest zawsze traktowany jako typ wskaźnikowy.

#### **Nowe biblioteki**

Wraz ze zmianami wprowadzonymi bezpośrednio do samego języka wprowadzone zostały również nowe elementy do biblioteki standardej. Jedne z nich wypełniają lukę, która pozostała po standardzie C++98. Inne zaś wprowadzają całkowicie nową funkcjonalność. W następnych paragrafach zamieszczony został krótki opis kilku istotnych nowości z odświeżonej biblioteki standardej.

Bez wątpienia zaletą języka C++ jest jego wydajność. Dziwić zatem może fakt, że standardowa biblioteka kontenerów C++98 nie zawierała najbardziej wydajnej (znanej ludzkości) asocjacyjnej struktury danych, jaką jest tablica mieszająca (ang. hashmap). Na szczęście luka ta została uzupełniona w wydaniu C++11. Wprowadzone zostały odpowiedniki istniejących już **set**, **multiset**, **map** oraz **multimap**: **unordered\_set**, **unordered\_multiset**, **unordered\_map** oraz **unordered\_multimap** odpowiednio.

Inne wprowadzone do standaru biblioteki to m.in. biblioteka wątków (**std::thread**), biblioteka czasu (**std::chrono**) oraz biblioteka wyrażeń regularnych (**std::regex**).

Pierwsza z nich to, ogólnie mówiąc, konsekwencja wprowadzenia do standaru języka koncepcji związanych z wielowątkowością. Wielowątkowości w języku C++ poświęcona została niejedna książka. W niniejszym artykule nie ma niestety miejsca na szczegółowe informacje, więc zamieszczony został jedynie krótki przykład przedstawiający korzystanie z biblioteki **std::thread** na **Listingu 15**. Zainteresowani czytelnicy znajdą informacje w klasycznej pozycji „C++ concurrency in action: practical multithreading”[9].

### **Listing 15** Przykład wykorzystania biblioteki **std::thread**

```

int main() {
    auto imageView = getImageView(); // shared_ptr
    imageView->set(getDefaultAvatar());

    // This probably will be performed in background.
    future<void> result{async(launch::async,
        [] (weak_ptr<ImageView> imageView) {
            auto avatar = downloadAvatar();
            if (auto iv = imageView.lock()) {
                iv->set(move(avatar));
            }
        },
        imageView)};
}

// Main thread is not blocked!
// ...
this_thread::sleep_for(5s);

// Wait if it didn't finish on time.
result.get();
}

```

W przykładzie przedstawionym na **Listingu 15** wykorzystany jest szablon **std::future** pozwalający na uruchomienie dowolnego zadania „w tle” i pobraniu wyniku w nieokreślonej przyszłości. Za zadanie w tym wypadku posłużyło wyrażenie lambda operujące na inteligentnym wskaźniku na zasób obrazka (**imageView**). Nie użyto jednak wskaźnika typu **std::shared\_ptr** lecz **std::weak\_ptr**, którego konstruktor nie podnosi licznika referencji. Dzięki temu w przypadku zakończenia się funkcji, w której wyrażenie lambda zostało użyte zasób obrazka będzie mógł być natychmiastowo zwolniony – bez oczekiwania na zakończenie się utworzonego poprzednio zadania.

**std::chrono** z kolei to wydajna implementacja biblioteki do operacji związanych z czasem. Dzięki niej programista może w łatwy sposób reprezentować punkty w czasie – za pomocą **std::chrono::time\_point**. Ponadto biblioteka ta obsługuje zakresy (**std::chrono::duration**). **Listing 16** prezentuje przykładowe wykorzystanie biblioteki **std::chrono** do mierzenia czasu wykonywania pewnej czynności.

### **Listing 16** Wykorzystanie biblioteki **std::chrono** do pomiaru czasu wykonywania się funkcji

```

int main() {
    auto start = chrono::system_clock::now();

    // Do some work...
    this_thread::sleep_for(5s);

    auto stop = chrono::system_clock::now();
    auto elapsed =
        chrono::duration_cast<chrono::milliseconds>
            (stop - start);

    cout << „Elapsed: „ << elapsed.count() << „ms“ << endl;
}

```

Biblioteka **std::regex** również jest praktyczną rzeczą wprowadzoną w C++11. Dzięki niej programiści są w stanie w wygodny sposób pracować z wyrażeniami regularnymi. Wcześniej należało w tym celu wykorzystywać zewnętrzne biblioteki, niekoniecznie dostępne dla każdego systemu operacyjnego. **Listing 17** przedstawia przykładowe wykorzystanie biblioteki **std::regex** do wypisania wszystkich kodów pocztowych znajdujących się w wejściowym łańcuchu znaków.

W nowym standardzie C++ wprowadzono więcej bibliotek, lecz omówienie ich wszystkich wykracza poza rama niniejszej publikacji. Zainteresowani czytelnicy powinni szukać informacji bezpośrednio w standardzie bądź w witrynach zamieszczonych w bibliografii [2], [3].

### **Listing 17** Przykładowe wykorzystanie biblioteki **std::regex**

```

void printPostalCodes(string const& input) {
    regex re {"([0-9]{2})-([0-9]{3})"};
    sregex_token_iterator const end;

    for (sregex_token_iterator i(input.cbegin(), input.cend(), re);
         i != end; ++i) {
        cout << *i << endl;
    }
}

```

## Literaty użytkownika

Całości wymienionych rzeczy, które obecnie zawiera język C++ dopełniają nowe literaty standardowe oraz możliwość definiowania własnych. Pierwsze z nich są rozwińciem już istniejącym. Przykładowo przyrostki **f**, **L**, **UL** odpowiadają odpowiednio typom **float**, **long**, oraz **unsigned long**. Wersja standardu C++14 wprowadza więcej standardowych literałów:

liczbowe

- **b** – binarny (np. 0B0101)
- **h, min, s, ms, us, ns** – czasowy (np. auto d = 10us)

łańcuchowe

- **s** – łańcuch znaków std::string (np. auto s = "hello world"s)
- **u, U** – łańcuchy znaków zakodowane odpowiednio UTF-16 i UTF-32
- **u8** – łańcuchy znaków zakodowane w UTF-8 (np. u8 "Witaj świecie!")
- **R** – surowy literał pozwalający na zdefiniowanie terminatora łańcucha znaków

Są one praktyczne, ponieważ upraszczają kod źródłowy oraz go skracają. Pozwalają również na przechowywanie w plikach źródłowych łańcuchów w innym kodowaniu niż dotychczasowe ASCII. Niemniej istotną zmianą wprowadzoną w C++11 jest możliwość definiowania własnych przyrostków przez programistę. Przykładową definicję wraz z wykorzystaniem ilustruje [Listing 18](#).

**Listing 18** Przykład definicji literala użytkownika wraz z wykorzystaniem

```
struct Meters {  
    long double value_;  
};  
  
constexpr Meters operator „_m(long double value) {  
    return Meters{ value };  
}  
  
int main() {  
    Meters m1 = 12.5_m;  
    auto m2 = 62.5_m;  
}
```

## Zakończenie

Różnica pomiędzy językiem C++14 a jego poprzednikiem – C++98 – jest zauważalna gołym okiem. Nowe wydanie jest bardziej nowoczesne, praktyczne oraz kompletne. W nowej wersji znaleźć można nowe konstrukcje ułatwiające pracę programistom. Wprowadzone zostały również nowe funkcjonalności do biblioteki standardowej języka,

dzięki czemu użytkownicy szybciej mogą tworzyć wydajne oprogramowanie. W odróżnieniu od zmian stosowanych w niektórych innych językach, wszystkie zmiany w C++ są wstępnie kompatybilne z wcześniejszymi wersjami języka – dzięki czemu przejście na nowy standard jest niebywale łatwe.

Bez wątpienia można powiedzieć, że C++ przeżywa swoją drugą młodość. W tym nowym wydaniu programiści mogą pisać zrozumialszy, wydajniejszy oraz bardziej zwięzły kod. Innymi słowy jakość tworzonego oprogramowania wzrasta wraz z wykorzystaniem standardu C++14, do czego dążyć powinny zarówno organizacje, jak i indywidualni programiści.

## Ciekawostka

Ciekawostką związaną ze standardem C++11 jest fakt wprowadzenia obsługi sąsiadujących ze sobą znaków zamkających nawiasy trójkątne w odniesieniu do szablonów. W poprzedniej wersji języka standard tego nie uściął, przez co wyrażenie to traktowane było zawsze jako operator przesunięcia bitowego w prawo. Rozwiążaniem tego problemu było wówczas włączenie rozszerzeń kompilatora lub częstsze umieszczanie dodatkowej spacji. Szczęśliwie C++11 rozwiązał ten trywialny problem.

## Wsparcie kompilatorów

Istnieje wiele kompilatorów języka C++. Te, które rozwijane są do dnia dzisiejszego swoje wsparcie dla nowo powstających wersji standardu języka C++ wprowadzają stopniowo. Z tego powodu warto korzystać z najnowszej wersji kompilatora(ów), który używany jest w danym projekcie. Warto również na bieżąco śledzić postęp prac zarówno w danym kompilatorze, jak i skojarzonej z nim bibliotece standardowej. Niezbędne informacje łatwo można znaleźć w sieci. Przykładowo, dla GCC będą to strony [4] i [5], a dla Clang [6] i [7]. C++14 jest w pełni wspierany przez Clang w wersji 3.4 i GCC w wersji 5.

## Bibliografia

- [0] Working Draft, Standard for Programming Language C++ (N3793)
- [1] <http://www.cplusplus.com/info/history/>
- [2] <http://www.cplusplus.com>
- [3] <http://en.cppreference.com/>
- [4] <https://gcc.gnu.org/projects/cxx0x.html>
- [5] <https://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.200x>
- [6] [http://clang.llvm.org/cxx\\_status.html](http://clang.llvm.org/cxx_status.html)
- [7] [http://libcxx.llvm.org/cxx1y\\_status.html](http://libcxx.llvm.org/cxx1y_status.html)
- [8] <http://www.stroustrup.com/C%2B%2BFAQ.html>
- [9] Williams A., C++ concurrency in action: practical multithreading, Manning Publications 2012.

## Autor o swojej pracy

Jestem absolwentem informatyki na Politechnice Wrocławskiej. Programuję odkąd pamiętam. Obecnie pracuję w firmie Nokia Networks na stanowisku Lead C++ Software Engineer. Zainteresowania: optymalizacja oraz jakość oprogramowania, security, budowa kompilatorów. Prowadzę bloga o programowaniu: <http://szborows.blogspot.com>.

## Sławomir Zborowski

Engineer, Software Development C++  
MBB SRAN OMCP CP 3

# Na terytorium kompilatora: libclang

Michał Bartkowiak  
Engineer, Software Development  
MBB FDD LTE Development Wrocław 1



W ostatnich latach kompilator Clang/LLVM zdobył ogólną popularność, stając się pełnowartościową alternatywą dla GCC. Z punktu widzenia użytkowników kompilatorów głównymi cechami, które to umożliwiły są: kompatybilność interfejsu linii komend z GCC, przejrzystość i jakość informacji zwrotnej o błędach i ostrzeżeniach podczas komplikacji oraz szybkie zapewnienie wsparcia dla nowego standardu języka C++.

Clang to jednak nie tylko kompilator. To również zestaw bibliotek do przetwarzania kodu źródłowego. Większość z nich oferuje interfejs w języku C++, który może zmieniać się pomiędzy kolejnymi wersjami Clanga i wymaga posiadania wiedzy o budowie i działaniu kompilatorów. Przygotowana została także wspólnie biblioteka oferująca stabilny interfejs w języku C: libclang [1, 2]. W dalszej części artykułu zaprezentowany zostanie zarys jej możliwości.

## Możliwości biblioteki libclang

Biblioteka libclang operuje na poziomie abstrakcyjnego drzewa rozbiornika (ang. Abstract Syntax Tree, AST), które reprezentuje kod programu w języku C, Objective-C lub C++. Wśród jej możliwości znajdują się:

- podział kodu na tokeny,
- budowa drzew AST,
- poruszanie się po drzewie AST za pomocą kurSORów,
- odwzorowanie pomiędzy kurSORami a lokalizacjami w kodzie,
- raportowanie informacji diagnostycznych,
- zapewnianie podpowiedzi kontekstowych.

Największe zalety biblioteki to:

- nieskomplikowany i stabilny interfejs w języku C,
- możliwość uzyskania dokładnej informacji o lokalizacji w kodzie na każdym etapie pracy z biblioteką,
- popularność i renoma stojącego za nią projektu.

## Od czego zacząć?

Aby rozpocząć pracę z biblioteką libclang należy skorzystać z odpowiedniego pliku nagłówkowego, a następnie stworzyć indeks oraz wczytać jednostkę translacji (ang. Translation Unit, TU). Przykład kodu realizującego te zadania przedstawiono na [Listingu 1](#). Zmienne `argv` oraz `argc` powinny przechowywać liczbę parametrów oraz same parametry komplikacji (np. `-std=c++11 -Wall file.cpp` w formie, jaką zwykle trafia do funkcji `main`). Kod operujący na jednostce translacji powinien trafić w miejsce oznaczone przez komentarz.

### [Listing 1](#) Tworzenie jednostki translacji

```
#include <clang-c/Index.h>

auto index = clang_createIndex(0, 0);
auto tu = clang_parseTranslationUnit(
    index, 0, argv, argc, 0, 0,
    CXTranslationUnit_None);

// operacje na jednostce translacji

clang_disposeTranslationUnit(tu);
clang_disposeIndex(index);
```

## Informacje o błędach i ostrzeżeniach

Korzystając z biblioteki libclang, można uzyskać wysokiej jakości informacje diagnostyczne, a następnie wykorzystać je we własnym programie. Kod ilustrujący otrzymywanie sformatowanej informacji przedstawiony został na [Listingu 2](#) a jego działanie na [Listingach 3 i 4](#).

### [Listing 2](#) Uzyskiwanie informacji o błędach i ostrzeżeniach

```
for (auto diagNum = 0u;
     diagNum < clang_getNumDiagnostics(tu); ++diagNum) {
    auto diag = clang_getDiagnostic(tu, diagNum);
    auto diagStr = clang_formatDiagnostic(diag,
                                           clang_defaultDiagnosticDisplayOptions());
    std::cout << clang_getCString(diagStr) << "\n";
    clang_disposeString(diagStr);
}
```

### Listing 3 Niepoprawny kod C++

```
class X {  
    const int a;  
}
```

### Listing 4 Informacja diagnostyczna dla kodu z listingu 3.

```
class.cpp:1:7: warning: class 'X' does not declare any constructor to initialize its non-modifiable members  
class.cpp:3:2: error: expected ';' after class  
class.cpp:2:15: warning: private field 'a' is not used [-Wunused-private-field]
```

Powyższy przykład jest najprostszą metodą otrzymywania informacji diagnostycznej. Każdy z jej komponentów: lokalizację, poziom i tekst, można uzyskać oddzielnie. Ponadto za pomocą funkcji `clang_getDiagnosticNumFixIts` i `clang_getDiagnosticFixIt` możliwe jest otrzymanie opcjonalnych podpowiedzi o tym, jak rozwiązać problem (ang. fix-its), o którym Clang informuje. Dla błędu braku średnika w kodzie z [Listingu 3](#). fix-it będzie następujący: „3:2 - 3:2: ;” (w trzeciej linii, w drugiej kolumnie postaw znak „;”).

Tak obszerne informacje diagnostyczne wykorzystywane są np. do analizy kodu źródłowego pod kątem występowania ostrzeżeń lub w zintegrowanych środowiskach programistycznych (ang. Integrated Development Environment, IDE) do prezentacji bieżącego stanu rozwijanego kodu.

### Drzewo rozbiórku

Drzewo AST generowane przez kompilator Clang można zobaczyć w formie tekstu, jeśli wywołany zostanie on z argumentami `-Xclang -ast-dump`. Drzewo dla kodu z [Listingu 5](#). prezentuje [Listing 6](#). Może ono być przydatne w analizie błędów podczas pisania kodu, ponieważ zawiera również elementy wygenerowane przez kompilator, np. domyślne konstruktory, operatory kopowania i instancje szablonów dla konkretnych typów.

### Listing 5 Przykładowy kod C++

```
int main() {  
    auto a = 1;  
    return 10 + a;  
}
```

### Listing 6 Drzewo AST (wybrane informacje) dla kodu z listingu 5.

```
TranslationUnitDecl 0x9c0fc00 <>> <>  
|-FunctionDecl 0x9c0ff60 <ex.cpp:1:1, line:4:1> line:1:5 main 'int  
(void)'  
`-CompoundStmt 0x9c10148 <col:12, line:4:1>  
| -DeclStmt 0x9c100d0 <line:2:5, col:15>  
| ` -VarDecl 0x9c10000 <col:5, col:14> col:10 used  
a 'int':'int' cinit  
| ` -IntegerLiteral 0x9c10030 <col:14> 'int' 1  
`-ReturnStmt 0x9c10138 <line:3:5, col:17>  
`-BinaryOperator 0x9c10120 <col:12, col:17> 'int' '+'  
| -IntegerLiteral 0x9c100e0 <col:12> 'int' 10  
`-ImplicitCastExpr 0x9c10110 <col:17> 'int':'int' <LValue  
eToRValue>  
`-DeclRefExpr 0x9c100f8 <col:17> 'int':'int' lvalue Var  
0x9c10000 'a' 'int':'int'
```

### Poruszanie się po drzewie AST

W bibliotece libclang udostępniony został interfejs wspierający poruszanie się po drzewie AST zbudowanym dla danej jednostki translacji. Umożliwia to analizę jednostki nie tylko pod względem składniowym, ale również semantycznym.

Przeglądanie drzewa AST zaimplementowano z użyciem wzorca wizytatora (ang. visitor pattern). Bytem, za pomocą którego odwidzamy węzły drzewa, jest `CXCursor`. Kursor może wskazywać m.in. takie elementy języka C++ jak deklaracje, definicje, wyrażenia lub referencje. Niesie on informacje o nazwie, typie, lokalizacji w kodzie źródłowym oraz swoich dzieciach.

Drzewo AST mogłoby zostać przejrzone w celu znalezienia deklaracji funkcji i metod, np. aby sprawdzić czy nazwy tych elementów są zgodne z przyjętą w projekcie konwencją. Działanie takie jest przedstawione na [Listingu 7](#). Funkcja wizytująca (`guest`) sprawdza rodzaj każdego kurSORA (`clang_getCursorKind`) i podejmuje odpowiednie akcje dla interesujących nas kurSORów oraz decyduje jak przeglądać drzewo. Aby podążyć w głąb niego konieczne jest zwrocenie wartości `CXChildVisit_Recurse`.

### Listing 7 Przeglądanie drzewa AST w celu znalezienia deklaracji funkcji i metod

```
CXChildVisitResult guest(CXCursor cursor,  
                           CXCursor parent,  
                           CXClientData client_data) {  
  
    switch (clang_getCursorKind(cursor)) {  
        case CXCursor_FunctionDecl:  
            std::cout << "function"; break;  
        case CXCursor_CXXMethod:  
            std::cout << "cxxmethod"; break;  
        default: break;  
    }  
    return CXChildVisit_Recurse;  
}  
  
clang_visitChildren(  
    clang_getTranslationUnitCursor(tu), guest, 0);
```

KurSOR udostępnia informacje nie tylko o sobie samym, lecz także o elementach z nim powiązanych. Dla kurSORów wskazujących deklaracje można znaleźć ich definicje (`clang_getCursorDefinition`) natomiast dla referencji otrzymać kurSOR, do którego się one odwołują (`clang_getCursorReferenced`). Nie jest problemem uzyskanie informacji o składniowym i semantycznym rodzicu danego kurSORA (`clang_getCursorLexicalParent`, `clang_getCursorSemanticParent`). Różnicę między nimi obrazuje definicja metody klasy znajdującej się poza ciałem tej klasy. Jej semantycznym rodzicem będzie klasa, do której należy, natomiast leksykalnym otoczenie definicji (np. pewna przestrzeń nazw).

Dzięki swobodnemu przemieszczaniu się po drzewie AST, można także wykonywać obliczenia związane ze strukturą kodu. Umożliwia to analizę programu reprezentowanego przez dane AST.

### Podpowiadanie semantyczne

Jedną z cech, z których znana jest biblioteka libclang, to generowanie podpowiedzi semantycznych dla tworzonego kodu. Środowiska takie XCode, QtCreator, KDevelop korzystają lub są w trakcie migracji do mechanizmów oferowanych przez libclang. Dotychczasowe rozwiązania, oparte najczęściej na mniej lub bardziej poprawnych parserach napisanych na potrzeby mechanizmu podpowiadania, okazały się zbyt mało elastyczne, by dostosować się do standardu C++11. Niewątpliwą zaletą podpowiadania przy pomocy libclang jest również zawarcie w zbiorze podpowiedzi elementów generowanych przez kompilator.

Eksperymenty z autouzupełnianiem można zacząć od narzędzia `c-index-test`, dostarczanego razem z biblioteką:

```
$ c-index-test -code-completion-at=\  
<plik>:<linia>:<kolumna> <plik + argument>
```

Kod, który umożliwia otrzymanie oraz przetworzenie podpowiedzi przedstawia [Listing 8](#). Niestety w tym momencie interfejs biblioteki staje się skomplikowany, dlatego na listingu zaprezentowany został tylko zarys jego użycia.

### Listing 8 Zarys kodu uzyskującego i przetwarzającego podpowiedź

```
auto compls = clang_codeCompleteAt(  
    tu, "file.cpp", line, column, 0, 0,  
    clang_defaultCodeCompleteOptions());  
  
for (auto i = 0u; i < compls->NumResults; ++i) {  
    auto &cStr = compls->Results[i].completionString;  
    for (auto j = 0u; j <  
        clang_getNumCompletionChunks(cStr); ++j) {  
        auto chunkStr =  
            clang_getCompletionChunkText (cStr, j);  
        std::cout << toString(chunkStr) << " ";  
    }  
}  
  
clang_disposeCodeCompleteResults(compls);
```

### Podsumowanie

Przykłady zaprezentowane powyżej dają zaledwie przedsmak tego, co można osiągnąć z wykorzystaniem biblioteki libclang. Dzięki niej można cieszyć się lepszymi środowiskami IDE. Deweloperzy mogą wykorzystać ją do tworzenia narzędzi do refaktorowania, formatowania, migracji, analizy i generowania kodu. Nie jest przy tym konieczne bezpośrednie używanie interfejsu C, ponieważ udostępnione zostały także wiązania w języku Python [3]. Gdy mechanizmy oferowane przez libclang okażą się niewystarczające względem potrzeb, można skorzystać z wewnętrznych interfejsów kompilatora Clang.

### Bibliografia:

- [1] [http://clang.llvm.org/doxygen/group\\_\\_CINDEX.html](http://clang.llvm.org/doxygen/group__CINDEX.html)
- [2] <http://llvm.org/devmtg/2010-11/Gregor-libclang.pdf>
- [3] <http://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang>
- [4] [http://cppwroclaw.pl/dokuwiki/\\_media/spotkania/005/01\\_libclang.pdf](http://cppwroclaw.pl/dokuwiki/_media/spotkania/005/01_libclang.pdf)

## Autor o swojej pracy

---

Pracuję w dziale MBB LTE C-Plane, w zespole K3.  
Tworzymy kompilator, środowisko uruchomieniowe  
oraz narzędzia dla języka TTCN-3, który  
wykorzystywany jest do testowania oprogramowania  
tworzonego w ramach projektu LTE C-Plane.

---

### **Michał Bartkowiak**

Engineer, Software Development  
MBB FDD LTE Development Wrocław 1

# Wprowadzenie do biblioteki STL

Artur Kochowski  
Engineer, Software Development  
MBB Single RAN

NOKIA

Na początku należy zapytać, czym właściwie jest STL? Standard template library (pol. standardowa biblioteka szablonów), bo tak rozwija się ten skrót, stanowi główną, przez wielu uznawaną za najciekawszą, część biblioteki standardowej języka C++. Udostępniona programistom takie narzędzia jak kontenery i algorytmy ogólnego zastosowania.

Niestety, wiele programów nauczania dotyczących C++ przemilcza istnienie biblioteki standardowej (w tym STL). Jednym z powodów może być to, że biblioteka STL nie zawsze była integralną częścią C++. W początkowej fazie rozwoju była projektowana zupełnie niezależnie (jej głównym twórcą jest Alexander Stepanov). Stała się ona jednak częścią już pierwszego standardu C++ z 1998 roku. Od tamtego czasu zarówno język, jak i sama biblioteka przeszły długą drogę. W chwili pisania tych słów wydany już został standard C++14 i trwają prace nad kolejnym – C++17. Parafrazując Bjarne Stroustrupą, twórcę języka C++, żaden istotny program nie jest napisany w „gołym” języku programowania.

W obecnych czasach znajomość STL jest konieczna, aby efektywnie pisać kod w nowoczesnym C++.

W kolejnych sekcjach zostaną poruszone następujące tematy:

1. kontenery oferowane przez STL,
2. koncepcja iteratorów,
3. wybrane algorytmy.

## 1. Kontenery

Kontenery ogólnego zastosowania, które udostępnia nam STL, są często pierwszą rzeczą, po której sięgają programiści. Jak można się domyślić są ku temu dobre powody. Dostarczone implementacje kolekcji pozwalają na przechowywanie zarówno typów standardowych, jak i użytkownika, udostępniając spójne, dobrze zaprojektowane interfejsy.

Oczywiście każdy kontener ma swoje wady i zalety, dlatego ważne jest, aby wyboru kolekcji dokonywać świadomie.

### 1.1 Vector

**vector** zachowuje się jak dynamiczna tablica, która sama zwiększa swój rozmiar. Standard nie wymaga wprawdzie, aby był on implementowany za pomocą dynamicznej tablicy, jednak jest to tylko formalność. Wymagania dotyczące złożoności obliczeniowej operacji wymagają takiej implementacji.

Właśnie dzięki ciągłej reprezentacji w pamięci możliwe jest załadunek kolejnych elementów vectora do pamięci podrzędnej (z ang. cache). Dzięki temu operacje na tym kontenerze są bardzo szybkie i może on być traktowany jako domyślny wybór. Oczywiście, nie należy go ślepo stosować wszędzie – priorytetem powinna być czytelność kodu. Jak mawia stare programistyczne porzekadło: przed-

wczesna optymalizacja jest źródłem wszelkiego złła. W przypadku, gdy nie wpływa to negatywnie na czytelność kodu **vector** jest prawdopodobnie dobrym wyborem. Z tego właśnie powodu kontener ten został opisany tutaj w zdecydowanie szerszym zakresie.

Jak wspomniano wcześniej, **vector** zachowuje się jak zwiększająca rozmiar tablica. Należy tutaj zwrócić uwagę na to, w jaki sposób przebiega to zwiększanie rozmiaru.

Otoż **vector** rezerwuje pamięć, w której może przechować określona liczbę elementów. Ta pamięć jest zarezerwowana (zajęta przez **vector**), nawet, gdy z logicznego punktu widzenia kontener jest pusty (przechowywane jest w nim 0 elementów).

Wstawiane do **vectora** elementy trafiają do wcześniejszej zaalokowanej pamięci. Gdy okazuje się, że nie ma już miejsca, aby wstawić kolejny element (podczas wstawiania elementu, nie przed) następuje realokacja. **vector** zajmuje większy fragment pamięci, jego pojemność się zwiększyła.

Jak można sprawdzić ile jeszcze **vector** pomieści elementów, zanim wymagana będzie realokacja pamięci? Metoda **size** zwraca liczbę elementów przechowywanych przez **vector**, natomiast metoda **capacity** zwraca liczbę elementów, które **vector** może przechować bez realokacji pamięci. Warto w tym momencie wspomnieć jeszcze o metodzie **reserve**, która pozwala nam na zaalokowanie dla **vectora** pamięci pozwalającej przechować co najmniej określona liczbę elementów. Należy jednak pamiętać, że metoda ta nie zmniejszy rozmiaru zaalokowanej pamięci – wywołanie jej dla wartości mniejszej od obecnej pojemności **vectora** nie wywoła żadnego efektu.

Omówione zostało dodawanie elementów. Ale co dzieje się w przypadku, gdy elementy są z **vectora** usuwane? Zmniejsza się jego rozmiar (podawany przez metodę **size**), ale nie zmniejsza się jego pojemność (zwracana przez metodę **capacity**). Usuwanie elementów z **vectora** nigdy nie powoduje realokacji pamięci. **vector** nigdy sam nie zmniejszy rozmiaru zajmowanej przez siebie pamięci.

Możemy natomiast wykorzystać sztuczkę z podmianą pamięci przy pomocy metody **swap** (z ang. **swap trick**), a od C++11 dodano metodę **shrink\_to\_fit**.

Jest jeszcze jedna rzecz, o której należy wspomnieć, czyli specjalizacja **vectora** dla typu **bool**. Dla tego typu przewidziana jest specjalna implementacja – taka, która zajmuje mniej miejsca w pamięci. Może wydawać się to logiczne – dla zwykłej implementacji **vectora** każdy element zajmowałby co najmniej jeden bajt, podczas gdy do przechowywania wartości logicznej wystarczy jeden bit. Potencjalnie ośmiokrotnie mniej zużytej pamięci. Co zatem jest problemem? Otoż w języku C++ najmniejszy obszar pamięci, który można zaadresować, ma rozmiar jednego bajta.

W związku z tym ta specjalizacja vectora musi obsługiwać nietypowe sytuacje dla iteratorów i referencji. Nie spełnia też ona niektórych wymagań dla klasy **vector**.

Jest to coś, o czym warto pamiętać. Wyjątek ten może zaskoczyć, zwłaszcza, przy pisaniu kodu wielowątkowego.

## 1.2 Array

**array**, podobnie jak **vector**, modeluje tablicę – z tą różnicą, że statyczną. Tak naprawdę jest to opakowanie na statyczną tablicę języka C, pozwalające na wygodne wykorzystanie w kodzie generycznym. W związku z tym nie ma możliwości zmiany liczby elementów znajdujących się w tym kontenerze, można jedynie zmienić ich wartość. Kolejną rzeczą, która odróżnia **array** od vectora jest to, że w przypadku alokacji kontenera na stosie również dane zostaną tam zaalokowane (w przypadku vectora przechowywane w nim elementy znajdują się na stercie, nawet, gdy sam **vector** zostanie zaalokowany na stosie). Dzięki temu dostęp do danych jest bardzo szybki.

Warto tutaj dodać, że w przypadku, gdy znamy liczbę elementów, które będziemy chcieli przechowywać i liczba ta nie będzie się zmieniać **array** jest prawdopodobnie lepszym wyborem niż **vector**.

## 1.3 Deque

**deque** jest w pewnym sensie kontenerem podobnym do vectora. Również modeluje dynamiczną tablicę i ma bardzo podobny interfejs. Różnica polega na tym, że modeluje tablicę rozszerzalną zarówno na końcu, jak i na początku – w założeniu umożliwia szybkie dodawanie i usuwanie elementów na obu końcach. W praktyce jednak dodawanie elementów nawet na początek vectora jest bardzo szybkie, ponieważ kopiowanie ciąglej pamięci nie stanowi problemu dla dzisiejszych komputerów.

Najczęściej kontener ten jest implementowany przy pomocy ciągu mniejszych tablic, z których pierwsza rozszerza się w jednym kierunku, a ostatnia w drugim.

## 1.4 List/Forward List

**list** jest kontenerem modelującym zachowanie listy dwukierunkowej. Podstawową zaletą jest szybkie dodawanie i usuwanie elementów w dowolnym miejscu (pod warunkiem, że miejsce to jest już znane). Operacje te nie unieważniają również iteratorów, co jest nie bez znaczenia na przykład przy przetwarzaniu równoległym.

Niestety, ceną jaką trzeba zapłacić jest brak swobodnego dostępu do elementów oraz, co bardziej istotne z punktu widzenia wydajności, fragmentacja pamięci.

**forward\_list** natomiast jest kontenerem modelującym listę jednokierunkową. Posiada zatem mniejszy narzut pamięciowy, jednak nie umożliwia iterowania wstecz. Został wprowadzony wraz ze standardem C++11.

## 1.5 Set/Multiset

**set** i **multiset** to kontenery, które przechowują wartości jako klucze. Klucze te posortowane są według określonego kryterium. Nie mogą być one modyfikowane, mogą zostać jedynie do kontenera dodane bądź z niego usunięte.

Kontenery te najczęściej implementowane są jako drzewa czerwono-czarne.

**multiset** umożliwia dodatkowo przechowywanie zduplikowanych kluczów.

## 1.6 Map/Multimap

**mapa** i **multimapa** to kontenery podobne do seta i multiseta – różnica polega na tym, że zamiast przechowywać klucze przechowują one pary klucz-wartość.

## 1.7 Unordered map/multimap/set/multiset

Kontenery z prefiksem **unordered\_** są w STL-u niczym innym, jak kontenerami haszującymi.

W przeciwieństwie do zwykłych wersji **set**, **map**, **multiset** i **multimap**, które najczęściej implementowane są jako drzewa czerwono-czarne, wersje **unordered\_** wykorzystują funkcję haszującą, która zwraca dla elementu tak zwany hasz. Na jego podstawie element jest zapisywany w tablicy.

Elementy znajdujące się w tych kontenerach nie są zatem posortowane (jak nazwa wskazuje), ale dostęp do nich, jak również dodawanie i usuwanie elementów jest zazwyczaj szybsze. Zazwyczaj, ponieważ w przypadku, gdy kilka elementów jest przypisanych przez funkcję haszującą w to samo miejsce wydajność znacznie spada (złożoność przechodzenia po liście tych elementów jest liniowa).

Pozostała jeszcze kwestia nazwy. Skoro są to kontenery haszujące, to dlaczego wybrano nazwę **unordered\_map** zamiast na przykład **hash\_map**? Powód jest historyczny.

W pierwszej wersji biblioteki STL, przy standardzie C++98, kontenery haszujące nie zostały wprowadzone ze względu na brak czasu. Z tego powodu pojawiło się wiele implementacji, które nie były wyseptyfikowane przez standard. Prefiks **unordered\_** wprowadzono, aby uniknąć konfliktów nazw.

## 1.8 Adaptery

Oprócz zwykłych kontenerów STL udostępnia jeszcze tak zwane adaptery. Udostępniają one określone interfejsy, wykorzystując przy tym implementację bazowych kontenerów. Należą do nich:

a) **stack** – czyli stos, ostatni włożony element jest pierwszym wyciąganym (LIFO, last in first out),

b) **queue** – czyli klasyczna kolejka, pierwszy element włożony jest pierwszym wyciąganym (FIFO, first in first out),  
c) **priority queue** – kolejka priorytetowa, elementy są posortowane za pomocą podanego przez programistę kryterium (domyślnie jest to operator mniejszości), element o największym priorytecie jest tym, który zostanie wyciągnięty jako pierwszy.

## 2. Koncepcja iteratorów

Iteratory są obiektami, które reprezentują pozycję elementów w kontenerze. Stanowią one swoistego rodzaju spivo, łączące kontenery z algorytmami. Muszą udostępniać kilka operacji:

- a) dostęp do elementu – realizowany przez **operator\***,
- b) przejście do kolejnego elementu – realizowane przez **operator++**,
- c) porównywanie iteratorów – poprzez **operator==** oraz **operator!=**,
- d) przypisanie – poprzez **operator=**.

Obiekt, który spełnia te wymagania, może być nazywany iteratorem. Zazwyczaj jednak iteratory udostępniają więcej możliwości. Dzielą się one na następujące kategorie:

- a) Iteratory naprzód (ang. forward iterator) – pozwalają na iterowanie tylko w jedną stronę (przy użyciu operatora inkrementacji);
- b) Iteratory dwukierunkowe (ang. bidirectional iterator) – pozwalają na iterowanie w dwie strony, naprzód (przy użyciu operatora inkrementacji) oraz wstecz (przy użyciu operatora dekrementacji);
- c) Iteratory swobodnego dostępu (ang. random access iterator)
  - posiadają właściwości iteratorów dwukierunkowych, dodatkowo umożliwiając swobodny dostęp do elementów, oferując podobną funkcjonalność do arytmetyki wskaźników.

Iteratory udostępniane przez standardowe kontenery zaliczają się do jednej z powyższych kategorii.

Iteratory można jeszcze podzielić ze względu na drugie kryterium:

- a) Iteratory wejściowe – pozwalają na odczyt danych,
- b) Iteratory wyjściowe – pozwalają na zapis danych.

## 2.1 Adaptery iteratorów

Aby ułatwić niektóre zadania do dyspozycji programisty oddano również adaptery iteratorów – klasy, które implementują interfejs iteratorów, ale ich zachowanie jest niestandardowe. Należą do nich:

- a) iteratory wstawiania: **back\_inserter**, **front\_inserter** oraz **inserter** – pozwalają one na dodawanie elementów do kontenera;
- b) adaptery na strumienie, które pozwalają na wypisanie do strumienia, jak również odczyt ze strumienia – dzięki temu możemy łatwo wypisywać kontenery na standardowe wyjście, czy w prosty sposób wczytywać je z plików.

## 2.2 Globalne funkcje

Wraz z C++11 wprowadzono globalne funkcje **begin** oraz **end**. Pozwalają one uzyskać iteratory wskazujące odpowiednio na początek i koniec kontenera bądź statycznej tablicy. Rozwiążanie to umożliwia także dostosowanie klas, które nie udostępniają kompatybilnego z STL-em interfejsu, do współpracy z algorytmami czy chociażby pętlą **for** opartą na zakresach bez ingerencji w same klasy. Wszystko to zdecydowanie ułatwia pisanie generycznego kodu.

W standardzie C++14 wyspecyfikowano również odpowiedniki pozwalające uzyskać dostęp do iteratorów odwrotnych (służących do iterowania przez zakres od tyłu): globalne funkcje **rbegin** oraz **rend**.

## 3. Algorytmy

STL oferuje ponad dziewięćdziesiąt algorytmów, które pozwalają nie tylko na przyśpieszenie i ułatwienie pisania kodu, ale również na zwiększenie jego czytelności. Niestety, ze względu na ograniczone miejsce, tylko wybrane zostaną tutaj przedstawione.

### 3.1 for\_each

Jak nazwa wskazuje, algorytm ten pozwala na wykonanie określonej operacji dla wszystkich elementów w zakresie. Z wejściem C++11 wprowadzone zostały pętle operujące na zakresach, oferujące podobną funkcjonalność w bardziej przystępny sposób, w związku z czym algorytm ten z pewnością straci na popularności.

### 3.2 Wyszukiwanie elementów

STL dostarcza różne algorytmy, które pozwalają na wyszukiwanie elementów. Poniżej zostaną krótko omówione wybrane z nich.

Do wyszukiwania pierwszego wystąpienia w zakresie służą algorytmy **find**, **find\_if** oraz **find\_if\_not**. Pierwszy z nich służy do wyszukiwania elementu o zadanej wartości, kolejne dwie wersje służą odpowiednio do znalezienia pierwszego elementu spełniającego bądź niespełniającego zadany predykat.

Aby znaleźć pierwszy podciąg w zakresie należy użyć algorytmu **search**. Podciąg do wyszukania jest przekazywany przy pomocy iteratorów. Aby wyszukać ostatni podciąg należy użyć algorytmu **find\_end** – nie jest to niestety pomyłka edytorska, a pewna niekonsekwencja w bibliotece STL.

Jest też algorytm pozwalający na znalezienie pierwszego z kilku elementów – **find\_first\_of**.

Jeżeli zakres jest posortowany można również wykorzystać algorytm **binary\_search**.

### 3.3 Wyszukiwanie minimum i maximum

STL udostępnia kilka algorytmów, które pozwalają na znalezienie minimalnych i maksymalnych wartości w zakresie. Są nimi **min\_element**, **max\_element** oraz **minmax\_element**. Zwracają one iteratory

(w przypadku **minmax\_element** parę iteratorów) do odpowiednich elementów.

Nie należy mylić ich z funkcjami **min**, **max** oraz **minmax**, które również są dostarczone przez STL. Służą one do porównywania pojedynczych elementów, nie zakresów.

#### 3.4 Kopiowanie i przenoszenie elementów

W STL-u jest kilka algorytmów służących do kopowania elementów: **copy**, **copy\_if**, **copy\_n** czy **copy\_backward**. Występują również algorytmy, które wykonują jakąś operację, a jej wynik kopią do innego kontenera. Mają one sufiks **\_copy** (na przykład **partial\_sort\_copy**). Warto sprawdzić, czy nie istnieje taka wersja algorytmu.

Należy też pamiętać, że wraz z wprowadzeniem C++11 dodano algorytmy umożliwiające przeniesienie obiektów: **move** oraz **move\_backward**.

#### 3.5 Transformacja elementów

Aby wykonać określoną operację na elementach, a wynik zapisać do innego iteratora można użyć algorytmu **transform**. Jego działanie jest bardzo podobne do algorytmu **for\_each**. Jedyną różnicą jest konieczność podania właśnie wyjściowego iteratora. Jeżeli iterator wyjściowy będzie taki sam jak początek zakresu wskazywany przez iterator wejściowy, to działanie tego algorytmu będzie identyczne jak w przypadku **for\_each**.

Warto wiedzieć, że algorytm ten posiada również wersję, w której podaje się dwa zakresy wejściowe.

#### 3.6 Usuwanie elementów

Algorytmy **remove** oraz **remove\_if** służą do usunięcia elementów z zakresu. Należy jednak pamiętać, że fizycznie ich nie usuwają, przesuwają jedynie elementy, które nie powinny zostać usunięte na początek zakresu. Aby faktycznie usunąć elementy należy użyć metody **erase** (tak zwany **erase-remove** idiom).

Istnieją również wersje **\_copy**, **remove\_copy** oraz **remove\_copy\_if**.

#### 3.7 Usuwanie duplikatów

Do usuwania duplikatów służy algorytm **unique**. Należy jednak pamiętać o dwóch bardzo istotnych rzeczach. Po pierwsze, podobnie jak algorytmy z rodziny **remove** elementy są tylko przesuwane – trzeba wywołać również **erase** aby fizycznie je usunąć.

Druga rzecz – jeszcze ważniejsza i zdecydowanie mniej intuicyjna:

algorytm ten bierze pod uwagę jedynie następujące po sobie identyczne elementy. Jest to podyktowane złożonością obliczeniową. Aby usunąć powtarzające się wartości z całego zakresu, nie tylko te następujące po sobie, trzeba wywołać algorytm **unique** dla posortowanego zakresu, po czym wywołać jeszcze **erase**.

#### 3.8 Sortowanie elementów

STL udostępnia kilka algorytmów umożliwiających sortowanie elementów. Dobra jest znać różnice między nimi, aby wybrać najlepiej spełniający zadanie.

- a) **sort** – czyli standardowy algorytm sortowania,
- b) **stable\_sort** – algorytm zapewniający zachowanie względnej kolejności elementów (elementy, które według kryterium mają identyczną wartość zachowają swoją kolejność względem siebie),
- c) **partial\_sort** – pozwala na posortowanie części zakresu, uwzględniając elementy z większego zakresu,
- d) **partial\_sort\_copy** – jak partial\_sort, ale wynik zapisywany jest do wyspecyfikowanego iteratora,
- e) **nth\_element** – dokonuje częściowego sortowania: na zadanym miejscu znajduje się element, który znajdowałby się tam, gdyby zakres był posortowany. Dodatkowo elementy poprzedzające są mniejsze, a następujące większe od zadanego.

#### Co dalej?

W tym artykule poruszone zostały podstawowe zagadnienia związane z biblioteką STL, ale oczywiście nie jest to wyczerpujące źródło wiedzy. Wyczerpującą lekturą na ten temat jest książka „C++. Biblioteka standardowa. Podręcznik programisty. Wydanie II” Nicolai Josuttisa, która została wydana przez wydawnictwo Helion. Jak wskazuje tytuł omówione są również zagadnienia związane z biblioteką standardową.

Inną ciekawą pozycją jest „Effective STL” autorstwa Scotta Meyersa. Dostarcza ona praktycznych porad, jak wykorzystać STL w codziennej pracy. Niestety, ta książka nie została jak do tej pory wydana w Polsce.

Niezastąpione są również źródła internetowe, takie jak na przykład cppreference ([www.cppreference.com](http://www.cppreference.com)).

#### Autor o swojej pracy

Jestem absolwentem Informatyki na Wydziale Informatyki i Zarządzania Politechniki Wrocławskiej. Obecnie pracuję jako C++ Software Developer. W naszym dziale zajmujemy się pisaniem oprogramowania dla stacji bazowych sieci komórkowych. Do moich zainteresowań należy oczywiście programowanie, C++, szeroko pojęta informatyka oraz motoryzacja.

#### Artur Kochowski

Engineer, Software Development  
MBB Single RAN

# C++ Lambda

Robert Matusewicz  
Engineer, Software Development  
MBB FDD LTE

NOKIA

Jednym z najbardziej rozpoznawalnych mechanizmów nowego standardu języka C++ są lambdy. Umożliwiają one definiowanie anonimowych funktorów, tzw. domknąć, w miejscu, w którym mają zostać użyte – zwiększa to lokalność i czytelność kodu oraz zmniejsza jego objętość i koszt utrzymania. W pierwszej części tego artykułu przedstawiono problemy związane z brakiem opisywanego mechanizmu we wcześniejszych wersjach standardu języka C++. Druga część artykułu wyjaśnia, czym są lambdy, jak są definiowane i jak mogą zostać wykorzystane. W ostatniej części omówione zostały zmiany dotyczące funkcji anonimowych, wprowadzone w najnowszej części standardu (C++14).

## Potrzeba funkcji anonimowych

Kanonicznym przykładem pokazującym, dlaczego lambdy są potrzebne, jest kod zaprezentowany na [Listingu 1](#). Jego zadaniem jest obliczenie pierwiastka z sumy kwadratów dla liczb podanych w wektorach wejściowych i zapisanie ich do wektora wyjściowego. Często zdarza się, że w rzeczywistych projektach funktor wykorzystywany w funkcji `std::transform`, w tym wypadku `SquareNormalized`, jest definiowany w zupełnie innej części projektu, często w innym pliku. Zmniejsza to lokalność i czytelność kodu oraz wymusza częstą zmianę kontekstu podczas jego czytania. Warto też zwrócić uwagę na liczbę linii niepotrzebnego kodu – definiowanie klasy wraz z operatorem wywołania funkcji. Zwiększa to jego objętość oraz narzuca utrzymywanie.

## [Listing 1](#) Kod wykorzystujący ręcznie zdefiniowany funktor i funkcję `std::transform`

```
std::transform::
struct SquareNormalized

{
    double operator()(double x, double y) const
    {
        return sqrt(x*x + y*y);
    }
};

// some code here, then usage of functor
// in1, in2, out are vectors of double

std::transform(in1.begin(), in1.end(), in2.begin(),
              std::back_inserter(out),
              SquareNormalized());
```

## [Listing 2](#) Wykorzystanie funkcji `std::bind2nd` i `std::minus`

```
std::transform(input.begin(), input.end(),
              std::back_inserter(output),
              std::bind2nd(std::minus<int>(), 10));
```

Kolejnym przykładem uzasadniającym wprowadzenie anonimowych funkcji do języka C++ jest kod zademonstrowany na [Listingu 2](#). Jego zadaniem jest odjęcie wartości od każdej liczby znajdującej się w wektorze wejściowym, w tym przypadku liczby 10. Do jego realizacji wykorzystane zostały funktor szablonowy `std::minus` oraz funkcja `std::bind2nd`, przekształcająca dwuargumentowy `std::minus` na funktor jednoargumentowy, poprzez przypisanie drugiemu argumentowi stałej wartości. Problemem tego kodu jest to, że do zrealizowania czynności trywialnej zapragnięte zostały zaawansowane mechanizmy języka – konkretnie, `std::bind2nd`.

## Budowa lambdy

Czy przykłady zaprezentowane w poprzednich paragrafach można napisać lepiej? Okazuje się, że korzystając z lambd języka C++ — tak. [Listing 3](#) zawiera kod realizujący to samo zadanie, co kod umieszczonej na [Listingu 1](#), jest on jednak bardziej czytelny i nie wymusza przeglądania całego projektu

## [Listing 3](#) Kod z [Listingu 1](#) wykorzystujący lambdę

```
std::transform(in1.begin(), in1.end(), in2.begin(),
              std::back_inserter(out),
              [](double x, double y) { return std::sqrt(x*x - y*y); });
```

w celu znalezienia definicji użytego funktora. Podobnie jest z kodem zaprezentowanym na [Listingu 4](#) – poprzez zastąpienie wywołania funkcji szablonowych lambdą, jest on o wiele bardziej przejrzysty, a osoba go czytająca nie będzie miała wrażenia, że czynność trywialna jest realizowana przez zbyt skomplikowane mechanizmy.

## [Listing 4](#) Kod z [Listingu 2](#) wykorzystujący lambdę

```
std::transform(input.begin(), input.end(),
              std::back_inserter(output),
              [](int elem) { return elem - 10; });
```

Najprostsza lambda, zaprezentowana na [Listingu 5](#), składa się z trzech podstawowych elementów. Jej definicja rozpoczyna się od nawiasów kwadratowych, w których opcjonalnie mogą znajdować się nazwy przechwytywanych zmiennych (przechwytywanie zmiennych zostanie omówione w dalszej części artykułu). Po nawiasach kwadratowych następują nawiasy okrągłe, które zawierają listę parametrów przyjmowanych przez anonimową funkcję, a za nimi znajduje się definicja ciała funkcji.

### [Listing 5](#) Przykład najprostszej lambdy

```
[]() { return 5; }
```

W przypadku definicji lambdy z [Listingu 5](#), kompilator potrafi samodzielnie określić typ zwracany przez funkcję anonimową. Kompilator wydedukuje zwracany typ, gdy jest to możliwe (jeśli wszystkie wyrażenia `return` w ciele lambdy zwracają ten sam typ). [Listing 6](#) przedstawia kod, w którym jawnie zadeklarowany jest typ zwracany zdefiniowanej funkcji anonimowej.

### [Listing 6](#) Przykład lambdy z jawnie podanym typem zwracanym

```
[]() -> int { int x = 10; return x; };
```

Dla każdej napotkanej definicji lambdy, kompilator wygeneruje wewnętrzny typ, tzw. domknięcie – zaprezentowane na [Listingu 7](#). Standard określa, że ten typ ma mieć postać funktora, czyli musi posiadać zdefiniowany stały operator wywołania funkcji, z liczbą parametrów odpowiadającej liczbie parametrów przekazywanych do funkcji anonimowej.

### [Listing 7](#) Domknięcie generowane przez kompilator

```
//Możliwe domknięcie wygenerowane dla lambdy
// [x]() { return 5; }
class __InternalCompilerType1
{
public:
    int operator()() const
    {
        return 5;
    }
private:
    int x;
};
```

### Przechwytywanie zmiennych

W ciele lambdy można wykorzystać zmienne zdefiniowane poza jej zasięgiem. Służy do tego tak zwana capture-list, czyli lista przechwytywanych zmiennych. Standard C++11 pozwala na przechwytywanie zmiennych przez wartość oraz przez referencję. [Listing 8](#) prezentuje lambdę przechwytyującą dwie zmienne: `byVal` oraz `byRef`. Zmienna `byVal` przechwytywana jest przez wartość, natomiast zmienna `byRef` przechwytywana jest przez referencję. Zmienne przechwytywane przez referencję, poprzedzone są znakiem '&'. Istnieje kilka podstawowych zasad, o których należy pamiętać podczas przechwytywania zmiennych – zasady te zostały zebrane w [Tabeli 1](#).

**1 Tabela** Możliwe sposoby przechwytywania zmiennych.

Symbol	Znaczenie
[]	Nie przechwytyuj żadnej zmiennej
[=]	Przechwyć wszystkie zmienne przez wartość
[&]	Przechwyć wszystkie zmienne przez referencję
[x, &y]	Przechwyć zmienną x przez wartość a zmienny y przez referencję
[=, &x]	Przechwyć wszystkie zmienne, poza zmienną x, przez wartość; zmienną x przechwyć przez referencję
[&, x]	Przechwyć wszystkie zmienne, poza zmienną x, przez referencję; zmienną x przechwyć przez wartość
[this]	Przechwyć wskaźnik this przez wartość
[x, &x]	Niedozwolone – zmienna x jest przechwytywana jednocześnie przez wartość i referencję
[=, x]	Niedozwolone – wszystkie zmienne przechwytywane przez wartość oraz jawnie przechwycenie przez wartość zmiennej x
[&, &x]	Niedozwolone – wszystkie zmienne przechwytywane przez referencję oraz jawnie przechwycenie przez referencję zmiennej x

Warto pamiętać, że przechwycenie zmiennej przez wartość nie umożliwia jednocześnie modyfikacji jej wartości. Ta zasada stanie się bardziej jasna po ponownym spojrzeniu na strukturę domknięcia z [Listingu 7](#). Widać na niej, że kopia zmiennych przechwytywanych przez wartość jest przechowywana jako pole tej struktury, a wygenerowane domknięcie jest niemodyfikowalne. Istnieje możliwość poinstruowania kompilatora, aby wygenerował domknięcie umożliwiające modyfikowanie zmiennych przechwyconych przez wartość. Służy do tego słowo kluczowe `mutable`, które powinno być umieszczone bezpośrednio za nawiasem zamykającym listę parametrów lambdy, tak jak jest to pokazane na [Listingu 9](#). Standard nie specyfikuje, w jaki sposób przechowywane są zmienne przechwytywane przez referencję.

### [Listing 8](#) Przechwytywanie zmiennych przez lambę

```
int main()
{
    int byVal = 10, byRef = 15;
    auto lambda = [byVal, &byRef]() { return byVal + byRef; };
    return 0;
}
```

### [Listing 9](#) Lambda umożliwiająca modyfikowanie zmiennych przechwyconych przez wartość

```
int y = 10;
auto x = [y]() mutable { y = 15; return y; };
```

### Zmiany wprowadzone w standardzie C++14

Standard C++14 wprowadził dwie ważne modyfikacje do mechanizmu lambd. Pierwszą z nich jest wprowadzenie uogólnionego mechanizmu przechwytywania zmiennych (ang. lambda generalized capture), który umożliwia definiowanie lokalnych dla lambdy obiektów na liście przechwytywania zmiennych i inicjalizowania ich poprzez przyznanie do nich wartości zadanej wyrażenia. Ten mechanizm jest bardzo przydatny podczas przechwytywania zmiennych, których nie można skopiować, na przykład obiektów klasy `std::unique_ptr` – tak jak to widać na [Listingu 10](#). Na [Listingu 10](#) można także zobaczyć, że nowy mechanizm łamie zasadę mówiącą, że przechwycona zmienna musi nazywać się tak, jak zmienna przechwytywana. W wielu wypadkach ułatwia to czytanie oraz rozumienie kodu.

### [Listing 10](#) Mechanizm uogólnionego przechwytywania zmiennych

```
auto x = std::make_unique<int>(5);
int y = 10, z = 16;
auto lambda = [x = std::move(x), y2 = y, &x2 = x]() { /*...*/ };
```

Drugą ważną zmianą wprowadzoną w standardzie C++14 jest tak zwana uogólniona lambda (ang. generic lambda), widoczna na [Listingu 11](#). Jest to lambda, której typ parametrów jest określany przez kompilator podczas komplikacji programu. Od klasycznych funkcji anonimowych różni się tym, że na zdefiniowanej liście parametrów znajduje się przynajmniej jedna zmienna poprzedzona słowem kluczowym `auto`. Tego rodzaju lambdy są bardzo przydatne w niektórych algorytmach z biblioteki STL. Ich przydatność polega na tym, że jeżeli zmienimy typ przechowywanego obiektu w konte-

nerze, na którym uruchamiamy algorytm, to w wielu przypadkach nie będziemy musieli modyfikować lambdy, która w tym algorytmie jest wykorzystywana.

### [Listing 11](#) Uogólniona lambda

```
auto generic_lambda = [](auto p1, auto p1, int p3) { /*...*/ };
```

### Zakończenie

Lambdy języka C++ są z pewnością jednym z ciekawszych i bardziej użytecznych mechanizmów wprowadzonych do standardu. Ich wykorzystanie umożliwia pisanie krótszego, ładniejszego i łatwiejszego w utrzymaniu kodu. Mam nadzieję, że artykuł ten w jaśnił czytelnikowi, czym są lambdy oraz jak je stosować tak dobrze, aby stały się one narzędziem wykorzystywanym w codziennej pracy.

### Autor o swojej pracy

Jestem inżynierem oprogramowania w Nokii, gdzie pracuję nad warstwą kontrolną LTE, oraz współzałożycielem C++ User Group Wrocław. W moim dziale pracujemy nad takimi technologiami jak Carrier Aggregation, Load Balancing czy LPP. W naszej pracy wykorzystujemy TDD i najnowszy standard języka C++ do rozwoju oprogramowania.

### Robert Matusewicz

Engineer, Software Development  
MBB FDD LTE

# Zapytania o ekstremum w przedziale

Łukasz Grządko  
Engineer, Software Development  
MBB FDD LTE



Stacja przekaźnikowa nowoczesnej telefonii komórkowej LTE może obsługiwać jednocześnie bardzo wiele połączeń telefonicznych. Z każdym takim połączeniem związane są różne obliczenia. Jednym z nich jest pomiar czasu przesłania sygnału od stacji przekaźnikowej do telefonu i z powrotem (ang. round trip time). Stacja na tej podstawie może zliczać odległość od siebie do dowolnego użytkownika podłączonego do stacji. Kiedy użytkownik połączy się ze stacją, przydzielony mu zostanie unikalny identyfikator  $p$ , tj. kolejny element ze zbioru liczb naturalnych. Przypuśćmy, że stacja może obsługiwać jednocześnie co najwyżej  $n$  takich użytkowników. W czasie t do stacji przychodzą zapytania o najbliższego użytkownika w pewnym przedziale identyfikatorów. Przedział to para  $[left, right]$ , gdzie  $0 \leq left \leq right < n$ . Stacja ma za zadanie odpowiedzieć na wiele takich zapytań w chwili t. W dalszej części artykułu będziemy używać skróconego zapisu dla  $left$  oraz  $right$ , tj.  $l$  oraz  $r$  odpowiednio. Wszystkie kody źródłowe zamieszczone na listingach zostały napisane w języku C++ oraz korzystają z biblioteki standardowej.

Niech  $n = 16$  oraz odległości  $d$  będą zdefiniowane jak w [Tabeli 1](#).

Przykładowo w przedziale  $[0, 4]$  najmniejszy pomiar odległości wynosi 5.

Pierwszy naiwny algorytm polega na przeglądaniu wszystkich przedziałów i wcześniejszym spamiętaniu wyników dla każdej pary  $[l, r]$ . Zbadajmy algebraicznie jego złożoność. Złożoność obliczeniowa instrukcji w najgłębszej położonej pętli **for** ([Listing 1](#)) ograniczona jest przez pewną stałą  $c > 0$ . Celowo pomijam instrukcję między drugą, a trzecią pętlą. Nie będzie to miało wpływu na ostateczną złożoność obliczeniową algorytmu. Dalej mamy:

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i+1}^j c = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j-i)c = \\ \sum_{i=0}^{n-1} \sum_{j=1}^{n-i} c = \sum_{i=0}^{n-1} \frac{c(n-i)(n+1-i)}{2} = c \sum_{i=1}^n \frac{i^2+i}{2} = O(n^3).$$

Złożoność czasowa pojedynczego zapytania to oczywiście  $O(1)$ .

**1 Tabela**

$d[0]$	$d[1]$	$d[2]$	$d[3]$	$d[4]$	$d[5]$	$d[6]$	$d[7]$	$d[8]$	$d[9]$	$d[10]$	$d[11]$	$d[12]$	$d[13]$	$d[14]$	$d[15]$
11	12	5	41	16	28	32	49	81	42	21	17	42	33	67	90

Przykładowy kod na [Listingu 1](#) zapisuje wyniki w tablicy  $P$  dla wszystkich przedziałów  $[l, r]$ . Zapytanie o przedział  $[l, r]$ , dla  $l \leq r$ , to  $P[l][r]$ .

## Listing 1

```
for (int l=0; l<n; l++)
    for (int r=l; r<n; r++)
    {
        P[l][r] = d[l];
        for (int k=l+1; k<=r; k++)
        {
            if (P[l][r] > d[k])
                P[l][r] = d[k];
        }
    }
```

Szybszy algorytm, o złożoności czasowej  $O(n^2)$  można skonstruować za pomocą programowania dynamicznego. Złożoność pamięciowa jest nadal taka jak w poprzedniej wersji algorytmu. Złożoność czasowa pojedynczego zapytania to  $O(1)$ .

Zauważmy, że najmniejsza liczba w przedziale  $[l, r]$  to, albo  $d[r]$ , albo najmniejsza liczba w przedziale  $[l, r-1]$ , którą obliczyliśmy wcześniej, tj.  $dp[l][r-1]$ . Poniżej przedstawiamy wzór rekurencyjny.

$$dp[l][r] = \begin{cases} d[r], \text{ jeśli } d[r] < dp[l][r-1] \\ dp[l][r-1], \text{ w przeciwnym przypadku} \end{cases}$$

Tak jak to bywa w rozwiązaniach rekurencyjnych, nie możemy zapomnieć o przypadku brzegowym. Dla przedziału jednoelementowego  $[l, l]$ ,  $dp[l][l] = d[l]$ .

Kod realizujący to rozwiązanie znajduje się na [Listingu 2](#). Zapytanie o przedział  $[l, r]$ , to  $dp[l][r]$ .

## Listing 2

```
//przypadek brzegowy
for (int l=0; l<n; l++)
    dp[l][1]=d[1];

//właściwy dynamik
for (int l=0; l<n; l++)
    for (int r=l+1; r<n; r++)
    {
        dp[l][r]=min(dp[l][r-1], d[r]);
    }
```

Okazuje się jednak, że zużycie pamięci można znacząco ograniczyć. Zauważmy bowiem, że naszą tablicę można podzielić na  $O(\sqrt{n})$  części długości  $O(\sqrt{n})$ . W każdej takiej sąsiadującej części długości  $\sqrt{n}$  wyciągamy lokalne minimum.

Na naszym przykładzie mamy cztery takie części jak w [Tabeli 2a](#).

Funkcja `localMinimum` z [Listingu 3](#) oblicza lokalne minimum w każdym z  $\sqrt{n}$  przedziałów długości  $\sqrt{n}$  (poglądowy przykład w [Tabeli 2a](#)) i zapamiętuje je w osobnej tablicy (tablica `localMin` na [Listingu 3](#)). Koszt czasowy procedury wynosi  $O(\sqrt{n} * \sqrt{n}) = O(n)$ . Złożoność pamięciowa wynosi oczywiście  $O(\sqrt{n})$ . W poszukiwaniu minimum można wyróżnić dwa podstawowe przypadki. W pierwszym przypadku rozważmy przedział  $[2,13]$ . Wyszukiwanie minimum można podzielić na trzy fazy. W pierwszej fazie wyszukujemy minimum w tablicy wejściowej  $d$  tj. w przedziale zaznaczonym kolorem czerwonym. W drugiej fazie wyszukujemy podobnie, ale w przedziale niebieskim. W ostatniej fazie wyszukujemy minimum we wszystkich spójnych częściach długości  $\sqrt{16}$  koloru zielonego, dla których minimum zostało wyciągnięte wcześniej w tablicy `localMin`. W naszym przykładzie jest ich dokładnie 2. W ogólności jest ich co najwyżej  $\sqrt{n}$ . Stąd złożoność pojedynczego zapytania wynosi  $O(\sqrt{n})$ . W ogólności wspomnianą

**2a** Tabela 2a i 2b

d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	d[8]	d[9]	d[10]	d[11]	d[12]	d[13]	d[14]	d[15]
11	12	5	41	16	28	32	49	81	42	21	17	42	33	67	90

$\min[0,3] = 5$        $\min[4,7] = 16$        $\min[8,11] = 17$        $\min[12,15] = 33$

d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	d[8]	d[9]	d[10]	d[11]	d[12]	d[13]	d[14]	d[15]
11	12	5	41	16	28	32	49	81	42	21	17	42	33	67	90

## Listing 3

```
vector<int> localMinimum(vector<int> const& d)
{
    vector<int> localMin;
    int sqrtSize = static_cast<int>(sqrt(d.size()));
    for (int i = 0; i < sqrtSize; i++)
    {
        int offset = i * sqrtSize;
        int minInSqrtPart = d[offset];
        for (int j = 0; j < sqrt(d.size()); j++)
            minInSqrtPart = min(minInSqrtPart, d[offset + j]);
        localMin.push_back(minInSqrtPart);
    }
    return localMin;
}
```

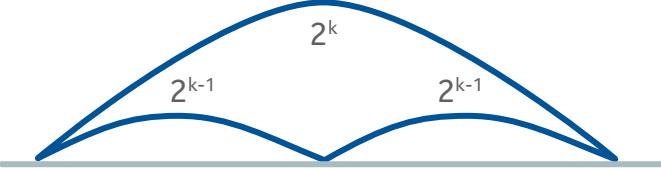
## Listing 4

```
int query(vector<int> const& d, vector<int> const& localMin,
          int left, int right)
{
    int sqrtSize = static_cast<int>(sqrt(d.size()));
    int startInLocalMin = (left + sqrtSize - 1) / sqrtSize;
    int endInLocalMin = right / sqrtSize - 1;
    int endOffFirstPart = startInLocalMin * sqrtSize - 1;
    int beginOfLastPart = endInLocalMin * sqrtSize;
    int minimum = d[left];
    for (int i = left; i <= min(right, endOffFirstPart); i++)
        minimum = min(minimum, d[i]);
    for (int i = max(left, beginOfLastPart); i <= right; i++)
        minimum = min(minimum, d[i]);
    for (int i = startInLocalMin; i <= endInLocalMin; i++)
        minimum = min(minimum, localMin[i]);
    return minimum;
}
```

nych faz może być mniej, kiedy rozważamy tylko przedział czerwony np.  $[2,3]$  lub tylko przedział niebieski np.  $[12,13]$  lub tylko spójne przedziały koloru zielonego np.  $[4,11]$  lub przedział koloru czerwono-niebieskiego np.  $[2,6]$  lub przedział koloru czerwono-zielonego np.  $[2,7]$  lub ostatecznie przedział koloru zielono-niebieskiego np.  $[4,13]$ . W drugim przypadku (patrz [Tabela 2b](#)) rozważamy przedział całkowicie zawarty w spójnym fragmencie długości  $\sqrt{n}$ . Oznaczony jest kolorem żółtym. W tym przypadku wyszukujemy minimum wprost w tablicy  $d$ .

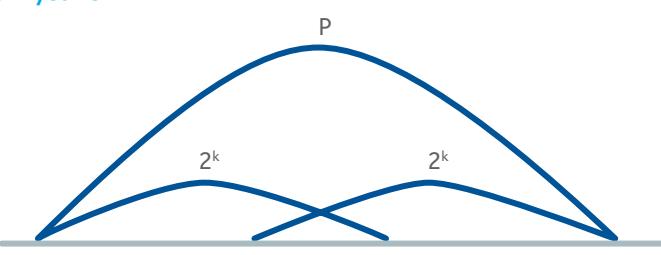
Pełny kod funkcji realizującej zapytanie zamieszczony jest na [Listingu 4](#).

## 1 Rysunek



Pokażemy algorytm, który wymaga  $O(n \log_2 n)$  pamięci, a zapytania realizuje w czasie stałym. Konstrukcja tablicy polega na wyliczaniu pośrednich wyników dla podtablicy długości  $2^k$ . Wyniki będziemy przechowywać w tablicy  $dp[0, n-1][0, \log_2 n]$ .  $dp[i][k]$  to najmniejsza wartość w podtablicy  $d$  długości  $2^k$  zaczynającej się pod indeksem  $i$ . Dla  $k=0$  i dowolnego  $i < n$ , mamy oczywiście  $dp[i][0] = d[i]$ . Dla  $k > 0$  spójny fragment długości  $2^k$  to dwa sąsiadujące ze sobą fragmenty długości  $2^{k-1}$ , przedstawione na [Rysunku 1](#). Wtedy  $dp[i][k] = \min(dp[i][k-1], dp[i+2^{k-1}][k-1])$ . Tablicę  $dp$  obliczamy dla każdego  $k$ , takiego że  $2^k \leq n$ . Przykładowa funkcja realizująca to zadanie jest na [Listingu 5](#).

## 2 Rysunek



## 3 Tabela

d[0]	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	d[8]	d[9]	d[10]	d[11]	d[12]	d[13]	d[14]	d[15]
11	12	5	41	16	28	32	49	81	42	21	17	42	33	67	90

$dp[2][1]$        $dp[4][1]$

$dp[2][2] = \min(dp[2][1], dp[2+2][1])$

## Listing 5

```
vector<vector<int>> preProcess(vector<int> const& d)
{
    vector<vector<int>> dp;
    dp.resize(d.size());
    for (int i = 0; i < d.size(); i++)
        dp[i].push_back(d[i]);
    for (int k = 1; (1 << k) <= d.size(); k++)
        for (int i = 0; i + (1 << k) <= d.size(); i++)
            dp[i].push_back(min(dp[i][k - 1],
                                dp[i + (1 << (k - 1))][k - 1]));
    return dp;
}
```

## Listing 6

```
int query(vector<vector<int>> const& dp,
          int left, int right) //przedział P to para [left, right]
{
    const int k = static_cast<int>(log2(right - left + 1));
    return min(dp[left][k], dp[right - (1 << k) + 1][k]);
}
```

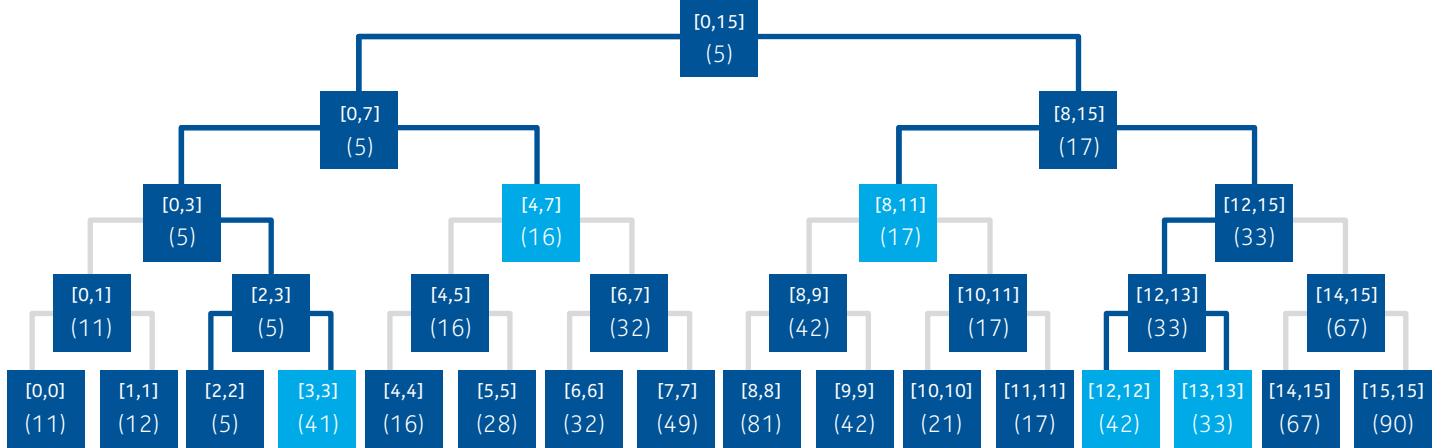
Przykład obliczenia  $dp[2][2]$  znajduje się w [Tabeli 3](#).

Aby teraz odpowiedzieć na zapytanie  $P$ , tj. znaleźć najmniejszą wartość w przedziale  $P$ , wystarczy znaleźć najdłuższy przedział długości  $2^k$ , który mieści się w przedziale  $P$ . Jeśli długość przedziału jest potęgą 2 to podajemy wartość wcześniejszej obliczoną w tablicy  $dp$ . W przeciwnym razie podajemy minimum z dwóch przedziałów tej samej długości, które nachodzą na siebie, tak jak na [Rysunku 2](#). Zapytanie realizuje funkcja `query` z [Listingu 6](#).

Podamy teraz nierekurencyjny algorytm, którego koszt pamięciowy wynosi  $O(n)$ , koszt zapytania wynosi  $O(\log_2 n)$ , zaś koszt zbudowania

tablicy wynosi  $O(n \log_2 n)$ . W tablicy będzie ukryte statyczne drzewo binarne, tzw. drzewo przedziałowe. Węzły drzewa są ponumerowane poziomami, tak jak w kopcu binarnym ([Rysunku 3](#)). Korzeń drzewa otrzymuje numer 1, synowie od lewej kolejno 2 oraz 3 itd. W ogólności synowie węzła o numerze  $v$  mają numery  $2^k v$  oraz  $2^k v + 1$ . Ojciec węzła o numerze  $v$  ma numer  $v/2$ . Na [Rysunku 3](#) liczby w nawiasach okrągłych w danym węźle drzewa oznaczają minimum z dwóch jego dzieci. W nawiasach kwadratowych jest przedział zwany bazowym, który dany węzeł reprezentuje. Na rysunku mamy już zbudowane całe drzewo dla 16 elementów. Łatwo zauważyc, że drzewo zajmuje tylko liniową pamięć. Jak wygląda wstawienie elementu? Zauważmy, że liściom drzewa odpowiadają elementy w tablicy o indeksach od 16 do 16 + 15. Jeśli wstawiamy element  $d[3] = 41$ , umieszczamy go w tablicy pod adresem 16 + 3. Następnie idziemy w góre drzewa i aktualizujemy minimum w każdym napotkanym węźle aż do korzenia. Ich indeksy łatwo obliczyć, a są nimi kolejno 19,  $19/2 = 9$ ,  $9/2 = 4$ ,  $4/2 = 2$ ,  $2/2 = 1$  (korzeń drzewa). Złożoność czasowa wstawienia do drzewa jest wprost proporcjonalna do długości ścieżki od liścia do drzewa, a więc  $O(\log_2 n)$ . Całkowita złożoność wstawienia wszystkich elementów jest zatem równa  $O(n \log_2 n)$ . Zapytanie o przedział jest już bardziej skomplikowane. Po więcej szczegółów odsyłamy do [4]. W przykładzie na [Rysunku 3](#) pytamy o przedział [3,13]. Musimy go rozłożyć na przedziały bazowe, które reprezentowane są przez węzły drzewa. Przedział [3,13] algorytm rozkłada na sumę mnogościową: [3,3] U [4,7] U [8,11] U [12,12] U [13,13]. Zapytanie rozpoczynamy na dwóch krańcach przedziału tj. w liściach [3,3] oraz [13,13]. Następnie idziemy w góre drzewa. Jeśli węzeł wewnętrzny drzewa ma brata po swojej prawej stronie to dodajemy ten węzeł do rozkładu, w tym przypadku jest to węzeł nr 5 dla bazowego przedziału [4,7]. Z drugiej strony postępujemy dualnie, tj. idąc od węzła reprezentującego prawy koniec przedziału patrzmy czy ma brata po swojej lewej stronie. Jeśli tak, dodajemy węzeł do rozkładu, tj. węzeł nr 28 oraz 6. Sprawdzanie czy węzeł jest bratem sprawdza się do obliczania parzystości numeru węzła w drzewie. Kod wsta-

### 3 Rysunek



wania elementu i zapytania o przedział znajduje się na [Listingu 7](#). Dla zapytania liczba przedziałów w rozkładzie jest rzędu  $O(\log_2 n)$ . Stąd zapytanie o przedział ma złożoność  $O(\log_2 n)$ . Parametr  $idx \geq 0$  funkcji `insert` z [Listingu 7](#) oznacza index liścia w drzewie licząc od lewej do prawej. Aktualizujemy go o  $m = 2^k$  tj. najmniejsze  $k$ , że  $2^k \geq n$ . Pierwsze  $m$  elementów tablicy to wewnętrzne węzły drzewa. Parametr `value` to wartość wstawianego elementu do drzewa `tree` (tablica  $2^m$ -elementowa, początkowo elementy zainicjowane na jakąś dużą dodatnią wartość, np. `numeric_limits<int>::max()`) pod indeksem  $m + idx$ . W funkcji `query` parametry  $m$  oraz `tree` mają to samo znaczenie. Parametry `left` oraz `right` to indeksy w tablicy `d` krańców przedziału, w którym szukamy minimum.

Zauważmy, że algorytmy o złożoności logarytmicznej zakładają rozmiar danych jako potęgę 2. Algorytm o złożoności pierwiastkowej zakłada, że  $n$  jest liczbą postaci  $k^2$  dla pewnego całkowitego  $k$ . Zostawiamy jako proste zadanie dla Czytelnika modyfikację algorytmów by obsługiwać dowolny rozmiar tablicy. Okazuje się, że istnieje jeszcze szybszy, optymalny algorytm. Polega on na redukcji problemu wyszukiwania minimum w przedziale (ang. Range Minimum Query) do problemu szukania najbliższego wspólnego przodka (ang. Lowest Common Ancestor) dwóch wierzchołków w drzewie ukorzenionym. Redukcja ta ma złożoność  $O(n)$ . Natomiast czas pojedynczego zapytania wynosi  $O(1)$ . Opis tego algorytmu można znaleźć w [1] i [2]. Warto też wspomnieć o bibliotece Boost ICL [7], która implementuje wstawianie i usuwanie przedziałów w czasie logarytmicznym. Jednakże biblioteka ta realizuje drzewo pokryciowe, stąd może nie mieć zastosowania do wyszukiwania minimum w przedziale. Z drugiej strony, takich drzew można użyć w rozwiązyaniu pewnych problemów geometrii obliczeniowej. Więcej o drzewach pokryciowych można przeczytać w [5] i [6].

W [Tabeli 4](#) przedstawiamy porównanie wszystkich omówionych algorytmów pod względem ich złożoności czasowej oraz pamięciowej.

### 4 Tabela

	Złożoność czasowa konstrukcji struktury danych	Złożoność czasowa pamięciowa	Złożoność czasowa zapytania
Algorytm naiwny	$O(n^3)$	$O(n^2)$	$O(1)$
Algorytm kwadratowy	$O(n^2)$	$O(n^2)$	$O(1)$
Algorytm pierwiastkowy	$O(n)$	$O(\sqrt{n} + n) = O(n)$	$O(\sqrt{n})$
Algorytm logarytmiczny	$O(n \log n)$	$O(n \log n)$	$O(1)$
Algorytm na drzewie przedziałowym	$O(n \log n)$	$O(n)$	$O(\log n)$

### Listing 7

```
void insert(vector<int>& tree, int m, int idx, int value)
{
    const int root = 1;
    int treeIndex = m + idx;
    tree[treeIndex] = value;
    while (treeIndex != root)
    {
        treeIndex /= 2;
        tree[treeIndex] = min(tree[treeIndex * 2],
                             tree[treeIndex * 2 + 1]);
    }
}

int query(int left, int right, vector<int> const& tree, int n)
{
    int treeLeftIndex = m + left, treeRightIndex = m + right;
    int result = min(tree[treeLeftIndex], tree[treeRightIndex]);
    while (treeLeftIndex / 2 != treeRightIndex / 2)
    {
        if (treeLeftIndex % 2 == 0) result = min(result,
                                                tree[treeLeftIndex + 1]);
        if (treeRightIndex % 2 == 1) result = min(result,
                                                tree[treeRightIndex - 1]);
        treeLeftIndex /= 2, treeRightIndex /= 2;
    }
    return result;
}
```

### Bibliografia:

- [1] Radoszewski J., O dwóch równoważnych problemach, „Delta” 2007, nr 9.
- [2] Radoszewski J., Problem RMQ, „Delta” 2007, nr 11.
- [3] Topcoder Tutorial, Range Minimum Query and Lowest Common Ancestor, by danielp, <http://www.topcoder.com/tc?d1=tutorial-s&d2=lowestCommonAncestor&module=Static>.
- [4] Radoszewski J., Wykład nr 2. Drzewa Przedziałowe, [w:] Wykłady z algorytmiki stosowanej, <[was.zaa.mimuw.edu.pl/](http://was.zaa.mimuw.edu.pl/)>.
- [5] Stańczyk P., Algorytmika praktyczna w konkursach informatycznych, Warszawa 2007.
- [6] Stańczyk P., Algorytmika praktyczna. Nie tylko dla mistrzów, Warszawa PWN 2009.
- [7] [http://www.boost.org/doc/libs/1\\_56\\_0/libs/icl/doc/html/index.html](http://www.boost.org/doc/libs/1_56_0/libs/icl/doc/html/index.html).

### Autor o swojej pracy

Jestem absolwentem Informatyki na Wydziale Matematyki i Informatyki Uniwersytetu Wrocławskiego. Pracuję na stanowisku Software Engineer w dziale MBB LTE C-Plane. Nasz dział R&D tworzy oprogramowanie na nowoczesne stacje przekaźnikowe. Warstwa naszej aplikacji kontroluje połączenia stacji z urządzeniem mobilnym na bazie wymiany wiadomości poprzez łącze komunikacyjne. Ze względu na mnogość urządzeń do naszych zadań należy również poprawa wydajności sieci. To ciekawa praca pełna wyzwań i odkrywania możliwości standardu C++11.

### Łukasz Grządko

Engineer, Software Development  
MBB FDD LTE

# Narzędzia programistyczne

2.1

**Wojciech Razik**  
System kontroli wersji: Git czy SVN?

40

2.2

**Kamil Pawłowski**  
Zastosowanie systemu operacyjnego  
Linux w firmie Nokia Networks

44

2.3

**Marcin Zawadzki**  
Statyczna/dynamiczna analiza kodu

50

2.4

**Marcin Załuski**  
Wyrażenia regularne

56

2.5

**Tomasz Drwięga**  
Jak przeszliśmy od Javy do JavaScript?

60

# System kontroli wersji: Git czy SVN?

Wojciech Razik  
C++ Software Developer  
MBB Single RAN CP/TUPC

**NOKIA**

Gdy zaczyna się przygoda z programowaniem, bardzo często zdarza się, że chcemy ulepszyć nasz kod. Na koniec okazuje się jednak, że zamiast dodać funkcjonalność, popsuliśmy wcześniej działającą wersję. Z biegiem czasu, naturalne staje się archiwizowanie działających wersji, kopiowanie ich do innych folderów lub też wgrywanie na serwer.

Pracując w korporacji zatrudniającej tysiące programistów, pliki zmieniają się bardzo często i w znacznym stopniu. Bywa tak, że nad jednym plikiem może pracować kilkanaście osób. W takiej sytuacji, konieczne jest używanie narzędzia ułatwiającego pracę – korzysta się z systemów kontroli wersji. W Nokia korzystamy głównie z dwóch systemów – Git oraz Subversion (SVN).

## Subversion

Subversion jest to darmowy system kontroli wersji bardzo po-wszechnie wykorzystywany zarówno w warunkach domowych, jak i w dużych projektach. Wyróżnia się on bardzo łatwą i intuicyjną ob-slugą oraz świetnym graficznym klientem o nazwie TortoiseSVN.

Aby zacząć pracę z istniejącym repozytorium, najpierw robimy tzw. „checkout”, czyli pobieramy wszystkie pliki znajdujące się na serwerze na nasz twardy dysk. Warto tutaj nadmienić, że ściągana jest tylko najnowsza wersja wszystkich plików – chcąc obejrzeć historię, musimy być podłączeni do odpowiedniego serwera (są sposoby trzymania historii lokalnie, jednak nie będą tutaj omawiane).

Mając już ściągnięty kod, możemy zacząć pracę i modyfikować nasz projekt. Po dokonaniu zmian, należy wgrać zmiany na serwer – nazywa się to „commitem”. Zanim jednak to się stanie, sprawdzane jest czy operacja jest możliwa – między innymi czy posiadamy aktualne wersje wszystkich plików lub np. czy ktoś nie założył blokady na pliku. Samo wgranie na serwer jest operacją atomową – albo została wprowadzone wszystkie zmiany albo nic się w repozytorium nie zmieni. Każdy commit można opisać, dodając tzw. „commit message” – czyli krótką wiadomość opisującą, co zostało zmienione. Pozwala to uporządkować w wygodny sposób historię wprowadzanych zmian oraz umożliwia szybkie odnajdywanie wybranych commitów.

Problem pojawia się, gdy natrafiamy na konflikt – jest to taka sytuacja, gdy na serwerze zmienił się plik, który my również zmodyfikowaliśmy. Należy wtedy zrobić „merge'a” – czyli połączyć zmiany lokalne ze zmianami, które zostały zrobione na serwerze. Jeśli zmiany są w dwóch różnych miejscach w pliku, Subversion będzie potrafił automatycznie dokonać zmian. W przeciwnym wypadku będziemy musieli zrobić merge'a ręcznie. SVN trochę nam w tym pomaga – po wykryciu konfliktu na dysku dostępne będą cztery wersje pliku: lokalna, najnowsza znajdująca się na serwerze, wersja bazowa, na której zaczęliśmy swoją pracę (ale bez naszych zmian) oraz plik będący połączeniem lokalnej wersji i tej z serwera – dzięki odpowiednim znacznikom w pliku widoczne są miejsca konfliktu, które możemy w prosty sposób wyedytować.

Kolejną ważną funkcjonalnością jest tworzenie własnych wersji danego oprogramowania – czyli tzw. branchy. Jest to nic innego jak skopiowanie danej wersji repozytorium w inne miejsce. Stworzony branch możemy do woli modyfikować, robić commity, a po zakończeniu pracy połączyć go z główną gałęzią. Całe repozytorium możemy przedstawić jako graf, gdzie każdy węzeł reprezentuje jeden commit. Na [Rysunku 1](#) zaprezentowano ogólny schemat – na głównym branchu dokonano jednej modyfikacji (r1), następnie utworzono nowy branch (r2) i nannesiono na niego zmiany (r4). W międzyczasie, na serwerze ktoś dokonał zmian r3 i r5. Chcąc połączyć naszą wersję z wersją główną, musimy dokonać merge'a (r6). Bardzo ważną właściwością Subversion jest to, że bez ingerencji w serwer nie ma możliwości zmiany historii repozytorium. Tworząc nowy branch, SVN już zawsze będzie go pamiętał. Ma to oczywiście swoje wady i zalety. Z punktu widzenia osób zajmujących się integracją jest to wygodniejsze – nie ma możliwości utracenia wprowadzonych przez kogoś zmian.

Najważniejszą cechą charakterystyczną Subversion jest zcentralizowana struktura. Dzięki temu łatwiejsze jest zarządzanie powstawaniem kolejnych wersji oprogramowania, wadą jest to, że trudniej jest pracować nad jednym projektem, gdy w projekcie jest zaangażowanych więcej osób. Ponadto ogranicza to możliwość tworzenia i łączenia branchy – nie ma na przykład możliwości zmiany wersji bazowej stworzonego brancha.

## Git

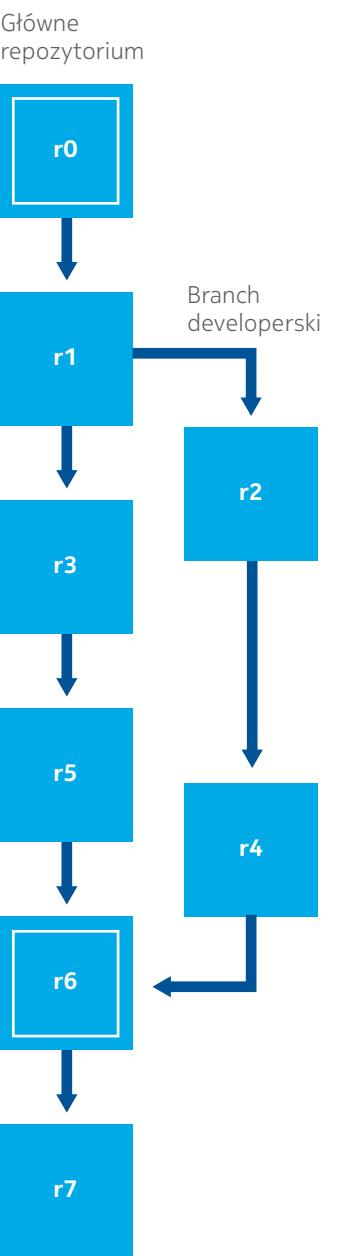
Drugim używanym przez Nokię systemem kontroli wersji jest Git. Został on stworzony m.in. przez słynnego Linusa Torvaldsa, odpowiedzialnego za jądro Linuxa. Oferuje on zupełnie inne podejście do systemów kontroli wersji niż Subversion.

Aby pobrać wszystkie pliki z serwera, używa się polecenia „clone” – wraz z plikami, pobierana jest również historia zmian. Możemy oglądać ją nawet nie mając podłączenia do sieci. Offline można również tworzyć własne, lokalne branche i pracować na nich do woli. Jedyne, do czego potrzebujemy sieci to uaktualnianie naszego repozytorium („git pull”) oraz wgrywanie zmian do głównego repozytorium („git push”).

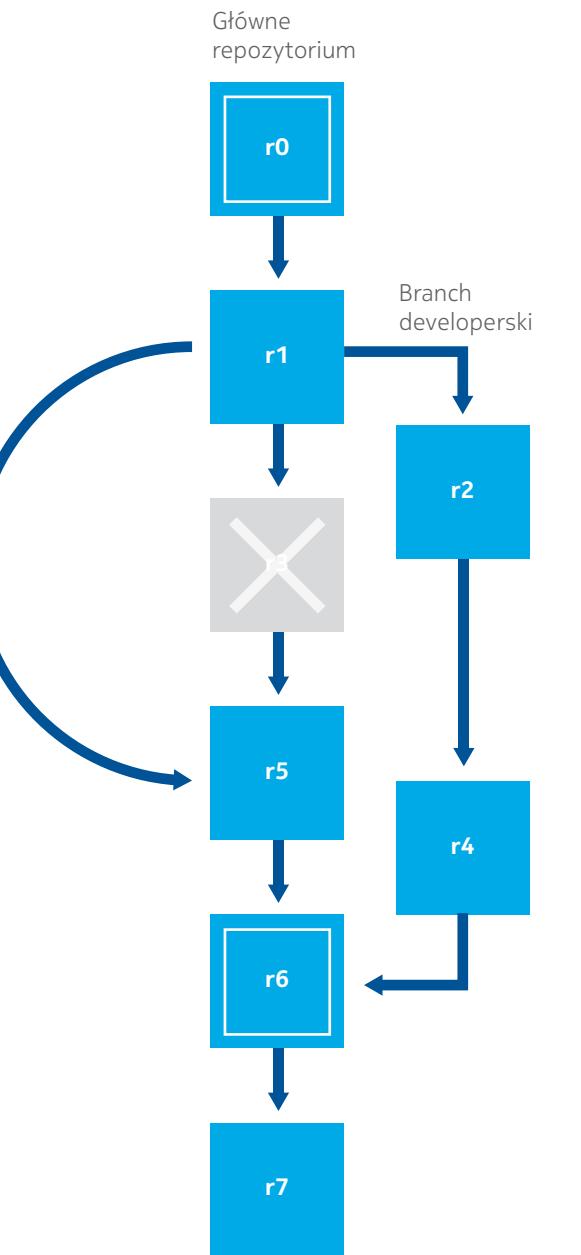
Git, dzięki temu że trzyma więcej informacji na dysku twardym, jest w działaniu wielokrotnie szybszy. Również jego możliwości są dużo większe – łączenie branchy, zmiana historii commitów czy też ich dwuwolna aranżacja (przykład – [Rysunek 2](#)). Właśnie ta funkcjonalność niesie z sobą ryzyko uszkodzenia repozytorium – możliwe jest usunięcie węzła, który posłużył do stworzenia brancha. Powoduje to, że można utracić pracę z kilku tygodni.

Do zalet Gita należy również większa elastyczność w pracy z lokalną wersją repozytorium. Niestety, Git jest przez to bardziej złożony i trudniejszy w obsłudze. Trudno jest początkującemu rozróżnić „checkouta” z „clonem”, czy też „committa” z „pushem”. Warto też wspomnieć, że Git posiada dużo lepsze wsparcie dla merge'owania, w tym kilka różnych algorytmów robiących to inteligentnie za nas.

**1 Rysunek** Graf reprezentujący SVN-a



**2 Rysunek** Graf reprezentujący Gita



Na korzyść Gita przemawia jeszcze prostota konfiguracji. Gdy chcemy wersjonować tylko naszą pracę, wystarczy jedno polecenie konsolowe „git init” i repozytorium jest gotowe do pracy. Korzystając z Subversion, początkowa konfiguracja jest dużo bardziej - trzeba znaleźć odpowiedni serwer, należy też zadbać o stworzenie odpowiednich użytkowników i nadać im uprawnienia. Cała operacja jest, więc dużo bardziej czasochłonna i problematyczna. W przypadku większych projektów szanse się wyrównują – Git również wymaga ustawienia odpowiednich uprawnień i konfiguracja wygląda bardzo podobnie.

Zatem który system kontroli wersji jest lepszy? Tak naprawdę żaden – oba są zupełnie inne i lepiej nadają się do innych zastosowań. Git jest szybszy i ma większe możliwości, Subversion jest za to dużo prostszy i bardziej intuicyjny. Patrząc na statystyki [1], obecnie bardziej popularny jest SVN (48%), Git stanowi natomiast 37% wszystkich repozytoriów.

Mimo, że systemy kontroli wersji stosuje się głównie do trzymania kodu źródłowego, nic nie stoi na przeszkodzie, by wykorzystywać je do trzymania np. dokumentów. Coraz więcej studentów właśnie w ten sposób decyduje się na wersjonowanie swoich prac dyplomowych. Warto pamiętać, że Gita i SVN-a nie powinno się używać do trzymania dużych plików binarnych. Szczególnie duże ryzyko niesie z sobą merge'owanie takich plików, również wydajność takiego rozwiązania pozostawia wiele do życzenia. W takiej sytuacji najlepiej skorzystać z systemu do tego dedykowanego – jak np. google boar. Tak naprawdę nie ma jednak znaczenia, czy zdecydujemy się na Gita czy też SVN-a – samo używanie jakiegokolwiek systemu kontroli wersji jest krokiem milowym do profesjonalnej pracy z kodem źródłowym.

#### Bibliografia:

- [1] “Compare repositories”, Black Duck Software, Inc.,  
<https://www.openhub.net/repositories/compare>.

#### Autor o swojej pracy

Pracuję na stanowisku Software Developer w dziale MBB WCDMA TCOM. Zajmujemy się tworzeniem nowoczesnego oprogramowania na stację bazową, czyli na pierwszy element sieci, z którym łączy się telefon komórkowy. Nasz kod odpowiada między innymi za przydzielenie użytkownikowi odpowiednich zasobów. Dzięki nam telefony alarmowe obsługiwane są z najwyższym priorytetem. W naszym dziale najważniejsza jest jakość. Każdy fragment kodu jest wielokrotnie sprawdzany i testowany. Tworzymy oprogramowania na wysokim poziomie w standardzie C++11.

**Wojciech Razik**  
C++ Software Developer  
MBB Single RAN CP/TUPC

# Zastosowanie systemu operacyjnego Linux w firmie Nokia Networks

Kamil Pawłowski  
C++ Software Developer  
MBB SingleRAN O&M



## Linux jako system wszechstronnego zastosowania

Od czasu swojej premiery (1991), jądro Linuksa przeszło długą drogę rozwoju, rozpowszechniając się na komputerach PC i innych platformach sprzętowych. Są dwie przyczyny tego zjawiska. Pierwsza, najbardziej oczywista, to otwarty kod źródłowy samego jądra, jak i warstwy użytkownika. Umożliwiło to Linuksowi zebranie wokół siebie rzeszy hakerów, którzy rozwijali ten system, dostosowywali do obsługi kolejnych generacji sprzętu, oraz przesuwali granicę jego możliwości coraz to dalej.

To jednak było wciąż za mało, by system tworzony przez pasjonatów stał się jednym z najbardziej rozpowszechnionych i uniwersalnych systemów operacyjnych na świecie. Potrzebne było jeszcze uzasadnienie komercyjne. Dziś jest wiele firm które oferują swoje komercyjne Linuksy, już nie tylko na stacje robocze, ale również na urządzenia wbudowane.

Obecnie Linuksa możemy spotkać dosłownie wszędzie, nie tylko na maszynach serwerowych, również na komputerach osobistych, komputerach firmowych, na urządzeniach z mikrokontrolerami, telewizorach, telefonach. Linuksa możemy spotkać także jako system operacyjny na stacjach bazowych (Base Transceiver Stations) firmy Nokia Networks, które zapewniają łączność telekomunikacyjną w technologiach 2G, 3G oraz 4G dla ponad miliarda użytkowników na całym świecie.

W dalszej części tego rozdziału, opiszę jak firma Nokia Networks wykorzystuje system Linux przy rozwijaniu i testowaniu oprogramowania.

## Linux w firmie Nokia Networks

Każdy developer oprogramowania w firmie Nokia Networks, na co dzień styka się z systemem operacyjnym Linux. Pod kontrolą tego systemu działają maszyny Continuous Integration, stacje bazowe (BTS-y), serwery oraz komputery pracowników. Nawet jeśli pracownik nie posiada Linuksa zainstalowanego na swoim laptopie, może korzystać z sieci serwerów działających pod kontrolą tego systemu, czyli serwerów LinSEE o których więcej opowiem później.

Nokia Networks udostępnia specjalnie przygotowane środowisko dedykowane dla Linuksa, zwane dalej LinSEE. Zapewnia ono komplet narzędzi potrzebnych do pracy developera. Od standardowych takich jak VIM, QTCreator, Eclipse, kompilator GCC czy debugger, po specjalnie przygotowane wersje zamkniętych aplikacji używanych np. do modelowania maszyny stanów.

W zależności od projektu pracownik firmy może mieć możliwość wyboru Linuksa jako systemu operacyjnego, który zainstaluje na swoim komputerze służbowym. W przypadku projektu do którego jestem przydzielony istnieje taka możliwość. Oficjalnie wspieraną dystrybucją w tym przypadku jest NSN Enterprise Linux. Jest to specjalna wersja systemu Red Hat Enterprise Linux dostosowana do codziennej pracy w firmie Nokia Networks. Nie jest to jednak jedyna opcja,

pracownik ma wolność wyboru a oficjalne repozytoria pakietów są kompatybilne z najpopularniejszymi dystrybucjami takimi jak Fedora czy Ubuntu.

## System operacyjny dla stacji bazowej

Oprogramowanie na stacje bazowe, zwane dalej BTS (Base Transceiver Station) tworzone jest w większości we Wrocławskim R&D firmy Nokia Networks. Firma kreatywnie wykorzystuje możliwości i wszechstronność systemu operacyjnego Linux. Dlatego od roku 2015, oprogramowanie dla wszystkich produkowanych BTS-ów firma działa pod kontrolą systemu operacyjnego Linux. Również w latach wcześniejszych część sprzętu już działała pod kontrolą Linuksa, jednak rok 2015 jest pod tym względem przełomowy, ponieważ system ten znajduje się w każdej nowo wydawanej paczce oprogramowania na BTS-y.

### 1 Ilustracja

Plik	Edycja	Widok	Wyszukiwanie	Terminal	Pomoc
5273 005173	27.10.16:57:34.640	[192.168.255.35]	77 FSP-1400-0-TCOMexe <01.01 01:02:45.485413>	B8 INF/	
5274 005172	27.10.16:57:34.640	[192.168.255.35]	78 FSP-1400-0-TCOMexe <01.01 01:02:45.485941>	B8 INF/	
5275 005173	27.10.16:57:34.640	[192.168.255.35]	79 FSP-1400-0-TCOMexe <01.01 01:02:45.485982>	B8 INF/	
5276 005174	27.10.16:57:34.640	[192.168.255.163]	9e FSP-1351 <01.01 01:02:40.400734>	22 INF/RAKEMASTE	
5277 005175	27.10.16:57:34.671	[192.168.255.35]	83 FSP-1400-0-TCOMexe <01.01 01:02:45.546996>	B8 INF/	
5278 005176	27.10.16:57:34.671	[192.168.255.35]	84 FSP-1400-0-TCOMexe <01.01 01:02:45.547146>	B8 INF/	
5279 005177	27.10.16:57:34.671	[192.168.255.35]	85 FSP-1400-0-TCOMexe <01.01 01:02:45.547320>	B8 INF/	
5280 005178	27.10.16:57:34.671	[192.168.255.171]	9e FSP-1431 <01.01 01:02:40.399649>	22 INF/RAKEMASTE	
5281 005179	27.10.16:57:34.687	[192.168.255.1]	83 FSP-1400-0-TCOMexe <01.01 01:02:45.546896>	B8 INF/T	
5282 005180	27.10.16:57:34.687	[192.168.255.1]	84 FSP-1400-0-TCOMexe <01.01 01:02:45.547146>	B8 INF/T	
5283 005181	27.10.16:57:34.687	[192.168.255.1]	85 FSP-1400-0-TCOMexe <01.01 01:02:45.547320>	B8 INF/T	
5284 005182	27.10.16:57:34.687	[192.168.255.1]	^M		
5285 005182	27.10.16:57:34.687	[192.168.255.164]	99 FSP-1361 <01.01 01:02:40.401136>	22 INF/RAKEMASTE	
5286 005183	27.10.16:57:34.703	[192.168.255.1]	4c FCM-1011-0-BTSMEx <01.01 01:02:23.0360648>	10C DBG/	
5287 005184	27.10.16:57:34.703	[192.168.255.1]	4d FCM-1011-0-BTSMEx <01.01 01:02:23.0360670>	10C DBG/	
5288 005185	27.10.16:57:34.703	[192.168.255.1]	4e FCM-1011-0-BTSMEx <01.01 01:02:23.0366756>	126 DBG/	
5289 005186	27.10.16:57:34.703	[192.168.255.1]	4f FCM-1011-0-BTSMEx <01.01 01:02:23.0366671>	126 DBG/	
5290 005187	27.10.16:57:34.703	[192.168.255.1]	50 FCM-1011-0-BTSMEx <01.01 01:02:23.0365715>	126 DBG/	
5291 005188	27.10.16:57:34.703	[192.168.255.1]	51 FCM-1011-0-BTSMEx <01.01 01:02:23.0366776>	126 DBG/	
5292 005189	27.10.16:57:34.703	[192.168.255.1]	52 FCM-1011-0-BTSMEx <01.01 01:02:23.0366801>	126 DBG/	
5293 005190	27.10.16:57:34.703	[192.168.255.166]	9e FSP-1381 <01.01 01:02:40.398681>	22 INF/RAKEMASTE	
5294 005191	27.10.16:57:34.734	[192.168.255.35]	8f FSP-1400-0-TCOMexe <01.01 01:02:45.485413>	B8 INF/	
5295 005192	27.10.16:57:34.734	[192.168.255.35]	9e FSP-1400-0-TCOMexe <01.01 01:02:45.499790>	B8 INF/	
5296 005193	27.10.16:57:34.734	[192.168.255.35]	9f FSP-1400-0-TCOMexe <01.01 01:02:45.500000>	B8 INF/	
5297 005194	27.10.16:57:34.734	[192.168.255.35]	92 FSP-1400-0-TCOMexe <01.01 01:02:45.600193>	B8 INF/	
5298 005195	27.10.16:57:34.734	[192.168.255.151]	9b FSP-1231 <01.01 01:02:40.394931>	22 INF/RAKEMASTE	
5299 005196	27.10.16:57:34.750	[192.168.255.1]	69 FCM-1011-0-BTSMEx <01.01 01:02:23.0369627>	124 DBG/	
5300 005197	27.10.16:57:34.750	[192.168.255.1]	6a FCM-1011-0-BTSMEx <01.01 01:02:23.0369735>	10C INF/	
5301 005198	27.10.16:57:34.750	[192.168.255.1]	6b FCM-1011-0-BTSMEx <01.01 01:02:23.0369866>	129 DBG/	
5302 005199	27.10.16:57:34.750	[192.168.255.1]	6c FCM-1011-0-BTSMEx <01.01 01:02:23.0369953>	129 DBG/	
5303 005200	27.10.16:57:34.750	[192.168.255.1]	6d FCM-1011-0-BTSMEx <01.01 01:02:23.0700044>	129 DBG/	
5304 005201	27.10.16:57:34.750	[192.168.255.1]	6e FCM-1011-0-BTSMEx <01.01 01:02:23.0700707>	129 DBG/	
5305 005202	27.10.16:57:34.750	[192.168.255.1]	6f FCM-1011-0-BTSMEx <01.01 01:02:23.070125>	129 DBG/	
5306 005203	27.10.16:57:34.765	[192.168.255.163]	9f FSP-1351 <01.01 01:02:40.400741>	22 INF/RAKEMASTE	
5307 005204	27.10.16:57:34.791	[192.168.255.35]	9e FSP-1400-0-TCOMexe <01.01 01:02:45.475401>	B8 INF/	
5308 005205	27.10.16:57:34.791	[192.168.255.35]	97 FSP-1400-0-TCOMexe <01.01 01:02:45.475457>	B8 INF/	
5309 005206	27.10.16:57:34.791	[192.168.255.35]	98 FSP-1400-0-TCOMexe <01.01 01:02:45.475795>	B8 INF/	
5310 005207	27.10.16:57:34.791	[192.168.255.35]	99 FSP-1400-0-TCOMexe <01.01 01:02:45.4760001>	B8 INF/	
5311 005208	27.10.16:57:34.791	[192.168.255.35]	9f FSP-1400-0-TCOMexe <01.01 01:02:45.4767626>	B8 INF/	
5312 005209	27.10.16:57:34.791	[192.168.255.35]	9b FSP-1400-0-TCOMexe <01.01 01:02:45.476671>	B8 INF/	
5313 005210	27.10.16:57:34.791	[192.168.255.171]	9f FSP-1431 <01.01 01:02:40.399657>	22 INF/RAKEMASTE	
5314 005211	27.10.16:57:34.828	[192.168.255.1]	ad FCM-1011-0-BTSMEx <01.01 01:02:23.0381352>	114 DBG/	
5315 005212	27.10.16:57:34.828	[192.168.255.1]	ae FCM-1011-0-BTSMEx <01.01 01:02:23.0381428>	114 DBG/	
5316 005213	27.10.16:57:34.828	[192.168.255.1]	af FCM-1011-0-BTSMEx <01.01 01:02:23.0381545>	167 INF/	
5317 005214	27.10.16:57:34.828	[192.168.255.1]	b0 FCM-1011-0-BTSMEx <01.01 01:02:23.0381603>	156 DBG/	

Takie rozwiązanie przyniosło wymierne korzyści. Najważniejszą z nich jest fakt, iż developerzy mogą testować oprogramowanie na środowisku bardzo zbliżonym do środowiska produkcyjnego. Mowa tutaj o komputerach typu PC z zainstalowanym Linuksem. Testowanie oprogramowania już na etapie rozwoju, na jak najbardziej zbliżonym środowisku, pozwala podnieść ostateczną jakość produktu. Testowanie takiego oprogramowania może odbywać się przy użyciu standardowego rodzaju testów (jednostkowych lub modułowych), lub poprzez uruchomienie pełnego skompilowanego pliku binarne-

go na symulatorze. Zarówno środowisko produkcyjne, jak i testowe działa pod kontrolą takiego samego systemu operacyjnego, możliwe jest wykrycie i naprawa wielu błędów, które nie pojawiłyby się podczas testów przeprowadzanych na systemie Windows.

### Środowisko LinSEE

Kluczowym elementem systemu Linux w naszej firmie jest LinSEE, czyli specjalnie przygotowane środowisko developeriske. Zawiera kilka wersji kompilatorów i różnego rodzaju narzędzi takie jak IDE, klienci systemu kontroli wersji, biblioteki, oraz wszystko inne co jest potrzebne do codziennej pracy developera.

Środowisko LinSEE, stworzone na potrzeby developerów Nokia Networks dostępne jest w wewnętrznym repozytorium firmy. Często dla jednego pakietu udostępnianych jest kilka dostępnych wersji. Dobieramy je w zależności od tego z jakim releasem soft-u pracujemy w danym momencie, lub na jaki target chcemy przeprowadzić komplikację.

Na komputerze można mieć zainstalowane po kilka wersji dla jednego pakietu, ponieważ developer jawnie decyduje, której z wersji chce w danym momencie używać. Odbywa się to mniej więcej tak: założymy że potrzebujemy interpretera Pythona w wersji 2.7.3:

```
$ source /opt/python/x86_64/2.7.3/interface/startup/linsee.env  
$ python --version  
Python 2.7.3
```

Oczywiście nie trzeba za każdym razem ręcznie wyszukiwać odpowiednich wersji pakietów. Razem z kodem źródłowym dystrybuowane są odpowiednie skrypty, które „zaciągną” potrzebne pakiety LinSEE dla konkretnej wersji źródeł. Wygląda to mniej więcej w ten sposób:

```
$ cd [KATALOG_ZE_ZRODLAMI]  
$ source CMake/set-env-linsee-bleeding-edge  
[info] BOOST_ROOT: /var/fpwork/boost_1_55_0_GLIBCXX_DEBUG  
[info] CMake/set-env.d/set-env.sh now set all required software  
versions needed by local compilation @LinSEE...  
[info] Test passed cmake -P CMake/set-env.d/test.cmake, no need  
to source anything  
[info] Test passed [[ $(python -V 2>&1) == Python\ 2.7.* ]], no  
need to source anything  
[info] Sourced /opt/protobuf/x86_64/2.5.0-2/interface/startup/  
linsee.env  
[info] Test passed [[ $(lcov --version) == *:\ LCOV\ version  
1.1?* ]], no need to source anything  
[info] Test passed [[ $(gcc --version|head -n1) == gcc\ \(\GCC\)\  
4.9.* ]], no need to source anything  
[info] Test passed [[ $(valgrind --version) == valgrind-3.8.*  
]], no need to source anything  
[info] Test passed [[ $(gdb --version | head -n1) == GNU\ gdb  
\(GDB\)\ 7.0 ]]  
(itd. ...)
```

Po takiej operacji mamy załadowane wszystkie narzędzia i biblioteki, które są potrzebne do skompilowania kodu. Jeśli zależy nam by dodatkowo zawsze mieć pod ręką narzędzia które nie są automatycznie ładowane przez skrypt build systemu, dodajemy je do pliku `~/.bashrc`.

To jest mój plik `~/.bashrc` „zaciągający” pakiety ze środowiska LinSEE:

```
source /opt/vim/x86_64/7.4/interface/startup/linsee.env  
source /opt/gdb/x86_64/7.5.1-1/interface/startup/linsee.env  
source /opt/cgdb/x86_64/0.6.7-1/interface/startup/linsee.env  
source /opt/subversion/x86_64/1.8.8/interface/startup/linsee.env  
source /opt/qt/x86_64/5.2.1-1/interface/startup/linsee.env  
source /opt/qt/x86_64/5.2.1-1/interface/startup/qt-manual.env  
source /opt/qtcreator/x86_64/3.0.1-1/interface/startup/linsee.env  
source /opt/doxygen/x86_64/1.8.5/interface/startup/linsee.env  
source /opt/valgrind/x86_64/3.8.1/interface/startup/linsee.env  
source /opt/lcov/noarch/1.10/interface/startup/linsee.env  
source /opt/distcc/x86_64/3.2/interface/startup/linsee.env  
  
alias maketags="ctags -R --c++-kinds=+p --fields=+iaS --extra=+q ."  
alias oamdiff="LANG='C' svn --force diff -x '--ignore-eol-style'"  
alias cmake_eclipse_cpp11='cmake -G"Eclipse CDT4 - Unix Makefiles" -DCMAKE_CXX_COMPILER_ARG1=-std=c++11'  
  
export SVN_EDITOR=vim
```

Korzystając z Linuksa można w łatwy sposób rozprzesyć komplikacje kodu na wiele maszyn, przy użyciu narzędzia distcc. Do komplikacji wykorzystywana jest wtedy sieć serwerów LinSEE. Aby tego dokonać trzeba „zaciągnąć” odpowiednie środowisko:

```
$ cd [KATALOG_ZE_ZRODLAMI]  
$ . CMake/set-env.d/linsee-distcc-3.2.sh
```

Następnie każde wywołanie polecenia „make” przekazać poprzez narzędzie „pump”, tak jak na poniższym przykładzie:

```
$ pump make all -j50
```

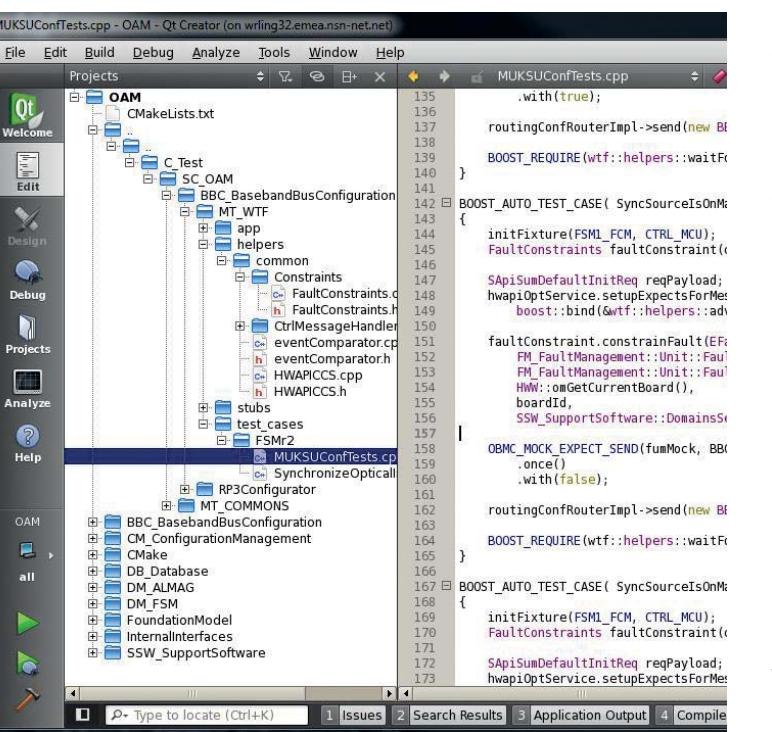
Kod źródłowy będzie komplikowany przy użyciu 50 procesów, na różnych serwerach z Linuksem i środowiskiem LinSEE.

### Serwery LinSEE

Jeśli ktoś zdecyduje się na instalację systemu Windows na swoim lokalnym komputerze lub jeśli po prostu potrzebuje więcej zasobów niż oferuje mu jego PC, może skorzystać z sieci potężnych serwerów LinSEE.

Dostęp do takiego serwera możliwy jest przez protokół SSH lub X11. Serwer oferuje pełną funkcjonalność środowiska LinSEE, ale przy tym znacznie większą moc obliczeniową niż lokalny komputer PC.

## 2 Ilustracja



Oczywiście lokalny Linux potrafi wszystko to co można zrobić na serwerze LinSEE, jednak czasem nawet dla użytkownika Linuksa, po prostu łatwiej jest zlecić coś do wykonywania na serwerze, a zasoby swojego PC-ta wykorzystać w tym czasie do innej pracy.

### Continuous Integration

Szczegółowe wyjaśnienie czym jest Continuous Integration, wykraczca poza zakres tego rozdziału. W tym miejscu wystarczy powiedzieć, iż realizacja CI polega na częstym integrowaniu i testowaniu kodu. Potrzebne są do tego maszyny, które przez 24h będą uruchamiać różnego rodzaju testy, by jak najszybciej dać informację o momencie w którym pojawi się jakiś defekt.

Te maszyny to dedykowane serwery LinSEE, działające pod kontrolą scentralizowanego systemu CI. Dlatego w przypadku wystąpienia jakiegoś błędu developer może go z łatwością odtworzyć na swoim PC lub serwerze LinSEE, w celu jego naprawienia.

CI jest bardzo ważną metodyką rozwoju oprogramowania. Poprzez zastosowanie do tego celu Linuksa możliwe jest łatwe skalowanie systemu. Przykładem może być możliwość bardzo elastycznego rozpraszania zadań pomiędzy maszynami, co przyspiesza działanie całego procesu.

### Debuggowanie

W systemie Linux standardowym narzędziem do debuggowania aplikacji jest GDB. Jest to bardzo potężne narzędzie, które wymaga pewnej wprawy w posługiwaniu się nim. Jest to aplikacja typowo konsolowa, jednak istnieją graficzne nakładki, najczęściej wykorzystywane to takie, które są zintegrowane ze środowiskami IDE.

Ciekawym sposobem wykorzystania GDB jest debuggowanie oprogramowania na środowisku „targetowym”, inaczej mówiąc, bezpośrednio na BTS-ie. Polega to na uruchomieniu instalacji gdb-server-a na BTS-ie oraz standardowego gdb na lokalnym PC. Po podłączeniu się do serwera, praca przypomina standardową sesję z gdb.

Czasem trzeba znaleźć miejsce wystąpienia błędu bez możliwości debuggowania działającego systemu. Jest to możliwe, ponieważ systemy Nokia Networks zapisują wystarczającą ilość informacji podczas wystąpienia błędu w pliku logów. Podam przykład skryptu, który może być użyty do łatwego odczytania callstack-a, z logów BTS-a:

```
$ cat [PLIK_LOGÓW] | grep -o '#[0-9]+\+ 0x[0-9a-f]\+ .*BTSMexe.*'  
| awk '{print "echo -n \"\$1\" :\t\"\$2\""}' | sh
```

W skrócie działania tej linijki polega na odpalaniu Linuksowej aplikacji addr2line z odpowiednimi parametrami. Otrzymamy wtedy mniej więcej taki wynik:

```
#5: LinuxMIPS_WN_FCT/COpponentModuleConnection.cpp:892  
#6: LinuxMIPS_WN_FCT/COpponentModuleConnection.cpp:667  
(...)
```

### Dostęp do natywnych narzędzi Windows na Linuksie

Jak wcześniej wspomniałem, dzięki zdalnym serwerom, można używać systemu Linux do pracy, mimo zainstalowanego Windowsa na lokalnym komputerze. Możliwe jest również wykorzystanie narzędzi typowo windowsowych takich jak np. Office czy Excel na systemie Linux. Istnieją na to dwa sposoby.

Pierwszym i najbardziej oczywistym z nich jest maszyna wirtualna. Chodzi tutaj o emulator procesora i peryferiów, który m.in. umożliwia uruchamianie systemu operacyjnego wewnętrzny innego systemu (naszego komputera służbowego). Za pomocą wewnętrznego kanału dystrybucji, przeprowadzamy sieciową instalację specjalnie przygotowanej do tego celu wersji systemu Windows. Posiada ona preinstalowane wszystkie standardowe aplikacje, m.in. MS Office oraz specyficzne dla firmy aplikacje wewnętrzne. W skrócie, dostępne jest wszystko to co na typowym stacjonarnym Windowsie.

Jeśli jednak nie chcemy ograniczać zasobów komputera poprzez uruchamianie wirtualnej maszyny, możemy skorzystać z drugiej

opcji, usługi Virtual PC. Wówczas nasz Windows działa na zdalnym serwerze, a szybki dostęp do niego uzyskujemy poprzez uruchomienie „cienkiego klienta” do niego. Dostęp do dysku udostępniany jest przez otoczenie sieciowe lub inaczej mówiąc przez protokół SMB. Sporą zaletą jest to że system jest cały czas uruchomiony, dlatego nie musimy czekać na „startup” przy logowaniu.

#### **Podsumowanie**

Nokia Networks wykorzystuje Linuksa w bardzo szerokim zakresie. Wynika to m.in. z faktu, iż sprzęt produkowany przez firmę również pracuje pod kontrolą tego systemu. Multi-platformowość Linuksa umożliwiła zbliżenie środowiska testowego i produkcyjnego, co pozwala utrzymać wysoką jakość produkcji wydawanego oprogramowania.

W firmie Nokia Networks w większości to developer decyduje jakich narzędzi developerskich i jakiego systemu operacyjnego będzie używał. Developerzy mają do dyspozycji sieć serwerów z Linuksem oraz maszyny CI (których odpowiedzialnością jest testowanie oprogramowania 24 godziny na dobę.)

---

#### **Autor o swojej pracy**

Pracuję w jednostce rozwijającej technologię 3G / WCDMA we Wrocławiu, w dziale konfiguracji pasma bazowego (Baseband Bus Configuration). Nasze oprogramowanie ma zapewniać, aby wszystkie dane trafiające do anten stacji bazowych były prawidłowo odkodowane i rozdysystrybuowane po systemie. Moja praca daje mi możliwość uczestniczenia w rozwoju technologii komórkowej i wielkiego systemu o znaczeniu globalnym napisanym w języku C++.

---

**Kamil Pawłowski**  
C++ Software Developer  
MBB SingleRAN O&M

# Statyczna / dynamiczna analiza kodu

Marcin Zawadzki  
R&D Manager  
MBB Single RAN OMCP

**NOKIA**

Narzędzia programistyczne są nieodłącznym towarzyszem pracy każdego programisty. Nie inaczej jest w przypadku programisty C++. W niniejszym artykule postaram się opisać kilka narzędzi ułatwiających i usprawniających programistę codzienną pracę. Dodatkowo będą to narzędzia, dzięki którym napisany kod będzie bardziej optymalny i pozbawiony wielu błędów. Ponieważ w Nokia oprogramowanie powstaje na platformy ze środowiskiem Linux, dlatego narzędzie przedstawione będą dla tego właśnie środowiska.

I tak kolejno przejdziemy przez narzędzia do statycznej i dynamicznej analizy kodu. Poprzez narzędzia do badania „pokrycia” kodu. Kończąc na narzędziach służących do badania wydajności kodu. Za- tem czas zacząć.

## 1. Statyczna analiza kodu

Jak nazwa wskazuje analiza ta odbywa się statycznie, czyli jest ona przeprowadzana bez uruchamiania programu. Oznacza to tyle, że do jej przeprowadzenia wystarczy komplilujący się kod. Analiza taka potrafi znaleźć wiele różnych problemów związanych z kodem (wy- cieki pamięci, przepelenienia buforów itd.). Wykrywa również błędy stylistyczne, które nie powodują nieprawidłowego działania kodu jednak sprawiają, że jest on mniej czytelny.

### 1.1

Pierwszym pomocnikiem programisty jest kompilator, który to może nas poinformować o wielu potencjalnych błędach w kodzie. Aby po- moc kompilatora mogła zadziałać potrzeba włączyć odpowiednie flagi komplilacji. I tak dla kompilatora gcc będą to flagi -Wall, -pedantic, -Wextra. Przy czym:

- flaga -Wall włącza wszystkie ostrzeżenia komplilacji
- flaga -pedantic będzie pilnowała zgodności naszego kodu ze standardem ANSI
- flaga -Wextra włącza dodatkowe ostrzeżenia nie znajdujące się pod flagą -Wall.

Dla kompilatora clang występuje jedna różnica, zamiast flagi -Wextra istnieje flaga -Weverything.

Co dla nas mogą zrobić takie flagi? Na przykład poinformować o tym, że używamy niezainicjalizowanych zmiennych, bądź o tym, że kolejność elementów na liście inicjalizacyjnej konstruktora klasy nie jest zgodna z kolejnością pól wewnątrz klasy, co z kolei może mieć opłakane skutki. Przykładowy problem podczas komplilacji, który mo- żemy zobaczyć po włączeniu flag -Wall oraz -Weverything obok:

```
code_analysis.cpp:22:13: warning: variable 'result' is
uninitialized when used here [-Wuninitialized]
    if (result > 0)
        ~~~~~
code_analysis.cpp:20:22: note: initialize the variable 'result'
to silence this warning
    size_t result;
                ^
= 0
code_analysis.cpp:72:14: warning: unused parameter 'argc'
[-Wunused-parameter]
int main(int argc, char *argv[])
^
code_analysis.cpp:72:26: warning: unused parameter 'argv'
[-Wunused-parameter]
int main(int argc, char *argv[])
^
3 warnings generated.
```

Przełącznikiem zasługującym na dodatkową uwagę jest -Weffc++. Kod skompilowany z przełącznikiem zostanie sprawdzony pod względem wytycznych z książek „Effective C++” oraz „More Effective C++” autorstwa Scotta Meyersa. Potencjalne błędy, o których zostaniemy ostrzeżeni to:

- brak zdefiniowanego konstruktora kopiącego i operatora przypisania dla klas z dynamiczną alokacją pamięci,
- przypisanie wartości polom klasy zamiast inicjalizacji na liście inicjalizacyjnej konstruktora,
- operator= nie zwracający wskaźnika na \*this,
- próbie zwrócenia referencji w przypadku, kiedy należy zwrócić obiekt,
- rozróżnienie pomiędzy prefiksową i postfiksową formą operatorów inkrementacji oraz dekrementacji,
- przeciążenie operatorów &&, || lub , (operator przecinka).

### 1.2

Kolejne narzędzie można wykorzystać podczas regularnej komplilacji kodu. Mowa tutaj o scan-build. Zasada jest prosta, podczas komplilacji kodu jest on jednocześnie sprawdzany przez statyczny analizator kodu. Narzędzia można używać z kompilatorami gcc oraz clang. W obecnej chwili nie znajduje on tuły błędów, co narzędzia opisane w dalszej części. Jednak ze względu na dynamiczny rozwój jest to program godny uwagi. Przykładowy wynik analizy na następnej stronie:



Na str. 53 znajduje się zrzut ekranu pokazujący przykładową analizę przy użyciu programu kcov.

Z powyższej analizy można wyczytać, iż plik został „pokryty” testami w 77.8% oraz np., że nigdy nie został uruchomiony destruktor klasy BaseClass.

To wszystko, jeśli chodzi o krótki przegląd narzędzi ułatwiających programiście codzienną pracę. Temat na pewno nie został wyczerpany. Jednakże powyższe przykłady są dobrą bazą narzędzi usprawniających pracę programisty. Warto zatem używać ich podczas codziennych zmagań z programowaniem w języku C/C++.

#### Bibliografia:

<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>  
<http://clang.llvm.org/docs/UsersManual.html>  
<http://clang-analyzer.llvm.org/scan-build.html>  
<http://www.klocwork.com/programs/ppc/static-analysis-code>  
<http://cppcheck.sourceforge.net/>  
<http://valgrind.org/>  
<http://kcachegrind.sourceforge.net/html/Home.html>  
<https://sourceware.org/binutils/docs-2.24/gprof/>  
<http://ltp.sourceforge.net/coverage/lcov.php>  
<https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>  
<http://simonagstrom.github.io/kcov/>  
<http://bcov.sourceforge.net/>

#### Autor o swojej pracy

Jestem absolwentem Elektroniki i Telekomunikacji (EIT) na Politechnice Wrocławskiej. Obecnie zajmuję stanowisko kierownika w Nokia Networks. Moją pasją jest rozwój narzędzi wspomagających pracę programistów. W tym roku w moim dziale będziemy rozwijali technologię 5G. Pracując u nas będziesz miał okazję nauczyć się najwyższych standardów rozwoju oprogramowania.

**Marcin Zawadzki**  
R&D Manager  
MBB Single RAN OMCP

# Wyrażenia regularne

Marcin Załuski  
Engineer, Software Development  
MBB FDD LTE Control Plane

**NOKIA**

Wyrażenia regularne są jak szwajcarski scyzoryk. Ich wsparcie można znaleźć w licznych aplikacjach biurowych, narzędziach uniksowych i językach programowania. W wielu momentach okazują się nieocenioną pomocą. Przydają się, gdy trzeba zaimplementować zaawansowane wyszukiwanie w bazie danych, gdy należy dokonać niebanalnej podmiany tysięcy linii, czy sprawdzić zgodność z oczekiwany formatem odpowiedzi w formularzu. Oferują olbrzymie możliwości i są potężnym narzędziem, choć czasem potrafią sprawić niemało problemów. Warto przytoczyć tutaj słowa twórcy projektu Mozilla, Jamiego Zawiskiego: „Some people, when confronted with a problem, think ‘I know, I’ll use regular expressions.’ Now they have two problems.”

## Historia

Wyrażenia regularne, nazywane często regex od angielskiego regular expressions, są związane z teorią języków regularnych i teorią automatów (vide Maszyna Turinga). Ojcem wyrażeń regularnych jest amerykański matematyk Stephen Cole Kleene, który w 1956 r. opisał języki regularne przy użyciu zbiorów regularnych.

Jednym z pierwszych narzędzi wykorzystujących regex był edytor QED, a konkretniej nowa wersja przeznaczona na Compatible Time-Sharing System napisana w 1968 r. przez Kena Thompsona po jego zatrudnieniu w Bell Labs. Zastosowane przez Thompsona nowatorskie rozwiązanie, jakim była wykorzystana metoda komplikacji w locie (JIT) z zastosowaniem niedeterministycznego automatu skończonego, doczekało się patentu. W dalszym czasie wyrażenia regularne zostały zaadaptowane do uniksowego edytora Ed. W efekcie zrodziło się jedno z najpopularniejszych narzędzi uniksowych do wyszukiwania ciągów w tekście, jakim jest grep. Jego nazwa wywodzi się od polecenia edytora Ed, które odpowiada za wyszukiwanie w tekście ciągów pasujących do zadanego wyrażenia regularnego: g/re/p (globally search a regular expression and print).

Dziś wyrażenia regularne są potężnym i szeroko wykorzystywany narzędziem. Obecnie najpopularniejszymi składniami są składnia POSIX i składnia perlowa, która zawiera szereg przydatnych rozszerzeń. Wsparcie dla regeksów można znaleźć w różnych IDE, procesorach i edytورach tekstu oraz językach programowania. Część z języków wspiera wyrażenia regularne natywnie (Perl, Ruby, Tcl, JavaScript), część poprzez bibliotekę standardową (Python, Java, C++11) oraz za pomocą zewnętrznych bibliotek, takich jak PCRE dla języka C.

## Podstawy

Przede wszystkim warto rozróżnić wyrażenia regularne od wieloznaczników (wildcard). Sporo narzędzi wspiera obie metody dopasowań w procedurze wyszukiwania. Dla przykładu, jeśli chcemy wyświetlić w powłoce Bash wszystkie pliki w bieżącym katalogu posiadające rozszerzenie zip, możemy użyć polecenia: „ls \*.zip”. Oznacza to tyle, co wypisz wszystkie pliki, których nazwa składa się z dowolnego ciągu i końcówek „.zip”. Znak „\*” odgrywa tu inną rolę, niż w przypadku regeksów, gdzie oznacza powtórzenie zero lub więcej razy

symbolu poprzedzającego. Różnicę tę ukazuje przykład z **Tabeli 1**, dopasowany ciąg został podkreślony. Jak widać, wyrażenia regularne są w praktyce zapisywane w postaci fragmentu szukanego tekstu z dodatkowymi operatorami. Warto zaznaczyć, że powszechnie stosowany zapis różni się w stosunku do notacji wykorzystywanej w informatyce teoretycznej – zawiera więcej operatorów, co pozwala skrócić wyrażenia. **Tabela 2** przedstawia przykłady zastosowania wyrażeń regularnych wraz z objaśnieniami.

**1 Tabela** Ilustracja różnicy między wieloznacznikami, a wyrażeniami regularnymi

Wildcards:	A*	Ala ma kota
Regex:	A*	Ala ma kota

**2 Tabela** Składnia regex

Wyrażenie	Wynik	Opis
A*B	CAAAAABBBB BBBBB	Dowolna ilość znaków A i znak B, oba ciągi pasują do wyrażenia.
A+B	CAAAAABBBB BBBBB	Co najmniej jeden znak A i jeden znak B, drugi ciąg nie pasuje do wyrażenia regularnego, bo nie zawiera co najmniej jednego znaku A.
AB+A	ABBA	Ciąg całkowicie pasuje do wyrażenia.
ABC?	ABBA ABCDE	Znak przed operatorem ? może, ale nie musi wystąpić.
A.A	ALA AGA ADA ANNA	Kropka pozwala na dopasowanie dowolnego znaku.
^A	AAAA _	Karetka dopasowuje początek ciągu, bądź linii.
A\$	AAAA _	Dolar dopasowuje koniec ciągu, bądź linii.
A B	AAABBBCAB	A lub B.
\.\+\*\*	** ABC ...	Wsteczny ukośnik wyłącza specjalne znaczenie następującego po nim symbolu.
[AB]	ABBA	Kwadratowe nawiasy zawierają zestaw dozwolonych znaków.
[^AB]	ABCD	Daszek w nawiasach kwadratowych oznacza negację zbioru – pasuje każdy znak oprócz wymienionych.

Ułatwieniem przy pisaniu regeksów są klasy i przedziały znaków, jak **[0-9]** dla oznaczenia dowolnej cyfry. Klasa cyfr może być zapisana także jako **[:digit:]** lub w perlowym dialekcie jako **\d**. Istnieją klasy dla znaków alfabetu, znaków białych, przestankowych itd.

Pewnym problemem przy korzystaniu z wyrażeń regularnych może być funkcjonowanie kilku różnych standardów, które różnią się między sobą zestawem elementów. Dobrze jest sprawdzić, jakie dialekty są obsługiwane przez bibliotekę, czy aplikację, z której się korzysta. Dla przykładu program grep standardowo stosuje podstawowe wyrażenia regularne w standardzie POSIX. Przy użyciu dodatkowych parametrów, można włączyć rozszerzone wyrażenia regularne (-E), bądź perlowe (-P).

## Grupy i referencje wsteczne

Często przydatne okazuje się grupowanie i odwoływanie się do fragmentów ciągu. Dzięki temu można dokonywać zaawansowanej podmiany w sposób generyczny. Za grupę uznaje się znaki zawarte między parą nawiasów okrągłych. Dzięki tej funkcjonalności można na przykład wyszukać zdania, w których jedno słowo może się zmieniać w określonym zakresie. Wyrażenie regularne 'Ala ma (kota|psa|rybki)' dopasuje trzy zdania: 'Ala ma kota', 'Ala ma psa' i 'Ala ma rybki'. Praktycznym przykładem jest szukanie plików z rozszerzeniem jpg, png i gif przy użyciu polecenia find:

```
find . -regextype posix-extended -iregex '.*\.(jpg|png|gif)'
```

Na uwagę zasługuje parametr **regextype**, włączający rozszerzone wyrażenia regularne, dzięki czemu nie trzeba poprzedzać ukośnikiem wstecznym nawiasów i operatora alternatywy.

Grupy tworzą referencje wsteczne, które następnie pozwalają odwoływać się do ich zawartości poprzez **\n**, gdzie **n** jest numerem referencji nadawanym kolejno począwszy od 1. Można sobie wyobrazić sytuację, w której plik CSV o formacie imię;nazwisko;miejscowość należy zmienić na format miejscowość;nazwisko;imię. Łatwo to zrobić z wykorzystaniem programu sed:

```
sed -r 's/^([;]+);([;]+);([;]+)/\3;\2;\1/g' dane.csv
```

Polecenie korzystając z rozszerzonych wyrażeń regularnych wyszukuje trzy rozdzielone średnikiem grupy podciągów niezawierających średnika, a następnie z wykorzystaniem referencji wstecznych odwraca ich kolejność.

Czasami, choćby ze względów optymalizacyjnych, warto zastosować grupowanie, które nie tworzy referencji. Dla przykładu wyrażenie 'Ala (?nie )?ma kota' dopasuje wersję z zaprzeczeniem, jak i bez zaprzeczenia, nie tworząc przy tym referencji wstecznej.

## Zachłanność i lenistwo

Operatory wyrażeń regularnych są domyślnie zachłanne (ang. greedy), co oznacza, że dopasowują tyle znaków, ile są w stanie. Dla przykładu wyrażenie A\* dla ciągu 'AAAAAAA' dopasuje cały ciąg. Często jest to bardzo pożąданie zachowanie. Czasami jednak zdarza się, że potrzeba dopasować jak najmniej znaków. Wtedy właśnie warto skorzystać z trybu leniwego (ang. lazy), zwanego także niezachłannym (ang. non-greedy). Różnica została przedstawiona na poniższym przykładzie:

```
$ echo 'Król Karol kupił królowej Karolinie korale...' | grep -P '^.*Karol'  
Król Karol kupił królowej Karolinie korale...  
$ echo 'Król Karol kupił królowej Karolinie korale...' | grep -P '^.*?Karol'  
Król Karol kupił królowej Karolinie korale...
```

Jak widać, aby operator stał się leniwy, wystarczy dodać za nim znak zapytania. Parametr '-P' jest wymagany, ponieważ grep obsługuje operatory niezachłanne tylko w składni perlowej. Operatory leniwe są dość kosztowne pod względem wydajności i przy optymalizowaniu zapytania wykonywanego setki razy na sekundę warto pomyśleć o zastosowaniu w ich miejsce negacji zakresu (**[^abcd]**).

## Liczby pierwsze

Swego rodzaju ciekawostką jest fakt, że przy użyciu wyrażeń regularnych można określić, czy dana liczba jest liczbą pierwszą:

```
$ perl -lne '(1x$_) =~ /^1?|$^(11+?)\1+$/ || print "$_ jest liczbą pierwską"  
1  
2  
2 jest liczbą pierwszą  
3  
3 jest liczbą pierwszą  
4
```

Przede wszystkim liczba musi być zapisana w postaci unarnej, w której do zapisu wartości liczbowej używany jest tylko jeden symbol, w tym przypadku 1. Powyższy skrypt czyta standardowe wejście i konwertuje liczbę na jej reprezentację unarną (1 – 1, 2 – 11, 3 – 111 itd.). Następnie zachodzi próba dopasowania, ponieważ wzorzec sprawdza, czy ciąg zawiera w sobie wielokrotność podciągu, zatem tak naprawdę jest to test na to, czy liczba jest złożona. Zatem dana liczba jest liczbą pierwszą, gdy ciąg nie zostanie dopasowany do wyrażenia regularnego, stąd operator OR (||). Sam regex składa się z dwóch części, również oddzielonych operatorem alternatywy. W pierwszej sprawdzane jest dopasowanie do ciągu pustego lub do 1. W drugiej wykorzystany jest operator niezachłanny oraz referencja wsteczna. Fragment ujęty w nawiasach jest dopasowany niezachłannie, czyli najpierw sprawdzany jest dla ciągu 11. Referencja wsteczna oznaczająca 11 jest powtarzana raz lub więcej razy i jeśli jest dopa-

sowany cały ciąg, oznacza to liczbę złożoną. W przeciwnym razie, dzięki operatorowi niezachłannemu sprawdzane jest dopasowanie dla 111 itd. aż do wyczerpania możliwości, bądź znalezienia odpowiedniej wielokrotności.

## Podsumowanie

Wyrażenia regularne są nieocenioną pomocą w wielu zadaniach związanych z przetwarzaniem danych. Pozwalają szybko uporać się ze złożonymi problemami. Dzięki nim można w prosty sposób skonstruować zaawansowany filtr danych, bądź dokonać niebanalnej podmiany setek linii w pliku. Należy jednak pamiętać, że z wielką mocą wiąże się wielka odpowiedzialność i trzeba mieć się na bacz-

ności, gdyż wyrażenia regularne czasami potrafią zastawić na nieświadomego użytkownika kilka pułapek. Dobrym pomysłem jest utworzenie kopii zapasowej danych, przed skorzystaniem z nieprzetestowanego wyrażenia regularnego. W takich sytuacjach warto sprawdzić, czy używa się dialekta właściwego dla danego narzędzia oraz podzielić złożone wyrażenie na mniejsze w celu osobnego przetestowania. Dodatkowo, czasami wybór wyrażeń regularnych jako rozwiązań dla danego problemu może być błędem. Przykładowo przetwarzanie języków bezkontekstowych typu 2. hierarchii Chomsky'ego (XML, HTML itp.) za pomocą regeksów jest niezalecane, ponieważ najlepiej spisują się one w językach regularnych (typ 3).

## Autor o swojej pracy

Pracuję w dziale C-Plane LTE. Zajmujemy się tam rozwijaniem oprogramowania na stacje bazowe. C-Plane skupia się na wszelkich danych kontrolnych, które służą do utrzymywania połączenia z urządzeniem abonenckim. Kodujemy w języku C++11.

### Marcin Załuski

Engineer, Software Development  
MBB FDD LTE Control Plane

# Jak przeszliśmy od Javy do JavaScript?

Tomasz Drwięga  
Specialist, Software Development  
Big Data & Network Engineering

**NOKIA**

Nokia to nie tylko BTSy i C++. Tworzymy również oprogramowanie, które usprawnia naszą wewnętrzną pracę i wspomaga nas w automatyzacji i wykonywaniu codziennych zadań.

W dziale Big Data & Network Engineering budujemy narzędzia, które pomagają przetwarzać i analizować duże ilości danych telekomunikacyjnych.

## Etap 1: Offline

Kiedy dołączyłem do naszego działu zajmowaliśmy się kilkoma narzędziami, które tworzone były jako aplikacje desktopowe. Pracowaliśmy nad projektem pisany w Javie (rozwijanym do dzisiaj), który używany jest podczas analizy pracy sieci oraz planowania nowych stacji bazowych.

Aplikacja desktopowa jest oczywiście instalowana na komputerze użytkownika i dzięki temu można ją uruchomić, nawet kiedy nie ma on dostępu do Internetu. Problem pojawia się, kiedy aplikacja jest intensywnie rozwijana i chcemy, żeby wszyscy użytkownicy zawsze korzystali z najnowszej wersji. Dzień, w którym wychodziła nowa wersja naszej aplikacji i okres, który poprzedzał release był najbardziej intensywnym i stresującym okresem – w momencie, kiedy przygotowaliśmy instalator i wysłaliśmy wiadomość do użytkowników o nowej wersji wszystko musiało być dopięte na ostatni guzik. Kiedy użytkownicy pobiorą instalator nie będzie już odwrotu i możliwości wysłania im poprawek.

Kolejną bolączką było tworzenie GUI. Stworzenie spójnego interfejsu wymagało od nas napisania wielu abstrakcyjnych komponentów, co było bardzo czasochłonne. Interfejs aplikacji desktopowej jest też ograniczony kontrolkami, które dostępne są w danym systemie operacyjnym. Nasze GUI było po prostu brzydkie i nie było mowy o wprowadzeniu żadnych animacji czy wizualnych efektów. Z tych powodów zaczęliśmy szukać alternatywnego podejścia do tworzenia naszych narzędzi.

## Etap 2: Going Online

Pierwszym podejściem do rozwiązania tych problemów była próba stworzenia aplikacji webowej, a właściwie strony internetowej. Idea jest prosta – zamiast uruchamiać aplikację na komputerze użytkownika, uruchamiamy ją na naszym serwerze, a użytkownikowi dajemy tylko adres, pod którym może ją znaleźć. Dzięki takiemu podejściu to my kontrolujemy wersję aplikacji, z której korzystają wszyscy użytkownicy, ale niestety dostęp do Internetu (czy przynajmniej tej samej sieci, w której znajduje się serwer) staje się niezbędny.

Na tym etapie nie wiedzieliśmy prawie nic o pisaniu aplikacji internetowych. Kilku z nas miało pewne doświadczenie w tworzeniu stron z wykorzystaniem HTML i jQuery, ale na tym nasza wiedza się kończyła.

Pierwszy prototyp stworzyliśmy korzystając z PHP w oparciu o framework Symfony. Najważniejszą zaletą aplikacji było to, że nie mu-

sieliśmy już prosić użytkowników o instalację programu (lub aktualnienia) i generowanie licencji – wystarczyło wysłać im link i od razu mogli z niej korzystać!

Korzystanie z technologii webowych pozwoliło nam także na szybkie tworzenie funkcjonalnego interfejsu. Stworzenie GUI było też o wiele prostsze niż w Javie i mogliśmy wykorzystać otwarte biblioteki oraz zasoby, których w Internecie jest dużo więcej dla technologii webowych niż dla Javy.

Sposoby tworzenia interfejsu również zmieniały się wraz z zagłębianiem się w to, co możliwe jest do zrobienia z użyciem HTML. Rozpoczynaliśmy od stylowania komponentów za pomocą CSS i problemów z nazewnictwem klas, specyficznością i kompatybilnością między przeglądarkami. Bardzo szybko zaczęliśmy korzystać z preprocesorów CSS (np. LESS), a ostatecznie użyliśmy framework Twitter Bootstrap, który zaadaptowaliśmy do naszych potrzeb. Dzięki takiemu podejściu mogliśmy łatwo tworzyć spójny interfejs i bardzo szybko prototypować różne układy strony. Większość z nas jest typowymi programistami, a nie designerami, więc pozwala nam to skupić się na tworzeniu logiki naszej aplikacji, która przy okazji wygląda całkiem nieźle.

Aplikacja w postaci strony internetowej ma oczywiście swoje wady. Każda akcja użytkownika na naszej stronie była wysyłana do serwera, który generował nowy kod do wyświetlenia użytkownikowi. Powodowało to spore opóźnienia, ponieważ za każdym razem nasza strona musiała się odświeżyć. Użytkownicy oczekiwali od naszego narzędzia dużo więcej – trzeba było coś z tym zrobić.

## Etap 3: JavaScript

Jedną z niewielu dostępnych opcji wprowadzenia dynamicznej zawartości na stronę jest użycie JavaScriptu. Kojarzył się nam wtedy z prostym językiem skryptowym, który nie nadaje się do tworzenia bardziej zaawansowanych aplikacji, a pisanie w nim jest największą karą, która może spotkać "prawdziwych programistów".

Początkowo traktowaliśmy wstawki JS jako dodatek do statycznie wygenerowanej strony, które wprowadzały dynamiczne zachowanie w niektórych miejscach aplikacji: proste obliczenia, walidację itd. W miarę zwiększenia się ilości kodu pisanego w JS, zaczęliśmy szukać informacji o tym jak pisać kod dobrze i jak o niego dbać. Coraz bardziej przekonywaliśmy się do JavaScriptu jako "prawdziwego" języka programowania.

Z obowiązkowych narzędzi każdego programisty JS należy tutaj wymienić JSLint (lub JSHint), który wykonuje statyczną analizę kodu, znajdując najczęściej popełniane błędy i pozwala także kontrolować, czy zachowane są reguły, według których formatujemy nasz kod. Warto skorzystać też z biblioteki lodash (lub underscore), zawierającej wiele pomocniczych funkcji.

Dynamiczne elementy strony wymagane były coraz częściej, a nasza wiedza na temat JS cały czas rosła. Wtedy padła szalona propozycja: "Zróbmy SPA!".

#### Etap 4: SPA

Idea Single Page Application (SPA) polega na tym, że użytkownik ładowa-je początkową stronę, która zawiera całą aplikację. Kolejne odwołania do serwera robione są nie po to, aby wygenerować kolejny widok, ale w celu pobrania od serwera wyłącznie danych. Cała logika ładowana jest w przeglądarce użytkownika i pisana w języku JavaScript!

Pisanie SPA wymaga zupełnie innego podejścia niż pisanie zwykłej strony internetowej. Rozdzielimy naszą aplikację na dwie części: pierwsza działa na naszym serwerze i wysyła dane (back-end), natomiast druga to widok i logika aplikacji (front-end). Back-end odpowiedzialny jest za komunikację z bazą danych, autoryzację i ekspozycję danych, najczęściej w formie REST-owego API. Jako format wymiany danych używany jest JSON, który jest łatwo parsowalny po stronie klienta za pomocą JS. Front-end zawiera wszystko to, co widać użytkownik i jest odpowiedzialny za obsługę interakcji z nim.

Przygotowania rozpoczęliśmy od wyboru frameworku JavaScriptowego, na którym miała być oparta aplikacja. Nasz pierwszy wybór padł na Backbone.js, którego krzywa wejścia jest dość łagodna (łatwo rozpocząć w nim pisać, jeżeli zna się trochę JS), a dzięki podziałowi na Model, Widok i Kontroler wymusza dobrą strukturę aplikacji. Dodatkowo, w celu rozwiązania problemu paczkowania wielu plików JS po stronie klienta, skorzystaliśmy z RequireJS, który dba o ładowanie wymaganych modułów. Przez wprowadzenie podziału na front-end i back-end nasz kod JavaScriptowy stał się de facto niezależną aplikacją, która podlega takim samym procesom projektowym jak np. kod w Javie.

Tworzenie wielu plików JS wymagało od nas także wprowadzenia dodatkowego kroku poprzedzającego deploy nowego kodu na produkcję. Uruchomienie naszej aplikacji wymaga od przeglądarki użytkownika pobrania wszystkich plików JS z serwera. Z tego powodu ładowanie strony trwałoby bardzo długo (przeglądarki mają limit równolegle ściąganych plików z danej domeny). Przed uruchomieniem aplikacji na serwerze produkcyjnym łączymy wszystkie pliki JS w jeden, który następnie minimalizujemy, korzystając z narzędzi takich, jak UglifyJS czy Closure Compiler.

W celu automatyzacji procesu budowania aplikacji używamy Grunta – narzędzia, które rozszerzone o odpowiednie pluginy, pozwala na uruchamianie wszystkich zadań związanych z przygotowaniem aplikacji do przesłania na serwer produkcyjny. Grunt generuje raport z JSLintem i uruchamia testy, natomiast UglifyJS tworzy pojedynczy plik zawierający kod całej aplikacji.

Dodatkowo, tak jak podczas pisania aplikacji desktopowych, nasz kod po trafieniu do repozytorium pobierany jest przez Jenkinsa – serwer

Continuous Integration. Jenkins uruchamia Grunta i przygotowuje najnowszą wersję aplikacji, która jest zawsze gotowa do testowania.

Proces tworzenia aplikacji webowych jest tak samo wymagający, jak proces tworzenia programów innego typu (desktop, mobile, serwer). W przypadku SPA dodatkowy problem może stanowić sam język JS. Pomimo tego, że łatwo zacząć w nim pisać, to jego opanowanie jest nie lada wyzwaniem.

#### Etap 5: Learning

W naszym przypadku, zarówno jako metodę uczenia nowych osób dołączających do projektu, jak i doskonalenia oraz rozwijania się pozostałych członków zespołu, świetnie sprawdziło się Code Review. Każda zmiana w kodzie była analizowana przez co najmniej 3 dodatkowe osoby, których zadaniem nie było wytykanie błędów czy niewiedzy, ale wspólna nauka i dbanie o dobrą jakość kodu całego projektu.

Podczas Code Review wymieniamy się i dyskutujemy nad sposobami rozwiązyania danego problemu i poznajemy nowe punkty widzenia. Obecnie nie wyobrażam sobie pracy w projekcie, w którym Code Review nie jest stosowane.

Kolejnym ważnym elementem pisania w JS (a właściwie we wszystkich językach dynamicznych) jest testowanie. Ponieważ kod jest interpretowany, najbardziej trywialny błąd może wyjść na jaw dopiero w momencie, gdy faktycznie ten fragment uruchomimy! Dzięki testom jednostkowym możemy zweryfikować, czy nasze poszczególne funkcje robią dokładnie to czego chcemy. Duża liczba testów daje nam też pewność, że dodając nowy kod lub refaktorując istniejący, niczego nie zepsuliśmy. Praktyka pisania testów oraz metodyka TDD pozwalały także poprawić jakość i architekturę kodu.

W przypadku JS ciągłe doskonalenie się jest niezwykle ważne. Całe środowisko bardzo dynamicznie się zmienia, co chwila powstają nowe biblioteki i rozwiązania, a przeglądarki zyskują kolejne możliwości w postaci API dostępnych z poziomu JSa.

Do naszego procesu oprócz Continuous Integration, Continuous Deployment i Continuous Improvement dołączliśmy także Continuous Learning.

#### ? + NoSQL + Node.js + AngularJS

Obecnie wszystkie projekty, które rozpoczynamy, tworzone są z front-endem pisany w AngularJS, który pozwolił nam znacznie przyspieszyć i uprzyjemnić pracę nad aplikacjami SPA. Bardzo dobra znajomość JavaScriptu pozwala nam pisać programy, które nie tylko działają w przeglądarce, ponieważ JS jest językiem obecnie dostępnym już wszędzie. JS używamy również po stronie serwera, korzystając z node.js. W ramach Continuous Learning organizujemy jednodniowe Hackathony, podczas których mieliśmy okazję spróbować pisania aplikacji mobilnych w JS (PhoneGap, Ionic) lub innych

technologii (Go). Pierwszy serwer w node.js również powstał podczas jednego z Hackathonów – wszyscy w ekspresowym tempie mieli okazję poznać dobre i złe strony tej technologii.

Przejście od pisania desktopowych aplikacji do tworzenia SPA w AngularJS nie było dla nas proste. Podczas tej drogi wiele razy musieliśmy wyjść poza naszą strefę komfortu i pokonać wiele przeszkód. Ale czy właśnie nie na tym polega bycie programistą? Zdecydowanie było warto!

#### Autor o swojej pracy

Pracuję w dziale Big Data & Network Engineering jako Software Developer. Zajmujem się naprawianiem świata (zaczynamy od tego małego – wewnętrznego) i tworzymy przyszłość. Technologię dopasowujemy do problemu, a nie odwrotnie. Hackujemy w JavaScript, Node.js, PHP, Javie, Scali i Pythonie, korzystając z MongoDB, PostgreSQL, ElasticSearcha i Cassandra.

#### Tomasz Drwięga

Specialist, Software Development  
Big Data & Network Engineering

# Praktyka inżynierii oprogramowania

3.1

---

**Krzysztof Bulwiński**  
Dobre praktyki inżynierii oprogramowania  
66

3.2

---

**Wojciech Pisarski**  
Testowanie jednostkowe oprogramowania  
72

3.3

---

**Krzysztof Matuszek**  
„Clean Design”  
78

3.4

---

**Michał Rudowicz**  
Utrzymanie starego kodu dla zabawy i zysku  
88

3.5

---

**Maciej Jaskot**  
Programowanie a bezpieczeństwo  
94

3.6

---

**Andrzej Lipiński**  
Software Configuration Management – czym jest i do czego służy?  
100

3.7

---

**Marcin Gudajczyk**  
Continuous Integration – integracja oprogramowania po „tuningu”  
106

# Dobre praktyki inżynierii oprogramowania

Krzysztof Bulwiński  
Engineer, Software  
MBB CEMOSS OSS RD Pol RCM4

NOKIA

Mówiąc o dobrych praktykach inżynierii oprogramowania trudno byłoby nie zacytować Kathy Sierry i Berta Batesa, którzy w jednej ze swoich publikacji napisali: „Pisz kod wiedząc, że gość, który przejmie go po tobie, jest maniakalnym mordercą, który wie gdzie mieszkasz.” Stwierdzenie to oczywiście jest lekko przesadzone. Niemniej jednak tworząc oprogramowanie powinno się mieć na uwadze fakt, że przedzej czy później twój program lub jego fragment, w miarę rozwoju aplikacji, będzie przez autora lub kogoś innego, zmieniony w mniejszym lub większym stopniu. Dlatego od Ciebie zależy, czy osoba, która podejmie się tego zadania będzie je miała ułatwione, czy też nie. To od ciebie zależy, czy ta osoba będzie przeklinała każdą linijkę twojego kodu, albo miała pozytywne wrażenie myśląc przy tym: „Ten deweloper miał sporą świadomość tego co robił, pisząc ten kod”. Ponadto, programując należy zdawać sobie sprawę, że późniejsze jego utrzymanie np. naprawa błędów może kosztować firmę niemałe pieniądze. Tym większe będą to wydatki, im więcej deweloper spędza czasu na zrozumieniu intencji poprzednika. A przecież są to te wydatki, które firma chce możliwie najefektywniej ograniczyć.

W tym artykule będziemy starali się zawrzeć kilka praktycznych porad, które pozwolą tworzyć oprogramowanie, będące nie tylko funkcjonalnym, ale i łatwiejszym w późniejszym jego rozwoju oraz utrzymaniu.

Warto wspomnieć, że są tu wymienione tylko niektóre aspekty bardzo obszernej tematyki praktyk inżynierii oprogramowania. Zgłębienie ich wymaga ciągłego doskonalenia i rozwoju swojej wiedzy oraz umiejętności. Z biegiem czasu, subiektywnie zaczniemy dostrzegać wady i zalety każdego z rozwiązań. Jednak zainwestowany wysiłek zwróci się nam w postaci wypracowanego warsztatu deweloperskiego, który pozwoli nam tworzyć kod wysokiej jakości.

## Prototypowanie

Tworząc aplikację nie powinniśmy się kierować intencją, że to, co obecnie piszemy jest tylko chwilowym prototypem. Bardzo często dochodzi do sytuacji, że kod, który w zasadzie zapomnieliśmy lub porzuciliśmy zostanie reaktywowany. Właśnie nasz zapomniany projekt, który miał być tylko założkiem czegoś większego, zostanie zauważony przez osoby, które może będą chciały zrobić z niego osobny produkt. Najprawdopodobniej zostaną przydzielone zasoby ludzkie. Możliwe, że nie będzie to autor, gdyż będzie uwikłany w inne zobowiązujące zadania. Bardzo często projekt będzie musiał być kontynuowany pod presją czasu i na ostatnią chwilę. Zatem nie będzie czasu na to, żeby zastanowić się czy warto zmienić architekturę systemu lub zainwestować czas w napisanie testów charakterystycznych. Ten projekt będzie po prostu kontynuowany z takim samym podejściem i nastawieniem, jak jego początkowe założenie, czyli byle szybko i byle jak. W efekcie zaczynamy tracić kontrolę nad ilością kodu i ilością powiązań między modułami, sukcesywnie zapędzając się w ślepą uliczkę. Może to doprowadzić do sytuacji, w której nasza aplikacja praktycznie staje się niemożliwa do rozszerzenia, natomiast każda zmiana może powodować ryzyko powstawania nowych

błędów. Nie mówiąc już, że utrzymanie takiego oprogramowania staje się prawdziwym problemem.

Oczywiście nie powinniśmy wzbraniać się przed prototypowaniem, ale musimy mieć na względzie to, że nasz kod może zostać użyty przez kogoś innego. Żeby nie wstydzić się tego, co ktoś po nas będzie kontynuował, musimy starać się, aby kod pisany był z użyciem dobrych praktyk i zasad projektowania oraz tworzenia oprogramowania.

## Programowanie sterowane testami czyli Test Driven Development

Kilka słów tematem wprowadzenia do metodyki Test Driven Development (TDD). Według Wikipedii TDD jest procesem wytwarzania oprogramowania, polegającym na powtarzalności bardzo krótkiego cyklu składającego się z trzech faz.

W fazie pierwszej deweloper pisze tylko test jednostkowy, którego egzekucja początkowo kończy się niepowodzeniem. Test powinien definiować nam ulepszenie istniejącej lub nową funkcjonalność. Innymi słowy jest to test, który nie pokrywa jeszcze implementacji kodu produkcyjnego. Podczas jego pisania, od programisty wymagane jest rozumienie specyfikacji oraz wymagań implementowanej funkcjonalności. Może zostać to osiągnięte poprzez rozpisanie przypadków użycia (use case) oraz historyjek użytkownika (user story), które powinny pokryć dane wymagania oraz wyjątkowe przypadki. Po napisaniu testu należy uruchomić istniejący zestaw testów, aby przekonać się, czy na pewno wykonanie nowego testu kończy się niepowodzeniem.

W drugiej fazie, deweloper zobligowany jest do napisania takiej ilości kodu, która zaspokoi test. To znaczy spowoduje, że jego egzekucja zakończy się powodzeniem. Nie musi to być czysty kod oraz może wydawać się, że został napisany w sposób przeczący ogólnie przyjętym zasadom i standardom. Wystarczy, że w możliwej najprostszym sposobie spowoduje zielony test, o czym należy się przekonać uruchamiając test lub ich zestaw. Takie podejście jest akceptowalne, gdyż zostanie to naprawione w następnej fazie.

W ostatniej, trzeciej fazie deweloper dokonuje refaktoryzacji kodu, stosując standardy oraz praktyki danego języka programowania. Innymi słowy porządkuje swój kod. Do tych czynności między innymi należą: przenoszenie bloków kodu do miejsc, do których pasują logicznie, usuwanie duplikacji kodu, nazewnictwo zmiennych, metod, itp. powinny odzwierciedlać swoje przeznaczenie, rozbicie długich metod na mniejsze, wprowadzanie wzorców projektowych oraz uporządkowanie architektury. Generalizując, wszystko to, co spowoduje, że nasz kod będzie po prostu czysty. Oczywiście podczas tych operacji, na bieżąco możemy weryfikować zmiany poprzez uruchamianie zestawu testów. Należy pamiętać, że refaktoryzacja odnosi się jednakowo do kodu produkcyjnego, jak również do testów jednostkowych. W metodyce TDD jedno nie jest mniej ważne od drugiego. Kent Beck, uważany

jest za ojca TDD, zachęca do stosowania prostych rozwiązań podczas kodowania. Podnosi to poziom zaufania do napisanego programu. Co to oznacza? Po pierwsze, piszemy tyle kodu, nie mniej nie więcej, aby zaspokoić test. Po drugie, programiści zazwyczaj ufają swojemu kodowaniu. Pisząc w podejściu tradycyjnym, zazwyczaj nie przykładają się do testów jednostkowych pisanych po fakcie albo nie piszą ich wcale. Problem zaczyna pojawiać się w momencie, gdy zaczynamy dokonywać zmian. Wówczas tak naprawdę nie mamy kontroli nad tym, co mogliśmy popsuć. Stosując TDD, kod produkcyjny jest z definicji w całości pokryty testami. Dzięki czemu nie obawiamy się, że możemy zepsuć funkcjonalność dokonując późniejszych refaktoryzacji.

Metodyka TDD poza wspomnianą zaletą bycia swojego rodzaju „strażnikiem zmian” posiada jeszcze jedną ważną zaletę. Testy jednostkowe są bardzo dobrym sposobem na dokumentowanie kodu. Stosowanie komentarzy nie zawsze jest dobrym wyjściem, ale także nie może być traktowane wyłącznie jako coś negatywnego. Kod, podczas rozwijania aplikacji, ulega częstym zmianom. Stosując TDD dostosowujemy także testy, natomiast komentarze bardzo często się przedawniają i tracą swoją aktualność. Powodem może być chociażby brak czasu na utrzymywanie dokumentacji w taki sposób, aby nadążała za zmianami. Czy też deweloper, który dokonuje zmian, nie ma zwyczaju pisania komentarzy. Natomiast w przypadku testów, jest on wręcz niejako przymusowany do dokonania ich aktualizacji. Ponadto bardzo często proces budowania aplikacji ściśle powiązany jest z systemem ciągłej integracji, który w wielu wypadkach, stosowany jest przez firmy oprogramowania jako kluczowe narzędzie zapewniające wysoką jakość oprogramowania. Jego konfiguracja nie pozwoli na wyprodukowanie paczek produkcyjnych bazujących na kodzie, który nie spełnia założonego poziomu pokrycia testów jednostkowych. Poza tym, testy zazwyczaj są wprost odzwierciedleniem procesu myślowego, którym kierował się programista podczas implementacji algorytmów programu.

Podsumowując, jeśli kod będzie pisany w taki sposób, że jest maksymalnie pokryty testami jednostkowymi, masz pewność, że twój następca będzie miał znacznie ułatwione zadanie czytając i próbując go zrozumieć. Zwalniasz go z konieczności ciągłego odpowiadania sobie na pytanie: co autor miał na myśli?

Metodyka nie powinna być także traktowana jako złoty środek na wszystkie problemy i bolączki inżynierii oprogramowania. TDD nie jest w stanie przetestować wszystkich przypadków, mogących wystąpić w działającej aplikacji. Nawet tych, które przewidzieliśmy na etapie jej projektowania. Przykładem tu mogą być testy funkcjonalne interfejsu użytkownika, programy, które współdziałają z bazą danych lub te, które są ściśle zależne od specyficznej konfiguracji środowiska, np. sieciowego. W takim przypadku metodyka zaleca, aby w modułach zatrzymać minimalną ilość logiki i przenieść ją do modułów, które są możliwe do przetestowania. Natomiast tak zwany świat zewnętrzny symulować między innymi przy pomocy obiektów mockowych, stubów, fake'ów czy też dummy obiektów.

Bolączką TDD jest fakt, iż testy oraz kod pisane są zazwyczaj przez tę samą osobę. Wyłączam tutaj sytuację pracy w parach, czy też recenzji kodu, ale o tym później. Programista w tym przypadku może nie przewidzieć pewnych zachowań programu, co może prowadzić do niezauważalnych, na etapie implementacji, błędów. Oczywiście, tak pisane testy również będą zawierały słabe punkty. Uruchamiane i kończące się pozytywnym wynikiem, dają nam złą ocenę sytuacji. Idąc dalej, posiadając spory zestaw takich testów, mamy fałszywe poczucia bezpieczeństwa.

Kolejnym negatywnym przykładem jest konieczność utrzymania testów, które jak wcześniej zostało wspomniane, muszą być tak samo traktowane jak kod produkcyjny. Stanowi to dodatkowy nakład wymaganej pracy, za który firma musi zapłacić. Szczególnie jest to niekorzystne, gdy testy napisane są w niewłaściwy sposób. Istnieje ryzyko, że kosztowne z punktu widzenia projektu testy zostaną zazwyczajnie zignorowane. Może to doprowadzić do sytuacji, w której pojawi się istotny i niespodziewany błąd, trudny do późniejszego wykrycia.

Niemniej jednak, rozważając wszystkie „za” i „przeciw”, warto jest tworzyć dobre testy jednostkowe. Pisząc i ucząc się poszerzamy swoją wiedzę oraz umiejętności w tym zakresie. Zawsze jest tak, że na pewnym etapie naszej przygody z programowaniem pojawia się moment, w którym oglądając się wstecz, myślimy sobie, że wiele rzeczy zrobiliśmy lepiej, inaczej. Cała trudność polega na tym, żeby na bazie własnych doświadczeń umieć krytycznie spojrzeć na swoje osiągnięcia i stwierdzić, że zawsze można coś ulepszyć, coś poprawić.

### Prostota kodu i refaktoryzacja

Pisząc kod programu starajmy się szukać prostych rozwiązań. Wykorzystujemy narzędzia i sposoby standardowe, które mogą pomóc w rozwiązyaniu danego problemu oraz są znane innym, na przykład wzorce projektowe. Jeśli możemy i jest to sensowne stosujmy je. Oczywiście, nie jest tak, że nie powinniśmy lub wręcz nie możemy stosować rozwiązań niestandardowych. Jeżeli wymaga tego sytuacja to dobrą praktyką jest skomentowanie tego w kodzie. Dodatkowo, dobrze jest umieścić informację, dlaczego tak zostało to zrobione. Zdarzają się deweloperzy, którzy do rozwiązywania prostych zadań stosują wyszukanych i wyrafinowanych sposobów. Starajmy się pokonywać mało skomplikowane problemy prostymi metodami.

Stosujmy podobne rozwiązania, jakie stosowane są przez innych. Dla przykładu, użycie pętli while zamiast for może być naszym wyróżnikiem innowacji, ale po co? Osoba, która będzie debugować nasz kod, będzie zachodziła w głowę, dlaczego akurat pętla while, a nie for. Metodyka nie powinna być także traktowana jako złoty środek na wszystkie problemy i bolączki inżynierii oprogramowania. TDD nie jest w stanie przetestować wszystkich przypadków, mogących wystąpić w działającej aplikacji. Nawet tych, które przewidzieliśmy na etapie jej projektowania. Przykładem tu mogą być testy funkcjonalne interfejsu użytkownika, programy, które współdziałają z bazą danych lub te, które są ściśle zależne od specyficznej konfiguracji środowiska, np. sieciowego. W takim przypadku metodyka zaleca, aby w modułach zatrzymać minimalną ilość logiki i przenieść ją do modułów, które są możliwe do przetestowania. Natomiast tak zwany świat zewnętrzny symulować między innymi przy pomocy obiektów mockowych, stubów, fake'ów czy też dummy obiektów.

Podsumowując, pisząc kod nie wyważajmy otwartych drzwi i nie starajmy się wynaleźć koła na nowo. Używajmy sprawdzonych rozwiązań i wzorców. Prosty i czysty kod jest łatwiejszy do zrozumienia, aby w modułach zatrzymać minimalną ilość logiki i przenieść ją do modułów, które są możliwe do przetestowania. Natomiast tak zwany świat zewnętrzny symulować między innymi przy pomocy obiektów mockowych, stubów, fake'ów czy też dummy obiektów.

Gdy dochodzimy do końcowego etapu rozwijania aplikacji, warto jest się na chwilę zatrzymać. Jest to dobry moment, żeby przejrzeć kod

źródłowy. Zobaczyć, czy jest coś, co można usprawnić. Sprawdzić, czy gdzieś nie powtarzamy implementacji. Jeśli tak, to może wyłączyć ten fragment do wspólnej metody, może wprowadzić abstrakcję, może zastosować wzorzec projektowy. Spożytkujmy ten czas, czytając kod jako deweloper, który będzie w przyszłości pełnił zadania utrzymywania. Próbując odpowiedzieć sobie na ważne pytanie, czy gdybym widział/widział ten kod po raz pierwszy na oczy, to czy byłbym/byłbym go w stanie zrozumieć?

### Zostaw po sobie kod w lepszej kondycji, niż gdy go zastałeś

Większość z nas niechętnie podchodzi do konieczności zmiany działającego kodu, tylko dlatego, że wygląda na „brzydkiego”. W większości przypadków, gdy coś zmieniamy w tych obszarach programu, czujemy przyzwolenie na to, żeby dodawać więcej „brzydkiego” kodu. Zrzucamy z siebie odpowiedzialność utrzymania kodu w „czystości”, tłumacząc sobie to w taki sposób, że przecież kod nie jest naszego autorstwa. Z takim nastawieniem niestety, pogarszamy i tak często złą sytuację. W pewnym momencie możemy doprowadzić do tego, że będziemy unikali tego kodu jak ognia. W wyniku czego, dostarczenie jakiekolwiek poprawki będzie wiązało się z myślą, że możemy zaburzyć jakąś funkcjonalność, która potencjalnie może być krytyczna dla działania całego systemu.

Dlatego jako deweloperzy powinniśmy przyjąć za świętą zasadę, że nie dopisujemy nowego kodu, dopóki nie uporządkujemy już istniejącego.

### Praca w parach i recenzja kodu

Praca w parach (pair working) jest to technika należąca do zwinnej (Agile) metodyki wytwarzania oprogramowania, polegająca na tym, że dwóch programistów pracuje razem przy jednej stacji roboczej. Jeden z nich, prowadzący, pisze kod, podczas gdy drugi pełni rolę obserwatora, recenzenta, który na bieżąco dokonuje analizy każdej powstającej linii kodu. Dodatkowo wspomaga rozwiązywanie problemów implementacyjnych, podsuwa nowe pomysły usprawniające funkcjonalność lub architekturę oraz zwraca uwagę na potencjalne przyszłe problemy działającej aplikacji. Dzięki temu prowadzący może skupić uwagę na dokończeniu swoich zadań, mając zabezpieczenie w postaci osoby obserwującej. Oczywiście nie jest tak, że role są z góry ustalone i sztywno utrzymywane przez cykl pracy. Wręcz przeciwnie, osoby powinny zamieniać się rolami w miarę możliwości jak najczęściej. Ponadto, pary także powinny się wymieniać między sobą. To jak często członkowie zespołu pracują ze sobą w parach powinno być ustalane przez nich samych.

Jak widać, w tym sposobie pracy nie ma nic nadzwyczajnego. Dwoje ludzi pracuje razem nad wspólnym problemem. Mogliby się wydawać, że jest to marnowanie zasobów ludzkich. Ponadto, każdy pracownik-programista posiada swoje indywidualne stanowisko pracy wyposażone co najmniej w komputer. Zatem można zauważać, marnotrawstwo sprzętu. Nasuwa się pytanie, dlaczego coraz więcej firm promuje tego typu rozwiązanie? Jak wiadomo duża konkurencja na

rynku informatycznym wymusza wprowadzenie czegoś, co będzie wyróżnikiem danej firmy. Coś, co pozwoli zbudować jej pozytywny wizerunek. Dobrym wyróżnikiem produktu jest jego jakość. Dlatego synergia dwóch, często zróżnicowanych umysłów, wydaje się tutaj właściwą odpowiedzią. Składa się na nią kilka czynników. Po pierwsze, praca dwóch różnych osób wprowadza różne doświadczenia, róźny punkt widzenia, następuje wymiana różnych możliwości rozwiązania problemu. Ścierające się ze sobą opinie prowadzą do wypracowania optymalnego i wspólnego rozwiązania. Po drugie, często podczas rozpracowywania danego problemu poszukujemy informacji, które przyczynią się do jego rozwiązania. Dwa różne umysły najprawdopodobniej będą dążyły do zdobycia tej informacji różnymi drogami. Dodatkowo zdobywając nową wiedzę. Możliwe iż, znalezione rozwiązania nie są tymi, których poszukujemy, ale na ich podstawie jesteśmy w stanie wypracować coś innowacyjnego, coś co będzie dobrem wspólnym. Zatem po raz kolejny różnorodność zdań prowadzi nas do osiągnięcia wspólnego celu. Po trzecie, ta sama osoba patrząc na problem z perspektywy roli prowadzącego, bądź recenzenta, może mieć zróżnicowane rozumowanie na temat problematyki danego zagadnienia. Czasami warto jest spojrzeć na problem z dystansu, wówczas najczęściej rodzą się trafione pomysły.

Następnym, bardzo ważnym czynnikiem przemawiającym za pracę w parach, jest budowanie więzi wewnętrz zespołu, poprzez lepsze poznanie mocniejszych oraz słabszych stron każdego z członków. Dodatkowo znacznie szybciej i łatwiej uczy się komunikować w zespole, gdyż problemy i ich rozwiązania stają się wspólne. Nie staramy się, z obawy przed negatywną oceną członków zespołu, zataić faktu, że nie potrafimy sobie poradzić z przeszkodą.

Kolejnym istotnym faktem jest nieustanna wymiana wiedzy między osobami w parze oraz innymi członkami zespołu. Mowa tu zarówno o zwykłych wskazówkach dotyczących reguł danego języka programowania, jak i sposobach projektowania architektury aplikacji. Ponadto pracując nad różnymi aspektami aplikacji, wiedza na jej temat propaguje się wewnętrz zespołu. Dzięki temu unikamy możliwości powstania zastoju w projekcie, spowodowanym chociażby tym, że kluczowa osoba zostaje wyłączona z pracy, na przykład z przyczyn zdrowotnych. Dodatkowo, ten typ pracy jest bardzo korzystny w przypadku, gdy para składa się z osób mniej i bardziej doświadczonych. Mniej doświadczony poszerza swoją wiedzę nabierając zaawansowanych umiejętności. Jednocześnie spojrzenie nowicjusza może rzucić nowe światło na pewne sprawy związane ze sposobem rozwiązania problemu.

Ostatnim bardzo ważnym czynnikiem, o którym warto wspomnieć, przemawiającym na korzyść, są względy ekonomiczne firmy. Przed wszystkim kod na bieżąco recenzowany charakteryzuje się przejrzystszą i prostszą architekturą, a co za tym idzie, mniejszą ilością powiązań wewnętrz i między poszczególnymi modułami. Przekłada się to, w oczywisty sposób, na mniejszą liczbę potencjalnych defektów. W późniejszej fazie projektu, pochłania mniej zasobów wyma-

ganych do utrzymania produktu. Wszystko to generuje mniej, często niepotrzebnych kosztów, skraca czas wypuszczenia wysokiej jakości produktu na rynek. Firma dzięki temu buduje swój dobry wizerunek. Grając wspólnie do tej samej bramki, ostatecznie wszyscy na tym zyskują.

Ze względu na różne czynniki, bardzo często projekty informatyczne realizowane są w oparciu o wąskie ramy czasowe. Wynikiem pośpiechu jest częste przymykanie oka na jakość kodu. Testy jednostkowe są pomijane, może pojawić się tłumaczenie, że działanie całej aplikacji od nich nie zależy. Często stosowane są „hacki”, tylko po to, żeby zrobić coś „na szybko”. Mając w zamyśle, że kiedyś do tego wrócimy i to poprawimy. Jednak bardzo często okazuje się, że zapominamy o tym, natomiast projekt się rozrasta, poprzez dochodzące nowe funkcjonalności. Wydawałoby się, że wszystko zmierza w dobrym kierunku. Problem zazwyczaj się pojawia, gdy oprogramowanie zostaje wypuszczone na rynek. Klienci zaczynają zgłaszać błędy, które muszą być poprawione, do tego przeważnie na wczoraj. Nasze uprzednie złe podejście zaczyna się na nas mścić. Firma zaczyna ponosić koszty za utrzymanie produktu. Można temu w dużym stopniu zapobiec stosując praktykę recenzji kodu. Jest to dobre rozwiązanie, gdy zespół projektowy, z pewnych przyczyn, nie ma w zwyczaju praktykować pracy w parach. Wówczas, w miarę postępu prac deweloperskich, warto od czasu do czasu poprosić kogoś, kto będzie mógł przejrzeć nasz kod oraz poznać nasz sposób myślenia. Postawi parę pytań dotyczących poszczególnych aspektów naszej implementacji. Zapyta o zastosowane rozwiązania danego języka programowania. Przedyskutowana może być architektura, funkcjonalność, czy też testy jednostkowe. Być może recenzent będzie w stanie podsunąć prostsze, bardziej funkcjonalne rozwiązania. Może także okazać się z kolei, że zastosowane rozwiązanie jest lepsze niż to, które było w jego zamyśle. Podobnie, jak przy pracy w parach, są to obopólne korzyści, z których warto skorzystać.

### Podsumowanie

Błędem jest myślenie o swoim kodzie źródłowym, że został napisany raz i nigdy nie będzie zmieniany lub też rozszerzany. Jak wiadomo, wymagania biznesowe projektów informatycznych ulegają częstym modyfikacjom. Prowadzą one nieuchronnie do zmian w aplikacji. Poçiaga to za sobą konieczność modyfikacji kodu źródłowego. Gdy te zmiany zachodzą w krótkim odstępie czasowym, od momentu zakoñczenia projektu, jest spora szansa, że programiści będą zorientowani w napisanym kodzie. W przeciwnym razie, sytuacja będzie często wymagała ponownego i dogłębnego przeanalizowania programu. Koniec końców może się okazać, że autor danego fragmentu kodu, będzie odpowiedzialny za wprowadzenie zmian lub za proces utrzymania aplikacji. Często, po dłuższym okresie zapominamy jakie rozwiązania stosowaliśmy podczas implementacji produktu. Właœciwie, będącmy czytać kod, jakbyœmy go widzieli po raz pierwszy. Dlatego warto go pisać czysto. Dołożyć starañ, aby w rezultacie by³ samodokumentujacy się. Miał pokrycie testami jednostkowymi, zawiœci były dobrze wyjaœnione w komentarzach, a moduły, z archi-

tektonicznego punktu widzenia, były powiązane ze sobą poprzez użycie jasno okreœlonych interfejsów API. Ostatecznie zmierzajmy do tego, aby powstający kod by³ łatwy w czytaniu i bez większego wysiłku zrozumiały przez innych oraz nas samych.

### Bibliografia

- [1] [http://en.wikipedia.org/wiki/Test-driven\\_development](http://en.wikipedia.org/wiki/Test-driven_development).
- [2] [http://en.wikipedia.org/wiki/Pair\\_programming](http://en.wikipedia.org/wiki/Pair_programming).

### Autor o swojej pracy

Pracuję w dziale NetAct Radio Configurator MBB CEM & OSS jako Java developer. Zajmuję się rozwijaniem oprogramowania, które pomaga oraz usprawnia operatorom zarządzanie sieciami komórkowymi. Podczas pracy korzystamy m.in. z narzędzi używanych podczas procesu konfiguracji sieci, czy elementów sieciowych technologii 2G, 3G i 4G.

### Krzesztof Bulwiński

Engineer, Software  
MBB CEMOSS OSS RD Pol RCM4

# Testowanie jednostkowe oprogramowania

Wojciech Pisarski  
Engineer, Software Development  
MBB SM SCM and Automation TA Core

NOKIA

Testowanie jest ważnym elementem procesu wytwarzania oprogramowania, jest też również bardzo często elementem lekceważonym. Programiści nie zawsze są w stanie dopatrzeć się wartości w pisaniu jakichkolwiek testów, w końcu nie wnosi to dodatkowych funkcji lub ulepszeń do gotowej aplikacji i zajmuje czas. Ponadto programiści mają w zwyczaju wykazywać dużą pewność siebie, przecież dobrze sprawdzają napisany kod i wszystko działa. Czasami zdarzy się błąd, ale przecież zawsze da się zrobić jakąś poprawkę w kodzie, a skoro coś zostało naprawione, to problem już nie wystąpi. Jak mylny jest to sposób myślenia przekonał się prawie każdy, kto pokazał swoją aplikację komuś do sprawdzenia. Często druga osoba potrafi znaleźć rażący błąd już po chwili używania aplikacji, bo autor zwyczajnie nie pomyślał, że użytkownik może użyć jej w taki, a nie inny sposób. Błąd zostaje naprawiony, ale chwilę później osoba testująca naszą aplikację znajduje następny, potem kolejny i tak dalej. Takie doświadczenie może być frustrujące, zapewne większość zdaje sobie sprawę, jak trudne jest znalezienie błędu we własnej pracy. W tym miejscu poczynający programista może zadać sobie pytanie, jak w takim razie radzić sobie i zapobiegać takim sytuacjom? W końcu nie w każdej aplikacji można pozwolić sobie na błąd, w niektórych zastosowaniach nawet najdrobniejszy może prowadzić do katastroficznych strat. W przypadku niektórych programów nie można pozwolić na wystąpienie nawet najmniejszego błędu, chociaż bywa, że są to ogromne i skomplikowane systemy pisane przez wiele osób, czasami na przestrzeni wielu lat.

Celem tego artykułu nie jest przedstawienie wszystkich możliwych elementów kontroli jakości w wytwarzaniu oprogramowania, lecz skupienie się na jednej z najbardziej podstawowych technik, jakim są testy jednostkowe (ang. unit tests). Zanim to się jednak stanie warto opowiedzieć o tym, czym jest testowanie i jakimi cechami charakteryzuje się dobry test.

## Czym jest test?

Test jest operacją, której przeprowadzenie pozwala określić, czy produkt lub proces spełnia stawiane mu wymagania. W kontekście wytwarzania oprogramowania naszym produktem jest aplikacja. Powinna ona działać zgodnie z przyjętymi z góry założeniami i celem naszego testu będzie stwierdzenie, czy faktycznie tak jest.

Testy jednostkowe są narzędziem, które w zautomatyzowany sposób na bieżąco informują programistę, czy jego kod działa zgodnie z założeniami.

Testy jednostkowe działają na bardzo niskim poziomie, testując niecałe funkcjonalności lub przypadki użycia aplikacji, ale poszczególne jednostki kodu, zazwyczaj funkcje lub metody klas. Takie testy są silnie sprzężone z tworzonym kodem, co daje wysoki poziom ich szczegółowości i umożliwia wykrywanie błędów na wczesnym etapie, zwykle zanim kod trafi z komputera programisty do repozytorium systemu kontroli wersji.

Podstawowa zasada działania opiera się tutaj o technikę podwójnego zapisu. Jest to technika stosowana w rachunkowości, dzięki której możliwe jest wykrycie błędów rachunkowych w momencie, w którym wystąpią poprzez zapisywanie każdej operacji podwójnie, dla rachunku wpływów oraz w postaci przeciwej dla należności. Bilans obu tych stron powinny sumować się do tej samej liczby. Wystąpienie różnicy oznacza wykrycie błędu i konieczność powtórnego sprawdzenia poprawności operacji. Zapobiega to sytuacjom, gdy jeden lub więcej błędów rachunkowych akumulowane jest w dłuższym czasie w sposób niezauważony.

W analogiczny sposób testy jednostkowe stanowią równoważną operację przeciwną do kodu, który testują. Kod odpowiada na pytanie, co się wydarzy przy wykonaniu operacji. Test jednostkowy jest pytaniem, czy to, co się wydarzyło jest oczekiwany wynikiem tej operacji. W przypadku rozbieżności test powinien zakończyć się niepowodzeniem, a programista powinien wstrzymać się z dalszym pisaniem nowego kodu przed określeniem i naprawieniem przyczyny rozbieżności.

## Jak testować?

Na początku należy zadać pytanie, co trzeba przetestować. Należy mieć jasny obraz celu: jakie są oczekiwania wobec tego konkretnego fragmentu kodu, jakie dane wejściowe przyjmuje, co zwraca, co jest oczekiwany wynikiem w poszczególnych przypadkach użycia, oraz kiedy wynik jest poprawny, a kiedy nie. Następny jest sposób, w jaki osiągamy pożądany efekt, czyli zastosowany algorytm. W momencie rozpoczęcia tworzenia testu rozwiązanie może być już gotowe, ale może też jeszcze nie istnieć. Przykładową techniką, w której testy powstają przed kodem, jest Test Driven Development (TDD). Więcej powiedziane zostanie w dalszej części artykułu. Kolejna jest procedura testu. Ta część opisuje, jak musimy zadziałać z kodem, żeby stwierdzić, czy dana funkcjonalność działa. Zawiera się w tym przygotowanie danych testowych, tj. takich danych wejściowych, dla których znamy oczekiwany efekt na wyjściu testowanego kodu. Ostatnim etapem jest weryfikacja, czyli jednoznaczne określenie wyniku testu. Odbywa się to zwykle poprzez porównanie danych wyjściowych z oczekiwany wynikami, ale może też być np. sprawdzeniem, czy nastąpiła określona sekwencja interakcji z innymi częściami systemu.

Pomimo tego pozornie złożonego opisu, modelowy test jednostkowy będzie zawierał w sobie trzy kroki. W literaturze taki schemat tworzenia testu opisywany jest jako given-when-then lub arrange-act-assert.

- Given/arrange – jest to krok przygotowania, tworzone są dane wejściowe, inicjalizowane są testowane klasy itp.
- When/act – wykonanie operacji, następuje użycie kodu zgodnie z testowanym scenariuszem i danymi przygotowanymi w pierwszym kroku.
- Then/assert – weryfikacja, sprawdzany jest efekt działania i porównywany z oczekiwany.

Przykład takiego podejścia przedstawiony jest na poniższym fragmencie kodu w Javie.

```
public class TestClass {  
  
    @Test  
    public final void maxReturnsBiggerNumber() {  
        // given/arrange  
        final int x = 1;  
        final int y = 3;  
  
        // when/act  
        final int max = Math.max(x, y);  
  
        // then/assert  
        assertTrue(max == 3);  
    }  
}
```

Wyraźnie widać podział na trzy etapy, przygotowanie, działanie i weryfikację. Zwrócić należy szczególną uwagę na trzeci krok. Wykorzystano tutaj metodę frameworka JUnit (standardowy framework do pisania testów w Javie) o nazwie **assertTrue()**. Jako jej parametr podane zostało wyrażenie, którego wynikiem jest wartość logiczna. Efektem działania jest poinformowanie frameworku o wartości tego wyrażenia. Możliwe do osiągnięcia są zatem dokładnie dwa stany, warunek został spełniony, co skutkuje przejęciem testu lub warunek nie został spełniony i test nie przeszedł. Jest to pierwsza i najważniejsza zasada pisania testów jednostkowych, musi on zawsze jednoznacznie stwierdzać jeden z dwóch stanów, albo test przeszedł, albo nie. Nie ma wartości pośrednich i nie ma też miejsca na interpretację wyniku, test tą interpretację przeprowadza za programistę.

### Kiedy i po co pisać testy?

Stosując technikę TDD czas poświęcony na pisanie testów jest bardzo duży, gdyż na przemian dopisuje się małe fragmenty testów i kodu oraz refaktoryzuje obie te części. Można powiedzieć, że pisanie testów zajmuje około połowy czasu poświęconego na programowanie. Wydaje się to absurdalnie dużą ilością w świetle tego, że testy nie są w ogóle częścią gotowego produktu, który otrzymuje klient. Stanowią jedynie pomoc przy jego tworzeniu. Dlatego wydaje się, że poświęcanie na nie takiego wysiłku jest nieuzasadnione. Zdarza się, że programiści tak myślą, ale jest to błędne myślenie. Testy nie są dodatkiem w procesie programowania, powinny one stanowić integralną i ściśle związaną z kodem produkcyjnym całość. Tylko wtedy w pełni można wykorzystać ich możliwości. Stanowią one niskopoziomową specyfikację działania kodu, co daje programistę pewność, że kod zachowuje się tak, jak powinien. To z kolei daje programistę pewność działania i zabezpieczenie przed własnymi błędami. Likwiduje też strach przed przypadkowym uszkodzeniem napisane-

go wcześniej kodu, czyli wystąpienia tzw. regresji. Czas i wysiłek potrzebny na naprawienie bieżącego problemu jest bardzo niski, gdyż dokładnie wiadomo, w jakim obszarze kodu wprowadzone zostały właśnie zmiany. Wykrycie błędu na późniejszym etapie zwykle prowadzi do żmudnego śledzenia, kiedy i w jaki sposób problem powstał oraz potencjalnie dalszego ryzyka w postaci poprawki do tego kodu, która może z kolei spowodować jeszcze inny problem sama w sobie.

Testy nie stanowią straty czasu, ale daleko idącą jego oszczędność. Ale to wszystko możliwe jest do osiągnięcia jedynie, jeśli testowanie kodu przebiega na bieżąco. Testy powinny powstawać równomierne z kodem, który testują. Powinny one również być często uruchamiane, żeby skrócić okresy, w których wprowadzane są modyfikacje w kodzie mogące spowodować problem.

Istnieją również długofalowe zyski z używania testów jednostkowych. Aplikacja może być tworzona i rozwijana na przestrzeni wielu lat. Wymagania mogą się zmieniać lub może zachodzić potrzeba dodawania nowych funkcjonalności, a w tym czasie zmieniać się może skład osobowy zespołu za nią odpowiedzialnego. W sytuacji, w której oryginalny autor kodu odchodzi z projektu, tracona jest duża część szczegółowej wiedzy o działaniu tego kodu. Autor może też zwyczajnie nie pamiętać, co i dlaczego robi kod napisany przez niego, jeśli od tej chwili minął dłuższy czas. Testy jednostkowe stanowią tutaj zabezpieczenie, ponieważ w nich zapisane są wymagania. Przy odpowiednio szerokim zakresie dobrze utrzymanych testów wdrożenie się programisty do nieznanego mu kodu przechodzi znacznie sprawniej.

Dodatkowym, pozytywnym efektem ubocznym pisania testów jest ich wpływ na jakość kodu produkcyjnego. Cechą dobrze napisanego kodu jest łatwość w jego przetestowaniu. Jeśli programista pisze kod z myślą o jego testowalności wymusza to na nim określone decyzje dotyczące np. jego odpowiedniego podziału na klasy i metody. Z kolei odpowiedni podział wprost przekłada się na czytelność, podział odpowiedzialności jednostek kodu i dobrą jego organizację.

### Cechy dobrego testu

Pisanie testów jednostkowych dla samego ich istnienia mija się z celem. Aby spełniały swoją rolę konieczne jest ściśle stosowanie się do zestawu reguł.

**1. Jednoznaczność** - Dopuszczalne są jedynie dwa stany, jakie może sygnalizować test: przeszedł lub nie przeszedł. Warunki muszą być jasno określone, a ich interpretacja nie nastręczać problemów. Tylko w ten sposób możliwa jest szybka całościowa analiza wyników dużych zestawów testów, jasno wskazane powinny zostać te fragmenty kodu, które swoich wymagań nie spełniają. Każda dodatkowa praca nad interpretacją wyniku prowadzi do zwiększenia wysiłku potrzebnego na stosowanie testów, a to prowadzi do mniejszej częstotliwości ich uruchamiania. Gdy testy nie są wykonywane tak często, jak modyfikowany jest kod zwiększa się ryzyko na ukrycie się błędów między kolejnymi wykonaniami, a jego naprawienie staje trudniejsze.

**2. Powtarzalność** - warunki i procedura przeprowadzania testu muszą dla danego stanu kodu dawać identyczny wynik przy każdym uruchomieniu testu. Nie może mieć miejsca sytuacja, w której test nie przechodzi raz na kilka uruchomień. Takie zachowanie testu prowadzi do znieczulenia programisty na podawany wynik i może prowadzić do niesłusznie ignorowania całego testu, co czyni taki test bezużytecznym.

**3. Wąski zakres** - test powinien zawsze testować jedną, niezależną i ściśle określona rzecz. Przykładowo prosta funkcja, której zwracany wynik zależy tylko i wyłącznie od podanego wejścia, a nie globalnego stanu, jest dobrym kandydatem na napisanie testu jednostkowego sprawdzającego jej działanie. Tak szczegółowy podział zakresu testu daje programistę precyzyjną informację, co do lokalizacji problemu w razie negatywnego wyniku. Jeśli sprawdzamy wiele rzeczy w jednym teście pojawią się element niepewności i konieczność bardziej szczegółowej inwestigacji faktycznego źródła problemu.

**4. Szybkość wykonania** - aby zapewnić, że testy są uruchamiane często niezbędne jest zapewnienie, żeby wykonywały się jak najkrócej. W idealnej sytuacji testy uruchamiane są automatycznie na komputerze programisty po każdej zmianie i zapisaniu pliku, co daje natychmiastową informację zwrotną o napotkanych problemach.

**5. Jakość kodu testu** - testy i kod produkcyjny stanowią spójną całość, zatem kod testów powinien być rozwijany stosując te same standardy, co w przypadku kodu produkcyjnego. To umożliwia szybkie dostosowanie testu do zmian w kodzie i kodu do zmian w teście wraz ze zmieniającymi się wymaganiami.

**6. Nazewnictwo** - nazwa testu powinna wprost oddawać, co jest testowane. W nazwie testu powinny być umieszczone informacje o tym, jaka część kodu jest testowana, jakie są warunki testu oraz spodziewany wynik.

**7. Dokładność** - dobre testy powinny sprawdzać nie tylko przykładowe, poprawne przypadki użycia danego kodu. Obok nich powinny znaleźć się również przypadki negatywne, czyli czy kod dobrze reaguje na niepoprawne dane wejściowe oraz warunki brzegowe, gdzie dane wejściowe są skonstruowane z granicznymi wartościami, tj. takimi na skrajnych miejscach w zakresie danych wejściowych lub w przypadkach szczególnych oraz ich okolicach, gdzie wykonanie testowanego kodu może przejść specjalnie przygotowanymi ścieżkami, które aktywują się tylko w danych warunkach.

**8. Niezależność** - wynik testu nie może zależeć od okoliczności wykonania. Test powinien w sobie zawierać całość danych i procedury koniecznych do jego przeprowadzenia i nie może polegać na zewnętrznych czynnikach np. na kolejności, w jakiej wykonywane są testy lub liczbie testów wykonywanych bezpośrednio po sobie lub równolegle.

Test nie powinien również wymagać do działania zewnętrznych zasobów, na przykład baz danych umieszczonych w sieci lub plików na dysku. Jest to zła praktyka, gdyż uzależniamy test od zewnętrznego środowiska, przez co tracimy na przykład zdolność przeniesienia naszego kodu wraz z testami w inne miejsce, za każdym razem musimy również odtwarzać stan środowiska.

Istnienie testów zależnych od zewnętrznych czynników świadczy zarówno o złej utrzymywanej projekcie, w którym nie zadbanio o dobrą organizację kodu poprzez separowanie odpowiedzialności na klasy i metody. Zadaniem programisty jest zadbać o to, by do takiej sytuacji nie doprowadzić lub dążyć do refaktoryzacji kodu, jeśli taki zostanie mu powierzony.

W praktyce nie zawsze istnieje możliwość jednorazowego uporządkowania źle napisanego projektu, głównie ze względu na pracochnośc takiego zadania. W takiej wypadku należy zadbać przynajmniej o to, by testy sprzątały po swoim wykonaniu. Sprzątanie po wykonaniu to cofnięcie wszelkich efektów ubocznych swojego wykonania, takich jak np. stworzone pliki tymczasowe lub połączenia sieciowe. Należy to traktować wyłącznie jako rozwiązanie przejściowe, które umożliwia pracę do czasu naprawienia problematycznego obszaru.

### Test Driven Development

Popularną w ostatnich latach techniką programowania jest Test Driven Development, czyli programowanie przez rozwój testów. Technika ta polega na podzieleniu pracy na trzy powtarzające się na przemian fazy, z których pierwszą jest napisanie prostego testu, następnie napisanie kodu spełniającego test, a na koniec refaktoryzacja stworzonego kodu w celu poprawy jego jakości. Rozpoczęcie pracy od napisania testu ma na celu zmuszenie programisty do wyjścia od założień do rozwiązania, dzięki czemu zachowana jest oryginalna intencja.

Zalecane jest sięgnięcie do literatury w celu lepszego zaznajomienia się z tą techniką.

### Podsumowanie

Pisanie testów jednostkowych jest niezbędnym elementem pracy zawodowego programisty ze względu na ochronę przed ludzkimi błędami, jak i w celu zapewnienia ciągłości w rozwoju i utrzymaniu projektów w obliczu zmieniających się składów zespołów za nie odpowiedzialnych. Testowanie jednostkowe umożliwia utrzymanie wysokiego tempa pracy nawet dla dużych, złożonych i starych aplikacji, gdyż daje programistę szybką informację o popełnionych błędach i daje możliwość ich wcześniego zwalczania. To podnosi pewność pracy i minimalizuje stres wynikający ze strachu przed popełnieniem błędu, a to ma bezpośredni wpływ na komfort i zadowolenie z pracy. Dlatego stosowanie testów jednostkowych nigdy nie powinno być pomijane lub traktowane jako rzecz o drugorzędnym znaczeniu.

## Bibliografia

Martin, R.C., Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall PTR 2008.

## Autor o swojej pracy

Pracuję w dziale Tools & Automation, gdzie zajmuję się głównie rozwojem aplikacji w Javie. Mój zespół rozwija rodzinę aplikacji Pegasus, które oparte są o platformę Eclipse RCP. Aplikacje te wykorzystywane są przede wszystkim do automatyzacji testowania i kontroli jakości komponentów stacji bazowych. Ja pracuję m.in. nad podstawowymi komponentami Pegasusa oraz funkcjonalnościami związanymi z testowaniem protokołów i komunikacji sieciowej. W dziale zajmujemy się też innymi projektami wykorzystywanymi przez zespoły badawczo-rozwojowe, w tym aplikacjami webowymi, ogólnymi zadaniami związanymi ze wsparciem i ulepszaniem procesów automatyzacji. W swojej pracy wykorzystujemy narzędzia automatyzacji takie jak system continuous integration Jenkins.

### Wojciech Pisarski

Engineer, Software Development  
MBB SM SCM and Automation TA Core

# „Clean Design”

Krzysztof Matuszek  
Engineer  
MBB SM SCM and Automation TA 2

NOKIA

## Wprowadzenie

Czym jest architektura systemu? Dlaczego jest tak bardzo istotna? Czym jest czysty kod? W niniejszym artykule zaprezentowano ogólne porady, z szerszego punktu widzenia, które pozwolą uczynić oprogramowanie lepszym i bardziej profesjonalnym. W dobie powszechnego dostępu do materiałów dydaktycznych, pozwalających na naukę dowolnego języka programowania, wydaje się, że umiejętność programowania nie jest żadną wyjątkową umiejętnością. Jednak pozory mogą mylić. Profesjonalne programowanie jest sztuką oraz dyscypliną, w której należy cały czas się kształcić i dążyć do doskonałości.

Gdy projekt dopiero się rozpoczyna, a żaden kod jeszcze nie został napisany (tzw. Greenfield Project), można ulec złudzeniu, że jakość kodu i jego czytelność zdaje się być drugorzędna. Najważniejsze jest to, że z zawrotną prędkością zleceniodawca ma dostarczane kolejne funkcjonalności, za które płaci.

Jednak po pewnym czasie sytuacja zaczyna się zmieniać. Oprogramowanie powoli staje się nieroziwalne. Poziom produktywności zespołu znacząco spada, a kod staje się skomplikowany. Dodanie funkcjonalności, która na początku zajęłaby kilka godzin lub dni, obecnie zdaje się wymagać nakładu kilku tygodni, miesięcy lub być nawet niemożliwa.

## Specyfika problemu

Często programiści próbują powierzchownie oraz szybko ratować bieżącą, bijącą na alarm sytuację, poprzez stosowanie różnego rodzaju „hacków”. Ostatecznie prowadzi to do jeszcze większego nieporządku oraz powstawania tzw. „ukrytej wiedzy”. Wraz z upływem czasu, proces ten zaczyna się coraz bardziej nasilać, dochodząc finalnie do punktu krytycznego, w którym programiści nie są już w stanie zarządzać swoim kodem. Co jest powodem tak wielkiego spowolnienia pracy? Problemem jest bałagan w kodzie, który został zrobiony na początku.

Czy zleceniodawca posiadający swoje plany biznesowe zaakceptuje zmniejszenie produktywności programistów o 80%? Czy dodatkowa presja na programistach jest w stanie cokolwiek zmienić? Pytania te są oczywiście retoryczne. Czy zatem zatrudnienie nowego programisty jest w stanie rozwiązać ten problem? Nowa osoba potrzebuje sporo czasu, żeby się wdrożyć. Dodatkowo któryś z obecnych programistów musi poświęcić nowej osobie czas, samemu odwlekając swoje własne zadania. Przepisanie całego systemu od nowa często jest nieakceptowane ze względów finansowych i czasowych przez organy zarządzające. Wniosek z tego jest następujący: jedynym sposobem, aby poruszać się szybko do przodu, jest poruszać się dobrze. Jedynym sposobem, aby poruszać się dobrze, jest poruszać się profesjonalnie.

Po czym można rozpoznać profesjonalizm danego programisty? Po jakości kodu przez niego opracowanego. Kod tworzony przez profesjonalistę jest elegancki i efektywny, prosty i bezpośredni, jest dokładnie taki, jakiego byśmy się spodziewali. Profesjonalny kod jest czytelny i łatwy do zmiany na nowe wymagania zleceniodawcy. Profesjonalny kod jest czysty.

Gdy pracownik wybiera się na urlop lub odchodzi z firmy, niedopuszczalną sytuacją jest pozostawienie przez programistę kodu, którego nikt inny nie byłby w stanie zrozumieć poza autorem. Nie dbając o swój kod, ile będzie potrzeba czasu po kilku miesiącach, aby ponownie wdrożyć się w swój kod i wprowadzić niezbędne zmiany w hobbyistycznym, prowadzonym samodzielnie projekcie?

W komercyjnym oprogramowaniu kod jest współdzielony przez wiele programistów, gdzie niejednokrotnie praca wymaga zaangażowania i współpracy wielu różnych osób w zespole. Sytuację dodatkowo komplikuje złożoność wymagań oraz rzeczywista liczba linii kodu, klas i metod implementowanego systemu. Poza wymogiem spełnienia wszystkich wymagań funkcjonalnych, system musi również spełnić dodatkowe wymagania względem efektywności, mechanizmów bezpieczeństwa, czasu odpowiedzi i możliwego obciążenia, obsługi transakcyjności, działania w środowisku rozproszonym i wiele, wiele innych.

Sam fakt poprawnego działania danego oprogramowania nie świadczy jeszcze o jego profesjonalizmie. Wraz z potrzebą dostarczania nowych funkcjonalności lub zmiany istniejących, wymogiem jest, aby kod nie tylko działał poprawnie, ale w sposób jasny wyrażał intencję autora, ułatwiając także jego zrozumienie, jak i wprowadzanie niezbędnych modyfikacji.

## Symptomy słabej architektury

Tak jak piękno można dostrzec dopiero w porównaniu z jego brakiem, tak też czysty kod można docenić dopiero po zetknięciu się z brudnym kodem. Istnieje kilka podstawowych objawów posiadania brudnego kodu oraz słabej architektury:

**Sztywność** (ang. rigidity) – architektura jest trudna do zmiany. Trudno dokonać jakichkolwiek zmian w systemie, gdyż wpływają one na wiele innych części systemu lub są one bardzo czasochłonne. Każda zmiana pociąga za sobą dużą ilość korekt kaskadowych wynikających z zależności pomiędzy modułami. Kiedy system posiada takie symptomy, programiści niechętnie coś zmieniają i ograniczają się tylko do wprowadzania krytycznych poprawek.

**Kruchość** (ang. fragility) – architektura jest podatna na awarie. Objaw blisko spokrewniony ze sztywnością. Dowolna modyfikacja aplikacji powoduje nieoczekiwane przerwanie poprawnego działania innych części aplikacji. Często miejsca, jakie ulegają uszkodzeniu, nie mają bezpośredniego związku z miejscem modyfikacji. Gdy kruchość systemu staje się coraz większa, prawdopodobieństwo kolejnej awarii wzrasta, co powoduje brak możliwości zarządzania oprogramowaniem. Dodatkowo każda poprawka wprowadza coraz więcej zamieszania, wprowadzając więcej błędów niż tych, które zostały

rozwiązane. Kadra kierownicza oraz zleceniodawcy mogą odnieść wrażenie, że programiści stracili kontrolę nad oprogramowaniem.

**Nieprzenośność** (ang. immobility) – niemożliwość ponownego użycia kodu w innym systemie, ponieważ dany moduł nie może być odseparowany od aplikacji. Często się zdarza, że programista odkrył, że podobna funkcjonalność została już napisana w innej aplikacji, jednak nie może jej użyć, ponieważ ryzyko oraz nakład pracy związane z oddzieleniem wybranej funkcjonalności jest zbyt duży i nie do zaakceptowania. W takich przypadkach programista musi przepisać na nowo daną funkcjonalność, zamiast skorzystać z tego, co już zostało zrobione.

**Lepkość** (ang. viscosity) – można wyróżnić lepkość architektury oraz lepkość środowiska (projektu). Natrafiając na potrzebę dokonania zmian w aplikacji, programiści mają najczęściej możliwość wykonania jej na kilka różnych sposobów. Niektóre ze sposobów podtrzymują dobrą architekturę systemu, inne z kolei ją łamą (tzw. „hacki”). Lepkość architektury jest wysoka, kiedy metody podtrzymujące dobrą architekturę są trudniejsze i dłuższe do zimplementowania niż metody łamiące architekturę. Z kolei lepkość środowiska jest wysoka, jeżeli środowisko pracy programisty jest wolne i nieefektywne. Przykładowo, jeżeli pobranie niezbędnych plików z systemu kontroli wersji zajmuje kilka godzin zamiast kilku sekund, programista może być skłonny do zredukowania swojej pracy i wykonania jak najmniejszej ilości zmian, nie przejmując się architekturą systemu.

#### Cechy dobrego oprogramowania

Oprócz poprawnego funkcjonowania oprogramowania (spełnienia wszystkich wymagań funkcjonalnych), należy dążyć do doskonałości w pięciu podstawowych cechach charakteryzujących dobry oprogramowanie, zdefiniowanych w ISO 9126 [1]:

**Niezawodność** (ang. reliability) – zdolność oprogramowania do wykonywania wymaganych funkcji w określonych warunkach przez określony czas lub dla określonej liczby operacji.

**Efektywność** (ang. efficiency) – zdolność oprogramowania do zapewnienia odpowiedniego poziomu wydajności, relatywnie do ilości zużytych zasobów, w określonych warunkach.

**Użyteczność** (ang. usability) – zdolność oprogramowania do bycia użytkownikiem, zrozumiałym, łatwym w nauce i atrakcyjnym dla użytkownika, gdy oprogramowanie to jest używane w określonych warunkach.

**Pielęgnowalność** (ang. maintainability) – łatwość, z którą oprogramowanie może być modyfikowane w celu naprawy defektów, dostosowania do nowych wymagań, modyfikowane w celu ułatwienia przyszłego utrzymania lub dostosowania do zmian zachodzących w jego środowisku.

**Przenaszalność** (ang. portability) – łatwość, z jaką oprogramowanie może być przeniesione z jednego środowiska sprzętowego lub programowego do innego środowiska.

Można zauważyć, że niektóre symptomy złej architektury, są bezpośredniem przeciwnieństwem powyższych cech dobrego oprogramowania:

- przenaszalność / nieprzenośność,
- niezawodność / kruchosć,
- pielęgnowalność / sztywność.

#### Zasady SOLID

Istnieje pięć podstawowych zasad dotyczących projektowania klas, nazywanych w skrócie SOLID, których należy przestrzegać, aby na bieżąco pilnować dobrej architektury systemu oraz ustrzec się przed symptomami złej architektury:

- zasada jednej odpowiedzialności (ang. Single responsibility principle),
- zasada otwarte-zamknięte (ang. Open/closed principle),
- zasada podstawienia Liskov (ang. Liskov substitution principle),
- zasada segregacji interfejsów (ang. Interface segregation principle),
- zasada odwrócenia zależności (ang. Dependency inversion principle).

Aby szerzej przybliżyć powyższe zasady, należy zrozumieć wpierw następujące pojęcia:

**Spójność** (ang. cohesion) – wyznacznik wskazujący czy dany moduł ma dobrze ukierunkowany jeden spójny cel.

**Zależność** (ang. coupling) – wyznacznik wskazujący stopień powiązania danego modułu z innymi.

Niski poziom spójności oraz wysoki poziom zależności modułu, powoduje trudności w jego rozwoju, testowaniu, zrozumieniu oraz ponownym użyciu. Celem dobrej architektury jest zapewnienie wysokiego poziomu spójności w modułach oraz zapewnienie niskiej zależności pomiędzy modułami.

#### Zasada jednej odpowiedzialności

Moduł powinien mieć dokładnie jedną, spójną odpowiedzialność oraz jeden powód do zmiany. Efektem stosowania się do powyższej zasady są projekty, w których klasy są małe i odpowiedzialne zaściśle określone zadania. Złamanie tej zasady najczęściej powoduje wzrost kruchosci, trudność w testowaniu danego modułu oraz niemożliwość jego ponownego użycia. Na kolejnej stronie zaprezentowano dwa przykłady złamania zasady, gdzie podane klasy posiadają kilka odpowiedzialności:

#### Listing 1

```
class PensjaRaport {  
    void polaczZBazaDanych() {  
    }  
  
    void generujRaport() {  
    }  
  
    void wypisz() {  
    }  
}
```

#### Listing 2

```
class Kalkulator {  
    void dodaj(int a, int b) {  
        wypisz(a + b);  
    }  
}
```

Klasa **PensjaRaport** odpowiada jednocześnie za wyliczenie raportu (logika), wypisanie (prezentacja) oraz połączenie z bazą danych (źródło danych). W klasie **Kalkulator**, metoda **dodaj** zarówno wylicza wynik (logika), jak i deklaruje sposób wyświetlania (prezentacja). Dla obu przypadków poprawnym rozwiązaniem byłoby rozdzielenie powyższych odpowiedzialności do różnych klas.

Jeżeli klasa ma więcej niż jedną odpowiedzialność, wtedy odpowiedzialności te zaczynają być od siebie zależne. Zmiany w jednej odpowiedzialności powodują osłabienie zdolności danej klasy do utrzymania innej odpowiedzialności. Efekty takiej architektury można dostrzec w trakcie dokonywania zmian, gdzie dana zmiana powoduje nieoczekiwane przerwanie poprawnego działania innych części aplikacji.

#### Zasada otwarte – zamknięte

Należy pisać tak moduły, aby mogły zostać rozszerzone, bez potrzeby ich modyfikacji. Innymi słowy, moduł powinien mieć możliwość zmiany wykonywanej pracy w celu spełnienia nowych wymagań biznesowych, bez rzeczywistej modyfikacji kodu źródłowego danego modułu.

Niestosowanie się do powyższej zasady będzie skutkowało ciągłą modyfikacją kodu przy pomocy instrukcji sterujących if/else oraz

switch w każdym fragmencie, w którym klasa sterująca jest zależna od konkretnej implementacji. Naturalną konsekwencją jest zwiększenie sztywności architektury oprogramowania.

Aby uniknąć zależności od konkretnych klas, należy stosować mechanizmy abstrakcji oraz odwrócenia zależności. Poniżej zaprezentowano przykład łamiący powyższą zasadę oraz poprawne rozwiązanie:

#### Listing 3

```
class FiguraPrezenter {  
  
    void prezentuj(String figura) {  
        if (figura.equals("prostokat"))  
            prezentujProstokat();  
        else if (figura.equals("kolo"))  
            prezentujKolo();  
    }  
  
    void prezentujProstokat() {  
    }  
  
    void prezentujKolo() {  
    }  
}
```

#### Listing 4

```
class FiguraPrezenter {  
  
    void prezentuj(Figura figura) {  
        figura.prezentuj();  
    }  
}  
  
class Prostokat extends Figura {  
  
    @Override  
    void prezentuj() {  
    }  
}  
  
class Kolo extends Figura {  
  
    @Override  
    void prezentuj() {  
    }  
}
```

Dzięki zastosowaniu zasady otwarte-zamknięte, dodanie kolejnej figury nie spowoduje potrzeby zmiany metody **prezentuj** w klasie **FiguraPrezenter**.

Stosowanie się do powyższej zasady nie zawsze jest możliwe, ponieważ w niektórych przypadkach trudno jest przewidzieć, czy będą potrzebne zmiany w danym module. Stosowanie powyższej zasady w każdym przypadku skutkowałoby zwiększym, niepotrzebnym stopniem skomplikowania oprogramowania. Dobrą praktyką jest stosowanie reguły zamknięte – otwarte w najbardziej oczywistych przypadkach oraz w sytuacji, gdy choć jedna zmiana została już wykonana.

### Zasada podstawienia Liskov

Reguła ta została sformułowana przez Barbarę Liskov w 1988 roku [2]:

Funkcje, które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów. Jeżeli S jest podtypem P, to obiekty typu P mogą być zastąpione przez obiekty typu S, jeśli zachowają w pełni wszystkie właściwości typu P.

Głównym celem zasady podstawiania Liskov, jest gruntowna analiza oczekiwania i zachowania danej klasy, zanim zostanie utworzona klasa dziedzicząca po niej.

Typowym przykładem złamania reguły jest kombinacja prostokąta oraz kwadratu. Chociaż koncepcjonalnie każdy kwadrat jest prostokątem, to odwzorowanie to nie może być przeniesione łatwo do świata programowania obiektowego. W poniższym przykładzie klasa **Kwadrat** dziedziczy po klasie **Prostokat**, łamiąc powyższą regułę ([Listing 5](#)).

Chociaż obie powyższe klasy w izolacji wydają się być poprawnie zdefiniowane, to jednak poprawność tego dziedziczenia można zweryfikować tylko na podstawie innej klasy – klienta, który będzie korzystał z metod klas **Prostokat** oraz **Kwadrat**.

Dlaczego powyższy przykład działa niepoprawnie? Klasa **Kwadrat** dziedzicząc z klasy **Prostokat**, przejmuje od niej dwie zmienne, które są jej niepotrzebne: wysokość i szerokość. Klasa **Kwadrat** posiada dodatkowy warunek początkowy, który wymusza równość szerokości i wysokości. W ten sposób klasa **Kwadrat** naruszyła kontrakt klasy bazowej **Prostokat**.

Programowanie kontraktowe (ang. Design by contract) jest koncepcją zdefiniowanym przez Bertranda Meyera [3]. Kontrakt pozwala zdefiniować, czego klient może oczekwać od danej klasy. Kontrakt składa się z warunków początkowych i warunków końcowych oraz invariantów. W kontekście dziedziczenia, obowiązuje dodatkowa reguła, która definiuje, iż klasy dziedziczące muszą honorować kontrakt klasy bazowej. Dokładna definicja brzmi:

### Listing 5

```
class Prostokat {
    protected int szerokosc;
    protected int wysokosc;
    void ustawSzerokosc(int x) {
        this.szerokosc = x;
    }
    void ustawWysokosc(int y) {
        this.wysokosc = y;
    }
    int wyznaczPole() {
        return szerokosc * wysokosc;
    }
}

class Kwadrat extends Prostokat {
    @Override
    void ustawSzerokosc(int szerokosc) {
        this.szerokosc = szerokosc;
        this.wysokosc = szerokosc;
    }
    @Override
    void ustawWysokosc(int wysokosc) {
        this.szerokosc = wysokosc;
        this.wysokosc = wysokosc;
    }
    public void test() {
        Prostokat prostokat = new Kwadrat();
        prostokat.ustawSzerokosc(10);
        prostokat.ustawWysokosc(20);
        assertEquals(prostokat.wyznaczPole(), 200); // BLAD
    }
}
```

- warunek początkowy w klasie dziedziczącej nie może być bardziej restrykcyjny niż w klasie bazowej;
- warunek końcowy w klasie dziedziczącej nie może być bardziej liberalny niż w klasie bazowej.

Innymi słowy, klasy dziedziczące nie mogą oczekiwać więcej oraz nie mogą oferować mniej. Bardziej restrykcyjny warunek oznacza, że

wszystkie ograniczenia nadal obowiązują oraz nowe zostały dodane. Bardziej liberalny warunek oznacza, że niektóre z poprzednich ograniczeń już nie obowiązują.

Zasada podstawienia Liskov powoduje, że trudno jest zaprojektować dobre hierarchie podklas z powodu możliwości złamania kontraktu. Z tego też powodu bardziej preferowanym stylem tworzenia zależności między klasami jest kompozycja niż dziedziczenie.

### Zasada segregacji interfejsów

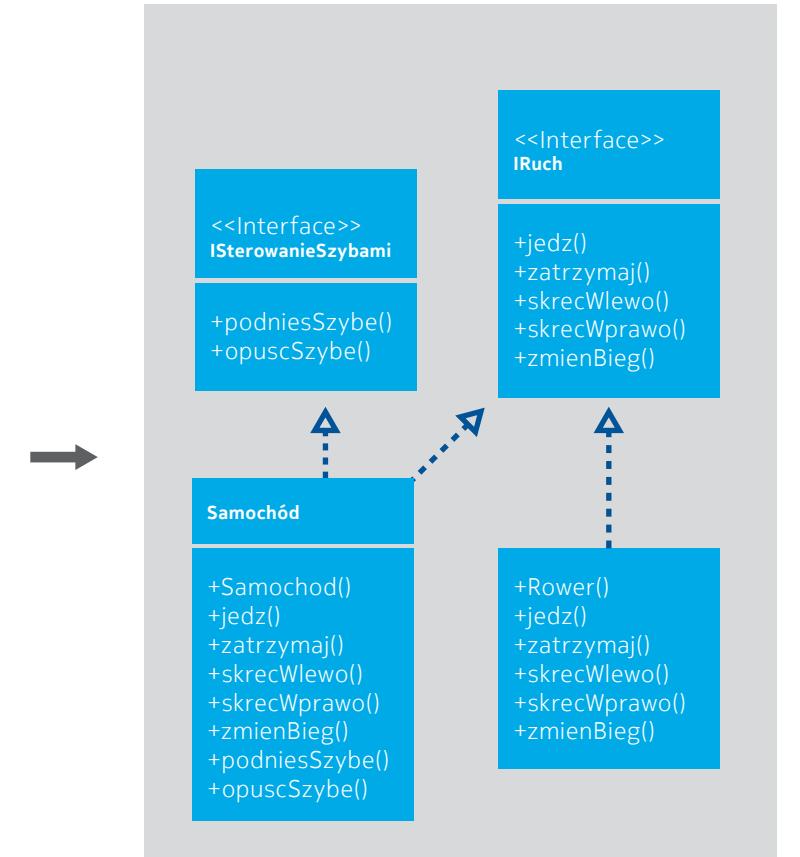
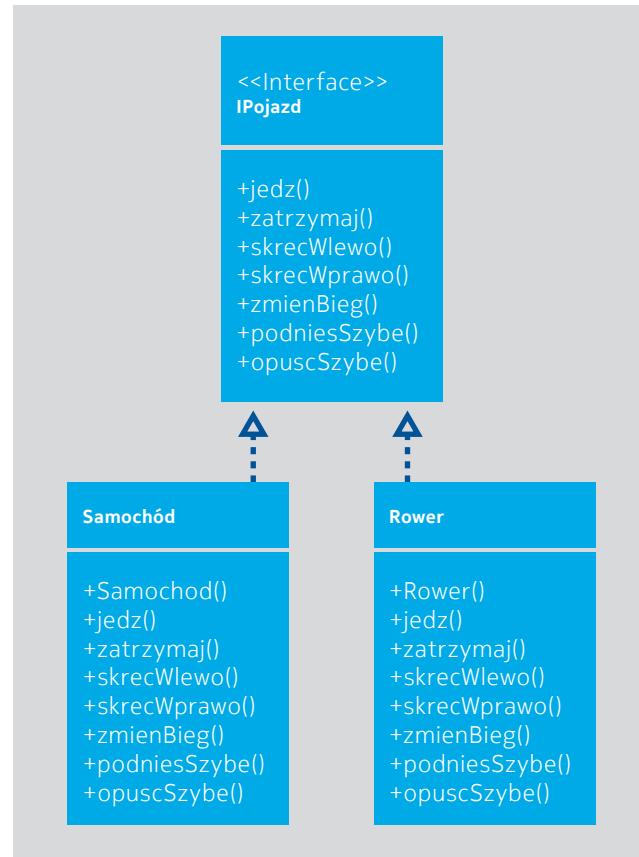
Klasy klienckie nie powinny polegać na interfejsach, których nie używają. Zamiast tego, klasy klienckie powinny korzystać tylko z niezbędnego interfejsów, posiadających minimalny zestaw odpowiedzialności.

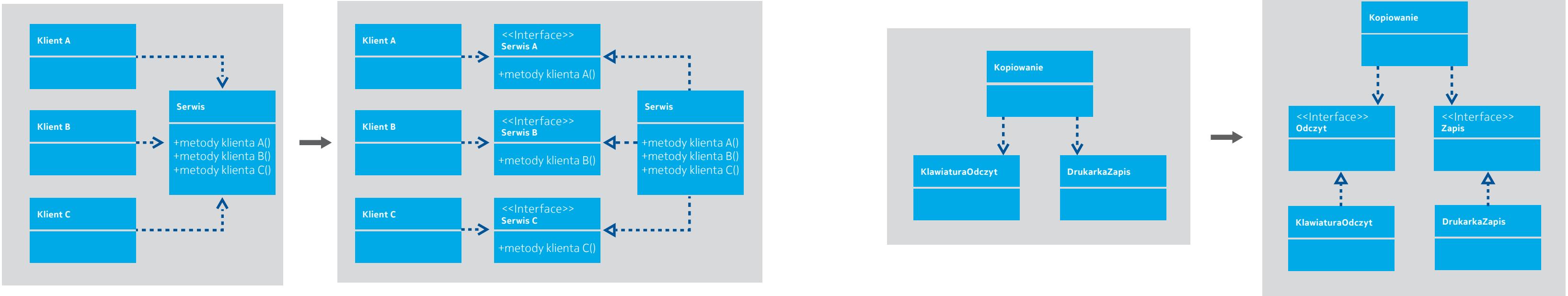
Jeżeli dany interfejs jest spójny oraz posiada jedną odpowiedzialność, jest wysoce prawdopodobne, że będzie mógł być ponownie użyty. Zasada ta jest blisko związana z zasadą jednej odpowiedzialności.

Poniżej zaprezentowano przykład łamiący powyższą zasadę – klasa **Rower** niepotrzebnie musi implementować metody **podniesSzybe** oraz **opuscSzybe**. W poprawnym rozwiążaniu, interfejs **IPojazd** został rozdzielony na interfejsy **IRuch** oraz **ISterowanieSzybami**, dzięki czemu interfejs **IRuch** może zostać wykorzystany przez klasę **Rower**.

Dodatkowo, dzięki rozdzieleniu odpowiedzialności na kilka interfejsów, zmiana w jednym interfejsie, od którego zależy jedna klasa kliencka, nie powoduje potrzeby rekompilacji oraz ponownego załączania innych klas klienckich. Jeżeli dane moduły mogą być ładowane niezależnie, oznacza to, że mogą być również rozwijane niezależnie, bez wzajemnego kolidowania i niepotrzebnego spowalniania pracy różnych zespołów programistów.

Zasada segregacji interfejsów jest również odpowiedzią na problem tzw. „tłustych klas”, czyli klas z bardzo dużą ilością metod, posiadających wiele odpowiedzialności, od których jest zależnych wiele innych klas. Klasy klienckie są zależne od takiej klasy z różnych po-





wodów biznesowych. Z tego powodu niepotrzebnie mają one możliwość wywołania wszystkich metod zdefiniowanych w danej klasie, choć będą korzystały w takich przypadkach tylko z minimalnego zestawu metod.

Jeżeli klasy klienckie zmuszone są do korzystania z interfejsu, którego nie używają, wtedy stają się one podległe zmianom, które zachodzą w tym interfejsie. Rezultatem takiego działania jest tworzenie niepotrzebnych zależności pomiędzy klasami klienckimi, co prowadzi do zwiększenia sztywności oraz kruczości.

Powyżej zaprezentowano przykład łamiący powyższą zasadę. W poprawnym rozwiązaniu każdy z klientów korzysta tylko z dedykowanego dla niego interfejsu. Serwis implementuje wszystkie interfejsy wymagane przez klientów.

**Zasada odwrócenia zależności**  
Wysokopoziomowe odpowiedzialności nie powinny polegać na niskopoziomowych, specyficznych implementacjach. Odwrócenie zależności oznacza poleganie bardziej na interfejsach lub klasach abstrakcyjnych, niż na konkretnych metodach czy klasach. Głównym powodem

złamania zasady segregacji interfejsów prowadzi do poważnych konsekwencji. Poniżej wymieniono kilka przykładowych symptomów:

- potrzeba utworzenia złożonej struktury obiektów, aby uruchomić dany test;
- potrzeba uruchomienia serwera webowego oraz podłączenia się do bazy danych, w celu przetestowania logiki biznesowej;
- potrzeba załadowania dodatkowej biblioteki, która nie będzie używana;
- potrzeba wywołania kilku innych metod, zanim będzie można wywołać docelową metodę.

Zasada segregacji interfejsów nie rekommenduje, że każda klasa korzystająca z danej usługi powinna mieć swój własny interfejs. Klasy klienckie powinny być raczej pogrupowane w kategorie, gdzie dla każdej kategorii powinien być utworzony osobny interfejs.

powstania tej zasady jest fakt, że konkretne implementacje często się zmieniają, z kolei elementy abstrakcyjne rzadko wymagają zmian. Abstrakcje pozwalają na rozszerzenie modułów bez potrzeby ich modyfikacji, co jest głównym punktem zasady otwarte – zamknięte.

Zasada odwrócenia zależności definiuje, że:

- Wysokopoziomowe moduły nie powinny być zależne od niskopoziomowych modułów. Obie grupy modułów powinny być zależne od abstrakcji.
- Abstrakcje nie powinny być zależne od szczegółów. Szczegóły powinny zależeć od abstrakcji.

Poleganie na abstrakcji oznacza, że każda zależność powinna być zwrócona w stronę interfejsu lub klasy abstrakcyjnej. Żadna zależność nie może być zwrócona w stronę konkretnej implementacji.

Powyżej zaprezentowano przykład łamiący powyższą zasadę – klasa **Kopiowanie** jest zależna od konkretnych implementacji **KlawiaturaOdczyt** oraz **DrukarkaZapis**. W poprawnym rozwiązaniu, klasa **Kopiowanie** jest zależna od abstrakcji **Odczyt** oraz **Zapis**. Klassy **KlawiaturaOdczyt** oraz **DrukarkaZapis** implementują odpowied-

nie interfejsy. Wszystkie zależności są zwrócone w stronę abstrakcji. Dzięki zastosowaniu zasady odwrócenia zależności, powyższy przykład mógłby być łatwo rozszerzony, przykładowo o odczyt danych ze skanera lub zapis do bazy danych. Klasa **Kopiowanie** stała się niezależna od detali implementacyjnych, dlatego dzięki obecnej swojej mobilności może zostać użyta w wielu różnych kontekstach. Niezależnie od tego ile klas będzie implementowało interfejsy **Odczyt** lub **Zapis**, klasa **Kopiowanie** nigdy nie będzie zależna od żadnej z nich. Tego rodzaju architektura nie ma wewnętrznych zależności, które mogłyby prowadzić do powiększenia sztywności lub kruczości.

Naturalną konsekwencją stosowania zasady odwrócenia zależności jest możliwość tworzenia architektury w oparciu o zestaw wtyczek. Główną zaletą mechanizmu wtyczek jest łatwość wymiany specyficznej wtyczki na inną. Dobra architektura zakłada, że wtyczkami są m.in. sposób komunikacji z użytkownikiem oraz mechanizm utrwalania danych. Dzięki takiemu rozwiązaniu, baza danych może być z łatwością zastąpiona mechanizmem zapisu do pliku, a graficzny interfejs użytkownika zastąpiony innym, przykładowo komunikacją poprzez konsolę.

Dobra architektura systemu powinna wyróżniać główne przypadki użycia danej aplikacji. Przykładowo, architektura systemu księgo-

wego powinna skupiać uwagę zespołu programistów na modułach wysokopoziomowych, sterujących regułami biznesowymi związanymi z płatnościami i operacjami księgowymi. To właśnie te moduły definiują tożsamość danej aplikacji i mają one pierwszeństwo przed modułami niskopoziomowymi. Z tego powodu moduły wysokopoziomowe nie powinny być nigdy zależne od modułów niskopoziomowych.

Gdyby moduły wysokopoziomowe zależały od modułów niskopoziomowych, każda zmiana w modułach niskopoziomowych skutkowałaby zmianą w modułach wysokopoziomowych. Reasumując, tylko moduły wysokopoziomowe, stanowiące trzon i serce danej aplikacji, mają prawo wymuszać zmianę w niskopoziomowych detalach implementacyjnych.

### Podsumowanie

W niniejszym artykule zaprezentowano podstawowe zagadnienia związane z utrzymywaniem stabilnej oraz czystej architektury, umożliwiającej łatwiejsze zarządzanie kodem, elastyczne wprowadzanie zmian oraz ponowne używanie komponentów. Zakres materiału nie pozwolił na opracowanie wielu innych, choć również istotnych, niskopoziomowych aspektów programowania obiektowego, wpływających na jakość oprogramowania, takich jak: konwencje nazewnicze, struktura metod i klas, wzorce oraz antywzorce obiektowe, techniki testowania czy przypadki użycia.

Poniżej podsumowano zbiór kilku dobrych praktyk z zakresu projektowania architektury oprogramowania, które warto zastosować w każdym projekcie:

- SOLID;
- YAGNI („You aren't gonna need it”) – nie należy pisać kodu, jeżeli rzeczywiście jeszcze nie jest potrzebny;
- DRY („Do not repeat yourself”) – nie należy powtarzać kodu;
- należy zawsze pozostawić kod bardziej czytelniejszym i czyściejszym niż się go zastało;
- nigdy nie należy rozwiązywać problemów z użyciem szybkich skrótów, które mogłyby prowadzić do powstawania tzw. „ukrytej wiedzy” w kodzie;
- należy wszędzie utrzymywać spójną konwencję;
- pisząc kod, należy być przewidywalnym dla innego programisty.

Warto zawsze pamiętać, że jakość kodu, jego architektura, uporządkowanie oraz czytelność, będzie świadczyć o jego autorze.

### Bibliografia

- [1] Słownik wyrażeń związanych z testowaniem, International Testing Qualification Board, 2011.
- [2] Liskov B., Keynote address – data abstraction and hierarchy, ACM SIGPLAN Notices 23, 1988.
- [3] Meyer B., Object-oriented Software Construction, Prentice Hall, New York 1988.
- [4] Martin R., Design Principles and Design Patterns, 2000, objectmentor.com.

### Autor o swojej pracy

Jestem absolwentem Informatyki na Wydziale Informatyki i Zarządzania Politechniki Wrocławskiej. Pracuję jako programista w dziale Tools & Automation. Jestem entuzjastą czystego kodu oraz dobrej architektury oprogramowania. Mój dział jest odpowiedzialny za tworzenie oprogramowania wspierającego testowanie stacji bazowych.

### Krzysztof Matuszek

Engineer  
MBB SM SCM and Automation TA 2

# Utrzymanie starego kodu dla zabawy i zysku

Michał Rudowicz  
Subject Matter Expert  
MBB Single RAN

NOKIA

Na studiach uczą wielu ciekawych i pozytycznych rzeczy, ale – jak pewnie wszyscy zainteresowani tematem zdążyli już zauważyc – są pewne kwestie, które są pomijane podczas wykładów, laboratoriów i ćwiczeń. Rozważmy przez chwilę wszystkie zadania, z którymi dane nam było się zmierzyć w trakcie edukacji na wyższej uczelni. Są to bardzo zróżnicowane zagadnienia, od baz danych, poprzez grafikę komputerową, aplikacje webowe na algorytmach heurystycznych kończąc. Z jednej strony mamy tutaj dość pokaźne spektrum problemów, jednak mają one wszystkie jedną wspólną cechę.

Opierają się na pisaniu kodu od zera.

Oczywiście, czasem dostajemy szkielet, na którym mamy oprzeć naszą pracę. Czasem zdarzy się otrzymać program z błędami, które należy poprawić. Jednak zawsze towarzyszy nam świadomość, że już za kilka tygodni zaprezentujemy owoc kilku, czasem kilkunastu wieczorów spędzonych przed komputerem, po czym program pojedzie w zapomnienie. Niektórzy zachowają go na pamiątkę gdzieś w zakamarkach swojego twardego dysku, inni podzielą się nim z koleżankami i kolegami z młodszego rocznika, aby oni mogli się nim zainspirować. Zwykle staramy się już nigdy o nim nie myśleć.

W tym artykule autor zabierze czytelnika w podróż do zeszłego dziesięciolecia. Wtedy autor był na pierwszym roku studiów, interesował się programowaniem i zajmował się nim w wolnym czasie, ale nie miał pojęcia na temat tego, jak powinno się programować. Z owej wycieczki przywieziemy pamiątkę – program implementujący Odwrotną Notację Polską, prezentujący przekrój błędów, ujawniający nieznajomość programowania obiektowego i ogólną żenadę. Autor postara się przewyciążyć wstyd towarzyszący otwieraniu każdego pliku po kolej i opisze, jak sprawić, aby kod stał się czytelny, łatwy w utrzymaniu i bardziej niezawodny.

## Środowisko pracy

Istnieje wiele błędnych sposobów na przechowywanie i organizację kodu źródłowego, najczęściej spotykany jest przechowywanie kolejnych wersji swojego programu w kolejnych katalogach. Jest natomiast jeden poprawny sposób i jest to użycie jakiegoś – w sumie dowolnego w granicach rozsądku – systemu kontroli wersji.

Autor swój kod źródłowy znalazł w formie skompresowanego archiwum – tak więc nie ma tam żadnej informacji o przeszłości i historii kodu. Ze względu na to, że lepiej zacząć późno niż wcale, zaczniemy rejestrowanie zmian w kodzie od teraz – i żadna nasza zmiana nie zostanie niezarejestrowana.

## Kontrola wersji

W tym artykule zostanie wykorzystany git, ale ilość dostępnych rozwiązań w tym temacie jest na tyle duża, że każdy znajdzie coś dla siebie. Od ulubionego przez pythonowców Mercuriala, poprzez bsr promowany przez Canonicala, SVN dla tradycionalistów (którego dodatkową zaletą jest prostota obsługi) aż po wynalazki w stylu Fossila.

Autor zaprezentuje tutaj podstawy git, który jest wykorzystywany podczas tworzenia jądra Linuksa i został w dużej mierze spopularyzowany poprzez GitHub. Autor zakłada, że git jest już wcześniej skonfigurowany, instrukcje opisujące, jak tego dokonać można znaleźć na wielu stronach internetowych, przykładowo w pomocy GitHuba [<https://help.github.com/articles/set-up-git/>].

Najwyższa pora odpalić konsolę. Należy wejść do katalogu, w którym wcześniej rozpakowany został kod programu. Następnie konieczna jest inicjalizacja środowiska gitowego, aby móc wykonać pierwszy commit:

**\$ git init**

I tak naprawdę to już wszystko – git jest gotów do pracy. Ale jeszcze żadna zmiana nie została zarejestrowana – pora więc wrzucić pierwszą wersję. Najpierw jednak należy powiedzieć gitowi, które pliki chcemy dołączyć. Można to zrobić w następujący sposób:

**\$ git add nazwapliku.cpp**

Można też dodać wiele plików jednocześnie, korzystając ze znajomej składni:

**\$ git add \*.cpp**

Następnie zmianę trzeba zarejestrować. Należy pamiętać, aby dodać czytelny opis, gdyż może to być w przyszłości bardzo pomocne:

**\$ git commit -m "Mój pierwszy commit"**

## Po co kontrola wersji?

Bardzo często przed nauczeniem się obsługi systemu kontroli wersji poczynający programista czuje opór, nie rozumiejąc korzyści płynących z wykorzystania owych rozwiązań. Dlatego poniżej zaprezentowane są najważniejsze zalety tych systemów:

- Możliwość oznaczenia działających wersji aplikacji;
- Wsparcie dla jednoczesnej pracy więcej niż jednego programisty nad kodem;
- Opis każdej wprowadzonej zmiany pozwala na śledzenie rozwoju aplikacji;
- Integracja z systemami zgłaszania błędów, pozwalająca na automatyczne zamykanie zgłoszeń;
- Ułatwione znalezienie autora danej zmiany w kodzie;
- Możliwość jednoczesnego rozwijania aplikacji i utrzymywania wersji stabilnej wraz z automatycznym synchronizowaniem zmian pomiędzy wszystkimi gałęziami kodu;

Wiele innych – dzięki czemu systemy kontroli wersji są standardem w każdej poważnej firmie informatycznej.

## CMake

W jaki sposób można skompilować naszą aplikację? Czy w jakikolwiek sposob sprawdzamy, czy wszystkie używane przez nas biblioteki są dostępne? Czy umożliwiamy konfigurację budowania aplikacji tak, aby w zależności od środowiska i preferencji użytkownika dostawał on inną binarkę? To wszystko są cechy dobrego systemu budowania aplikacji, które jest bardzo ciężko zapewnić w skryptach bashowych lub plikach .bat. Dodatkowo zależy nam na przenośności i łatwości obsługi.

W przypadku języka C++ bardzo popularnym narzędziem jest cmake, który zostanie użyty do zbudowania aplikacji.

Najpierw ustalmy minimalną wymaganą wersję CMake i nazwę projektu w nowo utworzonym w tym celu pliku **CMakeLists.txt**:

```
cmake_minimum_required(VERSION 2.8)
project(nazwa-naszego-projektu C CXX)
```

Następnie warto ustalić, z jakich katalogów powinny być brane pliki, które zostaną zaciągnięte poprzez użycie dyrektywy **#include**. Użyjemy w tym celu stałej CMake oznaczającej pełną ścieżkę do pliku **CMakeLists.txt**:

```
include_directories(${CMAKE_CURRENT_LIST_DIR}/src/ ${CMAKE_CURRENT_LIST_DIR}/libs/)
```

Następnie należy podać najważniejszą informację, czyli listę plików, które mają być przetworzone przez kompilator i użita do zbudowania wynikowego pliku wykonywalnego. Można tego dokonać w następujący sposób:

```
add_executable(${CMAKE_PROJECT_NAME}
    cli-onp.cpp
    Rownanie.cpp
    Stosik.cpp
    Znak.cpp)
```

Warto zwrócić uwagę na to, że pierwszym parametrem funkcji **add\_executable** jest nazwa pliku wynikowego – w tym przypadku jest ona taka sama, jak nazwa projektu.

Tak przygotowany plik CMakeList może posłużyć do otwarcia projektu w jednym z wielu wspierających go środowisk deweloperskich (np. Qt Creator czy CLion), lub do generacji projektów, które można otworzyć w innych środowiskach takich, jak np. Visual Studio. Na początek jednak wystarczy stworzenie pliku Makefile, dzięki któremu narzędzie make będzie w stanie zbudować nasz projekt. Warto również wykorzystać przydatną właściwość CMake: budowanie projektu w innym katalogu, aby móc łatwo oddzielić kod źródłowy od wynikowego (co jest przydatne np. przy commitowaniu zmian). Można do tego użyć graficznych nakładek na CMake, ale zdaniem autora

najprościej jest skorzystać z polecenia konsolowego:

```
$ mkdir output
$ cd output
$ cmake .. # konfiguracja projektu i tworzenie plików Makefile
$ make      # komplikacja i linkowanie
$ ./nazwa-naszego-projektu
```

Po wydaniu tych komend program zostanie zbudowany i uruchomiony. Jeśli wszystko poszło zgodnie z planem, warto odnotować to w systemie kontroli wersji – jeśli w przyszłości coś się nie uda, powrót do działającej, niezmodyfikowanej wersji projektu, którą można łatwo zbudować i uruchomić będzie się ograniczał do prostej gitowej komendy.

### Zabezpieczamy to, co już mamy

„Primum non nocere”

Każdy, kto używał kiedyś jakiekolwiek aplikacji, która wraz z kolejną wersją wprowadzała nowe błędy wie, że jest to najprostszы sposób na to, aby stracić użytkowników. Może także wzbudzić niechęć wobec aktualizacji oprogramowania do nowszej wersji, trzymając na dysku swoje ulubione narzędzie w wersji z 1997 roku na każdym dysku, ponieważ "nowsze wersje są mniej stabilne i wolniej działają". Modyfikując mało znany kod bardzo łatwo o popełnienie błędów i uszkodzenie poprawnie działającej funkcjonalności. Jedynym sposobem na uniknięcie tego problemu jest ciągłe testowanie i sprawdzanie, czy coś nie zostało popuszczone – ale czy koniecznie programista musi co chwilę uruchamiać swoją aplikację i zmudnie kontrolować każdą funkcję jedną po drugiej, aby upewnić się, że nie wystąpił żaden problem?

Cytując Billę Gatesa: „The computer was born to solve problems that did not exist before”, warto zmusić komputer, by wykonywał za programistę żmudną pracę. Szczęśliwie, do dyspozycji jest wiele narzędzi, które służą do automatyzacji testów – w artykule zostanie zaprezentowany Google C++ Testing Framework.

Kod testów powinien być wyraźnie oddzielony od kodu aplikacji – najlepiej jest stworzyć nowy podkatalog (np. **test**), w którym znajdzie się cały kod związany z testowaniem.

Dla uproszczenia w przykładzie zostanie zmodyfikowany istniejący plik CMake tak, aby od razu kompilował testy. Oczywiście w przypadku prawdziwych projektów powinno to być ustalone za pomocą flagi. Do pliku CMake należy dodać obsługę Google Testu:

```
enable_testing()
find_package(GTest REQUIRED)
```

Do **include\_directories** należy dodać katalog Google Testu, przechowywany w zmiennej  **\${GTEST\_INCLUDE\_DIR}**, uzyskiwanej

za pomocą wydanego wcześniej polecenia **find\_package**, do linkowania natomiast dodać biblioteki Google Testu i standardową funkcję **main** dołączaną do Google Testu, dzięki której testy będą uruchamiane automatycznie – i nic, z punktu widzenia programisty – nie trzeba więcej w tym kierunku robić. Oprócz tego Google Test wymaga dołączenia biblioteki **pthread**. Aby wykonać obie powyższe rzeczy należy dodać do pliku CMake następujące linijki:

```
target_link_libraries(${PROJECT_NAME} ${GTEST_LIBRARY} ${GTEST_MAIN_LIBRARY})
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread")
```

Uwaga: Należy pamiętać, aby z procesu komplikacji wyłączyć plik, w którym znajduje się napisana przez programistę funkcja **main**, ponieważ będzie ona w konflikcie z tą dostarczoną przez Google Test powodując błędy linkera. Należy również zadbać o dołączenie plików **.cpp** z testami. Aby tego dokonać, należy zmodyfikować listę plików **.cpp** branych do komplikacji w poleceniu **add\_executable**.

### Pisanie testów

Podstawowy plik z testami w Google Test wygląda następująco:

#### Listing 1

```
#include <gtest/gtest.h>
#include <Rownanie.h>

class ModuleTests : public testing::Test
{
protected:
    Rownanie tested;
};

TEST_F(ModuleTests, NazwaTestu) {
    EXPECT_EQ(1,1);
}
```

W tym pliku stworzone jest środowisko testowe (klasa dziedzicząca po **testing::Test**) składające się z jednego obiektu – instancji klasy **Rownanie**. Znajduje się tam również jeden test, o nazwie **NazwaTestu**, który sprawdza, czy 1 równa się 1. Każdy test przed rozpoczęciem tworzy środowisko testowe i niszczy je po zakończeniu, dzięki czemu możliwe jest uproszczenie kodu testów i uwspólnienie kodu potrzebnego w każdym teście.

Należy się teraz zastanowić, jakie przypadki użycia są obecne w programie. Po pierwsze, program musi poprawnie wykonywać podstawowe działania arytmetyczne – dodawanie, odejmowanie, mnożenie, dzielenie, pierwiastek kwadratowy. Po drugie, prawidłowa kolejność

wykonywania działań powinna być zachowana, włącznie z obsługą nawiasów. Po trzecie – należy pamiętać o sprawdzeniu obsługi liczb rzeczywistych.

W związku z tym należy napisać testy, które będą pokrywały całą powyższą funkcjonalność, ale nie będą się zgłębiały w szczegółach implementacyjnych, a jedynie korzystały z interfejsu dostępnego na najwyższym możliwym poziomie. Należy więc zaimplementować testy na wzór następującego:

```
TEST_F(CalcWrapperTests, MultipleCalculationsOnOneObject)
{
    EXPECT_EQ(tested.calc("2+2"),4);
    EXPECT_EQ(tested.calc("5+5"),10);
    EXPECT_EQ(tested.calc("5*5"),25);
}
```

Ze względu na ograniczoną ilość miejsca przedstawianie wszystkich możliwych testów w tym artykule jest niemożliwe, jednak są one pisane w analogiczny sposób – jedyne, co się zmienia to wartość wynikowa i wejściowe wyrażenie.

Czy jest jakiś sposób, aby sprawdzić, czy program jest już dostatecznie przetestowany? Tak naprawdę nie. Istnieją narzędzia mierzące stopień pokrycia kodu, jednak one tylko sprawdzają, czy dana linijka została wykonana czy nie – pozwalają, więc sprawdzić, czy dana część aplikacji została pominięta, ale w żaden sposób nie udowadniają tego, czy została poprawnie przetestowana. Możliwe bowiem, że wynik działania owego kawałka kodu nie był wzięty pod uwagę podczas pisania scenariusza testowego i nie jest porównywany z żadną wartością oczekiwana. Dlatego jedynym sposobem jest przeczytanie wszelkiej dostępnej dokumentacji, wymagającej dotyczącej działania programu (który to dokument jest tak naprawdę głównym wyznacznikiem tego, co tester powinien sprawdzać, dlatego posiadanie testów modułowych sprawdzających opisane scenariusze jest bardzo ważne) i przemyślenie, jakie testy powinny zostać napisane.

Należy tutaj zaznaczyć, że o ile najlepszą praktyką jest zawsze pisanie testów przed rozpoczęciem implementacji, tak w tym wypadku programista musi pracować z już napisanym, nieotestowanym kodem – jedynym możliwym wyjściem jest opieranie się na dokumentacji. Jeżeli takowa nie istnieje bądź jest niepełna – założenie, że kod w obecnej postaci jest bezbłędny i sposób jego działania musi pozostać zachowany.

Więcej informacji na temat funkcjonalności testowania można znaleźć w dokumentacji Google Test.

### Prawdziwa zabawa dopiero się zaczyna

Teraz, kiedy programista ma możliwość za pomocą jednej prostej komendy (no, może złożenia kilku komend w jednej, jednak jest możliwość podpięcia tego pod IDE, edytor bądź po prostu wywołanie

za pomocą klawisza strzałki w góre w terminalu) natychmiastowego wyłapania błędów popełnionych w trakcie modyfikacji kodu, można śmiało przystąpić do ambitniejszych działań.

Po pierwsze, dzięki napisaniu testów jesteśmy w stanie dokonywać modyfikacji w programie bez strachu o popuszczenie podstawowej funkcjonalności aplikacji. Oczywiście pewność siebie powinna być wprost proporcjonalna do jakości napisanych testów – ale jeśli praca została wykonana sumiennie, to nie powinno być z tym problemu. Programista jest więc przygotowany na poprawianie błędów w kodzie aplikacji, które będą zgłoszane przez testerów i użytkowników.

Regularnie należy również podejmować się prawdziwego refactoringu, czyli sprzątania kodu i sprawiania, aby zrozumienie go było prostym zadaniem. Oczywiście każda aplikacja posiada swoją specyfikę, i to programista powinien zadecydować, co jest największym problemem przy wprowadzaniu modyfikacji. Z doświadczeń autora najczęstszymi rzeczami, które można wykonać, aby przy relatywnie niskim ryzyku kod był czytelniejszy, są następujące zabiegi:

- Usunięcie jak największej ilości kodu na rzecz wykorzystania zewnętrznych bibliotek

Czy w kodzie znajdują się funkcje, które przeszukują łańcuchy znaków? Czy jest tam implementacja konwersji liczb na łańcuchy znaków i odwrotnie? Każda linijka kodu jest miejscem, w którym potencjalnie może ukrywać się błąd, najlepiej jest, więc usunąć te, które już zostały dla nas napisane. Przeszukiwanie tablic można wykonać za pomocą algorytmów STL-owych, parsowanie XML-a może być dla nas wykonane przez jedną z wielu bibliotek służących do tego celu. Wbrew pozorom wiele kodu zostało napisane przez osoby, które były nieświadome obecności wielu funkcjonalności w bibliotekach używanych w aplikacji – im szybciej własna implementacja (zwana często wynajdowaniem koła na nowo) zostanie usunięta, tym lepiej.

### Umieszczenie długich metod w osobnych klasach

Jednym z najczęstszych problemów w kodzie są metody, które nie mieścią się na jednym ekranie, posiadają wiele poziomów zagnieżdżenia, często nawet zduplikowany kod. Najprostszym sposobem jest przeniesienie takiej funkcji do osobnej klasy – początkowo klasa posiadałaby tylko jedną metodę, która zawierałaby cały wcześniejszy kod. Następnie wszystkie zmienne wykorzystywane w całym zakresie funkcji można przenieść do prywatnych pól klasy. Kolejnym krokiem jest już przenoszenie kawałków kodu do osobnych, prywatnych metod w tej klasie, wykonujących jedną prostą czynność.

Dzięki temu po niedługiej chwili programista będzie miał do czynienia z klasą zawierającą wiele małych metod, każda krótka i łatwa do zrozumienia i tylko jedną metodą publiczną, powodującą wykonanie całego scenariusza. Następnym krokiem może być znalezienie wspólnych metod w innych tego rodzaju klasach i uwspółnienie ich. Pozwala to na znaczne zwiększenie czytelności kodu przy dość ni-

skim ryzyku zepsucia czegokolwiek – kod nie jest modyfikowany, a jedynie przenoszony. Należy jednak pamiętać, że zawsze istnieje ryzyko pomyłki – dlatego tak ważne są testy oprogramowania. Jest to również dobry moment na napisanie testów charakteryzacyjnych (opisujących obecne działanie aplikacji) do takiej klasy.

### Zamiana wyrażeń logicznych na polimorfizm

Programista wiele razy staje przed problemem, w którym działanie pewnej klasy musi być tylko troszeczkę zmienione, tylko dla jednego, szczególnego przypadku. Najszybszym sposobem wydaje się wtedy dodanie do kodu nowej zmiennej typu bool, a następnie dodanie w odpowiednich miejscach warunków logicznych.

Jak łatwo się domyślić, takie podejście powoduje skomplikowanie klasy i należy go unikać. Jednak jest na to prosta metoda – można ów warunek przenieść do osobnej metody wirtualnej i stworzyć nową klasę dziedziczącą, bo klasie upraszczanej, w której owa metoda wirtualna będzie przeciążana. Dzięki temu metody będą krótsze, a kod czytelniejszy.

OCzywiście nie zawsze jest to dobry pomysł – zawsze należy pamiętać o tym, że najważniejsze jest zachowanie czytelności kodu, którym programista się opiekuje.

### Wykorzystanie map zamiast konstrukcji switch-case

Bardzo często w aplikacji wykorzystywana jest konstrukcja **switch-case**, lub – co gorsza – tzw. drabinki **if**-ów. Może to być łańcuch zastąpione mapą, gdzie kluczem jest wyrażenie, dla którego ma zostać wykonana dana czynność, natomiast wartością obiekt funkcyjny (klasa o zdefiniowanym operatorze nawiasów()), który wykonywałby daną akcję. Zalety takiego rozwiązania są następujące:

- Bardzo często scenariusze dla różnych wartości są podobne – można wykorzystać polimorfizm lub nawet parametryzację klas, aby uniknąć duplikacji kodu;
- Niemożliwe jest popełnienie błędu poprzez pominięcie instrukcji **break**;
- Możliwe jest dodawanie akcji w trakcie działania programu, dzięki czemu program może być rozszerzalny np. za pomocą wtyczek lub języka skryptowego;
- Kluczem mapy może być cokolwiek – programista ma pełną dowolność w napisaniu klasy, w której operatory porównania będą wykonywały dowolne czynności (oczywiście w granicach rozsądku).

Należy jednak pamiętać o tym, że **switch-case** ma również swoje zalety. Są to, między innymi:

- Większa czytelność w przypadku prostych akcji,
- Kod znajduje się w jednym miejscu i widać dokładnie, co kiedy się dzieje,
- Możliwość ominięcia instrukcji **break**, co daje programistie dodatkowe możliwości.

### Podsumowanie

Jak widać, utrzymywanie starego kodu nie musi być trudne, a jest zdecydowanie zajęciem potrafiącym dać sporo satysfakcji, choć trzeba przyznać, że jest to ten specyficzny rodzaj satysfakcji. Taki, jaki odczuwa osoba podziwiająca czysty dom po całym weekendzie sprzątania, odkurzania i wynoszenia śmieci. Bardzo często jest to praca ciężka i męcząca, jednak wdzięczność współdomowników, jak i świadomość wykonania dobrej roboty powoduje, że jest ona tego warta. No i nie można zapomnieć o możliwości mieskania w czystym i schludnym otoczeniu.

Na koniec należy wspomnieć, że nie są to jedyne możliwe sposoby na doprowadzenie do porządku starego kodu, jak i nie są to rzeczy, które należy robić za każdym razem. Zawsze należy się kierować zdrowym rozsądkiem, nie wolno zapomnieć, że najważniejszą rzeczą na początku pracy z kodem jest jego czytelność. Programista zawsze musi się wcześniej zastanowić, a najlepiej skonsultować z innym programistą bądź programistką, czy nie chce popełnić popularnego błędu zwanego over engineeringiem, gdy prosty problem jest rozwiązywany w niepotrzebnie skomplikowany sposób.

### Autor o swojej pracy

Od 3 lat jestem programistą w dziale rozwijającym oprogramowanie stacji bazowych dla WCDMA.  
Na co dzień korzystamy z C++ i Linuksa.

**Michał Rudowicz**  
Subject Matter Expert  
MBB Single RAN

# Programowanie a bezpieczeństwo

Maciej Jaskot  
Specialist, Software Development  
MBB Security

NOKIA

Bezpieczeństwo aplikacji jest istotnym elementem, który musi być uwzględniony podczas tworzenia oprogramowania. Wpływa na nie wiele aspektów jak na przykład użycia technologia. Obecnie programista ma do wyboru wiele języków programowania, bibliotek, które w mniejszym lub większym stopniu zapewniają bezpieczeństwo aplikacji. Jednak w tym artykule chciałbym się skupić na błędach popełnianych przez programistę, które mogą powodować luki w bezpieczeństwie. Świadomość zagrożeń oraz sposób ich zabezpieczenia z pewnością zmniejszy podatność na ataki tworzonego przez nas oprogramowania.

Jedną z najważniejszych cech dobrego programisty jest tworzenie kodu, który jest czytelny oraz dobrze udokumentowany. Dzięki temu będzie on łatwo utrzymywany dla nas i innych członków naszego zespołu. Stosowanie krótkich metod umożliwia na przykład szybkie wprowadzanie modyfikacji. Gdy logika programu jest skomplikowana i przepływ kodu nie jest czytelny, wówczas istnieje duże prawdopodobieństwo, że wprowadzona zmiana spowoduje błąd w programie. Funkcje oraz klasy powinny być raczej krótkie, dzięki temu będą one bardziej zrozumiałe.

W dalszej części dokumentu skupię się głównie na języku Java oraz błędach, jakich należy unikać, aby tworzyć bezpieczny kod. Większość opisanych sugestii można z powodzeniem stosować dla innych języków programowania np. C++.

## DoS – Denial of Service

Zagrożenie tego typu polega na zużyciu wszystkich zasobów dostępnych dla danego programu lub całego systemu. W wyniku czego program staje się nieużywalny dla innych użytkowników. Typowymi zasobami są: procesor, pamięć, dysk lub deskryptory plików. W sytuacji gdy program jest wykorzystywany przez jednego klienta i wystąpił na przykład problem z nadmiernym wykorzystaniem pamięci RAM, wówczas rozwiązaniem problemu jest zazwyczaj zamknięcie aplikacji. W systemie operacyjnym Linux możemy użyć słynnej komendy **kill -9 pid\_procesu**. W przypadku systemu, który obsługuje wielu użytkowników jednocześnie, takie rozwiązanie nie jest akceptowalne z punktu widzenia klientów oraz dostawcy takiej usługi. Wówczas nie można sobie pozwolić na restart całej aplikacji, ponieważ wszyscy aktywni użytkownicy zostaliby odcięci od systemu, na przykład systemu bankowego dostarczającego usługi online. W celu zabezpieczenia się przed takim atakiem, należy sprawdzać praktycznie wszystkie dane wejściowe, które będą przetwarzane przez program oraz limitować zasoby.

Należy unikać sytuacji, gdy zasoby będą nadmiernie użyte. Poniżej zebrałem listę przykładowych zagrożeń i sposobów zabezpieczenia się.

- Przekroczenie zakresu liczb całkowitych – jeśli zmienna całkowita używana jest jako indeks tablicy, może dojść do zmiany znaku zmiennej i odwołania się do pamięci znajdującej się przed tablicą

(ujemny indeks) – co zwykle spowoduje nadpisanie struktur kontrolnych programu.

- Przepelnienie bufora – problem z przepełnieniem bufora polega na wczytaniu zbyt dużej ilości danych do wskazanego przez programistę obszaru pamięci. Powoduje to zapis do sąsiadnego miejsca w pamięci. Problemy takie są ciężkie do wykrycia i mogą powodować niepoprawne zakończenie programu. W języku C i C++ należy stosować, na przykład funkcję **strncpy** zamiast **strcpy**.
- Bomba dekompresyjna – spreparowany plik archiwum zazwyczaj mały, który powoduje, że tworzony jest duży plik wynikowy podczas rozpakowywania, na przykład pliku archiwum. Najlepiej podczas rozpakowywania ustawić limit na rozmiar wynikowego pliku.
- Szczegółowe rejestrowanie działania aplikacji może prowadzić do szybkiego wyczerpania przestrzeni dyskowej. Jest to szczególnie istotne w systemach wbudowanych, gdzie dostępne zasoby są zazwyczaj skąpe. W celu uniknięcia powyższego problemu stare pliki z logami mogą być kompresowane, a gdy nie są już potrzebne powinny być usunięte. Należy również zastanowić się, jakie informacje są potrzebne, aby móc odtworzyć przebieg aplikacji.

## Pamiętaj o zwalnianiu zasobów

Każdy serwer i komputer w domu ma ograniczoną ilość zasobów, które są wykorzystywane przez aplikacje. Programista musi je zwalniać, ponieważ w przypadku ich wyczerpania może dojść do sytuacji, w której działanie aplikacji zostanie niespodziewanie zakończone. Może być to porównanie z zagrożeniem DoS. Otwarte pliki, mutexy, lock lub ręcznie zaallokowana pamięć są to zasoby, które muszą być szczegółowo nadzorowane. Utworzenie każdego z nich obliguje nas do jego zamknięcia w każdej sytuacji. Nawet w przypadku wystąpienia wyjątku. Na przykład w języku Java 7 dodano nową składnię **try-with-resource**. Umożliwia ona automatyczne zarządzanie zasobami. Dzięki niemu można zrezygnować z klasycznego bloku **finally**, w którym zwalniano zasób. Poniżej przedstawiony został przykład użycia:

```
try(Scanner sc = new Scanner(System.in)) {  
    String input;  
    System.out.println("Give input");  
    while(sc.hasNextLine() && !(input = sc.nextLine())) {  
        System.out.println(input);  
    }  
} catch(Exception e) {  
    //handling exception  
}
```

Obiekt musi implementować interfejs **AutoCloseable**, aby wykorzystać powyższą składnię. Inną techniką wykorzystywaną do zamykania zasobów jest wzorzec projektowy RAII, czyli Resource Acquisition is Initialization. Wzorzec ten jest bardzo popularny i często wykorzy-

stywany w języku C++. Na przykład **unique\_ptr** jest klasą opakowującą, która zarządza pamięcią danego obiektu. Zmienna ta prawie zawsze jest tworzona na stosie programu. Gdy kończy się zakres życia zmiennej, wówczas jej destruktor wywołuje **delete** na obiekcie, którym akuratnie zarządza. Dzieje się to również w przypadku wystąpienia sytuacji wyjątkowej. W takiej sytuacji stos jest czyszczony i zmienne, które były na nim utworzone będą niszczone (wywołane destruktory). Mechanizm RAII jest często wykorzystywany w języku C++ w przypadku zakładania blokad (**mutex**). Aplikacje wielowątkowe muszą zabezpieczać dostęp do zmiennych współdzielonych pomiędzy wątkami. Do tego celu używa się **mutex'ów** w C++ i **lock'ów** w Javie. Poniżej przykładowy kod w języku Java, który prawidłowo zwalnia zasób **locka**.

```
public void function() {
    lock.lock();
    try {
        //some actions
    } finally {
        lock.unlock();
    }
}
```

W przypadku braku **lock.unlock()** w sekcji finally mogłoby dojść do zablokowania działania aplikacji. Inny wątek w przypadku wywołania funkcji **function**, czekałyby na obiekcie **lock**, który nigdy nie zostałby zwolniony. W języku C++ do zwalniania blokad stosuje się mechanizm RAII, przykładowy kod na następnej stronie.

```
public void function(const std::string& message) {
    static std::mutex mutex;
    std::lock_guard<std::mutex> lock(mutex);
    std::ofstream file("file.txt");
    if (!file.is_open()) {
        throw std::runtime_error("Can't open file");
    }
    file << message << std::endl;
}
```

W sytuacji, gdy program nie będzie w stanie otworzyć danego pliku wówczas rzucony zostanie wyjątek. Obiekt **mutex** jest przekazany do **lock\_guard**, który implementuje mechanizm RAII. Gdy obiekt ten jest tworzony, wówczas na obiekcie **mutex** jest wywołana metoda **lock()**. Gdy obiekt jest niszczyony na obiekcie **mutex** jest wywołana metoda **unlock()**. Dzięki temu mechanizmowi programista nie musi pamiętać o zwolnieniu blokady oraz zapobiega to zablokowaniu aplikacji.

#### Dane poufne

Poufne dane powinny być traktowane przez programistę ze szczególną troską. Dotyczy to zwłaszcza informacji o loginach i hasłach. Dane te nie powinny być zapisywane w logach, gdyż mogą być one

przechwycone przez niepowołane osoby. Inną kwestią są informacje, jakie są umieszczane w wyjątkach. Można wyobrazić sobie sytuację, że podczas próby otwarcia pliku jest rzucany wyjątek, ponieważ plik lub katalog nie istnieje na danej maszynie. Zazwyczaj w takiej sytuacji programista umieszcza w wyjątku ścieżkę do pliku, który próbował otworzyć. Jednak taka informacja dla użytkownika końcowego może być cenną wskazówką na temat plików konfiguracyjnych ich lokalizacji itp. Należy pamiętać również, że atak na daną maszynę może być przeprowadzony w kilku fazach. Zdobycie informacji na temat systemu plików może być jedną z nich. Dlatego też informacje przekazywane w wyjątkach muszą być także przeanalizowane pod względem bezpieczeństwa. Informacje na temat struktury plików i katalogów zaliczają danych wrażliwych.

Język Java jak i wiele innych posiada mechanizm serializacji i deserializacji obiektów. Instancje klas są przekształcane w strumień bitów przy czym ich aktualny zostaje zapisany. Takie dane mogą być utrwalone na dysku lub przesłane do innego procesu. W Javie klasa musi implementować interfejs **Serializable**, aby móc korzystać z tej funkcjonalności. W przypadku procesu deserializacji stan obiektu zostaje odtworzony z wcześniej zapisanego strumienia bitów. Podobnie jak w poprzednich przypadkach należy unikać serializacji klas, które są ważne z punktu widzenia bezpieczeństwa. Gdy obiekt zostanie zapisany na dysku niepowołana osoba może dostać się do wszystkich składników klasy. Nawet tych, które są prywatne. Stąd zaleca się, aby poufne dane nie podlegały procesowi serializacji. Można to zrobić w następujący sposób. Pola, które nie mają być zachowane, muszą być poprzedzone słowem kluczowym **transient**. W przypadku procesu deserializacji należy sprawdzić, czy odtworzony obiekt jest poprawny. Założymy, że obiekt posiada zmienną **value**, która może być z zakresu od 0 do 10. Wówczas metoda **readObject** służąca do odtworzenia obiektu powinna mieć następującą postać:

```
public final class ExampleClass implements Serializable {
    private synchronized void readObject(java.io.ObjectInputStream input)
        throws IOException, ClassNotFoundException {
        ObjectInputStream.GetField fields = input.readFields();
        int value = fields.get("value", 0);
        if (value > 10 || value <= 0) {
            throw new InvalidObjectException("Value
not in range.");
        }
        this.value = value;
    }
}
```

Gdyby osoba niepowołana sprezarowałaaby deserializowany obiekt, wówczas powyższa implementacja metody wykryłaby taką sytuację i przerwałaaby deserializację.

Innym aspektem związanym z przechowywaniem danych poufnych w aplikacji jest czas ich przetrzymywania. Zdarzą się takie sytuacje, że błąd programu spowoduje utworzenie pliku core dump. Gdy taki plik dostanie się w nieodpowiednie ręce wówczas może być on użyty do wyśledzenia poufnych danych, które były przetrzymywane w pamięci aplikacji. Można oczywiście zminimalizować prawdopodobieństwo wystąpienia takiej sytuacji poprzez usuwanie z programu tego typu danych, kiedy są już one niepotrzebne.

#### Ataki SQL injection

Bazy danych zostały stworzone do przechowywania i grupowania informacji. Na rynku jest dostępnych wiele produktów. Najpopularniejsze są bazy danych, w których wykorzystywany jest język SQL (Structured Query Language). Ataki z grupy SQL injection wykorzystują fakt, że dane przekazane do aplikacji nie są w żaden sposób sprawdzane pod kątem ich poprawności. Jeśli klient wprowadzi dane w złym formacie wówczas może wystąpić błąd zapytania SQL. Górnzej jeśli wprowadzone dane zostaną sprezarowane w taki sposób, że zostanie wykonanie niezamierzone działania z punktu widzenia programisty, na przykład usunięcie całej bazy danych. Poniżej przedstawiony jest fragment kodu, który jest wrażliwy na atak SQL injection.

```
public void function() {
    con = pool.getConnection( );
    String command = "select * from user where username='"
    + username + "' and password='"
    + password + "'";
    statement = con.createStatement();
    obj = statement.executeQuery(command);
    if (obj.next()) {
        System.out.println("Logged in");
    } else {
        System.out.println("Not logged in, error");
    }
}
```

Jeśli osoba która próbuje się włamać jako **username** poda **admin**, a jako hasło następujący łańcuch: ' OR '1'='1 wówczas wynikowe zapytanie będzie miało postać:

```
select * from user where username='admin' and password='
' OR
'1'='1';
```

W takim wypadku, zamiast zwrócić krótką tylko wtedy, gdy podana nazwa użytkownika i hasło są poprawne, zapytanie zawsze zwróci pozytywny wynik, co więcej z uprawnieniami administratora. Taki niezamierzony błąd powoduje, że do systemu można się zalogować bez znajomości hasła użytkownika. Można również zmodyfikować atak poprzez zmianę parametru **username** na postać **admin') --**. W języku SQL dwa myślniki oznaczają komentarz, więc warunek dotyczący hasła przestanie być aktywny.

Wszystkie dynamiczne zapytania SQL powinny być parametryzowane. W takim wypadku powinny być używane **java.sql.PreparedStatement** lub **java.sql.CallableStatement**. Parametry przekazane do **PreparedStatement** będą automatycznie „eskejpowane” (escape) przez sterownik JDBC (Java DataBase Connectivity). Poniżej przykład użycia PreparedStatement.

```
String statement = "SELECT * FROM user WHERE Id = ?";
PreparedStatement prep = con.prepareStatement(statement);
prep.setString(404, Id);
ResultSet result = prep.executeQuery();
```

W powyższym rozdziale przedstawiłem jedynie mały fragment dotyczący ataków typu SQL injection. Zachęcam jednocześnie do dalszego studiowania tego zagadnienia zwłaszcza programistom, którzy korzystają z baz danych, ponieważ tego typu ataki są bardzo często stosowane. Podobnie jak w poprzednich rozdziałach nasuwa się wniosek, by dane wejściowe należy traktować jako niezaufane.

#### Programowanie obiektowe

Programowanie obiektowe wprowadza nowe zagrożenia dla kodu. Przy tworzeniu nowych klas należy używać modyfikatora dostępu **private** i słowa kluczowego **final** wszędzie tam, gdzie jest to możliwe. Metoda powinna być **public** tylko wtedy, gdy jest to wymagane przez inne klasy do poprawnego działania logicznego programu. Pola klasy również powinny być ukryte dla innych klas. Jeśli jest potrzeba ich pobrania lub modyfikacji z zewnątrz wówczas należy użyć metody **get** lub **set** dla danego pola. Używanie powyższych metod umożliwia programistom kontrolę nad poprawnością danych, także z punktu widzenia bezpieczeństwa. Na przykład może wypisać informację na ekranie kiedy dane pole było zmieniane i na jaką wartość. Poza tym można również sprawdzić poprawność nowej wartości. Jeśli jest niepoprawna, wówczas można odrzucić próbę modyfikacji.

Obecnie programiści często rozdzielają logikę programów na wiele mniejszych klas. Dzięki temu można użyć tego samego kodu w wielu miejscach jednocześnie. Gdy istnieje potrzeba zmiany klasy nadzornej programista musi zwrócić uwagę, czy wprowadzona np. nowa metoda nie spowoduje błędów w działaniu klas pochodnych. Dlatego należy starać się unikać takich modyfikacji. Jeśli jednak jest ona potrzebna należy zwrócić również uwagę na działanie klas pochodnych.

Należy unikać deklarowania zmiennej w klasie jako **public static**. Wynika to stąd, że praktycznie każda osoba może zmienić tę wartość. Może to spowodować błąd działania programu lub ujawnienie poufnych danych osobom trzecim. W przypadku, gdy zmieniona jest statyczna wówczas dwa lub więcej wątków mają do niej dostęp. Stwarza to niebezpieczeństwo, że jeden użytkownik, którego żądania są obsługiwane przez jeden wątek, będzie miał dostęp do danych drugiego użytkownika korzystającego z innego. Jeśli istnieje potrzeba udostępnienia pola należy dodać słowo kluczowe **final**, czyli **public static final**.

## **SecurityManager – Java**

W języku Java security manager jest obiektem aplikacji, który służy do sprawdzenia, czy dana operacja może być wykonana. Pełni on bardzo ważną rolę w definiowaniu i wymuszaniu polityki bezpieczeństwa. Domyślnie aplikacja jest uruchomiona bez skonfigurowanego **SecurityManagera**. Można go włączyć poprzez opcję **-Djava.security.manager** w JRE. Politykę można skonfigurować poprzez opcję **-Djava.security.policy==policyURL**.

Gdy **SM** jest uruchomiony i aplikacja próbuje wykonać niebezpieczną operację, np. zapis do pliku, wówczas sprawdza on czy jest ona dozwolona. Jeśli zdefiniowana polityka nie zezwala na zapis wówczas zostanie wyrzucony wyjątek **security exception**. Poniżej przykład implementacji metody **exit**.

```
public void exit(int status) {  
    SecurityManager security = System.getSecurityManager();  
    if (security != null) {  
        security.checkExit(status);  
    }  
    Shutdown.exit(status);  
}
```

Jak widać w powyższym przykładzie, gdy metoda **getSecurityManager()**wróci **null**, sprawdzenie, czy wywołanie metody **exit** nie będzie sprawdzone. Wykonanie potencjalnie niebezpiecznej operacji wiąże się każdorazowo z wywołaniem na obiekcie SM metody, która sprawdza, czy operacja jest akceptowalna. Należy pamiętać, że polityka bezpieczeństwa może się zmieniać podczas działania aplikacji. **SecurityManager** daje duże możliwości związane z nadawaniem uprawnień do konkretnych niebezpiecznych operacji, co wpływa na bezpieczeństwo aplikacji.

## **Podsumowanie**

Opisane powyżej sugestie oraz błędy są tylko małą częścią związaną z bezpieczeństwem. Zachęcam oczywiście do dalszego poszerzania wiedzy. Tworząc ten dokument, korzystałem zarówno z własnych doświadczeń, jak i dokumentów dostępnych na stronie [www.oracle.com](http://www.oracle.com).

## **Autor o swojej pracy**

Pracuję w dziale MBB Security. Zajmuję się głównie tworzeniem nowego oprogramowania. Obecnie jestem zaangażowany w projekt aplikacji związanej z bezpieczeństwem sieci szkieletowej. Projektowana aplikacja będzie wykorzystywała wiele technologii związanych np. z bazami danych, rule engine, interfejsem użytkownika, Java EE itp. Nasza praca wiąże się z różnymi aspektami bezpieczeństwa IT.

## **Maciej Jaskot**

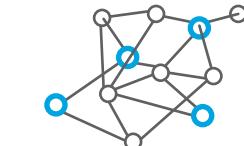
Specialist, Software Development  
MBB Security

# Software Configuration Management – czym jest i do czego służy?

Andrzej Lipiński  
R&D Manager  
SCM; MBB System Module

**NOKIA**

Tworzenie oprogramowania w dużych projektach jest bardzo złożonym procesem. Duże projekty programistyczne to takie, w których nad końcowym wynikiem pracują setki lub tysiące ludzi. Zasada jest prosta – im bardziej skomplikowany projekt, tym więcej ludzi jest w niego zaangażowanych. To właśnie ów efekt skali przyczynia się głównie do wprowadzenia złożoności. Dwóch lub trzech programistów pracujących nad prostą stroną WWW bez problemu będzie w stanie skoordynować swoją pracę i wyjaśniać na bieżąco pojawiające się problemy. Jednak inaczej to wygląda w projekcie pracującym nad systemem operacyjnym lub jak w jednym z projektów, nad którymi pracujemy w Nokii – oprogramowaniem uruchamianym na stacjach bazowych tzw. BTS-ach. Osoba pracująca w kilku – lub kilkunastoosobowym zespole może wówczas napotkać już duże problemy, próbując zrozumieć wpływ wprowadzonych przez siebie zmian w kodzie na efekt pracy osób w kilku innych powiązanych wzajemnymi zależnościami zespołów.



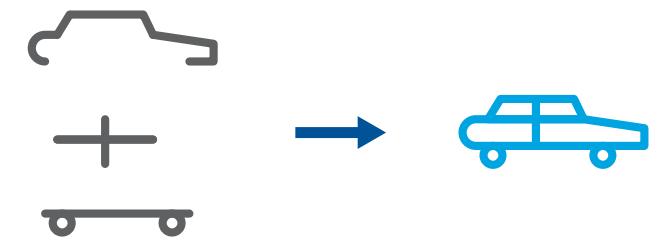
Chcąc zapewnić sukces całemu projektowi konieczne jest ustalenie zarówno reguł pozwalających na sprawne składanie oprogramowania tworzonego równolegle przez wiele zespołów jednocześnie w jedną spójną całość, jak i określenia jasnych reguł współpracy pomiędzy wieloma zespołami tworzącymi poszczególne części danego projektu.

To właśnie od Software Configuration Management oczekuje się, by dozbroić projekty programistyczne w odpowiednie narzędzia i procesy, które pozwolą sprawnie tworzyć kolejne generacje oprogramowania i nie zginąć w gąszczu rozwijanych równocześnie wariantów oraz ich wersji opracowywanych na przestrzeni czasu.

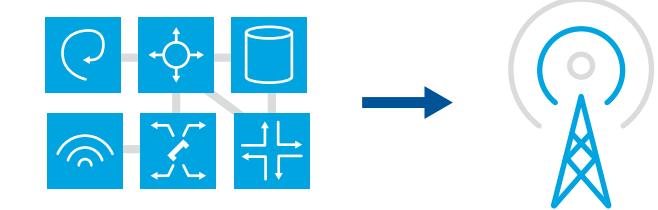
## Przez analogie

Programy czy aplikacje tak samo jak np. samochody składają się z wielu komponentów współpracujących ze sobą. Komponenty te są tworzone przez wysoko wyspecjalizowane w swojej dziedzinie zespoły. Wynik pracy takich zespołów (tworzących np. czujniki cofania, skrzynie biegów itd.) to materiał do pracy dla kolejnego zespołu zajmującego się składaniem wszystkich części składowych w jeden działający samochód. Na filmach z fabryk samochodów zwykle widzimy całe rzędy ramion robotów, które 24 godziny na dobę składają z dostępnych komponentów kolejne modele samochodów.

Ma to swój odpowiednik w świecie tworzenia oprogramowania i nazywa się „integracją oprogramowania”. Dedykowane zespoły zajmu-



ją się kompletowaniem odpowiednich komponentów aplikacji (odpowiednich – czyli w odpowiedniej wersji, zawierających odpowiednie funkcjonalności oraz zweryfikowanych przez odpowiednie testy jakości) i składaniem ich w jedną funkcjonalną całość.



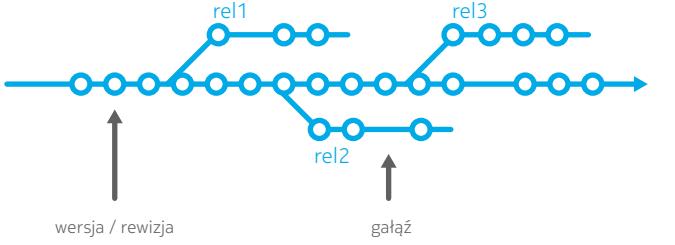
W dużych projektach programistycznych – tak samo jak na taśmie produkcyjnej w fabryce samochodów – składaniem oprogramowania również zajmują się automaty: dedykowane programy, skrypty, narzędzia, które automatycznie, jedynie pod nadzorem człowieka, integrują (łączą w całość) najnowsze wersje komponentów oprogramowania w jedną działającą aplikację.

W obu przypadkach – produkcji samochodów oraz tworzeniu oprogramowania na dużą skalę potrzebujemy reguł i odpowiednich narzędzi, które w automatyczny sposób zbiorą wynik pracy poszczególnych zespołów – równolegle pracujących nad różnymi komponentami – i w ustalony sposób złożą je w funkcjonalną całość. Te reguły opisują zasady naszej codziennej pracy noszą miano „procesów”.

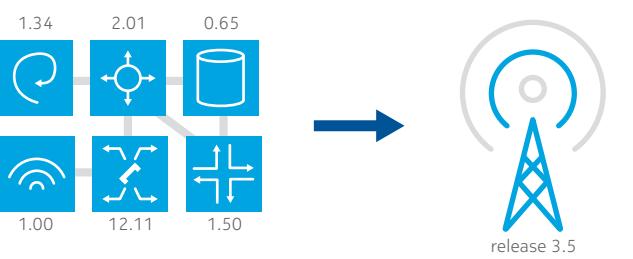
## Konfiguracje

Aby ułatwić sobie poruszanie się w ramach projektów programistycznych umawiamy się na pewne reguły nazywania elementów naszego środowiska pracy oraz relacji między nimi. Pisząc oprogramowanie wykorzystujemy „Systemy kontroli wersji” – obecnie najczęściej możemy się spotkać z następującymi systemami kontroli wersji: GIT, Subversion, ClearCase, Perforce, Mercurial, Bazaar. Pomagają nam one przechowywać kolejne „wersje” lub inaczej „rewizje” naszego programu rozwijanego w ramach różnych „gałęzi”.

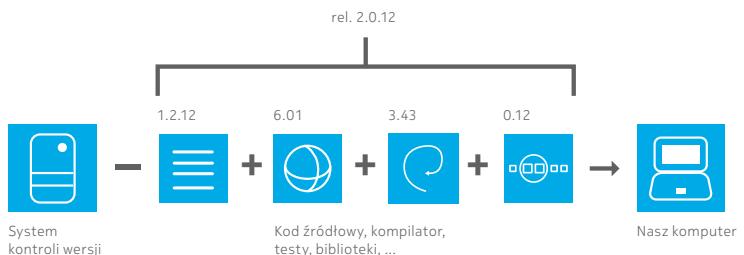
Pomyślnie „zbudowane” oraz zweryfikowane poprzez różnego rodzaju „testy” wersje naszej aplikacji są następnie „dostarczane” do naszych klientów. Wszelkie informacje, dotyczące powyższych elementów i relacji pomiędzy nimi, pozwalające nam jednoznacznie zidentyfikować konkretną wersję naszej aplikacji noszą miano “kon-



figuracji” (ang. configuration). Jakie komponenty weszły w skład konkretniej konfiguracji i w jakich wersjach, przy pomocy, jakich narzędzi zostały one zbudowane (i w jakich wersjach) oraz przy pomocy, jakich testów (i w jakich wersjach) została zweryfikowane – skompletowanie całości takiej informacji pozwala nam w razie potrzeby odtworzyć w dowolnym momencie dowolne środowisko produkcyjne i przeanalizować zgłoszone problemy.



Projekty programistyczne, na co dzień nie są w stanie wykryć wszystkich błędów w swoim kodzie źródłowym. Od danego projektu zależy jak wysoko stawia sobie poprzeczkę i jak dalece jego produkty zbliżają się do 100% bezbłędności. Aplikacje przed wypuszczeniem na rynek są gruntownie i długo testowane, a jakość testów i ilość scenariuszy testowych jest obiektem ciągłej pracy zmierzającej do ich poprawy. W jak najszybszym naprawianiu zgłoszonych problemów pomaga właśnie skrupulatne gromadzenie “konfiguracji” – dzięki zautomatyzowaniu naszych środowisk pracy jesteśmy w stanie odtworzyć odpowiednie środowisko (pobrać właściwą wersję kodu źródłowego naszego komponentu z systemu kontroli wersji, odpowiednie wersje kompilatora, bibliotek źródłowych, testów, ...) w ciągu chwili i natychmiast rozpocząć pracę nad korekcją błędów.



Podobnie w sytuacji, gdy projekt zadecyduje o rozpoczęciu prac nad alternatywną wersją swojej aplikacji, po podjęciu decyzji, na bazie, której wersji obecnej aplikacji powinno rozpocząć się prace, tylko kilka poleceń wystarcza do skompletowania funkcjonalnego środowiska pracy dla całego projektu.

### Koordynacja

Drugi aspekt – reguły współpracy – zapewnia nam wiedzę o tym, kto, za co w naszej projektowej rodzinie odpowiada i z kim należy rozmawiać, kogo słuchać, kto będzie słuchał. Generalnie tak, jak w każdej rodzinie, oszczędzamy w ten sposób na rozlewie krwi...

Owe reguły, czyli „procesy”, na co dzień zwykle dla nas tak dobrze znane, że aż niezauważalne, najlepiej stają się widoczne wówczas, gdy przestają działać. Wyobraźmy sobie sytuację – wracając do przykładu fabryki samochodów – w której podczas pracy nad skrzynią biegów odpowiedzialny za nią zespół wymienił całe okablowanie z miedzi na światłowód – prawdopodobnie słusznie zakładając, iż niezawodność nowego rozwiązania przełoży się na wzrost bezpieczeństwa pasażerów. Zespół zaczyna dostarczać nowe wersje skrzyni biegów, co jednak generuje problem i to nie tylko dla np. zespołu odpowiedzialnego za czujniki cofania, ale także dla kilku innych zespołów. Wszyscy dalej oczekują, iż informacje o tym, co robi skrzynia biegów pojawi się na ich miedzianych złączach.



To, czego zabrakło to stosowanie się do zasad współpracy pomiędzy zespołami (wspomnianych procesów), których nadzorem i wspólnym celem jest zbudowanie sprawnego samochodu. Zabrakło zarówno przestrzegania reguł koordynacji prac w projekcie (zespoły nie mogą autonomicznie wprowadzać nieuzgodnionych zmian w swoich komponentach) oraz odpowiedniej komunikacji (o nowym rozwiązaniu powinien być poinformowany zarazem cały zespół projektowy). Odpowiednikiem powyższej sytuacji w projekcie programistycznym jest wprowadzenie nieuzgodnionych zmian w kodzie źródłowym, które wprowadzają nowa funkcjonalność, jednak bez ko-

munikowania zmian wprowadzanych przez programistę, co skutkuje popuszczeniem innej funkcjonalności w programie.

Reguły współpracy mają na celu określenie sposobu, w jaki najbezpieczniej dla całego projektu powinno się wprowadzać zmiany.

Tworzenie, adaptowanie, wprowadzanie i koordynacja zasad współpracy oraz opieka nad definicjami wspomnianych konfiguracji to właśnie zarządzanie (ang. Management)ową współpracą.

### Kontrola

Każdy złożony projekt programistyczny polega na współpracy szeregu zespołów ludzi. Członkowie zespołów muszą koordynować powstające każdego dnia pomysły oraz ich implementację tak, aby projekt mógł się rozwijać bez przeszkód generowanych przez wzajemne oddziaływanie wprowadzanych zmian. Współpraca pomiędzy członkami zespołu projektowego musi skutkować dostarczaniem nowych wersji oprogramowania w założonym przez projekt czasie.

Potrzebujemy więc, aby jasne było, jak poszczególne części projektu wchodzą ze sobą w interakcje i jakie są pomiędzy nimi zależności. W dowolnym momencie naszej pracy musimy orientować się, nad jaką częścią programu / aplikacji pracujemy. Pozwala to spokojnie zająć się poprawianiem naszej funkcji wiedząc, iż poprawki wprowadzamy w tej wersji aplikacji, w której powinny się znaleźć i tylko w tej.

Kontrolowanie konfiguracji całego release'u aplikacji, nad którą pracujemy oraz wprowadzanych zmian w trakcie całego cyklu życia aplikacji, zapewnia, iż za każdym razem, gdy pojawią się jakieś zmiany będzie można zbudować kolejne jej wersje. Oznacza to nie tylko rejestrowanie kolejnych wersji kodu źródłowego, ale również wszystkich narzędzi, które zostały wykorzystane do jego produkcji. W ten sposób możliwe staje się na przykład wyłapanie problemów generowanych przez użycie odmiennych wersji bibliotek systemowych przez członków zespołu albo innej wersji kompilatora podczas kompilowania tych samych plików źródłowych.

Zapewnienie, aby procesy opisujące rozwój oprogramowania były przestrzegane przez wszystkie osoby w projekcie służy dobru całego projektu, jak i czyni pracę indywidualnego programisty łatwiejszą.

### Narzędzia

Do podstawowych narzędzi wykorzystywanych w projektach programistycznych należą wspomniane już systemy kontroli wersji, kompilatory, systemy budowania, środowiska do ciągłej integracji. Wykorzystywane są również własne narzędzia – pisane przez członków danego projektu przy pomocy różnego rodzaju języków skryptowych (np. bash, Python, Perl).

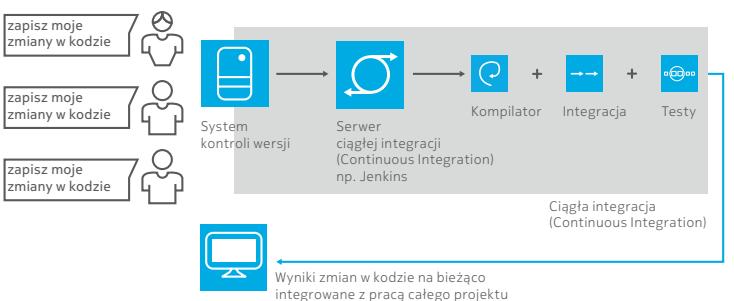
Systemy kontroli wersji to narzędzia rejestrujące kolejne wersje naszego kodu źródłowego. Za każdym razem, gdy jesteśmy zadowoleni z wyniku naszej pracy oprócz samego zapisania nowej zawartości

pliku z kodem źródłowym i tym samym nadpisaniu jego poprzedniej zawartości – zapisujemy w naszym systemie kontroli wersji kolejną jego wersję. Obsługa tego typu narzędzi już w pierwszym dniu korzystania staje się naturalna i wchodzi w nawyk. Za jednym razem otrzymujemy bezpieczne przechowywanie wszelkich zmian, które wprowadziliśmy do kodu źródłowego, jak i łatwe wyszukiwanie wszelkich wersji, które stworzyliśmy na przestrzeni życia projektu.

Kompilatory i narzędzia do budowania (make, ant, maven, ...) wplatane są zwykle w „systemy budowania” dopasowane do konkretnej specyfiki projektu. Ich zadaniem jest przygotowanie środowiska pracy programisty, pobranie odpowiednich wersji kodu źródłowego aplikacji oraz zapewnienie dostępności odpowiednich komponentów. System budowania wykonuje automatycznie wszystkie działania niezbędne do zbudowania całej aplikacji lub jej fragmentów po wprowadzeniu zmian w kodzie przez programistę.



Systemy ciągłej integracji posługują się systemami budowania, systemami kontroli wersji oraz wykorzystują przygotowane zestawy testów, tak by na bieżąco i w czasie rzeczywistym prezentować stan całego projektu, gdy zmiany do niego są wprowadzane niezależnie przez wielu programistów.



Dzięki tego typu systemom każdy z członków zespołu widzi, jaki efekt na cały projekt ma jego praca i czy wprowadzone zmiany nie generują problemów dla reszty projektu. Na poziomie dużego projektu ciągła integracja pozwala automatycznie śledzić integracje poszczególnych komponentów w jedną funkcjonalną aplikację.

### Podsumowanie

Zasady nadawania nazw elementom naszego projektu oraz zasady

ich kontrolowania (konfiguracje) odpowiadają nazewnictwu dróg, mostów, dzielnic i ułatwiają orientację, co do miejsca projektu, w którym aktualnie się znajdujemy.

Ustalanie reguł określających współpracę w projektach programistycznych pomaga nam wytyczać drogi, budować mosty, tunele i barierki zabezpieczające tak, by nawet najwięksi geniusze z nas, podczas szat tworzenia kolejnych funkcjonalności i podczas walki z kolejnymi hordami błędów w oprogramowaniu, mogli skupiać się wyłącznie na wyzwaniach programistycznych zamiast martwić o to, w jaki sposób ich praca wpływa na cały projekt. (Czemu aplikacja nie chce się zbudować, czemu się nie integruje w spójną i funkcjonalną całość, skąd wziąć odpowiednia wersje kodu źródłowego?...). I to właśnie aspekt współpracy z zespołami odpowiedzialnymi za tworzenie i testowanie oprogramowania czyni naszą pracę tak interesującą – można by powiedzieć, że na co dzień bierzemy udział w ciągłym wyścigu dwóch uzupełniających się dążeń. Z jednej strony programiści starający się za wszelką cenę dostarczyć rozwiązania problemów pozornie nie do rozwiązyania i dokonujący tego często wbrew prawom fizyki, z drugiej strony strażnicy zasad i procesów, „apostołowie” zdefiniowanych przez proces narzędzi – zespoły Software Configuration Management. Ów wyścig jest głównym motorem rozwoju dla całych projektów programistycznych i gwarantuje z jednej strony ciągłe poszukiwanie lepszych rozwiązań, będąc z drugiej strony pod kontrolą gwarantującą bezpieczeństwo projektowi.

#### Autor o swojej pracy

Pracuję w dziale MBB System Module (SCM). Opiekujemy się wszystkim, co jest potrzebne, by praca nad oprogramowaniem dla WCDMA czy LTE przebiegała jak najsprawniej. Systemy budowania, integracja oprogramowania, automatyzacja środowiska pracy programisty – czy to przy pomocy Git, SVN, Jenkins, Python, czy bash. Tworzymy środowiska do tworzenia oprogramowania dla projektów naprawdę dużej skali, gdzie tysiące programistów wspólnie każdego dnia pracuje nad kolejnymi generacjami najnowocześniejszych rozwiązań telekomunikacyjnych. Pracujemy zespołowo. Lubimy szukać dziury w całym i ulepszać to, co dobre.

**Andrzej Lipiński**  
R&D Manager  
SCM; MBB System Module

# Continuous Integration – integracja oprogramowania po „tuningu”

Marcin Gudajczyk

Senior Engineer, Software Configuration  
MBB System Module

**NOKIA**

Temat Continuous Integration jest zagadnieniem bardzo złożonym i musi być analizowany pod kątem konkretnych potrzeb, na rzecz których ma zostać wykorzystany. Można przedstawić elementy kluczowe całego procesu takie jak jego ogólne założenia, korzyści, jakie przynosi, koszty wprowadzenia i wiele aspektów technicznych. Chociaż da się wszystko zaprezentować z takiej właśnie perspektywy, podając zalety oraz powody, dla których stosuje się konkretne rozwiązania, można przeoczyć sedno całego podejścia do procesu wytwarzania oprogramowania. Niezauważalnie umkną detale, dla których taka właśnie "maszyneria" została stworzona i z powodzeniem wspiera prace nie tylko nad małymi, ale również naprawdę wielkimi korporacyjnymi projektami. Co więcej, stosowanie praktyk Continuous Integration jest obecnie w wielu firmach IT procesem fundamentalnym, bez którego sporo projektów może zostać skazanych na wydłużenie czasu ukończenia, jak i w najgorszym przypadku na niepowodzenie. Nie twierzę, że brak stosowania zasad Continuous Integration z góry przesądza o niepowodzeniu przedsięwzięcia, nie twierzę też, że takie podejście jest wymagane podczas prac związanych z tworzeniem kodu, ale będę chciał w tym artykule przedstawić w jak wielkim stopniu zastosowanie kilku prostych praktyk jest w stanie zdecydowanie ułatwić cały proces zarządzania, wytwarzania i testowania oprogramowania.

Czym więc właściwie jest Continuous Integration, które to wedle powyższego wstępma tak wiele zalet? Otóż jak sama nazwa wskazuje jest to proces wspomagający wytwarzanie oprogramowania opierający się o jego ciągłą integrację. Czym w takim razie jest owa ciągła integracja? Jest to praktyka definiująca pewne założenia dotyczące sposobu tworzenia i testowania oprogramowania. Jednym z aspektów jest zobowiązanie programistów, aby jak najczęściej i w możliwie jak najmniejszych porcjach dostarczali swoje zmiany kodu do wspólnego dla projektu repozytorium (jest to punk synchronizacji kodu). Może się wydawać, że takie założenie nakłada swego rodzaju więzy na programistów, którzy przez to odczuwają skrępowanie i presję bycia nadzorowanymi w swojej pracy. Nic bardziej mylnego! Oczywiście jest to wprowadzenie pewnego rodzaju zasad, ale ze względu na regularne dostarczanie zmian kodu możliwe jest szybkie rozpoznanie ewentualnie nowo powstałego błędu i jego sprawną eliminację. Programiści zauważając korzyść, jaka z tego płynie, chętnie akceptują główną regułę rzączącą procesem Continuous Integration. Wiodąca idea polega właśnie na nieustającej integracji kodu w celu jak najszybszego rozpoznania, zlokalizowania i usunięcia błędów. Proces Continuous Integration opiera się również na użyciu kilku narzędzi i mechanizmów, które przedstawię w dalszej części artykułu.

Znając już myśl przewodnią przeanalizujmy kolejne aspekty, które będą prowadziły do usprawnienia prac związanych z wytwarzaniem oprogramowania. Przyjrzyjmy się przykładowemu projektowi i pracy programistów zwracając uwagę na to, jakie błędy zostały popełnione podczas jego rozwoju i jak można było ich uniknąć.

Paczka znajomych ze studiów wpadła na pomysł stworzenia małej i prostej, ale zarazem bardzo przydatnej aplikacji mobilnej. Ponie-

waż spodobała się ona użytkownikom, jej twórcy zyskali niemałą popularność. Niesieni sukcesem postanowili zainwestować w rozwój swojego produktu. Stworzyli firmę i przystąpili do prac. Na początku bardzo szybko doszli do wniosku, że należy w jakiś sposób synchronizować dostarczenia zmian kodu, aby dzielić się efektami swojej pracy – założenie jak najbardziej szczytne. Postanowili stworzyć wspólny zasób sieciowy, do którego będą trafiały ich zmiany kodu. Każdy z nich rozwijał kod we własnym zakresie i przeprowadzał indywidualne testy swoich zmian. Takie było ustalenie – niech każdy z nich testuje własne zmiany, ponieważ jest za nie odpowiedzialny. Po pewnym czasie powstała kolejna wersja oprogramowania, którą można było wydać do użytku publicznego. Tym razem aplikacja była płatna, ale ponieważ pierwsza wersja odniosła spory sukces, jej druga odsłona nie miała problemów ze sprzedażą. Po raz kolejny posypały się pozytywne opinie oraz sugestie, w jaki sposób można rozwinąć aplikację w jej kolejnych odsłonach. W międzyczasie, prace rozwojowe zaczęły postępować wolniej, ale nie były niczym zablokowane.

Pewnego dnia, po wprowadzeniu do miejsca synchronizacji zmian przez jednego z programistów inny zauważył, że część jego kodu zniknęła. Okazało się, że osoba wprowadzająca własne zmiany przypadkowo usunęła fragment kodu swojego kolegi. Na szesnaste problem został zauważony w miarę szybko i brakująca część została odtworzona z kopii roboczej odpowiedniego developera. Któregoś dnia z kolei, po dostarczeniu przez jednego z programistów serii zmian, kod przestał się kompilować u pozostałe części grupy. Jak możliwe jest otrzymanie sytuacji, gdzie u jednego użytkownika kod się kompiluje, a u pozostałych nie? Po wspólnych analizach błędów, zespół zauważył, że część wprowadzonej zmiany wymaga zainstalowania nowszego kompilatora. Rzeczywiście odpowiedzialny za zmianę człowiek przypomniał sobie, że używał na własnej maszynie, na której pracował nowszej wersji kompilatora wprowadzającej możliwość wykorzystania niespotykanych wcześniej konstrukcji kodu. Po zainstalowaniu wymaganego oprogramowania u pozostałych programistów, problem zniknął. Podobne zdarzenia miały miejsce w przyszłości, ale drużyna wiedziona doświadczeniem wiedziała, że należy informować pozostałe osoby o nowych wersjach oprogramowania potrzebnych do zbudowania projektu. Na rynek została wypuszczona kolejna edycja aplikacji. Ponownie odniesiono sukces, ale tym razem, zaraz po jej pojawienniu się, wielu użytkowników znalazło kilka błędów, które zostały zgłoszone do zespołu. Nowe funkcjonalności były zachwycające, ale jakość oprogramowania zaczęła budzić już niestety wątpliwości. Programiści zabrali się za ich korekcję zastanawiając się równocześnie, dlaczego nie zauważali ich wcześniej. Wspólnie doszli do wniosku, że część kodu, nad którym pracowali była wykorzystywana przez funkcjonalności implementowane przez różne osoby. Nadpisywane w ten sposób sekcje kodu wprowadzały modyfikacje działania programu niewidoczne dla pozostałych jego twórców. Doszli do wniosku, że należy śledzić takie wspólne sektory kodu i powielać testy z nimi związane – urosła tym samym lista rzeczy, których należy pilnować. Nadszedł czas na korekcję błędów

jest rzeczą obowiązkową, ale przyjęte rozwiązanie nie było najlepszym pomysłem i zaowocowało kolejnymi problemami:

- znikanie fragmentów kodu – użytkownicy nadpisują czyjeś zmiany nie zawsze mając tego świadomość,
- niemożność ustalenia czasu wprowadzenia błędu do kodu – brak historii rozwoju kodu skutecznie uniemożliwia śledzenie rozwoju projektu,
- problemy z określeniem stanu projektu, z którego została wypuszczona wersja komercyjna – powoduje to problemy z dostarczeniami poprawek,
- spowolnienie procesu ustalenia, czy błędy w kodzie istniały we wcześniejszych wersjach oprogramowania,
- wybiorcze testowanie kodu poprzez pojedynczych programistów – każdy był odpowiedzialny wyłącznie za swoją część kodu,
- powszechnie stosowana praktyka pracy wielu osób nad wspólną częścią kodu – skutkowało to wprowadzaniem subtelnych zmian niewidocznych dla części osób pracujących na wspólnych sekcjach, przez co indywidualne testy mogły zanieść,
- niebudujący się kod przy wprowadzeniu nowszego kompilatora – jeżeli wprowadzano nowszą wersję oprogramowania należało o tym powiadomić cały zespół,
- aplikacja mobilna została wypuszczona z błędami, które nie były wykryte podczas prac nad nią – powoduje to niepożądane działanie oprogramowania i oczywiście niezadowolenie klientów,
- długi okres wdrożeniowy nowych osób – przyswojenie wielu niełatwych procesów utartych przez doświadczonych członków zespołu było czasochłonne do opanowania,
- sprawdzenie prac do utrzymania kodu i poprawiania błędów zamiast rozwoju nowych funkcjonalności.

W najczarniejszym scenariuszu taki projekt w dalszych fazach rozwoju mógłby zostać skazany na znaczające spowolnienie jakiegokolwiek postępu. Nie rozwiązywanie powstałych problemów, tylko obchodzenie ich poprzez wprowadzanie nadmiarowych procesów, o których ludzie musieliby pamiętać skutecznie opóźniałoby rozwijanie nowych funkcjonalności, jak również generowało podobne do omówionych problemów. Większość nakładu pracy została w ostatczności przerzucona na utrzymanie kodu i jego poprawę w celu usunięcia błędów ( dodatkowo należało poświęcić czas na śledzenie, w jakich miejscach zostały one wprowadzone). Rozwój oprogramowania byłby znacznie spowolniony. Powstała pętla zaciskałaby się bezlitośnie na projekcie, stopniowo zmniejszając szanse na jego racjonalną rozbudowę. Nawet najlepsi programiści swoimi zdolnościami nie byliby w stanie uratować sytuacji, bo tak naprawdę nie zabrakło tutaj zdolności i talentu, ale procesu zarządzania wytwarzaniem oprogramowania. To jego brak i niewłaściwe wybory doprowadziły do takiego stanu. Należy zaznaczyć, że zanieść mogła również nieobecność zastosowania praktyk, modeli i zasad opisanych chociażby w syllabusie ISTQB: [www.istqb.org](http://www.istqb.org).

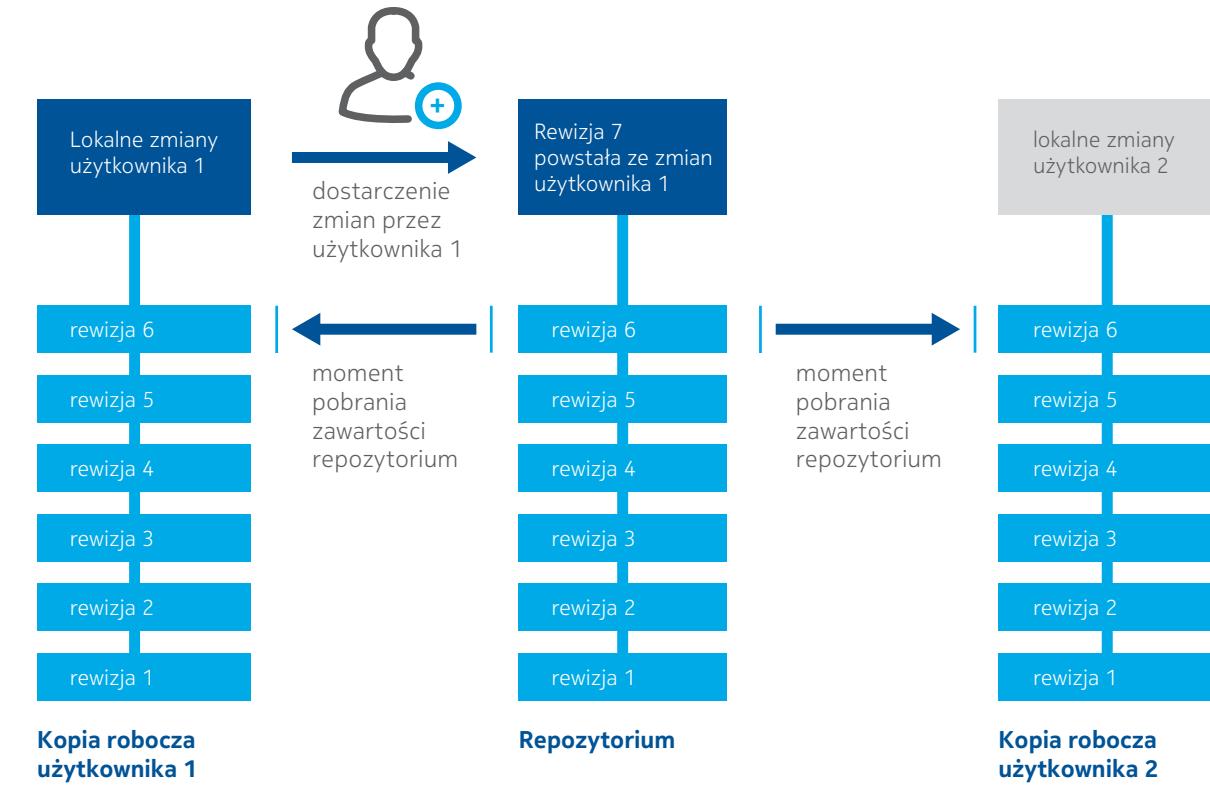
Na tym zakończymy naszą opowieść i przeanalizujmy, jakie błędy zostały popełnione. Są to elementy, które bardzo łatwo jest zignorować, a w konsekwencji prowadzą do nieszczęśliwych skutków. Odpowiednie dobranie procesów oraz narzędzi pozwala na unikanie komplikacji oraz na bezproblemową ewolucję kodu. Wymieńmy momenty, w których zaistniały problemy i zastanówmy się, jakie miały konsekwencje:

- synchronizacja zmian kodu poprzez zasób sieciowy – oczywiście zgadzam się, że proces udostępnienia kodu grupie programistów

jako swoistą "bazę" danych. Jest to tak naprawdę narzędzie, które gromadzi stan projektu dla każdej zmiany w nim zaistniałej oraz informacje na temat każdej ze zmian. Repozytorium jest punktem centralnym, do którego trafiają wszystkie zmiany kodu wprowadzone przez programistów i tam pozostają (część VCS, np. Git, pozwala na zmianę historii, ale takie praktyki są niewskazane). Każda z zaistniałych modyfikacji jest zawsze dostępna, nawet, jeżeli zostanie później usunięta. Historia będzie zawierała informację o wprowadzeniu poprawki oraz o jej usunięciu. Użytkownik domyślnie zawsze widzi najnowszą wersję danych, które repozytorium dostarcza, ale w każdej chwili może odwołać się do kolejnego z wcześniejszych. Do każdej zmiany jest przypisany unikalny i niezmieniony identyfikator nazwany rewizją. To właśnie za pomocą rewizji i powiązań między nimi (każda powinna wskazywać, na co najmniej jedną poprzedzającą wersję) możliwe jest prześledzenia historii zmian wprowadzonych do projektu. Pobierając zawartość repozytorium, użytkownik tworzy lokalnie kopię roboczą projektu, mając od razu dostęp do całego kodu źródłowego. Implementacje nowych zmian przebiegają następnie na

kopii roboczej (zawierającej metadane o historii), z której to później zostają przesłane do repozytorium i opatrzone unikalną rewizją. Każda nowo wprowadzona zmiana od razu jest dostępna dla wszystkich pozostałych użytkowników. Kolejną bezcenną funkcjonalnością VCS jest zabezpieczenie danych przed ewentualnością nadpisania fragmentu kodu, nad którymi pracuje kilku użytkowników. Jeżeli jeden z nich wprowadzi zmiany do wspólnego fragmentu, jako pierwszy, to każdy kolejny podczas próby dostarczenia własnych modyfikacji jawnie dostanie informację o tym, w którym miejscu wystąpił konflikt i które fragmenty kodu go powodują. Taki użytkownik jest świadomy zaszłych zmian i jest w stanie odpowiednio dopasować własne modyfikacje, aby uzyskać satysfakcyjny efekt, nie usuwając przy tym przypadkowych fragmentów kodu. Rewizjonowanie pozwala również na odnalezienie miejsca wystąpienia błędu i na precyzyjne określenie zmiany, która go wprowadziła. Dzięki czemu można stwierdzić, w których wersjach oprogramowania jest on wiadczny i odnaleźć autora modyfikacji.

**1 Rysunek** Uproszczony zarys wymiany danych poprzez repozytorium. Użytkownik 2 będzie musiał odświeżyć swoją kopię roboczą przed dostarczeniem zmian, aby sprawdzić, czy nie kolidują one za zmianami użytkownika 1



Aby uniknąć komplikacji związanych z doborem wersji oprogramowania potrzebnego do zbudowania aplikacji należy zapewnić spójne środowisko produkcyjne. Takie podejście gwarantuje, że każdy użytkownik będzie miał dostęp do wspólnego zestawu kompilatorów, linkerów, bibliotek (dostarczanych czy to poprzez system operacyjny, czy też wspólną platformę systemową), interpreterów języków skryptowych, powłok etc. Wiele rozwiązań można oprzeć o założenia "crossplatform compatibility", ale przy takim podejściu należy pamiętać, że nie zawsze da się utrzymać spójność kodu (poza spójnością środowiska oczywiście), więc przy tego typu projektach należy dostarczać zestawy środowisk odpowiednich dla wspieranych systemów. Ten cel można osiągnąć na kilka sposobów. Wykorzystanie wspólnego zasobu sieciowego, gdzie takie oprogramowanie będzie się znajdowało jest już nienajgorszym pomysłem. Taki zasób udostępniony dla maszyn będzie gwarantował użycie jednakowych zestawów narzędzi przez wszystkich programistów. Dobrą praktyką jest zdefiniowanie osobnego repozytorium, gdzie taki zbiór będzie przechowywany. Poza wspólnym miejscem dodatkowa korzyść płynie z wersjonowania środowiska w stosunku do wersji głównego projektu. Przy ustaleniu odpowiednich zależności, wprowadzona zostanie możliwość odtworzenia budowania każdej wersji produktu z dokładnie takim samym zbiorem narzędzi, jak to miało miejsce w odpowiednim etapie projektu.

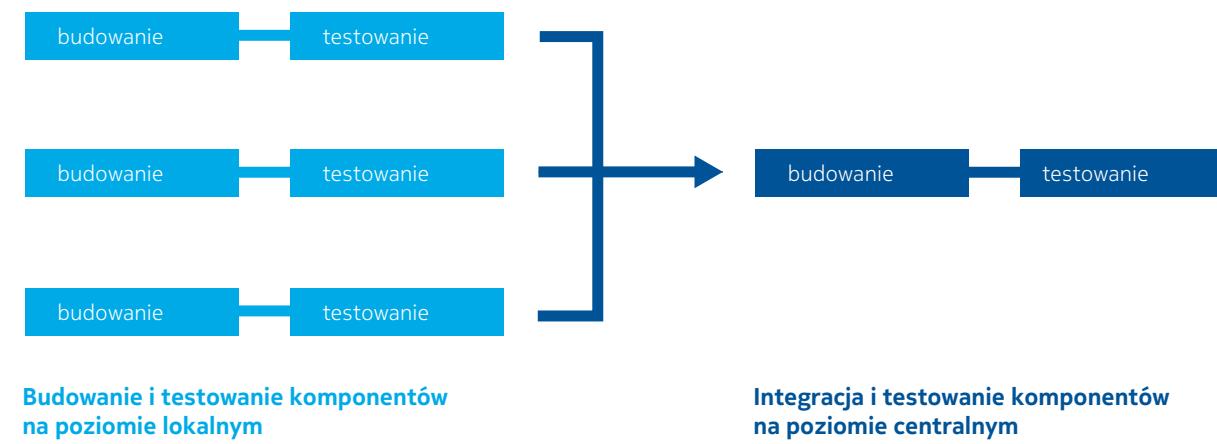
Kolejnym elementem, jaki należy dołączyć do procesu Continuous Integration jest zdefiniowanie systemu budowania. Tak naprawdę nie ma żadnego spreparowanego systemu budowania. Ten element Continuous Integration jest właściwie jedną z najbardziej elastycznych części całego procesu i definiuje się go w zależności od potrzeb samego projektu jak też jego twórców. System budowania w dużej mierze opiera się o narzędzia dedykowane do zdefiniowania zasad wytwarzania z kodu źródłowego plików wykonawczych, które są często gotowym produktem. Do tych celów używa się najczęściej Makefile, CMake, Ant, WAF oraz ich pochodne. Narzędzia te mają na celu automatyzację procesu samego budowania oprogramowania. Użytkownik zamiast po kolej wywoływać wszystkie komendy potrzebne do komplikacji, linkowania i innych procesów, może zamiast tego, po zdefiniowaniu odpowiednich recept zażądać zbudowania jednej z nich w celu uzyskania pożądanego wyniku. To system budowania ma być miejscem, gdzie po wywołaniu polecenia utworzenia plików wynikowych, zaraz na samym początku powinno być ustanowione spójne środowisko produkcyjne. To ten element Continuous Integration jest w stanie zminimalizować w dużym stopniu czas oczekiwania na wyniki dla każdego z programistów poprzez zapewnienie, co najmniej redukcji kroków koniecznych do uzyskania plików wynikowych. Wspomniane narzędzia posługują się mechanizmami śledzącymi zależności zdefiniowane dla procesu budowania, które pozwalają uniknąć ponownego tworzenia plików wynikowych, dla których treść kodu źródłowego się nie zmienia. Uzyskuje się przez to znaczną redukcję czasu budowania oraz eliminację redundantnych kroków. Dzięki temu skracą się czas potrzebny na wygenerowanie produktu. To również system budowania zapewnia możliwość

wywołania zupełnie dowolnych narzędzi wymaganych np. w jednym z kroków procesu wytworzenia oprogramowania. W końcu to w systemie budowania powinny znajdować się definicje, gdzie wszystkie wyniki produkcji powinny być umieszczone. W dużej mierze jest to nic innego jak zarządzanie kopią roboczą. Dodatkowo można zaimplementować również kroki dokonujące sprawdzenia stanu kopii roboczej oraz środowiska przed przystąpieniem do produkcji samego oprogramowania, jak też po jego zakończeniu w celu upewnienia się, że wszystkie wymagane elementy zostały utworzone.

Przedostatnim z najważniejszych elementów jest definicja i utworzenie wspólnej puli testów, która ma być dostępna dla każdego z twórców projektu. Istnieje wiele modeli testowania oprogramowania w zależności od tego, na którym poziomie się ono odbywa. Na chwilę obecną skupmy się na testach jednostkowych (Unit Tests), które zdefiniowane są na najniższym z nich. Poprawne wykonanie się takich testów powinno być kryterium wprowadzenia nowych zmian do repozytorium. Skupią się one na pokryciu największej ilości kodu i muszą sprawdzać wszystkie możliwe do przetestowania funkcjonalności aplikacji (zaimplementowane metody) wymagane do jej poprawnego działania. Takie testy definiują zestaw danych podawanych na wejściu oraz zestaw danych oczekiwanych na wyjściu (po przetworzeniu ich przez odpowiednie metody) w celu ustalenia, czy operacje wykonywane są poprawnie. Występuje wiele frameworków pozwalających utworzyć i utrzymywać pełną gamę testów, które mogą być zależne od samego języka programowania wykorzystawanego w projekcie np. Python Unit Test, GMock, Robot Framework, etc. Zdefiniowany zestaw powinien być uruchamiany zawsze po lokalnym ukończeniu prac implementacyjnych w celu upewnienia się, że nie wprowadzają one błędów w działaniu oprogramowania. Co więcej zestaw ten powinien być poszerzany w miarę dodawania nowych funkcjonalności tak, aby pokryć nowe obszary kodu. Wykorzystywane są również narzędzia odpowiadające za statyczną analizę kodu takie jak Pylint, Sonar, Klocwork, czy też Valgrind. Umożliwiają sprawdzenie źródeł pod kątem wszystkich możliwych błędów składowych, czy też przekroczenia zakresu adresowania, etc. Do tego dochodzą testy funkcjonalne działające na zasadzie czarnej skrzynki (black box). Uruchamiane są nie na kodzie, ale już na działającym pliku wykonawczym sprawdzając, czy zachowuje się on zgodnie z założeniami (podaje na wyjściu oczekiwane dane przy zleconych działaniach). Testy takie nie posiadają informacji na temat tego, co dzieje się wewnętrz kodu.

Bardzo często duże projekty podzielone są na komponenty. Każdy z nich jest odpowiedzialny za ścisłe określone funkcje i posiada własne testy tak jak to opisano powyżej. Takie komponenty tworzą wspólnie produkt, komunikując się ze sobą w pewien sposób. W takich przypadkach należy przeprowadzać testy integracyjne. Mają one na celu ustalenie, czy wymiana informacji pomiędzy odrębnymi częściami produktu za pomocą zdefiniowanych interfejsów przebiega poprawnie. Działają one również na zasadzie czarnej skrzynki.

2 Rysunek Uproszczony schemat procesu testowania. Osobne testy uruchamiane są lokalnie na poziomie komponentów, natomiast testy integracyjne na poziomie centralnym



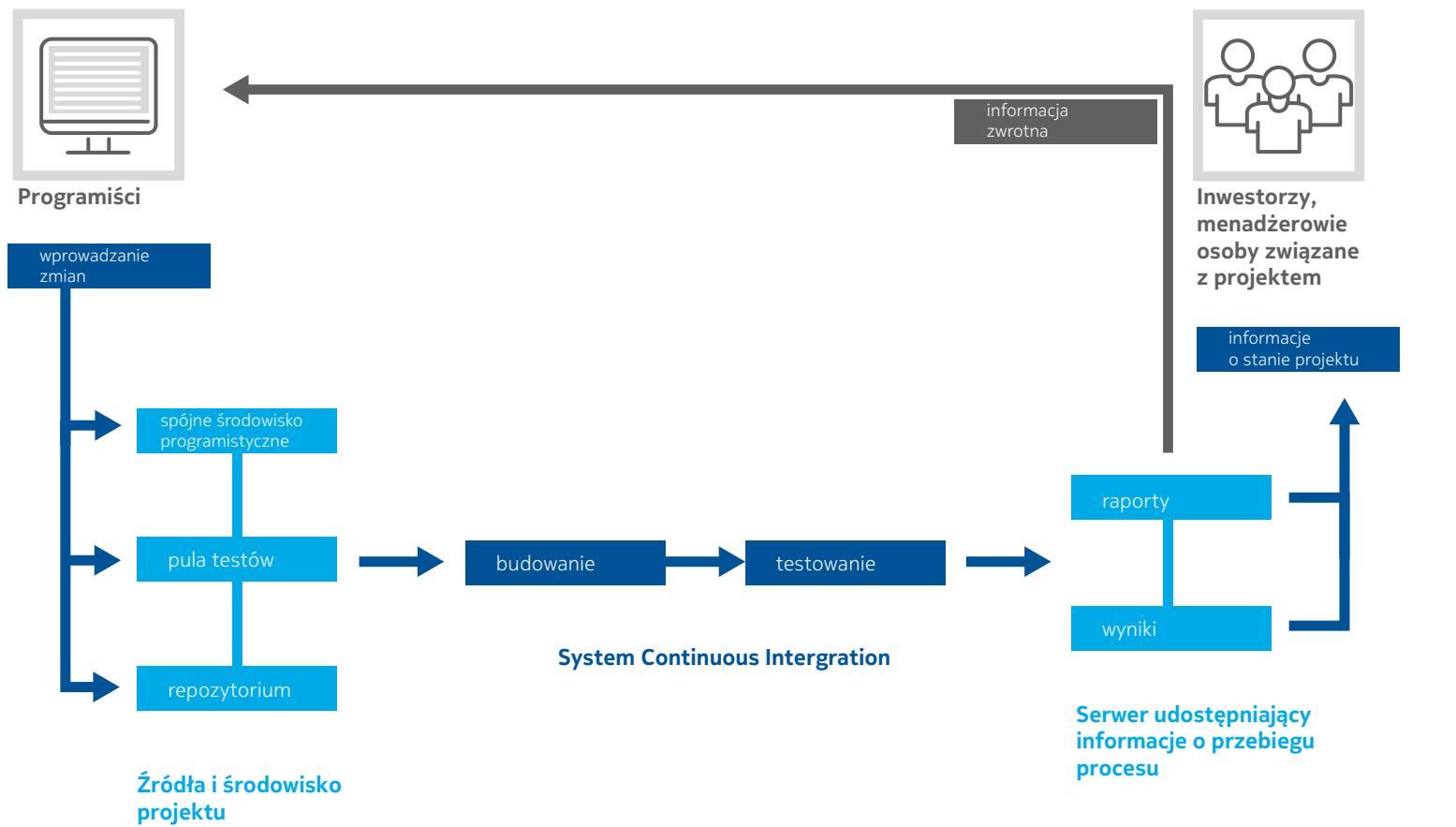
Dopiero w tym momencie, posiadając opisany już zestaw funkcjonalności, można przystąpić do złączenia wszystkich elementów w jedną całość za pomocą narzędzia dedykowanego do Continuous Integration. W praktyce wykorzystuje się gotowe rozwiązania takie jak np. Jenkins, Hudson, Team City, Buildbot, Bamboo, etc. Systemy ciągłej integracji pokrótce można opisać, jako centralny punkt do zarządzania maszynami produkcyjnymi, na których wykonywany jest zautomatyzowany proces budowania i testowania oprogramowania oraz prezentacji wyników całego procesu. Połączenie wszystkich elementów za pomocą narzędzia Continuous Integration to finalny krok, który w elastyczny sposób pozwala zaprezentować jawnie aktualny stan projektu. Przebieg wszystkich kroków oraz ich wyniki muszą być widoczne dla wszystkich osób biorących udział w pracach. Pojedynczy programista dokonując zmian w kodzie buduje go i testuje, po czym wprowadza je do repozytorium i na tym teoretycznie jego zadanie się kończy. Systemy Continuous Integration wykonują zdecydowanie więcej kroków.

Po pierwsze nadzorowane jest wspomniane właśnie repozytorium pod kątem zaistnienia w nim jakichkolwiek zmian. Wyzwalaczem powinna być również każda inna zmiana ze zdefiniowanych elementów Continuous Integration, nie tylko modyfikacja źródeł projektu, a na przykład również zmiana puli testów. Jeżeli modyfikacja zostanie zauważona rozpoczyna się proces identyfikacji zmiany w celu zdefiniowania, co było wyzwalaczem wykonania przebiegu. Pozwala to na późniejsze określenie, czy konkretna zmiana wprowadzona do projektu przeszła wszystkie kroki produkcji i nie spowodowała żadnych problemów. Stworzenie takiego punktu odniesienia pozwala w łatwy i szybki sposób na zaobserwowanie i odnalezienie powodów, dla których proces produkcji mógł się nie powieść. System Continuous Integration rezerwuje następnie instancję z puli maszyn o spójnym

środowisku, na której wykonane zostaną operacje produkcji. Pobrana zostaje kopia repozytorium z kodem oraz konfigurowane jest środowisko, następnie rozpoczyna się proces produkcji oprogramowania zdefiniowany w Systemie Budowania.

Po zakończeniu tego etapu przychodzi kolejna uruchomienie testów z ogólnej puli na rzecz wygenerowanych plików wynikowych w kopii roboczej. Jeżeli wszystkie kroki zakończą się sukcesem, zbielane zostają wyniki oraz raporty z wykonania się wszystkich etapów procesu. Może również zostać wygenerowane dowolnie zdefiniowane powiadomienie o osiągniętym sukcesie (np. poprzez maila). Zebrane wyniki muszą być umieszczone w ogólnie dostępnym miejscu, ponieważ dobrze zdefiniowane elementy Continuous Integration są w stanie zapewnić wytworzenie oprogramowania z każdej rewizji w dokładne taki sam sposób. Wyniki całego procesu nie muszą być rewizjonowane i można przetrzymywać tylko kilka najnowszych z nich w celu zaoszczędzenia przestrzeni dyskowej. Dla każdego z przeprowadzonych w ten sposób przebiegów powinny zostać wykonane dodatkowe testy (czasochłonne i uruchamiane najczęściej nocą), które są również istotne, dla określenia jakości oprogramowania. Można np. przeprowadzać weryfikację pod kątem wycieków pamięci (memory leak). Należy zadbać, aby podstawowe kroki, będące kryterium stabilności oprogramowania i biorące udział w jego wytwarzaniu były możliwe najkrótsze. Nie można tego robić jednak kosztem jakości – musi zaistnieć kompromis między czasem użyskania wyników a zapewnieniem możliwie najwyższej stabilności produktu. Jeżeli którykolwiek z kroków się nie powiedzie, wszystkie możliwe do zebrania elementy powinny być zgromadzone, a raporty wygenerowane. Posłużą one do odnalezienia przyczyny powstałego problemu i jego usunięcia. W takiej sytuacji również powiadomienie o niepowodzeniu operacji powinno zostać wysłane. Cały proces (po-

3 Rysunek Przykładowy schemat przebiegu procesu Continuous Integration



czawszy od śledzenia zmian w repozytorium aż do wykonania testów powinien być zdefiniowany nie tylko dla pojedynczych komponentów (jeżeli istnieje taki podział), ale również na rzecz ich integracji. Jeżeli tak jak wcześniej zostało wspomniane, programiści będą dostarczać do repozytorium możliwe jak najczęściej i jak najmniejscze porcje zmian, zdefiniowany w taki sposób proces pełnoprawnie będzie zasługiwał na miano Systemu Continuous Integration. Odpowiednim do zaprezentowania przykładem Systemów Ciągłej Integracji jest Jenkins Apache Software Foundation (<https://builds.apache.org/>).

Podsumowując można śmiało stwierdzić, że projekty wykorzystujące powyższe zasady, drastycznie zwiększą swoją szansę na powodzenie. Dzięki zastosowaniu systemów kontroli wersji zyskuje się precyzyjną możliwość śledzenia wszystkich zaistniałych w repozytorium zmian. Co więcej, to właśnie użycie repozytorium pomaga w synchronizacji prac pomiędzy programistami zapobiegając sytuacjom, w których jeden z nich nieświadomie usuwa kod kogoś inne-

opisanych procesów pomaga w szybkim zrozumieniu praktyk panujących w projekcie. Ponieważ jakiekolwiek zaistniałe błędy są eliminowane w najkrótszym możliwym czasie, takie podejście gwarantuje, że programista może pracować na stabilnym i zawsze działającym kodzie. Pozwala to uniknąć sytuacji, gdzie należy się zastanawiać, czy napotkany podczas implementacji problem już istniał, czy może został utworzony w kopii roboczej.

Zastosowanie całego opisanego procesu gwarantuje utrzymanie kontroli nad wytwarzaniem i testowaniem oprogramowania. Podejście Continuous Integration nie jest lekarstwem idealnym. Nie eliminuje wszystkich błędów, ale za to uwypukla je, sprawiając, że kują w oczy i czynią je łatwiejszymi do zlokalizowania oraz usunięcia.

#### Autor o swojej pracy

Pracuję w dziale MBB System Module (Software Configuration Management), gdzie na co dzień zajmujemy się automatyzacją produkcji oprogramowania. Wykorzystując systemy Linux oraz narzędzia takie jak systemy kontroli wersji, języki skryptowe, czy systemy ciągłej integracji, kreujemy środowiska wykorzystywane do wytwarzania i testowania oprogramowania LTE oraz WCDMA. Nasze rozwiązania gwarantują sprawne działanie procesów mających na celu przyspieszenie i ułatwienie prac rozwojowych dla dużych projektów. Dynamiczne i ciągle zmieniające się środowisko zapewnia wiele możliwości rozwoju.

#### Marcin Gudajczyk

Senior Engineer, Software Configuration  
MBB System Module

# Technologie radiowe

4.1

**Artur Orzechowski i Marcin Miernik**  
Parametry sygnałów radiowych

116

4.2

**Marcin Domański**  
Pomiar parametrów radiowych stacji bazowych LTE (eNodeB) przy użyciu wektorowego generatora sygnałowego i analizatora widma

122

4.3

**Łukasz Gostkowski**  
Sieć mobilna – cybernetyczne pole walki

128

4.4

**Piotr Małyński**  
Testowanie systemu LTE w środowisku end-to-end

136

4.5

**Agnieszka Szufarska**  
Czego spodziewać się po systemie 5G?

142

# Parametry sygnałów radiowych

Artur Orzechowski  
Integration Project Leader  
MBB RF RFSW

Marcin Miernik  
R&D Manager, RF Projects  
MBB RF RFSW

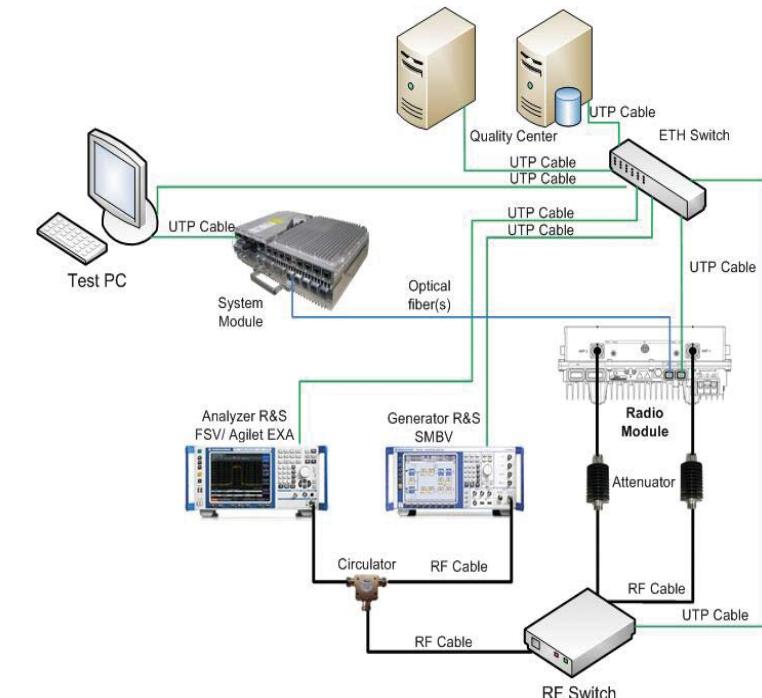
**NOKIA**

Chcąc spełnić rosnące oczekiwania klientów dotyczące wysokiej jakości oferowanych usług sieci mobilnych, konieczna jest optymalizacja parametrów radiowych. Dlaczego aspekt ten jest tak ważny dla klienta? Otóż, jak wiadomo zasoby radiowe są bardzo ograniczone. Zatem aby móc wprowadzać na rynek międzynarodowy nowoczesne rozwiązania telekomunikacyjne zgodne z obowiązującymi wymogami prawnymi, dostawcy rozwiązań telekomunikacyjnych, a wśród nich firma Nokia, nieustannie koncentrują swoje działania na optymalizacji sygnałów radiowych, podnoszeniu jakości swoich produktów oraz tworzeniu elastycznych rozwiązań.

Niezmiernie ważne jest, aby nowo zbudowana stacja bazowa nie zakłócała pracy istniejących już systemów telekomunikacyjnych, po przejrzu np. przedostawanie się produktów intermodulacji do sąsiadujących kanałów, oraz aby generowała stabilną moc wyjściową.

Istnieje wiele norm dotyczących wartości parametrów sygnałów radiowych, zostały one opracowane przez niezależne instytuty standardyzujące takie jak 3GPP, ITU oaz ETSI. Organizacje te sprawują pieczę nad wieloma aspektami wymogów dotyczących sygnałów radiowych w stacjach bazowych. Niektóre z tych wytycznych przedstawimy w niniejszym artykule, wraz z ogólnym zarysem metodologii testowania sygnałów radiowych stosowanej w firmie Nokia.

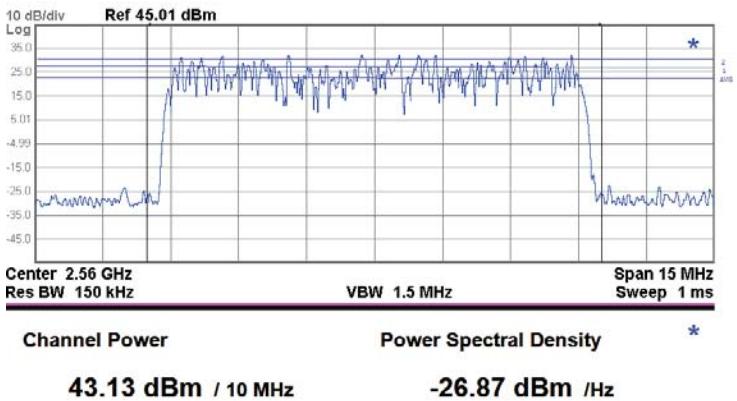
**1 Rysunek** Środowisko testowe do pomiarów DL/UL stosowane w laboratoriach Nokia



Generalnie, większość pomiarów radiowych przeprowadzanych w laboratoriach Nokia wymaga użycia analizatora widma – do pomiarów DL (ang. downlink, łącze w dół), oraz generatora sygnałów – do pomiarów UL (ang. uplink, łącze w góre) (**Rysunek 1**). Podczas testów transmisji radiowej używane są dane testowe w postaci tzw. modeli testowych, zgodnie ze specyfikacją 3GPP TS 125 141 oraz 3GPP TS 136 141. Specyfikacja 3GPP wyraźnie określa, który model testowy powinien zostać zastosowany do pomiaru każdego z następujących parametrów: Output Power, ACLR, EVM, Frequency error oraz CCDF.

**Output Power** – parametr ten określa rzeczywistą ilość mocy radio-modułu w zajmowanej szerokości pasma (**Rysunek 2**). Maksymalna moc wyjściowa stacji bazowej to poziom średniej mocy, mierzonej na złączu antenowym, podczas trwania transmisji. W normalnych warunkach moduły montowane wewnętrz budynków wspierają moce o wartościach do 37 dBm, a moduły zewnętrzne do 49 dBm. W zależności od wymagań klienta oraz zagęszczenia komórek w sieci, moc wyjściowa stacji bazowej może być regulowana.

**2 Rysunek** Pomiar mocy wyjściowej w laboratoriach Nokia

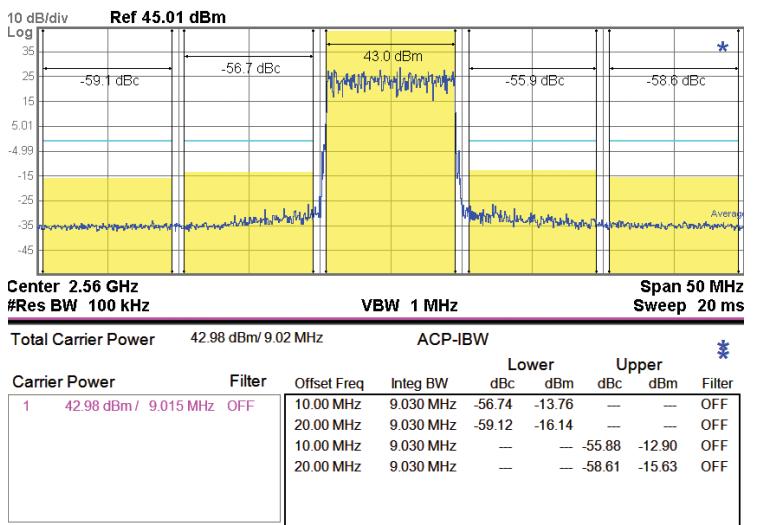


**ACLR** (Adjacent Channel Leakage Ratio) – ten parametr określa ilość mocy, która „wycieka” do sąsiednich kanałów częstotliwościowych. Moc zakłócenia wpływa na wydajność systemu, ponieważ zakłóca transmisje w kanałach sąsiednich. Parametr ten obrazuje również sprawność algorytmów linearyzacji charakterystyki wzmacniacza mocy. Wartość ACLR powyżej progu (biorąc pod uwagę bezwzględna wartość mocy) oznacza, że predystorsja adaptacyjna nie działa prawidłowo lub sygnał wejściowy jest zniekształcony. Specyfikacja 3GPP TS 36.141 określa minimalne wymagania wartości parametru ACLR dla nośnych E-UTRA (LTE) oraz UTRA (WCDMA) (**Rysunek 3**). Przykład pomiarów przedstawia **Rysunek 4**.

**3 Tabela** Wymogi dot. parametru ACLR zgodnie z normą 3GPP TS 36.141

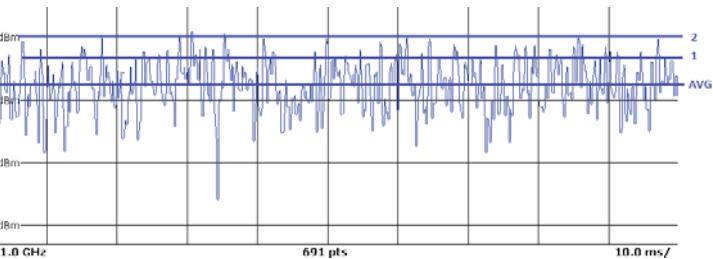
E-UTRA transmitted signal channel bandwidth [MHz]	BS adjacent channel center frequency offset below the first or above the last carrier centre frequency transmitted	Assumed adjacent channel carrier	Filter on the adjacent channel frequency and corresponding filter bandwidth	ACLR limit
1.4, 3.0, 5, 10, 15, 20	BW <sub>Channel</sub>	E-UTRA of same BW	Square (BW <sub>Config</sub> )	44.2dB
	2 x BW <sub>Channel</sub>	E-UTRA of same BW	Square (BW <sub>Config</sub> )	44.2dB
	BW <sub>Channel</sub> /2+2.5 MHz	3.84 Mcps UTRA	RRC (3.84 Mcps)	44.2dB
	BW <sub>Channel</sub> /2+7.5 MHz	3.84 Mcps UTRA	RRC (3.84 Mcps)	44.2dB

**4 Rysunek** Pomiar parametru ACLR



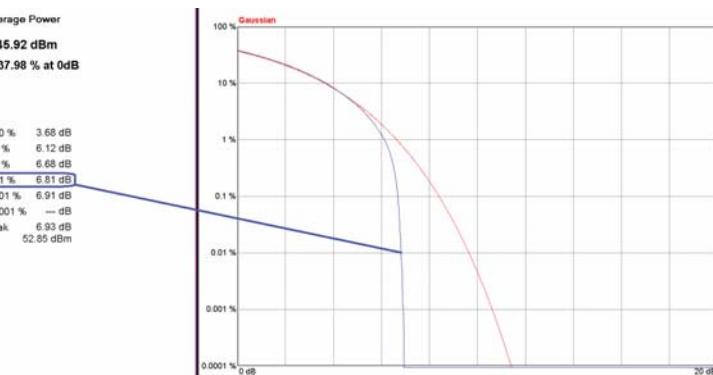
**CCDF** (Complementary Cumulative Distribution Function) – określa statystycznie jak często sygnał osiąga lub przekracza pewien poziom mocy. Z matematycznego punktu widzenia, pomiar CCDF jest dystrybuantą rozkładu punktów pomiaru poziomu mocy. Innymi słowy, funkcja CCDF pokazuje przez jaki okres czasu sygnał znajduje się na określonym poziomie mocy lub go przekracza (wykres przedstawiający poziom mocy w funkcji czasu). **Rysunek 5** przedstawia wykres, na którym zaznaczony jest średni poziom mocy oraz dwie dodatkowe wartości poziomu mocy. Widać, przez jaki okres czasu sygnał znajduje się powyżej określonych poziomów mocy. W praktyce, jeżeli poziom sygnału przez dłuższy czas przekracza średni poziom mocy, może to mieć negatywny wpływ na żywotność tranzystorów wzmacniacza mocy.

**5 Rysunek** Sygnał radiowy w dziedzinie czasu



Z powyższego wykresu trudno jest określić wartości CCDF, a więc w praktyce bardziej użyteczna jest charakterystyka opisująca zależność pomiędzy mocą średnią oraz czasem, kiedy sygnał przekracza średni poziom mocy (CCDF wyrażany jest w procentach).

**6 Rysunek** Krzywa CCDF

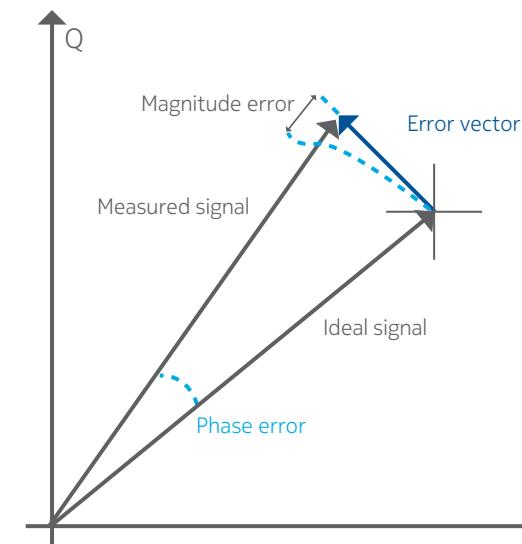


Na przykład, przy wartości  $t = 0,01\%$  wskaźnik stosunku mocy szczytowej do średniej wynosi 6,81dB. Oznacza to, że moc sygnału

przekracza średni poziom mocy o co najmniej 6,81dB przez 0,01% czasu. Powyższa analiza pokazuje praktyczne podejście do pomiarów CCDF oraz interpretację wyników. Natomiast przyczyną zbyt wysokich wartości CCDF może być niska skuteczność algorytmów linearyzacji charakterystyki wzmacniacza.

**EVM** (Error Vector Magnitude) jest jednym z parametrów opisujących jakość sygnału. Zgodnie ze specyfikacją TS 134 121-1 wydaną przez ETSI, wskaźnik EVM mierzy różnicę pomiędzy sygnałem rzeczywistym a sygnałem oczekiwany.

**7 Rysunek** EVM



Innymi słowy, EVM opisuje ilość zniekształceń wprowadzanych przez badany system czy układ do wejściowego sygnału (zwykle sygnału wzorcowego), które możemy wyrazić dwojako – procentowo:

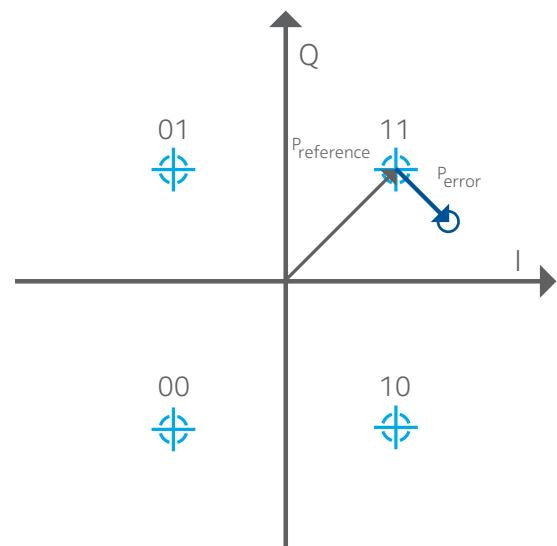
$$EVM(\%) = \sqrt{\frac{P_{\text{error}}}{P_{\text{reference}}}} \times 100$$

lub w dB:

$$EVM(\text{dB}) = 10 \log_{10} \left( \frac{P_{\text{error}}}{P_{\text{reference}}} \right)$$

EVM wyraża stosunek średniej wartości błędu wektora mocy ( $P_{\text{error}}$ ) do średniej wartości mocy odniesienia ( $P_{\text{ref}}$ ), wyrażany w decybelach.

**8 Rysunek** Ilustracja obrazująca średnią wartość wektora mocy odniesienia i wektora błędu mocy



W praktyce, do pomiarów parametru EVM używa się danych wzorcowych. Wzorce te, określone przez 3GPP, nazywane są modelami testowymi (ang. Test Model). Odpowiedni model testowy w postaci danych I/Q wprowadzany jest na wejście testowanego radio modułu. Sygnał wygenerowany na wyjściu mierzony jest przy pomocy analizatora, który zawiera dane analogiczne do danych pochodzących z modelu testowego. Urządzenie dokonuje porównania sygnału wzorcowego (opartego na znanym modelu testowym) z sygnałem odbieranym z radio modułu i na tej podstawie wylicza EVM.

**Frequency Error** – ten parametr określa, czy nadajnik lub/i odbiornik pracują na właściwej częstotliwości. Błąd częstotliwości jest zwykle wyrażany w jednostkach ppm (parts per million) bądź ppb (parts per billion), np. 10ppb dla częstotliwości 1,8GHz oznacza błąd częstotliwości +/- 18Hz. Zbyt wysoka wartość błędu częstotliwości może prowadzić do problemów z przenoszeniem połączeń (ang. handover) oraz do pogorszenia przepustowości komórki. Wytyczne dotyczące pomiarów błędu częstotliwości są określone w ustępie 6.5.1 specyfikacji 3GPP TS 36.101.

Podsumowując, dostawcy systemów telekomunikacyjnych muszą spełniać wymogi dotyczące jakości sygnałów radiowych, zgodnie z wytycznymi organizacji standaryzujących takich jak 3GPP, ITU i ETSI. Wszystkie przedstawione powyżej parametry są poddawane rygorystycznej weryfikacji w laboratoriach Nokia, zanim produkt zostanie przekazany w ręce odbiorcy. Oczywiście istnieje szereg innych wskaźników opisujących jakość sygnałów, jak i wewnętrznych wymagań firmy Nokia, których nie przedstawiono w tym opracowaniu, a które również mają wpływ na poziom jakości naszych produktów.

**Autor o swojej pracy**

Jestem odpowiedzialny za weryfikacje oprogramowania na radiomodulach pracujących w trybie TDD. Moimi codziennymi obowiązkami jest dbanie o środowisko automatyczne, delegowanie obowiązków, planowanie testów, weryfikacja wyników pomiarów radiowych.

**Artur Orzechowski**

Integration Project Leader  
MBB RF RFSW

**Autor o swojej pracy**

Pracuję dla działu RFSW, który zajmuje się tworzeniem oraz testowaniem oprogramowania pracującego na radiomodułach. W dziale RFSW można rozwijać swoją wiedzę w zakresie nowoczesnych technologii telekomunikacyjnych oraz w obszarze developmentu oprogramowania. Nasza praca daje również możliwości rozwoju umiejętności w obszarach takich jak: pomiary sygnałów radiowych, nowoczesne metodologie testowania oprogramowania, a także umiejętności w obszarze języków skryptowych i metodologii tworzenia oprogramowania opartego o praktyki Agile.

**Marcin Miernik**

R&D Manager, RF Projects  
MBB RF RFSW

# Pomiar parametrów radiowych stacji bazowych LTE (eNodeB) przy użyciu wektorowego generatora sygnałowego i analizatora widma

Marcin Domański  
Hardware Integration Engineer  
MBB RF Common HW I&V L1

**NOKIA**

## Stacja bazowa eNodeB jako nadajnik i odbiornik radiowy

Każdemu z nas, inżynierowi lub przyszłemu inżynierowi, nieraz zdążyło się usłyszeć z ust torpedowanego terminologią telekomunikacyjną laika następujące pytanie: „Czym właściwie jest ta stacja bazowa? Stacja bazowa telefonii komórkowej, w naszym przypadku eNodeB, element sieci LTE, to z perspektywy abonenta nic innego jak nadajnik i odbiornik radiowy w jednym. Z pewnością te pojęcia, funkcjonujące w społeczeństwie już przeszło wiek, łatwiej trafią do świadomości współrozmówcy o mniejszym doświadczeniu technicznym. Oczywiście, większość z nich słowo radio skojarzy z transmisją rozgłoszeniową (ang. broadcast), tak jak to ma miejsce w popularnej do dziś radiofonii FM, ale już drogą dedukcji każdy łatwo dojdzie, że ten sam proces musi odbywać się w drugą stronę.

Z tych samych powodów pomiary parametrów radiowych stacji bazowych należy traktować jak pomiary dowolnego radioodbiornika i radionadajnika, z poprawką na stosowane w systemie wielodostęp, modulację i protokół. W LTE używane są następujące metody wielodostępu:

- OFDMA (ang. Orthogonal Frequency-Division Multiple Access) wykorzystywana w transmisji od stacji bazowej do abonenta (ang. DL, od downlink), a więc w nadajniku stacji bazowej (ang. TX, od transmitter);
- SC-FDMA (ang. Single-carrier Frequency-Division Multiple Access) wykorzystywana w transmisji od abonenta do stacji bazowej (ang. UL, od uplink), a więc w odbiorniku stacji bazowej (ang. RX, od receiver).

Różnica występująca między tymi dwoma metodami jest dość istotna, ale w niniejszym artykule nie będziemy jej szczegółowo wyjaśniać. Nadmienimy tylko, że zastosowanie nieco innej modulacji w UL ma związek z wysokimi wymaganiami energetycznymi modulatora OFDMA (duży stosunek mocy maksymalnej do mocy średniej). A komuż z nas nie zależy na jak najdłuższym działaniu smartfona na baterii? Możliwość używania Facebooka w wielogodzinnej podróży w pociągu bez gniazdek jest chyba warta zapłacenia tak niewielkiej ceny, jaką jest wprowadzenie innej, na pozór bardziej skomplikowanej modulacji.

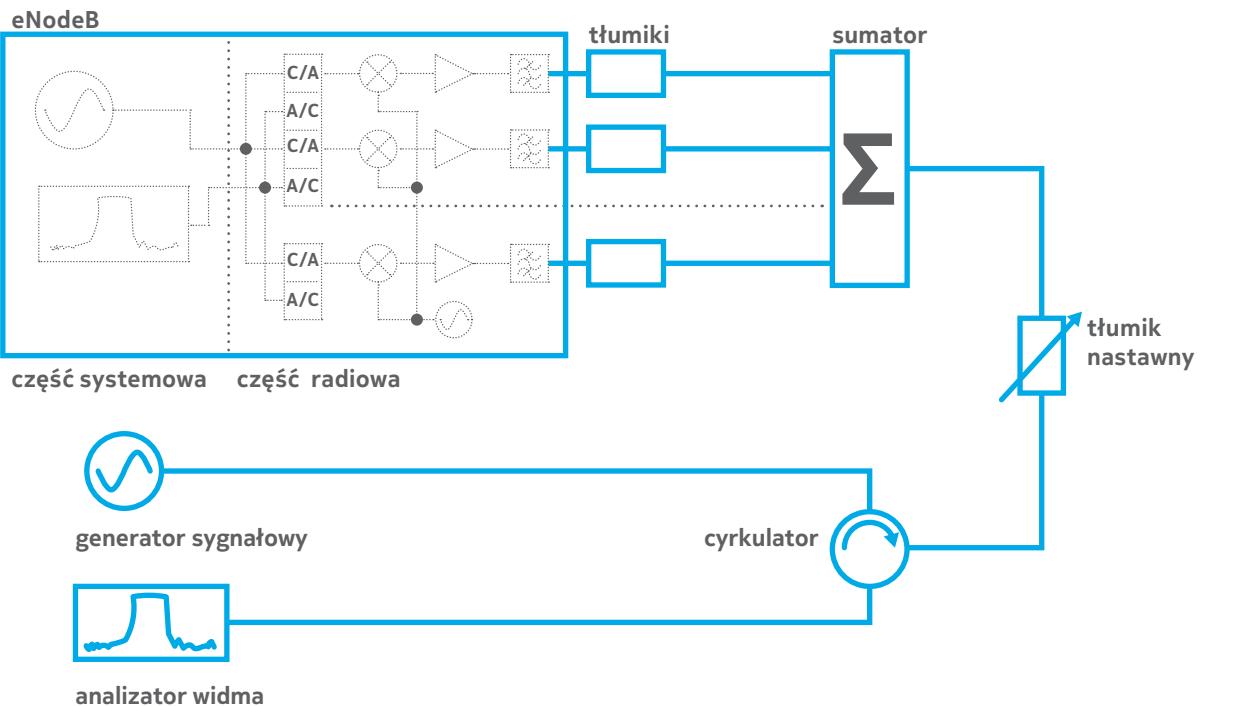
Z naszego punktu widzenia najistotniejszy będzie fakt podziału całej szerokości pasma, w którym operujemy (a więc 1,4; 3; 5; 10; 15 lub 20 MHz), na niezależne podnośne o szerokości 15kHz. Ich liczba zależy oczywiście od wybranej szerokości pasma. Na każdej z podnośnych alokowane są zakodowane dane. Nomenklatura radiowego interfejsu sieci LTE jest oparta o tzw. resource blocki, które tworzą 12 podnośnych. Te z kolei, w domenie czasowej tworzą tzw. ramkę, której czas trwania wynosi 10 ms. Większość ramki zajmują dane użytkownika. To na ich podstawie wylicza się maksymalną możliwą prędkość transmisji w sieci LTE, wartość tak pożądaną do celów marketingowych. Dane użytkownika tworzą jeden z tzw. kanałów fizycznych (PDSCH – Physical Downlink Shared Channel), obok nich znajdują się kanały wykorzystywane do niesienia informacji kontrolnych. Ich opis można odnaleźć w bardzo przystępnej formie m.in. w książce „Long Term Evolution in bullets” autorstwa Chrisa Johnsona [1], z kolei do eksploracji zawartości ramki bardzo przydatny jest aplikacja dostępna na stronie internetowej lte-bullets.com [2]. Z punktu widzenia układu pomiarowego, oprócz kanałów fizycznych, bardzo ważne są zlokalizowane w stałym miejscu sygnały synchronizacyjne. Bez prawidłowej synchronizacji nie mógłby się odbyć żaden pomiar. Dlatego, oprócz sygnału mierzonego, do analizatora widma powinniśmy dostarczyć sygnał referencyjny 10 MHz, a w przypadku pomiaru parametrów czasowych również sygnał wyzwalania (ang. trigger), którego częstotliwość wynika wprost z czasu trwania ramki (a więc 10 ms) i wynosi 100 Hz. Obydwa sygnały są generowane ze stacji bazowej.

Pomiary parametrów radiowych są o tyle ważne, że jakość transmisji wpływa nie tylko na poziom usług u danego operatora, ale może także zakłócać cały radiowy ekosystem, ze szczególnym uwzględnieniem częstotliwościowych sąsiadów. Stąd też wzorzec sygnałów przeznaczonych do testów, wraz z zestawem obowiązkowych pomiarów jest ściśle zdefiniowany przez organizację standaryzacyjną 3GPP w zaleceniu 36.141 [3]. Szczęście każdego testera w DL nosi nazwę Test Modeli, natomiast w UL – Fixed Reference Channels (FRC). W ich ramach zawarte są wszystkie potrzebne kanały wraz z ich wypełnieniem. Wszak urządzenie pomiarowe nie wiedząc jakie dane wysłaliśmy nie byłoby w stanie zweryfikować poprawności transmisji. W świecie makroskopowym, w jakim niewątpliwie osadzone są techniki wielkiej częstotliwości, trudno bowiem o istnienie znanego z mechaniki kwantowej, jednocześnie żywego i martwego kota Schrödingera. Zresztą znane z wylegówania się w ciepłych miejscach domowe drapieżniki mogłyby prędko nabawić się problemów zdrowotnych z zupełnie innych przyczyn, takich jak np. laboratoryjny zgietek.

## Układ pomiarowy

Układ stosowany przy pomiarach prototypów stacji bazowych w ramach integracji warstwy fizycznej, jaka ma miejsce we wrocławskim oddziale Nokii, został przedstawiony na **Rysunku 1**. Najczęściej stację bazową rozdziela się na dwie części: systemową i radiową. Ta pierwsza, poza wszystkimi zadaniami, jakie wypełnia w kontekście połączenia z resztą sieci LTE, w DL ma przede wszystkim z nadchodzących danych utworzyć zmodulowany sygnał i wysłać go w kierunku części radiowej, natomiast w UL zdemodulować sygnał przychodzący od radia. Z naszej perspektywy będzie ona tylko generatorem sygnału DL i analizatorem sygnału z UL. Część radiowa jest niczym innym jak przetwornikiem analogowo-cyfrowym, mieszanym częstotliwości, wzmacniaczem oraz filtrem pasmowoprzepustowym. Ten drugi przesuwa zmodulowany sygnał w pożądane pasmo. Listę dozwolonych pasm można odnaleźć w zaleceniu 3GPP 36.101 [4] lub na stronie niviuk.free.fr [5], ich użycie zależy od regionu geograficznego, w którym moduł będzie pracował. W Europie najczęściej wykorzystywane jest pasmo pierwsze, zlokalizowane w sąsiedztwie 2100 MHz w DL i 1900 MHz w UL.

**1 Rysunek** Schemat blokowy układu wykorzystywanego do pomiarów stacji bazowej eNodeB. Przerywaną linią wewnętrz eNodeB nanieiono schemat zastępczy stacji bazowej z perspektywy integracji warstwy fizycznej



W celu uzyskania właściwych pomiarów mocy należy zmierzyć tłumienie toru radiowego. Zazwyczaj robi się to wysyłając predefiniowany sygnał o częstotliwości wykorzystywanej w DL przez układ pomiarowy i mierząc go na używanym analizatorze. Trzeba zaznaczyć, że bez skalibrowanego sprzętu pomiarowego przeprowadzane pomiary mocy są w zasadniczo jedynie obserwacjami, nie zaś dokładnymi pomiarami. Analiza jakościowa, bardzo ważna w pierwszych etapach integracji, stanowi podstawę do dalszych, bardziej szczegółowych pomiarów, które odbywają się w kolejnych fazach testów.

Do rozważań na temat mocy sygnału przyda się nam dobrodziejsztwo wynalezionej w XIX wieku logarytmu (choć są przesłanki ku temu by twierdzić, że blisko jego odkrycia był już sam Euklides [6]). W naszym przypadku bardziej niż jego właściwość „spłaszczenia” wielkich wartości, użyteczny będzie fakt, że operacje multiplikatywne następuje on operacjami addytywnymi. Dodawanie i odejmowanie wielkości rzędu dziesiątek jest zazwyczaj dużo przyjemniejsze niż mnożenie i dzielenie przez tysiące. W skali logarytmicznej moc sygnału wynosi  $10 \cdot \log(P)$ , najczęściej moc P podajemy w miliwatach, wtedy jednostką jest dBm (decybel odniesiony do jednego miliwata). Wszelkie tłumienia podajemy w dB, a dodając tą wartość do wartości zmierzonych w dBm uzyskamy prawidłowy wynik, również w dBm.

W rzeczywistości nie ma potrzeby ręcznego dodawania tej poprawki, bo uwzględnia ją za nas analizator widma, po wprowadzeniu jej w odpowiednie miejsce. Na pewno warto zapamiętać, że dodanie 3dB oznacza podwojenie mocy, natomiast 40dBm to 10W. Wychodząc z tego punktu łatwo już będzie sobie wyliczyć najpowszechniej występujące moce rzędu dziesiątek wat (np.  $46 \text{ dBm} \equiv 40 \text{ W}$ ).

Prześledźmy zatem kolejne elementy toru pomiarowego:

• **Tłumiki** – wkręcane są bezpośrednio na wyjście antenowe, mają za zadanie radykalnie obniżyć moc emitowanego sygnału. Moce rzędu dziesiątek watów są bowiem dla urządzeń testowych falą uderzeniową, której mogą nie przetrwać, a niektórzy twierdzą nawet, że mogą wpływać na sprawność prokreacyjną męskiej części testerów. Ich tłumienie jest najczęściej rzędu 30-50dB i stanowi główny element tłumienia toru radiowego. Innym ważnym parametrem jest maksymalna moc wejściowa, jaką może przyjąć tłumik. Należy pamiętać, aby była ona większa od maksymalnej mocy, którą będziemy wypromieniowywać z wyjścia antenowego.

• **Tłumiki regulowane** – w kryzysowych sytuacjach potrzeba dodania dodatkowego tłumienia bywa bardzo przydatna.

- **Rozdzielacze/sumatory sygnału radiowego** – wprowadzone przez nie tłumienie jest zależne od liczby wyprowadzeń, jakie posiadają.

- **Cyrkulatory** – w których sygnał przekazywany jest jednokierunkowo, jedynie pomiędzy sąsiednimi wyjściami. Za ich pomocą możemy np. spowodować, żeby sygnał z DL nie wchodził na wejście generatora.

- **Separator składowej stałej** – składowa stała dla urządzeń pomiarowych wielkich częstotliwości również bywa bardzo groźna.

- **Kable** – z reguły zakończone najpowszechniejszymi w technikach wielkiej częstotliwości złączami N lub SMA. Poza ścisłe technicznymi parametrami (w tym impedancją falową wynoszącą  $50 \Omega$ ) bardzo ważna jest wygoda ich użycia. Złącza N bardzo łatwo wkręcić ręcznie, natomiast mocowanie złącz SMA bez odpowiedniego narzędzia wymaga zręcznego operowania palcami.

Pośród licznych pułapek, jakie tworzy płytanina kabli w torze radiowym, oprócz ryzyka zapłatańskiego się testera we własną sieć, do najbardziej powszechnych należą przede wszystkim: jakość wykonanych własnoręcznie złącz lub kabli, częstotliwościowy zakres pracy posiadanych elementów toru radiowego (głównie rozdzielaczy i cyrkulatorów; przed testami należy sprawdzić czy są zgodne z częstotliwością na której pracuje testowana stacja bazowa) oraz poprawna identyfikacja wejść i wyjść rozdzielaczy (nie wszystkie kombinacje wprowadzają jednakowe tłumienie).

### Pomiar i interpretacja wyników

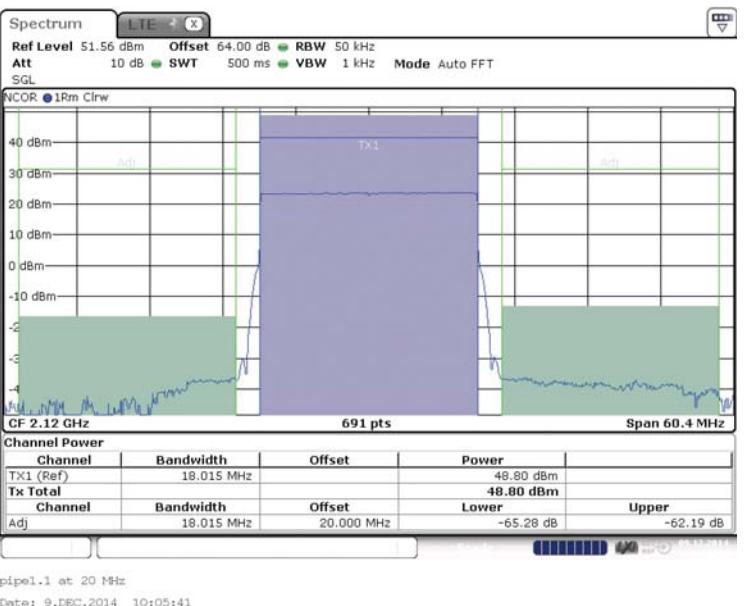
Wykorzystywany we wrocławskich laboratoriach sprzęt pomiarowy cechuje się dużą automatyzacją. Posiadając wykupione odpowiednie dodatki programowe można z łatwością przygotować analizator do interpretacji odpowiedniego Test Modelu oraz generator do wysyłania odpowiedniego FRC.

Na **Rysunkach 2 do 5** przedstawiono zrzuty ekranów z pomiarów parametrów DL w analizatorze widma. Elementem wspólnym jest częstotliwość środkowa, na której wysyłamy sygnał (w analizowanej wersji eNodeB – 2120 MHz) oraz zmierzona uprzednio tłumienie toru, które w tym przypadku wynosi 64dB. Do podstawowych pomiarów w DL zaliczamy:

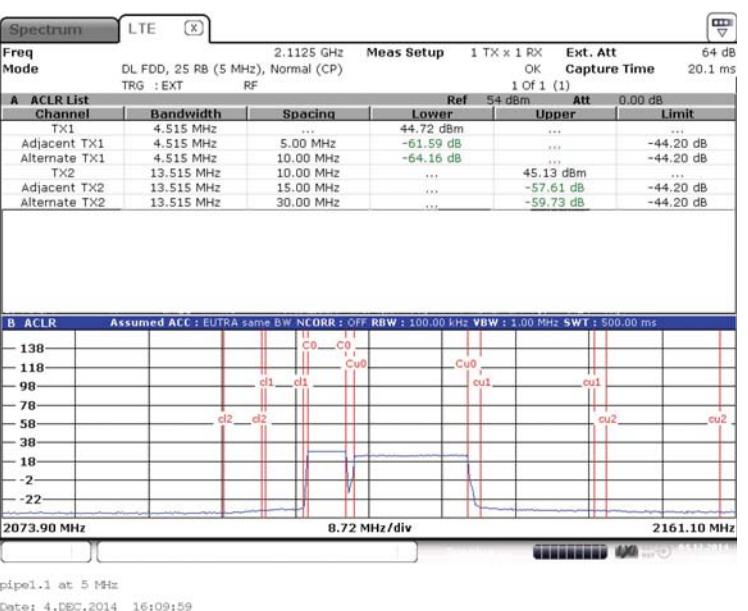
- **Pomiar mocy** wypromienowanej przez stację bazową. Do tego pomiaru wykorzystuje się Test Model E-TM1.1. Przykład dla największej szerokości pasma (20 MHz) pokazano na **Rysunku 2**. Wysłano sygnał z maksymalną możliwą mocą 60 W (47,78 dBm), natomiast wartość zmierzona wynosi 48,8 dBm. Różnica ta może się w normie stawianej w zaleceniu 36.141 [3], czyli  $\pm 2$  dB.

- **Pomiar ACLR** (ang. Adjacent Channel Leakege Ratio). Jak wskazuje rozwinięcie angielskiego akronimu, to właśnie ten parametr jest tym, który najbardziej świadczy o wpływie na radiowy „ekosystem”. Leakege – czyli przeciek do sąsiada

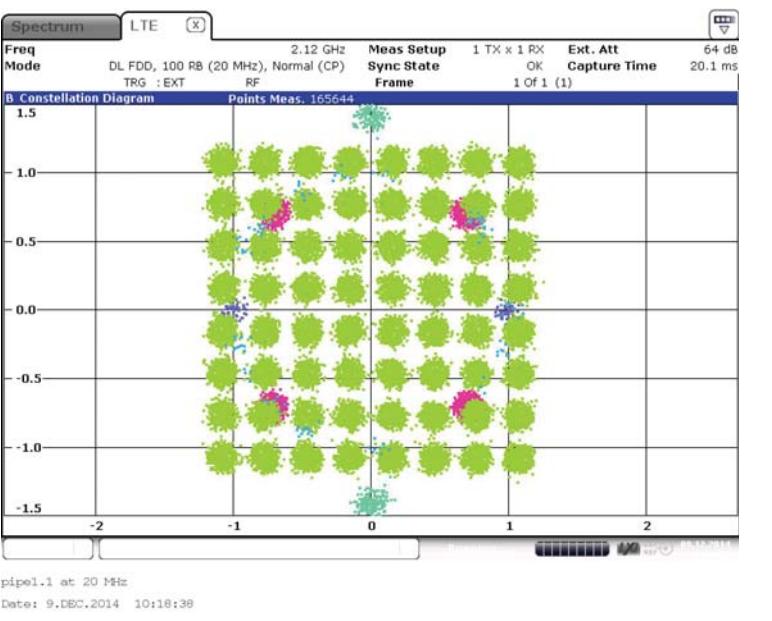
**2 Rysunek** Wykres gęstości widmowej sygnału o szerokości 20 MHz wykorzystywany do pomiaru mocy.



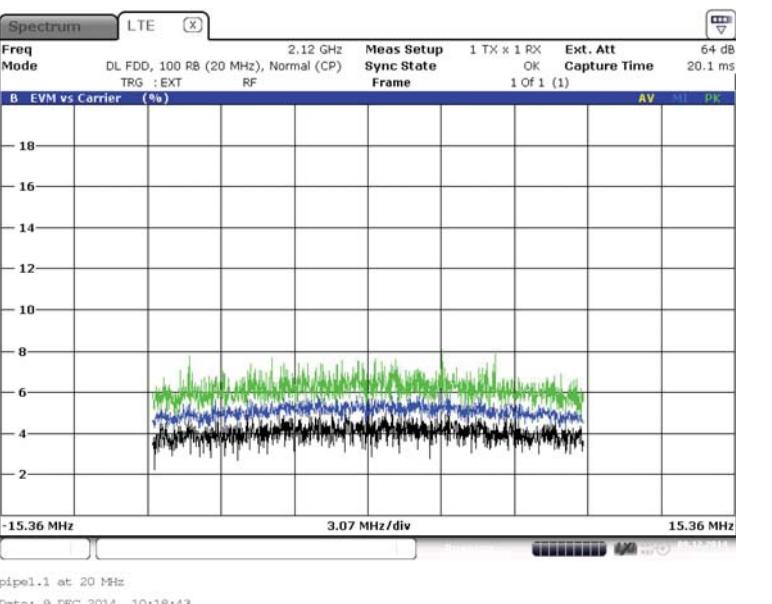
### 3 Rysunek Pomiar ACLR



#### 4 Rysunek Wykres konstelacji sygnału z Test Modelu E-TM3.1.



#### 5 Rysunek Wykres zmierzonego EVM w dziedzinie podnośnych



(adjacent channel), nikt z nas chyba nie chciałby się znaleźć w takiej sytuacji. Kojarzy się ona z odszkodowaniami i zepsuciem relacji. Dlatego na takie zdarzenia warto być ubezpieczonym. Pomiar ACLR pokazano na **Rysunkach 2 i 3**. Na wykresie gęstości widmowej sygnału odpowiednimi liniami rozgraniczono kanał mierzony oraz kanały sąsiednie. W przypadku pokazanego na **Rysunku 3** pomiaru zestawu dwóch nośnych naraz – o szerokości odpowiednio 5 i 15 MHz – pomiar ACLR odbywa się dla kanałów będących sąsiadami wobec całości transmitowanego sygnału. Wynikiem są dwie wartości – tzw. Lower ACLR, będący różnicą między mocą mierzoną w kanale z sygnałem użytecznym, a mocą w kanale znajdującym się poniżej jego częstotliwości oraz Upper ACLR, gdzie punktem odniesienia jest kanał powyżej częstotliwości pracy stacji bazowej. Limit wynosi 44,2 dB, jak widzimy jest on spełniony z nawiązką we wszystkich przedstawionych tutaj pomiarach. Warto zwrócić uwagę na to, że nośne dla 5 i 15 MHz nie mają jednakowej „wysokości”. Wynika to z tego, że sumaryczna moc 60 W jest podzielona po równo pomiędzy oba kanały (po 30W, a więc 44,78 dBm). Aby średnie moce się zgadzały (jednakowe pole pod wykresem), dla 5MHz poszczególne podnośne muszą posiadać moc wyższą niż dla 15MHz. W tym przypadku grubszy zawsze będzie niższy.

• **Pomiar EVM** (ang. Error Vector Magnitude). Na każdej z podnośnych w łącznym sygnale LTE wysyłany jest zestaw danych zmodulowanych kwadraturowo. Mogą być to, więc modulacje BPSK, QPSK, QAM16 i QAM64, przenoszące odpowiednio 1, 2, 4 i 6 bitów na jeden symbol. W przypadku wykorzystywanego przez nas Test Modelu E-TM3.1 dane użytkownika (kanal fizyczny PDSCH) są modulowane za pomocą QAM 64. Przykład konstelacji na płaszczyźnie zespalonej pokazano na **Rysunku 4**. Zielone punkty to dane użytkownika z całej ramki. Innymi kolorami naniesione są konstelacje kanałów kontrolnych. Niewątpliwie wykresy konstelacji posiadają duży walor artystyczny, ale pomijając zdolności niektórych inżynierów w zakresie krytyki sztuki, same w sobie nie są w stanie dać obiektywnej odpowiedzi na temat poprawności transmisji. W tym właśnie celu wprowadzono EVM, który, mówiąc kolokwialnie i kontynuując kosmiczną analogię, jest znormalizowaną różnicą wektorów od punktu zerowego płaszczyzny zespalonej do odpowiednio: punktu, w którym powinna znajdować się gwiazda i punktu, w którym gwiazda znalazła się naprawdę. Gwiazdami są oczywiście reprezentacje sygnałów z kolejnych podnośnych na płaszczyźnie zespalonej. Wszystkie widzimy na wykresie konstelacji. Z kolei wykres procentowych wartości EVM odpowiadających wszystkim podnośnym można zobaczyć na **Rysunku 5**. Tutaj także nieprzekroczony jest wyznaczony przez normę 36.141 limit 8%.

W przypadku UL wysyłamy predefiniowany sygnał z wektorowego generatorka sygnałowego, aby uzyskać modulację QAM16 (teoretycznie największą możliwą jest nawet QAM64) wykorzystujemy FRC

A4-8. Używany przez nas generator posiada możliwość generowania takich sygnałów automatycznie. W przeciwnym wypadku koniecznie byłoby stworzenie odpowiednich wzorców. Częstotliwość pracy generatora to w przypadku przedstawionych pomiarów 1720 MHz.

Oprócz obserwacji konstelacji analogicznej do tej z DL (w tym przypadku sprawdzanej tylko dla kanału z danymi użytkownika PUSCH – Physical Uplink Shared Channel), w UL do podstawowych obserwacji dokonywanych w naszym środowisku można zaliczyć:

- Pomiar **BLER** (ang. BLock Error Rate) – jest to procentowa wartość stosunku ilości przekłamanych bloków do wszystkich wysłanych bloków. Blok jest jednostką umowną i nie należy go mylić z resource blockiem. BLER jest odpowiednikiem bitowej stopy błędów (BER, ang. Bit Error Rate) w skali bloków.
- Pomiar SNR (ang. Signal to Noise Ratio) – czyli stosunek mocy sygnału do szumu, podany w dB.

Oczywiście, omówione pomiary nie wyczerpują wszystkich możliwości. Ich pełny zakres można znaleźć w normie 36.141 [3]. Znajdują się tam m.in. pomiary błędu częstotliwości, szerokości wykorzystywanej pasma, zakłóceń intermodulacyjnych oraz czułości i selektywności odbiornika. Są to bardzo ciekawe doświadczenia, wykonywane jednak w następnych fazach testów, przy kolejnych wersjach prototypu stacji bazowej. O uniwersalności tych pomiarów można przekonać się chociażby zasięgając do popularnego podręcznika dla technikum [7].

We Wrocławskim dziale integracji i weryfikacji sprzętowej ten niedosyt pomiarowy zaspakajemy testowaniem reszty technologii komórkowych, a więc również GSM i WCDMA (radiowego interfejsu sieci UMTS). Ponadto wiele wyzwań stoi już na etapie przygotowania stacji bazowej do nadawania i odbioru, gdzie wykorzystuje się jeszcze nie w pełni funkcjonalne oprogramowanie. Te czynności są nieraz bardziej kiształcące niż sam pomiar, który jest już tylko wisienką na wysokoczęstotliwościowym torcie.

#### Bibliografia

- [1] Chris Johnson „Long Term Evolution in bullets”
- [2] <http://www.ite-bullets.com/>
- [3] 3GPP TS 36.141 „Base Station (BS) conformance testing” <http://www.3gpp.org/dynareport/36141.htm>
- [4] 3GPP TS 36.101 „User Equipment (UE) radio transmission and reception” <http://www.3gpp.org/dynareport/36101.htm>
- [5] [http://niviuk.free.fr/lte\\_band.php](http://niviuk.free.fr/lte_band.php)
- [6] Maciej Sysło, Anna Kwiatkowska „Myśl logarytmicznie!” [http://www.deltami.edu.pl/temat/matematyka/analiza/2014/11/28/Mysl\\_logarytmicznie\\_/](http://www.deltami.edu.pl/temat/matematyka/analiza/2014/11/28/Mysl_logarytmicznie_/)
- [7] Tomasz Bogdan „Urządzenia radiowe”

#### Marcin Domański

Hardware Integration Engineer  
MBB RF Common HW I&V L1

# Sieć mobilna – cybernetyczne pole walki

Łukasz Gostkowski  
Software Developer  
MBB Security R&D

**NOKIA**

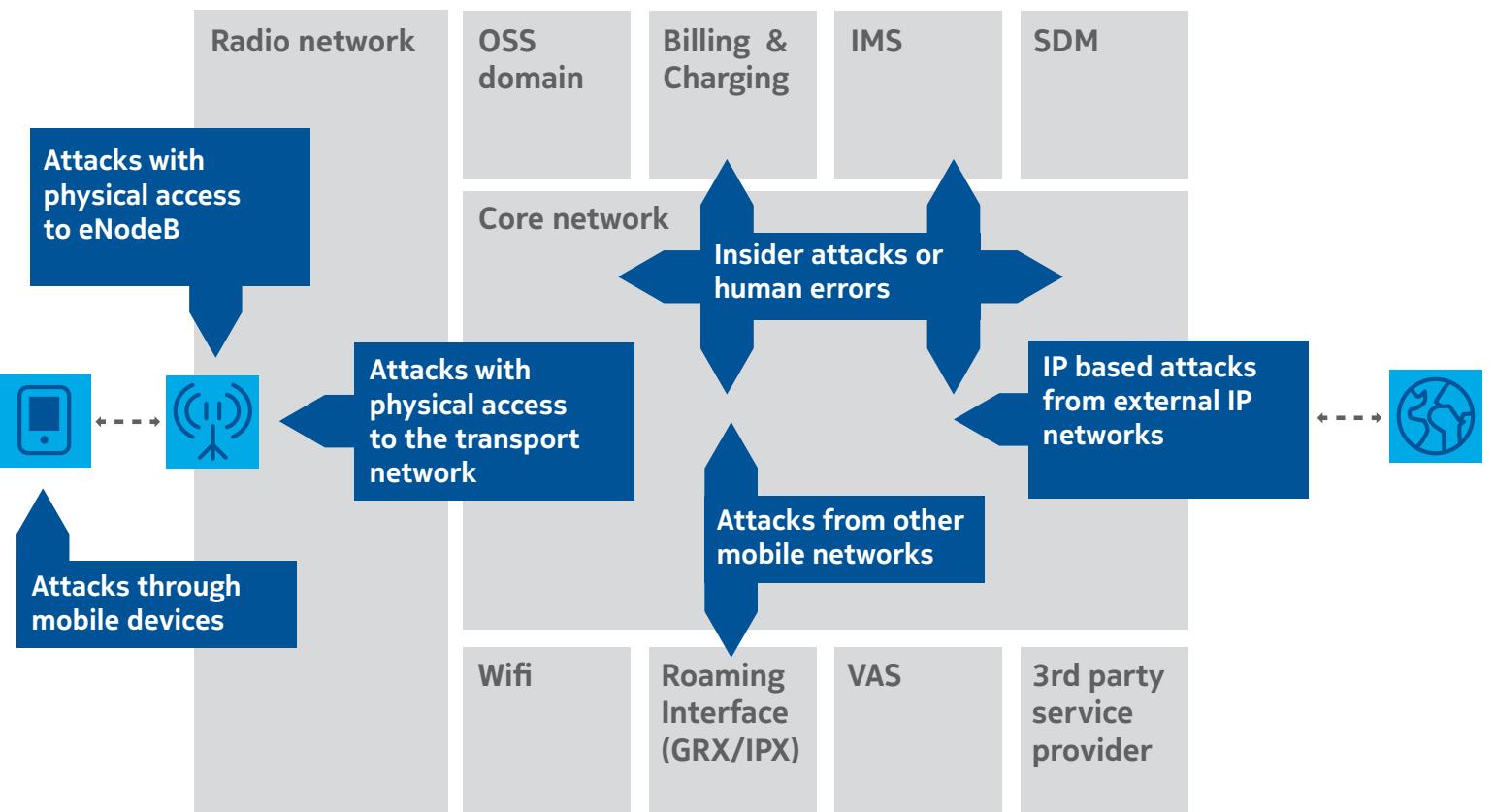
Rozwój nowych generacji sieci mobilnych niesie ze sobą nowe korzyści i możliwości, zarówno dla użytkowników końcowych, jak i dla operatorów. Nowe technologie oznaczają jednak również nowe wyzwania, także w sferze bezpieczeństwa.

Sieci komórkowe są dzisiaj jednym z najdynamiczniej rozwijających się obszarów IT. Wraz z wprowadzaniem technologii 3.5G czy 4G, takich jak LTE (Long Term Evolution) czy LTE-A (LTE-Advanced), rośnie poziom i jakość dostępnych usług. Sama możliwość dzwonienia, wysyłania wiadomości tekstowych (SMS) czy multimedialnych (MMS) przestaje być podstawowymi funkcjonalnościami. Rosnący przesył danych za pomocą bezprzewodowego, szerokopasmowego dostępu do internetu, możliwość łączenia się z sieciami Wi-Fi, instalacja różnego rodzaju aplikacji czy gier, geolokalizacja i dostęp do map, usługi związane z bankowością – oto możliwości, z których coraz częściej korzystamy i bez których ciężko sobie obecnie wyobrazić „telefon” komórkowy. Smartfonom dużo bliżej dziś do zaawansowanych komputerów niż do tradycyjnie rozumianych telefonów. Niestety, niesie to za sobą także pewne konsekwencje. Jeszcze nie tak dawno temu

niewyobrażalnym było pojęcie wirusa na telefonie komórkowym. Dziś urządzenia mobilne nie tylko są podatne na różnego rodzaju złośliwe oprogramowanie, ale stanowią jeden z krytycznych elementów w kontekście bezpieczeństwa telefonii komórkowej. Oczywiście za rozwój nowych generacji i stopniowym wprowadzaniem nowych technologii stoi nie tylko rozwój samych telefonów. Konieczny jest także rozwój samej struktury sieci zarówno w obszarze sieci dostępowej, jak i szkieletowej. **Rysunek 1** przedstawia podstawowe grupy ataków, na jakie narażona jest sieć komórkowa, z uwzględnieniem niżej wymienionych obszarów:

- Ataki związane bezpośrednio z urządzeniem mobilnym, wykorzystujące na przykład złośliwe oprogramowanie (malware) lub podatności (vulnerabilities) aplikacji mobilnych;
- Ataki związane z bezpośredniem dostępu do stacji bazowej (eNodeB);
- Ataki związane z dostępem do sieci transportowej (transport network) łączącej radiową sieć dostępową (radio network) z siecią szkieletową (core network);

1 Rysunek Możliwe obszary ataków w sieci mobilnej



- Wewnętrzne ataki w sieci szkieletowej, takie jak ataki socjotechniczne czy ataki wykorzystujące przejęty przez atakujących element wewnętrz sieci;
- Ataki związane z interfejsem GRX (GPRS roaming interface) lub IPX (IP exchange) łączącym różne sieci mobilne (np. sieci różnych operatorów);
- Ataki pochodzące spoza sieci operatora.

Jak widać, dla operatorów telefonii komórkowej ataki skierowane bezpośrednio w urządzenia końcowe, czyli nasze telefony, stanowią tylko jedną z możliwych grup ataków.

Sieć LTE to sieć w pełni pakietowa (all-IP network), w której każdy element posiada przydzielony adres IP. Oznacza to, że każdy taki element podatny jest na tradycyjne techniki ataków cybernetycznych. W następnych sekcjach przyjrzymy się bliżej wybranym obszarom infrastruktury sieci telefonii komórkowej oraz związanymi z nimi typami możliwych ataków.

#### Urządzenia mobilne

Urządzenia końcowe są pierwszym, najbardziej podatnym elementem w telefonii komórkowej. Z tymi urządzeniami związane są dwa podstawowe sposoby ataków:

- Ataki związane z protokołem komunikacji – GSM, Wi-Fi, Bluetooth;
- Ataki związane z warstwą aplikacji, realizowane przez złośliwe oprogramowanie lub wykorzystujące luki w wykorzystywanych przez abonentów aplikacjach mobilnych.

Do pierwszej z tych grup możemy zaliczyć różnego rodzaju ataki typu IMSI Catcher, mające na celu przechwycenie numeru IMSI (International Mobile Subscriber Identity), czyli unikatowego numeru abonenta przypisanego do każdej karty SIM. Tego typu ataki realizowane są na przykład poprzez podstawienie fałszywej stacji bazowej. W takim przypadku telefon komórkowy, znajdując mocniejszy sygnał sieci, przepina się na fałszywy nadajnik, który z kolei łączy się z oryginalną siecią. Jest to typowy przypadek ataku MITM (Man in the middle) pozwalający na podsłuch połączeń czy transmisji danych.

Podobny atak, można przeprowadzić na warstwie Wi-Fi. W tym przypadku atakujący podstawa fałszywy punkt dostępu (access point), z którym łączy się interfejs telefonu ofiary. Przy tego typu atakach nie jest możliwy podsłuch samej rozmowy telefonicznej (o ile nie jest prowadzona z wykorzystaniem technologii VoIP), ale za to możliwe jest przechwycenie transmisji pakietowej i jej modyfikacja.

Druga grupa ataków obejmuje wszelkiego rodzaju ataki związane z oprogramowaniem na telefonie komórkowym. W tej grupie znajdują się wszelkiego rodzaju ataki wykorzystujące złośliwe oprogramowanie (malware) lub też podatności zainstalowanych aplikacji. Na szczególną uwagę zasługują tutaj aplikacje wykorzystujące połączenie z internetem. Słaba implementacja mechanizmów zabezpiecza-

jących, słaba autoryzacja i autentykacja, dane przechowywane w niezabezpieczony sposób czy nieszyfrowane połączenie klient-serwer – są to elementy, które sprawiają, że aplikacja mobilna może stać się łatwym celem (exploitation), co z kolei prowadzi do pozyskania przez atakującego poufnych informacji dotyczących abonenta lub też ułatwia wprowadzenie modyfikacji w telefonie ofiary i zainstalowanie złośliwego oprogramowania, wykorzystywanego następnie do dalszych ataków lub zamieniającego telefon ofiary w jeden z węzłów botnetu.

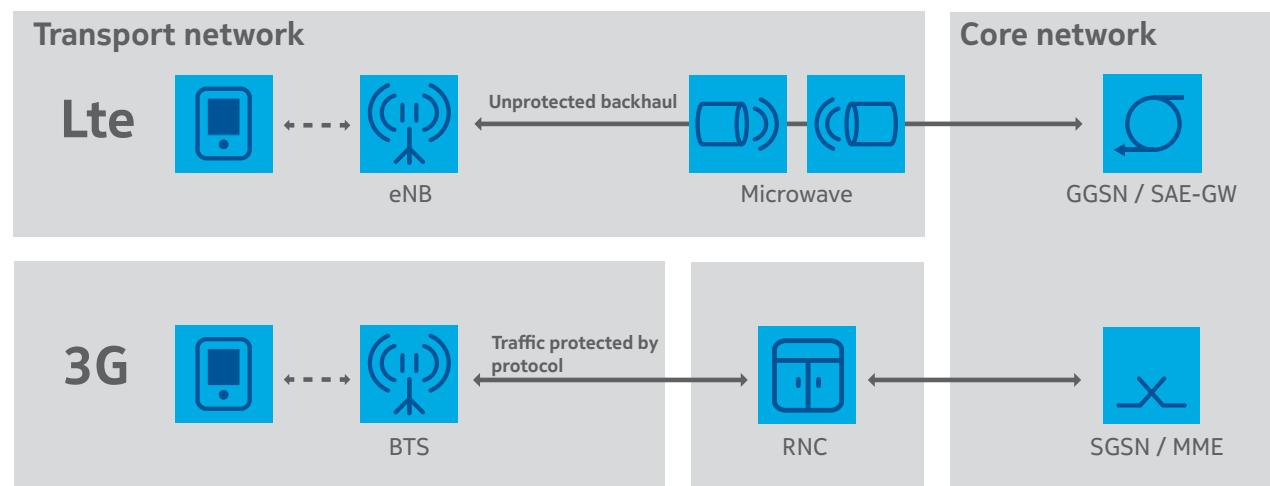
Według badań Mobile Threat Center (MTC) firmy Juniper ilość oprogramowania typu malware wzrosła o 614% od marca 2012 r. do marca 2013 r. Ataki celujące w urządzenia końcowe stanowią nie tylko zagrożenie dla abonentów, ale także dla operatorów. Dlatego też na operatorach spoczywa po części obowiązek zadbania o bezpieczeństwo swoich klientów w kontekście ochrony przed cyberatakami. Jednym z rozwiązań jest monitoring pakietów użytkowników w sieci szkieletowej, realizowany przez oprogramowanie Mobile Guard firmy Nokia. Celem tego systemu jest ochrona użytkowników końcowych przed złośliwym oprogramowaniem, także w przypadku, kiedy abonent nie jest w posiadaniu żadnego rozwiązania zabezpieczającego (anti-malware) przed tego typu programami. Mobile Guard, zintegrowany z siecią szkieletową, analizuje pakiety użytkowników pod kątem znanych zagrożeń. Do ich wykrywania używa dostarczanej przez partnera Nokii, firmę F-Secure, bazy sygnatur wirusów i malware, z którą stara się skorelować podejrzany pakiet. W przypadku odnalezienia pakietu pochodzącego z zainfekowanego urządzenia mobilnego Mobile Guard powiadamia zarówno użytkownika końcowego, jak i operatora. Dodatkowo jest w stanie przedsięwziąć określone akcje w celu zredukowania zagrożeń, na przykład zablokować transmisję danych. Rozwiązywanie wypożyczone jest także w mechanizm samokształcenia, pozwalający na rozwój metod i algorytmów analizy.

#### Sieć dostępowa

Sieć dostępowa obejmuje radiową część infrastruktury telefonii mobilnej. Jej pierwszym, podstawowym elementem jest stacja bazowa (BTS, Base Transceiver Station) zwana w technologii LTE eNodeB. Jest to także pierwszy element bezpośrednio narażony na atak, na przykład poprzez fizyczne uzyskanie dostępu do urządzenia poprzez port Ethernet. Dzięki temu atakujący uzyskuje dostęp do sieci operatora (poprzez połączenie transportowe, także sieci szkieletowej) i jest w stanie podsłuchać dane pakietowe abonentów (atak MITM), zainfekować przesypane pakiety czy przeprowadzić ataki typu DoS (Denial of Service) na inne komponenty sieci. Dodatkowym czynnikiem ułatwiającym atakującemu przeprowadzenie ataku jest fakt, że w technologii LTE część transportowa (backhaul), łącząca stację bazową z siecią szkieletową, jest często niezabezpieczona. **Rysunek 2** pokazuje różnice pomiędzy technologiami 3G (UMTS) a LTE w kontekście połączenia transportowego.

W technologii LTE funkcjonalność kontrolera sieci radiowej (RNC, Radio Network Controller) przejmuje sama stacja bazowa (eNB), więc

**2 Rysunek** Różnice połączenia stacji bazowej w technologiach LTE i 3G



szyfrowanie połączenia ma miejsce tylko na linii urządzenie końcowe – eNB. Wprowadziło to nowe zagrożenia w porównaniu ze starszymi technologiami, a pojedyncza stacja bazowa staje się punktem dojęcia do całej infrastruktury sieci. Pozwala to na przykład na wykonanie skanowania sieci, stworzenie jej topologii i zidentyfikowanie podatnych na atak celów. Wykonując następnie rozproszony atak odmowy usługi (DDoS), atakujący jest w stanie doprowadzić do wyłączenia sieci na obszarach danej stacji bazowej i stacji sąsiadujących, odcinając dostęp do sieci w całym regionie.

Według badań Heavy Reading w 2013 roku 20% operatorów sieci komórkowych stawało się celem minimum trzech ataków DDoS miesięcznie. Stanowi to niemal trzykrotny wzrost w porównaniu z rokiem 2012 (wówczas do takiej ilości ataków przyznawało się tylko 8% dostawców usług telefonii komórkowej). 43% operatorów przyznało, że było celem od kilku rocznie do dwóch ataków miesięcznie. Co więcej, w 2013 roku 60% operatorów, na skutek ataków, nie było w stanie dostarczyć usług dostępu do sieci co najmniej przez jedną godzinę (w skali roku).

Łatwość przeprowadzenia ataku na radiową część sieci powoduje, że także i w tym obszarze operatorzy muszą szukać rozwiązań zwiększających bezpieczeństwo. Jednym z takich rozwiązań jest wprowadzenie metod szyfrowania i autoryzacji na poziomie transportowym. Mimo, że to rozwiązanie nie jest częścią standardu, coraz więcej operatorów decyduje się na wprowadzenie tego typu zabezpieczenia. Polega ono przede wszystkim na wprowadzeniu szyfrowania protokołu IP (IPSec) i wprowadzeniu autoryzacji stacji bazowych za pomocą infrastruktury klucza publicznego (PKI, Public Key Infrastructure). **Rysunek 3** przedstawia bezpiecznie realizowa-

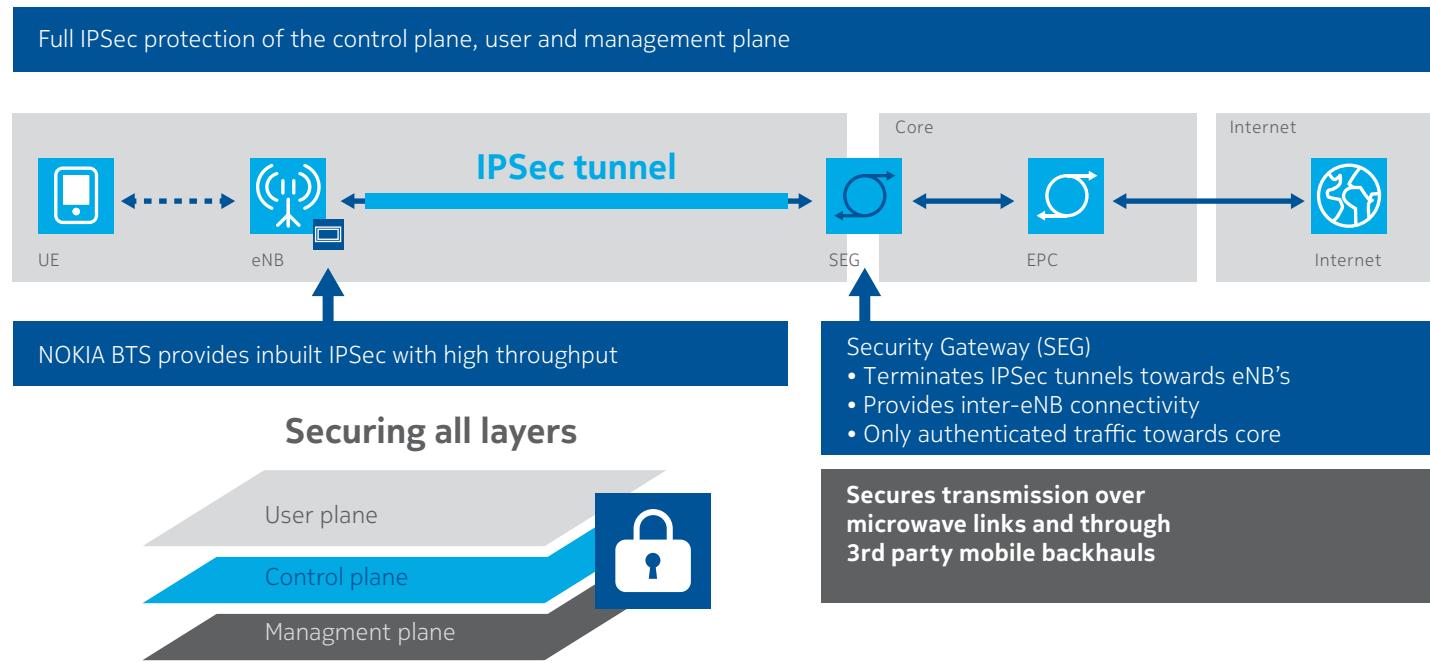
ne połączenie sieci radiowej z siecią szkieletową, z użyciem bramy dostępowej (SEG, Security Gateway), zapewniającej odpowiedni poziom autoryzacji i autentykacji ruchu sieciowego dla każdej warstwy.

**3 Rysunek** W zabezpieczeniu sieci dostępowej dodatkowo uwzględnia infrastrukturę klucza publicznego (PKI).

#### Sieć szkieletowa

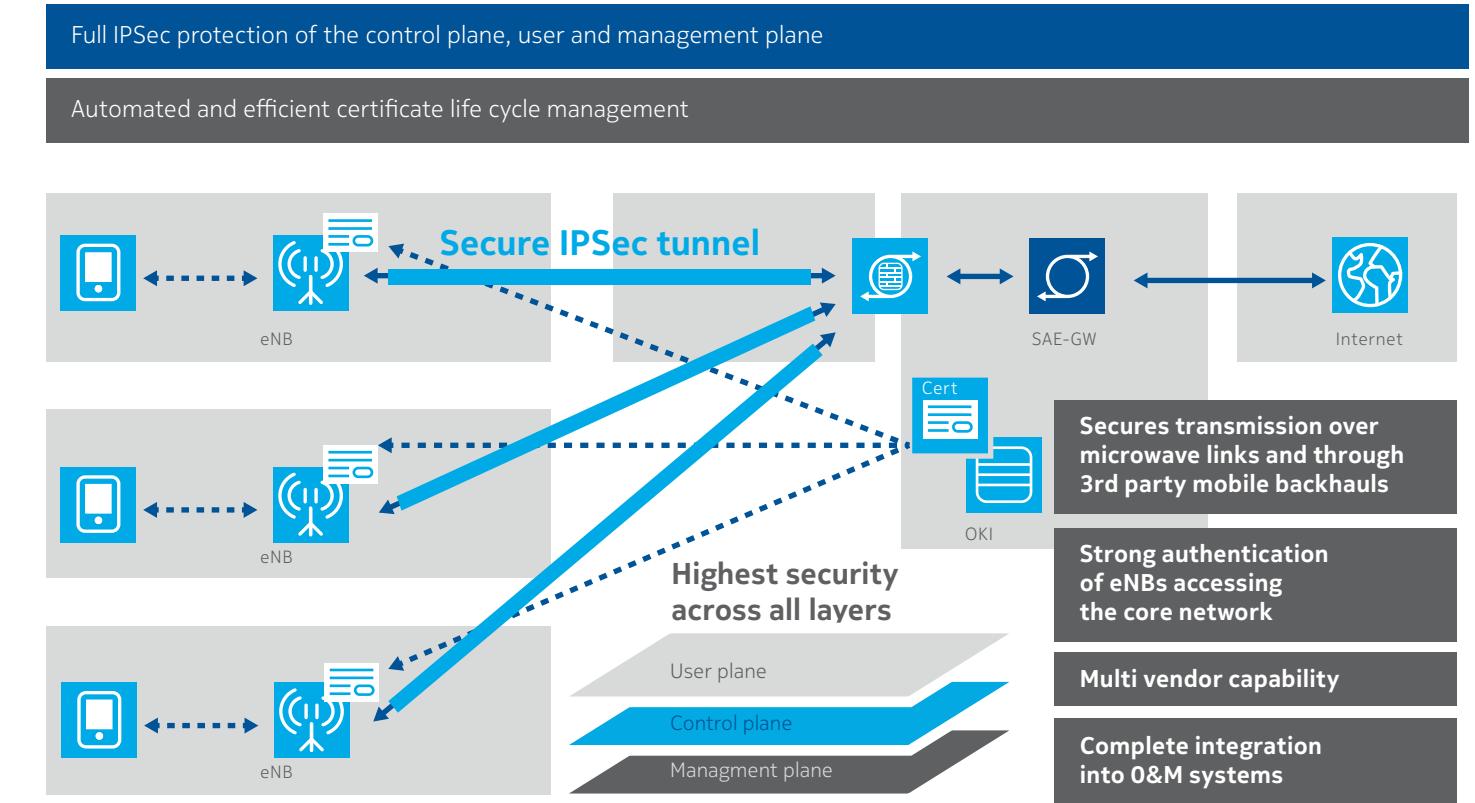
Ostatnim i jednocześnie najbardziej złożonym obszarem sieci mobilnej jest sieć szkieletowa. To właśnie tutaj znajdują się elementy odpowiedzialne za „logikę” działania telefonii komórkowej. Serwery baz danych, połączenia z sieciami telefonii stacjonarnej, połączenia z internetem, interfejsy komunikacji pomiędzy operatorami, komponenty odpowiedzialne za konfigurację elementów sieci (zakładającej dostępowej, jak i szkieletowej), połączenia z siecią korporacyjną operatora, systemy bilingwowe czy zintegrowane systemy OSS (Operation Support Systems) – to tylko niektóre z komponentów, jakie występują w sieci szkieletowej. Niejednokrotnie też niektóre z tych systemów składają się z większej liczby elementów realizujących daną funkcjonalność, co z kolei zwiększa liczbę możliwych celów ataku. Podatne są nie tylko urządzenia, ale także działające na nich specyficzne aplikacje, często udostępniające webowy interfejs dla użytkownika. Dodatkowo wiele komponentów realizowanych jest w środowisku zwirtualizowanym (virtual network) i w chmurze (cloud), co dodatkowo zwiększa liczbę możliwych ataków. Tak duża liczba systemów, połączeń, interfejsów i wysoki stopień złożoności sprawiają, że sieć szkieletowa jest w naturalny sposób podatna na wszelkiego rodzaju ataki (DDoS, Spoofing, SQL Injection, eavesdropping itp.), zarówno te pochodzące z wewnętrz (przejęte maszyny, pracowników, ataki socjotechniczne), jak i z zewnętrz (internet, IPX, GRX),

3 Rysunek Połączenie sieci radiowej z siecią szkieletową z użyciem IPSec

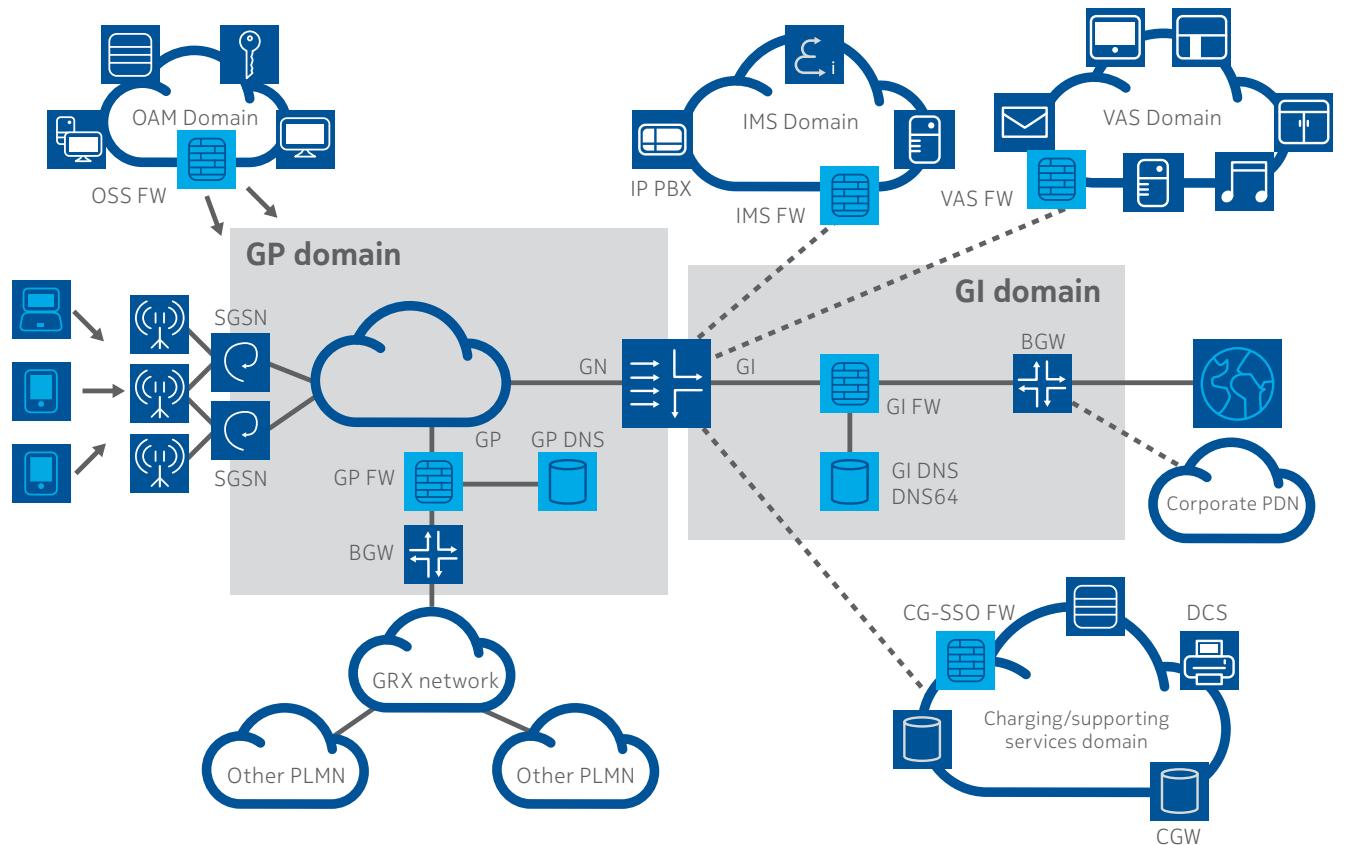


a zapewnienie bezpieczeństwa tego obszaru stanowi proces ciągły i bardzo złożony. Operator zobowiązany jest zapewnić bezpieczeństwo, zarówno systemem funkcjonalnym, minimalizując niebezpieczeństwo przerwy w dostawie usług swoim klientom, jak i poufne dane personalne samych abonentów. Aby w maksymalnie możliwy sposób chronić sieć, instalowane są wszelkiego rodzaju mechanizmy zabezpieczające w postaci systemów detekcji włamania (IDS, Intrusion Detection System), systemów zapobiegania włamaniom (IPS, Intrusion Prevention System), analizatorów pakietów, systemów monitorujących sieć, firewalli, antywirusów czy systemów chroniących przed złośliwym oprogramowaniem (anti-malware). Obecne są także elementy sprawdzające w danych interwałach czasowych bezpieczeństwo sieci. Są to takie elementy jak skanery podatności (vulnerabilities scanners), skanery zabezpieczeń aplikacji webowych (web application scanners) czy skanery sprawdzające aktualną wersję oprogramowania i poziom aktualizacji systemów operacyjnych. Aby w lepszym stopniu zapanować nad złożonością sieci, jest ona

4 Rysunek Zabezpieczenie sieci dostępowej z użyciem IPSec i PKI



5 Rysunek Przykładowy układ sieci mobilnej



dzielona na logiczne obszary, odpowiedzialne za realizację poszczególnych zadań. Takie obszary są dodatkowo separowane elementami zabezpieczającymi. **Rysunek 5** przedstawia przykładowy, koncepcyjny układ sieci mobilnej, z uwzględnieniem podziału na strefy funkcjonalne i użycie niektórych elementów zabezpieczających.

#### Podsumowanie

Dynamiczny rozwój technologii i wprowadzanie nowych rozwiązań w warstwie zarówno sprzętowej (np. stacje bazowe), jak i logicznej (np. wirtualizacja sieci) powodują, że rozwojowi ulegają także metody ataków. Stają się one coraz bardziej wyszukane i zaawansowane. Często nieautoryzowany dostęp do sieci i jej zasobów uzyskiwany jest za pomocą łączenia różnego typu metod i różnych wektorów ataku. Przykładem może być sieć Orange, która w 2014 dwukrotnie padła ofiarą hakerów. W lutym weszli oni w posiadanie danych, zawierających emaile, hasła, adresy i numery telefonów 800 tysięcy abonentów sieci. Atak ponowiono w maju, a w jego rezultacie hake-

rzy uzyskali dostęp do kolejnych poufnych danych dotyczących tym razem 1,3 mln klientów sieci.

Sieć mobilna stanowi jeden z najbardziej złożonych rodzajów sieci, w której odpowiedzialność za bezpieczeństwo spada przede wszystkim na operatorów, twórców aplikacji mobilnych i dostawców, odpowiedzialnych za stworzenie i dostarczanie sprzętu i oprogramowania poszczególnych komponentów infrastruktury telekomunikacyjnej. Firma Nokia, będąc odpowiedzialna za rozwój owej infrastruktury, bierze aktywny udział w rozwoju metod zabezpieczania sieci mobilnych na każdym poziomie. Wobec rosnącej liczby złożonych ataków rośnie potrzeba coraz bardziej wyrafinowanych metod obrony. Badania i prace nad zaawansowanymi, inteligentnymi systemami wykrywania zagrożeń, korelacja zdarzeń związanych z bezpieczeństwem czy wdrażanie systemów ochronnych na poziomie zarówno aplikacji, jak i samej sieci to podstawowe zadania działu Security R&D firmy Nokia.

#### Bibliografia

- [1] Security for mobile broadband: new threads, new opportunities: <http://networks.nokia.com/news-events/event/2014/security-for-mobile-broadband>.
- [2] Radio Access Security: [http://networks.nokia.com/sites/default/files/document/nsn\\_radio\\_access\\_security\\_solution\\_executive\\_summary.pdf](http://networks.nokia.com/sites/default/files/document/nsn_radio_access_security_solution_executive_summary.pdf).
- [3] Mobile Guard: [http://networks.nokia.com/sites/default/files/document/nsn\\_mobile\\_guard\\_executive\\_summary.pdf](http://networks.nokia.com/sites/default/files/document/nsn_mobile_guard_executive_summary.pdf).
- [4] MBB Security: [http://networks.nokia.com/sites/default/files/document/nsn\\_end\\_to\\_end\\_security\\_executive\\_summary.pdf](http://networks.nokia.com/sites/default/files/document/nsn_end_to_end_security_executive_summary.pdf).
- [5] Radio Access Security datasheet: [http://networks.nokia.com/sites/default/files/document/nsn\\_itc\\_security\\_datasheet.pdf](http://networks.nokia.com/sites/default/files/document/nsn_itc_security_datasheet.pdf).
- [6] thehackernews.com: France Telecom Orange Hacked Again, Personal Details of 1.3 Million Customers Stolen, <http://thehackernews.com/2014/05/france-telecom-orange-hacked-again.html>.

#### Autor o swojej pracy

Pracuję w dziale MBB Security R&D. Tworzymy oprogramowanie zarządzające bezpieczeństwem sieci telekomunikacyjnych. Nasza praca łączy się również z tworzeniem zaawansowanych systemów biznesowych, odpowiedzialnych za gromadzenie informacji dotyczących bezpieczeństwa cybernetycznego i konfiguracji urządzeń zabezpieczających. Technologie odpowiedzialne za różne warstwy złożonego oprogramowania zarówno udostępniające interfejs użytkownika (technologie webowe), jak również te, odpowiedzialne za logikę biznesową (technologie enterprise'owe (Java EE), silniki reguł, systemy ekspertowe), a także systemy rozproszone, modelowanie i przechowywanie danych w różnego rodzaju bazach (m.in. PostgreSQL, Cassandra, Redis) to wszystko nasza codzienność.

**Łukasz Gostkowski**  
Software Developer  
MBB Security R&D

# Testowanie systemu LTE w środowisku end-to-end

Piotr Malatyński

R&D Technical Leader, LTE Feature Verification

Feature Verification 1, LTE System Verification Common

**NOKIA**

Nokia jest światowym liderem w dziedzinie rozwiązań i produktów teleinformatycznych oferowanych m.in. operatorom i dostawcom usług LTE. Zanim jednak gotowe rozwiązanie opuści inżynierski warsztat Nokii, a końcowy produkt np. w formie stacji bazowej, zostanie dostarczony klientom, musi ono przejść dobrze zdefiniowany proces, w którym jednym z kroków jest weryfikacja w środowisku E2E (End-to-End).

Weryfikacja w środowisku E2E odnosi się do testowania i ewaluacji jakości systemu końcowego, reprezentowanego przez rzeczywisty produkt przeznaczony do użytku przez klienta. Działem odpowiedzialnym za testy typu E2E produktów LTE wytwarzanych przez Nokię jest System Verification, który znajduje się między innymi we wrocławskim oddziale Nokii.

## Dlaczego w ogóle należy testować oprogramowanie?

Każda osoba, która kiedykolwiek napisała przynajmniej jedną linijkę kodu na pewno zadała sobie pytanie, czy i dlaczego należy testować oprogramowanie. Większość początkujących twórców oprogramowania sprawdza swój program, nie myśląc o tym, że pewne czynności można nazwać testem. Wystarczy przecież skontrolować wartości wejściowe i już w ten sposób przeprowadzany jest prosty test. Ale czy zastanawialiście się jak duże znaczenie ma prawidłowo przeprowadzona kontrola jakości?

Większość z nas, nawet jeśli nie ma nic wspólnego z informatyką zetknęła się z efektami, jakie niesie ze sobą nieprzywiązywanie wag do dokładnego sprawdzenia oprogramowania pod kątem występowania błędów – wszyscy kojarzymy na przykład słynne „blue screeny” z systemów Windows.

Żeby uniknąć takich sytuacji i dostarczyć produkt, który spełni oczekiwania klientów i będzie komfortowy w użyciu oraz odporny na błędy, Nokia przykłada wielką wagę do jakości oprogramowania, które tworzymy. W naszej firmie istnieją działy, których głównym zadaniem jest przeprowadzanie testów, weryfikowanie działania oraz wyszukiwanie błędów w wytwarzanych przez Nokię produktach. Na każdym etapie tworzenia dowolnego produktu istnieją procesy i procedury, które sprawdzają, czy spełnione są kryteria jakościowe. Takie podejście pozwala na zminimalizowanie ryzyka występowania różnych problemów podczas użytkowania dostarczanych produktów.

Nasza firma wychodzi z założenia, że tylko dzięki produktom wysokiej jakości można się wyróżnić na wymagającym rynku nowych technologii jakim jest rynek ICT.

## Proces tworzenia oprogramowania dla systemu LTE w Nokii

W znakomitej większości funkcjonalności stworzone przez Nokię są zestandardyzowane. Co to dokładnie znaczy? Otóż cały system LTE jest opisany w dokumentach zdefiniowanych przez inicjatywę 3GPP (3<sup>rd</sup> Generation Partnership Project), która jednoczyn organizacje zajmujące się standaryzacją technologii używanych w telekomunikacji.

Każdy producent oprogramowania albo sprzętu dla sieci telekomunikacyjnych i systemów ICT musi się stosować do standardów zdefiniowanych przez 3GPP, a sama inicjatywa zapewnia, że systemy i produkty dostarczane przez różnych dostawców będą ze sobą kompatybilne. W przypadku Nokii proces tworzenia oprogramowania dla systemu LTE można przedstawić tak jak na [Rysunku 1](#).

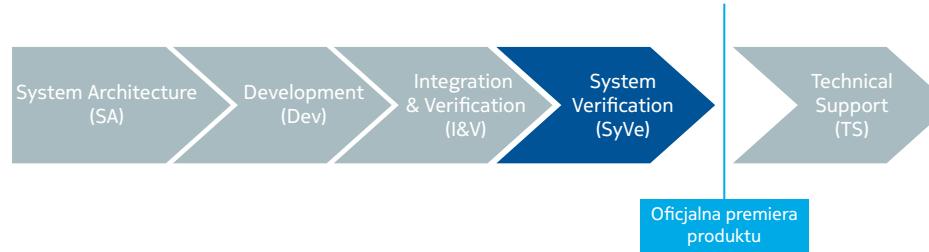
## Proces tworzenia oprogramowania

**System Architecture.** Pierwszym stadium tworzenia oprogramowania jest analiza wymagań stawianych przez 3GPP (lub klienta, w przypadku tworzenia oprogramowania na konkretne zamówienie). Jest to zadanie naszych architektów systemu, którzy przygotowują specyfikację nowej funkcjonalności dopasowaną do wewnętrznych wymagań i procedur firmy.

Już na tym etapie następuje pierwsza faza kontroli jakości oprogramowania (które, de facto jeszcze nie powstało!). Każda specyfikacja musi przejść przez fazę inspekcji, podczas której są wyłapywane potencjalne błędy.

**Development.** Następnie tak przygotowane dokumenty trafiają do programistów, którzy na podstawie specyfikacji tworzą oprogramowanie. Tutaj także następuje sprawdzenie jakości stworzonego kodu (np. testy jednostkowe).

**Integration & Verification.** Kiedy konkretna funkcjonalność jest zaimplementowana i przetestowana przez programistów trafia ona do działu integracji i weryfikacji systemu, gdzie następuje integracja konkretnej części systemu (np. komponentu) z gotowych funkcjonalności i sprawdzenie poprawności jej działania pod kątem



zarówno wymagań stawianych w specyfikacji, jak i odporności na błędy. Wszystko odbywa się na poziomie tzw. entity, czyli jednostki. W przypadku LTE może to być po prostu element eNB (eNodeB).

**System Verification.** Ostatnim etapem produkcji oprogramowania przed wprowadzeniem go na rynek jest jego weryfikacja na poziomie całego systemu, ponieważ w przypadku wcześniejszych testów, weryfikowany był konkretny fragment. Do tej pory wiedzieliśmy, że wprowadzana funkcjonalność pracuje zgodnie z założeniami, a musimy jeszcze sprawdzić, jak zachowią się cały system.

Takie zadanie ma wyznaczone dział weryfikacji systemu (System Verification). W tym przypadku testy mają na celu odwzorowanie zachowania całego, zmodyfikowanego systemu, ze szczególnym naciskiem na scenariusze, w jakich będzie on używany przez końcowych użytkowników. Dlatego weryfikacja działania na tym etapie jest przeprowadzana na „żywym” systemie, bez użycia symulatorów dla brakujących elementów oraz przy wykorzystaniu sprzętu dośćnego komercyjnie. Oczywiście jeśli dana funkcjonalność nie jest jeszcze powszechnie dostępna zachodzi potrzeba użycia urządzeń prototypowych.

Kontrola jakości przeprowadzana przez dział weryfikacji systemu ma kluczowe znaczenie:

- Jest to ostatni etap przed wprowadzeniem produktu na rynek, czyli ostatnia możliwość zidentyfikowania potencjalnych błędów, które obok zaoferowanych funkcjonalności, wpływają najbardziej na zadowolenie klientów.
- Na tym etapie istnieje możliwość spojrzenia na wprowadzone zmiany globalnie z poziomu systemu oraz weryfikacja poprawności współpracy poszczególnych komponentów.

## 2 Rysunek Globalny zasięg Nokii



## Jak wygląda praca w LTE System Verification?

Pracując w LTE System Verification można należeć do jednego z zespołów:

- Feature Verification weryfikujący wybrane funkcjonalności należące do portfolio Nokii;
- Trials and Pilots odpowiedzialny za wsparcie projektów wprowadzających nowe funkcjonalności u klientów przed ich oficjalną premierą rynkową.

Każdy pracownik jest odpowiedzialny za pracę i utrzymanie własnego środowiska testowego, dokładnie takiego samego, jakiego używają nasi klienci. Tak jak pisaliśmy wcześniej, w naszej pracy nie używamy symulatorów, jeśli nie jest to absolutnie konieczne. Każdy używany przez nas element sieciowy pracuje w dokładne takich samych warunkach jak u konkretnych klientów Nokii, niezależnie czy dana stacja bazowa znajduje się w Polsce czy w dowolnie innym miejscu na świecie.

W tym momencie można zadać sobie pytanie – jak to jest możliwe? Odpowiedź jest prosta. Dzięki bliskiej współpracy z zespołami opiekującymi się nasi największymi klientami możemy dokładnie odwzorować daną konfigurację w naszym laboratorium. Wówczas jesteśmy w stanie sprawdzić jak nasz system będzie się zachowywał w realnym środowisku oraz zapewnić jak najwyższą jakość naszego produktu. Dzięki naszej pracy operatorzy świadczący usługi telekomunikacji mobilnej mogą być pewni, że zamówione przez nich rozwiązania będą pracować stabilnie i efektywnie zaraz po aktualizacji do najnowszego oprogramowania.

Wyobraźcie sobie, co mogłyby się stać bez takiego działu jak LTE System Verification. Na przykład nikt nie byłby pewny jak wyglądałaby sieć łącząca miliony ludzi po aktualizacji oprogramowania.

Dzięki temu każdy z naszych pracowników ma dostęp do najnowszego oprogramowania, nie tylko tworzonego przez naszą firmę, ale także może wypróbować jak w rzeczywistości sprawuje się smartfon lub modem jeszcze niedostępny na rynku.

## Środowisko testowe i narzędzia

Każdy pracownik LTE System Verification, poza opieką nad swoją częścią laboratorium, jest odpowiedzialny za przeprowadzanie weryfikacji działania systemu.

Co to znaczy dokładnie?

- Weryfikacja poprawności działania danej funkcjonalności, poprzez sprawdzenie rzeczywistych scenariuszy, które będą używane przez naszych klientów. Za przykład może tu posłużyć sprawdzenie czy aktualizacja oprogramowania nie przynosi niespodziewanych efektów.

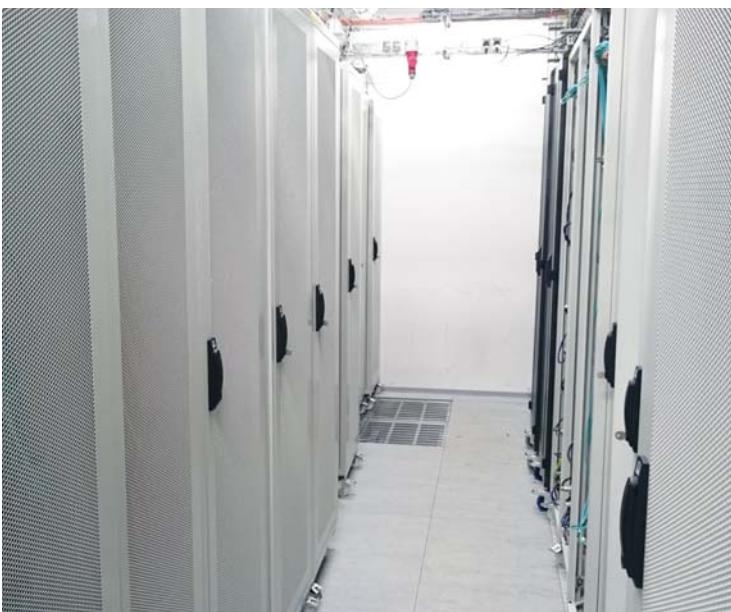
- Szukanie nieoczekiwanych zachowań systemu, próby „złamania” stabilności jego działania. Na przykład przeciążenie buforów nadmierną ilością danych, próby wprowadzania błędnych danych.
- Sprawdzanie nieoczywistych lub rzadko występujących scenariuszy, przykładowo sprawdzenie działania systemu przy niestandardowych ustawieniach.

Specyfikacje do wykonywanych scenariuszy piszą sami pracownicy, bazując na dokumentacji technicznej i doświadczeniu, także jest to bardzo ciekawa i innowacyjna praca, która powoduje, że każdego dnia można nauczyć się czegoś nowego.

Inżynierowie działu LTE System Verification mają do dyspozycji szereg narzędzi, które pozwalają optymalnie wykorzystać czas pracy przy jednoczesnej gwarancji wysoce obiektywnej oceny jakości wytwarzanych produktów.

Po pierwsze, organizacja naszych laboratoriów uwzględnia potrzebę ich ciągłej rozbudowy i modernizacji, a modułowość rozwiązań teleinformatycznych oferowanych przez Nokii i implementacja infrastruktury laboratoryjnej umożliwiają połączenie niemal dowolnych elementów z różnych ich zakątków. W tym celu zaprojektowane zostały odpowiednie panele krosownicze RF (Radio Frequency), które pozwalają między innymi odseparować sygnały radiowe; panele krosownicze światłowodowe, które umożliwiają zestawienie właściwych ścięzek optycznych pomiędzy urządzeniami; a wszystkie urządzenia sieciowe zorganizowane są w ergonomicznych szafach typu rack. Ponadto wdrożona infrastruktura sieciowa oparta jest o najnowsze

## 3 Grafika Wnętrze laboratorium



dostępne na rynku rozwiązania stosowane w sieciach lokalnych, co umożliwia i w wielu przypadkach ułatwia ciągłą pracę zdalną.

Po drugie, weryfikacja typu E2E, która z definicji opiera się na systemie rzeczywistym wymaga stosowania rzeczywistych telefonów i modemów, w szczególności tych, które są komercyjnie dostępne dla użytkowników końcowych. Tym samym w zasobach działu LTE System Verification znajdują się liczne smartfony wiodących światowych producentów, modemy USB obsługujące różne technologie mobilne, w tym LTE, a także różnego typu prototypy, pozwalające na weryfikację nowych funkcjonalności, które w komercyjnych sieciach ruchowych będą wdrażane w przyszłości.

Po trzecie, prawidłowe zestawienie środowiska testowego wymaga zastosowania elementów pozwalających odzwierciedlać tor radiowy z różnymi jego efektami. W tym celu swoje zastosowanie znajdują pudełka ekranujące (ang. shieldboxes), tłumiki programowalne (ang. programmable attenuators) oraz odpowiednio zaprojektowane anteny lub kable RF.

Po czwarte, niezwykle istotną grupą narzędzi są analatory i generatory pakietów, narzędzia monitorujące interfejs radiowy po stronie telefonów czy modemów oraz po stronie stacji bazowej w tym narzędzia pozwalające na podgląd zawartości stosów protokołów implementowanych na interfejsach w dowolnym miejscu architektury systemu LTE. Jako przykład mogą posłużyć analatory warstwy łączącej radiowego i protokołów takich jak RRC, RLC, MAC, analatory sieciowe skonfigurowane dla interfejsów S1, X2, S5, S11 itd., oraz narzędzia do badania wydajności przepływów sieciowych, takie jak generatory ruchu oraz analatory natężeń przepływności w sieci.

Ponadto, do dyspozycji inżynierów LTE System Verification jest szereg aplikacji webowych i sieciowych opartych głównie na wykorzystaniu protokołów telnet, SSH, SCTP, zdalny pulpit, FTP, a także rozwiązania oparte o VPN, VLAN i wirtualizację zasobów. Dodatkowo, wewnętrz zespołu rozwijane są skrypty i aplikacje mające na celu ułatwianie i zwiększanie efektywności naszej pracy.

Weryfikacja działania systemu LTE wiąże się także z koniecznością raportowania wyników testów, a w szczególności zgłoszenia usterek i uczestniczenie w ich naprawie. W tym celu wdrożone zostały odpowiednie narzędzia i procedury, które pozwalają na przejrzyste planowanie i nadzór nad testami, na analizę, planowanie i wykonywanie testów, a także na raportowanie i ocenę kryteriów zakończenia testów. Narzędzia te umożliwiają także śledzenie postępów procesu weryfikacji i pomagają w ocenie jakości testowanego systemu i produktów.

W przypadku odnalezienia usterek należy podjąć odpowiednie kroki, które umożliwiają ich wyeliminowanie, a także zwiększą prawdopodobieństwo uniknięcia podobnych defektów w przyszłości. W związku

z tym konieczne jest tworzenie odpowiednich raportów opisujących odnalezione usterki. Raporty takie mają na celu dostarczenie uczestnikom różnych etapów tworzenia produktu (np. programistom) informacji na temat odnalezionych problemów, aby umożliwić poprawną ich identyfikację oraz naprawę. W skład takiego raportu powinny wchodzić niezbędne szczegóły, które pozwolą na jednoznaczną i zrozumiałą charakterystykę usterki. Do najbardziej podstawowych elementów raportu usterki zalicza się: autora; datę zgłoszenia; opisanie różnic w wynikach oczekiwanych a rzeczywistych; wskazanie na środowisko testowe i np. na konfigurację sprzętową; zdefiniowanie powtarzalności; zdefiniowanie wpływu obecności usterki na działanie całego systemu lub np. na interes klientów czy priorytet sprawy. Do raportu najczęściej dołącza się pliki z odpowiednimi logami, zrzutami z baz danych, można wspomóc się dokumentacją fotograficzną lub audiowizualną itd. tak, aby opisana usterka była w rzeczywistości jednoznacznie identyfikowalna oraz — jeżeli to możliwe — odtwarzalna.

Po odnalezieniu i odpowiednim zgłoszeniu usterki następuje jej naprawa, a następnie wymagana jest ponowna weryfikacja działania danej funkcjonalności w celu potwierdzenia lub odrzucenia poprawki. W taki sposób budowana jest jakość produktu końcowego.

#### Autor o swojej pracy

---

Pracuję w dziale Feature Verification. Mój zespół jest odpowiedzialny za weryfikację systemu LTE w środowisku end-2-end. Główny nacisk kładziemy na część EUTRAN systemu oraz na jego współpracę z oprogramowaniem służącym do zarządzania całą siecią. Dzięki testom, które przeprowadzamy, oprogramowanie dostarczane naszym klientom spełnia najwyższe standardy jakościowe. Ścisłe współpracujemy z programistami tworzącymi oprogramowanie Nokii oraz z zespołami opiekującymi się naszymi klientami na całym świecie. Stosując takie podejście jesteśmy w stanie efektywnie znajdować błędy w naszych produktach przed ich rynkową premierą.

---

#### Piotr Malatyński

R&D Technical Leader, LTE Feature Verification  
Feature Verification 1, LTE System Verification Common

# Czego spodziewać się po systemie 5G?

Agnieszka Szufarska  
Radio Research Manager  
Technology & Innovation; Research; Radio Systems

**NOKIA**

Ogromny wzrost ilości danych wymienianych w sieciach mobilnych oraz jednocześnie rosnące wymagania użytkowników tychże sieci sprawiają, że pomimo sukcesu systemu LTE już teraz zaczęto zastanawiać się nad technologiami, które nadziejają po nim. Światowa społeczność zajmująca się telekomunikacją ruchomą już od kilku lat prowadzi badania nad możliwymi nowymi zastosowaniami komunikacji bezprzewodowej, wymaganiami stawianymi nowym systemom czy wreszcie rozwiązaniami technicznymi, które pozwolą spełnić te wymagania.

## Różnorodność zastosowań i wymagań

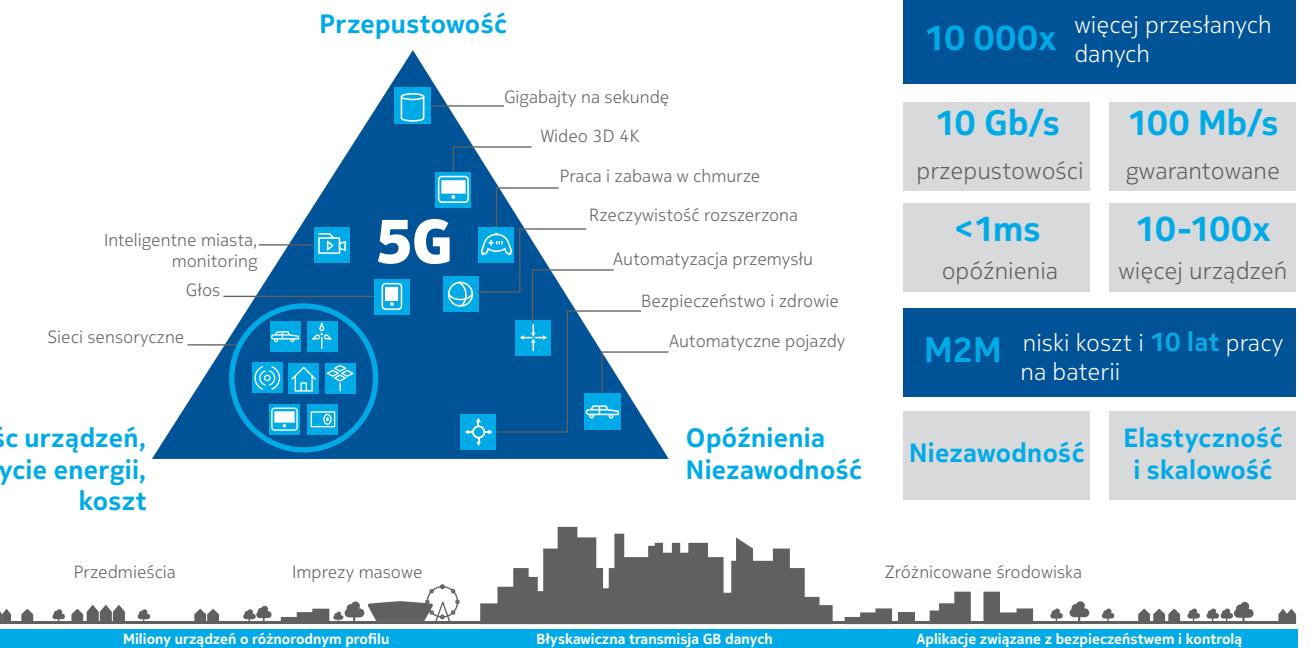
W przeszłości telefony komórkowe używane były głównie do wykonywania połączeń głosowych. Później pojawiły się również wiadomości tekstowe. Obecnie urządzenia mobilne tj. smartfony czy tablety są jednym z podstawowych kanałów dostępu do bezprzewodowego Internetu czy rozmów wideo. W przyszłości tych zastosowań będzie jeszcze więcej.

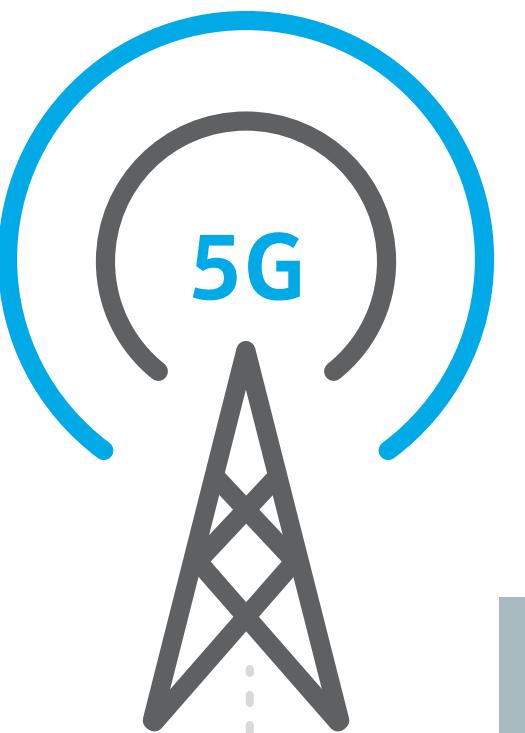
Ponadto, wzrasta znaczenie bezpośredniej komunikacji pomiędzy urządzeniami (device-to-device, D2D) i ogólnie ilość i różnorodność urządzeń podłączonych do sieci bezprzewodowych (Internet of Things, IoT). Wiąże się to z pojawianiem się nowych zastosowań sieci komórkowych.

Tak duża ilość i zróżnicowanie zastosowań sieci 5G znajduje odzwierciedlenie w różnorodności – czasem sprzecznych – wymagań stawianych sieciom radiowym. Na poniżej ilustracji widać, że niektóre usługi wymagają dużych przepływności (jak na przykład wideokonferencje w wysokiej rozdzielcości), inne potrzebują opóźnień w sieci radiowej niższych niż 1 milisekunda (zdalne sterowanie np. dostarczającym paczki dronem) czy jak najniższego kosztu i poboru mocy w przypadku mnóstwa drobnych sensorów w sprzętach, pojazdach czy ubraniach.

Jednoczesne spełnienie wszystkich tych wymagań może być bardzo trudne, na pewno nie jest ekonomicznie opłacalne, a co więcej nie jest nawet konieczne – żadna z poszczególnych usług nie wymaga maksymalnych wartości wszystkich parametrów. W związku z tym bardzo ważną cechą systemów piątej generacji jest ich łatwa skalowalność pomiędzy różnymi rodzajami usług i stawianych im wymagań.

W zależności od wymagań jakościowych (quality of service, QoS) poszczególnych usług i aplikacji użytkownicy mogą być połączeni przy pomocy różnych technik dostępu do łączysko radiowego (Radio Access Technology, RAT) – na przykład UMTS/HSPA, LTE (w tym LTE-Advanced lub przystosowane specjalnie do komunikacji z maszynami LTE-M), Wi-Fi czy nowych technik (RAT) przygotowanych specjalnie z myślą o 5G.





## Nowa generacja sieci dużego zasięgu

Niezawodność i jednorodne doznanie zawsze i wszędzie

## Bardzo gęste sieci komórkowe

Minimalne opóźnienia i gigabajtowa przepustowość zawsze gdy potrzeba

Istniejące oraz nowe technologie zintegrowane i wzajemnie się dopełniające

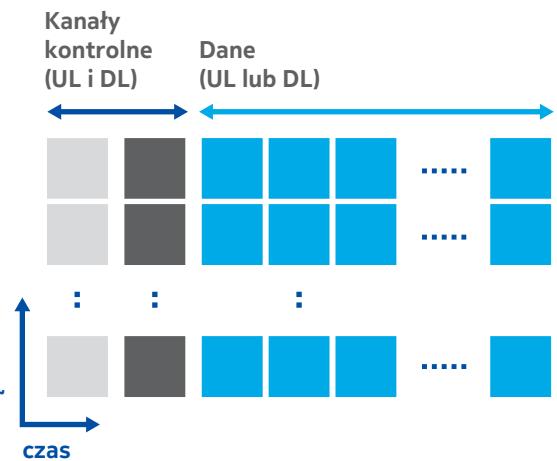


Nowe elementy systemu piątej generacji (oznaczone na powyższym rysunku kolorem ciemnoniebieskim) i innowacyjna architektura sieci łącząca w optymalny sposób wszystkie technologie są obecnie głównymi obszarami badań w firmie Nokia.

### **Nowe rozwiązania techniczne**

Przykładem rozwiązania technicznego, które pozwala na osiągnięcie bardzo niskich opóźnień transmisji radiowych (niższych nawet niż 1 milisekunda, podczas gdy najniższe opóźnienia LTE to 10 ms) a jednocześnie łatwego dostosowania konfiguracji łącza do wymagań użytkownika jest nowa struktura ramki do komunikacji w łączu radiowym. W połączeniu z nowym sposobem transmisji danych dynamic TDD umożliwia ona bardziej elastyczne wykorzystanie dostępnych zasobów radiowych.

W przypadku transmisji pomiędzy gęsto rozmieszczonymi węzłami małej mocy w paśmie tzw. fal centymetrowych (3-30 GHz) pojedyncza ramka dynamic TDD może trwać 0,25 ms. Ramka taka zawiera zawsze zarówno kanały kontrolne (dla łączu w górę (UL) i w dół (DL)) jak i miejsce na transmisję danych.



Od typowych rozwiązań stosowanych obecnie odróżnia się taka ramka tym, że decyzja o rodzaju transmisji w kanale danych zapada na bieżąco zależnie od rodzaju/priorytetu transmisji do wykonania czy stanu zapełnienia buforów nadawczych. Oprócz zastosowania takiego systemu transmisji dla danych w obu kierunkach (UL i DL) opisana powyżej ramka może także być stosowana do transmisji pomiędzy sąsiednimi węzłami sieci (self backhauling) czy terminalami (device-to-device).

Zastosowanie w transmisji TDD krótkich ramek radiowych i zmiennego przeznaczenia kanału danych oznacza, że system szybko jest w stanie dostosować się do chwilowych warunków w sieci i zasoby radiowe wykorzystane mogą być lepiej wykorzystane. Technika ta jest jednym z rozwiązań, które rozważane są dla systemu 5G – inne

to na przykład rozwój transmisji z użyciem wielu elementów antenowych (Massive MIMO), która zwiększa efektywność widmową czy znacznie zwiększającą dostępne zasoby częstotliwościowe transmisja w zakresie fal milimetrowych.

Te oraz inne techniki są obecnie jedynie kandydatami do zastosowania w nowym systemie – standaryzacja systemu 5G jeszcze się nie rozpoczęła. Obecnie w ITU-R (sekcji radiokomunikacyjnej Międzynarodowego Związku Telekomunikacyjnego) trwają prace nad specyfikacją scenariuszy i wymagań dla IMT-2020, ale jeszcze przez najbliższe kilka lat dokładny sposób działania sieci piątej generacji nie będzie znany. Można jedynie opisywać efekt ich spodziewanego działania.

Po roku 2020 lawinowo wzrośnie ilość urządzeń połączonych z siecią, ilość i różnorodność sposobów komunikacji, ale pojawią się także techniki dokładnie i efektywnie odpowiadające na te potrzeby. Sprawi to, że ograniczenia systemów komunikacji bezprzewodowej jak na przykład zbyt mała przepustowość łączą nie będą blokować ogólnego rozwoju technologii, ekonomii czy społeczeństwa.

### **Autor o swojej pracy**

Pracuję w dziale zajmującym się badaniem najnowszych trendów i rozwiązań technologicznych w sieciach radiowych: Radio Systems Research. Opracowujemy koncepcje sposobu w jaki powinny działać obecne i przyszłe systemy telekomunikacji ruchomej oraz tworzymy symulatory (programy komputerowe w C++ i Matlabie modelujące działanie sieci) w których sprawdzamy stworzone koncepcje. Dla dobrych pomysłów zgłaszamy wnioski patentowe, wyniki badań publikujemy na konferencjach naukowych i prezentujemy naszym klientom. Pracujemy nad definiowaniem najnowszych technologii oraz nad systemami, które będą rozwijane jeszcze przez wiele lat.

**Agnieszka Szufarska**  
Radio Research Manager  
Technology & Innovation; Research; Radio Systems

Od pierwszego na świecie połączenia w sieci GSM, po pierwsze połączenie w sieci LTE

Nokia jako globalny lider rozwiązań telekomunikacyjnych inwestuje w technologie umożliwiające komunikowanie się miliardów urządzeń na całym świecie. Działalność firmy skupia się na trzech obszarach biznesowych:

## Nokia Networks

odpowiedzialna za oprogramowanie, sprzęt oraz usługi rozwijające infrastrukturę sieciową

## Nokia Technologies

rozwija zaawansowane technologie i ich licencjonowanie

## HERE

rozwija inteligentne systemy lokalizacyjne (m.in. mapy)

Każdy z wymienionych wyżej działów jest liderem w swojej dziedzinie. W Polsce w trzech miastach (Wrocław, Kraków i Warszawa) prowadzone są prace nad mobilną technologią transmisji szerokopasmowej (w ramach działu Nokia Networks).

## Nokia we Wrocławiu

Europejskie Centrum Oprogramowania i Inżynierii Nokia Networks we Wrocławiu powstało w 2000 roku i z roku na rok powiększa grono swoich specjalistów.

Dzisiaj jest to największa instytucja badawczo-rozwojowa w sektorze ICT w Polsce i jedna z największych na świecie.

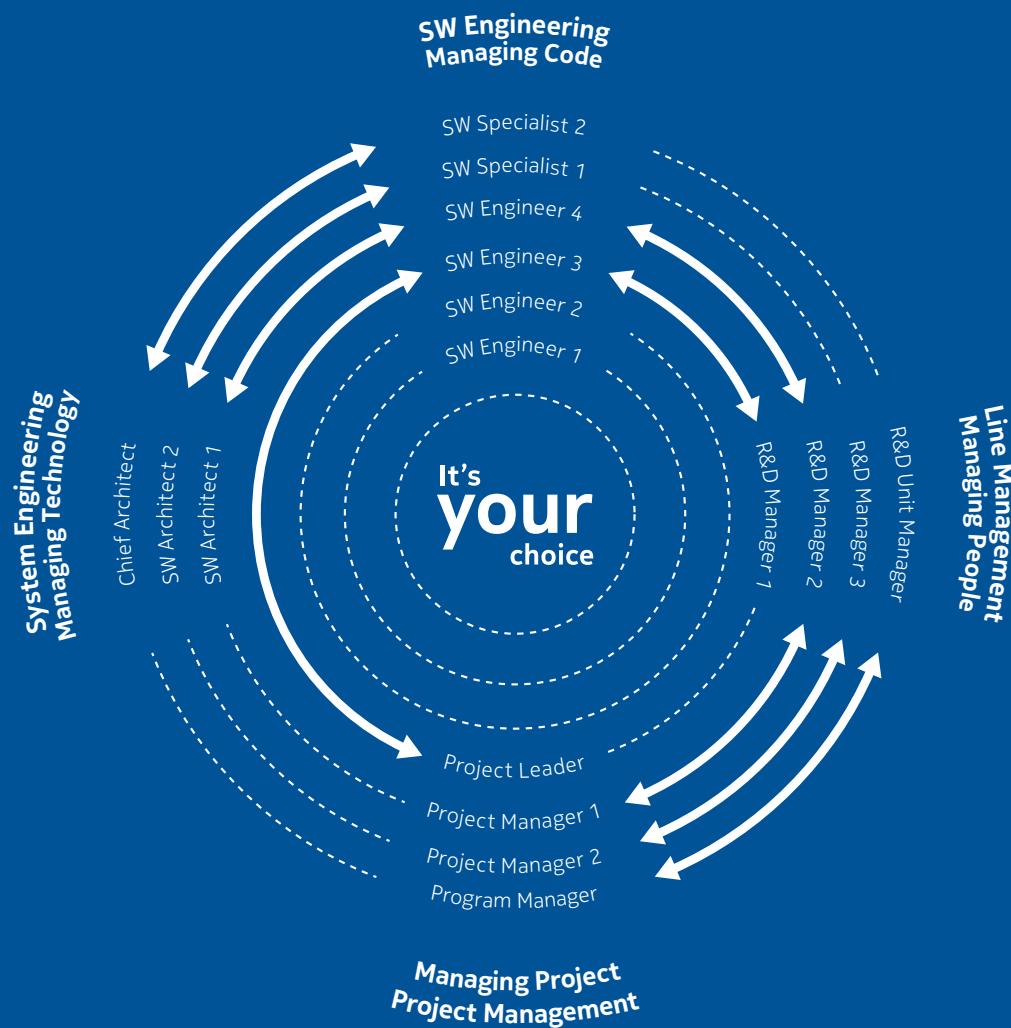
Specjaliści z Nokia Networks uczestniczą zarówno w procesie wytwarzania produktów, jak i rozwoju technologii, począwszy od konceptacji aż po procesy wdrożenia. Jako praktycy z wieloletnim stażem zdajemy sobie sprawę, że rozwój pracowników branży IT jest najważniejszy i dlatego stawiamy przed nimi ambitne zadania. Nasi programiści, testerzy i weryfikatorzy mierzą się z takim projektami jak: budowa szerokopasmowej autostrady Internetu LTE, standaryzacja technologii 5G czy tworzenie w pełni zautomatyzowanego środowiska do budowania i testowania oprogramowania dla stacji bazowych. Każdego roku wrocławski oddział zgłasza ponad 1000 patentów telekomunikacyjnych. Jesteśmy dumni z faktu, że z naszych rozwiązań codziennie korzysta ponad 2 miliardy ludzi na całym świecie!

# Rozwój w Nokii

Różnorodność projektów prowadzonych we wrocławskiej placówce daje możliwość rozwoju personalnego w kilku ciekawych dziedzinach związanych z telekomunikacją. Każdy nowo zatrudniony pracownik, na początku swojej ścieżki rozwoju, wybiera określoną ścieżkę rozwoju na podstawie swoich zainteresowań, kompetencji oraz wcześniejszych doświadczeń zawodowych.

Aktualne informacje o możliwościach rozwoju znajdziesz na naszej stronie w zakładce KARIERA.

<http://nokiawroclaw.pl/>





**ul. Strzegomska 36**  
53-611 Wrocław

Recepja I p., Green Towers B

Telefon na recepcję: +48 71 777 38 00  
Fax: +48 71 777 30 30  
E-mail: kontakt@nokiawroclaw.pl

[www.nokiawroclaw.pl](http://www.nokiawroclaw.pl)

**Plac Generała J. Bema 2**  
50-265 Wrocław

Recepja wejście A, 4 p.

Telefon na recepcję: +48 71 777 41 30  
Fax: +48 71 777 41 98