

O ZALETACH PROTOTYPÓW, CZYLI ZWINNY SPOSÓB NA DOKUMENTACJĘ

Index: 285358

[www.programistamag.pl](http://www.programistamag.pl)

Magazyn programistów i liderów zespołów IT



11/2017 (66)

grudzień/styczeń

Cena 23,90 zł (w tym VAT 5%)

**nowy STANDARD**

## BARIERY PAMIĘCI

ISSN 2084-9400



11

9 772084 940701

Źródło: fotolia.com

Komponenty – podstawowa  
składowa współczesnych UI

Testowanie aplikacji  
rozproszonych

Google Architecture  
Components

Praktyczna  
steganografia



Join our team of Makers!

[career.cybercom.com](http://career.cybercom.com)

**Change  
tomorrow  
with us**

[www.cybercom.pl](http://www.cybercom.pl)



Cybercom Poland Sp. z o.o.

ul. Hrubieszowska 2, 01-209 Warszawa / ul. Składowa 35, 90-127 Łódź / ul. Unii Lubelskiej 4c, 85-059 Bydgoszcz

## Testowanie, C++17 i steganografia – koktajl idealny?

Jednym z największych problemów, z jakimi borykają się projektanci najnowocześniejszych komputerów, jest różnica pomiędzy szybkością działania procesora i pamięci operacyjnej. Walka o wydajność toczy się na wszystkich możliwych frontach – procesory starają się scalać instrukcje i przestawiać je w taki sposób, aby oczekiwanie na dostęp do danych był jak najmniej dolegliwy. Jednostka wykonawcza stara się również przewidzieć, które dane będą wykorzystane w najbliższej przyszłości, a później zbuforować je do szybkiej pamięci podręcznej procesora (tzw. cache).

O wydajność powalczyć może również programista, ale do tego konieczne jest zrozumienie mechanizmów znajdujących się w CPU. Czym są bariery pamięci i jak można je zastosować? To zagadnienie skrupulatnie opisał w swoim artykule Tomasz Widenka.

Zmieniając temat: wygodę programistów C++ nieustannie zwiększa komitet standaryzacyjny tego języka, który od 2011 roku prowadzi niezwykle intensywne działania. Niedawno opracowana została ostateczna koncepcja C++17, która zgodnie ze zwyczajem opisuje Paweł "KrzaQ" Zakrzewski. Jego artykuł stanowi obszerny przegląd ciekawostek, takich jak optional, polityki wykonywania algorytmów, statyczne bloki warunkowe rozwijywane w trakcie komplikacji i inne.

Jednym z najtrudniejszych zagadnień, z jakimi przyjdzie zmierzyć się testerowi, jest sprawdzenie poprawności działania aplikacji rozproszonych. O złożoności tego tematu, problemach możliwych do napotkania przeczytacie w artykule Sławomira Zborowskiego.

Natomiast o praktycznych zastosowaniach i implementacji steganografii (a także jej krótkim szkicu historycznym) poczytać można w artykule Macieja Szymkowskiego oraz Pawła Czochańskiego.

Na koniec zapraszamy do lektury o podstawowych elementach składowych współczesnego UI, czyli komponentach, autorstwa Przemysława Smyrdka.

Michał Leszczyński, Mariusz Witkowski

Życzymy wszystkiego dobrego w 2018 roku!

## BIBLIOTEKI I NARZĘDZIA

### Google Architecture Components

Wojciech Sura

4

## JĘZYKI PROGRAMOWANIA

### C++17 - nowy, miłościcie panujący nam standard C++

Paweł "KrzaQ" Zakrzewski

20

## PROGRAMOWANIE SYSTEMOWE

### Bariery pamięci

Tomasz Widenka

34

## PROGRAMOWANIE APLIKACJI WEBOWYCH

### Komponenty – podstawowa składowa współczesnych UI

Przemysław Smyrdek

40

### Poznaj Magento 2

Mateusz Borowik, Marek Milewski

44

## PRZETWARZANIE RÓWNOLEGŁE I ROZPROSZONE

### Testowanie aplikacji rozproszonych

Sławomir Zborowski

48

## ALGORYTMIKA

### Złożoność obliczeniowa algorytmów – ciąg dalszy (algorytmy rekurencyjne)

Marek Żukowicz

56

## ZARZĄDZANIE PROJEKTAMI

### O zaletach prototypów, czyli zwinny sposób na dokumentację

Katarzyna Małecka

64

## LABORATORIUM SYMMETRA

### Być jak profesor Moriarty. Praktyczna steganografia

Maciej Szymkowski, Paweł Jan Czochański

68

## KLUB DOBREJ KSIĄŻKI

### Opus Magnum C++11. Programowanie w języku C++

Marek Sawerwain

74

redakcja

Zamów prenumeratę magazynu Programista

przez formularz na stronie:

<http://programistamag.pl/typy-prenumeraty/>

lub zrealizuj ją na podstawie faktury Pro-forma. W spawie faktur Pro-forma prosimy kontaktować się z nami drogą mailową:  
[redakcja@programistamag.pl](mailto:redakcja@programistamag.pl).

Prenumerata realizowana jest także przez RUCH S.A.

Zamówienia można składać bezpośrednio na stronie: [www.prenumerata.ruch.com.pl](http://www.prenumerata.ruch.com.pl)

Pytania prosimy kierować na adres e-mail: [prenumerata@ruch.com.pl](mailto:prenumerata@ruch.com.pl)

lub kontaktując się telefonicznie z numerem:

801 800 803 lub 22 717 59 59, godz. 7:00 – 18:00 (koszt połączenia wg taryfy operatora).

Magazyn Programista wydawany jest przez Dom Wydawniczy Anna Adamczyk

**Wydawca/Redaktor naczelny:** Anna Adamczyk ([annaadamczyk@programistamag.pl](mailto:annaadamczyk@programistamag.pl)).

**Redaktor prowadzący:** Michał Leszczyński ([mleszczyński@programistamag.pl](mailto:mleszczyński@programistamag.pl)).

**Korekta:** Tomasz Łopuszański. **Kierownik produkcji:** Havok. **DTP:** Havok.

**Dział reklamy:** [reklama@programistamag.pl](mailto:reklama@programistamag.pl), tel. +48 663 220 102, tel. +48 604 312 716.

**Prenumerata:** [prenumerata@programistamag.pl](mailto:prenumerata@programistamag.pl).

**Współpraca:** Michał Bartylewski, Mariusz Sierakiewicz, Dawid Kaliszewski, Marek Sawerwain, Łukasz Mazur, Łukasz Łopuszański, Jacek Matulewski, Sławomir Sobótka, Dawid Borycki, Gynvael Coldwind, Bartosz Chrabski, Rafał Kocisz, Michał Sajdak, Michał Bentkowski, Mariusz „maryush” Witkowski, Paweł „KrzaQ” Zakrzewski.

**Adres wydawcy:** Dereniowa 4/47, 02-776 Warszawa.

**Druk:** <http://www.moduss.waw.pl/>, Nakład: 4500 egz.

Nota prawa

Redakcja zastrzega sobie prawo do skrótów i opracowania tekstu oraz do zmian planów wydawniczych, tj. zmian w zapowiadanych tematach artykułów i terminach publikacji, a także nakładzie i objętości czasopisma.

O ile nie zaznaczono inaczej, wszelkie prawa do materiałów i znaków towarowych/firmowych zamieszczanych na łamach magazynu Programista są zastrzeżone. Kopiowanie i rozpowszechnianie ich bez zezwolenia jest zabronione.

Redakcja magazynu Programista nie ponosi odpowiedzialności za szkody bezpośrednie i pośrednie, jak również za inne straty i wydatki poniesione w związku z wykorzystaniem informacji prezentowanych na łamach magazynu Programista.

# Google Architecture Components

Google na konferencji I/O'17 zaprezentowało nową bibliotekę narzędziową dla Androida – Architecture Components. Ułatwia ona obsługę czasu życia komponentów aplikacji (aktywności, fragmentów), pomaga w utrzymywaniu czytelnej architektury MVVM, ale jednocześnie jest bardzo modularna – każdego z jej elementów możemy użyć pojedynczo, w miarę potrzeb. W ramach Architecture Components debiutuje również nowy ORM dla Androida o nazwie Room. Zobaczmy, co tym razem Google chce zaoferować programistom aplikacji mobilnych.

## ZANIM ZACZNIEMY

Każdy, kto z programowania desktopowego albo webowego przeniesie się do środowiska aplikacji mobilnych, z pewnością przeżyje dosyć duży szok. O tanie i łatwo do tej pory dostępne zasoby – takie jak czas procesora czy pamięć – trzeba zaciekle walczyć, interfejs użytkownika musi zostać zaprojektowany w zupełnie inny sposób, różnorodność docelowych urządzeń może przyprawić o zawrót głowy, a złożoność cyklu życia i nieprzewidywalność czasu życia komponentów aplikacji zmusza do poszukiwania zupełnie nowego rodzaju błędów.

Część wspomnianych cech wynika bezpośrednio ze specyfiki środowiska. Smartfony mają coraz mocniejsze podzespoły, ale przekłada się to na zwiększone zapotrzebowanie na energię: w większości przypadków akumulatory są w stanie zasilić urządzenie średnio przez jeden, góra dwa dni normalnego użytkowania. Aby przedłużyć czas pracy urządzenia, trzeba więc zacząć oszczędzać – głównie poprzez ograniczenie pracy tych komponentów, które pożerają najwięcej prądu, oczywiście na czele z procesorem. To z kolei pociąga za sobą dalsze konsekwencje – na przykład system operacyjny nie wahając się za bardzo zwalniać aktywności, które nie są w danym momencie na wierzchu, aby nie zajmowały cennych zasobów. Przy życiu może pozostać niewizualna część aplikacji (na przykład klasa aplikacji i jej zależności) oraz te jej części, które zostały *explicitly* oznaczone jako długo żyjące, na przykład serwisy czy broadcast receivers. No i niestety to na programie spoczywa większa część odpowiedzialności za to, żeby aplikacja działała sprawnie nawet w tak nieprzewidywalnym środowisku.

Niektoře z utrudnień pojawiających się podczas pisania aplikacji mobilnych mają jednak zupełnie inne korzenie. Na przykład dużym problemem w Androidzie przez długi czas była fragmentacja – obecność na rynku urządzeń z mocno różniącymi się wersjami oprogramowania. O ile więc Apple musi dbać zaledwie o kilkanaście modeli telefonów (efektywnie znacznie mniej, bo część zakończyła już swój cykl aktualizacji), to Androida mają na pokładzie setki, jeżeli nie tysiące różnych urządzeń. W pewnym momencie coraz trudniej było napisać aplikację, która bez większych modyfikacji pracowała na większości nich.

Żeby tego było mało, po jakimś czasie obecności Androida na rynku zaczęły uwidaczniać się, a potem coraz bardziej przeszkaźdać niektóre z pierwszych decyzji projektowych. Na przykład komunikacja z widokami (komponentami wizualnymi) wymagała pisania dużych ilości kodu boilerplate – czyli kodu-kleju, który nie wnosił nic nowego do projektu, a trzeba było go po prostu za każdym razem bezmyślnie napisać.

Na szczęście Google nie zostawiło programistów samym sobie. Od czasu do czasu – przeważnie na konferencjach z serii I/O – firma ta prezentowała kolejne biblioteki będące odpowiedzią na coraz bardziej palące problemy ekosystemu. Jedną z pierwszych była biblioteka Android Support Library, która w dużym stopniu niwelowała problem kompatybilności wstępnej. Widoki i klasy z „compat” w nazwie korzystają z istniejącego API na nowszych wersjach Androida oraz zapewniają funkcjonalność zastępczą, gdy system takiej nie dostarcza. Z ważniejszych bibliotek nie sposób nie wspomnieć oczywiście o Data Binding Library, która zakończyła zmorę pisania niezliczoną liczbą razy `findViewById`. Znacznie uprościła ona również projektowanie aplikacji androidowych w architekturze MVVM, wprowadzając funkcjonalność wiązania (binding) dostępną obecnie w bodaj każdym szanującym się framework'u – od desktopu po web.

Teraz – w postaci Android Architecture Components – przyszła pora na ułatwienie programistom życia w kolejnej kwestii: zarządzania czasem życia komponentów androidowej aplikacji.

## ACTIVITY LIFECYCLE

Cykł życia aktywności, czyli pojedynczego, kompletnego widoku androidowej aplikacji, jest dosyć skomplikowany. Założymy na przykład, że użytkownik jest w trakcie pracy z naszą aplikacją. W pewnym momencie ktoś do niego dzwoni, więc aplikacja telefonu natychmiast otrzymuje priorytet i odsyła naszą na dalszy plan. I teraz może zdarzyć się kilka rzeczy – w zależności od urządzenia, z jakiego korzystamy, niektórych cech naszej aplikacji i wielu innych czynników. W pierwszym przypadku cała aplikacja wraz z jej aktywnościami pozostanie przy życiu i użytkownik po zakończonym połączeniu będzie mógł do niej bezpośrednio wrócić. Jeżeli telefon nie ma dostatecznie dużo wolnej pamięci operacyjnej, aplikacja może pozostać przy życiu, ale bez aktywności – zwolnionych, by odzyskać część potrzebnych w danym momencie zasobów. W skrajnym przypadku cała aplikacja może zostać zatrzymana i będzie musiała później zostać uruchomiona ponownie – tak dzieje się często w przypadku tych z nich, które pożerają wyjątkowo dużo zasobów, na przykład gier.

Aplikacja telefonu – jako wywieszająca tę aktywną – nie jest jedynym przypadkiem nagłego zatrzymania i zwolnienia aktywności. Dzieje się tak również wówczas, gdy użytkownik obróci telefon z pozycji portretowej do panoramicznej – wtedy cała aktywność jest zwalniana, a potem tworzona na nowo (potencjalnie z użyciem osobnego layoutu).

# Big Data Technology Warsaw Summit 2018

**FEBRUARY 22nd 2018**  
**AIRPORT HOTEL OKĘCIE, WARSAW**

We invite you all to join us at the conference - Big Data Tech Warsaw Summit 2018. It's an exciting conference with purely technical talks in the field of Big Data analysis, scalability, storage and search. The conference is unique in the whole CEE! More than 500 Big Data professionals will join the 4th edition. We look forward to see you there!

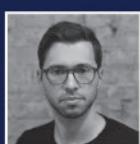
## What to expect?

- 4 simultaneous sessions
- a lot of interesting case studies
- 3 workshops
- roundtables session

**Independent Big Data conference  
with purely technical presentations**



## WELL-KNOWN SPEAKERS:



**Nicola Bortignon**  
Director of Engineering & Science,  
Spotify



**Jon Bratseth**  
Distinguished architect,  
Yahoo!



**Karthik Ramasamy**  
Data Science Manager,  
Uber



**Roman Studenikin**  
Software Developer,  
Booking.com



**Boxun Zhang**  
Sr. Data Scientist,  
GoEuro

## Full day workshops (day before conference):

- Introduction to Big Data Technologies  
Piotr Krewski, *GetInData*
- Real-Time Stream Processing  
Krzysztof Zarzycki, *GetInData*; Dawid Wysakowicz, *GetInData*
- Large scale text mining with Apache Spark  
Rafał Prókopi, *YND*; Tomasz Żukowski, *GetInData*

**Sign up now!**

Join till January 20th!

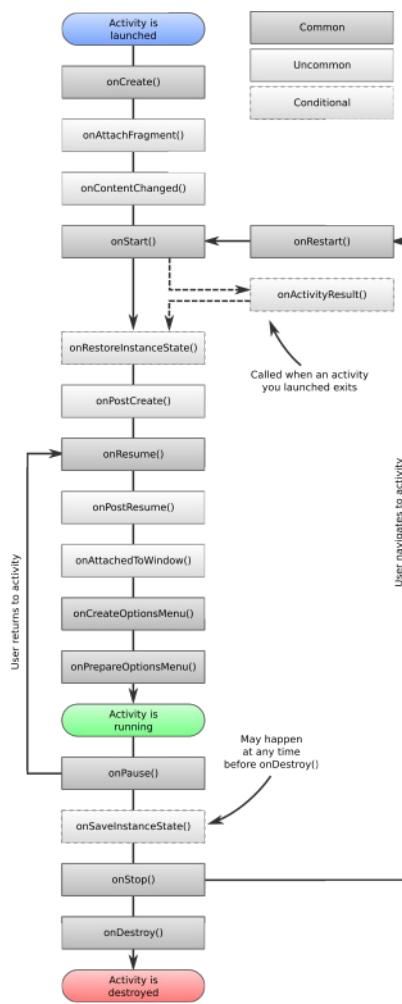
Use special promotional code "Programista"  
and participate in:

- BDTWS conference  
for 899 pln (210 euro)
- BDTWS conference  
and workshops of choice  
for 1999 pln (470 euro)

[www.bigdatatechwarsaw.eu/registration/](http://www.bigdatatechwarsaw.eu/registration/)

# BIBLIOTEKI I NARZĘDZIA

Kompletny cykl życia aplikacji możemy zobaczyć na Rysunku 1. Jego złożoność powoduje, że jest to środowisko bardzo podatne na wycieki pamięci. Wystarczy bowiem jeden zapomniany listener przypięty do jakiegoś widoku, by przytrzymać przy życiu całą aktywność, uniemożliwiając jej sprzątanie przez *Garbage Collector*. Duży problem mogą stanowić również wyścigi. Wyobraźmy sobie na przykład wykonanie zapytania przez HTTP – z uwagi na spore opóźnienia czasowe zapytania takie muszą być realizowane asynchronicznie w osobnym wątku. W momencie jednak, gdy przychodzi odpowiedź od serwera, nie mamy żadnej gwarancji, że aktywność wciąż istnieje – mogła w międzyczasie zostać zniszczona. Ale to jeszcze mało. Po obróceniu ekranu aktywność jest niszczona i tworzona ponownie. Jeżeli użytkownik zdążył w międzyczasie wprowadzić do pól jakieś wartości, znikną one po restartie aktywności. Programista musi więc zadbać o to, żeby sytuację taką prawidłowo obsługiwać i przywrócić w odpowiednim momencie zawarte w widokach dane.



Rysunek 1. Cykl życia aktywności (źródło: <https://github.com/xxv/android-lifecycle>)

Właśnie takim i wielu innym problemom próbują zaradzić składniki biblioteki Android Architecture Components. Przyjrzymy im się teraz trochę bliżej.

## DOŁĄCZENIE BIBLIOTEKI DO PROJEKTU

Zanim zacznę omawiać poszczególne elementy biblioteki, dowiedzmy się najpierw, jak dołączyć ją do projektu.

Pierwszym krokiem jest dodanie repozytorium do pliku *build.gradle* dla całego projektu. Należy wprowadzić tam następującą linijkę:

### Listing 1. Modyfikacja build.gradle dla projektu

```
allprojects {  
    repositories {  
        jcenter()  
        maven { url 'https://maven.google.com' }  
    }  
}
```

Teraz możemy dodawać następujące wpisy do plików *build.gradle* poszczególnych modułów lub aplikacji, w zależności od tego, których składników Architecture Components chcemy użyć:

### Listing 2. Modyfikacje build.gradle modułów i aplikacji

```
dependencies {  
    // (...)  
    // Aby skorzystać z Lifecycle:  
    implementation "android.arch.lifecycle:runtime:1.0.3"  
    annotationProcessor "android.arch.lifecycle:compiler:1.0.0"  
  
    // Aby skorzystać z LiveData i ViewModel:  
    implementation "android.arch.lifecycle:extensions:1.0.0"  
  
    // Aby skorzystać z Room:  
    implementation "android.arch.persistence.room:runtime:1.0.0"  
    annotationProcessor  
        "android.arch.persistence.room:compiler:1.0.0"  
  
    // Aby skorzystać z Paging:  
    implementation "android.arch.paging:runtime:1.0.0-alpha3"  
    // (...)  
}
```

## LIFECYCLE

Złożony cykl życia androidowej aktywności i fragmentu to tylko jeden z problemów, na które natykamy się podczas projektowania aplikacji. Drugim z nich – z pewnością utrudniającym utrzymanie czytelnej architektury – jest wysoki poziom decentralizowania obsługi zmian stanów cyklu życia. Mam tu na myśli fakt, że pierwszym – i jedynym – obiektem, który dowiaduje się o jakiejś zmianie w cyklu życia, jest on sam. Zarówno w aktywności, jak i we fragmentach możemy bowiem zaimplementować wirtualne metody, takie jak *onStart*, *onStop*, *onResume* czy *onPause*. Jeżeli jednak ktokolwiek inny – na przykład jakiś serwis – jest również zainteresowany obsługą takiej zmiany, musimy już powiadomić go ręcznie. Powoduje to, że część obsługi takiego serwisu musi w pewnym sensie zostać przeniesiona do klasy aktywności, a to znaczco zmniejsza czytelność kodu. Aby wyjść temu problemowi naprzeciw, Google w nowej bibliotece wprowadza klasę o nazwie *Lifecycle*, stanowiącą abstrakcję cyklu życia komponentu androidowej aplikacji.

Klasa *Lifecycle* wprowadza dwa ważne typy wyliczeniowe. Pierwszym z nich jest *State*, który opisuje stan cyklu życia aktywności:

### Listing 3. Typ Lifecycle.State

```
public enum State {  
    DESTROYED,  
    INITIALIZED,  
    CREATED,  
    STARTED,  
    RESUMED  
}
```

Drugim, niejako bliźniaczym, typem jest Event, który opisuje możliwości przejść pomiędzy stanami:

#### Listing 4. Typ Lifecycle.Event

```
public enum Event {
    ON_CREATE,
    ON_START,
    ON_RESUME,
    ON_PAUSE,
    ON_STOP,
    ON_DESTROY,
    ON_ANY
}
```

Poza wewnętrznymi typami klasa Lifecycle jest wyjątkowo mała i sprowadza się do dwóch metod pozwalających zarejestrować i wyrejestrować „słuchaczy”, czyli funkcje reagujące na zmiany stanów cyklu życia, oraz gettera, umożliwiającego pobranie aktualnego stanu:

#### Listing 5. Klasa Lifecycle

```
public abstract class Lifecycle {
    (...)

    @MainThread
    public abstract void addObserver(@NonNull LifecycleObserver
        observer);
    @MainThread
    public abstract void removeObserver(@NonNull LifecycleObserver
        observer);
    @MainThread
    public abstract State getCurrentState();
}
```

Oprócz klasy Lifecycle wprowadzone zostają również dwa interfejsy: LifecycleOwner oraz LifecycleObserver. Pierwszy z nich oznacza obiekt, który ma cykl życiowy i umożliwia wszystkim zainteresowanym subskrybowanie się do jego zmian:

#### Listing 6. Interfejs LifecycleOwner

```
public interface LifecycleOwner {
    @NonNull
    Lifecycle getLifecycle();
}
```

Zaimplementowanie przez klasę interfejsu LifecycleObserver oznacza natomiast, że chce ona nasłuchiwać zmian cyklu życia jakiegoś komponentu. Jej struktura może jednak wydać się nieco zagadkowa:

#### Listing 7. Interfejs LifecycleObserver

```
public interface LifecycleObserver {
```

Jest tak dlatego, że Google pozostawia programistom dużą dowolność w zakresie decyzji o tym, które zdarzenia cyklu życia chce on obsłużyć. Aby wywołać jakąś metodę w określonym momencie cyklu życia obserwowanego komponentu, trzeba oznaczyć ją adnotacją @OnLifecycleEvent. Choć dosyć niestandardowe, podejście to pozwala na pisanie bardzo zwięzłego i czytelnego kodu.

#### Listing 8. Przykładowa implementacja LifecycleObserver

```
public class MyClass implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    public void onStart() {
        // Allocate resources
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    public void onStop() {
        // DeAllocate resources
    }
}
```

## JAK ZIMPLEMENTOWAĆ LIFECYCLEOWNER

Najpierw warto zaznaczyć, że w wielu przypadkach nie będzie to w ogóle potrzebne. Na przykład AppCompatActivity wchodząca w skład Support Library w wersji 26.1.0 lub wyższej od razu implementuje LifecycleOwner – podobnie rzeczą ma się z fragmentami – wystarczy więc po nich odziedziczyć i sprawę mamy z głowy. Jeżeli jednak z jakiegoś powodu nie możemy z nich skorzystać, możemy zrobić to ręcznie – i to dosyć niewielkim nakładem sił.

Klasa Lifecycle jest wprawdzie abstrakcyjna, ale w bibliotece Architecture Components znajdziemy dziedziczącą po niej klasę o nazwie LifecycleRegistry. Możemy użyć jej w następujący sposób:

#### Listing 9. Implementujemy LifecycleOwner

```
public class MyActivity extends Activity implements
    LifecycleOwner {
    private LifecycleRegistry lifecycleRegistry;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        lifecycleRegistry = new LifecycleRegistry(this);
        lifecycleRegistry.markState(Lifecycle.State.CREATED);
    }

    @NonNull
    @Override
    public Lifecycle getLifecycle() {
        return lifecycleRegistry;
    }

    @Override
    public void onStart() {
        super.onStart();
        lifecycleRegistry.markState(Lifecycle.State.STARTED);
    }

    // ...
}
```

Jedyną niedogodnością pozostanie fakt, że będziemy musieli ręcznie informować klasę LifecycleRegistry o zmianach w cyklu życia aktywności. Ale za to MyActivity będzie współpracowała ze wszystkimi mechanizmami obecnymi w bibliotece.

## KORZYSTAMY Z LIFECYCLE

Teorią – spróbujmy użyć teraz opisanych klas i interfejsów w jakiś sensowny sposób. Na warsztat weźmy więc napisanie klasy, która utworzy wygodny pomost pomiędzy naszą aplikacją a systemowym FusedLocationProviderClient, dostarczającym informacji o bieżącej lokalizacji użytkownika.

# BIBLIOTEKI I NARZĘDZIA

## Listing 10. Implementacja klasy PositionTracker

```
public class PositionTracker implements LifecycleObserver {
    // Private types -----
    Na początku definiujemy interfejs, który będzie musiał zostać zaimplementowany przez wszystkich zainteresowanych zmianą lokalizacji telefonu.

    public interface LocationListener {
        void locationFound(LocationResult result);
    }
    // Private fields-----
    private final FusedLocationProviderClient locationProvider;
    private final LocationRequest locationRequest;
    private final LocationListener locationListener;
    private final Context context;
    private final Lifecycle lifecycle;
    private boolean enabled = false;
    private boolean active = false;
    // Private event handlers -----
    Jeżeli przeznaczeniem naszej aplikacji jest wyświetlenie bieżącej lokalizacji użytkownikowi, nie ma sensu raportować o zmianach, gdy aktywność nie jest widoczna. Możemy tu wykorzystać wygodną metodę isAtLeast pozwalającą określić, czy aktywność znajduje się przynajmniej w stanie, którego oczekujemy (w sensie: w tym stanie lub późniejszym, w kolejności, w jakiej występują one w typie State).

    private LocationCallback locationCallback = new
    LocationCallback() {
        @Override
        public void onLocationResult(LocationResult location) {
            if (lifecycle.getCurrentState().isAtLeast(Lifecycle.State.
                RESUMED)) {
                locationListener.locationFound(location);
            }
        }
    }

    private void activate() {
        try {
            locationProvider.requestLocationUpdates(locationRequest,
                locationCallback, null);
            active = true;
        } catch (SecurityException e) {
            throw new RuntimeException("No permission for localization
                checking!");
        }
    }

    private void deactivate() {
        locationProvider.removeLocationUpdates(locationCallback);
        active = false;
    }

    public PositionTracker(Context context, LifecycleOwner owner,
        LocationListener listener) {
        this.context = context;
        this.lifecycle = owner.getLifecycle();
        this.lifecycle.addObserver(this);
        this.locationProvider = LocationServices.getFusedLocationPro
        viderClient(App.getContext());
        locationRequest = new LocationRequest();
        locationRequest.setInterval(2000);
        locationRequest.setFastestInterval(500);
        locationRequest.setPriority(LocationRequest.
            PRIORITY_HIGH_ACCURACY);
        this.locationListener = listener;
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void resumed() {
        if (enabled) {
            Log.d("PositionTracker", "Activating");
            activate();
        }
    }
}
```

W momencie, gdy aktywność wychodzi na pierwszy plan, aktywujemy klasę, to znaczy zaczynamy nasłuchiwać zmian położenia telefonu i raportować je wszystkim zainteresowanym.

```
@OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
public void paused() {
    if (active) {
        Log.d("PositionTracker", "Deactivating");
        deactivate();
    }
}
```

Przejście aktywności w tło powoduje automatyczne wyłączenie nasłuchiwania.

```
@OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
public void paused() {
    if (active) {
        Log.d("PositionTracker", "Deactivating");
        deactivate();
    }
}
```

Pamiętajmy, że nasłuchiwanie zmian lokalizacji wymaga poproszenia użytkownika o odpowiednie uprawnienia. Zrobić to możemy niestety tylko wewnątrz aktywności, dlatego możemy włączyć klasę PositionTracker dopiero w momencie, gdy mamy pewność, że uprawnienie to zostało już nadane. Jeżeli stanie się to w momencie, gdy aktywność jest już na pierwszym planie, możemy wtedy automatycznie włączyć nasłuchiwanie.

```
public void setEnabled(boolean enabled) {
    if (this.enabled != enabled) {
        this.enabled = enabled;
        if (this.enabled) {
            if (lifecycle.getCurrentState().isAtLeast(Lifecycle.
                State.RESUMED)) {
                activate();
            }
        } else {
            if (active) {
                deactivate();
            }
        }
    }
}
```

Tak napisaną klasę możemy użyć w viewmodelu aktywności, która wyświetla lokalizację:

## Listing 11. Korzystamy z klasy PositionTracker

```
public class PositionTrackingActivityViewModel {
    // Private fields -----
    private IPositionTrackingActivityAccess access;
    private PositionTracker positionTracker;
    private final ObservableField<String> latitude;
    private final ObservableField<String> longitude;
    // Private event handlers -----
    private final PositionTracker.LocationListener
    locationListener = new PositionTracker.LocationListener() {
        @Override
        public void locationFound(LocationResult result) {
            latitude.set(String.valueOf(result.getLastLocation().
                getLatitude()));
            longitude.set(String.valueOf(result.getLastLocation().
                getLongitude()));
        }
    };
    // Public methods -----
}
```

Napisany przez nas PositionTracker potrzebuje LifecycleOwnera, którego cykl życia będzie śledzić. W tym przypadku obiektem tym jest aktywność, więc viewmodel musi otrzymać ją przekazaną przez parametr konstruktora, aby móc przekazać ją dalej.

```
public PositionTrackingActivityViewModel(IPositionTracki
ngActivityAccess access, Context context, LifecycleOwner
lifecycleOwner) {
    this.access = access;
    positionTracker = new PositionTracker(context,
        lifecycleOwner, this.locationListener);
    latitude = new ObservableField<String>();
    longitude = new ObservableField<String>();
}
```

Aktywność musi *explicit* powiadomić viewmodel o nabyciu uprawnień do nasłuchiwania lokalizacji.

```

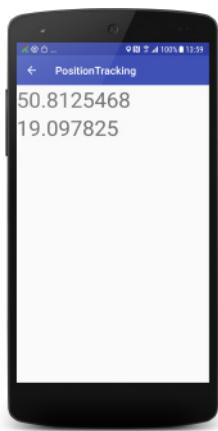
public void locationPermissionGranted() {
    positionTracker.setEnabled(true);
}

// Public properties -----
public ObservableField<String> getLatitude() {
    return latitude;
}

public ObservableField<String> getLongitude() {
    return longitude;
}
}

```

Uruchamiamy aplikację i voilà – możemy odczytać naszą bieżącą lokalizację.



Rysunek 2. Wyświetlanie bieżącej pozycji

## VIEWMODEL

Fakt, iż aktywność w czasie normalnej pracy może zostać niespodziewanie (przynajmniej przez programistę) zniszczona i ponownie utworzona, zawsze przysparza dużo kłopotów. Nie trzeba wiele – wystarczy na przykład zmienić orientację telefonu. Biblioteka Android Lifecycle Components również i w tym przypadku dostarcza ciekawe rozwiązanie, które pomaga rozwiązać ten problem.

Pierwszym ze składników, o których mowa, jest klasa `ViewModel`. Google przewidziało ją jako klasę bazową dla wszystkich viewmodeli objętych dodatkowym wsparciem w zakresie cyklu życia aktywności. Używa się jej stosunkowo łatwo – poza odziedziczeniem po niej nie trzeba robić nic więcej:

### Listing 12. Korzystamy z klasy ViewModel

```

public class PersistenceActivityViewModel extends ViewModel {

    // Private fields -----
    private final ObservableField<String> name;
    private final ObservableField<String> surname;

    // Public methods -----
    public PersistenceActivityViewModel() {
        name = new ObservableField<String>();
        surname = new ObservableField<String>();
    }

    public ObservableField<String> getName() {
        return name;
    }

    public ObservableField<String> getSurname() {
        return surname;
    }
}

```

Drugim składnikiem jest klasa `ViewModelProviders`, której używamy, by pozyskać instancję viewmodelu wewnątrz aktywności:

### Listing 13. Pozyskujemy instancję viewmodelu w aktywności

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    // ...
    binding = DataBindingUtil.setContentView(this, R.layout.
activity_persistence);
    viewModel = ViewModelProviders.of(this).get(PersistenceActivit
yViewModel.class);
    binding.setViewModel(viewModel);
    // ...
}

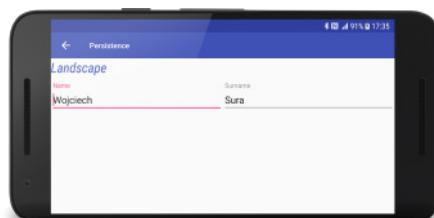
```

Jak to działa? Otóż `ViewModelProviders` reguluje czas życia viewmodeli dla komponentów zgodnie z ich czasem życia. Jeżeli więc na przykład zmienimy orientację telefonu, co spowoduje restart aktywności,.viewmodel pozostanie przy życiu i zostanie przekazany kolejnej instancji w niezmienionej postaci. Jeśli natomiast aktywność zostanie zakończona (na przykład poprzez wywołanie `finish()`),.viewmodel zostanie porzucony, a aktywność otwarta po raz drugi otrzyma jego nową instancję. Viewmodel zamkniętej aktywności ma możliwość wyczyszczenia przechowywanych zasobów (w tym na przykład odpięcia listenerów) – w tym celu należy zaimplementować wirtualną metodę `onCleared`. Momentem zwolnienia viewmodelu aktywności jest jej zakończenie (`finish`), zaś fragmentu – jego odlączenie (`detach`).

Opisane rozwiązanie bardzo dobrze współpracuje z Data Binding Library. Ponieważ stan aktywności jest przechowywany w viewmodelu, ewentualny jej restart nie spowoduje utraty wprowadzonych przez użytkownika danych – można to zobaczyć na Rysunkach 3 i 4. Automatyczne przytrzymywanie viewmodelu przy życiu może sprawdzić się również w innych sytuacjach – choćby w przypadku wywołań asynchronicznych. Ponieważ inicjatorem jest viewmodel, może on reagować na efekt asynchronicznego wywołania niezależnie od tego, czy aktywność jest przy życiu, czy nie.



Rysunek 3. Użytkownik wprowadza dane



Rysunek 4. Dane są na swoim miejscu po obróceniu urządzenia

Cykl życia viewmodelu jest wiązany bezpośrednio z obiektem Lifecycle, przy użyciu którego został on pozyskany (metoda `ViewModelProviders.of()`). Fakt ten można wykorzystać kreatywnie – na przykład do współdzielenia danych pomiędzy fragmentami. W każdym z nich możemy bowiem.viewmodel pobrać w następujący sposób:

#### Listing 14. Pozyskiwanie przez fragment.viewmodelu aktywności

```
viewmodel = ViewModelProviders.of(getActivity()).  
get(SharedViewModel.class);
```

Korzystanie z klasy `ViewModel` pociąga za sobą pewne ograniczenia. `ViewModel` nigdy nie powinien przechowywać instancji widoku, klasy `Lifecycle` lub jakiegokolwiek innej klasy, która może przechowywać referencję do kontekstu aktywności. O ile wewnątrz `ViewModelu` można przechowywać obiekty implementujące `LifecycleObserver` (na przykład `LiveData`, o którym za chwilę), nie może on bezpośrednio obserwować zmian w obserwowlanych obiektach, które uwzględniają cykl życia komponentów (lifecycle-aware observables). Jeżeli natomiast potrzebujemy w viewmodelu instancji kontekstu (`Context`) – na przykład do pobrania systemowego serwisu – możemy odziedziczyć go po `AndroidViewModel` – otrzyma on wtedy w parametrze konstruktora obiekt typu `Application`, który oczywiście rozszerza `Context`.

## PRZEKAZYWANIE DANYCH

Niejednokrotnie zajdzie potrzeba przekazania nowo tworzonymu viewmodelowi jakichś danych. Mogą to być na przykład serwisy, które pobieramy z kontenera zależności (wzorzec Service Locator) lub identyfikator obiektu, który ma być wyświetlony lub edytowany. Również i w takim przypadku można skorzystać z klasy `ViewModel` – konieczne będzie jednak przygotowanie odpowiedniej fabryki. Klasa `ViewModelProvider` dostarcza interfejs `Factory`, który możemy zaimplementować, by utworzyć własną fabrykę. Wygląda to następująco:

#### Listing 15. Viewmodel, który potrzebuje parametrów do pracy

```
public class PassingDataActivityViewModel extends ViewModel {  
  
    // Private fields -----  
  
    public final ObservableField<String> name;  
  
    // Public methods -----  
  
    Viewmodel wymaga przekazania mu dodatkowego parametru do  
    prawidłowego działania.  
  
    public PassingDataActivityViewModel(String newName) {  
        name = new ObservableField<String>(newName);  
    }  
  
    // Public properties -----  
  
    public ObservableField<String> getName() {  
        return name;  
    }  
  
    // Factory -----  
  
    Przygotowujemy klasę fabryki dziedziczącą po ViewModelProvider.Factory. Może ona (chwilowo) przechować dane, które potrzebujemy przekazać viewmodelowi.  
  
    public static class Factory implements ViewModelProvider.  
Factory {
```

```
        private String name;  
  
        public Factory(String name) {  
            this.name = name;  
        }  
  
        @NonNull  
        @Override  
        public <T extends ViewModel> T create(@NonNull Class<T>  
modelClass) {  
  
            return (T)(new PassingDataActivityViewModel(name));  
        }  
    }
```

W tym momencie przekazujemy viewmodelowi wszystkie potrzebne mu dane.

Naturalnie sposób pozyskania viewmodelu również ulegnie zmianie:

#### Listing 16. Pozyskiwanie viewmodelu z dodatkowym parametrem – fragment klasy aktywności

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
  
    super.onCreate(savedInstanceState);  
    binding = DataBindingUtil.setContentView(this, R.layout.  
activity_passing_data);
```

Potrzebne dane wczytujemy z obiektu Intent, za pomocą którego została wywołana aktywność.

```
String name = getIntent().getStringExtra(EXTRA_NAME);  
Konstruujemy fabrykę, przekazując jej parametry dla viewmodelu.
```

```
PassingDataActivityViewModel.Factory factory = new  
PassingDataActivityViewModel.Factory(name);
```

Pozyskujemy viewmodel, przekazując fabrykę, przy użyciu której powinien on zostać zainstancjonowany. Pamiętajmy, że fabryka niekoniecznie zostanie użyta – jeżeli aktywność jest właśnie restartowana po zmianie orientacji, otrzyma ona poprzednią instancję viewmodelu.

```
viewModel = ViewModelProviders.of(this, factory).get(PassingDataActivityViewModel.class);  
binding.setViewModel(viewModel);  
}
```

Wygodnie jest przygotować statyczną metodę konstruującą odpowiedni obiekt Intent, przy użyciu którego można wystartować aktywność. Dzięki temu mamy pewność, że wszystkie dodatkowe dane zostaną prawidłowo przekazane.

```
public static Intent createOpenIntent(Context context, String  
name) {  
    Intent intent = new Intent(context, PassingDataActivity.  
class);  
    intent.putExtra(EXTRA_NAME, name);  
    return intent;  
}
```

## LIVEDATA

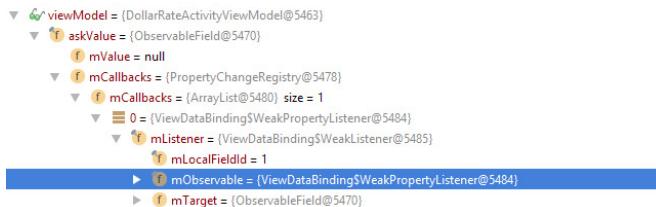
Przechodzimy teraz do zagadnienia przechowywania i przekazywania danych przy użyciu mechanizmów biblioteki Architecture Components. Również i w tym przypadku otrzymujemy od Google klasę przeznaczoną specjalnie do tego celu – `LiveData`. W dużym skrócie jest to:

- » Obserwowlany kontener na dane,
- » Świadomy cyklu życia komponentów,
- » Automatycznie zarządzający subskrybcjami.

Dla porównania wprowadzony wcześniej `ObservableField` możemy określić jako:

- » Obserwalny kontener na dane.

Dla ścisłości: nie jest to do końca prawda. W sieci możemy bowiem znaleźć artykuł Frode Nilsena, który zadał sobie trud zaglądnienia do zrzutu zawartości `ObservableField` w czasie wykonania aplikacji i odkrył, że referencje do subskrybentów są przechowywane jako `WeakReference`. Oznacza to, że tak naprawdę `ObservableField` również automatycznie zarządza subskrybcjami, tylko nie robi tego od razu w momencie, gdy subskrybent nie ma już możliwości nasłuchiwać zmian (jak na przykład aktywność, która została zakończona), ale dopiero po jakimś czasie.



Rysunek 5. `ObservableField`

Scenariusz użycia `LiveData` jest dosyć podobny do `ObservableField`: instancjonujemy lub pozyskujemy skądinąd i przechowujemy go w viewmodelu, subskrybijemy się z poziomu widoku, a później nasłuchujemy zmian i w razie potrzeby aktualizujemy kontrolki UI. Założymy na przykład, że chcemy pobrać bieżący kurs waluty – ponieważ wywołanie jest asynchroniczne, faktyczne dane dostaniemy już po załadowaniu i wyświetleniu aktywności.

#### Listing 17. Implementacja viewmodelu

```
public class DollarRateActivityViewModel extends AndroidViewModel {
    // Private fields -----
    NBPSERVICE jest opakowaniem Retrofitowego interfejsu. Nie zamieszczam go tu, aby nie stracić skupienia na LiveData, kod źródłowy aplikacji prezentującej wszystkie rozwiązania obecne w tym artykule będzie dostępny do ściągnięcia ze strony WWW magazynu.

    private final NBPSERVICE nbpservice;
    Obserwalne pola powiązemy z widokiem, aby zaprezentować dane.

    private final ObservableField<String> bidValue;
    private final ObservableField<String> askValue;
    private final ObservableField<String> dueDate;

    Klasa LiveData domyślnie nie publikuje metod pozwalających na modyfikację danych – robi to dopiero odziedziczena po niej MutableLiveData. Możemy w ten sposób regulować, kto może mieć dostęp do modyfikacji danych.

    private final MutableLiveData<RateModel> rate;
    SingleLiveEvent to mała klasa, którą możemy znaleźć w jednym z przykładów Google. Wspomina o niej Jose Alcerreca w wątku na GitHubie dotyczącym realizowania w viewmodelu operacji wymagających kontekstu aktywności (link na końcu artykułu).

    private final SingleLiveEvent<String> message;
    // Public methods -----
    Viewmodel wymaga przekazania mu dodatkowych serwisów, więc będziemy musieli przygotować również odpowiednią fabrykę – pod koniec listingu.
```

```
public DollarRateActivityViewModel(Application application,
    NBPSERVICE nbpservice) {
    super(application);
    this.nbpservice = nbpservice;
    rate = new MutableLiveData<RateModel>();
    bidValue = new ObservableField<String>();
    askValue = new ObservableField<String>();
    dueDate = new ObservableField<String>();
    message = new SingleLiveEvent<>();
}
```

Upewniamy się, że mamy uprawnienia do wykonania zapytania przez Internet – dostaniemy tę informację od aktywności.

```
public void networkPermissionGranted() {
    nbpservice.getDollarRate(new NBPSERVICE.BackendCallListener<RatesModel>() {
        @Override
        public void success(RatesModel result) {
            if (result != null && result.getRates() != null && result.getRates().size() > 0) {
                rate.setValue(result.getRates().get(0));
            }
        }
        @Override
        public void failure() {
            message.setValue("Cannot get dollar rate!");
        }
    });
}
// Public properties -----
public LiveData<RateModel> getRate() {
    return rate;
}
public ObservableField<String> getBidValue() {
    return bidValue;
}
public ObservableField<String> getAskValue() {
    return askValue;
}
public ObservableField<String> getDueDate() {
    return dueDate;
}
public SingleLiveEvent<String> getMessage() {
    return message;
}
// Factory -----
public static class Factory implements ViewModelProvider.Factory {
    private final Application application;
    public Factory(Application application) {
        this.application = application;
    }
    @NotNull
    @Override
    public <T extends ViewModel> T create(@NotNull Class<T> modelClass) {
        Przekazujemy tu do viewmodelu wszystkie potrzebne zależności pobrane z kontenera zależności, którego rolę pełni klasa App. Możemy – w razie potrzeby – pójść jeszcze o krok dalej i przekazywać je również do fabryki zewnętrznej.
    }
}
```

Przekazujemy tu do viewmodelu wszystkie potrzebne zależności pobrane z kontenera zależności, którego rolę pełni klasa App. Możemy – w razie potrzeby – pójść jeszcze o krok dalej i przekazywać je również do fabryki zewnętrznej.

```
return (T)new DollarRateActivityViewModel(application,
    App.getNbpservice());
}
}
```

Zobaczmy teraz, jak będzie wyglądał kod aktywności współpracującej z napisanym viewmodelem:

# BIBLIOTEKI I NARZĘDZIA

## Listing 18. Implementacja aktywności

```
public class DollarRateActivity extends AppCompatActivity {  
    // Private constants -----  
    private final int PERMISSION_REQUEST = 1;  
    // Private fields -----  
  
    private ActivityDollarRateBinding binding;  
    private DollarRateActivityViewModel viewModel;  
  
    // Private event handlers -----  
  
    rateObserver jest implementacją interfejsu Observer<T> – słuchacza zmian LiveData. Do zimplementowania mamy tylko pojedynczą metodę onChanged, która powiadamia nas o zmianie wartości.  
  
    private final Observer<RateModel> rateObserver = new  
    Observer<RateModel>() {  
        @Override  
        public void onChanged(@Nullable RateModel rateModel) {  
            if (rateModel != null) {  
  
                Dane chcemy opublikować w widoku przy użyciu ObservableField obecnych w viewmodelu, więc ustawiamy tu ich wartości.  
  
                viewModel.getBidValue().set(String.valueOf(rateModel.  
                    getBid()));  
                viewModel.getAskValue().set(String.valueOf(rateModel.  
                    getAsk()));  
                viewModel.getDueDate().set(rateModel.getEffectiveDate());  
            }  
        }  
    };  
}
```

Pamiętacie SingleLiveEvent? Implementujemy tu słuchacza zmian, który w reakcji na pojawienie się nowej wartości wyświetli ją w postaci komunikatu na ekranie.

```
private final Observer<String> messageObserver = new  
Observer<String>() {  
    @Override  
    public void onChanged(@Nullable String s) {  
        App.getMessagingService().inform(DollarRateActivity.this,  
            "Dollar rate", s);  
    }  
};  
  
// Private methods -----  
  
private void initializeControls() {  
    setSupportActionBar(binding.drToolbar);  
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);  
    getSupportActionBar().setDisplayShowHomeEnabled(true);  
}  
  
// Protected methods -----  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    binding = DataBindingUtil.setContentView(this, R.layout.  
        activity_dollar_rate);  
  
    DollarRateActivityViewModel.Factory factory = new  
    DollarRateActivityViewModel.Factory(getApplicationContext());  
    viewModel = ViewModelProviders.of(this, factory).  
    get(DollarRateActivityViewModel.class);  
  
    binding.setViewModel(viewModel);  
}
```

W tym miejscu wiążemy obiekty LiveData ze zdefiniowanymi wcześniej słuchaczami.

```
    viewModel.getRate().observe(this, rateObserver);  
    viewModel.getMessage().observe(this, messageObserver);  
  
    checkPermissions();  
}
```

Upewniamy się, że aplikacja ma uprawnienia do wysyłania zapytań przez Internet.

```
private void checkPermissions() {  
    if (ActivityCompat.checkSelfPermission(this, Manifest.  
        permission.INTERNET) != PackageManager.PERMISSION_GRANTED) {  
        ActivityCompat.requestPermissions(this, new String[] {  
            Manifest.permission.INTERNET }, PERMISSION_REQUEST);  
    } else {  
        viewModel.networkPermissionGranted();  
    }  
}
```

Jeżeli tak nie jest, możemy spróbować poprosić użytkownika o nadanie takich uprawnień.

```
@Override  
public void onRequestPermissionsResult(int requestCode, @  
NonNull String[] permissions, @NonNull int[] grantResults) {  
    super.onRequestPermissionsResult(requestCode, permissions,  
        grantResults);  
    if (requestCode == PERMISSION_REQUEST && permissions.length  
        == 1 &&  
        permissions[0].equals(Manifest.permission.INTERNET) &&  
        grantResults.length == 1 &&  
        grantResults[0] == PackageManager.PERMISSION_GRANTED) {  
        viewModel.networkPermissionGranted();  
    }  
}
```

Uruchamiamy aplikację i obserwujemy efekt.



Rysunek 6. Obiekt LiveData w akcji

## CO DOSTAJEMY W ZAMIAN?

Podstawową zaletą korzystania z LiveData jest automatyczna obsługa cyklu życia nasłuchujących obiektów. Jeżeli aktywność nasłuchująca zmian zostanie zniszczona, LiveData automatycznie usunie subskrypcję. Co więcej, jeżeli aktywność nie jest przynajmniej w stanie STARTED, LiveData nie będzie wysyłać jej powiadomień o zmianach – uczyni to tylko raz, w momencie, gdy aktywność z powrotem zostanie uruchomiona. Ogranicza to między innymi błędy aplikacji związane z zatrzymanymi aktywnościami.

Po klasie LiveData możemy również dziedziczyć, co daje dodatkowe możliwości. Udostępnia ona na przykład dwie chronione metody: onActive i onInactive, które wywoływane są, odpowiednio, w momencie, gdy pojawia się pierwszy obserwator i gdy znika ostatni. Możemy w ten sposób – dla przykładu – zaimplementować dostęp do systemowych serwisów i nasłuchiwać w nich zmian tylko wówczas, gdy mamy kogoś, kto będzie się nimi interesował.

## ROOM

Na koniec zostawiłem sobie bodaj najbardziej efektowny składnik biblioteki Android Architecture Components, czyli Room.

Odkąd baza danych stała się praktycznie nieodłączną częścią większości aplikacji – niezależnie, czy webowych, desktopowych czy mobilnych – namnożyło się dużo ORMów (Object Relational Mapper), czyli narzędzi do możliwie przezroczystego przechodzenia pomiędzy światem obiektowym i relacyjnym. Mogę zaryzykować tezę, że prym w tej kwestii wiedzie środowisko .NET, ponieważ rozbudowana możliwość analizowania kodu w trakcie wykonania (expressions) umożliwiła powstanie mechanizmu LINQ-to-SQL, pozwalającego budować zapytania SQLowe z łańcuchowych wywołań metod, którym przekazywane są wyrażenia lambda. Przykładowe zapytanie może więc wyglądać tak – krótko, zwięzło, czytelnie i niesamowicie wygodnie:

#### **Listing 19. Przykład wykorzystania LINQ-to-SQL do uproszczenia dostępu do danych**

```
int maxAge = unitOfWork.Users.Query()
    .Where(u => u.Name.StartsWith("A"))
    .Select(s => s.Age)
    .Max();
```

Nie trzeba być programistą .NET, by zrozumieć, jakie dane pobiera z bazy powyższe wyrażenie.

W Androidzie od dłuższego czasu królował SQLiteOpenHelper, który robił swoją robotę, ale jednocześnie miał sporo ograniczeń. Jedynymi obiektami, które prawidłowo mapował, były encje – jeżeli wykonaliśmy zapytanie zwracające niestandardowe dane, trzeba było już mapować ręcznie. Część zapytań można było budować w kodzie przy użyciu obiektów QueryBuilder, na przykład tak:

#### **Listing 20. Przykład wykonania zapytania przy użyciu QueryBuildera**

```
return databaseHelper.getCategoryDao().queryBuilder()
    .where()
    .like(Category.NAME_FIELD, name)
    .queryForFirst();
```

Jeżeli jednak zachodziła potrzeba wykonania bardziej skomplikowanego zapytania, trzeba było już dostarczyć SQL, pobrać listę tablic ciągów znaków i moźliwie przetworzyć dane ręcznie.

Room zmienia tę sytuację diametralnie. Zresztą zobaczymy na przykładzie.

Pierwszym krokiem jest przygotowanie encji, które będą używane w aplikacji. Podczas projektowania możemy użyć adnotacji, by umożliwić mechanizm Room odpowiednie przygotowanie tabel w bazie:

#### **Listing 21. Encja User**

```
@Entity(tableName = User.TABLE_NAME)
public class User {
```

Dla ścisłości: Room nie wymaga definiowania nazw pól w stałych – to już jest moje przyzwyczajenie. Dzięki zdefiniowaniu nazw pól i tabel we wnętrzu klasy encji nie muszę potem zastanawiać się, jakie nazwy noszą pola w bazie – pomaga mi Code Insight, a poza tym unikam wprowadzania „magicznych” stałych. Poza tym ewentualna refaktoryzacja bazy przebiega znacznie szybciej i łatwiej.

```
public static final String TABLE_NAME = "User";
public static final String ID_FIELD = "Id";
public static final String NAME_FIELD = "Name";
public static final String SURNAME_FIELD = "Surname";
public static final String AGE_FIELD = "Age";
```

Klucz główny encji – obowiązkowy.

```
@PrimaryKey(autoGenerate = true)
```

```
@ColumnInfo(name = ID_FIELD)
private int id;

@ColumnInfo(name = NAME_FIELD)
@NonNull
private String name;

@ColumnInfo(name = SURNAME_FIELD)
@NonNull
private String surname;

@ColumnInfo(name = AGE_FIELD)
@NonNull
private int age;

public User() {
}

Getter i settery są konieczne dla prawidłowego działania Room.

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getSurname() {
    return surname;
}

public void setSurname(String surname) {
    this.surname = surname;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}
```

Nazrie nie ma tu niczego nowego – w SQLiteOpenHelper encje definiuje się w podobny sposób. Ale przejdźmy teraz do zaimplementowania DAO, czyli Database Access Object – obiektu, który będzie stanowił pomost pomiędzy aplikacją a bazą. W SQLiteOpenHelper mieliśmy gotowe DAO, z których można było, a właściwie trzeba było korzystać. W Room definiujemy je samodzielnie, na przykład w taki sposób:

#### **Listing 22. Przykładowy obiekt DAO w Room**

```
@Dao
public interface UserDao {

    @Insert(onConflict = OnConflictStrategy.FAIL)
    public void insert(User user);

    @Update
    public void update(User user);

    @Delete
    public void delete(User user);

    @Query("SELECT * FROM " + User.TABLE_NAME)
    public List<User> getAll();

    @Query("SELECT * FROM " + User.TABLE_NAME + " " +
        " WHERE " + User.NAME_FIELD + " LIKE \'%\' || :search || "
        "\'\%\'' " +
        " OR " + User.SURNAME_FIELD + " LIKE \'%\' || :search || "
        "\'\%\''")
    public List<User> find(String search);
```

```
@Query("SELECT substr(" + User.NAME_FIELD + ", 1, 1) ||  
substr(" + User.SURNAME_FIELD + ", 1, 1) as initials " +  
"FROM " + User.TABLE_NAME + " " +  
"ORDER BY initials")  
public List<UserInitialsModel> getUserInitials();  
}
```

Tak jest, to nie przywidzenie. DAO jest interfejsem – jego kod generowany jest dla nas przez procesor adnotacji podczas komplikacji projektu. Adnotacje @Insert, @Update i @Delete nie wymagają zbyt wielu komentarzy, natomiast sporo można powiedzieć o @Query.

Muszę przyznać, że na początku trochę się skrzywiłem, widząc, że w Roomie trzeba pisać samodzielnie SQL. Nie ma co ukrywać: Entity Framework z .NETu mocno mnie roznielił. Ale potem uświadomiłem sobie dwie rzeczy: po pierwsze, praktycznie we wszystkich dotychczasowych projektach przedżej czy później musiałem napisać własny kawałek SQLa, bo mechanizmy dostarczane przez SQLiteOpenHelper po prostu nie wystarczały. Po drugie zaś, możliwość napisania SQLa samodzielnie pozwala utrzymać pełną kontrolę nad tym, jakie dane pobieramy z bazy i w jaki sposób. EF jest znany z tego, że w przypadku bardziej skomplikowanych operacji wyszukiwania mocno przekomplikowuje generowane zapytania SQL – tak, że stają się one często bardzo nieoptymalne. Pisząc SQL samodzielnie, można uniknąć tego problemu.

Procesor adnotacji podczas komplikacji odwala kawał roboty – między innymi analizuje wprowadzony SQL i sprawdza, czy udało się wyniki zapytania przemapować na obiekt będący rezultatem wywołania metody. Jeżeli uda się zrobić to w całości, komplikacja przejdzie bez komentarza, jeżeli części pól nie udało się zmapować, zobaczymy ostrzeżenie, a jeżeli żadnego pola nie udało się zmapować – błąd komplikacji. Co więcej, nie musimy już za każdym razem używać encji jako typu wynikowego – może to być zupełnie dowolny model (w powyższym przykładzie metoda getUserInitials zwraca listę obiektów UserInitialsModel). Idąc dalej, procesor adnotacji wyłuskuje parametry z zapytania (w powyższym przykładzie :search) i mapuje je z parametrów wywołania metody. Deklaratywne DAO? Nie wiem, czy mogliśmy wyobrazić sobie coś bardziej wygodnego.

Na koniec pozostało nam tylko klasa bazy danych:

## Listing 23. Klasa bazy danych

```
@Database(entities = {User.class}, version = 1)  
public abstract class DemoDatabase extends RoomDatabase {  
    public abstract UserDao getUserDao();  
}
```

Również i w tym przypadku implementacja generowana jest podczas komplikacji. Jeżeli chcemy pozyskać instancję bazy danych, musimy poprosić o nią bibliotekę Room:

## Listing 24. Pozyskiwanie instancji bazy danych

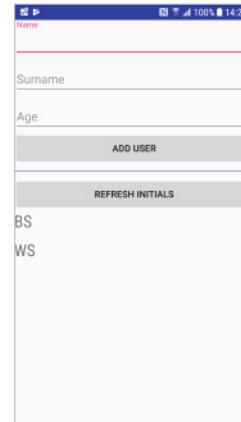
```
public static DemoDatabase getDatabase() {  
    if (database == null) {  
        database = Room  
            .databaseBuilder(getApplicationContext(), DemoDatabase.class,  
            "demo-database")  
            .allowMainThreadQueries()  
            .build();  
    }  
    return database;  
}
```

Google z każdym nowym API coraz bardziej wymusza na programistie, żeby czasochłonne operacje przenosił do wątków w tle – nie inaczej jest tym razem. Jeżeli jednak chcemy zmigrować starszą aplikację na Room, możemy wymusić ten wyłączyć – skorzystałem z niego dla zachowania prostoty przykładu.

Korzystanie z tak przygotowanego DAO jest oczywiście dzinnie proste:

## Listing 25. Korzystanie z DAO

```
public void addUser() {  
    User user = new User();  
    user.setName(name.get());  
    user.setSurname(surname.get());  
    user.setAge(Integer.valueOf(age.get()));  
    database.getUserDao().insert(user);  
}  
  
public void refreshInitials() {  
    List<UserInitialsModel> initials = database.getUserDao().  
        getUserInitials();  
    this.initials.clear();  
    this.initials.addAll(initials);  
}
```



Rysunek 7. Dane pobrane z bazy

Dodam tylko na marginesie, że do wizualizacji listy wykorzystałem bibliotekę ForgetAdapters, udostępnianą na licencji OpenSource przez PGS Software. Dzięki niej wyświetlanie list w RecyclerView i ListView jest znacznie prostsze (i nie wymaga pisania adapterów). Nareszcie.). Link do biblioteki można znaleźć na końcu artykułu. Reklama, reklama.

Warto na koniec wspomnieć, że Room jest skonstruowany w taki sposób, by współpracować ze wszystkimi pozostałymi składnikami Android Architecture Components. Na przykład zamiast encji możesz zwrócić LiveData<T> i procesor adnotacji wygeneruje również i w tym przypadku odpowiedni kod źródłowy.

## OGRANICZENIA

Choć nie można zaprzeczyć temu, że Room jest niesamowicie wygodny, ma również swoje ograniczenia. Kluczowym z nich jest fakt, iż Room nie pobiera automatycznie zależności encji. Mało tego: wewnątrz encji nie można zdefiniować pola, które jest obiektem wskazującym na inną encję. To programista jest odpowiedzialny za to, by samodzielnie pobrać odpowiednie dane.

Dlaczego tak się dzieje? Google wyjaśnia, że chce zmusić developerów do skupienia się nad wydajnością aplikacji. Leniwe ładowanie danych w skrajnych przypadkach mogłoby spowodować

bardzo duże ilości zapytań, które wywoływanie byłyby w głównym wątku. Dlatego też teraz podczas projektowania aplikacji trzeba samodzielnie zaprojektować mechanizm, który pobierze z bazy odpowiednie dane – najlepiej w tle, z wyprzedzeniem i inteligentnie – tak, żeby nie miało to wyraźnego wpływu na wrażenia użytkownika z używania aplikacji.

## NA KONIEC

Nie ma co ukrywać – Google włożyło dużo pracy w kolejną bibliotekę, która ma w założeniu ułatwić programistom pisanie aplikacji na Androida. Muszę jednak przyznać szczerze, że większość ze składników biblioteki przyjąłem z dużą rezerwą, jeśli nawet nie niechęcią. Już tłumaczę, dlaczego.

Trio `Lifecycle`, `LifecycleObserver` i `LifecycleOwner` to świetny pomysł – cykl życia opakowany w obiekt, który można przekazywać i do którego można się subskrybować, jest znacznie wygodniejszy niż implementowanie metod `on*` i przekazywanie informacji o zmianach łańcuskami wywołań. Viewmodel, który jest przytrzymywany przy życiu podczas restartu aktywności, również jest bardzo wygodny. Jednak fakt, iż nie może on przechowywać żadnej referencji do widoku (aktywności) lub czegokolwiek, co przechowuje kontekst, jest bardzo poważnym ograniczeniem.

Jestem w pełni świadom, że w koncepcji MVVM viewmodel powinien być obiektem całkowicie niezależnym od platformy. Niestety jednak na Androidzie jest to prawie nie do uniknięcia: API tego frameworka w wielu sytuacjach potrzebuje kontekstu albo wręcz kontekstu aktywności – na przykład podczas wyświetlania komunikatów. Ale nie tylko – proces proszenia o uprawnienia również musi zostać zrealizowany wewnątrz aktywności, bo to właśnie aktywność dostaje informację zwrotną o odpowiedzi użytkownika.

W dotychczasowych aplikacjach rozwiązywałem ten problem w taki sposób, że aktywność przekazywała się viewmodelowi przez interfejs – `IActivityAccess`. W ten sposób viewmodel mógł poprosić aktywność o zrealizowanie czynności, których sam nie mógłby wykonać, ale jednocześnie – dzięki interfejsowi – nie miał pojęcia, z kim tak naprawdę rozmawia. Jeżeli natomiast w viewmodelu wykonywałam operację wymagającą kontekstu, przekazywałam mu kontekst aplikacji przez konstruktor – tak samo jak w przykładach od Google. Jest to już niestety naruszenie koncepcji MVVM, ale jestem programistą pragmatycznym – zachowanie „czystości” architektury wymagałoby dużo kodu-kleju, a nie przyniosłoby żadnych wyraźnych korzyści.

Jeden z linków na końcu artykułu odnosi się do dyskusji na temat zgłoszonego przez użytkownika problemu z dostępem do aktywności z poziomu ViewModelu. Google proponuje wprawdzie użycie obiektu `SingleLiveEvent`, który pozwala na przenoszenie danych z viewmodelu do słuchającego widoku, ale rozwiązaniu takiemu brakuje estetyki – wyświetlenie komunikatu jest akcją, wykorzystanie do tego obserwowlanego obiektu jest dla mnie mocno prowizoryczną protezą.

Problem ten można byłoby rozwiązać poprzez powiadamianie viewmodelu o podłączeniu lub odłączeniu widoku – tak aby umożliwić choćby najprostszą komunikację pomiędzy nimi (na przykład poprzez interfejs, który zapewni odpowiedni poziom abstrakcji). Być może rozwiązanie takie pojawi się w następnych wersjach biblioteki.

Jest jeszcze jedna kwestia: klasa `ViewModel` nie dziedziczy po `BaseObservable`. Oznacza to, że możemy zapomnieć o `notify-`

`PropertyChanged` – chyba że ręcznie zaimplementujemy interfejs `Observable`.

Również i do `LiveData` mam trochę zarzutów. Głównym z nich jest fakt, że nie jest on obserwowlany przez mechanizmy `DataBinding`. O ile więc sam mechanizm powiadamania i reagowania na zmiany w cyklu życia słuchaczy jest wygodny, to pobieranie danych i wyświetlanie ich w interfejsie użytkownika zmusza nas albo do powrotu do krainy `findViewById` (ewentualnie zastąpionego przez binding, który zrobi to za nas) i ręcznego ustalania danych w kontrolkach oraz subskrybowania się na ich zmiany, albo na rozwiązanie-potworek, które zaprezentowałem w tym artykule, a które wziąłem bezpośrednio z przykładów Google. Mowa tu oczywiście o pobieraniu przez widok danych z `LiveView`, a następnie ręczne przepakowywanie ich do obiektów `ObservableField` w viewmodelu, co wyzwoli mechanizmy Data Binding i po raz drugi przepchnie dane do widoku, tylko tym razem już do konkretnych kontrolek. Mówiąc krótko – zamiast pisać mniej kodu, musimy pisać go więcej. Zamiast upraszczać strukturę aplikacji – musimy ją jeszcze bardziej skomplikować.

Dla osłody, pomimo swoich ograniczeń, Room wydaje mi się rozwiązaniem po prostu rewelacyjnym. Za każdym razem, gdy kopiowałem i wklejałem wielką metodę repozytorium, by dostosować ją do innego złożonego zapytania, miałem wrażenie, że powinno dać się to zrobić wygodniej. Automatyczne generowanie kodu – jak ma to również miejsce w przypadku biblioteki Data Binding, jest bardzo wygodne – zdejmuję z programisty konieczność pisania kodu-kleju, który jest nudny i łatwo popełnić w nim błędy, będące czasami trudnymi do wyśledzenia.

Myszę, że warto dać bibliotece szansę – szczególnie że każdego jej elementu można użyć osobno, według potrzeb. A nawet jeżeli rozwiązania zaproponowane przez Google nie są idealne, można skorzystać z zaproponowanych tam koncepcji, by usprawnić architekturę własnej aplikacji lub poczekać na kolejne wersje biblioteki, które wprowadzą nowe udogodnienia.

## W sieci

- ▶ <https://goo.gl/j5ZztK> – dokumentacja Google.
- ▶ <https://goo.gl/1t3tmL> – wzorce i antywzorce architektury korzystającej z ViewModeli i LiveData.
- ▶ <https://goo.gl/cbuZ8R> – o tym, jak korzystać z viewmodeli i Data Binding Library.
- ▶ <https://goo.gl/EPXKXN> – więcej o viewmodelach i Data Binding Library.
- ▶ <https://goo.gl/MNEvUg> – dyskusja na temat realizowania w viewmodelu operacji wymagających kontekstu aktywności.
- ▶ <https://goo.gl/ZwCM6D> – biblioteka ForgetAdapters ułatwiająca implementowanie list.
- ▶ <https://goo.gl/RUktAM> – szablony MVVM dla Android Studio – przypisują implementowanie aplikacji w tej architekturze. Korzystałem z nich dosyć często podczas pisania demonstracyjnej aplikacji.



### WOJCIECH SURA

wojciechsura@gmail.com

Programuję od przeszło dziesięciu lat w Delphi, C++ i C#, prowadząc również prywatne projekty. Obecnie pracuje w polskiej firmie PGS Software S.A., zajmującej się tworzeniem oprogramowania i aplikacji mobilnych dla klientów z całego świata.

# X-Continuo – autorska infrastruktura produkcyjna dla aplikacji klasy JEE

## czyli oparta na technologii Docker alternatywa dla publicznych usług cloud

Platforma zarządzania procesami tłumaczeniowymi XTRF ([www.xtrf.eu](http://www.xtrf.eu)) jest aplikacją JEE o dużych wymaganiach sprzętowych, charakterystycznych dla systemów klasy ERP (6 GB Heap). Działa w oparciu o system JBoss i bogate portfolio bibliotek Java (JSF/SEAM/RF, Spring, Hibernate, RestAPI, AngularJS) wspieranych przez bazę danych PostgreSQL.

**U**ruchomienie pojedynczej instancji XTRF nie jest specjalnym problemem. Ale co zrobić, gdy produkt jest sukcesem? Gdy musisz dostarczyć i zarządzać wieluset instancjami rozsianymi po całym świecie? Jak zbudować efektywną infrastrukturę i równocześnie nie zbankrutować, jakiej technologii zaufać?

Jest więcej niż pewne, że już wcześniej spotkałeś się z podobnymi problemami, zdajesz sobie sprawę z istnienia różnych chmur obliczeniowych i współczesnych rozwiązań wirtualizacyjnych, które działają od razu po wyjęciu z pudełka (np. EC2, DigitalOcean itp.), które są łatwe do integrowania i skalowania. Zazwyczaj dają one o wiele więcej, niż potrzebujesz, jednak ich intensywne produkcyjne wykorzystanie kończy się miesięcznym rachunkiem, który może nie pasować do Twoich planów budżetowych i polityki bezpieczeństwa.

W kolejnych akapitach chcielibyśmy podzielić się historią sukcesu XTRF i wyjaśnić, w jaki sposób autorska architektura XTRF X-Continuo pomogła pokonać te problemy i pozostać agnostycznym względem dostawcy infrastruktury (IAAS) oraz utrzymać w rynach stronę kosztową.

### WŁASNY KLASTER

Zarządzanie kilkudziesięciomainstancjami z perspektywy czasu wydaje się proste. Zawsze chciałeś zbudować własny klasterek (my też), więc go tworzysz, dzieląc się z kolegami sukcesami coraz bardziej zaawansowanych skryptów, które tworzysz do automatyzacji

jego konfiguracji. Cheff, Puppet, Pacemaker są Twoimi największymi przyjaciółmi. W tle baza danych (replikacja, partycjonowanie) oraz współdzielony system plików. Działa stabilnie, dostałeś nawet premię, bo koszt infrastruktury dla pojedynczej instancji to jedyne 12 euro (wobec szybko obliczonymi 120 euro w „prawdziwej” chmurze).

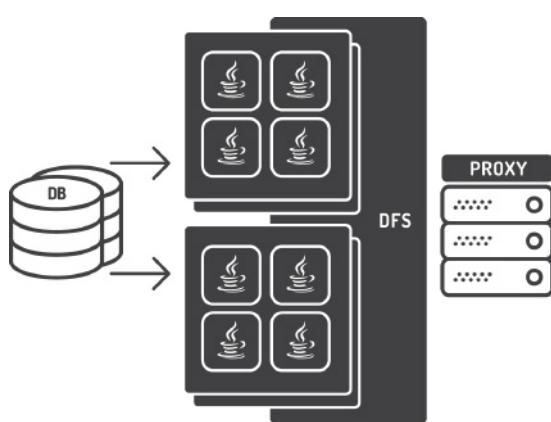
Zalety	Wady
<ul style="list-style-type: none"><li>» Niski koszt ~ 55 PLN/JVM (jeśli wypełniony 99%)</li><li>» Bezpośredni dostęp do konfiguracji i parametrów systemu operacyjnego</li><li>» Wysoka wydajność (bare metal)</li><li>» To samo centrum danych (szybka komunikacja węzlowa)</li><li>» Bezpośredni dostęp dla klientów (ssh, ftp) w razie potrzeby</li></ul>	<ul style="list-style-type: none"><li>» Ogranicza 100-120 JVM na klastrze</li><li>» SPOF (baza, proxy)</li><li>» Czas potrzebny na budowę</li><li>» Pojedyncze centrum danych</li><li>» Czas regeneracji – godziny</li><li>» Skrypty automatyzacji</li><li>» niestandardowe rozwiązania</li></ul>

Jednak wraz z dodawaniem kolejnych węzłów widzisz, że niedługo osiągniesz limit, że Twoje rozwiązanie nie jest odporne na awarię Data Center (kto doświadczył braku prądu w OVH w listopadzie 2017, ręka w góre), masz coraz większą świadomość istnienia w tej architekturze SPOFów (ang. *Single Point of Failure* – pojedynczych punktów awarii). Męczy Cię czas wdrażania nowych pracowników oraz to, że jesteś niezastąpiony w modyfikacji skryptów, z których byłeś tak dumny.... Osiągnąłeś limit, desperacja narasta....

### JAK SIĘ SKALOWAĆ?

Zanim pobiegnesz do szefa i przyznasz, że jedyna nadzieja w chmurze firmy X, pamiętaj, że chmura to nie tylko zasób obliczeniowy (procesor + pamięć), ale przede wszystkim cały szereg zaawansowanych usług o różnym sposobie działania i cenie. Kluczowym elementem sukcesu jest świadomość WSZYSTKICH usług, z których korzysta Twoja aplikacja.

- » Jakich RDBMS używasz? Czy są dostępne w chmurze X?
- » Czy polegasz na lokalnym systemie plików? Jak go używasz?
- » Czy możesz sobie pozwolić na refaktoryzację aplikacji w celu dostosowania do chmury?
- » Buforowanie, przechowywanie, model wyceny – czy pasuje do Twojej aplikacji?
- » Jakie są ograniczenia (np. w ilości/szybkości dodawania kolejnych węzłów)?



Rysunek 1. Własny klasterek ma jednak wady



Rysunek 2. Aplikacja wykorzystuje wiele usług, musisz znać je wszystkie

Wszystkie usługi? A DNS i http proxy? Musisz mieć PEWNOŚĆ, że wypisałeś wszystko ZANIM zdecydujesz...

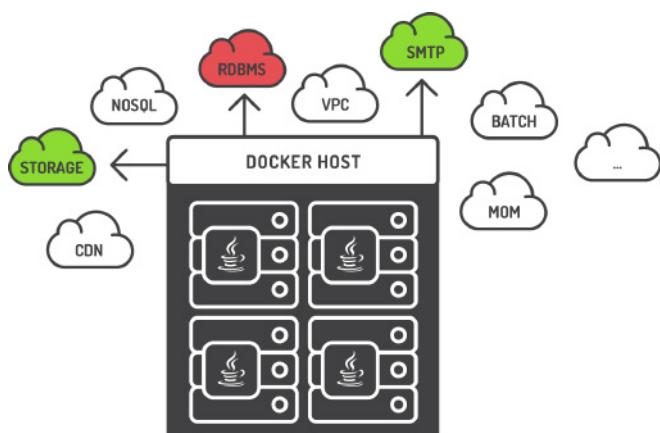
## Cena

W kontekście uruchamiania XTRF skalkulowaliśmy cenę wiodących dostawców w oparciu o ich publicznie dostępne cenniki – wynik jest kwotą od 80 do 400 euro za miesiąc (720 godzin) działania jednej instancji. Przy planowanym 1000 instancji mam niemały miesięczny wydatek, który mówiąc kolokwialnie „nie przejdzie”.

Przy okazji pamiętałeś, że Twoja aplikacja wykorzystuje lokalny system plików jako miejsce składowania logów?... Kolejna usługa, którą musisz uwzględnić.

## ZRÓB SAM, ALE NIE WSZYSTKO

Szczegółowa analiza składowych kosztów usług cloud pokazuje, że najdroższym elementem są węzły robocze (ang. *working nodes* – pamięć i procesor), natomiast inne usługi są stosunkowo tanie. Równocześnie analiza większości aplikacji pokazuje, że kluczowym dla zapewnienia stabilności i skalowalności są usługi posiadające stan, podczas gdy skalowanie usług bezstanowych jest zagadniением stosunkowo prostym.



Rysunek 3. Twój infrastruktury powinna wykorzystywać inne usługi stanowe

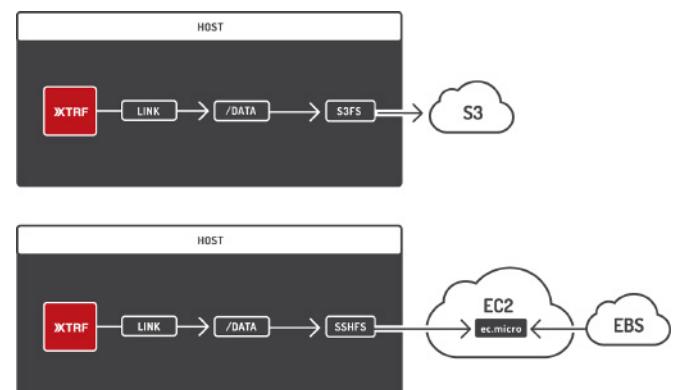
Zdecydowaliśmy zatem przenieść do chmury elementy stanowe, zyskując ich praktycznie nieograniczoną skalowalność oraz ponosząc koszt jedynie faktycznego wykorzystania, natomiast samą aplikację uruchamiać jako bezstanową usługę w oparciu o opłacaną z góry, wynajętą infrastrukturę sprzętową (opłata za kolejne węzły w klastrze), wykorzystując duet Ansible i Docker.

- » Niezależność od dostawcy IaaS,
- » Lepiej kupować usługi stanowe (pamięci, bazy danych, dzienniki), niż budować je samodzielnie (ktoś inny zadba o problem, jeśli się wydarzy),

- » Spójność konfiguracji,
- » Wysoka dostępność (HA), tylko jeśli potrzebna (akceptacja 2-minutowej niedostępności może istotnie obniżyć koszty infrastruktury),
- » Łatwo przywracalne, bezstanowe węzły robocze,
- » Proste, jednorodne węzły, brak wydzielonych węzłów specjalnych,
- » Węzły autystyczne – czyli takie, które są nieświadome siebie nawzajem.

Przyjrzyjmy się dokonanym wyborom w kontekście konkretnych usług. W tym miejscu musimy podkreślić, że charakterystyka Twojej aplikacji może prowadzić do innych wyborów.

## Usługa 1 – system plików/Global Storage



Rysunek 4. System plików z chmury może być dołączony do aplikacji na różne sposoby

XTRF jest specyfczną aplikacją (jaka aplikacja nie jest specyficzna...), która w trakcie zakładania projektu tworzy strukturę katalogów projektu – może być ich kilkadziesiąt. Zmierzliśmy czasy tworzenia katalogów w chmurze; uzyskane wyniki były nieakceptowalne.

	S3 over s3fs	SSHFS to EBS
]<\$ mkdir x 150 ]\$ ls	3 mins. 10 sec.	50 sec.

Prosty mechanizm podłączenia (ang. *mount*) systemu plików z chmury okazuje się dla XTRF niewystarczający. W efekcie musiano wykonać refactoring aplikacji, aby tworzenie katalogów nie było potrzebne.

## Usługa 2 – PostgreSQL jako usługa?

W zależności od sposobu wykorzystania aplikacji wydajność w dostępie do bazy danych może być istotna lub bardzo istotna. Szczególnie rozwiązania oparte na Hibernate mają w tym względzie wysokie wymagania. Porównaliśmy szybkość operacji realizowanej z różnej lokalizacji geograficznej (Data Centers) – wyniki poniżej:

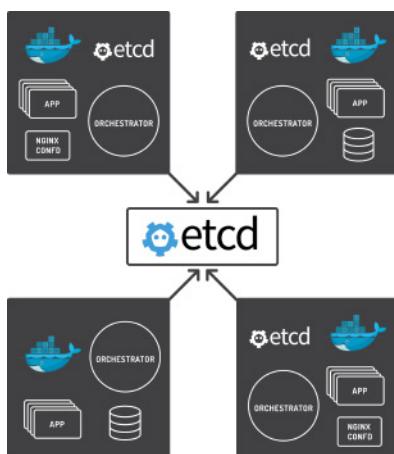
	EC2->RDS (West Virginia)	OVH (Quebec)->RDS	KRK->RDS	Hetzner (DE) Direct access
pg_bench	367 TX/sec.	14 TX/sec.	7 TX/sec.	1573 TX/sec.

W efekcie pomimo wcześniejszego planu nie zdecydowaliśmy się użyć bazy danych jako usługi i zarządzamy nią samodzielnie. Za nim jednak pójdzieś tą drogą, sprawdź, czy na pewno 360 TX na sekundę jest dla Twojej aplikacji niewystarczające.

## Usługa 3 – rozproszona konfiguracja

Kluczem do właściwego, efektywnego zarządzania instancjami jest stabilna, rozproszona, odporna na awarię usługa konfiguracyjna. Takie własności da się uzyskać poprzez połączenie usług ETCD i CONFD.

**ETCD** jest rozproszoną bazą danych typu klucz-wartość, łatwo dostępną przez API i bardzo efektywną (ponad 1000 zapisów na sekundę). Jest też prosto integrowana z Dockerem oraz wspiera operacje warunkowego zapisu (ang. *conditional writes*). Nadaje się idealnie do przechowywania informacji konfiguracyjnych o instancjach aplikacji (status, wielkość pamięci, parametry JVM itp.).



Rysunek 5. ETCD propaguje konfigurację między klastrami

**CONFID** umożliwia utrzymywanie lokalnej konfiguracji węzła poprzez automatyczne informowanie lub restart aplikacji w celu pobrania nowych ustawień konfiguracyjnych.

Powyższy duet zapewnia propagowanie informacji konfiguracyjnej do węzła bez potrzeby posiadania bezpośredniego dostępu do tego węzła z konsoli konfiguracyjnej (a nawet świadomości, gdzie się on teraz znajduje :-).

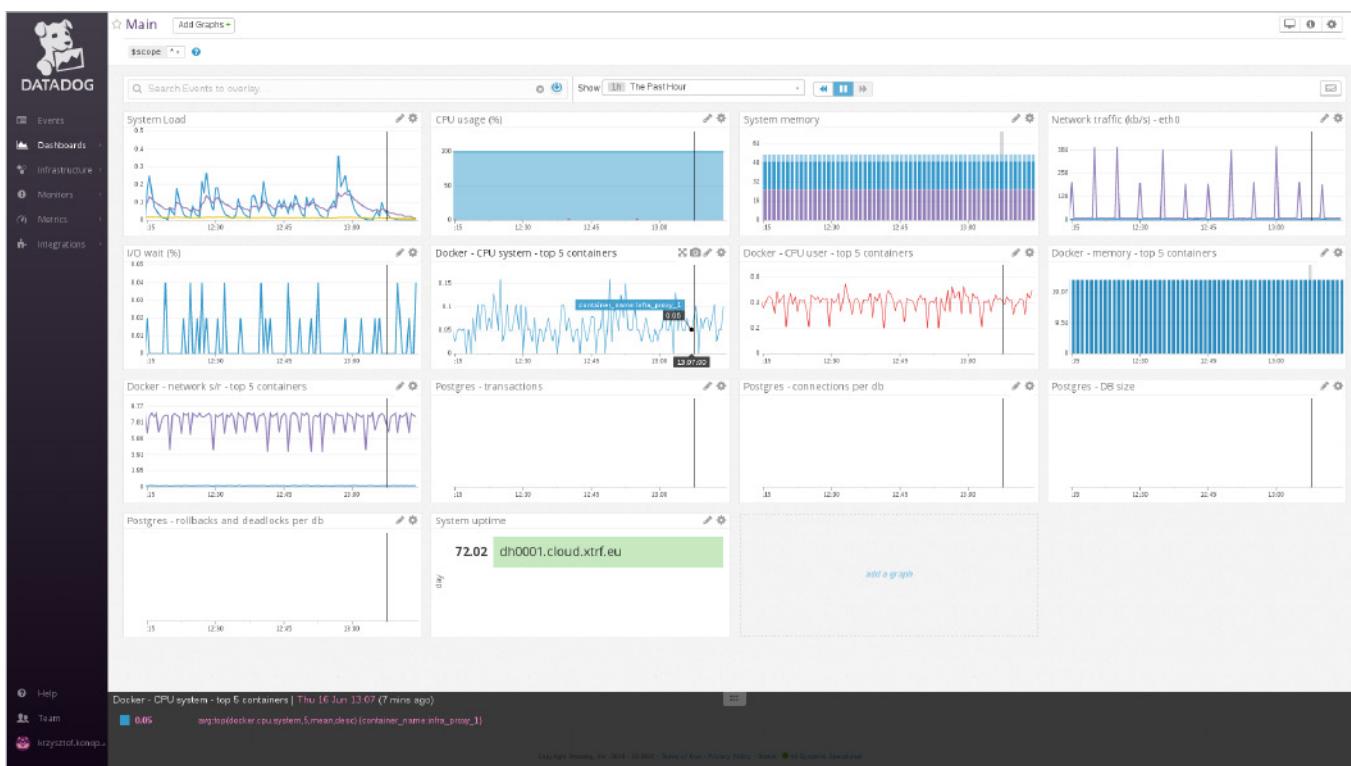
## Usługa 4 – monitoring

Zarządzanie kilkusetinstancjami aplikacji wymaga bardzo efektywnych mechanizmów monitorowania, zarówno samej infrastruktury, jak i poszczególnych aplikacji. Niezbędnymi elementami są alarmowanie o niepożądanych zdarzeniach oraz możliwość automatycznego działania w przypadku ich wystąpienia (np. restart instancji, zwiększenie pamięci itp.). Koszt istniejących w chmurze rozwiązań skutecznie odwrócił nas od próby tworzenia tych mechanizmów na własną rękę.

## Inne usługi infrastrukturalne

Praktycznie każda aplikacja wysyła dziś emaile (choćby do resetowania hasła). Zamiast wykorzystywać własny serwer, wygodniej wykorzystać stabilną i tanią usługę zewnętrzną – my wybraliśmy SMTP – **Amazon SES** z uwagi na wygodne SDK, wysoki limit ilości wysyłanych wiadomości oraz niski poziom odrzuceń (ang. *bounce rate*).

Aby szybko przełączać aplikacje pomiędzy węzłami w klastrze (lub nawet między klastrami), niezbędny jest mechanizm rozwiązywania nazwy (ang. *DNS*), który umożliwia szybką zmianę adresu. Klasyczne usługi nawet jeśli zapewniają API, to ograniczają czas aktualizacji informacji o zmianach do minut (lub nawet godzin czy dni). Wybrana przez nas usługa **DNSSimple** posiada wygodne REST API oraz aktualizuje wpis bez zbędnych opóźnień.



Rysunek 6. A ty jaką usługę monitoringu wybierzesz?

## PODSUMOWANIE

Dzięki sprytnemu połączeniu technologii Docker, Ansible, ETCD, Confd, S3, IAM, DNSSimple udało się zaprojektować i zbudować własną infrastrukturę chmury do obsługi kilku tysięcy aplikacji. XTRF X-Continuo osiągnął możliwości produkcyjne i został wykorzystany jako platforma do hostowania serwerów JBoss w kilku regionach. Wstępne testy wykazały, że powinien skalować do ponad 1k węzłów.

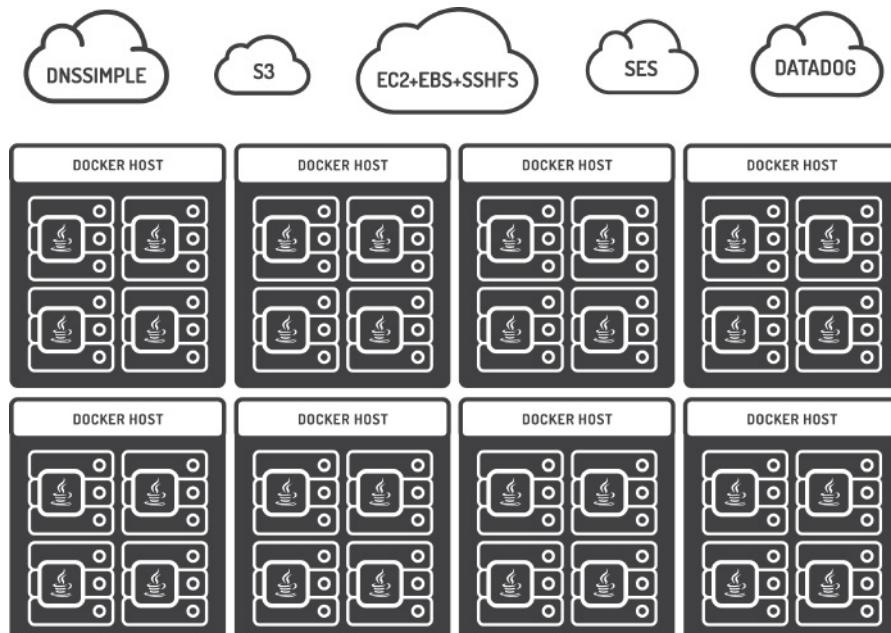
Zastosowanie technologii konteneryzacji pozwoliło na odseparowanie problemu doboru właściwego węzła obliczeniowego. Infrastruktura sama skaluje się w górę poprzez brak zależności między węzłami. Pozwala to na jej rozszerzanie zarówno z wykorzystaniem węzłów fizycznych, jak i wynajmowanych od różnych dostawców.

Zastosowanie platformy X-Continuo umożliwiło znaczącą redukcję czasu wdrożenia klienta. Rozwiążanie pozwoliło na zachowanie kosztów utrzymania jednej instancji systemu na poziomie kilkukrotnie niższym niż w chmurach wiodących dostawców.

wanie kosztów utrzymania jednej instancji systemu na poziomie kilkukrotnie niższym niż w chmurach wiodących dostawców.

## PODZIĘKOWANIA

„Ciągłość procesów i bezpieczeństwo danych w rozproszonym systemie zarządzania tłumaczeniami”. Projekt współfinansowany ze środków Europejskiego Funduszu Rozwoju Regionalnego w ramach Programu Operacyjnego Innowacyjna Gospodarka, działanie 1.4.



Rysunek 7. Skalowalna, wydajna, heterogeniczna architektura X-Continuo



### DR INŻ. DOMINIK RADZISZOWSKI

Wizjoner technologii, ekspert chmury obliczeniowej, architekt, ekspert łączący świat IT i biznesu. Współzałożyciel XTRF Management Systems Ltd. – wiodącego systemu zarządzania tłumaczeniami używanego przez ponad 2000 project managerów dziennie.



### MICHAIŁ KRZYWUCKI

Kierownik projektów IT zaciekle broniący dostępu do technicznych aspektów swojej pracy, zwolennik zwinnych metodyk zarządzania, entuzjasta technologii JEE i chmur obliczeniowych.

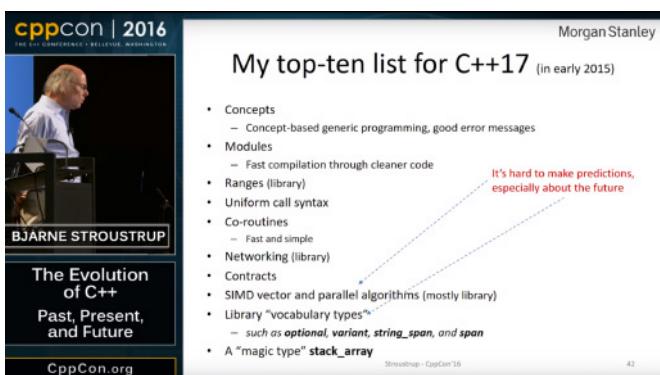
# C++17 – nowy, miłościnnie panujący nam standard C++

Nieco ponad rok temu (Programista 10/2016) zapowiedziany został nadchodzący standard języka. Od tego czasu komitet standaryzacyjny zdążył się jeszcze spotkać i dokonać pewnych zmian.

**F**inalny kształt C++17 poznaliśmy po marcowym spotkaniu komisji standaryzacyjnej w miejscowości Kona na Hawajach, gdy szkic standardu został poddany głosowaniu organów narodowych (ang. *national bodies*). 6 września głosowanie zostało zakończone jednogłośną akceptacją [1], co pozwoliło pominąć dalsze głosowanie i przejść bezpośrednio do publikacji. Organizacja ISO opublikowała C++17 w grudniu 2017 jako ISO/IEC 14882:2017 [2]. Wedle relacji członków komitetu 9 miesięcy od zakończenia prac oznacza bardzo szybką publikację...

Od dawna wiadome było, że z szumnych zapowiedzi ewangelistów oraz z listy życzeń Bjarne Stroustrupa, oryginalnego twórcy języka, niewiele udało się zrealizować. Znakomicie obrazuje to *keynote* tego ostatniego na konferencji C++Con 2016, gdzie z wymienionych dziesięciu zmian żadna nie została w pełni zaimplementowana (Rysunek 1). Mimo to zmiany były liczne i w znaczącej większości pozytywne – choć w równie znaczającej większości drobne i nastawione na ułatwienie kodowania.

W tym artykule opisana zostanie ostateczna formula nowego standardu. Zmiany uszeregowane będą mniej więcej od najistotniejszych dla programistów C++ do tych mniej ważnych, lub ważnych tylko dla specyficznych grup, np. twórców bibliotek.



Rysunek 1. Bjarne Stroustrup przedstawia listę życzeń dla C++17 (źródło: [https://youtu.be/\\_wzc7a3McOs?t=3538](https://youtu.be/_wzc7a3McOs?t=3538))

## USUNIĘCIE PRZESTARZAŁOŚCI

Wymienione tu części języka zostały *kompletnie usunięte* ze standardu, co oznacza, że kod je zawierający nie powinien się skompilować<sup>1</sup>, ponieważ nie jest poprawnym kodem C++17 – tak samo jak kod zawierający przypisanie literała ciągu znaków do mutowalnego wskaźnika na char (Listing 1) nie powinien się skompilować w C++11 ani późniejszych [3].

1. Należy rozróżnić „nie powinien się skompilować” od „nie skompiluje się”. Kompilatory nigdy w pełni restrykcyjnie nie trzymają się standardu, szczególnie przy domyślnych ustawieniach, więc czasem zamiast komunikatu błędu wyświetla informację z ostrzeżeniem.

### Listing 1. Niedozwolone przypisanie literała ciągu znaków do mutowalnego wskaźnika na char

```
int main()
{
    char* foo = "bar";
}
```

### Specyfikacja throw()

Od teraz dozwolone jest wyłącznie noexcept. Wyjątkiem jest puste `throw()`, które staje się aliasem dla `noexcept(true)`. `throw(std::exception)` przedstawione w Listingu 2 jest niepoprawnym kodem.

### Listing 2. Dynamiczna specyfikacja wyjątków

```
void foo() throw();
void bar() throw(std::exception) {}
```

### Auto\_ptr

`std::auto_ptr` to potworek pozostały po C++98, gdzie nie możliwa była poprawna implementacja `std::unique_ptr`. Wedle wiedzy autora we wszystkich sensownych zastosowaniach `std::auto_ptr` można zastąpić `std::unique_ptr`.

### Listing 3. Użycie typu nieistniejącego już w bibliotece standardowej: std::auto\_ptr<int>

```
int main()
{
    std::auto_ptr<int> a(new int);
}
```

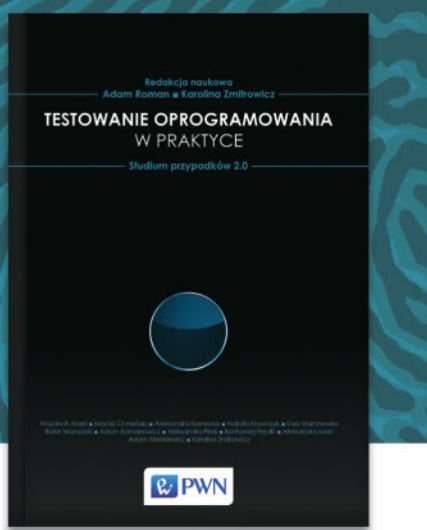
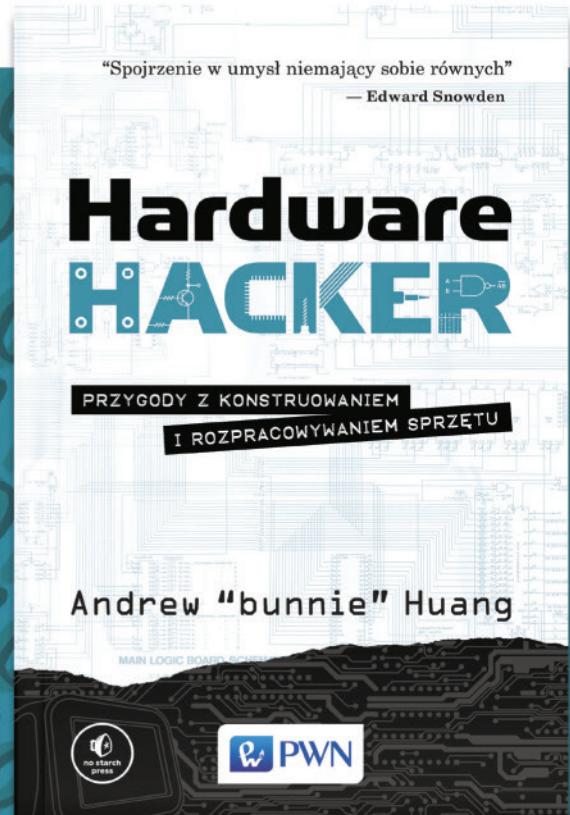
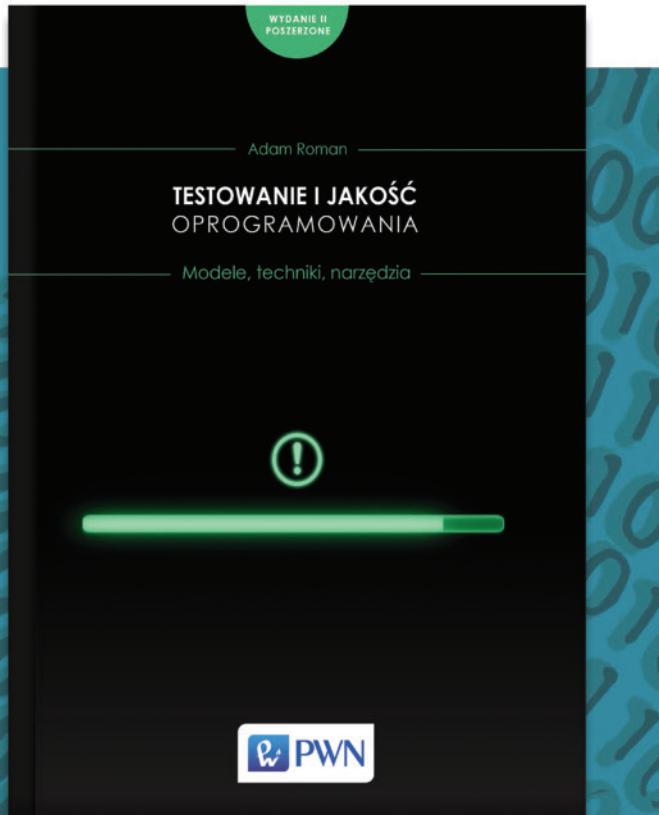
### Operator ++ dla bool

Samo istnienie operatora ++ dla `bool` może być dla wielu osób zaskoczeniem, choć od strony implementacyjnej wydaje się ono zrozumiałe, ponieważ `bool` jest w C++ (oraz w C) realizowany jako zmienna liczbowa o wielkości 1 bajta. W Listingu 4 przedstawiono niedozwolone od C++17 użycie.

### Listing 4. Wykorzystanie operatora ++ dla bool

```
int main()
{
    bool b = false;
    b++;
    assert(b == true);
}
```

# Najlepsze od



KSIAŻKI DOSTĘPNE NA: [WWW.KSIEGARNIA.PWN.PL](http://WWW.KSIEGARNIA.PWN.PL)

Odwiedź nas na:  
**IT.PWN.PL**

## Trójznaki

Trójznaki (ang. *trigraphs*) to pozostałość po burzliwym rozwoju komputerów w latach 70-tych i 80-tych, znajdująca się w C++ dla kompatybilności z C, co było szczególnie istotne na początku istnienia języka. Komputery z tamtej epoki bardzo się od siebie wzajemnie różniły, a jedną z tych różnic były znaki dostępne na klawiaturach i zestawach znaków różnych platform.

Aby umożliwić korzystanie ze znaków [ ] { } | ~#^`\, które nie były dostępne na wszystkich platformach, C wprowadziło trójznaki (Tabela 1). Są to specjalne sekwencje trzech znaków, które były zamieniane na niedostępne na klawiaturze danego komputera znaki. Jedną z wymienianych wad, poza znacząco obniżoną czytelnością, jest zamiana trójznaków w pierwszej fazie translacji programu. Oznacza to, że zamieniane są przez kompilator przed czymkolwiek innym, nawet makrami preprocesora.

Obecnie nie są one prawie nigdzie wykorzystywane, nawet międzynarodowy konkurs zobfuskowanego (czyli takiego, którego czytanie zostało celowo utrudnione) kodu w C (ang. *The International Obfuscated C Code contest*) sugeruje ich unikanie [5]. Przykład ich złośliwego wykorzystania znajduje się w Listingu 5, gdzie ??/ zamieniane jest na \, co powoduje wcięgnięcie wyrażenia warunkowego do komentarza i bezwarunkowe wywołanie funkcji poniżej.

Sekwencja	Zamieniana na
??=	#
??/	\
??'	^
??(	[
??)	]
??!	
??<	{
??>	}
??-	~

Tabela 1. Trójznaki (źródło: [4])

### Listing 5. Użycie trójznaku ??/ w celu wywołania trzeciej wojny światowej [6]

```
void launch_nuclear_missiles();

int main()
{
    bool we_are_at_war = false;
    // only send nuclear missiles if we're at war
    // we don't want needless deaths, do we??
    if(we_are_at_war)
        launch_nuclear_missiles();
}
```

Poza trójznakami C++ ma jeszcze dwuznaki (ang. *digraphs*) oraz zamianę specjalnych tokenów. Więcej przeczytać można o tym w [25] [26].

## INICJALIZACJA W WYRAŻENIU WARUNKOWYM

Jest to uproszczenie dla programistów, pozwalające na zapisanie wewnętrz instrukcji warunkowych `if` i `switch` inicjalizacji obiektu

oraz warunku, co pozwala na uniknięcie dodatkowych zagnieżdżeń, jeśli obiekt używany jest tylko w części kodu wykonywanej warunkowo. Na przykład kod w C++ z Listingu 6 może zostać zastąpiony tym z Listingu 7. Analogicznie można inicjalizować obiekt w warunku `switch`.

### Listing 6. Przykładowy kod w C++14

```
int main()
{
    map<int, int> graph;

    {
        auto result = graph.insert(make_pair(0, 42));
        if(result.second) {
            // stuff
        }
    }
}
```

### Listing 7. Kod analogiczny do tego z Listingu 6, korzystający z nowości w C++17

```
int main()
{
    map<int, int> graph;

    if(auto result = graph.insert(make_pair(0, 42)); result.second) {
        // stuff
    }
}
```

Ciekawostką dla niektórych może być informacja, że już pierwszy standard C++, C++98, zezwalał na inicjalizację w warunkach wyrażeń warunkowych i pętli – ale tylko jeśli świeżo zdefiniowany obiekt był konwertowalny do wartości logicznej. W większości przypadków ograniczało to użyteczność do funkcji, które zwracały `nullptr` lub `0` w przypadku niepowodzenia, a takich, wbrew popularom, nie ma wiele. Przykładowe użycie znajduje się w Listingu 8.

### Listing 8. Inicjalizacja wewnętrz warunku, poprawna od początku ustandaryzowanego C++

```
void foo(void*);

int main()
{
    if(void* ptr = malloc(1048576)) {
        foo(ptr);
        free(ptr);
    }
}
```

## STRUCTURED BINDINGS

Pierwotną formalną nazwą tej nowinki było *decomposition declarations*, choć potocznie wszyscy – wraz z twórcami – nazywali ją *structured binding declarations*. W marcu komitet standaryzacyjny ujednolicił nazewnictwo w tym zakresie, przyjmując popularniejszą potoczną nazwę.

Deklaracja *structured bindings* zezwala na przypisanie w jednej deklaracji zmiennych do elementów inicjalizatora, bez jawnego tworzenia dodatkowych zmiennych. „Rozpakowane” mogą zostać kontenery standardowe o statycznie znanej wielkości (`std::tuple`, `std::pair`, `std::array`) oraz typy zdefiniowane przez użytkownika, jeśli w takim typie:

- » wszystkie niestatyczne elementy są dostępne publicznie, lub są elementami jego jednoznacznej publicznej klasy bazowej, oraz nie zawiera on anonimowych unii, lub

» jeśli oferuje poprawną specjalizację `std::tuple_size`, `std::tuple_element` i `get` (nie `std::get`, tylko `get` dostępne jako element klasy lub funkcja dostępna przez ADL<sup>2</sup>).

Deklaracja wygląda następująco: `auto [z1, z2, ..., zn] = foo;`. Dodatkowo `auto` może być opatrzone referencją kwalifikowaną `const` lub `volatile`. Taką deklarację można porównać do utworzenia nienazwanej zmiennej, do której aplikowane są kwalifikatory przy `auto`, i użyciu jej do inicjalizacji kolejnych referencji do elementów nienazwanej zmiennej. Brzmi to trochę zawile, ale jest to łatwo zobrazować poprzez przykładowe zestawienie analogicznych fragmentów kodu ze standardów C++14 i C++17, które znajdują się w Listingach 9 i 10.

#### Listing 9. Kod w C++14

```
int main()
{
    auto tmp1 = std::make_pair("answer"s, 42);
    auto& a = tmp1.first;
    auto& b = tmp1.second;

    auto&& tmp2 = std::make_tuple("answer"s, 42, true);
    auto& c = std::get<0>(tmp2);
    auto& d = std::get<1>(tmp2);
    auto& e = std::get<2>(tmp2);

    auto const& tmp3 = std::make_tuple(1);
    auto& f = std::get<0>(tmp3);

    struct foo{ int bar; std::string baz; };
    foo qux{42, "answer"};
    auto& tmp4 = qux;
    auto& g = tmp4.bar;
    auto& h = tmp4.baz;

    int arr[2] = {42, 43};
    auto&& tmp5 = arr;
    auto& i = tmp5[0];
    auto& j = tmp5[1];
}
```

#### Listing 10. Kod analogiczny do tego z Listingu 9, z użyciem structured bindings

```
int main()
{
    auto [a, b] = std::make_pair("answer"s, 42);
    auto&& [c,d,e] = std::make_tuple("answer"s, 42, true);

    auto const& [f] = std::make_tuple(1);

    struct foo{ int bar; std::string baz; };
    foo qux{42, "answer"};
    auto& [g, h] = qux;

    int arr[2] = {42, 43};
    auto&& [i, j] = arr;
}
```

W Listingu 11 przedstawiona jest definicja własnej specjalizacji w celu zapewnienia obsługi *structured bindings* dla typu zdefiniowanego przez człowieka (ang. *user-defined*), którego liczba rozpoznawanych elementów różni się od liczby elementów klasy.

#### Listing 11. Własne tuple\_size/tuple\_element/get

```
namespace kq
{
    struct foo
    {
        virtual ~foo() = default;
```

```
        virtual std::string const& name() const=0;
        virtual int id() const=0;
    };

    template<size_t> auto get(foo const& f)
    {
        return f.id();
    }

    template<> auto get<1>(foo const& f)
    {
        return f.name();
    }

    struct bar : foo
    {
        bar(std::string const& n, int i): name_(n), id_(i) {}

        std::string const& name() const override
        {
            return name_;
        }

        int id() const override
        {
            return id_;
        }

        private:
            std::string name_;
            int id_;
    };
}

namespace std
{
    template<>
    class tuple_size<kq::foo>;
    public integral_constant<size_t, 2> {};

    template<size_t I>
    class tuple_element<I, kq::foo>
    {
    public:
        using type = decltype(get<I>(declval<kq::foo>()));
    };
}

int main()
{
    kq::foo const& f = kq::bar{"answer", 42};

    auto&& [id, name] = f;
    std::cout << id << ", " << name << '\n';
}
```

Używając *structured bindings*, można poprawić czytelność kodu z Listingu 7. W Listingu 12 przedstawiono takie usprawnienie.

#### Listing 12. Przykład z Listingu 7 wzbogacony o structured bindings

```
int main()
{
    map<int, int> graph;

    if(auto [it, success] = graph.insert(make_pair(0, 42));
       success)
    {
        // stuff
    }
}
```

## NOWE TYPY POMOCNICZE W BIBLIOTECE STANDARDOWEJ

Chodzi tutaj o `std::any`, `std::optional` oraz `std::variant`. Są one bardzo zbliżone do typów o analogicznych nazwach, ale różnią się nieznacznie semantyką i dostępnymi funkcjami.

2. Argument Dependant Lookup, znany również jako Koenig Lookup [A].

## Any

`std::any` (z nagłówka `<any>`) to typ opakowujący, potrafiący przechowywać dowolny przenaszalny lub kopialny typ. Aby odzyskać wartość, należy użyć `std::any_cast` ze statycznie znanym typem. Jeśli podany typ będzie niepoprawny, zostanie rzucony wyjątek `std::bad_any_cast`. Przykładowe użycie znajduje się w Listingu 13.

**Listing 13. Przykładowe użycie `std::any` [7]**

```
int main()
{
    std::any foo;
    foo = 42;
    std::cout << std::any_cast<int>(foo) << std::endl;
    foo = "answer"s;
    std::cout << std::any_cast<std::string>(foo) << std::endl;
}
```

## Optional

Zdefiniowany w nagłówku `<optional>` typ `std::optional` służy do opakowywania wartości, które mogą być potencjalnie puste, np. po wywołaniu API zakończonym niepowodzeniem. Jest to zblizone podejście do zwracania `nullptr` jako wskaźnika, tylko w ustandaryzowany sposób.

W odróżnieniu od wersji z Boost nie posiada on funkcji `get()`, zastępując ją `value()` oraz `value_or()`. Do odzyskania obiektu można też użyć przeładowanego operatora dereferencji (`operator*()`), lub bezpośrednio uzyskać dostęp do elementów przechowywanego obiektu za pomocą syntaktyki wskaźnikowej, udostępnionej przez przeładowany operator `->()`. Sprawdzenie, czy obiekt przechowuje wartość, odbywa się przez `explicit operator bool()` lub funkcję `has_value()`.

Przykładowe użycie `std::optional` znajduje się w Listingu 14.

**Listing 14. Przykładowe użycie `std::optional` [8]**

```
int main()
{
    std::optional<std::pair<int, std::string>> foo =
        std::make_pair(0, "foo"s);
    if(foo) {
        std::cout << foo->first << ", "
        << (*foo).second << std::endl;
    }
}
```

## Variant

Zwany też unią z tagiem (ang. *tagged union*) `std::variant` (`<variant>`) można rozumieć jako unię wiedzącą, który typ jest w niej w danym momencie zapisany. Pozwala to na uniknięcie powielania własnych implementacji tego podstawowego rozwiązania.

Sprawdzenie aktualnie przechowywanego typu odbywa się za pomocą funkcji `index()` lub `holds_alternative()`, a odzyskanie wartości za pomocą funkcji `std::get()` albo `get_if()`. Przykładowe użycie znajduje się w Listingu 15.

**Listing 15. Przykładowe użycie `std::variant` [9]**

```
int main()
{
    std::variant<int, std::string> foo = "foo"s;
    assert(std::holds_alternative<std::string>(foo));
    std::cout << std::get<std::string>(foo) << std::endl;
}
```

Opisane powyżej funkcje działają i są formalnie poprawne. Jednak idiomatycznie `std::variant` należy używać wraz z funktorami wizytującymi, za pomocą funkcji `std::visit()`. Dzięki temu można uniknąć drzewka `if-ów`, zastosować rozwiązania szablonowe, podzielić logicznie kod, stosując te same wizytatory dla różnych specjalizacji `std::variant`. Przykładowe użycie w Listingu 16.

**Listing 16. Użycie `std::visit` [B]**

```
struct visitor
{
    void operator()(int& val) const {
        std::cout << "int: " << val << '\n';
    }

    template<typename T> void operator()(T& t) const {
        std::cout << "templated: " << t << '\n';
    }
};

int main()
{
    std::variant<int, double, std::string> foo = 42;
    std::visit(visitor{}, foo);

    foo = 4.76;
    std::visit(visitor{}, foo);

    foo = "foo"s;
    std::visit(visitor{}, foo);
}
```

## STRING VIEW

Programiści C++ byli przyzwyczajeni do dość niewygodnego obchodzenia się z łańcuchami znaków, mając, bez dodatkowych bibliotek, dwa dostępne sposoby:

- » C-stringi, które w większości zastosowań sprawdzają się do wskaźników na pierwszy element ciągu, czyli `char const*`. Ich największym problemem jest to, że długość napisu nie jest przekazywana wraz ze wskaźnikiem, więc często zbędnie powtarzane jest obliczanie tejże długości, a przekazanie wycinka napisu wymaga innego wywołania.
- » Użycie typu `std::string`, ale jego wadą jest wymuszanie zbędnych kopii i alokacji (pomijając optymalizację małych stringów, ang. *small string optimization* [C] [D]).

C++17 wprowadza nową klasę – widok na ciąg znaków: `std::string_view`. Nie odpowiada ona za zwolnienie zasobów, a wyłącznie za przekazanie informacji o początku i długości ciągu. Wiele z funkcji operujących na `std::string` zostało wyposażonych w przeładowania akceptujące także `std::string_view`. Dodany również został literał `sv` do tworzenia instancji `std::string_view` z literalów ciągów znakowych. Przykład użycia znajduje się w Listingu 17.

**Listing 17. Przykład użycia `std::string_view` [E]**

```
void foo(std::string_view v)
{
    std::cout << v << "\n"sv;
}

int main()
{
    auto bar = "baz"s;
    foo(bar);
}
```

## DEDUKCJA PARAMETRÓW SZABLOŃOW KLAS

Dedukcja typów w szablonach funkcji była obecna w C++ od pierwszego standardu, ale aby utworzyć obiekt, należało jawnie podać parametry, co często prowadziło do duplikacji informacji zawartych w kodzie. Aby tego uniknąć, powstały obejścia w postaci serii funkcji `make_x`, tworzących instancję odpowiednio skonkretyzowanego typu szablonowego na podstawie przekazanych argumentów. Przykłady tego podejścia można znaleźć w Listingu 18, a w Listingu 19 przedstawiono jawnie przekazanie parametrów szablonów.

**Listing 18. Użycie funkcji `std::make_pair` i `std::make_tuple` [10]**

```
int main()
{
    auto p = std::make_pair(42, "answer"s);
    auto t = std::make_tuple(p, "foo"s, 3.14);
}
```

**Listing 19. Kod analogiczny do tego z Listingu 18, ale bez użycia funkcji `make_x`. [10]**

```
int main()
{
    std::pair<int, std::string> p{42, "answer"s};
    std::tuple<
        std::pair<int, std::string>,
        std::string,
        double
    > t{p, "foo"s, 3.14};
}
```

C++17 pozwala pozbyć się większości funkcji z tej rodziny<sup>3</sup>, wprowadzając dedukcję typów dla szablonów klas. Techniczna specyfikacja tego zachowania jest dość skomplikowana (więcej w: [F]), ale sprawdza się do tego, że kompilator porównuje argumenty przekazane jako inicjalizatory do dostępnych konstruktorów. Dzięki temu kod z Listingu 19 można znaczco uprościć, co przedstawiono w Listingu 20.

**Listing 20. Uproszczenie kodu z Listingu 19 [11]**

```
int main()
{
    std::pair p{42, "answer"s};
    std::tuple t{p, "foo"s, 3.14};
}
```

Jeśli powiązanie między inicjalizatorem a parametrem szablonu jest nietrywialne, na przykład w przypadku `std::vector` inicjalizowanego parą iteratorów, twórca klasy może zdefiniować własne sugestie w postaci prowadnic (ang. *deduction guides*). Przykład na podstawie klasy `foo` znajduje się w Listingu 21.

**Listing 21. Definiowanie własnych deduction guides [12]**

```
template<typename T, size_t I>
struct foo
{
    static constexpr auto value = I;
    using type = T;

    template<typename U> foo(T, U){}
};
```

3. Należy jednak zauważać, że nie wszystkie funkcje pasujące do tego nazewnictwa przestały być użyteczne – nie jest tak np. w przypadku `std::make_unique` lub `std::make_shared`. W uproszczeniu zasada brzmi następująco: jeśli funkcja `make_x` wymagała jawnego podania parametru szablonowego, to nadal może być przydatna.

```
template<typename T, typename U>
foo(T, U) -> foo<T, U::value>;
```

```
int main()
{
    foo f{
        std::string{},
        std::integral_constant<int, 42>{}
    };

    static_assert(f.value == 42);
    static_assert(
        std::is_same_v<decltype(f)::type, std::string>
    );
}
```

## ALGORYTMY WSPÓŁBIEŻNE

Wiele algorytmów dostępnych w nagłówkach `<algorithm>` i `<numeric>` dostało alternatywny zestaw przeładowań akceptujących klasy definiujące zasady wykonania (ang. *execution policy*). Standard definiuje trzy takie klasy, ale poszczególne implementacje mogą oczywiście wzbogacić wybór:

- » `std::execution::sequenced_policy` – wykonanie sekwencyjne,
- » `std::execution::parallel_policy` – wykonanie może być zrównoległe,
- » `std::execution::parallel_unsequenced_policy` – wykonanie może być zrównoległe i zwektryzowane.

Standard dostarcza globalne instancje powyższych klas, których należy używać, aby wybrać zasadę paralelizacji. Są to odpowiednio `std::execution::seq`, `std::execution::par` i `std::execution::par_unseq`.

We wszystkich wymienionych przypadkach kolejność nie jest sprecyzowana, co oznacza, że wywołanie korzystające z `std::execution::sequenced_policy` nie jest równoważne klasycznemu wywołaniu.

Przykładowe użycie znajduje się w Listingu 22. Autor zaznacza jednak, że w chwili publikacji artykułu żaden kompilator jeszcze nie wspierał tej części nowego standardu.

**Listing 22. Przykładowe wykorzystanie zrównoległonego wykonania**

```
int main()
{
    std::vector foo{1,2,3,4,5};
    auto result = std::reduce(
        std::execution::par,
        foo.cbegin(),
        foo.cend(),
        std::multiplies{}
    );
    std::cout << result << '\n';
}
```

## FILESYSTEM

Nowy nagłówek wzorowany na bibliotece o identycznej nazwie wchodzącej w skład Boosta. W końcu programiści C++ mogą za pomocą biblioteki standardowej operować na systemie plików. Przykładowy kod znajduje się w Listingu 23.

**Listing 23. Przykładowe użycie biblioteki `filesystem` [13]**

```
#include <filesystem>
#include <initializer_list>
#include <iostream>

namespace fs = std::filesystem;
```

```

int main()
{
    fs::create_directory("test");
    for(auto const& el : fs::directory_iterator(".")) {
        std::cout << el << '\n';
    }
}

```

Uwaga: w chwili pisania tego artykułu najnowsza wersja gcc ma tylko eksperymentalną implementację tej biblioteki, co powoduje, że jest ona faktycznie w nagłówku `<experimental/filesystem>`, przestrzeni nazw `std::experimental::filesystem` oraz wymaga linkowania biblioteki `stdc++fs (-lstdc++fs do opcji linkera)`.

## USPRAWNIENIA ATRYBUTÓW, NOWE ATRYBUTY

W nowym standardzie komitet kontynuuje sukcesywne rozszerzanie przydatności wprowadzonych w C++11 atrybutów:

- » od teraz atrybuty mogą być nadawane przestrzeniom nazw i pojedynczym wartościom w enumeracjach (Listing 24),
- » uściślono, co kompilator powinien robić, napotykając nieznane atrybuty – powinien je po prostu ignorować,
- » wprowadzono alternatywny uproszczony zapis atrybutów w przestrzeniach nazw (Listing 25),
- » wprowadzono trzy nowe standardowe atrybuty (wszystkie znajdują się w Tabeli 2):
  - » `[[fallthrough]]`,
  - » `[[maybe_unused]]`,
  - » `[[nodiscard]]`.

**Listing 24. Atrybuty w nowych miejscach**

```

namespace [[deprecated("use foo")]] bar
{
}

enum class Foo
{
    foo,
    bar [[deprecated("foo > bar")]]
};

```

**Listing 25. Alternatywny zapis atrybutów w porównaniu do klasycznego**

```

[[long_namespace_name::foo, long_namespace_name::bar]]
void foo();

[[using long_namespace_name: foo, bar]]
void bar();

```

## IF CONSTEXPR

Dotychczas w C++, tak jak i w wielu innych językach, kod w obu odnóżach instrukcji warunkowej `if` musiał być poprawny zarówno syntaktycznie, jak i semantycznie. Było to wymagane, nawet gdy w trakcie komplikacji wartość wyrażenia warunkowego była znana i stała, powodując, że jedna z odnów będzie martwym kodem.

Przykładowy niekomplilujący się kod znajduje się w Listingu 26. Kompilator poinformuje, że `int` nie ma metody `size()` – niezależnie od tego, że nie ma możliwości, aby była ona na tym typie wywołana.

Tradycyjnym rozwiązaniem tego problemu było podzielenie kodu odpowiedzialnego za pracę na typach o różnych charakterystykach na różne szablony funkcji i *tag dispatching* pomiędzy nimi. Przedstawiono to w Listingu 27. Typ `std::is_same<T, std::string>` dziedziczy po `std::true_type` albo

Atrybut	std	Opis
<code>[[carries_dependency]]</code>	C++11	Informacja dla optymalizatora o zachowaniu zależności <code>std::memory_order</code>
<code>[[deprecated]]</code> <code>[[deprecated("why")]]</code>	C++14	Oznaczenie przestarzałości, z opcjonalnym powodem
<code>[[fallthrough]]</code>	C++17	Informacja, że brak instrukcji <code>break</code> ; pomiędzy <code>case'ami</code> w <code>switchu</code> jest celowy
<code>[[maybe_unused]]</code>	C++17	Informuje kompilator, że zmienna może pozostać nieużyta i nie jest to pomyłka programisty
<code>[[nodiscard]]</code>	C++17	Informacja, że rezultat tak oznaczonyj funkcji nigdy nie powinien być ignorowany. W przypadku aplikacji do typu jest to aplikowane do wszystkich funkcji zwracających obiekty tego typu
<code>[[noreturn]]</code>	C++11	Określa, że wykonanie funkcji nigdy nie wróci do obecnego zakresu wykonania, np. <code>std::terminate()</code>

*Tabela 2. Atrybuty w C++*

`std::false_type`, w zależności od tego, czy podane typy są identyczne. Ponieważ między tymi dwoma typami nie ma możliwości konwersji, inicjalizowane jest tylko jedno przeładowanie `foo_impl()`, poprawne dla danego `T`.

**Listing 26. Niepoprawny kod, int nie ma metody size(), nawet jeśli nigdy nie jest ona wywoływana [15]**

```

template<typename T>
void foo(T t)
{
    if(std::is_same_v<T, std::string>) {
        std::cout << "string: " << t <<
        " size: " << t.size() << '\n';
    } else {
        std::cout << "other type: " << t << '\n';
    }
}

int main()
{
    foo("123"s);
    foo(456);
}

```

**Listing 27. Poprawiony kod z Listingu 26 – komplikacja przebiega poprawnie i działa zgodnie z zamierzeniami [16]**

```

template<typename T>
void foo_impl(T t, std::true_type)
{
    std::cout << "string: " << t <<
    " size: " << t.size() << '\n';
}

template<typename T>
void foo_impl(T t, std::false_type)
{
    std::cout << "other type: " << t << '\n';
}

template<typename T>
void foo(T t)
{
    foo_impl(t, std::is_same<T, std::string>());
}

int main()
{
    foo("123"s);
    foo(456);
}

```

C++17 wprowadza `if constexpr`, który rezygnuje z wymogu semantycznej poprawności kodu odnów – musi się on jedynie poprawnie parsować. Wobec tego bogate w słowa rozwiązań z Listingu 27 można zamienić na to z Listingu 28. Różni się ono od tego z Listingu 26 dodaniem tylko jednego słowa: `constexpr`.

#### Listing 28. If `constexpr` [17]

```
template<typename T>
void foo(T t)
{
    if constexpr(std::is_same_v<T, std::string>) {
        std::cout << "string: " << t <<
        " size: " << t.size() << '\n';
    } else {
        std::cout << "other type: " << t << '\n';
    }
}

int main()
{
    foo("123"s);
    foo(456);
}
```

Jeśli warunków jest więcej, należy umieścić `constexpr` po każdym `if-ie`. Może to wydawać się zbyteczne, ale jest to spowodowane tym, jak gramatyka C++ określa `else if`. Może to być zaskakujące nawet dla doświadczonych programistów, ale takiego konstruktu nie ma wcale. Kod z Listingu 29 jest rozumiany przez kompilator tak samo jak ten z Listingu 30. Konieczność używania `constexpr` po każdym `if-ie` powinna być w tym momencie zrozumiała.

#### Listing 29. Drabinka `if-ów`

```
if(a)
    foo();
else if(b)
    bar();
else
    baz();
```

#### Listing 30. Kod z Listingu 29 w interpretacji kompilatora

```
if(a) {
    foo();
} else {
    if(b) {
        bar();
    } else {
        baz();
    }
}
```

## FOLD EXPRESSIONS

Znana programistom języków funkcyjnych rodzina funkcji wyższego rzędu *fold* została wprowadzona do C++. Dzięki temu nie ma konieczności pisania pseudo-rekurencyjnych wywołań z coraz mniejszą liczbą argumentów funkcji. Dostępne są cztery formy wyrażeń *fold* (ang. *fold expression*):

- »  $(... @ E)$  – jednoargumentowy *left fold* –  $((E_1 @ E_2) @ ... ) @ E_n$
- »  $(E @ ...)$  – jednoargumentowy *right fold* –  $E_1 @ (... @ (E_{n-1} @ E_n))$
- »  $(V @ ... @ E)$  – dwuargumentowy *left fold* –  $((V @ E_1) @ E_2) @ ... @ E_n$
- »  $(E @ ... @ V)$  – dwuargumentowy *right fold* –  $E_1 @ (... @ (E_{n-1} @ (E_n @ V)))$

W Listingu 31 przedstawiono przykładową funkcję sumującą wszystkie argumenty bez zastosowania wyrażeń *fold*. Autor zwraca uwagę, że kod już został uproszczony dzięki `if constexpr`. W Listingu 32 przedstawiono analogiczny kod z ich wykorzystaniem.

#### Listing 31. Sumowanie parametrów funkcji bez wyrażeń *fold*

```
template<typename T, typename... Us>
auto sum_params(T t, Us... us)
{
    if constexpr(sizeof...(Us) == 0) {
        return t;
    } else {
        return t + sum_params(us...);
    }
}

int main()
{
    std::cout << sum_params(1, 2L, 3.f, 4.0, 5);
}
```

#### Listing 32. Wyrażenia *fold* w praktyce

```
template<typename... Us>
auto sum_params(Us... us)
{
    return (... + us);
}

int main()
{
    std::cout << sum_params(1, 2L, 3.f, 4.0, 5);
}
```

## AUTO TEMPLATES

Jest to kolejna zmiana mająca na celu zwiększenie czytelności kodu. Dotychczas, aby użyć parametru szablonu niebędącego typem, należało podać jego typ. Jeśli typ ten nie był znany, musiał być innym parametrem szablonu. Za przykład może posłużyć tutaj obecny w bibliotece standardowej typ `std::integral_constant`, który musi być konkretyzowany poprzez podanie zarówno typu, jak i wartości, np. `std::integral_constant<size_t, 42>`. Od C++17 parametr szablonu może być oznaczony jako `auto`, wtedy jego typ będzie dedukowany z podanego argumentu.

Powyższy przypadek można uznać jeszcze za w miarę czytelny, ale już w przypadku wskaźnika na funkcję jest to znaczco mniej wygodne. W Listingu 33 przedstawiono życiową sytuację – opakowanie wskaźnika na funkcję celem przekazania go jako parametru *deleter* do `std::unique_ptr` [18]. W Listingu 34 przedstawiono semantycznie ekwiwalentny kod, w opinii autora znacznie czytelniejszy.

#### Listing 33. Definicja `foo` jako `unique_ptr` do przetrzymywania FILE\*

```
template<typename T, T* func>
struct function_caller
{
    template<typename... Us>
    auto operator()(Us&&... us) const
        -> decltype(func(std::forward<Us...>(us...)));
    {
        return func(std::forward<Us...>(us...));
    }

    using foo =
        std::unique_ptr<
            FILE,
            function_caller<decltype(fclose), &fclose>>;
};
```

#### Listing 34. Kod odpowiadający temu z Listingu 33, z parametrem szablonu `auto`

```
template<auto func>
struct function_caller
{
    template<typename... Us>
```

```

auto operator()(Us&&... us) const
{
    return func(std::forward<Us...>(us...));
}
};

using foo = unique_ptr<FILE, function_caller<&fclose>>;

```

## INNE ZMIANY

W tej sekcji znajdują się zmiany istotne dla konkretnych grup programistów (np. twórcy bibliotek) lub drobne, choć dostatecznie istotne, aby o nich wspomnieć.

### Zmiana znaczenia listy inicjalizacyjnej z auto

Od teraz, przy inicjalizacji dedukowanego typu (auto) bez znaku równości, lista inicjalizacyjna może mieć tylko jeden element i to jego typ zostanie użyty. Choć oficjalnym powodem tej zmiany jest zwiększenie intuicyjności, w opinii autora jest dokładnie odwrotnie, ponieważ złamana została zasada, że inicjalizacja typ nazwa{init} jest równoznaczna z typ nazwa = {init}.

	C++14	C++17
auto foo{1};	std::initializer_list<int>	int
auto foo{1,2};	std::initializer_list<int>	Niepoprawny kod
auto foo = {1};	std::initializer_list<int>	std::initializer_list<int>
auto foo = {1,2};	std::initializer_list<int>	std::initializer_list<int>

Tabela 3. Dedukcja typów w C++14 i C++17

### Static assert bez wiadomości

Jeśli warunek jest oczywisty, od teraz można wywoływać static\_assert bez podawania wiadomości.

### Ranged for z różnymi typami begin/end

Jest to zmiana mająca na celu ułatwienie wprowadzenia biblioteki ranges, lub implementację własnych bibliotek o podobnych funkcjonalnościach. Dzięki innemu typowi końca zakresu można w prosty sposób zmienić zachowanie operatora porównania i, na przykład, oznajmić, że łańcuch znakowy się kończy, gdy jego ostatni znak to null, bez względu na jego pozycję. Bardzo dokładnie opisał to Eric Niebler na swoim blogu [1A] [1B] [1C] [1D].

### Noexcept wchodzi do sygnatury funkcji

Od C++17 noexcept jest częścią typu funkcji, przez co w pełni poprawny kod C++14 z Listingu 35 spowoduje błąd komplikacji w najnowszym standardzie. Jest tak, ponieważ funkcje foo i bar są typu void() w C++14, ale bar jest typu void() noexcept w C++17.

### Listing 35. Zobrazowanie zmian w języku – noexcept staje się częścią typu funkcji

```

void foo() {}
void bar() noexcept {}

int main()
{
    using foo_t = decltype(foo);
    using bar_t = decltype(bar);
    static_assert(std::is_same<foo_t, bar_t>::value, "");
}

```

## Akceptacja słowa kluczowego typename w szablonowych parametrach szablonów

W deklaracji szablonu, którego parametrem był kolejny szablon, tego zewnętrznego szablonu nie można było nazwać, używając słowa kluczowego typename, a wyłącznie class. Od C++17 już można.

### Listing 36. Kod poprawny również w C++14

```

template<template <typename T> class foo>
struct bar{}

```

### Listing 37. Kod poprawny tylko w C++17

```

template<template <typename T> typename foo>
struct bar{}

```

## Zmienne inline

Od C++17 można stosować słowo kluczowe *inline* do definicji zmiennych globalnych. Zgodnie z nieoczywistym znaczeniem tego słowa kluczowego [1E] również w tym przypadku chodzi o zezwolenie na wielokrotną definicję zmiennej bez łamania zasady pojedynczej definicji (ang. *One-definition rule*).

W Listingu 38 przedstawiono zastosowanie zmiennych *inline* w C++17. W Listingu 39 można znaleźć emulację zbliżonego rozwiązania za pomocą szablonów w C++11.

### Listing 38. Zmienne inline

```

inline std::string const version = "1.0.0.42";

struct foo
{
    inline static std::string mut = "foo";
};

```

### Listing 39. Rozwiążanie poprawne w C++11 zbliżone do tego z Listingu 38

```

template<typename>
struct foo_impl
{
    static std::string mut;
};

template<typename T>
std::string foo_impl<T>::mut = "";

using foo = foo_impl<void>;

```

## Nowe funkcje matematyczne

Często używana do obliczania odległości na kartezjańskiej siatce współrzędnych funkcja std::hypot() doczekała się przeładowania dla trzech parametrów. Od C++17 pojawiła się w bibliotece standardowej funkcja do obliczania najmniejszej wspólnej wielokrotności – std::lcm(), oraz największego wspólnego dzielnika – std::gcd(). Warta uwagi jest również funkcja std::clamp(), zamkijająca wartość w podanym przedziale – można ją zobrazować w następujący sposób:

### Listing 40. Pseudokod obrazujący działanie funkcji std::clamp

```

template<typename T>
auto clamp(T a, T b, T c)
{
    return std::max(b, std::min(a, c));
}

```

## to\_chars/from\_chars

Dostępne w nagłówku `<charconv>` funkcje o mocno ograniczonej funkcjonalności przeznaczone do, odpowiednio, serializacji i deserializacji liczb. Dzięki temu, że nie rzucają wyjątków, nie obsługują `locale` oraz nie alokują pamięci, można spodziewać się ich wysokiej wydajności. Czyni je to kandydatami do użycia w bibliotekach serializujących, np. `json`, i w systemach wbudowanych, gdzie wysoka wydajność jest ważniejsza niż lekko kuriozalne API.

W Listingu 41 pokazano przykładowe użycie tych funkcji, wraz z nietypowym sposobem, w jaki można sprawdzić, czy wywołanie się powidło.

**Listing 41. Wykorzystanie `from_chars` i `to_chars` [28]**

```
int main()
{
    char buf[] = "42.5";
    int val;
    auto r = std::from_chars(buf, buf+sizeof(buf), val);
    if(auto&& [ptr, err] = r; !bool(err)) {
        // prints 42
        std::cout << val << '\n';
    }
    char out[32] = {};
    // base 21
    auto r = std::to_chars(out, out+32, val, 21);
    if(auto&& [ptr, err] = r; !bool(err)) {
        auto distance = ptr - out;
        // prints 20 (42 is 20_21)
        std::cout << std::string_view{out, distance};
    }
}
```

## Gwarantowana optymalizacja RVO

RVO – *return value optimization* (również *copy elision*) [1F] – to optymalizacja, która była opcjonalna od C++98, a teraz staje się obowiązkowa. Jeśli inicjalizowany jest obiekt wynikiem funkcji, która inicjalizuje obiekt tego samego typu (lub konwertowalnego do niego) w instrukcji `return`, kompilator miał prawo (teraz obowiązek) nie wygenerować obiektów tymczasowych, tylko zainicjalizować obiekt docelowy, pomijając przy tym wykonanie konstruktorów kopiących/przenoszenia. Przed C++17 te konstruktorły musiały być dostępne, ponieważ ta optymalizacja była opcjonalna; od teraz nie są, co pozwala zwracać nieprzenaszalne i niekopialne klasy z funkcji. W Listingu 42 pokazano to na przykładzie `std::mutex`.

**Listing 42. Zwracanie nieprzenaszalnego i niekopialnego typu `std::mutex` z funkcji**

```
std::mutex make_mutex()
{
    return std::mutex{};
}

int main()
{
    auto m = make_mutex();
}
```

## *Using z ...*

Celem ułatwienia np. dziedziczenia po operatorach wywołania wszystkich argumentów paczki szablonu, `using` może korzystać z rozpakowywania za pomocą operatora trzech kropiek.

## *Listing 43. Using z ...*

```
template<typename... T>
struct simple_overloader : T...
{
    simple_overloader(T... ts) : T{ts}... {}
    using T::operator()...;
};

int main()
{
    simple_overloader foo{
        [](int){ return 42; },
        [](double x){ return x * 2; }
    };
    std::cout << foo(0) << ", " <<
        foo(668.5) << '\n';
}
```

## *Tablicowy shared\_ptr*

Wprowadzono obecne w `std::unique_ptr` przeładowanie dla tablic.

**Listing 44. Tablicowy `shared_ptr`**

```
int main()
{
    std::shared_ptr<std::string[]> x(new std::string[10]);
    x[0] = "foo";
}
```

## *Constexpr lambdy*

Od C++17 lambdy mogą znajdować się w wyrażeniach obliczanych w trakcie komplikacji. Są też niejawnie deklarowane jako `constexpr` przez kompilator, jeśli istnieje taka możliwość.

## *Kopia \*this w lambdach*

Wprowadzono możliwość przekazania `*this` jako elementu listy obiektów przekazanych do wyrażenia lambda. Dzięki temu wewnętrz lambdy nie trzeba się martwić czasem życia obiektu zewnętrznego.

**Listing 45. Przekazanie `*this` do lambdy**

```
struct foo
{
    int x;

    auto cpp14()
    {
        return [copy = *this]{
            return copy.x;
        };
    }

    auto cpp17()
    {
        return [*this]{
            return this->x;
        };
    };

    int main()
    {
        auto a = foo{42}.cpp14();
        auto b = foo{42}.cpp17();

        std::cout << a() << ", " <<
            b() << '\n';
    }
}
```

## Przenoszenie elementów map/setów bez realokacji

Klasy `std::set`, `std::map`, `std::multiset`, `std::multimap` oraz ich odpowiedniki z `unordered` w nazwie wzbogacone zostały o funkcje `merge()` i `extract()`, a ich funkcje `insert()` o przeładowania obsługujące typ zwracany przez `extract()`. Pozwalały one na przenoszenie elementów pomiędzy różnymi obiektami tych klas, gdy spełnione zostaną następujące warunki:

- » kontenery są tego samego typu lub są swoimi odpowiednikami zezwalającymi lub zabraniającymi powtarzania kluczy,
- » typ elementu jest identyczny,
- » typ alokatora jest identyczny.

Porównanie lub hash nie muszą się zgadzać, więc można przenosić elementy np. pomiędzy odwrotnie posortowanymi instancjami `std::set`, co pokazano w Listingu 46.

**Listing 46. Użycie extract() i insert()**

```
int main()
{
    std::set<int, std::greater<>> a{1,2,3};
    std::set<int, std::less<>> b{2,3};

    b.insert(a.extract(1));
    assert((b == std::set<int, std::less<>>{1,2,3}));
    assert((a == std::set<int, std::greater<>>{2,3}));
}
```

## Krótszy zapis zagnieżdżonych przestrzeni nazw

Od C++17 zapis `namespace foo::bar {}` jest poprawny i nie trzeba zapisywać każdej przestrzeni ręcznie:  
`namespace foo{ namespace bar {} }.`

## has\_include

Wprowadzony został standardowy sposób na sprawdzenie, czy dany plik nagłówkowy jest dostępny w momencie komplikacji. Pozwala to na użycie biblioteki tylko jeśli jest dostępna, lub dostarczenia zastępczej funkcjonalności, jeśli preferowanej brak. Sztuczny przykład znajduje się w Listingu 47.

**Listing 47. Użycie \_\_has\_include**

```
#if __has_include(<memory>
#include <memory>
template<typename T>
using shared_ptr = std::shared_ptr<T>;
#elif __has_include(<boost/shared_ptr.hpp>
#include <boost/shared_ptr.hpp>
template<typename T>
using shared_ptr = boost::shared_ptr<T>;
#else
static_assert(false, "no shared_ptr available");
#endif
```

## Algorytmy szukające

Funkcja `std::search()` zyskała przeładowanie akceptujące własne algorytmy szukające. Wraz z tym w standardzie zdefiniowano trzy takie algorytmy:

- » `std::default_searcher` – domyślny algorytm szukający z biblioteki standardowej,
- » `std::boyer_moore_searcher` – algorytm szukający Boyera-Moore'a dla stringów,

» `std::boyer_moore_horspool_searcher` – algorytm szukający Boyera-Moore'a-Horspools dla stringów.

**Listing 48. Przykład użycia std::boyer\_moore\_searcher [20]**

```
int main()
{
    auto txt = "foo bar baz"s;
    auto sought = "bar"s;

    std::boyer_moore_searcher searcher{
        sought.begin(),
        sought.end()
    };

    auto result = std::search(
        txt.begin(),
        txt.end(),
        searcher
    );

    if(result != txt.end()) {
        auto pos = std::distance(txt.begin(), result);
        std::cout << "Found at position " << pos << '\n';
    } else {
        std::cout << "Not found\n";
    }
}
```

## Ułatwienia w metaprogramowaniu

Wprowadzone zostają standardowe implementacje funkcji `std::apply()`, `std::invoke()` oraz typu `std::void_t`. Obok wprowadzonych do standardu C++14 typów `std::index_sequence`/`std::integer_sequence` są one podstawą metaprogramowania. Chociaż wszystkie wymienione można z powodzeniem zaimplementować już w C++11, to standaryzacja zezwoli na swobodne ich wykorzystanie w kodzie, bez obaw o wielokrotną implementację – ani bez jej konieczności.

`std::invoke()` oferuje stały interfejs do wywoływanego funkcji, zbliżony do tego z `std::bind()` lub `std::thread`. Pierwszym parametrem jest *callable*, czyli dowolny wywoływalny obiekt lub wskaźnik. Jeśli pierwszym argumentem jest wskaźnik do niestatycznej funkcji klasy, to następnym argumentem jest obiekt, na którym zostanie on wywołany. Pozostałe argumenty to argumenty przekazywane do wywoływanej funkcji. Pozwala to na ujednolicenie wywołań przy pisaniu generycznych funkcji wyższego rzędu.

**Listing 49. Przykładowe użycie std::invoke [21]**

```
int main()
{
    auto ptr = &std::string::size;
    std::cout << std::invoke(ptr, "123"s) << '\n';

    char up = std::invoke(&::toupper, 'x');
    std::cout << up << '\n';
}
```

`std::apply()` pozwala na przekazanie jako argumentów funkcji elementów klasy `std::tuple` lub innej, zgodnej z nią. Zgodna klasa posiada odpowiednią specjalizację `std::tuple_size` oraz wspiera funkcję `std::get()`.

**Listing 50. Przykładowe użycie std::apply [22]**

```
int main()
{
    std::tuple t{2, 10.0};
    std::cout << std::apply(&::pow, t);
}
```

Are you  
proficient in IT?

So are we. Join us.

# Software Architect **wanted!**

More offer details:

<http://nokiawroclaw.pl/oferty-pracy/cc-sw-architect/>



`std::void_t` to alias dowolnego typu lub sekwencji typów na `void`. Jest to bardzo użyteczne narzędzie w powiązaniu ze SFINAE<sup>4</sup> i pozwala na odrzucanie niepoprawnego kodu bez powodowania błędu komplikacji. W Listingu 51 zawarto przykład oparty na przykładzie z `cppreference` [23].

## Listing 51. Przykładowe użycie `std::void_t` [24]

```
template<class, class = std::void_t<>>
struct has_type_member:
    std::false_type{};

template<class T>
struct has_type_member<T, std::void_t<typename T::type>>:
    std::true_type{};

int main()
{
    std::cout << has_type_member<int>{} << '\n' <<
    has_type_member<std::false_type>{};
}
```

## Polimorficzne alokatory

C++17 wprowadził polimorficzne alokatory w przestrzeni nazw `std::pmr`. Wraz z nimi w tej przestrzeni pojawiły się odpowiednio dostosowane kontenery, np. `std::pmr::vector`. Polimorficzne alokatory, jak nazwa wskazuje, mogą wykazywać różne zachowanie dla różnych instancji tego samego typu alokatora. Pozwala to na lepszą współpracę własnych alokatorów z kontenerami standardowymi.

## Zarezerwowane przestrzenie nazw

Zarezerwowano dla biblioteki standardowej C++ wszystkie przestrzenie nazw, które można opisać wyrażeniem regularnym `:/std\/*`. Czyli na przykład `std`, `std2` lub `std2017` w globalnej przestrzeni nazw.

## `std::launder`

Nazwa tej funkcji jest analogią do prania pieniędzy (ang. *money laundering*), lecz odnosi się do pamięci. Tak jak wyprane pieniądze stają się znów legalnym środkiem płatniczym, ponieważ służby nie są w stanie wysledzić ich pochodzenia, tak „wyprana” pamięć może być użyta do innych celów, ponieważ kompilator „zapomina”, co w niej wcześniej było. Ma to istotne znaczenie, gdy w danym miejscu znajdował się obiekt, którego odczyty kompilator miał prawo zoptymalizować.

Jej zastosowanie pokazano w Listingu 52 zaczerpniętym z serwisu StackOverflow [27]. Bez zastosowania `std::launder()` asercja mogłaby zakończyć się niepowodzeniem, ponieważ kompilator miał prawo zoptymalizować dostęp do `u.x.n`, które jest stałą.

## Listing 52. Użycie `std::launder()`

```
int main()
{
    struct X { const int n; };
    union U { X x; float f; };

    U u = {{ 1 }};
    X *p = new (&u.x) X {2};

    assert(*std::launder(&u.x.n) == 2);
}
```

## PODSUMOWANIE

Pomimo że niniejszy artykuł jest obszerny, to i tak nie opisano w nim wszystkich zmian i nowości w C++17. W opinii autora te najważniejsze zostały jednak poruszone. Ocenę, czy C++17 spełnia pokłady w nim nadzieję – zarówno pod kątem obiecanych funkcjonalności, jak i jako założenia znaczący standard – autor pozostawia czytelnikom.

## W sieci:

- [1]: <https://herbsutter.com/2017/09/06/c17-is-formally-approved/>
- [2]: <https://www.iso.org/standard/68564.html>
- [3]: <https://goo.gl/78oE9C>
- [4]: <https://goo.gl/TbTv7Y>
- [5]: <https://www.ioccc.org/2013/guidelines.txt>
- [6]: <https://wandbox.org/permlink/Oolg8K4wZxRPZtIP>
- [7]: <https://wandbox.org/permlink/RPSCKS5GpbWiLS79>
- [8]: <https://wandbox.org/permlink/pRflCz6TTMn78yBZ>
- [9]: <https://wandbox.org/permlink/AFBAirwnaKmDBjx8>
- [A]: <http://en.cppreference.com/w/cpp/language/adl>
- [B]: <https://wandbox.org/permlink/HNlExe4tTV9nwij4>
- [C]: <https://stackoverflow.com/a/21710033/2456565>
- [D]: <https://youtu.be/kPR8h4-qZdk>
- [E]: <https://wandbox.org/permlink/AC8e1O3OUzfsN1Kt>
- [F]: <https://goo.gl/vw4vo8>
- [10]: <https://wandbox.org/permlink/Hia6pmnsgDHv0kMk>
- [11]: <https://wandbox.org/permlink/fOVWvngwyEBhbMg9>
- [12]: <https://wandbox.org/permlink/EWeanhKbj26qCoPm>
- [13]: <https://wandbox.org/permlink/a9xsieNobtWhh8Q>
- [14]: <https://wandbox.org/permlink/ZYznuRoTkdlV5FA>
- [15]: <https://wandbox.org/permlink/FGsJRqEXc9gZcmmp>
- [16]: <https://wandbox.org/permlink/bKQtrDhrcvANG6I9>
- [17]: <https://wandbox.org/permlink/SyBUipzYxmBy3lzl>
- [18]: <https://goo.gl/k1XD4A>
- [19]: <https://timsong-cpp.github.io/cppwp/n4659/diff.cpp14.dcl.dcl>
- [1A]: <http://ericniebler.com/2014/02/16/delimited-ranges/>
- [1B]: <http://ericniebler.com/2014/02/18/infinite-ranges/>
- [1C]: <http://ericniebler.com/2014/02/21/introducing-iterables/>
- [1D]: <http://ericniebler.com/2014/02/27/ranges-infinity-and-beyond/>
- [1E]: <https://dsp.krzaq.cc/post/352/co-oznaca-słowo-kluczowe-inline/>
- [1F]: [https://en.wikipedia.org/wiki/Copy\\_elision](https://en.wikipedia.org/wiki/Copy_elision)
- [20]: <https://wandbox.org/permlink/60Mu1Os3deo4iLPi>
- [21]: <https://wandbox.org/permlink/uRcg3ufDpz5gsn9z>
- [22]: <https://wandbox.org/permlink/p5e8tzlxkbS6bKKw>
- [23]: [http://en.cppreference.com/w/cpp/types/void\\_t](http://en.cppreference.com/w/cpp/types/void_t)
- [24]: <https://wandbox.org/permlink/kAsnMoyGIX5gSVHI>
- [25]: [http://en.cppreference.com/w/cpp/language/operator\\_alternative](http://en.cppreference.com/w/cpp/language/operator_alternative)
- [26]: <https://goo.gl/BwvCWv>
- [27]: <https://stackoverflow.com/a/39382728/2456565>
- [28]: <https://wandbox.org/permlink/2ZQzVQq2oXAnCOlz>

## PAWEŁ "KRZAQ" ZAKRZEWSKI

<https://dev.krzaq.cc>

Absolwent Automatyki i Robotyki na Zachodniopomorskim Uniwersytecie Technologicznym. Pracuje jako starszy programista C++ w firmie Huuuge Games. Programowaniem interesuje się od dzieciństwa, jego ostatnie zainteresowania to C++ i metaprogramowanie.

4. Niemożność podstawienia [parametru szablonu] nie jest błędem [kompilacji] (ang. Substitution Failure Is Not An Error).

# SZUKASZ PRACY MARZEŃ?



## ZRÓB PIERWSZY KROK!

**PRZYJDŹ DO SZKOŁY JĘZYKA ANGIELSKIEGO SPEAK UP**

- 🕒 NAUCZYMY CIĘ ANGIELSKIEGO SZYBKO I SKUTECZNIE
- ▶ STAWIAMY NA TWÓJ ROZWÓJ POPRZEZ INNOWACYJNE METODY NAUCZANIA
- ⌚ GWARANTUJEMY ELASTYCZNOŚĆ I INDYWIDUALNE PODJĘŚCIE
- 🏆 Z NAMI ZDOBĘDZIESZ CERTYFIKAT JĘZYKOWY

The background image shows a panoramic view of the Kuala Lumpur city skyline at dusk or night. The iconic Petronas Twin Towers are brightly lit, standing prominently against the dark sky. Numerous other skyscrapers and buildings are visible, their lights reflecting off the water in the foreground. In the bottom right corner, there is a large red circular logo for "SPEAK UP THE ENGLISH SCHOOL". The logo contains the text "THE ENGLISH SCHOOL" in small white letters above the word "SPEAK" in large white letters, with "UP" in black letters below it.

# Bariery pamięci

W czasach kiedy jednym z wąskich gardeł oprogramowania jest dostęp procesora do RAM, prawdopodobnie nikt nie wątpi w przydatność pamięci podręcznej (cache). Wielokrotnie szybszy dostęp do pamięci niesie jednak ze sobą pewne konsekwencje, z którymi nie zawsze jest się łatwo uporać. W tym artykule zajrzymy w ciemniejsze strony połączenia procesorów i pamięci operacyjnej oraz opiszemy wynikające z tego problemy.

## DOSTĘP DO PAMIĘCI

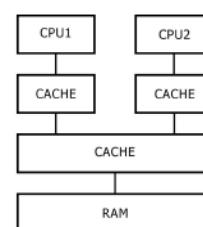
Optymalizacje wprowadzone do większości nowoczesnych procesorów poza sporym przyspieszeniem wykonania programów skutkują często skomplikowaniem samego programu. Wprowadzenie wielordzeniowości procesorów oraz systemu pamięci podręcznych wymaga dodatkowego nakładu pracy, aby zachować podstawowe zasady, którymi często kierują się programiści – wynik działania dwóch identycznych programów powinien być zawsze taki sam, niezależnie od środowiska, na którym są one uruchamiane – w idealnym świecie optymalizacje kompilatora mogą zostać wprowadzone, jedynie jeśli nie zmieniają wyniku programu, a pamięć podręczna nie spowoduje niepoprawnego działania. Aby wyeliminować wpływ agresywnych optymalizacji kodu, często wymagane jest „uszeregowanie” zapisów i odczytów do pamięci – służy do tego tytułowa bariera pamięci (ang. *memory fence*, *memory barrier*).

## INTERFEJS CACHE-CPU

Na Rysunku 1 przedstawiony jest podstawowy schemat poglądowy połączenia współczesnego procesora i pamięci operacyjnej – elementem pośredniczącym zwykle jest jeden lub więcej poziomów cache – przeznaczony dla rdzenia procesora oraz współdzielone przez rdzenie. Jeśli wykonywany kod próbuje wykonać zapis lub odczyt do pamięci, najpierw weryfikuje, czy adres, na którym ma zostać wykonana operacja, znajduje się już w cache – jeśli tak, operacja wykonywana jest początkowo tylko na pamięci podręcznej. Taka budowa skutkuje koniecznością synchronizacji procesorów – jeśli procesor 1 ma zamiar zapisać dane pod adres, który znajduje się w pamięci cache procesora 2, musi mu przekazać o tym informację – inaczej dla programisty ten sam adres pamięci zawierały dwie różne wartości – zależnie od tego, na którym procesorze byłby uruchamiany kod. Do przeprowadzania synchronizacji cache służą protokoły zachowania spójności pamięci podręcznej (ang. *cache coherence protocol*). „Pakietem” protokołu można nazwać linię cache – obszar w pamięci o danym rozmiarze, który jest przesyłany pomiędzy procesorami – możliwe najmniejsza operacja na pamięci podręcznej dotyczyć będzie właśnie linii cache.

Informację uaktualniającą zmienną lub informację o zapisie często nazywa się „inwalidacją” (ang. *invalidate*) adresu w cache [1][4]. Procesor 1 zapisując do pewnego adresu, daje znać procesorowi 2, że adres ten przestaje być aktualny. Do obowiązków procesora 2 po otrzymaniu takiej wiadomości należy wyrzucenie go ze swojej pamięci podręcznej oraz przesłanie do procesora 1 potwierdzenia odebrania wiadomości. Jak można się domyślić, przesyłanie dużej ilości takich informacji pomiędzy procesorami powoduje pewien narzut na wydajność. Ponieważ aktualnie moc obliczeniowa jest

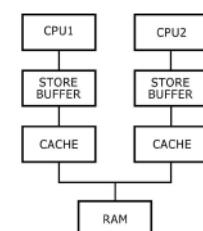
bardzo cennym towarem, inżynierowie zaimplementowali pewne techniki, które pozwalają zoptymalizować cały proces.



Rysunek 1. Połączenie procesora i pamięci poprzez cache

## BUFOR ZAPISÓW

Należy zwrócić uwagę, że każda prośba o inwalidację cache musi zostać potwierdzona przez odbierający procesor. Nietrudno wyobrazić sobie sytuację, kiedy procesor wysyła wiadomość o inwalidacji danego adresu, po czym czeka na odpowiedź, marnując cenne cykle, w których mogłyby zostać wykonana pewna praca. Aby uniknąć przestoju w pracy procesora, wprowadzono bufor zapisów [1] (Rysunek 2), do którego wstępnie dokonywany jest zapis, który „przenika” do cache, dopiero kiedy pozostałe procesory wysiąły potwierdzenie inwalidacji. Dla procesora dokonującego zapisu zgodność danych jest zachowana – każdy odczyt zmiennej wymaga w pierwszej kolejności zbadania, czy istnieje ona w buforze zapisów (ang. *store forwarding*). Jeśli tak – wartość ta jest brana jako aktualna. Jeśli nie, procesor sięga do cache.



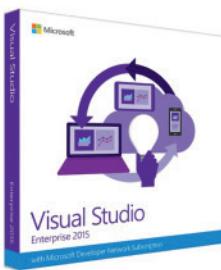
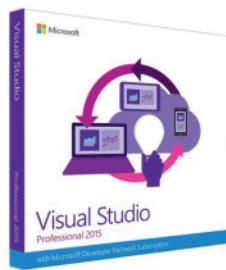
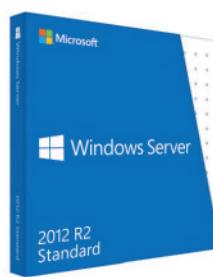
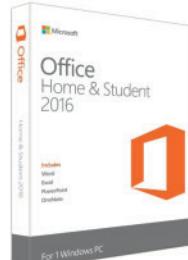
Rysunek 2. Bufor zapisów

## KOLEJKA INVALIDACJI

Przestój związany z oczekiwaniem na potwierdzenie można zmniejszyć jeszcze bardziej dzięki mechanizmowi kolejki inwalidacji [1] (Rysunek 3). Procesor odbierając żądanie inwalidacji cache, zamiast wykonywać operację synchronicznie, zmuszając wysyłający procesor na czekanie (lub opóźniając przeniknięcie danych z bufora zapisów do cache), może natychmiast odesłać potwierdzenie „obiecując”, że wykona uaktualnienie później. W praktyce obietnica dotyczy linii cache – procesor obiecuje, że nie wyśle żad-



**TTS Company rekomenduje oprogramowanie Microsoft ®**



**www.OprogramowanieKomputerowe.pl**

Microsoft Azure

Office 365

OneDrive

Więcej informacji: ☎ (22) 272 94 94 ✉ [sales@tts.com.pl](mailto:sales@tts.com.pl)

Sprzedaż

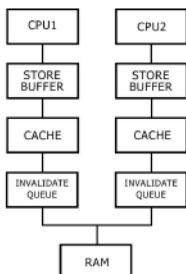


Dystrybucja



Import na zamówienie

nich informacji dotyczących inwalidowanego adresu do innych procesorów, dopóki adres ten nie przestanie być dla niego aktualny (usunięty z pamięci cache).



Rysunek 3. Kolejka inwalidacji

Bezpośrednio z wprowadzonych optymalizacji (kolejki inwalidacji i bufora zapisów) wynikają problemy, dla których rozwiązaniem są bariery pamięci. Należy zdać sobie sprawę z faktu, że procesor nie rozumie zależności pomiędzy zmiennymi, jeżeli nie wynika ona z wykonywanych instrukcji – procesor i kompilator mają prawo zmienić kolejność zapisów i odczytów tak, aby jak najlepiej wykorzystać swoje zasoby. Jako przykład można podać kod w Listingu 1.

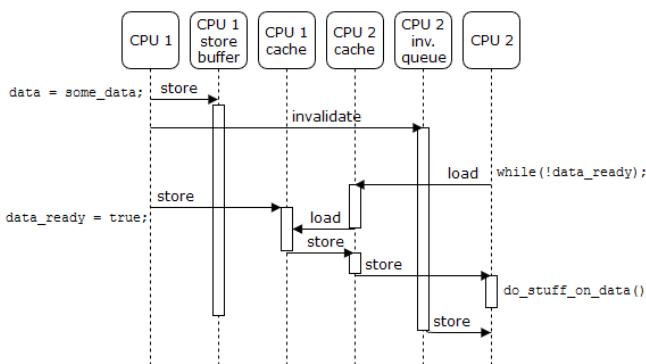
**Listing 1. Przykład kodu uruchamianego na dwóch procesorach**

#### Procesor 1

```
data = some_data;
data_ready = true;
```

#### Procesor 2

```
while(!data_ready)
;
do_stuff_on_data(data);
```



Rysunek 4. Przykładowy przebieg programu

Zakładając, że wstępnie zmienna `data_ready` ma wartość `false`, procesor 2 będzie czekał, aż procesor 1 przypisze dane do zmiennej `data`, a następnie ustawi flagę gotowości `data_ready`. Warunek w kodzie procesora 2 pozwoli przejść do kolejnych instrukcji i wykonać pewne operacje (`do_stuff_on_data()`) na teoretycznie aktualnych danych. W praktyce, jeśli zmienna `data` znajduje się w `cache` procesora 2, natomiast zmienna `data_ready` w `cache` procesora 1, można wyobrazić sobie następujący scenariusz (Rysunek 4): procesor 1 wykonuje przypisanie do zmiennej `data`. Następuje sprawdzenie, czy adres zmiennej `data` znajduje się w `cache` – wynik jest negatywny, procesor 1 umieszcza `data` w swoim buforze zapisów i informuje procesor 2 o konieczności inwalidacji konkretnej linii `cache`. W międzyczasie procesor 2 wykonując pętlę, zauważa, że zmienna `data_ready` nie istnieje w jego `cache` – wysyła więc żądaniny wczytania zmiennej do `cache`. Procesor 1 wykonuje przypisanie do flagi `data_ready` – zmienna ta istnieje w jego `cache`, więc mody-

fikuje jedynie wartość w pamięci podrzędnej. Następnie odbiera żądanie wczytania od procesora 2 zmiennej `data_ready` i przesyła do niego jej nową wartość z własnego `cache` (`true`). Procesor 2 odbiera nową wartość, kończy wykonywanie pętli i przechodzi do operacji na danych, które są nieaktualne. Informacja procesora 1 o konieczności inwalidacji linii `cache` zawierającą zmienną `data` nie dotarła do procesora 2 w odpowiednim czasie. Prosty program, którego wynik wydawał się oczywisty, zadziałał niepoprawnie. Jak wspomniano wcześniej, procesor nie „rozumie” tak jak programista zależności zmiennych `data` i `data_ready`. Istnieje jednak możliwość poinformowania procesora o tym fakcie przez umieszczenie w kodzie bariery pamięci (Listing 2).

**Listing 2. Przykład zastosowania bariery pamięci**

#### Procesor 1

```
data = some_data;
memory_barrier();
data_ready = true;
```

#### Procesor 2

```
while(!data_ready)
;
do_stuff_on_data(data);
```

Dzięki temu kod procesora 1 zostanie uszeregowany – żaden procesor w systemie nie będzie w stanie „zobaczyć” nieaktualnej zmiennej `data` i aktualnej zmiennej `data_ready` – możliwe są tylko przypadki aktualności obu zmiennych lub żadnej z nich. Instrukcja bariery pamięci w tym przypadku spowoduje „oznaczenie” wszystkich zmiennych w buforze zapisów – każdy kolejny zapis zostanie umieszczony w buforze zapisów – nawet jeśli zapisywany adres istnieje w `cache`. Dopiero w momencie, w którym każda zmienna opuści bufor zapisów (przeniknie do `cache`), przywrócone zostanie normalne działanie – zapis będzie mógł być dokonywany bezpośrednio do `cache`, z pominięciem bufora zapisów. Bariera działająca w ten sposób nazywana jest często barierą typu „zapis” (`write, store lub release`) [1] [3].

Jak można się domyślić, jest to jedynie pierwszy pokonany problem w powyższym kodzie. Jeśli zastosować barierę pamięci typu „zapis” tak jak pokazano wcześniej, zdarzyć się może następująca sytuacja: procesor 1 dokonuje zapisu do zmiennej `data`, dodając ją do swojego bufora zapisów i wysyłając żądanie inwalidacji. Procesor 2 wysyła żądanie wczytania zmiennej `data_ready` do `cache`, a następnie odbiera żądanie inwalidacji od procesora 1, dodaje je do swojej kolejki inwalidacji i natychmiast potwierdza odebranie. Procesor 1 odbiera potwierdzenie, przenosi zmienną `data` z kolejki zapisów do swojego `cache` oraz przechodzi do przypisania do zmiennej `data_ready`. Ponieważ adres zmiennej `data` znajduje się już w jego `cache`, podmienia jedynie jego wartość w swojej pamięci podrzędnej. Następnie odbiera wcześniejsze żądanie wczytania linii `cache` zawierającej zmiennej `data_ready` od procesora 2 i odpowiada nową wartością `data_ready`. Procesor 2 kończy wykonanie pętli i wykonuje operacje na nieaktualnych danych – `data` nadal znajduje się w jego `cache`. Następnie przetwarza swoją kolejkę inwalidacji i aktualnia swoją pamięć podrzelną – m.in. zmienią `data`, niestety już po czasie, w którym spójność zmiennej była potrzebna. Aby ostatecznie naprawić kod, należy użyć jeszcze jednej bariery pamięci – Listing 3.

**Listing 3. Poprawiony kod**

#### Procesor 1

```
data = some_data;
memory_barrier();
data_ready = true;
```

#### Procesor 2

```
while(!data_ready)
;
memory_barrier();
do_stuff_on_data(data);
```

Bariera w procesorze drugim działa analogicznie – oznacza wszystkie wpisy, jednak nie w buforze zapisów, a w kolejce invalidacji. Od tego momentu każde wczytanie zmiennej do pamięci będzie musiało poczekać, aż wszystkie oznaczone wpisy w kolejce invalidacji zostaną obsłużone przez procesor i wczytane do cache. Dzięki temu linia cache zawierająca zmienną data zostanie usunięta zanim wykona się operacja na już nieaktualnej wartości. Taką barierę nazywa się barierą typu „odczyt” (*read, load lub acquire*) [1] [3].

## DOUBLE-CHECKED LOCKING

Aby zdemonstrować praktyczne zastosowanie barier pamięci, można przyjrzeć się dość często stosowanemu idiomowi języka C++, m.in. w implementacjach wzorca projektowego „singleton”.

**Listing 4. Singleton**

```
struct Singleton
{
    static Singleton *getInstance()
    {
        if(s == NULL)
            s = new Singleton;
        return s;
    }

    static Singleton *s;
};

Singleton *Singleton::s = NULL;
```

W Listingu 4 przedstawiono przykładowy kod z obiektem tworzonym w sposób leniwy, przy pierwszym wywoaniu metody `getInstance()`, który w C++ jest niepoprawny w środowisku wielowątkowym [2]. Kuszące może być naprawienie kodu przez prostą modyfikację metody `getInstance()`, tak jak pokazano w Listingu 5:

**Listing 5. Próba naprawienia kodu metody getInstance**

```
static Singleton *getInstance()
{
    ScopeLock lock;
    if(s == NULL)
        s = new Singleton;
    return s;
}
```

Warto jednak zauważyć, że mutex (`ScopeLock`) nie jest potrzebny w całym zakresie metody `getInstance()` – nieodzowny jest jedynie w momencie tworzenia instancji klasy `Singleton`. Wniosek ten prowadzi do kodu stosującego tzw. blokadę z podwójnym załatwieniem (*double-checked locking pattern*) – Listing 6.

**Listing 6. Double-checked locking pattern**

```
static Singleton *getInstance()
{
    if(s == NULL)
    {
        ScopeLock lock;
        if(s == NULL)
            s = new Singleton;
    }
    return s;
}
```

Jeśli inny wątek zdążył zainicjalizować instancję klasy `Singleton` po pierwszym sprawdzeniu (i przed zajęciem mutexa przez drugi wątek), drugie sprawdzenie zmiennej `s` zablokuje jej nadpisanie i



A TY CZEGO  
nauczysz się  
W NOWYM ROKU?



Helion

pozwoli uniknąć wycieku pamięci. Problematyczna jednak pozostaje linia, która może wydawać się poprawna na pierwszy rzut oka: `s = new Singleton;` W rzeczywistości jest to wielokroka operacja składająca się z alokacji pamięci, konstrukcji klasy `Singleton` oraz przypisania wskaźnika do przydzielonej wcześniej pamięci. Nic nie stoi na przeszkodzie, aby kompilator wykonał te operacje w innej kolejności – przypisując najpierw wskaźnik do przydzielonej pamięci, a dopiero w następnym kroku konstruując klasę `Singleton`. Nietrudno wyobrazić sobie skutki wykonywania operacji na niezainicjalizowanej pamięci. Można próbować nadal prowadzić nierówną walkę z kompilatorem i procesorem, wykonując operacje na tymczasowej zmiennej – Listing 7.

**Listing 7. Double-checked locking pattern, przykład 2**

```
static Singleton *getInstance()
{
    Singleton *tmp = s;
    if(tmp == NULL)
    {
        ScopeLock lock;
        tmp = s;
        if(tmp == NULL)
        {
            tmp = new Singleton;
            s = tmp;
        }
    }
    return s;
}
```

Niestety powyższy kod również nie jest poprawny – w trakcie wstępnie przypisywania do zmiennej `tmp` zmienność `s` może być nieaktualna (kopia przechowywana w `cache`, inny procesor przesłał informację o inwalidacji), jednak jest ona nieprzetworzona – zakolejkowana w buforze inwalidacji). Drugim miejscem, które może powodować problem, jest przypisanie do zmiennej `tmp` nowo tworzonyj instancji klasy `Singleton`. Jeśli zostanie wykonany zapis nowej wartości do zmiennej `tmp`, która aktualnie nie będzie znajdowała się w `cache`, nowa wartość trafi do bufora zapisów, po czym procesor może wysłać żądanie wczytania zmiennej `tmp` do `cache` i odczytać nieaktualną już zmienność `tmp` – procesor powinien najpierw „zajrzeć” do bufora zapisów lub uszeregować zapis do `tmp` z odczytem `tmp` i zapisem do `s`.

Biorąc pod uwagę wszystkie wspomniane czynniki, powstaje finalny kod, przedstawiony w Listingu 8:

**Listing 8. Poprawna implementacja double-checked locking pattern**

```
static Singleton *getInstance()
{
    Singleton *tmp = s;
    membar_read();
    if(tmp == NULL)
    {
        ScopeLock lock;
        tmp = s;
        if(tmp == NULL)
        {
            tmp = new Singleton;
            membar_write();
            s = tmp;
        }
    }
    return s;
}
```

Starając się zaadresować problematyczną sytuację, komitet języka C++ wprowadził od standardu C++11 istotny paragraf 6.7.4, dotyczący obiektów ze statycznym czasem życia:

*If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.*

Wspomniana modyfikacja standardu pozwala na implementację powyższego skomplikowanego kodu w sposób przedstawiony w Listingu 9 – kompilator sam wygeneruje za programistę blokadę z podwójnym zatwierdzeniem.

**Listing 9. Metoda getInstance w C++11**

```
static Singleton &getInstance()
{
    static Singleton s;
    return s;
}
```

## SILNY I SŁABY MODEL PAMIĘCI

Na koniec warto wspomnieć o tzw. „silnym” i „słabym” modelu pamięci procesora. Model pamięci pozwala na stwierdzenie, jakie możliwości ma procesor, jeśli chodzi o zamiany kolejności operacji na pamięci. Procesory Intel x86 plasują się w kategoriach silniejszego modelu pamięci – pewne scenariusze zamiany zapisów i odczytów na tym procesorze nigdy nie nastąpią. Z drugiej strony procesor ARM pojawię się po drugiej stronie skali – część operacji może nie zostać wykonana w kolejności, której spodziewa się programista. Legendarny już, pod względem słabego modelu pamięci, jest powoli odchodzący w zapomnienie DEC Alpha [3][4].

Jak można się domyślić, w powyższym tekście zastosowano dużo uproszczeń – w rzeczywistości współczesne protokoły zachowania spójności danych są bardzo skomplikowanymi maszynami stanów, a nowoczesne kompilatory i procesory stosują dużo dodatkowych metod przyspieszających odczyty i zapisy programu. Na szczęście dla programistów problemy z nich wynikające są zwykle spotykane jedynie w specyficznych przypadkach, na które niełatwo się natknąć – „w programowaniu najtrudniejsze jest nazywanie zmiennych i inwalidacja cache”.

## Źródła:

- [1] *Memory barriers: a hardware view for software hackers*  
– Paul E. McKenney
- [2] *C++ and the perils of double-checked locking*  
– Scott Meyers, Andrei Alexandrescu
- [3] <http://preshing.com> – Preshing on Programming
- [4] *Linux kernel memory barriers*  
– Howells D., McKenney P. E., Deacon W., Zilstra P.



**TOMASZ WIDENKA**

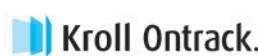
widenka.tomasz@hotmail.com

Programista systemów wbudowanych.  
W wolnych chwilach gra na gitarze i ćwiczy na siłowni.



Największy wybór profesjonalnego oprogramowania w Polsce !

... w ofercie programy ponad 500 producentów ...



Więcej informacji:

📞 (22) 868 40 42



[sales@tts.com.pl](mailto:sales@tts.com.pl)

Sprzedaż



Dystrybucja



Import na zamówienie

TTS Company Sp. z o.o.

ul. Domaniewska 44A

02-672 Warszawa

[www.tts.com.pl](http://www.tts.com.pl)

# Komponenty – podstawowa składowa współczesnych UI

Komponenty są tym, co łączy popularne obecnie biblioteki i frameworki służące do budowania interfejsu użytkownika. Nie wszyscy znają jednak ich cechy i zależności, które wnoszą do życia twórców aplikacji webowych. Aby poznać odpowiedzi na tego typu pytania, przyjrzyjmy się frameworkowi Angular, którego rozwój daje nam pogląd na to, w którą stronę zmierza współczesny front-end.

## ZACZĘŁO SIĘ NIEPOZORNIĘ

Jeszcze kilka lat temu światem web developmentu rządziła prosta, niepodważalna reguła – część kliencka aplikacji webowej powinna być tak prosta jak to możliwe, a każda bardziej skomplikowana operacja powinna być delegowana do serwera. Taki stan rzeczy był spowodowany m.in. ówczesnymi ograniczeniami sprzętowymi po stronie klienta oraz ubogimi możliwościami przeglądarek internetowych tamtych czasów.

Kiedy w 2009 r. jeden z inżynierów firmy Google, niejaki Misko Hevery, prezentował światu roboczą wersję narzędzia przedstawianego wtedy jako `<angular/>`, świat odpowiedział umiarkowanym entuzjazmem. Słuchacze jego prezentacji martwili się raczej o kompatybilność rozwiązania z istniejącymi na rynku przeglądarkami, a nie o możliwość budowania zaawansowanych interfejsów z ładowanymi asynchronicznie komponentami. Czas jednak mijał, a obietnica łatwego budowania dynamicznych aplikacji webowych zyskiwała kolejnych zwolenników.

Czym w tamtym okresie wyróżniał się Angular? Kiedy większość dostępnych na rynku bibliotek skupiała się na wyciskaniu maksimum z imperatywnych manipulacji przeglądarkowym DOMem (jQuery), albo ukrywania znanego dla programistów środowiska opartego o HTML (ExtJS), Angular starał się po prostu... rozszerzać możliwości przeglądarki. „Tak wyglądały HTML, gdyby został zaprojektowany do tworzenia aplikacji webowych” – brzmiało jedno z pierwszych hasł mających przekonać programistów do Angulara.

**Listing 1. Najprostszy sposób na zaprezentowanie możliwości Angulara**

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
  <meta charset="utf-8">
  <title>Angular.js experiments</title>
  <script src="https://code.angularjs.org/1.0.0/angular-1.0.0.js"></script>
</head>
<body>
  <input type="text" name="name" ng-model="name"/>
  <p>{{ name }}</p>
</body>
</html>
```

Programista chcący stworzyć dynamiczną aplikację nie potrzebował zbyt wiele. W powyższym fragmencie kodu widać, że poza dwoma atrybutami z prefiksem `ng-` oraz wąsatymi nawiasami w tagu p reszta to najwyklejszy HTML. W ten sposób można podsumować całą filozofię pierwszych wersji Angulara – pozostać tak blisko znanych sposobów na pisanie strony WWW jak to możliwe,

przy jednoczesnym dostarczeniu szerokiej palety gotowych do wykorzystania funkcjonalności.

## JAK BUDOWAĆ INTERFEJS UŻYTKOWNIKA

Interfejs użytkownika oparty o losowo porozrzucane po stronie atrybuty, takie jak w Listingu 1, nie wytrzyma próby czasu – kod z bieżącym czasu stanie się po prostu niemożliwy do utrzymania. Ale czy potrzebujemy czegoś więcej? Zdecydowanie tak – budowanie interfejsu użytkownika nie różni się zbyt wiele od tworzenia kodu w innej warstwie aplikacji. Tak samo jak na back-endzie, front-end powinien wykorzystywać moduły o małej, ścisłe określonej odpowiedzialności, które dodatkowo mogłyby być reużywalne i możliwe do wymiany bez wpływu na cały projekt. Co w tej kwestii miało do zaoferowania Angular?

Przez kilka pierwszych wydań tego frameworka (Angular.JS, czyli 1.x) sztandarowym podejściem do budowania aplikacji było jej pisanie zgodnie z czymś, co przypominało wzorzec MVC. Model mógł być dowolny obiekt zawierający dane, natomiast na styku „V” oraz „C” rządził kontroler sterujący widokiem.

**Listing 2. Deklaracja nowego kontrolera w aplikacji wykorzystującej Angular.JS**

```
angular.controller('ListController', function($scope) {
  $scope.items = [
    { name: 'Thomas' },
    { name: 'Dieter' },
    { name: 'Rolf' }
  ];
});
```

**Listing 3. Wykorzystanie stworzonego kontrolera w widoku**

```
<body ng-controller="ListController">
  <ul>
    <li ng-repeat="item in items">{{item.name}}</li>
  </ul>
</body>
```

Kontrolery enkapsulowały funkcjonalność danego fragmentu strony, a tym, co łączyło świat JavaScriptu z widokiem w HTMLu, był obiekt `$scope`, który spełniał rolę czegoś na wzór *view modelu*. Był to model zawierający wszystkie niezbędne dane wymagane do wyrenderowania określonego fragmentu naszej aplikacji.

Kod z Listingów 2 i 3 nie wydaje się być czymś szczególnie skomplikowanym – tworzymy kontroler, do którego wstrzykujemy

obiekt `$scope`, ten z kolei wypełniamy danymi, a danych finalnie używamy w widoku. Przynajmniej tak mówiła teoria. W praktyce nie wyglądało to tak kolorowo – głównym problemem takiego podejścia był fakt, że często w miarę wzrostu rozmiaru projektu kontrolery rozrastały się do rozmiarów, których utrzymanie było bardzo trudne.

Co prawda dokumentacja na stronie Angulara niemal od samego początku mówiła, żeby logikę biznesową naszej aplikacji delegować do zewnętrznych modułów i serwisów, jednak nic nie stało na przeszkodzie, żeby robić „po swojemu”. Kontrolery miały jeszcze jedną wadę, którą był (aż do wprowadzenia wersji 1.5) brak jasno zdefiniowanego cyklu ich życia. W trakcie swojej pracy w projektach opierających się o Angular.JS widziałem kilkanaście różnych implementacji funkcji `init`, `initialize` czy jakichkolwiek innych, które miały być miejscem, w którym zainicjalizujemy nasz model czy pobierzemy niezbędne dane z zewnętrznych serwisów. Swoboda, którą Angular zostawił programistom, okazywała się po czasie przynosić więcej szkód niż korzyści. Koniec końców na kontrolerach świat się jednak nie kończył.

Angular.JS wprowadził jeszcze jeden sposób na tworzenie dynamicznych interfejsów użytkownika – tym sposobem są dyrektywy. W dokumentacji czytamy, że „dyrektywy są sposobem na nauzczenie HTMLa nowych tricków”, ale dla nas to po prostu pewnego rodzaju znaczniki, dzięki którym najpierw w kodzie HTML możemy oznaczyć element, którego wygląd lub zachowanie chcemy modyfikować, a potem w kodzie JS określić, czym ta nowa funkcjonalność ma być.

Dzięki dyrektywom możemy np. wchodzić w bezpośrednią interakcję z DOMem, modyfikować i rozszerzać działanie istniejących elementów na stronie, a także... budować reużywalne komponenty. W kontekście dzisiejszego tematu to właśnie ta ostatnia możliwość interesuje nas szczegółowo. Jak to wygląda w praktyce?

#### **Listing 4. Deklaracja dyrektywy pełniącej rolę nowego elementu interfejsu użytkownika**

```
angular.directive('listItem', function () {
  return {
    restrict: 'E',
    scope: {
      item: '='
    },
    template: '<li>{{item.name}}</li>'
  };
});
```

#### **Listing 5. Użycie dyrektywy z poziomu kodu HTML**

```
<ul>
  <list-item ng-repeat="item in items" item="item"></list-item>
</ul>
```

W kodzie z Listingu 4 widzimy przykład tzw. *directive definition object*, czyli obiektu konfiguracyjnego dla nowej dyrektywy. Pole `restrict` dotyczy sposobu deklaracji dyrektywy (w naszym przypadku E, czyli element), `scope` dotyczy parametrów, z jakimi możemy ją zainicjalizować, a `template` to szablon (widok). W kodzie z Listingu 5 widzimy z kolei, jak posługiwać się dyrektywą z poziomu HTMLa.

Dzięki dyrektywom zyskujemy praktycznie nieograniczone możliwości związane z tworzeniem elementów nieistniejących w standardzie HTML, co koniec końców przekłada się na tworzenie bardziej deklaratywnego i wysokopoziomowego kodu interfejsu użytkownika (`<avatar />` zamiast `<img />`, `<dashboard />` zamiast `<div />` itd.). Czy tak użyteczne rozwiązanie jest więc tym, na czym Angular postanowił się zatrzymać?

## **NOWE CZASY, NOWE ROZWIĄZANIA**

Dyrektywy, pomimo bycia jedną z najważniejszych funkcjonalności frameworka Angular.JS, nie są rozwiązaniem bez wad. Krytykuje się je m.in. za skomplikowaną konfigurację oraz stosunkowo wysoki próg wejścia dla osób poczynających. W zasadzie nie ma się czemu dziwić – istnieją na przykład aż cztery możliwe sposoby na zadeklarowanie dyrektywy po stronie kodu HTML (jako tag, klasa, atrybut i... komentarz) – które rozwiązanie jest w takim razie tym właściwym? A może wszystkie są tak samo dobre? No i co z kontrolerami, o których pisałem kilka akapitów wcześniej?

Zespół odpowiedzialny za Angulara 2.0 (teraz nazywanego po prostu Angularem) był świadomy problemów, które wymieniłem, dlatego w 2014 r. zdecydował się na dość radykalny krok – zrywamy ze starymi przyzwyczajeniami na rzecz podejścia *component-first*. Rezygnujemy z kontrolerów, rezygnujemy z obiektu `$scope`, rezygnujemy ze skomplikowanych obiektów konfiguracyjnych (DDO). W nowym podejściu stawiamy przede wszystkim na komponenty. Ale czym te komponenty właściwie są?

Komponent to podstawowa składowa interfejsu użytkownika. W Angularze reprezentowana jest ona przez klasę posiadającą kilka charakterystycznych elementów.

- » Po pierwsze – stan. Stan komponentu w najprostszej formie przechowujemy tak jak stan instancji obiektu danej klasy, czyli po prostu we właściwościach (ang. *properties*) klasy.
- » Po drugie – cykl życia. Funkcjonalność komponentu w Angularze opiera się na konkretne zdefiniowanych *hookach*, czyli funkcjach, które Angular wywołuje na konkretnym etapie życia komponentu (np. `ngOnInit()`, `ngOnChanges()` etc.).
- » Po trzecie – kompozycja. Interfejs użytkownika złożony jest z wielu komponentów, które mogą się ze sobą komunikować. Wejściem dla komponentu są pola oznaczone dekoratorem `@Input`, natomiast sam komponent może powiadomić jego otoczenie o zdarzeniach wewnętrznych poprzez tzw. *EventEmittery* – mechanizm opierający się o eventy.
- » Po czwarte – dekorator `@Component`. Zawiera on metadane konkretnego komponentu, takie jak selektor czy zestaw stylów.

#### **Listing 6. Angular – deklaracja nowego komponentu**

```
@Component({
  selector: 'app-image-preview',
  templateUrl: './imagePreview.component.html',
  styleUrls: ['./imagePreview.component.scss']
})
export class ImagePreviewComponent {
  isPreviewVisible: boolean = false;

  @Input()
  imageUrl: string;

  @Output()
  close: EventEmitter<any> = new EventEmitter();
}
```

Komponenty w nowym Angularze nie przypominają tego, do czego przyzwyczajono programistów posługujących się na co dzień dyrektywami. Zmiany można zauważać już na samym poziomie języka (TypeScript) – widać, jak dużą rolę odgrywają teraz klasy, dekoratory czy typy. O ile w kwestii migracji starszych projektów nie obejdzie się bez dokumentacji, tak z punktu widzenia nowych aplikacji Angular zrobił ogromny krok w przyszłość.

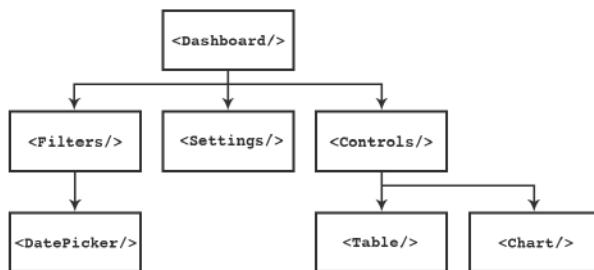
Jeśli chodzi o widok aplikacji, to najbardziej istotną nowością jest teraz rola komponentu – o ile poprzednio aplikacja mogła opierać się o klasyczne widoki uzupełniane dyrektywami, tak teraz

rozwiązań, które budujemy, składamy w całości z komponentów. Wiąże się to z faktem, że Angular obrał kierunek platformy, na której opiera się cała aplikacja, a nie tylko narzędzia, które ma sprawić, że statyczna strona zyska dynamiczne elementy (choć trzeba przyznać, że trwają pracę nad powrotem do korzeni – zainteresowanych odsyłam do tematu *Angular Elements*).

**Listing 7. Angular – kompozycja komponentów na stronie**

```
<app-container>
  <app-image-preview (close)="onPreviewClose()"></app-image-preview>
</app-container>
```

Zapoznaliśmy się z pojedynczym komponentem, ale skoro komponent to pewna atomowa część interfejsu użytkownika, to jasne jest, że na jednym komponencie nie może się skończyć. Wzbogacając naszą aplikację o nowe funkcjonalności, będziemy ją też rozbudowywać o nowe komponenty, a w rezultacie takiego procesu będziemy mieć przed sobą tzw. drzewo komponentów. Drzewo komponentów to nic innego jak struktura, która odzworowuje układ komponentów naszej aplikacji. Może to wyglądać np. tak:

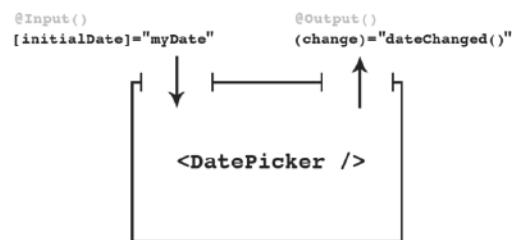


Rysunek 1. Drzewo komponentów przykładowej aplikacji Dashboard

Jak na dloni widać tutaj modularny charakter całego projektu. Przy odpowiednio wydzielonej odpowiedzialności poszczególnych komponentów ich wymiana nie powinna w żadnym stopniu wpływać na inne obszary aplikacji – na przykład zmiana komponentów odpowiedzialnych za prezentowanie danych nie powinna wpływać na mechanizm filtrowania. W wielu projektach możliwość wymiany komponentów nie ogranicza się jedynie do funkcjonalności, ale i do technologii, dzięki której realizujemy daną część aplikacji.

Jeśli jesteśmy już przy schemacie, który pokazuje układ komponentów naszej aplikacji, to warto powiedzieć jeszcze co nieco o typach komponentów i przepływie danych pomiędzy nimi. Założymy, że Dashboard jest komponentem, w którym na początku jego cyklu życia pobierane są pewne dane inicjalizacyjne dla aplikacji. Z powodu jego odpowiedzialności za pobranie danych i wykonanie logiki innej– tej związanej ściśle z interfejsem użytkownika – możemy go zaliczyć do grupy *smart components*. Jego przeciwieństwem może być komponent Table, którego jedynym zadaniem jest stworzenie tabeli na podstawie przekazanych do niego danych. Nie ma zależności na inne części aplikacji, nie pobiera dodatkowych danych i nie przechowuje żadnego stanu – jego określimy mianem *dumb* lub *presentational*.

Zagadnienie przepływu danych pomiędzy komponentami łatwo jest sobie wyobrazić po zaprezentowaniu całego procesu podobnie jak poprzednio – w formie graficznej (Rysunek 2). W Angularze przepływ danych pomiędzy komponentami jest *jednokierunkowy*. Patrząc na Rysunki 1 i 2, moglibyśmy powiedzieć, że propagacja danych odbywa się z góry na dół, natomiast propagacja eventów odbywa się z dołu do góry. Poprzez pola oznaczone



Rysunek 2. Przepływ danych i eventów w komponencie DatePicker

pokazanym wcześniej dekoratorem *@Input* deklarujemy swego rodzaju API, przez które będziemy przekazywać dane do środka komponentu, natomiast *@Output* używany jest do deklarowania eventów rozgłaszańskich na zewnątrz. W odróżnieniu od promowanego przez framework AngularJS podejścia *two-way binding* całą ścieżkę przepływu danych możemy teraz rozbić na dwa etapy (wejście i wyjście), przez co cały proces staje się łatwiejszy do zrozumienia i wykorzystania w praktyce.

## CO NA TO ŚWIAT?

Praktyka pokazuje, że podejście *component-first* przynosi zdecydowanie więcej korzyści w postaci lepszej architektury aplikacji oraz bardziej deklaratywnego, reużywalnego i łatwiejszego do utrzymania kodu, w porównaniu do dynamicznych widoków opartych o spłatane ze sobą kontrolery i manipulację DOMem. Korzyści, o których piszę, są na tyle znaczące, że w obecnych czasach komponenty znajdziemy w większości popularnych bibliotek i frameworków front-endowych.

Na ich podstawie zbudowana została filozofia Reacta, który od początku stawał na taki sposób budowania interfejsu użytkownika, ma je AngularJS (w wersji 1.5+), w którym zostały wprowadzone, aby ułatwić migrację do nowszych wersji Angulara, mają je też Ember, Vue czy rozwiązania zyskujące na popularności jak Svelte czy Stencil. Co najciekawsze, podejście oparte o komponenty zostało przyjęte na tyle dobrze, że obecnie trwają intensywne prace nad ich natywną, ustandaryzowaną wersją w postaci Web Components (<https://www.webcomponents.org>).

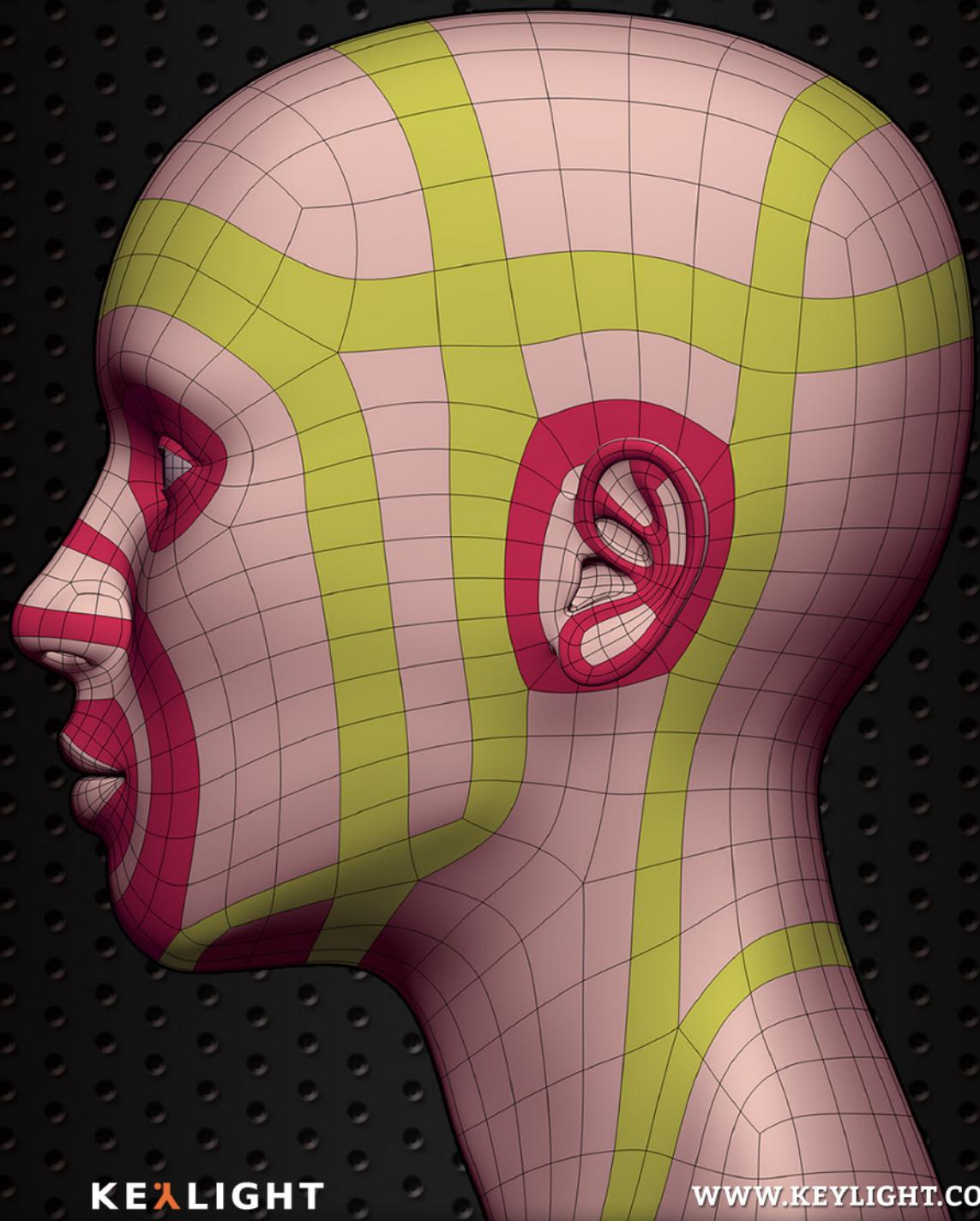
Dla obiektywnego przedstawienia tematu muszę na koniec zaznaczyć, że największą rolę w tworzeniu front-endu nadal odgrywa... czynnik ludzki. Wiąże się to z faktem, że nawet najmodniejszy framework oparty o komponenty nie sprawi automatycznie, że wszystkie dotychczasowe problemy w projekcie zostaną rozwiązane. W podejściu, które dzisiaj opisałem, istnieje ryzyko przeniesienia zbyt dużej części logiki biznesowej do warstwy widoku czy powstania skomplikowanego modelu zarządzania stanem aplikacji. Właśnie dlatego najlepszym sposobem na swobodną pracę z komponentami jest doświadczenie i dostosowanie tzw. „dobrych praktycznych” do swojego sposobu pracy – wszystko po to, żeby wspomniane przeze mnie korzyści nie były jedynie teorią.



**PRZEMYSŁAW SMYRDEK**

[przemek.smyrdek@gmail.com](mailto:przemek.smyrdek@gmail.com)

Kiedyś grafik komputerowy i full-stack developer, dzisiaj pasjonat front-endu oraz wszystkiego, co związane jest z szeroko rozumianym designem. Współtwórca bloga [PoznajProgramowanie.pl](http://PoznajProgramowanie.pl), dzielący się swoim doświadczeniem na rozmaitych konferencjach dla programistów. Przewrażliwiony na punkcie bzdur o JavaScriptie i skomplikowanych interfejsów użytkownika.

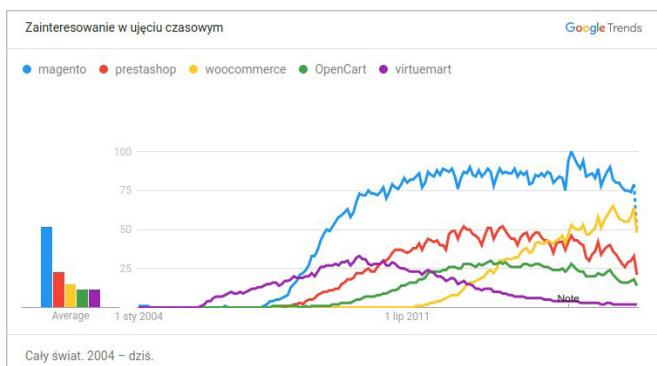


# Poznaj Magento 2

Systemy e-commerce na świecie dzielą się na dwie grupy: proste oraz te znaczająco rozbudowane. Do drugiej grupy należy właśnie Magento. Wiele dużych e-commerce opiera swój biznes właśnie na Magento, m.in. takie marki jak: Coca-Cola, Ford, Castorama, Reserved i wiele, wiele innych. Magento posiada dwie wersje: Magento Open Source (dawniej Magento Community Edition) oraz Magento Commerce. W naszym artykule skupimy się właśnie na tej pierwszej. Chcemy zapoznać czytelnika z tym systemem i pokazać, że warto opierać na nim swoje e-biznesy.

**M**agento – Open Software License – e-commerce System. Pierwsza wersja oprogramowania została oficjalnie wydana 31 marca 2008 r. Niedawno Magento doczekało się długo wyczekiwanej aktualizacji do wersji 2.0. Było to na tyle duże i spektakularne wydarzenie, że zmieniło system o 180 stopni. Przebudowano całkowicie strukturę kodu. Intuicyjny układ panelu administracyjnego oraz zupełnie nowa szata graficzna pozwolą na szybsze zarządzanie e-commerce'm. System zbudowany jest w oparciu o podstawowe zasady architektury aplikacji zorientowanej obiektowo, opartej o Zend Framework 2.

Magento jest bardzo popularną i zaawansowaną platformą do prowadzenia sklepów internetowych. Oferuje swoim użytkownikom pełną elastyczność, kontrolę nad wyglądem, zawartością i funkcjonalnością sklepu. System do dzisiaj kontynuuje dominację na rynku open source e-commerce i nic nie zwiastuje, że sytuacja może ulec zmianie w najbliższych latach. Potwierdza to „Google Trends”, na poniższym wykresie zobrazowano popularność Magento na tle innych open source najpopularniejszych platform sklepowych.



Rysunek 1. Raport Google Trends obrazujący popularność Magento (źródło: [goo.gl/ZVMzRk](http://goo.gl/ZVMzRk)).

Raport przygotowany przez Internet Retailer wskazuje, iż na 1000 spośród największych sklepów na świecie 202 wykorzystuje właśnie platformę Magento.

## TECHNOLOGIE UŻYTE W MAGENTO

Magento 2 to oprogramowanie oparte o język PHP i relacyjne bazy danych MySQL, które z powodzeniem wykorzystuje rozwiązania obecne w dwóch najpopularniejszych frameworkach PHP (Symfony, Zend). Korzysta z najnowszych rozwiązań i technologii dostępnych na rynku. Od strony frontendu oferuje wsparcie dla HTML5,

CSS3 i AMD. Wspiera narzędzia automatyzujące pracę takie jak Grunt/Gulp. Cały system napisany jest według ogólnie przyjętych standardów kodowania PSR. Szybkość sklepu zapewnia wbudowana obsługa rozwiązań do obsługi cache. Do instalacji modułów czy aktualizacji sklepu można używać Composera, a poprawność działania aplikacji sprawdzić, przeprowadzając rozmaite testy w oparciu o framework PHPUnit.

Rozwiązania technologiczne użyte w Magento 2	
Serwer WWW	Apache 2.2/2.4 Nginx 1.7 lub wyższe
Język	PHP 7.x ( z wyłączeniem 7.0.0, 7.0.1, 7.0.3, 7.0.5)
Bazy danych	MySQL 5.6, 5.7 (oraz inne kompatybilne)
Frameworki PHP	Zend Framework 1, Zend Framework 2, Symfony
Cache	Redis 2.x /3.x Varnish 3.x / 4.x Memcached 1.4.x
Backend	PHPUnit Composer Twig PSR-0 do PSR-4 XML
Frontend	HTML5 CSS3 LESS Jquery RequireJs Knockout Grunt Gulp

Tabela 1. Rozwiązania technologiczne użyte w Magento 2

## PLUSY I MINUSY MAGNETO 2

Tak jak każdy system, Magento ma swoich zwolenników i przeciwników.

Magento nie cieszy się dobrą opinią wśród małych biznesów, których przeraża zasobowość systemu ze względu na masę konfiguracji oraz złożoność kodu aplikacji. Z drugiej strony cenią go średnie i większe biznesy, które oczekują stabilności oraz skalowania systemu obsługującego duży ruch i sporą ilość zamówień.

Zalety platformy Magento	Wady platformy Magento
<b>oprogramowanie</b> – jest na otwartej licencji i jest darmowe	<b>wymagania sprzętowe</b> – wymaga zdecydowanie większej mocy obliczeniowej serwerów
<b>konfiguracja</b> – czyste Magento posiada w sobie bardzo dużo opcji konfiguracyjnych i dostosowujących system	<b>brakuje programistów</b> – na rynku brakuje doświadczonych specjalistów Magento
<b>społeczność</b> – spora rzesza pasjonatów rozwijających platformę	<b>koszty wdrożenia</b> – samo oprogramowanie jest darmowe, należy jednak ponieść koszty dostosowania systemu do własnych potrzeb
<b>elastyczność systemu</b> – system doskonale sprawdzi się jako średni i duży e-commerce	<b>aktualizacje systemu</b> – głównie poprawki bezpieczeństwa oraz poprawki funkcjonalności ostatnimi czasami pojawią się dość często w stosunku do Magento 1.x
<b>skalowalność</b> – idealnie sprawdza się w skalowaniu poziomym	
<b>market z wtyczkami</b> – oficjalny market posiadający dodatki oraz rozszerzenia istniejących modułów, zarówno płatnych, jak i darmowych	
<b>multistore</b> – możliwość tworzenia wielu sklepów z poziomu jednego panelu administracyjnego	
<b>SEO</b> – jeden z lepszych systemów e-commerce pod względem optymalizacji pod wyszukiwarki	

Tabela 2. Wady i zalety Magento

## JAK NAPISAĆ PROSTY MODUŁ W MAGENTO 2

Przed rozpoczęciem pracy nad nowym modułem należy przełączyć Magento w tryb developer, wpisując w konsoli polecenie:

### Listing 1. Przełączanie Magento w tryb developerski

```
php bin/magento deploy:mode:set developer
```

Pisanie modułu rozpoczynamy od stworzenia następujących folderów:

- » app/code/Programista
- » app/code/Programista/SampleModule

W katalogu *SampleModule* tworzymy folder *etc.*, w którym będzie znajdowała się konfiguracja naszego modułu:

### Listing 2. Główna konfiguracja modułu Magento /app/code/Programista/SampleModule/etc/module.xml

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Module/etc/module.xsd">
<module name="Programista_SampleModule" setup_version="1.0.0">
<sequence></sequence>
</module>
</config>
```

- » atrybut *name* składa się z nazwy Vendra i nazwy modułu,
- » atrybut *setup\_version* określa wersję modułu,
- » w sekcji *sequence* możemy zamieścić listę modułów, z których mamy zamiar korzystać i które będą załadowane przed naszym modułem.

Aby zarejestrować nasz moduł, należy utworzyć plik *registration.php*.

### Listing 3. Plik rejestracyjny modułu /app/code/Programista/SampleModule/registration.php

```
<?php
\Magento\Framework\Component\ComponentRegistrar::register(
    \Magento\Framework\Component\ComponentRegistrar::MODULE,
    'Programista_SampleModule',
    __DIR__
);
```

Następnie definiujemy router:

### Listing 4. Router modułu Magento /app/code/Programista/SampleModule/etc/frontend/routes.xml

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:App/etc/routes.xsd">
<router id="standard">
<route id="samplemodule" frontName="samplemodule">
<module name="Programista_SampleModule" />
</route>
</router>
</config>
```

Określamy w nim, jakiego modułu dotyczy routing (sekcja *module*), oraz definiujemy *frontName*, czyli nazwę będącą jednocześnie pierwszą częścią adresu URL.

Po stworzeniu routingu przystępujemy do napisania kontrolera. Poniższy kod umieszczamy w pliku:

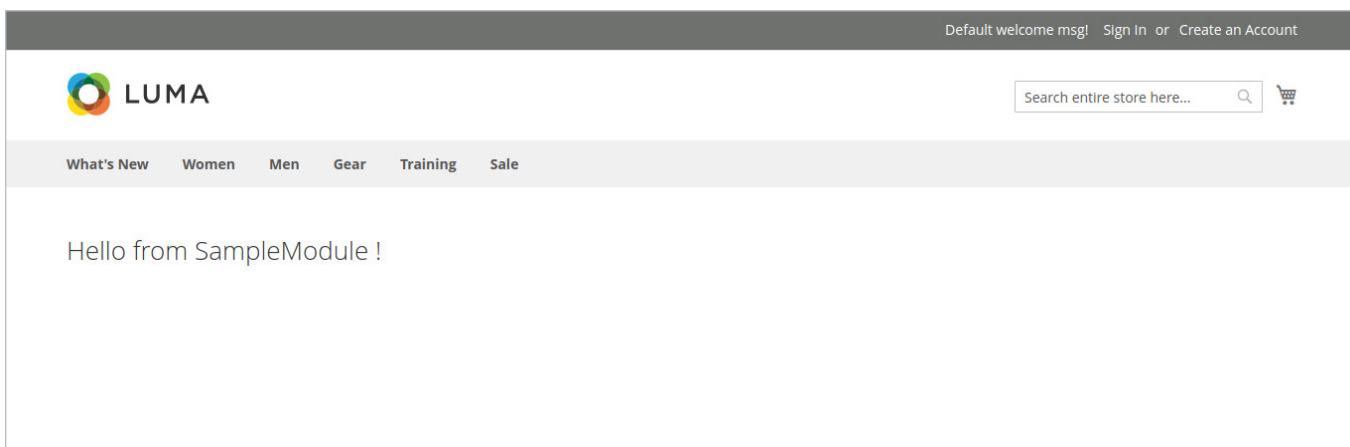
### Listing 5. Kontroler modułu Magento /app/code/Programista/SampleModule/Controller/Index/Index.php

```
<?php
namespace Programista\SampleModule\Controller\Index;
use Magento\Framework\App\Action\Context;
class Index extends \Magento\Framework\App\Action\Action
{
    protected $_resultPageFactory;
    public function __construct(Context $context, \Magento\Framework\View\Result\PageFactory $resultPageFactory)
    {
        $this->_resultPageFactory = $resultPageFactory;
        parent::__construct($context);
    }
    public function execute()
    {
        $resultPage = $this->_resultPageFactory->create();
        return $resultPage;
    }
}
```

Aby wyświetlić przykładowy tekst na stronie, tworzymy blok zawierający metodę *getHelloText*:

### Listing 6. Blok modułu Magento /app/code/Programista/SampleModule/Block/SampleModule.php

```
<?php
namespace Programista\SampleModule\Block;
class SampleModule extends \Magento\Framework\View\Element\Template
{
    public function getHelloText()
    {
        return 'Hello from SampleModule !';
    }
}
```



Rysunek 2. Hello w module SampleModule

Blok umieszczamy w pliku z layoutem, w którym rozszerzamy jeden z kontenerów dostępnych na stronie sklepu. W tym celu w folderze z modułem tworzymy subfolder view. Folder ten może zawierać 3 katalogi w zależności od tego, dla której części Magento 2 będziemy tworzyć layout:

- » frontend – pliki używane w sekcji frontend,
- » adminhtml – pliki dla panelu administracyjnego,
- » base – pliki wspólne dla obu sekcji.

Ponieważ nasz moduł będzie wyświetlał przykładowy tekst od strony użytkownika, tworzymy folder *frontend*, a w nim plik *samplemodule\_index\_index.xml*:

**Listing 7. Layout modułu Magento** /app/code/Programista/Sample-Module/view/frontend/layout/samplemodule\_index\_index.xml

```
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi: noNamespaceSchemaLocation="../../../../lib/internal/Magento/Framework/View/Layout/etc/page_configuration.xsd" layout="1column">
<body>
<referenceContainer name="content">
<block class="Programista\SampleModule\Block\SampleModule" name="samplemodule" template="samplemodule.phtml" />
</referenceContainer>
</body>
</page>
```

W pliku tym za pomocą komendy *referenceContainer* rozszerzamy podstawowy kontener *content*. Umieszczamy w nim blok, który utworzyliśmy wcześniej. W atrybucie *class* podajemy ścieżkę do klasy naszego bloku, natomiast w atrybucie *template* odwołanie do szablonu o nazwie *samplemodule.phtml*. Plik z templatką umieszczamy w folderze *frontend* i subfolderze *templates*.

**Listing 8. Szablon modułu Magento** /app/code/Programista/Sample-Module/view/frontend/templates/samplemodule.phtml

```
<h2>
<?php echo $this->getHelloText(); ?>
</h2>
```

Na koniec pozostało nam już tylko włączyć stworzony moduł. W tym celu z poziomu konsoli wykonujemy polecenie:

### **Listing 9. Aktywacja modułu Magento**

```
php bin/magento module:enable Programista_SampleModule
```

oraz:

### **Listing 10. Upgrade modułów Magento**

```
php bin/magento setup:upgrade
```

Po wpisaniu na stronie sklepu adresu */samplemodule/index/index* powinniśmy zobaczyć obraz jak na Rysunku 2.

## PODSUMOWANIE

W ramach tego artykułu przedstawiliśmy krótki opis całego systemu oraz pokazaliśmy, jak stworzyć pierwszy moduł „Hello world”. Mamy nadzieję, iż będzie to początek waszej przygody z najpopularniejszą platformą sprzedawczą. Jednocześnie warto zaznaczyć, że Magento wymaga dużo czasu i aspiracji. Trzeba go poświęcić znacznie więcej niż na przeczytanie tego artykułu.

### W sieci:

- ▶ The Official Magento User Guide: <https://magento.com/resources/technical>
- ▶ Forum Magento: <https://community.magento.com/>



**MATEUSZ BOROWIK**

[m.borowik@auroracreation.com](mailto:m.borowik@auroracreation.com)

Certyfikowany developer Magento, na co dzień pracujący i rozwijający Software House Aurora Creation. Od 10 lat związany z programowaniem, które z biegiem czasu zostało zamienione na zarządzanie zespołem programistów.



**MAREK MILEWSKI**

Magento developer w Aurora Creation. Programista z zawodu i zamiłowania, od 12 lat projektuje i rozwija aplikacje w języku PHP.

Specjaliści od  Magento®

[auroracreation.pl/kariera](http://auroracreation.pl/kariera)

# Testowanie aplikacji rozproszonych

Gdyby porównać ze sobą tempo rozwoju różnych dyscyplin naukowych, informatyka prawdopodobnie znalazłaby się w ścisłej czołówce, jeżeli nie na pierwszym miejscu. Prędkość zmian, którą można zaobserwować w świecie IT, jest fascynująca i niepokojąca zarazem. Niemal każdy aspekt ulega ciągłym zmianom. Testowanie oraz rodzaje środowisk, w których uruchamiane są aplikacje, również. Tematy testowania i aplikacji rozproszonych z osobna są ostatnio popularne, jednak w połączeniu tworzą swoistą niszę.

**W**zrost popularności i wagi testowania oprogramowania spowodowany był i jest przez kilka czynników. Należą do nich m.in. większy nacisk na jakość oprogramowania oraz rewolucja w metodach prowadzenia projektów. Niegdyś w tzw. waterfallu testowanie oprogramowania było ostatnim etapem jego wytworzenia – tuż przed przejściem w tryb utrzymania [5]. Ponieważ zaplanowanie pracy na długi okres jest bardzo ciężkie, w waterfallu często pojawiały się opóźnienia. Etap testowania był z tego względu niejednokrotnie najbardziej poszkodowany. W popularniejszej obecnie metodzie – Scrum – problem ten nie powinien występować, ponieważ testowanie traktuje się jako codzienne obowiązek i wykonywane jest podczas wytwarzania kodu [6]. Częstsze cykle testowania stworzyły z kolei większe zapotrzebowanie na automatyzację testów, co w efekcie dodatkowo spopularyzowało temat samego testowania.

Środowiska, w których uruchamiane są aplikacje, również ulegają ciągłym przemianom. Jeszcze kilkanaście lat temu większość aplikacji działała na pojedynczych komputerach. Jednak ludzie z roku na rok wytwarzają coraz więcej danych, co powoduje, że zasoby dostarczane przez pojedynczy węzeł są niewystarczające. Z tych oraz z ekonomicznych względów wskazane jest rozproszenie aplikacji, aby w możliwie krótkim czasie obsłużyć dane pochodzące od wielu użytkowników naraz.

Testowanie oraz aplikacje rozproszone (określone również jako aplikacje działające w chmurze) są tematami rokrocznie zyskującymi popularność. Mimo to temat testowania aplikacji rozproszonych nie jest często poruszany. Trendy wskazują na to, że aplikacje w chmurze będą sukcesywnie wypierać tradycyjne. Zatem znajomość technik i sposobów na gruntowne przetestowanie takich aplikacji w przyszłości może okazać się niezbędna. W dalszej części artykułu autor wprowadza do tematu testowania aplikacji rozproszonych z prostym przykładem oraz wskazuje na pewne nieuniknione utrudnienia wynikłe z pracy z aplikacjami rozproszonymi.

## APLIKACJE ROZPROSZONE – Z CZYM TO SIĘ JE I DLACZEGO JEST TWARDE JAK ORZECH

Przed przejściem do tematu samego testowania warto pokróćce zapoznać się z różnicą w stosunku do tradycyjnych aplikacji oraz problemów, które nie występują w rozwiązaniach jednowęzłowych. Znakomitą większość różnic i wynikłych z tego problemów sprawdzić można do zdalnej komunikacji. Komunikacja lokalna w rozwiązaniach jednowęzłowych możliwa jest na wiele sposobów, ale zawsze bazuje na wsparciu sprzętu (procesora) lub systemu opera-

cyjnego. Natomiast komunikacja pomiędzy maszynami w systemie rozproszonym zazwyczaj możliwa jest tylko za pomocą sieci. W zdecydowanej większości jest to sieć lokalna ethernet.

O błąd w sprzęcie albo systemie operacyjnym jest ciężko. Można zatem założyć, że komunikacja lokalna jest niezawodna. Sieci nie są już niestety tak niezawodne i o awarię jest o niebo łatwiej.

Kolejnym utrudnieniem jest to, że aplikacja uruchomiona na pojedynczym węźle jego awarię traktuje jako awarię całkowitą. Zatrzymanie działania aplikacji w takim scenariuszu jest akceptowalne. W przypadku aplikacji rozproszonych awaria zazwyczaj dotyczy tylko części systemu. Zatem wymaganiem, któremu musi sprostać rozproszona aplikacja, jest kontynuacja działania nawet w przypadku wystąpienia awarii częściowej (ang. *partial failure*). Jak można się domyślić, nie jest to łatwe zadanie i jest jednym z głównych powodów, dla których aplikacje rozproszone są tak złożone [7].

## TESTOWANIE APLIKACJI ROZPROSZONEY

Złożoność aplikacji rozproszonych przekłada się bezpośrednio na złożoność testów. Liczba możliwych scenariuszy jest tak ogromna, że często największym problemem jest przewidzieć, co może pójść nie tak. Po zaplanowaniu samego scenariusza sprawdzenie za pomocą testu, czy aplikacja poprawnie się zachowuje, jest przysłowią wiśnią na torcie.

Nawet w przypadku małych aplikacji rozproszonych znalezienie np. brzegowych scenariuszy (ang. *edge cases*) może być trudnym zadaniem. W celu zobrazowania tej tezy przygotowany został prosty, skrojony pod potrzeby artykułu, system, na podstawie którego oparte są następne sekcje. Jego konstrukcja przedstawiona jest na Rysunku 1.

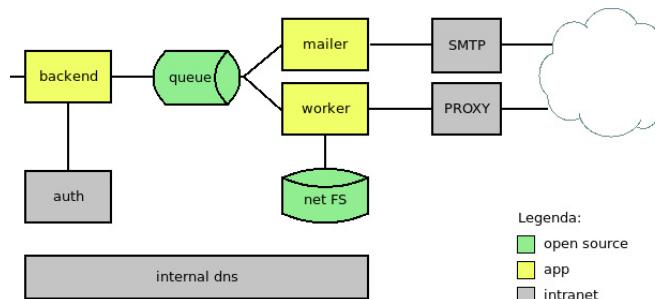
Przykładowym funkcjonalnym zadaniem systemu może być np. pobieranie i przetwarzanie plików binarnych. Składa się on z kilku serwisów:

- » backend – serwis obsługujący żądania użytkowników w formie zapytań HTTP. Każde żądanie zawiera parametr określający lokalizację pliku do przetworzenia (URL),
- » queue – rozproszona kolejka zadań,
- » worker – serwis pobierający i przetwarzający pliki,
- » net fs – usługa udostępniająca pojemy system plików jako zasób sieciowy,
- » mailer – serwis wysyłający wiadomości email do użytkowników po zakończeniu przetwarzania pliku.

Dodatkowo założone jest, że system jest wdrożony wewnętrz sieci firmowej, w której występują:

- » auth – wewnętrzna usługa służąca do uwierzytelnienia (np. LDAP),
- » SMTP – wewnętrzny serwer poczty wychodzącej,
- » proxy – proxy HTTP dla całego ruchu wychodzącego „na świat”,
- » internal DNS – wewnętrzny serwer DNS obsługujący wewnętrzne firmowe domeny (intranet).

Zadanie systemu nie jest skomplikowane. Nie jest on sam również duży. Jak się jednak okaże, przetestowanie takiej aplikacji nie należy do najłatwiejszych zadań.



Rysunek 1. Przykładowy prosty system rozproszony

## CO MOŻE PÓJŚĆ NIE TAK?

Pewne scenariusze wydają się być oczywiste, takie jak np.:

- » błąd uwierzytelnienia użytkownika,
- » błędny URL, w tym:
  - » nieistniejący serwer,
  - » nieistniejący zasób na działającym serwerze,
  - » nieobsługiwany protokół.
- » nieobsługiwany rodzaj pliku.

Inne scenariusze są trochę mniej oczywiste, jednak wprawny tester nie będzie miał problemu z ich zidentyfikowaniem. Mogą to być dla przykładu:

- » nieoczekiwane problemy HTTP(S),
  - » obsługa HTTP 3xx, 403, 500 itp.,
  - » wymagane uwierzytelnienie (HTTP basic auth),
  - » problem z certyfikatem SSL.
- » żądanie ściągnięcia pliku przekraczającego dostępny RAM na maszynie *worker*,
- » problemy I/O podczas zapisu, np. po zapełnieniu się dysku sieciowego.

Jednak nadal nie były to scenariusze, które sprawdzają, jak aplikacja radzi sobie w przypadku awarii związanych bezpośrednio z jej rozproszeniem. Te scenariusze są zarówno rzadziej występujące, jak i o wiele ciekawsze:

- » niedostępność usług zewnętrznych (*auth*, *proxy*, *SMTP*, *DNS*) i wewnętrznych (*queue*, *net fs*, *mailer*, *worker*, *backend*),
  - » niedostępność maszyn (IP),
  - » odrzucenie połączenia przez maszyny (TCP),
  - » problemy związane z DNS.
- » opóźnienia i zawodna sieć,
  - » sztuczne opóźnienia,
  - » tymczasowe partycjonowanie sieci,
  - » losowe gubienie, duplikowanie i zmiana porządku pakietów.
- » obsługa restartów usług wewnętrznych,

- » odporność na restart usług: *backend*, *worker*, *mailer*, *queue* podczas obsługi żądań klientów,
- » restart usługi udostępniającej dysk sieciowy podczas jego użycia.
- » obsługa wielu użytkowników naraz podczas symulowanych problemów sieciowych.

Zdecydowanie nie są to wszystkie scenariusze, które mogą wystąpić. Jednak biorąc pod uwagę tylko te wymienione, wyraźnie widać, że reprezentują one potencjalne problemy, których nie wolno zignorować – chociażby w myśl starej zasady „Better safe than sorry!”. Pomyśły już są, zatem czas na wspomnianą „wisienkę na torcie”. Czy faktycznie będzie łatwo zaimplementować same testy?

## SYMULOWANIE ZAWODNEJ SIECI ZA POMOCĄ TECHNIKI FAULT INJECTION

Większość z wymienionych scenariuszy polega na tymczasowej niedostępności jednego z niezbędnych zasobów. Jeżeli nie bezpośrednio, to symulując niedostępność, można osiągnąć odpowiednie warunki.

Dobór techniki zależy od konkretnej sytuacji, jednak najprawdopodobniej wystarczy do tego symulacja awarii sieci. Takie podejście dostarcza cały wachlarz możliwości sterowania parametrami takimi jak np. czas niedostępności oraz rodzaj niedostępności (jednokierunkowa, dwukierunkowa). Symulowanie awarii sieci ma również małą „bezwładność” – zazwyczaj szybciej można przekonfigurować sieć niż np. zresetować całą usługę.

W jaki sposób sprawnie symulować różne warunki sieciowe, włączając w to awarie? W wielu przypadkach zamiast wykorzystywać do tego fizyczne lub wirtualne maszyny można skorzystać z funkcjonalności dostarczanych przez jądro Linuxa i bardzo popularną ostatnio technologię kontenerów – Docker. Kontenery nie są całkowicie odizolowane od siebie, ponieważ współdzielą kernel, jednak z punktu widzenia symulacji różnych warunków sieciowych zazwyczaj wystarczająca jest sama izolacja na warstwie sieciowej. Kontenery, podobnie jak wirtualne maszyny, mogą być przypisane do wirtualnych interfejsów sieciowych. Zdecydowaną zaletą wykorzystania kontenerów jest ich szybkość działania w porównaniu do tradycyjnych wirtualnych maszyn. Co ciekawe, dzięki tej szybkości testy mogą zostać zautomatyzowane i włączone do procesu testowania po każdej zmianie w kodzie.

Czytelnicy obeznani z systemem Linux zapewne już teraz domyślają się, że do symulowania warunków sieciowych przy założeniu, że każdy kontener jest sieciowo odizolowany, wykorzystać można program tc (ang. *traffic control*) [1]. Jego poprawne wykorzystanie wymaga jednak zapoznania się z całą gamą pojęć oraz może zależeć od konfiguracji samej maszyny. Na szczęście powstał projekt *Blockade*, który ukrywa przed użytkownikami wszelkie trudności, wystawiając bardzo intuicyjny interfejs [0].

## BLOCKADE

Projekt blockade to narzędzie, które jest oparte o Dockera i stworzone zostało, aby ułatwić testowanie awarii sieci w aplikacjach rozproszonych. W działaniu podobne jest trochę do również popularnego narzędzia Docker Compose umożliwiającego uruchamianie wielu kontenerów naraz, bazując na pojedynczym pliku konfiguracyjnym. Blockade nie jest jednak tak rozbudowany, a jego plik konfiguracyjny nie jest kompatybilny z żadną wersją plików konfi-

guracyjnych tego pierwszego. Istnieje jednak sporo podobieństw, co jest znacznym ułatwieniem dla inżynierów już zaznajomionych z Docker Compose. W dalszej części artykułu przedstawione zostanie użycie blockade wraz z plikiem konfiguracyjnym opartym o przykład z Rysunku 1. Cały kod źródłowy użyty w artykule jest dostępny pod adresem [3].

W Listingu 1 zilustrowano, jak wygląda plik konfiguracyjny *blockade.yaml*, który musi znaleźć się w katalogu, w którym wykonywane będą przedstawione w dalszej części tekstu polecenia. Plik ten jest okrojony i nie odzwierciedla całej aplikacji (w zasadzie to ogranicza się do usługi *backend*). Rozważenie w artykule całej aplikacji niepotrzebnie zaciemniłoby obraz.

## Listing 1. Plik konfiguracyjny blockade.yaml

```
containers:
  file_server:
    image: nginx:1.11-alpine
    volumes:
      "./files": "/usr/share/nginx/html/files"

  backend:
    image: blockade-example/backend
    links:
      - file_server
    volumes:
      "./backend": "/app"

network:
  flaky: 30%
  slow: 5000ms 4000ms distribution normal
```

Konfiguracja składa się z dwóch sekcji. Pierwsza z nich definiuje kontenery, a druga sieć. W pliku z Listingu 1 tworzone są dwa kontenery: serwer plików oraz *backend*.

Serwer plików (*file\_server*) to *nginx*, który serwuje pliki do przeprowadzenia testów. Wykorzystany obraz to *nginx* w wersji *1.11-alpine*. Do kontenera przypięty jest katalog *files*, który powinien się znajdować w katalogu głównym projektu.

Drugim kontenerem jest *backend*, którego zachowanie będzie testowane. Wykorzystany obraz to *blockade-example/backend*, który należy samemu wybudować. Backend połączony jest do serwera plików za pomocą dyrektywy *links*, aby móc odwoływać się do niego po nazwie, bez znajomości adresu IP. Kod źródłowy backendu podmontowany jest w kontenerze pod ścieżką */app*. Przykładowy kod usługi *backend* został zamieszczony w Listingu 2. Kod ten celowo nie obsługuje wszystkich możliwych scenariuszy.

Sekcja *network* pliku *blockade.yaml* definiuje parametry sieci. Parametr *flaky* steruje zawodnością, do której zaliczają się m.in. losowe gubienie pakietów oraz ich duplikowanie. Ustawienie tego parametru na 30% powoduje, że w przybliżeniu 1/3 pakietów będzie zgubiona oraz 1/3 zduplikowana. Drugi parametr *slow* decyduje o opóźnieniach, które będą wprowadzane, gdy użytkownik zacznie korzystać z programu. W przedstawionym przypadku opóźnienia będą wały się od 1 do 9 sekund.

## Listing 2. Przykład implementacji serwisu backend

```
#!/usr/bin/env python3

from aiohttp import web
import asyncio
import hashlib
import logging
import requests
import time

NUM = 0
logging.basicConfig(level=logging.INFO)
```

```
async def handle_download(request):
    global NUM
    NUM += 1

    post_data = await request.post()
    if 'url' not in post_data:
        raise web.HTTPBadRequest()

    url = post_data['url']
    logging.info(f'request #{NUM}: {url}')
    start = time.time()

    response = requests.get(post_data['url'])
    data = response.content
    headers = response.headers
    status_code = response.status_code

    stop = time.time()
    hsh = hashlib.md5(data).hexdigest()

    with open(f'/tmp/{hsh}.bin', 'wb') as f:
        f.write(data)

    logging.info(f'request #{NUM}: downloaded {len(data)}/1024/1024MB in {(stop - start):.2f}s. code: {status_code}, hsh: {hsh}')
    return web.Response()

async def init(loop):
    app = web.Application()
    app.router.add_route('POST', '/dl', handle_download)
    return await loop.create_server(app.make_handler(), '0.0.0.0', 8000)

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(init(loop))
    loop.run_forever()
```

Po przygotowaniu pliku konfiguracyjnego można rozpocząć właściwe eksperymentowanie. Tabela 1 zawiera listę poleceń dostępnych w konsolowym kliencie *Blockade*. Naturalnie pierwszą aktywnością będzie uruchomienie konfiguracji. Jeżeli wszystko pójdzie zgodnie z planem, w odpowiedzi otrzymamy status kontenerów i sieci, tak jak zostało to przedstawione poniżej:

```
~/c/blockade-example $> blockade status
NODE      CONTAINER ID STATUS IP          NETWORK   PARTITION
backend   f3b7c651ee4f UP     172.17.0.3 NORMAL
file_server c5b74dfb8104 UP     172.17.0.2 NORMAL
```

Kolumny zawierają nazwę węzła, identyfikator kontenera, status, adres IP, tryb sieci oraz partycję sieci, w której znajduje się dany kontener. Sprawdzenie poprawności uruchomienia najlepiej przeprowadzić organoleptycznie. W tym celu najlepiej ręcznie poinstruować *backend*, aby ściągnął plik z serwera. Najpierw trzeba ów plik wygenerować. Jego zawartość nie ma znaczenia, więc można użyć np. polecenia dd do wygenerowania losowego pliku o rozmiarze 1 MB:

```
dd if=/dev/urandom of=files/1MB.bin bs=1M count=1
```

Aby maksymalnie ułatwić sobie życie, najlepiej również stworzyć prosty skrypcik, który będzie inicjował ściągnięcie przez *backend* pliku z serwera. W Listingu 3 zawarto przykład takiego skryptu, który zakłada, że cały projekt znajduje się w katalogu o nazwie *blockadeexample*.

## Listing 3. Prosty skrypt inicjujący ściągnięcie przez serwis *backend* pliku z serwera *file\_server*

```
#!/usr/bin/env bash
IP=`docker inspect --format '{{ .NetworkSettings.IPAddress }}' blockadeexample_backend`
exec curl -XPOST "$IP:8000/dl" -d "url=http://file_server/files/1MB.bin"
```

Od tego momentu wystarczy wywołać skrypt (autor nazwał go dl) i obserwować, co się dzieje w wyjściu standardowym kontenera backend. Wygodnie jest to wyjście gdzieś wyświetlić, np. w konsoli z boku, używając poniższego polecenia:

```
docker logs -f blockadeexample_backend
```

Uruchomienie skryptu ./dl powinno spowodować pojawienie się logów z kontenera backend podobnych do tych przedstawionych poniżej:

```
INFO | request #1: http://file_server/files/1MB.bin
INFO | request #1: downloaded 1.0MB in 0.01s. code: 200, hsh: e4b7f62da691604a692844e93304de6b
INFO | 172.17.0.1 - - [01/Dec/2017:00:26:08 +0000] "POST /dl HTTP/1.1" 200 0 "-" "curl/7.50.1"
```

## EKSPERYMenty

Plik o rozmiarze 1 MB ściąga się w 1/100 sekundy, zatem ciężko coś przetestować. Polecenie blockade slow file\_server spowoduje, że ściąganie nie będzie już takie szybkie. Po jego uruchomieniu można sprawdzić status, aby się przekonać, że stan sieci uległ zmianie:

```
~/c/blockade-example $> blockade slow file_server
~/c/blockade-example $> ./dl
~/c/blockade-example $> blockade status
NODE      CONTAINER ID STATUS   IP      NETWORK PARTITION
backend   f3b7c651ee4f UP      172.17.0.3 NORMAL
file_server c5b74dfb8104 UP      172.17.0.2 SLOW
```

Tym razem uruchomienie samego skryptu ./dl powinno przynieść oczekiwane wyniki:

```
INFO | request #2: http://file_server/files/1MB.bin
INFO | request #2: downloaded 1.0MB in 22.29s. code: 200, hsh: e4b7f62da691604a692844e93304de6b
INFO | 172.17.0.1 - - [01/Dec/2017:00:28:53 +0000] "POST /dl HTTP/1.1" 200 0 "-" "curl/7.50.1"
```

Jak widać, spowolnienie sieci spowodowało wzrost czasu ściągania do 23 sekund. Suma md5 jest nadal taka sama, więc plik dotarł w całości. Długi czas ściągania się pliku umożliwia testowanie wiele innych scenariuszy.

Innym przydatnym poleceniem Blockade jest partition. Umożliwia on podzielenie sieci między kontenerami w taki sposób, aby kontenery w różnych grupach nie mogły się ze sobą komunikować na czas trwania podziału. W przedstawianym przykładzie, aby tymczasowo całkowicie odseparować serwer od backendu, należy użyć polecenia blockade partition file\_server. Zapytanie o status powinno odzwierciedlić wdrożony stan:

NODE	CONTAINER ID	STATUS	IP	NETWORK	PARTITION
backend	f3b7c651ee4f	UP	172.17.0.3	NORMAL	2
file_server	c5b74dfb8104	UP	172.17.0.2	SLOW	1

Zawarta w artykule przykładowa implementacja nie radzi sobie już tak dobrze, jeżeli używa się partycjonowania sieci, następnie zacznie ściągać plik i na końcu przywróci sieć do normalności za pomocą polecenia blockade join. Ilustruje to poniższy log, z którego niektóre części zostały usunięte dla czytelności. Wytyłuszczenia zostały dodane przez autora:

```
INFO | request #3: http://file_server/files/1MB.bin
ERROR | Error handling request
Traceback (most recent call last):
  File "/usr/local/lib/python3.6/site-packages/urllib3/
connection.py", line 141, in _new_conn
    (self.host, self.port), self.timeout, **extra_kw)
[...]
TimeoutError: [Errno 110] Operation timed out

During handling of the above exception, another exception
occurred:

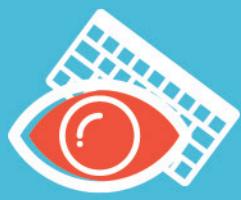
Traceback (most recent call last):
  File "/usr/local/lib/python3.6/site-packages/urllib3/
connectionpool.py", line 601, in urlopen
    chunked=chunked)
[...]
  File "/usr/local/lib/python3.6/site-packages/requests/
adapters.py", line 508, in send
    raise ConnectionError(e, request=request)
requests.exceptions.ConnectionError:
HTTPConnectionPool(host='file_server', port=80): Max
retries exceeded with url: /files/1MB.bin (Caused by
NewConnectionError('<urllib3.connection.HTTPConnection object
at 0x7fe57ee51748>: Failed to establish a new connection:
[Errno 110] Operation timed out',))
```

Jak wiadomo, nieobsłużone wyjątki mogą spowodować wystąpienie problemów w szerszym kontekście, więc traktować je można jako błędy.

reklama



**BOOTCAMP**  
**Tester Automatyczny**



**BOOTCAMP**  
**Full-stack**



**BOOTCAMP**  
**Data Science**

PODEJMIJ WZYWANIE

**Zdobądź**  
**zawód**  
**przyszłości**

Dowiedz się więcej na  
**odołamacz.pl**

**sages**

Ciekawostką jest to, że otrzymany rezultat wynika z wykorzystania TCP w HTTP. Wprowadzanie opóźnień i partycji pomiędzy usługą korzystającą z UDP może przynieść nawet ciekawsze wyniki z powodu nie zgłoszenia błędu w transmisji danych przez biblioteki w ogóle.

Ostatnim bardzo ciekawym trybem sieci, który można uzyskać, korzystając z blockade, jest tryb `flaky`. Wprowadza on losowe duplikacje i opóźnienia wraz z intencjonalnym gubieniem części pakietów. Po uruchomieniu polecenia `blockade flaky --all`, a następnie uruchomieniu skryptu `./d1` wynik zazwyczaj jest taki sam jak przy wprowadzaniu partycji. Jeżeli jednak w trakcie, gdy sieć jest w trybie `flaky`, a backend ściąga plik, przywróci się sieć do normalnego stanu, wyniki mogą być nieoczekiwane i zaskoczyć nawet wytrawnych programistów [4]. Ilustrują to rezultaty, które autor wielokrotnie reprodukował – co przedstawiono w Listingu 4. Wytyłuszczenie wskazuje najważniejsze fragmenty.

Jak widać, nie zawsze zastosowana implementacja ściągą plik w całości. Problem jest nawet większy niż w przypadku wyjątków, ponieważ jest przemilczany przez komponent, w którym wystąpił, i naprawdopodobniej pojawi się niedługo w innej części systemu (lub, co gorsza, po dłuższym czasie).

Po sprawdzeniu zawartości plików, dla których suma `md5` się nie zgadzała, okazało się, że są one ściągnięte tylko w części. Z eksperymentów wynika, że około 1-2% żądań kończy się właśnie w taki sposób. Testy autor przeprowadził, korzystając z Pythona w wersji 3.6-alpine, Blockade w wersji 0.4.0 oraz biblioteki requests w wersji 2.18.4.

Zmiana implementacji na ekwiwalenty wykorzystujące `urllib.request.urlopen` i `urllib3.connectionpool` powoduje, że w miejscach wspomnianych 1-2% pojawiają się trzy rodzaje nieobsłużonych wyjątków. W pierwszym z nich liczby naturalnie są różne w zależności od zaistniałej sytuacji.

- » `http.client.IncompleteRead: IncompleteRead(305528 bytes read, 743048 more expected)`
- » `http.client.RemoteDisconnected: Remote end closed connection without response`
- » `urllib3.exceptions.ProtocolError`

Jest kilka wniosków płynących z przedstawionych rezultatów. Najistotniejszym z nich z perspektywy artykułu jest to, że jeżeli przedstawione błędy występują nawet pomimo zastosowania bardzo dobrze przetestowanych bibliotek (requests), to ich mnogość w rozwiązaniach własnych może być o wiele większa. Zatem warto pokusić się o napisanie odpowiednich testów i potwierdzenie, że aplikacja w przypadku zaistnienia różnych awaryjnych sytuacji zachowuje się poprawnie. Blockade jest nieodzownym narzędziem, dzięki któremu uwaga może zostać skupiona na scenariuszach, bez wgłębiania się w techniczne szczegóły symulowania awarii.

Przedstawiony w artykule przykład jest prosty, jeżeli nie trywialny. W przypadku bardziej skomplikowanych systemów rozproszonych, jak np. rozproszone bazy danych, ilość scenariuszy jest spłogowana, ponieważ wiele operacji takich jak wybór mastera czy rozproszone lockowanie zależą od warunków sieciowych (Tabela 1).

## NIE TYLKO SIEĆ

Z pomocą Dockera bardzo łatwo przetestować można również scenariusze niezwiązane z siecią, a występujące w systemach rozproszonych. Za przykłady mogą posłużyć synchronizacja i czas. Te aspekty w aplikacjach rozproszonych odgrywają bardzo ważną rolę. Ponieważ kontenery są od siebie niezależne, w każdym z nich można różnie ustawić czas i przetestować zachowanie się aplikacji jako całości. Do symulacji czasu wewnętrz kontenerów posłużyć może biblioteka libfaketime [2].

Warto również przetestować scenariusz interakcji z usługami zewnętrznymi w przypadku, gdy czas jest źle ustawiony. Tutaj przykładem może być popularna usługa S3. Klienci API często podpisują żądania, korzystając z lokalnego czasu, co w przypadku przesunięcia (np. czas zimowy/letni) może wywołać błędów. Błąd, co prawda, nie leży wówczas w samej aplikacji, tylko w konfiguracji systemu operacyjnego, lecz system ten naprawdopodobniej był konfigurowany przez rozwiązanie CM (ang. *configuration management*), które bazowało na recepturach dostarczonych przez aplikację. Tajemniczą polisynonimą jest to, że zadania programistów aplikacji rozproszonych często przeplatają się z tym, czym na co dzień zajmują się koledzy z działu DevOps.

**Listing 4. Fragmenty wyjścia standardowego usługi backend podczas trybu flaky**

```
[...]  
INFO | request #49: http://file_server/files/1MB.bin  
INFO | request #49: downloaded 1.0MB in 47.81s. code: 200, hsh: e4b7f62da691604a692844e93304de6b  
INFO | request #50: http://file_server/files/1MB.bin  
INFO | request #50: downloaded 0.082855224609375MB in 64.80s. code: 200, hsh: 0d7e20c9f7be82c70c0c5f66664c1552  
INFO | request #51: http://file_server/files/1MB.bin  
INFO | request #51: downloaded 1.0MB in 35.27s. code: 200, hsh: e4b7f62da691604a692844e93304de6b  
INFO | request #52: http://file_server/files/1MB.bin  
INFO | request #52: downloaded 1.0MB in 35.32s. code: 200, hsh: e4b7f62da691604a692844e93304de6b  
INFO | request #53: http://file_server/files/1MB.bin  
INFO | request #53: downloaded 1.0MB in 48.23s. code: 200, hsh: e4b7f62da691604a692844e93304de6b  
INFO | request #54: http://file_server/files/1MB.bin  
INFO | request #54: downloaded 1.0MB in 48.49s. code: 200, hsh: e4b7f62da691604a692844e93304de6b  
INFO | request #55: http://file_server/files/1MB.bin  
INFO | request #55: downloaded 1.0MB in 37.27s. code: 200, hsh: e4b7f62da691604a692844e93304de6b  
[...]  
INFO | request #67: http://file_server/files/1MB.bin  
INFO | request #67: downloaded 1.0MB in 41.97s. code: 200, hsh: e4b7f62da691604a692844e93304de6b  
INFO | request #68: http://file_server/files/1MB.bin  
INFO | request #68: downloaded 0.082855224609375MB in 91.45s. code: 200, hsh: 0d7e20c9f7be82c70c0c5f66664c1552  
INFO | request #69: http://file_server/files/1MB.bin  
INFO | request #69: downloaded 0.20023345947265625MB in 62.63s. code: 200, hsh: 81b6ef76b496ad69860973b2ea5a4c12  
INFO | request #70: http://file_server/files/1MB.bin  
INFO | request #70: downloaded 1.0MB in 36.43s. code: 200, hsh: e4b7f62da691604a692844e93304de6b
```

Polecenie	Opis
up	Uruchamia całą konfigurację, zgodnie z zawartością <i>blockade.yaml</i>
destroy	Zatrzymuje i niszczy wszystkie kontenery
status	Wyświetla status każdego z kontenerów, status sieci oraz przynależność kontenerów do partycji sieciowych
start, restart, stop	Odpowiednio: startuje, restartuje oraz zatrzymuje pojedynczy kontener
kill	Zatrzymuje kontener za pomocą sygnału <i>KILL</i>
logs	Wyświetla logi (wyjście standardowe) kontenerów
flaky	Przełączanie sieci w tryb zawodny
slow	Przełączanie sieci w tryb wolny
duplicate	Włącza duplikowanie pakietów w sieci
fast	Przywraca sieć do normalnej szybkości
partition	Tworzy partycje sieciowe między podanymi kontenerami
join	Usuwa wszystkie istniejące partycje
chaos	Wchodzi w tryb losowego uruchamiania awarii
events	Zwraca listę wszystkich zdarzeń wraz z timestampem
daemon	Przechodzi w tryb serwera, udostępniając interfejs REST
add	Pozwala na dynamiczne dołączenie kontenerów trzecich do sieci
version	Wyświetla wersję blockade i kończy działanie

Tabela 1. Lista polecen blockade

## Blockade chaos

Jedną z opcji dostępną w blockade jest *chaos*, która prawdopodobnie wzoruje się na znany z Netflixem *Chaos Monkey*. Jest to polecenie blokujące, które losowo emuluje różne awarie sieci. Mechanizm ten swoją ideą zbliżony jest do tzw. fuzzowania. Można zatem po prostu uruchomić swoją aplikację rozproszoną, wykonać polecenie *blockade chaos* i poczekać kilka godzin w celu sprawdzenia, czy aplikacja przetrwa w całości.

*blockade chaos* przyjmuje dodatkowe opcje, którymi można sparametryzować jego pracę. Przedstawione one zostały w Tabeli 2.

Parametr Blockade Chaos	Opis
--events	Wybór rodzajów zdarzeń. Domyślnie wszystkie są włączone
--degrade-delay-min/max	Minimalny/maksymalny czas, który blockade odczeka pomiędzy emulowaniem kolejnych zdarzeń
--degrade-runtime-min	Minimalny/maksymalny czas trwania kolejnych zdarzeń
--containers-at-once-min	Minimalna/maksymalna liczba kontenerów objętych zdarzeniem naraz

Tabela 2. Opcjonalne parametry polecenia blockade chaos

Innym przykładem testowania niezwiązanego bezpośrednio z siecią jest emulacja długich cykli GC (ang. *garbage collecting*), jeżeli serwisy aplikacji napisane są w językach zarządzanych. W ten sposób można zaobserwować, co się dzieje, kiedy w buforach systemowych gromadzą się nieprzetworzone wiadomości od innych serwisów aplikacji.

Ostatecznie wspomniany blockade posiada pewną opcję, której nie ma w samym Dockerze, jak i Docker Compose: *start\_delay*. Dzięki niej przetestować można scenariusz, w którym konkretna usługa wstaje z opóźnieniem. Taka sytuacja jest często spotykana „w naturze”.

## KIEDY DOCKER NIE WYSTARCZA

We wcześniejszych sekcjach wspomniane było, że kontenery zapewniają pewną izolację, która wystarcza do przeprowadzenia przedstawionych scenariuszy testowych. Nie sposób jednak za pomocą Dockera przetestować wszystkiego. Kontenery współdzielą bowiem kernel, więc wszystkie usługi, które żyją w przestrzeni kernela, wymagają zastosowania innych rozwiązań, aby je przetestować.

Przykładem usługi zintegrowanej z kerolem, a która jest bardzo narażona na warunki sieci, jest chociażby NFS – standardowy linuksowy sieciowy system plików.

## Oskryptowywanie blockade

Blockade może pracować w tzw. trybie demona (ang. *daemon*). W tym trybie blockade startuje serwer HTTP, który wystawia RESTowe API. Wszystkie polecenia, które można wykonać z linii poleceń, mają swoje odpowiedniki w API. Co ciekawe, za pomocą API sterować można kilkoma konfiguracjami naraz.

Komunikacja z serwerem polega na wysyłaniu dokumentów JSON (stosując *curl*, nie można zapomnieć o ustawieniu nagłówka Content-Type na wartość *application/json!*). W Tabeli 3 zawarto wystawiane przez blockade końcówki API.

Ścieżka	Metoda	Opis
/blockade	GET	Wszystkie zarejestrowane konfiguracje
/blockade/<nazwa>	GET	Informacje o konfiguracji
	POST	Tworzenie konfiguracji
	PUT	Modyfikacja konfiguracji
	DELETE	Kasowanie konfiguracji
/blockade/<nazwa>/action	POST	Wywołanie zdarzenia: <code>start, stop, restart, kill</code>
/blockade/<nazwa>/partitions	POST	Tworzenie partycji sieci
	DELETE	Usuwanie partycji sieci
/blockade/<nazwa>/network_state	POST	Zmiana trybu działania sieci: <code>flaky, slow, fast, duplicate</code>
/blockade/<nazwa>/events	GET	Pobranie listy zdarzeń wraz z timestampami
/blockade/<nazwa>/chaos	GET	Pobranie statusu trybu chaos
	POST	Rozpoczęcie trybu chaos
	PUT	Modyfikacja parametrów trybu chaos
	DELETE	Zakończenie trybu chaos

Tabela 3. Końcówki API serwera blockade

Jeżeli aplikacja korzysta z NFSa, lub odpowiednika, tę część przetestować można, wykorzystując wirtualne maszyny. Do całej reszty prawdopodobnie nadal będzie można skorzystać z blockade.

## DODATKOWE UTRUDNIENIA W APLIKACJACH ROZPROSZONYCH

Po wdrożeniu i uruchomieniu aplikacji rozproszonej każda czynność związana z jej utrzymaniem jest cięższa niż gdyby aplikacja była jednowzłowa. Ten temat wykracza poza ramy tego artykułu, zatem nie zostanie zgłębiony. Warto jednak o nim pamiętać. Czynności takie jak monitoring wydajności, zbieranie logów, debugowanie, reprodukowanie, backup, snapshot itp. w aplikacjach rozproszonych są zdecydowanie bardziej skomplikowane i często wymagają wdrażania dodatkowych zcentralizowanych rozproszonych rozwiązań, komplikując samą aplikację jeszcze bardziej. Innym nieporuszonym, a zarazem ważnym tematem jest testowanie obciążenia (ang. *stress test*).

## PODSUMOWANIE

Rzeczą oczywistą jest to, że oprogramowanie zawiera błędy. Prawdopodobnie dużo więcej błędów, niż wyobrażają to sobie programiści. Testowanie jest zatem koniecznością. W przypadku aplikacji rozproszych staje się to jeszcze bardziej istotne ze względu na rozszerzoną liczbę możliwych scenariuszy.

Przedstawione w artykule narzędzie blockade pozwala na symulowanie awarii sieci, co zazwyczaj umożliwia pokrycie większości przypadków testowych aplikacji rozproszonej. Z pomocą blockade skupić się można na implementacji konkretnych scenariuszy. Blockade jest skuteczne nawet w przypadku rozwiązań rozwijanych od lat i przetestowanych w wielu projektach.

### W sieci

- [0] <https://github.com/worstcase/blockade>
- [1] <https://goo.gl/sYF7vE>
- [2] <https://github.com/wolfcw/libfaketime>
- [3] <http://github.com/szborows/blockade-example>
- [4] <https://github.com/requests/requests/issues/4415>
- [5] [https://en.wikipedia.org/wiki/Waterfall\\_model](https://en.wikipedia.org/wiki/Waterfall_model)
- [6] [https://en.wikipedia.org/wiki/Continuous\\_testing](https://en.wikipedia.org/wiki/Continuous_testing)
- [7] <https://goo.gl/o4BRvn>



**SŁAWOMIR SZBOROWSKI**

<http://szborows.blogspot.com>

Absolwent Informatyki Politechniki Wrocławskiej. Programuje odkąd pamięta. Obecnie pracuje w firmie Nokia Wrocław na stanowisku Distributed Systems Engineer. Zainteresowania: optymalizacja oraz jakość oprogramowania, systemy rozproszone, budowa kompilatorów.



## AUDYTY KODU!

### CZY ZNASZ JUŻ WSZYSTKIE DEFEKTY W SWOIM KODZIE?

Nie czekaj zanim będzie za późno i Twój klient znajdzie je na prodzie.

Profesjonalne audyty dostępne od ręki i to jeszcze dziś!

ZADZWOŃ: +48 533 315 222

## TESTOWANIE FUNKCJONALNE

Nie chcesz sam testować swojego softu? Zleć to profesjonalistom z [testerzy.pl](#)

Najlepsze testy w tej części Europy.

Odwiedź nas w biurze i porozmawiajmy o testowaniu.

ul. Mickiewicza 15 p. 411, Katowice

## OGŁOSZENIA DROBNE

## OFERUJEMY!

### AUDYT AUTOMATYZACJI TESTÓW

Jeśli Twoją automatyzację robi 2 juniorów w przerwach między klikaniem, a nauką z syllabusa ISTQB możesz potrzebać oceny tej pracy.

Zadzwoń teraz. Pomożemy!

+48 32 733 09 58

## TESTY WYDAJNOŚCI

## BEZ POŚREDNIKÓW

*Twoja strona ładuje się godzinami? Klienci odchodzą? Nie martw się.*

Nie tylko przetestujemy stronę, ale i podpowiemy co należy zoptymalizować.

Zapraszamy do kontaktu.

[kontakt@testerzy.pl](mailto:kontakt@testerzy.pl)

## POSZUKIWANY

- programista C#
- programista Frontend ze znajomością JS/Angular/jQuery

Stabilne warunki zatrudnienia, atrakcyjne wynagrodzenie w zależności od posiadanych kompetencji, ciekawe projekty, nietuzinkowy zepół.

Sprawdź aktualne oferty pracy

[testerzy.pl](#)

## TESTY & TESTOWANIE

Różne umiejętności i doświadczenie  
Zawsze najwyższej jakości  
Dostępne od ręki



[testerzy.pl](#)

# Złożoność obliczeniowa algorytmów – ciąg dalszy (algorytmy rekurencyjne)

Celem napisania tego artykułu było przedstawienie metod wyznaczania rzędu złożoności obliczeniowej algorytmów rekurencyjnych. Algorytmy rekurencyjne stosuje się w informatyce w obszarach, w których rozwiązywanie zadania wymaga podziału problemu na mniejsze podproblemy. Do zjawiska rekurencji należy podchodzić z dystansem, ponieważ algorytm rekurencyjny nie zawsze charakteryzuje się dobrą wydajnością. Pisząc artykuł, autor zaktąda, że czytelnik zna podstawowe pojęcia związane ze złożonością obliczeniową algorytmów. Niniejsza praca jest uzupełnieniem do artykułu „Wyznaczanie złożoności obliczeniowej algorytmów”, który pojawił się w numerze 10/2017.

## Zjawisko rekurencji w naukach ścisłych

**Rekurencja** jest techniką programowania, dzięki której procedura, funkcja lub podprogram jest w stanie w swoim ciele wywołać sam siebie. Dzięki rekurencji łatwo można znaleźć rozwiązanie zadania, w którym potrzebne są wyniki cząstkowych obliczeń w celu obliczenia całości. Rekurencję definiuje się poprzez podanie **wartości początkowych** oraz z równania wyrażającego **ogólną wartość** za pomocą wartości wcześniejszych wyrazów. Jednym z klasycznych przykładów w zagadnienniu rekurencji jest liczenie wartości ciągu Fibonacciego:

### Przykład 1.1

$$f(n) = \begin{cases} 1, & n \in \{0,1\} \\ f(n-1) + f(n-2), & n > 1 \end{cases}$$

Rekurencja może być (z dobrym skutkiem) stosowana w najefektywniejszych algorytmach sortowania lub przeszukiwania zbioru. W celu wyznaczenia wyniku zadania. Jedną ze strategii projektowania algorytmów jest metoda „dziel i zwyciężaj”, która na ogół korzysta z algorytmów rekurencyjnych. W strategii tej problem dzieli się rekurencyjnie na dwa lub więcej mniejszych podproblemów tego samego typu tak długo, aż cząstkowe problemy staną się wystarczająco proste do bezpośredniego rozwiązania. Z kolei rozwiązania otrzymane dla podproblemów łączy się w całość, łącząc rozwiązania pierwotnego zadania. Rekurencja jest wykorzystywana w wielu sytuacjach, wśród których najpopularniejsze to:

- » algorytmy wyszukiwania (na przykład quick sort),
- » rekurencyjne struktury danych (na przykład lista wiązana),
- » specyficzne algorytmy, w których wykorzystanie rekurencji jest czymś naturalnym.

Rekurencyjny opis obliczeń (funkcji czy metod) jest na ogół **zwarty** (**krótszy, przejrzysty**) w przeciwieństwie do funkcji, które nie używają rekurencji. Należy jednak zwrócić uwagę na to, że zwartość opisu rekurencyjnego nie zawsze idzie w parze z efektywnością realizacji algorytmów. Dlatego też algorytmy rekurencyjne powinny być stosowane rozsądnie. Ciekawy jest również fakt, że prawie każde

rozwiązanie rekurencyjne można odwzorować w formie klasycznej (podać jawnego wzór takiej funkcji albo algorytm iteracyjny)[4]. Często jest to jednak problem skomplikowany nie tylko pod względem programistycznym, ale także matematycznym. Istnieje jednak pewna klasa funkcji rekurencyjnych, które można przedstawić jawnym wzorem typu  $f(n)$ , gdzie  $n \in N$ , poprzez rozpisanie funkcji do takiego momentu, aż uda się zauważyc pewną zależność, a następnie na tej podstawie przedstawić wzór procedury programu sposobem klasycznym.

W matematyce za pomocą rekurencji można tworzyć takie obiekty jak [3]:

- » Wielomiany Hermite'a,
- » Wielomiany Legendre'a,
- » Algorytm Euklidesa,
- » Silnię,
- » Symbol Newtona,
- » Cechę podzielności przez 3 dla liczby w zapisie dziesiętnym,
- » Funkcje wykładnicze,
- » Niektóre funkcje logarytmiczne,
- » Schemat Hornera.

## WYZNACZANIE ZŁOŻONOŚCI OBLCZENIOWEJ ALGORYTMÓW REKURENCYJNYCH

Wyznaczanie złożoności obliczeniowej algorytmu rekurencyjnego wymaga pewnych umiejętności matematycznych, związanych z algebraicznymi metodami rozwiązywania równań rekurencyjnych (funkcyjnych). Nie jest to rzecz trywialna, wobec czego będzie zaprezentowana krok po kroku, aby czytelnik mógł zrozumieć metodę wyznaczania złożoności obliczeniowej algorytmów rekurencyjnych.

### Przykład 2.1

Załóżmy, że istnieje pewne drzewo, którego wzrost można opisać następującymi regułami:

1. Po upływie pierwszego miesiąca roślina urosła do pewnej jednostki o wartości 1,
2. Podstawową jednostką wzrostu jest miesiąc,



Zapraszamy na autorskie szkolenia  
z zakresu **bezpieczeństwa IT**

- { Bezpieczeństwo aplikacji WWW }
- { Offensive HTML, SVG, CSS and other Browser-Evil }
- { Wprowadzenie do bezpieczeństwa IT }
- { Szkolenie przygotowujące do egzaminu CEH  
( Certified Ethical Hacker ) }

[www.securitum.pl/oferta/szkolenia](http://www.securitum.pl/oferta/szkolenia)

Patroni medialni: [sekurak.pl](http://sekurak.pl)



[rozwal.to](http://rozwal.to)



3. Musi minąć dwa razy więcej miesięcy, aby drzewo urosło o kolejną jednostkę o wartości 1 w stosunku do trwania tego czasu w przeszłości, w którym drzewo również zwiększyło wysokość o 1.

Naszym zadaniem jest napisanie funkcji, która obliczy wysokość rośliny po określonym miesiącu, oraz wyznaczenie lub oszacowanie jej złożoności obliczeniowej (nie zawsze da się wyznaczyć jawnego wzór prezentujący rząd złożoności obliczeniowej). Treść zadania sugeruje, że można wykorzystać do rozwiązywania postawionego problemu zależność rekurencyjną. Zapisanie takiej funkcji w języku C# ma następującą postać:

```
namespace WzrostRosliny
{
    class Program
    {
        public static void Main(string[] args)
        {
            int n = 16;
            Console.WriteLine(Wzrost(n));
            Thread.Sleep(4000);
        }
        public static int Wzrost(int n)
        {
            if (n == 1) return 1;
            else return Wzrost(n/2)+1;
        }
    }
}
```

Matematyczny zapis takiej funkcji można przedstawić analogicznie jak funkcję z przykładu 1.1:

$$f(n) = \begin{cases} 1, & n = 1 \\ f(n/2) + 1, & n > 1 \end{cases}$$

gdzie  $n \in N$ ,  $n = 2^k$ ,  $k \in N$  lub  $k = 0$ . Jeśli do analizy złożoności weźmiemy takie operacje komputera jak:

- » Podstawienie (nadawanie wartości zmiennej),
- » Wykonanie operatora arytmetycznego, relacyjnego lub logicznego,
- » Obliczenie wartości zmiennej indeksowanej (tablicy),
- » Wywołanie metody,
- » Przekazanie aktualnej wartości parametru,
- » Wykonanie instrukcji wejścia i wyjścia,

to funkcja, która liczy złożoność obliczeniową naszego algorytmu, będzie miała postać:

$$c(n) = \begin{cases} 2, & n = 1 \\ c(n/2) + 3, & n > 1 \end{cases}$$

Ponieważ dla  $n = 1$  mamy:

```
public static int Wzrost(int n)
{
    if (n == 1) return 1;
    else return Wzrost(n/2)+1;
}
```

natomiast, gdy  $n > 1$ , to:

```
public static int Wzrost(int n)
{
    if (n == 1) return 1;
    else return Wzrost(n/2)+1;
}
```

Istnieje kilka sposobów, aby rozwiązać równanie rekurencyjne. Wchodzą tutaj w grę umiejętności matematyczne. Zanim jednak zaczniemy rozwiązywać równanie rekurencyjne, to należy określić, czy jest ono określone dla wszystkich liczb naturalnych. W wyżej przedstawionym przykładzie funkcja obliczająca wartość dla poprzedniego argumentu dostaje argument  $n/2$ , a to oznacza, że liczba  $n$  musi być wielokrotnością liczby 2, czyli  $n = 2^k$ , gdzie  $k \in N$  lub  $k = 0$ . Sposób, który zaprezentujemy do rozwiązywania równania  $c(n)$ , będzie opierał się na rozpisaniu równania tak, aby uzyskać fragment, w którym osiągniemy  $c(1)$  – po to, aby odgadnąć jawnego wzoru naszego równania. Po otrzymaniu (odgadnięciu rozwiązania) zostanie przeprowadzony dowód indukcyjny, który będzie potwierdzeniem poprawności naszego rozwiązania. Podsumowując, rozwiązanie równania rekurencyjnego składa się z takich kroków jak:

1. Ustalenie dziedziny rozwiązania,
2. Rozwiązywanie równania – wyznaczenie wzoru,
3. Dowód indukcyjny poprawności rozwiązania.

**Krok 1** został już, w naszym przypadku, wykonany:  $n = 2^k$ , gdzie  $k \in N$  lub  $k = 0$

#### Krok 2:

$$c(n) = c\left(\frac{n}{2}\right) + 3 = \left(c\left(\frac{n}{4}\right) + 3\right) + 3 = c\left(\frac{n}{4}\right) + 3 + 3 = c\left(\frac{n}{4}\right) + 2 \cdot 3$$

$$c(n) = c\left(\frac{n}{4}\right) + 2 \cdot 3 = \left(c\left(\frac{n}{8}\right) + 3\right) + 2 \cdot 3 = c\left(\frac{n}{8}\right) + 3 \cdot 3$$

Gdy powyższe równanie będziemy rozpisywać dalej, to dojdziemy do momentu, gdy:

$$c(n) = c\left(\frac{n}{2^k}\right) + (3 + 3 + \dots + 3)$$

Liczba 3 powtarza się dokładnie  $k$ -razy. Można to wywnioskować np. z tego, że:

$$c\left(\frac{n}{8}\right) = c\left(\frac{n}{2^3}\right) + 3 + 3 + 3$$

Idąc dalej:

$$c(n) = c\left(\frac{n}{2^k}\right) + (3 + 3 + \dots + 3) = c\left(\frac{n}{2^k}\right) + k \cdot 3 = c(1) + k \cdot 3$$

Wyrażenie  $n = 2^k$  jest równoważne temu, że  $k = \log(n)$ , co wynika z definicji logarytmu (przez  $\log(n)$  będziemy oznaczać logarytm przy podstawie 2). Wobec tego:

$$c(n) = 2 + \log(n) \cdot 3 = 2 + 3 \log(n)$$

Korzystając kolejny raz z własności logarytmów:

$$2 = \log(4)$$

$$3 \log(n) = \log(n^3)$$

mamy:

$$c(n) = \log(4) + \log(n^3) = \log(4 \cdot n^3)$$

**Krok 3:** Dowód indukcyjny. Dowody korzystające z zasady indukcji matematycznej składają się z dwóch kroków. Pierwszym jest dowód prawdziwości dla warunków początkowych, co w praktyce jest zwykle dość proste, ale nie wolno zaniedbywać tego kroku. W drugim kroku zakłada się prawdziwość otrzymanego wzoru dla zmiennej  $n$  – założenie to jest hipotezą indukcyjną i pod tym założeniem dowodzi się prawdziwości wzoru dla kolejnego względem  $n$  argumentu. Dowód przez indukcję nie będzie pełny (ani poprawny), jeśli przeprowadzi się tylko pierwszy krok, a pominie drugi bądź wykona drugi z kroków, a opuści pierwszy [1].

- Niech  $n = 1$ , wobec czego  $\log(4 \cdot 1^3) = \log(4) = 2$ , co należało wykazać.
- Zakładamy, że wzór  $c(n) = \log(4 \cdot n^3)$  jest prawdziwy dla każdej liczby naturalnej  $n$ , a wykażemy, że wzór jest prawdziwy dla każdej liczby  $2n$ :

$$\begin{aligned} c(2n) &= \log(4 \cdot (2n)^3) = \log(4 \cdot 2^3 \cdot n^3) = \\ &= \log(2^3) + \log(4 \cdot n^3) = 3 + c(n), \end{aligned}$$

co należało udowodnić. Wobec powyższego rozumowania stwierdzamy, że złożoność obliczeniowa algorytmu Wzrost jest rzędu  $\Theta(\log(n))$ .

### Przykład 2.2 [2]

Założmy, że pewna firma handlowa potrzebuje aplikacji, która usprawni zarządzanie wynagrodzeniami dla pracowników (w tym również systemem wypłaty premii). Należy zaimplementować metodę, która ma liczyć premię od ilości sprzedanych napojów, które są pakowane w pakiety. Premia może być wypłacona w dowolnym czasie (ale od całych pakietów z napojami), o czym decyduje osoba, której ta premia się należy. Zarząd firmy chce, aby:

- Premia od sprzedaży uzależniona była od ilości poprzednio sprzedanych pakietów,
- Do bieżącej sprzedaży dodawane było wyrażanie  $m - 1$ , gdzie  $m$  – ilość wszystkich sprzedanych pakietów.

Wobec tego równanie matematyczne przedstawiające kwotę premii będzie miało postać:

$$f(m) = \begin{cases} 0, & m = 1 \\ f(m - 1) + m - 1, & m > 1 \end{cases}$$

Kod samej funkcji  $f(m)$  umieszczony w pewnej klasie w języku C# ma postać:

```
public static int Premia(int m)
{
    if (m == 1) return 0;
    else return Premia(m - 1) + m - 1;
}
```

Naszym zadaniem będzie wyznaczenie złożoności obliczeniowej równania  $f(m)$  (chodzi o złożoność w zakresie liczby wywołań funkcji) – mamy na myśli złożoność algorytmu w zakresie liczby wywołań funkcji. Analizując wspomnianą złożoność, można tak naprawdę skorzystać ze wzoru funkcji  $f(m)$ , ponieważ rzad złożoności wyjdzie taki sam, jak w przypadku analizy metody Premia.

Niech zatem  $m > 1$ . Rozpisując lewą stronę równania, otrzymujemy wyrażenie:

$$\begin{aligned} f(m) &= f(m - 1) + m - 1 = f(m - 2) + m - 1 - 1 + m - 1 = \\ &= f(m - 2) + (m - 2) + (m - 1) = f(m - 3) + (m - 3) + (m - 2) + (m - 1) \end{aligned}$$

Podczas kolejnych przejść do poprzednich argumentów funkcji dochodzimy do kroku:  $m - k = 1$ , czyli  $k = m - 1$ ,  $k \in N$ :

$$\begin{aligned} f(m - k) + km - \sum_{i=1}^{m-1} i &= 0 + (m - 1)m - \frac{m(m - 1)}{2} = \\ &= m^2 - m - \frac{m^2 - m}{2} = \frac{m^2 - m}{2} \end{aligned}$$

Analogicznie jak w poprzednim przykładzie, będziemy dowodzili poprawności postępowania indukcyjnie, pokazując dzięki temu, że poniższe rozwiązanie jest poprawne:

$$\frac{m^2 - m}{2}$$

- Dla  $m = 1$  mamy  $f(m) = 0$ .
- Założymy, że równanie jest poprawnie rozwiązane dla każdego  $m \in N$ . Wykazać należy, że dla każdego  $(m + 1) \in N$ . Wobec tego:

$$\begin{aligned} f(m + 1) &= \frac{(m + 1)^2 - (m + 1)}{2} = \frac{m^2 + 2m + 1 - m - 1}{2} = \\ &= \frac{m^2 - m}{2} + m = f(m) + m \end{aligned}$$

Z drugiej strony wstawiając do postaci rekurencyjnej  $m + 1$  za  $m$  mamy:

$$f(m + 1 - 1) + m + 1 - 1 = f(m) - m$$

co należało wykazać, zatem rzad złożoności funkcji  $\Theta(f(m)) = \Theta(m^2)$ .

### Przykład 2.3

Naszym zadaniem będzie wyznaczyć złożoność obliczeniową czasową funkcji danej wzorem:

$$c(n) = \begin{cases} 1, & n = 1 \\ c\left(\frac{n}{3}\right) + n, & n > 1 \end{cases}$$

Kod programu realizującego obliczenie wartości funkcji  $c$  w języku C# ma postać:

```
namespace KlasaTestowa
{
    class Program
    {
        public static void Main(string[] args)
        {
            int n = 16;
            Console.WriteLine(TestowaFunkcja(n));
            Thread.Sleep(4000);
        }
        public static int TestowaFunkcja(int n)
        {
            if (n == 1) return 1;
            else return TestowaFunkcja(n / 3) + n;
        }
    }
}
```

W celu wyznaczenia złożoności obliczeniowej funkcji  $f$  w zależności od rozmiaru danych  $n$  będziemy korzystać z jej wzoru matema-

tycznego. Funkcja  $f$  jest tak skonstruowana, że liczba, która jest jej argumentem, musi być podzielna przez 3, zatem  $n = 3^k$ ,  $k$  – liczna naturalna. Wobec tego:

$$c(n) = c\left(\frac{n}{3}\right) + n = c\left(\frac{n}{9}\right) + \frac{n}{3} + n = c\left(\frac{n}{27}\right) + \frac{n}{9} + \frac{n}{3} + n = \dots$$

Łatwo zauważać zależność taką, że:

$$c\left(\frac{n}{27}\right) + \frac{n}{9} + \frac{n}{3} + n = c\left(\frac{n}{27}\right) + \frac{n}{9} + \frac{n}{3} + \frac{n}{1} = c\left(\frac{n}{3^3}\right) + \frac{n}{3^2} + \frac{n}{3^1} + \frac{n}{3^0}$$

Idąc dalej, otrzymujemy równanie:

$$c(n) = c\left(\frac{n}{3^k}\right) + \frac{n}{3^{k-1}} + \dots + \frac{n}{9} + \frac{n}{3} + \frac{n}{1}$$

$k \in N$  lub  $k = 0$ . Ponieważ:

$$c\left(\frac{n}{3^k}\right) = c\left(\frac{n}{n}\right) = 1$$

a wyrażenia:

$$\frac{n}{3^{k-1}}, \dots, \frac{n}{9}, \frac{n}{3}, \frac{n}{1}$$

tworzą skończony  $k$ -elementowy ciąg geometryczny o ilorazie:

$$q = \frac{1}{3}$$

to ich suma jest równa:

$$S_k = \sum_{i=0}^{k-1} \frac{n}{3^i} = n \cdot \frac{1 - \left(\frac{1}{3}\right)^k}{1 - \frac{1}{3}}$$

gdzie  $k = \log_3(n)$ , oraz:

$$S_k = n \cdot \frac{1 - \left(\frac{1}{3}\right)^{\log_3(n)}}{1 - \frac{1}{3}}$$

Z własności logarytmów otrzymujemy:

$$\begin{aligned} S_k &= n \cdot \frac{1 - (3^{-1})^{\log_3(n)}}{1 - \frac{1}{3}} = n \cdot \frac{1 - (3)^{-\log_3(n)}}{\frac{2}{3}} = \\ &= n \cdot \frac{1 - (3)^{\log_3(n-1)}}{\frac{2}{3}} = n \cdot \frac{1 - \frac{1}{n}}{\frac{2}{3}} = \frac{3}{2} \cdot (n-1) \end{aligned}$$

Ostatecznie jawni wzór prezentujący złożoność obliczeniową czasową w zależności od rozmiaru danych  $n$  ma postać:

$$c(n) = 1 + \frac{3}{2} \cdot (n-1) \quad (*)$$

Dowód poprawności rozwiązania:

1. Niech  $n = 1$ , wtedy:

$$c(n) = 1 + \frac{3}{2} \cdot (1-1) = 1 + 0 = 1$$

2. Zakładamy prawdziwość wzoru (\*). Sprawdzamy, czy  $c(3n) = c(n) + 3n$ . Podstawiając argument  $3n$  do równania, mamy:

$$\begin{aligned} c(3n) &= 1 + \frac{3}{2}(3n-1) = 1 + \frac{9}{2}n - \frac{3}{2} = 1 + \frac{3}{2}n + \frac{6}{2}n - \frac{3}{2} = \\ &= 1 + \frac{3}{2}n - \frac{3}{2} + \frac{6}{2}n = 1 + \frac{3}{2}(n-1) + 3n = c(n) + 3n \end{aligned}$$

Zgodnie z zasadą indukcji matematycznej wzór (\*) jest prawdziwy dla każdej liczby naturalnej  $n$ , co należało udowodnić, zatem  $\Theta(c(n)) = \Theta(n)$ . Po dalszej analizie przykładów 2.1 oraz 2.3 możemy wyciągnąć następujące wnioski:

**Wniosek 2.1.** Jeżeli metoda rekurencyjna da się opisać ogólnym wzorem postaci:

$$g(n) = \begin{cases} a, & n = 1 \\ g\left(\frac{n}{p}\right) + b, & n > 1 \end{cases}$$

to rzad jej złożoności obliczeniowej w zależności od rozmiaru danych  $n$  jest równy  $\Theta(\log_p(n))$ . Jeżeli natomiast metoda rekurencyjna da się opisać ogólnym wzorem postaci:

$$h(n) = \begin{cases} a, & n = 1 \\ h\left(\frac{n}{p}\right) + n, & n > 1 \end{cases}$$

to rzad jej złożoności w zależności od rozmiaru danych  $n$  jest równy  $\Theta(n)$ , gdzie  $a, b, p = \text{const}$ ,  $a, b, p > 0$  oraz  $a, b, p$  są liczbami naturalnymi.

**Dowód.** Wykażemy teraz dowód dla funkcji  $g$ . Przyjmujemy założenie, że zmienna  $n$  jest podzielna przez liczbę  $p$ , oczywiście  $p^k = n$ , gdzie  $k \in N$  lub  $k = 0$ . Zatem:

$$g(n) = g\left(\frac{n}{p}\right) + b = g\left(\frac{n}{p^2}\right) + b + b = g\left(\frac{n}{p^3}\right) + 3 \cdot b$$

Idąc dalej, otrzymamy równanie:

$$g(n) = g\left(\frac{n}{p^k}\right) + k \cdot b = g\left(\frac{n}{n}\right) + k \cdot b = a + k \cdot b = a + b \cdot \log_p(n)$$

Indukcyjny dowód poprawności rozwiązania jest analogiczny jak ten z przykładu 2.2, więc go pomijamy. Wykazanie, że funkcja  $h$  ma liniowy rzad złożoności obliczeniowej, można przeprowadzić analogicznie do dowodu pokazanego w przykładzie 2.3.

**Wniosek 2.2.** Jeżeli metoda rekurencyjna da się opisać ogólnym wzorem postaci:

$$h(n) = \begin{cases} a, & n = 1 \\ h\left(\frac{n}{p}\right) + n^s, & n > 1 \end{cases}$$

to rzad jej złożoności w zależności od rozmiaru danych  $n$  jest równy  $\Theta(n^s)$ , gdzie  $a, b, s = \text{const}$ ,  $a, b, s > 0$  oraz  $a, b, s$  są liczbami naturalnymi.

**Dowód.** Zakładamy, że zmienna  $n$  jest podzielna przez  $p$ , zatem  $p^k = n$ , gdzie  $k \in N$  lub  $k = 0$ . Rozpisując wzór poprzez cofnięcie do poprzednich argumentów, otrzymujemy równanie:

$$\begin{aligned}
h(n) &= h\left(\frac{n}{p}\right) + n^s = h\left(\frac{n}{p^2}\right) + \left(\frac{n}{p}\right)^s + n^s = \\
&h\left(\frac{n}{p^3}\right) + \cdots + \left(\frac{n}{p^2}\right)^s + \left(\frac{n}{p}\right)^s + n^s = \\
&= h\left(\frac{n}{p^4}\right) + \left(\frac{n}{p^3}\right)^s + \left(\frac{n}{p^2}\right)^s + \left(\frac{n}{p^1}\right)^s + \left(\frac{n}{p^0}\right)^s = \\
&= h\left(\frac{n}{p^k}\right) + h\left(\frac{n}{p^{k-1}}\right) + \cdots + \left(\frac{n}{p^3}\right)^s + \left(\frac{n}{p^2}\right)^s + \left(\frac{n}{p^1}\right)^s + \left(\frac{n}{p^0}\right)^s = \\
&= a + \sum_{i=0}^{k-1} \left(\frac{n}{p^i}\right)^s = a + n^s \cdot \sum_{i=0}^{k-1} \left(\frac{1}{p^i}\right)^s = \\
&= a + n^s \cdot \frac{1 - \left[\left(\frac{1}{p}\right)^s\right]^k}{1 - \left(\frac{1}{p}\right)^s} = a + n^s \cdot \frac{1 - \left[\left(\frac{1}{p}\right)^s\right]^{\log_p(n)}}{1 - \left(\frac{1}{p}\right)^s} = \\
&= a + n^s \cdot \frac{1 - \left(\frac{1}{p}\right)^{s \cdot \log_p(n)}}{\frac{p^s}{p^s} - \frac{1}{p^s}} = a + n^s \cdot \frac{1 - (p^{-1})^{s \cdot \log_p(n)}}{\frac{p^s - 1}{p^s}} = \\
&= a + n^s \cdot \frac{p^s}{p^s - 1} (1 - p^{\log_p(n-s)}) = \\
&= a + n^s \cdot \frac{p^s}{p^s - 1} \left(1 - \frac{1^s}{n^s}\right) = \\
&= a + n^s \cdot \frac{p^s}{p^s - 1} \cdot \left(\frac{n^s - 1}{n^s}\right) = a + \frac{p^s}{p^s - 1} (n^s - 1)
\end{aligned}$$

czyli:

$$h(n) = a + \frac{p^s}{p^s - 1} (n^s - 1)$$

Po wyznaczeniu wzoru należy przeprowadzić dowód indukcyjny jego poprawności:

1. Niech  $n = 1$ . Wtedy:

$$h(1) = a + \frac{p^s}{p^s - 1} (1 - 1) = a + 0 = a$$

2. Wykażemy, że rozwiązanie jest poprawne dla argumentu równego  $p \cdot n$ . Wobec tego należy obliczyć wyrażenie:

$$\begin{aligned}
h(pn) &= a + \frac{p^s}{p^s - 1} ((pn)^s - 1) = a + \frac{p^s (pn)^s}{p^s - 1} - \frac{p^s}{p^s - 1} = \\
&= a + \frac{[(p^s - 1) + 1](pn)^s}{p^s - 1} - \frac{p^s}{p^s - 1} = \\
&= a + \frac{p^s n^s}{p^s - 1} + \frac{p^s n^s (p^s - 1)}{p^s - 1} - \frac{p^s}{p^s - 1} = \\
&= a + \frac{p^s n^s}{p^s - 1} + p^s n^s - \frac{p^s}{p^s - 1} = a + \frac{p^s n^s}{p^s - 1} - \frac{p^s}{p^s - 1} + p^s n^s = \\
&= a + \frac{p^s}{p^s - 1} (n^s - 1) + p^s n^s = h(n) + p^s n^s, \\
&= a + \frac{p^s n^s}{p^s - 1} + p^s n^s - \frac{p^s}{p^s - 1} = a + \frac{p^s n^s}{p^s - 1} - \frac{p^s}{p^s - 1} + p^s n^s = \\
&= a + \frac{p^s}{p^s - 1} (n^s - 1) + p^s n^s = h(n) + p^s n^s,
\end{aligned}$$

co należało wykazać, zatem  $\Theta(h(n)) = \Theta(n^s)$ .

Jak wynika z rozwiązania równań w wyżej pokazanych przykładach oraz wnioskach, szeregi liczbowe są bardzo dobrym oraz pomocnym narzędziem w analizie złożoności obliczeniowej algorytmów rekurencyjnych. Możemy postawić pytanie, co by było, gdy szereg liczbowy, ilustrujący czas działania algorytmu w zależności

reklama



# devstyle.pl

ŚWIAT OKIEM PROGRAMISTY

od danych wejściowych, nie będzie spełniał żadnego z kryteriów zbieżności albo będzie nieskończony? Odpowiedź na to pytanie jest taka, że rozmiar danych wejściowych nie jest nieskończony, a w takim przypadku suma szeregu liczbowego nie jest nieskończonością, czyli jest jakąś liczbą. Innym mankamentem jest czas działania algorytmu – dlatego należy ostrożnie postępować z rekurencją.

**Przykład 2.4.** Ostatnim przykładem będzie analiza złożoności obliczeniowej algorytmu QuickSort (szybkie sortowanie). Kod takiego algorytmu zaimplementowany w języku C# ma postać:

```
namespace QuickSort
{
    class QuickSort
    {
        public static void QuickSort(int[] array, int left, int right)
        {
            var i = left;
            var j = right;
            var pivot = array[(left + right) / 2];
            while (i < j)
            {
                while (array[i] < pivot) i++;
                while (array[j] > pivot) j--;
                if (i <= j)
                {
                    var tmp = array[i];
                    array[i++] = array[j];
                    array[j--] = tmp;
                }
            }
            if (left < j) QuickSort(array, left, j);
            if (i < right) QuickSort(array, i, right);
        }
        static void Main(string[] args)
        {
            var rand = new Random();
            var array = new int[100];
            for (int i = 0; i < array.Length; i++)
            {
                array[i] = rand.Next(1000);
            }
            Console.WriteLine("Before: ");
            Console.WriteLine(string.Join(" ", array));
            QuickSort(array, 0, array.Length - 1);
            Console.WriteLine("After: ");
            Console.WriteLine(string.Join(" ", array));
            Console.ReadLine();
        }
    }
}
```

Czas działania algorytmu sortowania szybkiego oraz zapotrzebowanie na pamięć są uzależnione od postaci tablicy wejściowej. Od tego zależy, czy podziały dokonywane w algorytmie są zrównoważone, czy też nie, a to z kolei zależy od wybranego klucza podziału. W przypadku gdy podziały są zrównoważone, algorytm jest równie szybki jak np. sortowanie przez scalanie. Gdy natomiast podziały są niezrównoważone, sortowanie może przebiegać asymptotycznie tak wolno, jak sortowanie przez wstawianie. Złożoność algorytmu QuickSort możemy rozpatrywać, analizując trzy możliwości uzależnione od danych wejściowych algorytmu:

- a) przypadek optymistyczny,
- b) przypadek przeciętny,
- c) przypadek pesymistyczny.

Z wersją optymistyczną mamy do czynienia, gdy każdy podział daje równe podzbiory danych. Wówczas złożoność obliczeniowa jest rzędu  $\Theta(\log(n))$ . Przypadek pesymistyczny wystąpi wtedy, gdy za każdym razem będzie wybierany element największy albo najmniejszy w sortowanym fragmencie tablicy, a wtedy złożoność obliczeniowa jest rzędu  $\Theta(n^2)$ . Najciekawszy jest jednak przypadek przeciętny. W takim przypadku podziały elementu najmniejszego bądź największego, w sortowanym fragmencie tablicy, wypadają w połowie. Dla tak równomiernego rozkładu prawdopodobieństwa wyboru elementu, względem którego dokonujemy podziału, algorytm sortowania szybkiego wymaga czasu działania  $\Theta(n \log(n))$ . Wzór rekurencyjny takiego właśnie przypadku ma postać [6]:

$$f(n) = \begin{cases} n + 1 + \frac{1}{n} \sum_{k=1}^n (f(n-k) + f(k-1)), & n \geq 2 \\ 0, & n \in \{0, 1\} \end{cases}$$

Rozwiązywanie tego równania (dość uproszczone) znajduje się w pozycji [6], dlatego nie będziemy go przedstawiać w artykule. Algorytm ten jest o tyle ciekawy, że należy rozpatrywać 3 przypadki złożoności obliczeniowej podczas analizy, a nie tylko 2, tak jak w przypadku wielu innych algorytmów rekurencyjnych.

## PODSUMOWANIE

Głównym celem artykułu było przedstawienie sposobu analizy złożoności obliczeniowej algorytmów rekurencyjnych poprzez cofanie do momentu, aż osoba wykonująca analizę dojdzie do warku początkowego. Jak wynika z artykułu, taka analiza nie jest sprawą trywialną, ponieważ wymaga umiejętności rozwiązywania równań oraz stosowania szeregiem liczbowych (w tym głównie geometrycznych). Artykuł został również napisany jako uzupełnienie do artykułu „Wyznaczanie złożoności obliczeniowej algorytmów” z poprzedniego numeru, który przedstawiał sposób analizy złożoności obliczeniowej dla algorytmów iteracyjnych.

## BIBLIOGRAFIA

- [1] [https://pl.wikipedia.org/wiki/Indukcja\\_matematyczna](https://pl.wikipedia.org/wiki/Indukcja_matematyczna)
- [2] <http://testerzy.pl/baza-wiedzy/testy-jednostkowe-metody>
- [3] <https://pl.wikipedia.org/wiki/Rekurencja>
- [4] <https://goo.gl/DC95NQ>
- [5] <https://goo.gl/xvzXSM>
- [6] [toyoteczka.mainrc.com/pliki/asd/quicksort.pdf](http://tojoteczka.mainrc.com/pliki/asd/quicksort.pdf)



**MAREK ŻUKOWICZ**

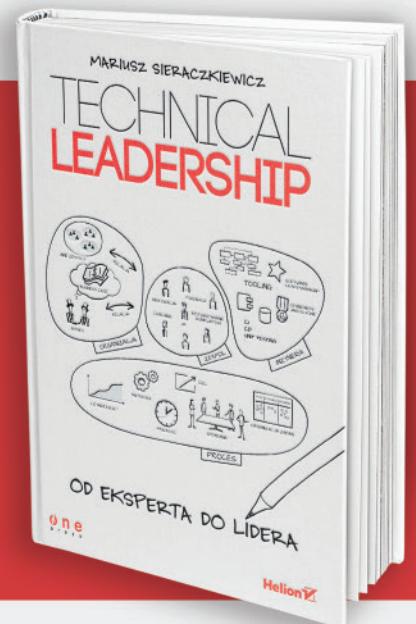
[bobmarek@o2.pl](mailto:bobmarek@o2.pl)

Absolwent matematyki na Uniwersytecie Rzeszowskim. Obecnie pracuje jako tester oprogramowania. Jego zainteresowania skupiają się wokół obszarów testowania oprogramowania, zapewnienia jakości oprogramowania. Interesuje się głównie zastosowaniem modeli matematycznych w procesie testowania oprogramowania. Pisze publikacje naukowe, artykuły popularno-naukowe z tematyki testowania oraz zapewnienia jakości. Hobby: muzyka lat 80-tych.

# BNS IT - SZKOLENIA OTWARTE

WARSZAWA / 08-10.01.2018  
**TECHNICAL LEADERSHIP™**  
ROLA LIDERA TECHNICZNEGO

1. Rola lidera technicznego
2. Motywacja własna i innych
3. Ludzie
4. Zespół
5. Kompetencje lidera



WARSZAWA / 14-16.02.2018  
**WZORCE PROJEKTOWE**  
I REFAKTORYZACJA DO WZORCÓW

1. Wprowadzenie do wzorców projektowych
2. Jakość kodu źródłowego
3. Refaktoryzacja
4. Wzorce GoF

WARSZAWA / 21-23.02.2018  
**NOWOCZESNE**  
**ARCHITEKTURY APLIKACJI**

1. Wprowadzenie do pojęcia architektura oprogramowania
2. Domain-Driven Design
3. Micro Services
4. Ports & Adapters
5. Clean and Onion Architecture
6. Reactive Architecture
7. Serverless Architecture

## P O Z O S T A Ł E   T E R M I N Y :

Nowoczesne architektury aplikacji	Warszawa	11-13.07.2018	2100,00 PLN
Technical Leadership™	Warszawa	18-20.04.2018	2100,00 PLN
Technical Leadership™	Warszawa	04-07.07.2018	2100,00 PLN
Tworzenie Microservices z użyciem Spring Boot	Warszawa	28-30.04.2018	2100,00 PLN
Wzorce projektowe i refaktoryzacja do wzorców	Warszawa	16-18.05.2018	2100,00 PLN CENY NETTO

# O zaletach prototypów, czyli zwinny sposób na dokumentację

Prototypowanie to szybki i prosty sposób na zastąpienie części dokumentacji dotyczącej wymagań funkcjonalnych, wzbogacony o projektowanie doświadczeń użytkownika końcowego od samego początku prac.

## DOKUMENTACJA TO SAME PROBLEMY, PO CO WIĘC JĄ TWORZYĆ?

Odpowiedź jest prosta. Jest niezbędną i nie da się jej całkowicie uniknąć. Zanim w ogóle projekt wystartuje, dokumentacją są wszystkie spisane wymagania funkcjonalne i niefunkcjonalne, będące najczęściej też załącznikiem do umowy i podstawą do rozliczenia projektu po jego zakończeniu. Następnie dokumentacją stają się wszystkie raporty z postępu prac, statusy, opisy wprowadzanych zmian, podsumowania testów, nie mówiąc o dokumentacji technicznej tworzonego oprogramowania. Stos papierów rośnie. W dużych organizacjach czy bardzo rozbudowanych projektach nierzadko zdarza się, że samo wytwarzanie dokumentacji oraz aktualizowanie już istniejącej zajmuje więcej czasu niż realna praca, której skutkiem jest produkt zamówiony przez klienta. Mimo tego – jest niezbędną.

Jak więc sobie poradzić z dokumentacją? W przypadku projektów, których celem jest wytworzenie oprogramowania, możliwe jest zastąpienie części dokumentacji w taki sposób, by było łatwiej zapoznać się z wymaganiami klienta, przy równoczesnym ułatwieniu walidacji tych wymagań w czasie, gdy zmiany nie powodują dużych nakładów pracy grafików czy programistów. Mowa o prototypowaniu.

## CZYM JEST PROTOTYPOWANIE

Mając na względzie wytwarzanie oprogramowania, posłużymy się przykładem. Zadanie, jakie stoi przed zespołem grafików i programistów, to stworzenie strony internetowej z prenumeratą magazynu „Programista”, na której znajduje się informacja o produkcji i jego cenie, kontakt z redakcją, jest także możliwy zakup wraz z płatnością internetową. Strona wydaje się być prosta.

W tradycyjnie pojmowanej dokumentacji wymagań dla takiej strony znalazłyby się obszerne opisy, obejmujące:

- a. wymagania dotyczące szaty graficznej (w nich często już uwzględnia się nie tylko same wytyczne dotyczące projektu graficznego, ale też wymagania związane z UX – *user experience* – które można też uznać za „użyteczność” i umieszczać w wymaganiach niefunkcjonalnych).
- b. wymagania funkcjonalne (czyli: co będzie umożliwiała strona).
- c. wymagania niefunkcjonalne (czyli: jak będzie działała strona, tu uwzględniamy wydajność, kompatybilność, bezpieczeństwo, niezawodność, łatwość utrzymania itp.).

Przygotowanie wymagań dla wyżej wymienionej strony internetowej z podziałem na a), b) i c) to długie godziny pracy i jeszcze więcej godzin potrzebnych później na przeczytanie (ze zrozumieniem) wielu stron tekstu. Co więcej, każdy uczestnik projektu

może mieć inną wizję i standardowa, opisowa dokumentacja tego problemu nie rozwiązuje (zwłaszcza jeśli kontakt z klientem jest utrudniony i nie ma możliwości, by wspólnie i na bieżąco wyjaśniać wątpliwości). A tak naprawdę, widząc powyższe zadanie, każdy ma już w swojej głowie pewien obraz tego, jak taka strona mogłaby wyglądać, jaki byłby mniej więcej układ elementów, jak działałby zakup produktu itd. Wystarczy stworzyć taki jeden, wspólny dla wszystkich uczestników projektu (w tym klienta) obraz. To jest właśnie prototyp.

Prototyp odzwierciedla większość wymagań w postaci wstępnej wizji strony internetowej, jaka ma powstać. W prototypie zwrzemy już architekturę informacji (takie zorganizowanie treści, by dostęp do nich był logiczny i łatwy dla przyszłego użytkownika), zastanowimy się nad tym, jak zaprezentować i komunikować produkt, uwzględnimy nawigację, przemyślimy układ elementów interfejsu. Wybierzemy priorytety, dookreślając funkcjonalności, które będą kluczowe, a które tylko dodatkiem; ważnym, ale jednak nie najważniejszym w procesie na przykład zakupu produktu. Prototyp uświadomi nam też, o czym zapomnieliśmy albo w ogóle nie pomyśleliśmy. Dlatego jest to tak dobre rozwiązanie już na samym początku prac nad projektem – uświadamia braki, których dodanie na późniejszym etapie mogłoby być już czasochłonne i kosztowne.

## CZTERY ELEMENTY, W KTÓRYCH PROTOTYP WYGRYWA!

**Objetość.** Najczęściej im bardziej rozbudowany projekt, tym więcej stron zajmują objaśnienia wymaganych funkcjonalności. Dzięki prototypowi wszyscy uczestnicy projektu mogą zobaczyć, jak to będzie wyglądać czy działać, bez konieczności analizowania długich opisów.

**Dosłowność.** W przypadku tworzenia oprogramowania jest bardzo duże ryzyko złej interpretacji dokumentacji. Tu pojawia się zwykle kwestia zgadywania „co autor miał na myśli?”, co może prowadzić do klęski projektu, jeśli klient jest słabo dostępny i trudno z nim wyjaśnić wątpliwości na bieżąco. Prototyp sprawia, że wszystkie takie niejasności muszą być wyjaśnione na bardzo wczesnym etapie.

**Forma.** Jeśli się nad tym zastanowić, czy ma sens tworzenie szczegółowego opisu tego, jak menu na stronie powinno się rozwinać po najechaniu kursorem? W prototypie możemy od razu zaprezentować rozwiązanie, jakie proponujemy, i zderzyć je z wymaganiami klienta.

**Aktualność.** Jak często pierwsza wersja dokumentacji jest aktualizowana? Z reguły nie jest to priorytet, gdy prace już trwają. Wszelkie zmiany, uzgodnione w trakcie prac, zwykle są więc rozrzucone w korespondencji czy w notatkach ze spotkań. Nie jest to oczywiście reguła, ale jeśli tak jest – może to być utrudnienie w momencie przekazania do odbioru zakończonego oprogramowania. Prototyp

ułatwia szybką aktualizację oraz akceptację naniesionych zmian (łatwiej jest „przekliknąć się” przez prototyp i sprawdzić modyfikacje, niż analizować opisy tychże).

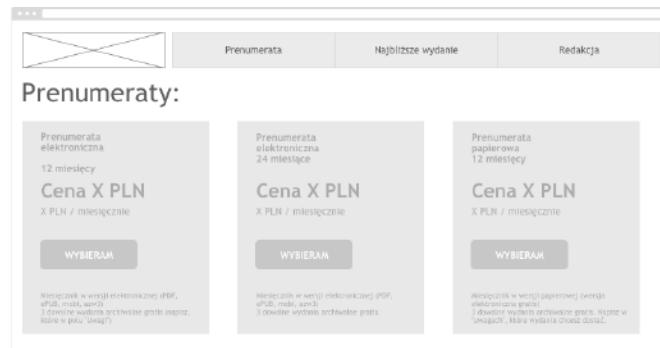
## JAKI PROTOTYP WYBRAĆ? RODZAJE

Zależnie od tego, o jaki poziom szczegółowości i skomplikowania prototypu nam chodzi, wybierzemy jeden z poniższych ich rodzajów:

### Prototyp lo-fi (*low fidelity*, mała dokładność)

Przygotowanie takiego prototypu jest szybkie i nisko kosztowe. Wymaga małego, albo nawet zerowego, nakładu pracy grafików i programistów, bowiem najczęściej składa się ze szkiców – schematów ekranów danej strony (tak zwanych *wireframes*). Mogą być one przygotowane w formie statycznej (prezentowane osobno, poukładane tylko w logiczny ciąg) albo w wersji klikalnej (umożliwiające pokazanie interakcji w taki sposób, jak docelowy).

Prototyp lo-fi często jest tworzony „na szybko”. Celem jest zwizualizowanie w bardzo prosty sposób ścieżki naszego użytkownika, zaaranżowanie wszystkich niezbędnych elementów w tym procesie. Wykonując zadanie, o którym była mowa powyżej, prototyp lo-fi mógłby wyglądać tak jak na Rysunku 1.



Rysunek 1. Przykład prototypu lo-fi

Patrząc na prototyp, zarówno klient, jak i graficy oraz programiści wiedzą, jak to będzie działać, a nie jak to będzie wyglądać, czyli:

- » jakie funkcjonalności będą dostępne,
- » jakie są priorytety poszczególnych elementów i zasady ich pokazywania,
- » jaka jest struktura informacji,
- » jaka jest podstawowa, główna ścieżka użytkownika,
- » jakie są możliwe przypadki skrajne w ścieżce użytkownika.

Opcjonalnie: możemy pokazać wczesny design, jednak z zaznaczeniem, że warstwa wizualna będzie dopracowana później.

Warto też zauważyć, że już na tym etapie powinno się zrezygnować z popularnego zastępowania realnych tekstów abstrakcyjnym *lorem ipsum*. Dzięki temu członkowie zespołu lepiej rozumieją to, co tworzą. Łatwiej też wczuć się w perspektywę użytkownika końcowego – jeśli bowiem już w tym momencie trudno zrozumieć, o co na danej stronie chodzi, tym bardziej nie zrozumie tego przyszły konsument.

Brzmi skomplikowanie? Warto wiedzieć, że prototypowanie lo-fi takie nie jest, wręcz przeciwnie. Tak nazywa się już szkic na kartce papieru (lo-fi pen and paper). *Wireframes* to nic innego, jak schematyczne szkice, które można stworzyć w prezentacji *power point*. Oczywiście wygodniej będzie przygotować je już w programie, który umożliwia dalszą ewolucję takiego prototypu – np. od wersji statycznej, poprzez dodanie interakcji, dodanie szczegółów, finalnych grafik. W ten sposób prototyp z lo-fi zmieni się w...

### Prototyp hi-fi (*high fidelity*, duża dokładność)

Celem takiego prototypu jest zaprezentowanie grafiki, by zebrać jak najszybszy komentarz do warstwy wizualnej i odczuć związań z jej odbiorem. Tak zwane *mockupy* są tworzone przez grafików i jeśli mają formę statycznych, osobno przygotowanych reprezentacyjnych ekranów strony, służą tylko wyżej wymienionemu celowi. Dzięki temu cały zespół wie, jak to będzie wyglądać. Realizując przykładowe zadanie, jakie zostało sformułowane w tym artykule, prototyp hi-fi mógłby wyglądać tak jak na Rysunku 2.

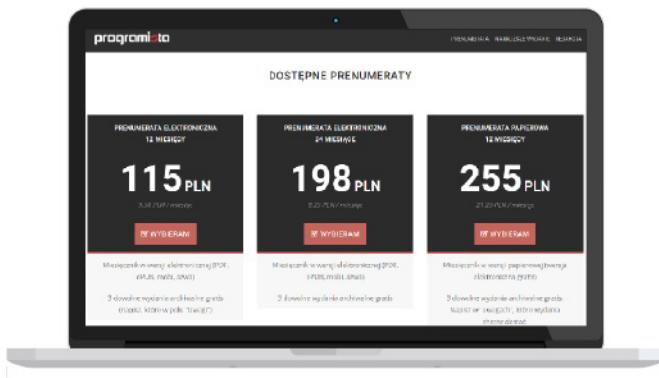
reklama



Obserwuj  
nas  
na Twitterze



@PROGRAMISTAMAG



Rysunek 2. Przykład prototypu hi-fi

Jest też możliwość przygotowania klikalnego prototypu, wykorzystującego *mockupy* z zachowaniem całej struktury opisanej przy okazji prototypu lo-fi. Dzięki takiemu podejściu cały zespół rozumie, jak strona będzie wyglądać i działać. Warto to rozróżnić.

## KTO TWORZY PROTOTYPY

To, co jest bardzo istotne w przypadku tworzenia prototypów, to fakt, że najczęściej nie są wymagane specjalistyczne kompetencje graficzne czy programistyczne. Wymagana jest zdolność myślenia :) A także wiele umiejętności miękkich: trzeba umieć słuchać wymagań klienta i zrozumieć, które są kluczowe, a które są poboczne. Trzeba też rozumieć docelowego użytkownika, by wiedzieć, jak ułożyć ścieżkę oraz elementy na stronie, żeby używanie tejże było zarówno proste, logiczne, jak i przyjemne. Przyda się umiejętność współpracy z całym zespołem grafików i programistów, gdyż prototyp nie istnieje w próżni, powinien być stale aktualizowany, a gdy pojawią się pytania, trzeba na nie umieć odpowiedzieć, biorąc pod uwagę wszystkich wyżej wymienionych zainteresowanych.

Obecnie prototypowanie coraz częściej wchodzi w zakres projektanta UX (*user experience designera*). Jest to z pewnością bardzo ciekawa praca. Osoba tworząca prototyp sprawia, że cały projekt przestaje być abstrakcyjny. Widzi też reakcje otoczenia i może bardzo szybko wprowadzać zmiany, mając dzięki temu realny wpływ na powstający produkt w bardzo krótkim czasie.

## SŁOWO O METODYKACH ZWINNYCH

Zaletą tworzenia prototypów jest ich duża elastyczność. Dzięki temu bardzo mocno wpisują się w zwinne podejście do tworzenia oprogramowania, zakładające iteracyjność (co niestety nie jest

dostępne dla wszystkich branż). Od samego początku tworzone jest coś, co jest już gotowe do testów. Umożliwia to bardzo szybką weryfikację zarówno z klientem, jak i użytkownikiem docelowym. Co to daje? Na przykład zminimalizowanie kosztów, jeśli okaże się, że pewne funkcjonalności są zupełnie zbędne albo źle przemyślane. O wiele łatwiej jest wprowadzać zmiany na wcześniejszych etapach tworzenia oprogramowania, niż gdy jest już wszystko gotowe do startu (łącznie z wykupionymi mediani do promocji strony i produktu).

## PODSUMOWANIE, CZYLI JAK SIĘ MA PROTOTYPOWANIE DO DOKUMENTACJI

Prototypowanie to całkiem ciekawy sposób zastąpienia dokumentacji o tradycyjnej, opisowej formie poprzez zobrazowanie części wymagań w formie wizualnej. Zależnie od poziomu dokładności prototyp pełni różne funkcje:

- » prezentuje podstawowy koncept strony, niezbędne funkcjonalności i podstawową ścieżkę użytkownika.

**Wady:** mało szczegółów, możliwe uwagi do niedopracowanych elementów.

**Zalety:** bardzo szybki i tani sposób na zebranie komentarza od klienta i potwierdzenie, że cały zespół rozumie, co jest do zrobienia.

- » prezentuje finalny koncept strony, z ostateczną oprawą graficzną, komunikacją i pełnym spektrum funkcjonalności.

**Wady:** dużo szczegółów, większy nakład pracy i dłuższy czas przygotowania.

**Zalety:** umożliwia zebranie uwag i wprowadzenie ich na etapie, gdy ich wdrożenie nadal kosztuje mniej, niż zmiana gotowej strony.

Prototyp w każdej z powyższych wersji może już skutecznie zastąpić bardzo dużą część dokumentacji opisowej. Jednak nie wyeliminuje jej całkowicie. Trudno w formie prototypu zatrzymać wymagania niefunkcjonalne, dotyczące na przykład bezpieczeństwa danych użytkowników strony, niezawodności i wydajności strony, dostosowania do działań SEO, analizy konwersji itd.

Warto jednak wiedzieć, że wymagania funkcjonalne, zwłaszcza te bezpośrednio związane z produktem czy użytkownikiem, można zatrzymać za pomocą prototypu, unikając tym samym formy opisowej. Dzięki temu możliwe jest zastąpienie dużej części zbędnych, jak się okazuje, opisów na rzecz formy bardziej strawnej dla zespołu.



### KATARZYNA MAŁECKA

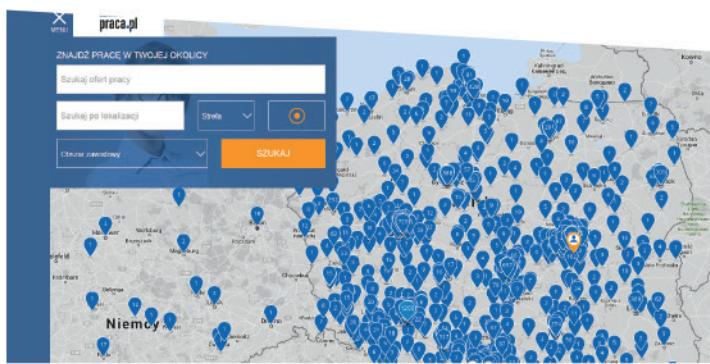
katarzyna@teina.co

Autorka od 9 lat odpowiada za prowadzenie projektów IT, posiada także praktyczne doświadczenie z zakresu UX: tworzenia logiki userflows, prototypowania (makiety lo-fi) oraz prowadzenia badań jakościowych w formie indywidualnych wywiadów, przy użyciu eye-trackera oraz EEG. Zawodowo: właścicielka agencji badawczej. Prywatnie: autorka bajek dla dzieci.



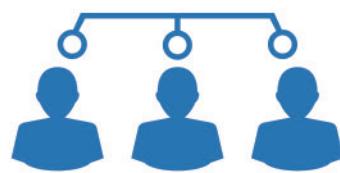
Chcesz dobrze zarobić?

Na Praca.pl codziennie znajdziesz ponad 3 000 ofert pracy z obszaru IT i nowe technologie



Znajdź pracę w Twojej okolicy

Lokalna.praca.pl



Poleć znajomego do pracy  
i zgarnij 1 000 zł

[Praca.pl/rekomendacje.html](http://Praca.pl/rekomendacje.html)

# Być jak profesor Moriarty. Praktyczna steganografia

Bezpieczeństwo danych, ataki hakerskie na banki i korporacje, łamanie zabezpieczeń... – z tymi wszystkimi hasłami spotykamy się praktycznie codziennie. Nasze dane, nawet w formie zaszyfrowanej, nie mogą być już uznawane za bezpieczne. W tym artykule przedstawiamy pewne sposoby, które pozwolą na ukrycie informacji w przesyłanych komunikatach czy obrazach.

## „POLOWANIE POD LONDYNEM ROZPOCZĘTE...”

Profesor James Moriarty, wielki przeciwnik Sherlocka Holmesa i geniusz zbrodni. Jednocześnie genialny uczony, który doskonale znał zróżnicowane metodyki ukrywania informacji. Podczas lektury przygód najgroźniejszego rywala profesora, reprezentanta jasnej strony mocy, możemy natknąć się na opowiadanie „The Adventure of the <<Gloria Scott>>”. W opowieści tej pojawia się list, który pozornie składa się z ciągu niepowiązanych ze sobą zdań. Jednakże odczytanie co trzeciego wyrazu nadaje rzeczywistej wiadomości realny sens. Ten list miał postać następującą – „Polowanie pod Londynem rozpoczęte. Główny łowczy Hudson zarządził chyba wszystko. Wyraźnie już powiedział: Będzie wielka oblawa! Dlatego trzeba ratować bażancich samic życie.” Jest to jeden z koronnych przykładów ukrywania informacji. Dzisiaj naukę o takiej komunikacji, w której ukryta jest obecność innego komunikatu, nazywamy steganografią. Możemy ją zaobserwować nie tylko w formie tekstu, ale także pod postacią obrazów. Dzisiaj ta gałąź nauki ma jednak pewne solidne wsparcie w stosunku do czasów, w jakich egzystował profesor Moriarty. Otóż jest ona wspierana technologicznie przez programy komputerowe, które bardzo szybko potrafią przygotować taką wiadomość (czy też obraz), która zawiera dodatkową, niewidoczną gołym okiem informację. W niniejszym artykule odniesiemy się do zróżnicowanych metod, które pozwalają na przygotowanie takich komunikatów.

## STEGANOGRAFIA I JEJ HISTORIA

Na początku tego artykułu pojawiła się nazwa „steganografia”. Co zatem oznacza to słowo? Czego dotyczy? Odnosząc się do jego etymologii, musimy przenieść się do Grecji. Otóż dosłowne tłumaczenie oznacza „ukryte pismo”. Jednakże bardziej złożona definicja mówi nam o tym, że jest to nauka, która zajmuje się przesyaniem komunikatów w taki sposób, aby zawierały one dodatkowe, niejawne informacje. W tym miejscu warto byłoby przytoczyć generalną różnicę pomiędzy wspomnianą gałęzią nauki a kryptografią. Otóż w przypadku steganografii negowana jest obecność jakiegokolwiek dodatkowego komunikatu – osoba wysyłająca wiadomość twierdzi (pomimo że jest inaczej), że jest to jawnym komunikatem bez żadnych dodatkowych elementów. Tymczasem kryptografia nie neguje informacji, jakie są przesyłane, tylko wskazuje na to, że są to wiadomości niejawne. Zanim przejdziemy do historycznego aspektu steganografii, warto byłoby wskazać, jaką jest generalna klasyfikacja systemów steganograficznych.

## Klasyfikacja systemów steganograficznych

W ogólnym ujęciu wyróżniamy trzy rodzaje takich systemów – są to systemy steganografii czystej, z kluczem prywatnym oraz z kluczem publicznym.

Pierwszy z wymienionych rodzajów polega na założeniu, że strona atakująca nie zna metodów związanych ze steganografią. Mianowicie ukryta wiadomość stanie się jawną w momencie, w którym znana jest metoda, z użyciem której może zostać ona wydobyta. W tym miejscu jednak należy również wskazać, że ten rodzaj steganografii nie spełnia zasad Kerckhoffsa. Zasada ta mówi nam o tym, że system kryptograficzny (oraz steganograficzny) powinien być bezpieczny nawet wtedy, gdy są znane wszystkie szczegóły jego działania (poza kluczem). Zauważmy zatem, że w tym przypadku, gdy będą nam znane detale działania systemu steganograficznego tego rodzaju, będziemy mogli bez najmniejszych problemów uzyskiwać ukryte wiadomości.

Całkowicie inną strategię możemy zaobserwować w przypadku steganografii z kluczem prywatnym. Analizując zasady tego typu systemu, możemy skonkludować, że mamy do czynienia z metodą, która jest całkowicie jawną, a jej detale są powszechnie znane. Jednakże aby mogła być ona efektywnie używana, strony, które będą brać udział w komunikacji, muszą przed jej rozpoczęciem ustalić między sobą klucz steganograficzny, który będzie używany w trakcie komunikacji. Ta metoda nie jest jednak również wolna od wad. Jednym z przykładów jest problem związany z efektywną i bezpieczną metodą wymiany klucza steganograficznego. Jak również warta rozważenia byłaby strategia, jaka powinna zostać zastosowana w momencie, w którym klucz steganograficzny zostanie złamany.

Ostatnim rodzajem, który jest wymieniany w ogólnej klasyfikacji, jest steganografia z kluczem publicznym. Należy zauważyć, że są to systemy bardzo podobne do systemów kryptograficznych opartych o kryptografię asymetryczną. Mianowicie mając pewną nośną, w ramach której chcemy umieścić ukrytą informację, najpierw szyfrujemy ją z wykorzystaniem klucza publicznego osoby, do której chcemy przesłać daną wiadomość. Odbiorca uzyskuje jawną postać przesyłanego tekstu poprzez odszyfrowanie go z użyciem swojego klucza prywatnego.

## Historia steganografii

Steganografia, podobnie jak kryptografia, jest nauką towarzyszącą człowiekowi już od zarania dziejów. Niczym bliźniaczki, obie te

dziedziny zajmujące się zatajaniem informacji rozwijały się wspólnie i często razem były wykorzystywane.

Pierwsze istotne wzmianki traktujące o wykorzystaniu technik steganografii możemy znaleźć już przed 2500 laty w starożytnej Grecji. Znany jest przypadek Demaratos, współkróla Sparty w latach 515-491 p.n.e., który wiadomość o nadciągającym na swe państwo perskich wojskach Xerxesza zawarł na drewnianej tabliczce pokrytej woskiem. Jednakże zamiast zapisać ją w wosku, jak zazwyczaj robiono, zapisał ją na drewnie, które następnie woskiem pokrył. Tabliczki wydawały się dzięki temu niezapisane, a ich przekazanie odbyło się bez problemu, mimo wrogich strażników na drogach [1].

Inną znaną historią z tego okresu jest przypadek Histajosa, władcy Miletu z około roku 493 p.n.e., który wiadomość o zbliżającym się ataku Persów wytatuaował na skórze głowy swojego niewolnika. Następnie wysłał on go jako posłańca, gdy tylko mężczyźnie odrośły włosy. Metoda okazała się bardzo efektywna – jej skuteczności dorównywała prawdopodobnie niestety jej czasochłonność [2].

Na przełomie średniowiecza i renesansu powstała pierwsza książka traktująca bezpośrednio o kryptologii, w szczególności o steganografii. Była to trylogia zatytułowana „Steganographia”, pióra Johanna Trithemiusa, niemieckiego benedyktyna.

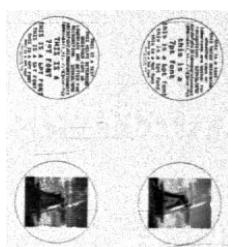


Rysunek 1. Johannes Trithemius (źródło: <http://www.phil-fak.uni-duesseldorf.de/philo/galerie/neuzeit/trithe.htm>)

Na pierwszy rzut oka może się wydawać, że jest to książka okultyścienna, traktująca o rozmowie z duchami. Jednakże po dogłębnej analizie okazało się, że jest to jedynie sposób zakodowania prawdziwej treści dzieła, związanej z kryptografią i ukrywaniem informacji [1].

Jedną z bardzo interesujących, lecz mało znanych metod steganograficznych będących dziełem Trithemiusa jest kodowanie „Ave Maria”, umożliwiające ukrycie dowolnej informacji pod przykrywką łacińskich modlitw.

Bliższa nam czasom analogowa metoda steganograficzna to mikrokropka. Używano jej do przesyłania informacji w czasie II Wojny Światowej, lecz nie straciła na popularności również dzisiaj i nadal wykorzystywana jest np. w przemyśle [3][4].



Rysunek 2. Zastosowanie metody mikrokropki (źródło: <http://www.wonko.net/>)

Metoda ta polega na zmniejszeniu informacji do wielkości pojedynczej kropki w tekście jawnym – kończącej zdanie, lub znajdującej się nad literą „i” lub „j”. W celu odczytania konieczny jest dobrej jakości mikroskop lub analogiczne urządzenie optyczne.



Rysunek 3. Szkło światłoczułe (źródło: [https://en.wikipedia.org/wiki/Photosensitive\\_glass](https://en.wikipedia.org/wiki/Photosensitive_glass))

Całkowicie inną strategię stanowi wykorzystanie światłoczułego szkła. Sposób jego wytwarzania opracowany został przez Stanleya Donalda Stookey w 1937 roku. Przez wiele lat stanowił sekret – materiał ten wykorzystywany był jako klisza w celu przesyłania tajnych informacji w postaci szklanych przedmiotów. Zapisane dane możliwe były do odczytu, tylko gdy szkło naświetlano promieniami ultrafioletowymi [5].

## PODSTAWOWE METODY STEGANOGRAFII CYFROWEJ

Kolejną, niezwykle ważną częścią naszej pracy jest przybliżenie czytelnikowi kilku najprostszych i najpowszechniej wykorzystywanych metod steganograficznych. W niniejszym podroziale przedstawimy zarówno metody, które działają na obrazach, jak i metody posługujące się tekstem. Zanim jednak do tego przejdziemy, chcielibyśmy wspomnieć o zasadzie „Security through obscurity” (bezpieczeństwo przez niejawność), która to jest innym ujęciem zasady Kerckhoffsa. Jak podają źródła internetowe, jest to zasada, która jest używana w bezpieczeństwie teleinformatycznym, a jej istotą jest ukrywanie detali dotyczących implementacji formatów i protokołów przed rywalami [6]. Otóż należy zauważyć, że dana zasada nie dotyczy steganografii. Jak już zostało wspomniane we wcześniejszej części artykułu – istnieje taki rodzaj steganografii, który pozwala na upublicznenie szczegółów systemu steganograficznego. Jednakże pomimo tego, że takie detale są publiczne, osoby atakujące nie są w stanie uzyskać informacji, jaka jest ukryta w przesyłanych wiadomościach. W tym miejscu chcielibyśmy po prostu taki przykład wskazać, że zasada „Security through obscurity” nie musi być spełniona, aby system gwarantował wysoki poziom bezpieczeństwa.

W niniejszym podroziale przedstawimy przykłady metod steganograficznych, które przekazują główną ideę stanowiącą podstawkę tejże gałęzi nauki.

### Steganografia tekstu

Steganografia tekstowa w erze Internetu musi podać innym wymogom aniżeli jej prekursorki z epoki preelektronicznej. Analogowe ukrywanie informacji w tekście przestało być opłacalne. Nowoczesne metody skupiają się na dodaniu niewidocznej informacji

do niepozornego tekstu, który następnie można zamieścić wśród tysiąca podobnych, np. jako komentarz pod ogólnodostępnym i chętnie komentowanym artykułem.

W Listingu 1 przedstawiamy prostą metodę, z użyciem której możemy zakodować informację w wybranym tekście. Wykorzystuje on cechę kodowania Unicode – niewidoczne, białe znaki ZWJ (zero width joiner) i ZNWJ (zero width non-joiner) wykorzystywane w niektórych alfabetach (arabskim, indyjskim) do wpływania na wyświetlanie znaków („grafemów”), które traktowane są jako bity 0 i 1 w celu prostego kodowania znaków ASCII [7].

#### Listing 1. Prosta metoda kodowania informacji w wybranym tekście

```
#!/usr/bin/env python3

from itertools import zip_longest
import sys
SECRET = sys.argv[1]
TEXT = sys.argv[2]

def encode(character):
    return bin(ord(character))[2:1].replace('0', '\u200D').replace('1', '\u200C')

encoded = map(encode, SECRET)
paired = zip_longest(encoded, TEXT, fillvalue='')
combined = (s+t for s, t in paired)
joined = ''.join(combined)
print(joined)
```

Kod programu jest bardzo zwięzły dzięki wykorzystaniu operatorów funkcyjnych. Tekst do ukrycia znajduje się w zmiennej SECRET, tekst jawnny w zmiennej TEXT.

Funkcja encode stanowi klucz rozwiązań. Każdy znak tekstu niejawnego jest nim traktowany w celu jego ukrycia (linia 12). Zamienia ona symbol z postaci znakowej do jego reprezentacji binarnej, następnie każde zero reprezentacji binarnej następuje znakiem ZWJ, a każdą jedynkę ZWNJ.

Tak zakodowany tekst jest łączony parami w formie symbolu niejawnego-symbol jawnego (linie 13-14). Konstruowany jest ciąg wynikowy (linia 15) wypisany następnie na ekran. Dla zwięzości pominięto obsługę błędów. Jest to znaczco uproszczona wersja algorytmu z [7]. Udoskonalenie polegałoby na wygenerowaniu własnego alfabetu znaków, zależnego od częstotliwości występowania liter w alfabetie źródłowym tekstu niejawnego za pomocą np. kodowania Huffmmana, algorytmu bezstratnej kompresji, dzięki któremu zminimalizowalibyśmy liczbę niejawnych symboli. Pozostawiamy to jako ćwiczenie dla czytelnika.

Dużą zaletą opisanego wyżej algorytmu jest jego prostota i możliwość wykorzystania na praktycznie dowolnej stronie społecznościowej. Tekst steganograficzny łatwo może wówczas zniknąć w natłoku innych komentarzy i wpisów.

Dużą wadą powyższego rozwiązania jest fakt, że niejawnna informacja nie przetrwa sytuacji wydrukowania i ponownego wprowadzenia do komputera. Niestety jest to cecha bardzo wielu systemów steganografii cyfrowej. Ponadto sekretny tekst nie może przekraczać długości tekstu niejawnego. Ponownie rozwiązanie jest stosunkowo proste i z tego względu pozostawiamy je czytelnikowi.

#### Listing 2. Algorytm dekodowania informacji

```
#!/usr/bin/env python3

TEXT = input()
secret = ''
for character in TEXT:
```

```
    if character in ('\u200D', '\u200C'):
        secret += character
    elif secret != '':
        decoded = secret.replace('\u200D', '0').replace('\u200C', '1')
        print(chr(int(decoded, 2)), end='')
        secret = ''
print()
```

Algorytm dekodowania jest zaledwie odrobinę dłuższy od algorytmu kodowania. W zadanym tekście wyszukiwane są znaki ZWJ i ZNWJ i grupowane są one w ciągi (linie 7-8). Następuje transformacja odwrotna ZWJ do zera, ZNWJ do jedynki (linia 10). Tak powstała liczba binarna przekształcana jest na znak (linia 11) i wypisywana na ekran (linia 12).

#### Listing 3. Przykład użycia opisanego programu

```
$ ./encode.py "Marysia ma psa" "Paweł ma fretki" > wiadomosc
$ cat wiadomosc
Paweł ma fretki
$ cat wiadomosc | ./decode.py
Marysia ma psa
```

Argumenty podawane są jako parametry przy wywołaniu programu encode. Pierwszy argument to tekst niejawnny, drugi to tekst jawnny, pod którym ukryjemy tajną wiadomość.

W przypadku próby wypisania wiadomości na ekran nieauważymy żadnych nieprawidłowości (linia 3). Podobnie byłoby, gdybyśmy zamieściliów wiadomość na jakiejkolwiek platformie społecznościowej.

Dopiero zastosowanie programu decode pozwoli na odkrycie jej tajnej części (Listing 4).

Powyżej widzimy, że prosta analiza programem hexdump, wypisującego reprezentację szesnastkową pliku „wiadomość”, pozwala zauważać, że znajdują się w nim nadmiarowe dane. Konkretnie są to sekwencje „e2 80 8d” oznaczające ZWJ i „e2 80 8c” oznaczające ZWNJ.

Naszym przyjacielem w tym przypadku jest ukrycie wiadomości w Internecie pośród setek tysięcy podobnych, tak by jej znalezienie okazało się możliwie trudne i do analizy wcale nie doszło, gdyż w innym przypadku odkrycie tekstu niejawnego jest bardzo proste – nie jest on w żaden sposób szyfrowany (kolejny sposób na udoskonalenie algorytmu).

## Steganografia na obrazach

Odnosząc steganografię na obrazach do strategii, jaka została wykorzystana w przypadku tekstu, musimy powiedzieć, że jest to całkowicie odmienne podejście. Jako przykład w ramach niniejszego podrozdziału przedstawimy strategię opartą o drobną modyfikację obrazu. Mianowicie będziemy dokonywać modyfikacji na najmniej znaczącym bicie piksela.

### Najmniej znaczący bit

Pierwsza ze strategii, którą chcielibyśmy opisać, to strategia, która opiera się o najmniej znaczący bit. Zauważmy, że mając do dyspozycji obraz, każdy jego piksel jest opisany przez trzy wartości, które reprezentują poszczególne kanały z przestrzeni barw RGB. Czy jednak zauważymy dużą zmianę, jeżeli np. zamiast wartości 254 na kanale R zostanie ustawiona wartość 255? Zanim się temu przyjrzymy, w Listingu 5 przedstawiamy prosty program, który zmienia wartość ostatniego bitu na wszystkich kanałach, wszystkich pikseli na 1.

**Listing 4.** Analiza wiadomości programem hexdump

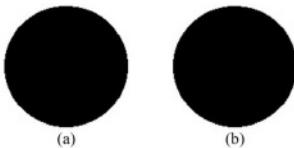
```
$ cat wiadomosc | hexdump -c
00000000 e2 80 8c e2 80 8d e2 80 8d e2 80 8c e2 80 8c e2 | .....P.....
00000010 80 8d e2 80 8c 50 e2 80 8c e2 80 8c e2 80 8d e2 | .....a.....
00000020 80 8d e2 80 8d e2 80 8d e2 80 8c 61 e2 80 8c e2 | .....w.....
00000030 80 8c e2 80 8c e2 80 8d e2 80 8d e2 80 8c e2 80 | .....e.....
00000040 8d 77 e2 80 8c e2 80 8c e2 80 8c e2 80 8c e2 80 | .....e.....
00000050 8d e2 80 8d e2 80 8c 65 e2 80 8c e2 80 8c e2 80 | .....m.....
00000060 8c e2 80 8d e2 80 8d e2 80 8c e2 80 8c c5 82 e2 | .....a.....
00000070 80 8c e2 80 8c e2 80 8d e2 80 8c e2 80 8d e2 80 | .....f.....
00000080 8d e2 80 8c 20 e2 80 8c e2 80 8c e2 80 8d e2 80 | .....r.....
00000090 8d e2 80 8d e2 80 8d e2 80 8c 6d e2 80 8c e2 80 | .....f.....
000000a0 8d e2 80 8d e2 80 8d e2 80 8d e2 80 8d 61 e2 80 | .....r.....
000000b0 8c e2 80 8c e2 80 8d e2 80 8c e2 80 8c e2 80 8d | .....t.....
000000c0 8c e2 80 8c 20 80 8c e2 80 8c e2 80 8d e2 80 8d | .....ki...
000000d0 82 80 8d e2 80 8d e2 80 8c 66 e2 80 8c e2 80 8d | .....ki...
000000e0 82 80 8d e2 80 8d e2 80 8d e2 80 8d 72 e2 80 8c | .....ki...
000000f0 82 80 8c e2 80 8c e2 80 8d e2 80 8d e2 80 8d e2 | .....ki...
00000100 80 8d 65 e2 80 8c e2 80 8c e2 80 8c e2 80 8d e2 | .....ki...
00000110 80 8d e2 80 8c e2 80 8c 74 e2 80 8c e2 80 8c e2 | .....ki...
00000120 80 8d e2 80 8d e2 80 8d e2 80 8d e2 80 8c 6b 69 | .....ki...
00000130 0a | ..|.
00000131
```

**Listing 5.** Kod, który służy do ustawienia wszystkich ostatnich bitów na 1

```
try {
    BufferedImage imageInput = ImageIO.read(new File("./resources/test.jpg"));
    BufferedImage imageOutput = new BufferedImage(imageInput.getWidth(), imageInput.getHeight(), BufferedImage.TYPE_INT_RGB);
    for(int w = 0; w < imageInput.getWidth(); w++) {
        for(int h = 0; h < imageInput.getHeight(); h++) {
            Color c = new Color(imageInput.getRGB(w, h));
            int red = c.getRed() | 1;
            int green = c.getGreen() | 1;
            int blue = c.getBlue() | 1;
            Color output = new Color(red, green, blue);
            imageOutput.setRGB(w, h, output.getRGB());
        }
    }
    ImageIO.write(imageOutput, "jpg", new File("./resources/testOutput.jpg"));
} catch (IOException ex) {
    ex.printStackTrace();
}
```

Na Rysunku 4 przedstawiamy obraz wejściowy oraz obraz wynikowy, uzyskany z użyciem powyższego kodu.

W tym miejscu możemy postawić czytelnikowi pytanie, czy według niego obraz oryginalny różni się od obrazu po przeprowadzeniu modyfikacji? Według nas nie. Dlaczego tak się dzieje? Otóż spójrzmy najpierw na kod. W pierwszym kroku wczytujemy obraz, następnie tworzymy drugi obraz o dokładnie takich samych wymiarach jak obraz oryginalny. Następnie przehodząc po wszystkich pikselach, ustawiamy wartość ostatniego, najmniej znaczącego bitu na 1 (jeżeli już taka wartość była zawczas ustawiona, wtedy nie ma potrzeby jej zmiany). Zauważmy bowiem, że odpowiedzią na wcześniej postawione pytanie jest fakt, który świadczy o tym, że tak delikatnej zmiany nie jesteśmy w stanie wychwycić gołym okiem. Zwróćmy również naszą uwagę na to, że wiedząc, które piksele powinny zawierać ważną informację, jesteśmy w stanie w obrazie



Rysunek 4. Obraz oryginalny (a) oraz obraz zmodyfikowany (b)

(w formie kodu binarnego) zakodować nawet całe zdania (odczytując odpowiednie piksele z wykorzystaniem kodów ASCII). Jest to bardzo prosta metoda, która pozwala nam na ukrycie komunikatu w obrazie.

## Steganografia z wykorzystaniem pliku zip

Ostatnią techniką, którą chcielibyśmy przedstawić, jest metoda steganograficzna oparta o proste dołączenie archiwum do obrazu. Ktoś mógłby zapytać, jak to zrobić? Otóż wystarczy podstawowa znajomość systemu Linux.

Załóżmy, że dysponujemy plikiem *testCompress.tar.gz* oraz plikiem *test.jpg*. Najprostszym rozwiązaniem na ukrycie tego pliku w obrazie będzie wykorzystanie komendy, którą prezentujemy w Listingu 6.

**Listing 6.** Komenda, z użyciem której dopisujemy archiwum do pliku

```
echo testCompress.tar.gz >> test.jpg
```

Czy to zmienia cokolwiek w odczycie samego obrazu? Otóż nie, jedyną różnicę będziemy mogli zaobserwować, podglądając plik, chociażby z użyciem polecenia *cat test.jpg | hexdump -c*. Uzyskaną przez nas końcówkę pliku przedstawiamy w Listingu 7.

Jak możemy zaobserwować w Listingu 7, plik skompresowany został dopisany na koniec pliku (zgodnie z założeniami). Jednakże ta metoda steganografii jest bardzo łatwa do detekcji. Ten plik przecież został przez nas zaobserwowany z wykorzystaniem zwykłej metody *cat* połączonej przez pipe z *hexdump*. Również wadą tej metody jest trudność w ponownym pozyskaniu danych, które zostały ukryte w pliku. Ktoś mógłby zatem zapytać, po co w takim razie ją przedstawiamy. Otóż została ona zaprezentowana ze względu na jej prostotę oraz łatwość wykonania – nie musimy bowiem posiadać dużej wiedzy programistycznej, aby móc takiej operacji dokonać. Co więcej, spełnia ona najważniejsze założenie, jakie jest stawiane przez steganografię. Mianowicie wiadomość ukryta nie wpływa na sposób wyświetlania wiadomości bazowej. Wyświetlając zmodyfikowane zdjęcie, nie zaobserwujemy jakiejkolwiek różnicy w stosunku do zdjęcia oryginalnego.

## Podsumowanie

W niniejszym podręczniku zostały zaprezentowane bardzo proste metody, które nie zapewniają adekwatnego poziomu bezpieczeństwa dla naszych ukrytych danych. Zauważmy bowiem, że poprzez odpowiednią ich analizę jesteśmy w stanie w bardzo prosty sposób wychwycić modyfikacje i uzyskać jawną formę przesyłanego tajnego komunikatu. W kolejnym podręczniku skupimy się na metodzie, która zagwarantuje nam wyższy poziom bezpieczeństwa aniżeli ten, który został uzyskany z wykorzystaniem dotychczas opisanych metod.

**Listing 7.** Końcówka zmodyfikowanego pliku

```
00001510 5c 32 22 34 cd 14 08 88 64 29 1c 6a 58 8c 91 1a | \2"4....d).jX...
00001520 8c e0 0c 76 94 51 40 05 14 51 40 05 14 51 40 05 | ...v.Q@..Q@..Q@...
00001530 14 51 40 1f ff d9 74 65 73 74 43 6f 6d 70 72 65 | .Q@...testCompre...
00001540 73 73 2e 74 61 72 2e 67 7a 0a | ss.tar.gz.|
```

## ZAAWANSOWANE METODY STEGANOGRAFII CYFROWEJ

W tej części skupimy się na bardziej zaawansowanych metodach steganografii cyfrowej. W tym przypadku nasze przykłady będą dotyczyć obrazów oraz wprowadzania tajnych informacji w procesie kompresji JPEG. Aby nie powtarzać informacji w obydwu podpunktach przynależnych do tego podrozdziału, rozpoczynamy naszą analizę od krótkiego przedstawienia kroków algorytmu kompresji JPEG.

### Algorytm kompresji JPEG – kroki algorytmu

1. Konwersja obrazu z RGB na luminancję i dwie chrominancje (YCbCr). Luminancja musi być przenoszona w wysokiej rozdzielcości, natomiast chrominancje mogą mieć mniejszą rozdzielcość, gdyż niosą ze sobą różnice barw (które są trudniej wychwytywane przez człowieka aniżeli różnice w jasności).
2. Redukcja kanałów barwy.
3. Podział kanałów na bloki o rozmiarze 8x8 pikseli.
4. Wykonanie na blokach dyskretnej transformaty cosinusowej (DCT). Zamiast poszczególnych wartości pikseli, dysponujemy w tym momencie średnią wartością wewnątrz bloku, a także częstotliwością zmian wewnątrz bloku – obie są wyrażane liczbami zmiennoprzecinkowymi.
5. Zastąpienie średnich wartości w bloku poprzez różnice z wartością poprzedzającą.
6. Kwantyzacja – czyli zmiana wartości zmiennoprzecinkowych poprzez liczby całkowite.
7. Zygzkakowe uporządkowanie współczynników DCT (w celu uniknięcia sytuacji, gdy zera leżą obok siebie).
8. Niezerowe współczynniki są kompresowane algorytmem Huffmana.

W tym miejscu nie będziemy skupiać się na wyjaśnieniu poszczególnych kroków algorytmu, gdyż nie jest to naszym celem w ramach niniejszego artykułu. Najważniejsze dla czytelnika w kontekście dwóch kolejnych algorytmów jest to, że w kompresji JPEG występuje operacja DCT wykonywana na blokach o rozmiarach 8x8 pikseli.

### Algorytm JSTEG

Pierwszym algorytmem, który chcielibyśmy omówić w ramach naszego artykułu, jest algorytm JSTEG. Jego skrócony opis możemy przeczytać w ramach artykułu [9]. Ogólnie można stwierdzić, że algorytm JSTEG jest metodą opartą o algorytm LSB (najmniej znaczącego bitu), z tym że jest ona wykonywana w domenie częstotliwościowej. Na czym zatem polega ta zmiana? Otóż w momencie, w którym wykonywana jest dyskretna transformata cosinusowa (DCT), następuje modyfikacja współczynników transformaty. Oznacza to, że tajny komunikat nie jest przekazywany bezpośrednio w wartościach pikseli, ale w ramach współczynników transformaty. W uproszczeniu możemy powiedzieć, że DCT jest odpowiednikiem Transformaty Fouriera obejmującej jedynie część rzeczywistą wyniku.

Niestety ten algorytm nie jest wolny od wad. Otóż zostało dowiedzione [10], że jest on bardzo podatny na ataki statystyczne. Jedną z metod, którą można złamać ten algorytm steganograficzny, jest atak Chi-Square.

Ten algorytm można w bardzo prosty sposób zaimplementować, korzystając z języka Python oraz wykorzystując pakiet `pysteg.jsteg`.

### Algorytm F5

Znacznie ciekawszym algorytmem jest algorytm F5 [9], będący swoistym następcą takich algorytmów jak JSTEG czy F3. Uważny czytelnik zauważyczy, że opis, który przedstawiamy w ramach niniejszego podrozdziału, równie dobrze mógłby odpowiadać opisowi wcześniej przedstawionego algorytmu JSTEG. Na czym jednak polegają różnice pomiędzy wcześniej opisaną metodą a algorytmem F5? Otóż w przypadku algorytmu JSTEG wykonywaliśmy modyfikację LSB na wartościach współczynników transformaty. Gdy rozpatrujemy jednak algorytm F5, zauważyczy, że współczynniki transformaty cosinusowej są modyfikowane dopiero, gdy zostaną one skwantyzowane. Ponadto modyfikacja polega na zmniejszeniu wartości bezwzględnej wybranego współczynnika w zależności od wartości ukrywanego bitu informacji. Przykład takiej modyfikacji prezentujemy w Tabeli 1 oraz Tabeli 2.

69	1	5	4
3	0	-1	-7
2	0	1	1
1	8	1	2

Tabela 1. Oryginalne wartości współczynników DCT

68	1	5	4
3	-1	-2	-7
2	-1	0	0
1	8	1	2

Tabela 2. Zmodyfikowane wartości współczynników DCT

W Tabeli 2 zmodyfikowane wartości współczynników DCT zostały pogrubione. Cechą algorytmu F5, która pozwala na zwiększenie bezpieczeństwa przenoszonych danych, jest rozproszenie zmodyfikowanych wartości zgodnie z wybranym hasłem (kluczem). Natomiast liczba zmodyfikowanych zmian jest redukowana poprzez zastosowanie techniki kodowania macierzy.

Analogicznie jak w przypadku algorytmu JSTEG możemy zaimplementować rzeczony algorytm poprzez zastosowanie pakietu `pysteg.jsteg`.

### Podsumowanie

W tej części artykułu przedstawiliśmy dwa bardziej zaawansowane algorytmy steganograficzne. Pominięte zostały implementacje ze względu na ich prostotę. Zachęcamy czytelników do samodzielnej implementacji wspomnianych rozwiązań i zweryfikowania poprawności ich działania. Zamieszczone w tym podrozdziale algorytmy stanowią dobrą podstawę do zapoznawania się z jeszcze bardziej zaawansowanymi algorytmami steganograficznymi. W obu przedstawionych tutaj przypadkach zastosowana została steganografia z wykorzystaniem obrazów, jednakże jak już zaznaczyliśmy wcześniej w ramach tego artykułu, technika ta może być wykorzystywana również w ramach samego tekstu.

## UŻYCIE STEGANOGRAFIИ W CODZIENNYM ŻYCIU

Użycie steganografii możemy zaobserwować także w życiu codziennym. Jednym z przykładów jest historia zamieszczona na portalu niebezpiecznik.pl [8]. Otóż amerykańska wokalistka Britney Spears regularnie wrzuca zdjęcia na swój profil na Instagramie (co tak naprawdę czyni dużą rzeszą ludzi). Oczywiście jak to w przypadku znanych osób, każde ze zdjęć ma bardzo dużo komentarzy (nawet do kilkunastu tysięcy). Ze względu na ich mnogość bardzo trudno jest je wszystkie przeczytać. Z podobnego założenia wyszła grupa o nazwie „Turla”. Na czym polegał ich sposób wykorzystania steganografii? Otóż tworząc komentarz, trzymali się zasad (oraz ich delikatnych modyfikacji), które zostały przedstawione w ramach niniejszego artykułu w podrozdziale „Steganografia tekstu”. Jednakże czy wszystkie komentarze mogły zawierać ukrytą informację? Oczywiście, że nie. Grupa posługiwała się prostym algorytmem i parsowała jedynie te komentarze, których wartość liczbową była równa dokładnie 183. Aplikacja botnetu każdy z takich komentarzy przepuszczala przez wyrażenie regularne, które miało postać następującą (`(?:\u200d(?:#|@)(\\w))`). Weryfikowane było przez to, czy odczytany komentarz zawiera znak o kodzie \200d, jeżeli występował po nim # albo @, były one ignorowane, a odczytywana była kolejna litera. Zabieg z użyciem @ lub # został wykorzystany w celu ominienia zasad działania portalu Instagram. W ten sposób po odseparowaniu niepotrzebnych znaków uzyskiwany był pewien ciąg, który stanowił część adresu URL. Aby uzyskać dostęp do rzeczywistej strony, która była podstawowym celem grupy, należało dopisać uzyskany ciąg do adresu `bit.ly/`.

Oczywiście pytanie, które może się w tym momencie nasunąć czytelnikowi, jest następujące: po co ktoś ukrywał tak skrzętnie adres URL do pewnej strony? Otóż odpowiedź jest banalnie prosta. Jeżeli komputer zwykłego użytkownika został już wcześniej podpięty do botnetu, wtedy po uzyskaniu adresu URL był przekierowywany do strony, z której to pobierał instrukcje do wykonania. Kolejne pytanie, które może zostać zadane, to w jaki sposób przeciwdziałać temu zjawisku. Otóż jest to bardzo trudne, ponieważ nie istnieje możliwość zablokowania wszystkich groźnych adresów URL (nie istnieje bowiem żadna globalna baza danych takich adresów – ponadto trzeba byłoby je codziennie aktualizować, a i tak nie mielibyśmy żadnej gwarancji, czy strona spoza listy nie jest groźna). Jak również nie istnieje możliwość zakazania użycia znaku specjalnego, który służy do oddzielania emotikonów od siebie.

Czy zatem nie istnieje możliwość przeciwdziałania? Oczywiście, że istnieje. Na ten moment oszuści rozprzestrzeniali złośliwą aplikację (która analizowała tekst) jako rozszerzenie XPI, która używała przestarzałych metod API z Firefoxa. Od najnowszej wersji przeglądarek te rozszerzenia przestaną być obsługiwane.

Oczywiście nie znaczy to, że jesteśmy już całkowicie bezpieczni, albowiem oszuści potrafią znaleźć coraz to bardziej wymyślne luki w systemie w celu podłożenia nam szkodliwej aplikacji czy też kodu. Jest to ciemna strona użycia steganografii. Profesor James Moriarty byłby dumny!

## PODSUMOWANIE

W powieści „Zabójstwo przy moście” Sherlock Holmes stwierdza: „Szukajmy logiki. Tam, gdzie jej brakuje, możemy podejrzewać przestępstwa”. Odnosząc ten cytat do opisywanej w ramach niniejszego artykułu steganografii, możemy skonkludować, że ukryty

komunikat w pewnym stopniu ucieka poza ramy logiki. Czy może stać się on przyczynkiem do przestępstwa? A może samym przestępstwem? Steganografia jest nauką, która może służyć zarówno jasnej, jak i ciemnej stronie mocy. W dobrych rękach może zostać użyta chociażby do przekazywania tajnych informacji umożliwiających efektywną obronę przed wrogiem. Jednakże może ona posłużyć również do przekazywania danych, które mogą doprowadzić do kradzieży albo przyłączenia urządzeń do botnetu.

Wiele źródeł informuje, że steganografia w dzisiejszym świecie nie jest już potrzebna oraz że już dawno temu odeszła w zapomnienie. Parafrując klasyka: „pogłoski o śmierci steganografii są mocno przesadzone”. Jak mogliśmy zaobserwować w ramach tego artykułu, steganografia jest wciąż w użyciu. Według nas w świecie cyfrowym steganografia może być jednym z najbardziej efektywnych sposobów ukrycia komunikatu. Co więcej, w dzisiejszym świecie, w którym mamy do czynienia z coraz bardziej wymyślnymi sposobami kradzieży plików, haseł etc., to steganografia, a nie kryptografia, może być efektywnym sposobem na ochronę naszych danych.

W tym artykule przedstawiliśmy trzy proste sposoby na ukrycie danych. Zachęcamy oczywiście czytelników do samodzielnego zgłębiania meandrów steganografii. Równocześnie w razie pytań czy też wątpliwości zachęcamy do kontaktu.

## Bibliografia

- [1] <https://goo.gl/f2ovZT> (dostęp 1.12.2017)
- [2] <https://en.wikipedia.org/wiki/Steganography> (dostęp 1.12.2017)
- [3] <https://en.wikipedia.org/wiki/Microdot> (dostęp 1.12.2017)
- [4] <https://goo.gl/6wqiXi> (dostęp 1.12.2017)
- [5] [https://en.wikipedia.org/wiki/Photosensitive\\_glass](https://en.wikipedia.org/wiki/Photosensitive_glass) (dostęp 1.12.2017)
- [6] [https://en.wikipedia.org/wiki/Security\\_through\\_obscurity](https://en.wikipedia.org/wiki/Security_through_obscurity) (dostęp 9.12.2017)
- [7] <https://goo.gl/DnqE3P> (dostęp 1.12.2017)
- [8] <https://goo.gl/HCgBjk> (dostęp 10.12.2017)
- [9] A. Westfeld – „F5 - A Steganographic Algorithm. High Capacity Despite Better Steganalysis.”
- [10] <https://goo.gl/1aeHQs> (dostęp 17.12.2017)



### MACIEJ SZYMKOWSKI

[maciej.szymkowski@symmetra.pl](mailto:maciej.szymkowski@symmetra.pl)

Inżynier Informatyk, Junior Software Developer w Symmetra Sp. z o.o., student informatyki (II stopień) na Wydziale Informatyki Politechniki Białostockiej. Do swoich zainteresowań zalicza: biometrię, przetwarzanie i analizę obrazów oraz sygnałów, elektronikę, architekturę komputerów oraz zagadnienia bezpieczeństwa komputerowego. Uwielbia poszerzać swoją wiedzę poprzez uczestnictwo w konferencjach oraz lekturę książek i artykułów. Programuje w takich językach jak: Java, C#, C++, C, Python oraz Swift.



### PAWEŁ JAN CZOCHAŃSKI

[pawel.czochanski@symmetra.pl](mailto:pawel.czochanski@symmetra.pl)

Magister Inżynier Informatyk, Team Leader w Symmetra Sp. z o.o., był wykładowcą na Wydziale Informatyki Politechniki Białostockiej. Zainteresowany ogólnopojętym security, inżynierią oprogramowania i architekturą komputerów. Miłośnik fantastyki i podróży.

# Opus Magnum C++11. Programowanie w języku C++



*Opus Magnum C++11* to trzecia książka (choć lepiej napisać, że podręcznik) Jerzego Grębosza, po *Pasji C++* oraz *Symfonii C++*, przeznaczona dla osób, które chcieliby nauczyć się języka C++. Trzeba wspomnieć, że poprzednie wydania cieszyły się popularnością, choć nie – to za mało powiedziane – raczej ogólną popularnością i nie będzie przesadą napisać, iż mamy już kilka roczników programistów C++, którzy uczyli się programować, korzystając z książek pana Grębosza.

Sam język C++ podlega zmianom, uzupełnieniom, zatem jak można było się spodziewać, kolejna odsłona serii była tylko kwestią czasu.

Nowe wydanie jest też najobszerniejsze i można je krótko podsumować: trzy tomu (pewnie za jakiś czas doczekamy się jednego ogromnego tomu) i niemal 1700 stron opowieści o programowaniu dla osób, które chcą rozpocząć przygodę z pisaniem programów – czy ogólnie z programowaniem. Dość powiedzieć, że sam spis treści liczy sobie 20 stron.

Książki Jerzego Grębosza, jak każdy porządkowy podręcznik, zawierają sporo rad i porad dla osób rozpoczynających naukę programowania. Niemniej nie są one kierowane tylko dla początkujących; również osoby już z pewnym doświadczeniem wyniosą dużo z lektury tej książki. Walory edukacyjne to niewątpliwa zaleta *Opus Magnum*. Książka została napisana prostym językiem, a wiele podstawowych pojęć, które doświadczyonym programistom wydają się trywialnymi, wbrew pozorom są dość abstrakcyjne – i w książce zostały przedstawione w sposób intuicyjny. W licznych przykładach omówiono ich znaczenie i przedstawiono ich rolę w tworzonych przez nas programach.

Główną zmianą tudzież nowością względem poprzednich wydań jest opis konstrukcji, jakie znajdziemy w standardzie C++11. Sam standard jest już obecny we wszystkich głównych kompilatorach, zatem książka jak najbardziej jest na czasie, jeśli chodzi o rozwój języka C++. Nie ma natomiast odniesień do nowego standardu C++14, jak również C++17. Ponieważ C++14 nie wprowadza aż tak dużo zmian jak wersja C++11, toteż nie jest to duży problem, natomiast brak odniesień do C++17, no cóż – standard formalnie został przyjęty na wiosnę 2017, zatem musimy jeszcze poczekać na jego pełne wdrożenie.

Nowa książka wzorem poprzednich ma też bardzo czytelną typografię. Mimo że pozycji tej nie wydano w kolorze, to i tak czyta-

się ją bardzo dobrze, właśnie ze względu na wszystkie oznaczenia, zmiany czcionek, wyróżnienia czy uwagi i przypisy. Sporą zaletą jest też możliwość czytania wybranych rozdziałów. Przy czym jeśli istotnie dopiero rozpoczynamy naukę języka C++, i dodatkowo jest to nasz pierwszy bądź drugi język programowania, to warto przeczytać pierwszy tom od deski do deski. Dużym ułatwieniem jest też skorowidz umożliwiający odszukiwanie potrzebnych nam pojęć.

*Opus Magnum* to trzecia książka poświęcona C++. Materiał, jaki zawiera, to naturalnie rozwinięcie tekstu z poprzednich wydań, zatem mamy rzetelnie opracowany tekst i materiał na bardzo wysokim poziomie merytorycznym, co jest szczególnie ważne dla osób początkujących. Jest to więc pozycja, którą można i trzeba polecać. Można by dodać, iż przydałoby się zaprezentować więcej elementów ze standardowej biblioteki, ale z drugiej strony zwiększyłoby to objętość i tak już dużej książki.

Mamy też sporą liczbę zadań, do których przydałyby się odpowiedzi. Można by, wzorem książki Tony'ego L. Hansena *C++ zadania i odpowiedzi*, przygotować opracowanie zadań właśnie w postaci książki. Jednakże oprócz odpowiedzi przydałoby się też oficjalne repozytorium kodu, w którym znalazłyby się kody przykładów zamieszczonych w poszczególnych tomach, a także odpowiedzi do zadań opracowywanych przez społeczność. Miejmy nadzieję, że doczekamy się oficjalnego repozytorium kodu, którego brakuje tak dobrej książce, jaką jest *Opus Magnum*.

W książce zabrakło też innego elementu, a mianowicie omówienia systemu kompilacji, choćby opartego o system make, lub chociaż gotowych projektów z kodem źródłowym do takich środowisk jak Xcode, Visual Studio czy np. Eclipse. Cechą C++ jest też umiejętność posługiwania się kompilatorem, a tego w pozycji Jerzego Grębosza niestety nie znajdziemy. To wszystko jednak nie ujmuję niczego samej książki, autor *Opus Magnum* celowo unika wszelkich technicznych aspektów, w końcu przecież programujemy w głowie, w pamięci, komputer tylko wykonuje nasze programy :-)

Marek Sawerwain

Tytuł:	<i>Opus Magnum C++11. Programowanie w języku C++</i>
Autor:	Jerzy Grębosz
Stron:	1696
Wydawnictwo:	Helion
Data wydania:	2017-10-23



Szkolenia i warsztaty eksperckie - **bo umysł to Twoje najważniejsze narzędzie**



DDD



ARCH



TEST&CRAFT



AGILE&SOFT



JAVA



.NET



C&CPP



WEB



BAZY



MOBILNE



EIP

SPRAWDŹ **200**  
**AUTORSKICH**  
**PROGRAMÓW**  
SZKOLEŃ



ERICSSON

# JOIN THE C-TEAM

## Praca

Chcesz mieć wpływ na to, w którym kierunku zmierza postęp technologiczny? Brać udział w prestiżowych projektach IT realizowanych przez jednego z kluczowych graczy branży ICT? Rазвивать się zawodowo и жить полно жизни по работе?

Dołącz do Ericsson. Stwarzamy naszym zespołom możliwości rozwijania umiejętności i dzielenia się pasjami. U nas zrealizujesz ciekawe projekty w dobrej, przyjacielskiej atmosferze.

Dowiedz się więcej o pracy w Ericsson:  
[Ericsson.com/careerspoland](http://Ericsson.com/careerspoland)



[tiny.cc/theEteam](http://tiny.cc/theEteam)