Roman Ulan

Wzorce silników zdarzeń w C++

Część VI: Testowanie modułowe vs jednostkowe

W kolejnych artykułach przedstawiałem kolejne wzorce, które znacznie porządkują i upraszczają implementację silników obsługi zdarzeń, a w ostatnim przedstawiłem rozwiązanie integrujące wszystkie wcześniej omawiane wzorce w jedną konfigurowalną aplikację. Dotychczas temat testowania przewijał się jedynie w ocenie poszczególnych rozwiązań. W tej części te proste sformułowania nabiorą głębszego sensu, a przy tym przedstawię jeden z ciekawszych frameworków dedykowany do tego typu testów w języku C++.

UNIT VS MODULE

Na początek rozbijmy podstawową nomenklaturę testów developerskich – tak będę nazywał zbiorczo testy jednostkowe (UT) i modułowe (MT). Zacznijmy od tej najbardziej ogólnej części – idei.

Czym są i jakie jest zadanie testów developerskich?

Przede wszystkim testy developerskie są podstawową składową kodu, którego nie będziemy nazywać kodem „legacy" - zastanym, niepodatnym na zmiany, nieposiadającym testów. Są drugą (po kompilatorze) linią zaporową przed wprowadzaniem nowych błędów – stanowiąc swoistą regresję dla przyszłych zmian nie tylko w fazie utrzymania, ale także w trakcie aktywnego rozwoju nowych funkcjonalności. Są szczególnie istotne w dużych projektach, w których często występują dwa osobne środowiska:

- środowisko pracy developerskiej (edycja i budowanie kodu)
- środowisko docelowe (miejsce, w którym kod jest wykonywany, wymagające często mniej lub bardziej skomplikowanej procedury instalacji tzw. binarek).

W takim wypadku, dla celu klasycznego rozwoju oprogramowania (np. z pomocą debuggera) istnieje silna potrzeba stworzenia możliwości uruchomienia naszego kodu na maszynie developerskiej. Najprostszą i najbardziej pożyteczną metodą są wówczas testy UT i MT.

UT i MT – różnice i cechy wspólne

Owe testy z założenia uruchamiane są na tej samej maszynie (w tym samym środowisku), gdzie jest rozwijane dane oprogramowanie.

Podstawowe różnice między nimi uwypukla poniższa tabela:

Unit Test	Module Test
Testuje jednostkę kompilacji (pojedynczy plik c, cpp lub tylko klasę)	Testuje jednostkę wdrożeniową (najczęściej bibliotekę DLL)
Ogranicza się do wywołania publicznego API pojedynczego obiektu	Testuje cały przebieg w obrębie komponentu/podsystemu
Zaślepiane są zależności testowanego obiektu	Zaślepiane jest całe środowisko wokół testowanego modułu
Nie testuje całości przebiegu danego przypadku użycia (ang. *use case*)	Trudność w uzyskaniu wysokiego pokrycia w metryce linii kodu

O ile trzy pierwsze wiersze są zupełnie zrozumiałe, tak już zaznaczone na czerwono wady niekoniecznie. Weźmy na początek wadę UT – poważnym problemem w kontekście nietestowania całości przebiegu jest to, że znacznie ograniczamy pole wykrywalności problemów wielowątkowych takich jak „deadlock", które najczęściej spotyka się pomiędzy obiektami.

Jeśli chodzi o wadę MT, to wyjdźmy od przyczyn – te przeważnie leżą w wymogach, jakie stawiają najczęściej między sobą korporacje, by (często tylko pozornie) mierzyć jakość kodu. Najczęściej spotykaną w takich sytuacjach metryką staje się konieczność pokrycia 100% linii kodu (każda linia kodu produkcyjnego jest przynajmniej raz wykonana przez program testowy). W takich warunkach, gdy dołożymy do tego złożoną pajęczynę zależności modułu, testowanie zorientowane na tego rodzaju statystyki staje się niezwykle trudne. Najczęściej kończy się to podwajaniem szacunków na dalszy rozwój oprogramowania z uwagi na ciągłą potrzebę rozwijania/zmiany już skomplikowanych testów.

Let them work together

Jak więc temu zaradzić, by z jednej strony posiadać odpowiednio wysokie pokrycie, ale i by nasze testy nie były dodatkowym balastem i były solidną zaporą w przyszłości? Otóż jak zwykle nie ma jednego wytrychu, który za nas załatwi sprawę. Z mojego skromnego doświadczenia wynika, że właściwie jedynie synteza obu podejść da nam najwięcej korzyści.

Moja propozycja to: zróbmy rzetelne testy, by pokryć kompleksowo nasze przypadki użycia (ang.UC – *use cases*) w stylu „end-to-end", implementując testy modułowe w oparciu o dokumentację, ze szczególnym uwzględnieniem wyszczególnionych w niej wyjątków. Statystyki uzupełnijmy testami jednostkowymi, jawnie separując te mało wartościowe (nastawione na pokrycie) od tych dających miarodajne wyniki dla nas – kowali danego rozwiązania.

PRZYKŁAD

Strategia dla naszej aplikacji

Kluczowymi punktami naszej aplikacji są oczywiście same silniki zdarzeń, a głównymi przypadkami użycia są: akceptacja nowego połączenia oraz odpowiedź echo, więc to na ich testowaniu należy się skupić.

Przy tak zdefiniowanych celach testowych należy się zastanowić, co i jakie obiekty mają największy wpływ na nasze UC, i jednocześnie czym będziemy sterować. Odpowiedź kryje się właściwie na diagramie klas naszego rozwiązania – są to obiekty najbardziej wysunięte na brzeg diagramu nie posiadające żadnych zależności (w mojej nomenklaturze nazywam je klasami liściowymi). W tym wypadku są to klasy: `TcpSocket` oraz `Epoll`. Świadomie pomijam klasę `KeyboardSocket` z uwagi na to, że nie należy ona do głównych UC zdefiniowanych w celach.

Ostatnie pytanie, które prawdopodobnie siedzi w głowie, to: jak sterować zachowaniem wyodrębnionych klas? Tu z pomocą przychodzi nam znów teoria testów developerskich. Rzecz tkwi w tym, że w naszych testach nie wykorzystamy prawdziwych obiektów klas `Epoll` i `TcpSocket`, a obiekty klas, które będą nam pomagały skutecznie je udawać. Do dyspozycji mamy właściwie trzy typy takich obiektów:

- *dummy* – tzw. „głupia zaślepka" będąca trywialną implementacją danego interfejsu, posiada z góry przewidziane zachowanie (jedna wartość zwracana);
- *stub* – tzw. „proteza", to prosta implementacja danego interfejsu z ograniczonym sterowaniem (najczęściej kontrola wartości zwracanej);
- *mock* – tzw. „inteligentna zaślepka", najczęściej implementowana przy pomocy jakiegoś frameworka (FW) (np. Gmock), posiada pełną kontrolę nie tylko wartości zwracanej, ale także możliwości: sprawdzenia ilości wywołań metod, kontroli i przechwytywania argumentów tych metod itp.

Patrząc na powyższe (uproszczone) definicje i stawiając je obok naszych celów, musimy skłonić się ku ostatniemu rodzajowi obiektów zaślepiających. Jest to podyktowane przede wszystkim zdaniem traktującym o obiektach posiadających największy wpływ na nasze UC, których zachowaniem będziemy chcieli sterować.

Dobór narzędzi

Zasadniczą sprawą przed implementacją, a po jasno określonej strategii testowania, jest dobór odpowiednich narzędzi. Nie jest to trywialna sprawa, bo lista bibliotek do testów developerskich dla języka C++ jest dość długa. Jedno z ciekawszych zestawień można znaleźć pod adresem: http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C.2B.2B.

Na wyróżnienie w mojej opinii zasługuje otwarty i darmowy FW Google Test (Gtest), wzmocniony rozszerzeniem Google C++ Mocking Framework (Gmock). Gtest posiada proste i czytelne API, a dzięki przejrzystemu wyjściu łatwo go zintegrować w pętli ciągłej integracji (ang. *continuous integration loop*). Gmock natomiast to FW znacznie ułatwiający pisanie mocków, dostarczający całą gamę mechanizmów pozwalających na pełną kontrolę i sterowanie zachowaniem obiektu. Na te właśnie postawiłem w przykładzie poniżej.

Dla wzmocnienia efektu warto zastanowić się nad wybraniem narzędzi dla sprawdzenia podstawowych statystyk pokrycia kodu testami.

Jest przynajmniej kilka narzędzi świetnie się do tego nadających, m.in. gcov, lcov, trucov. O każdym z nich można znaleźć mnóstwo informacji w Internecie, ja zdecydowałem się na wykorzystanie lcov z uwagi na: wsparcie od strony kompilatora (jest dedykowany dla GCC) oraz statystyki, jakie zbiera domyślnie (procent pokrytych linii oraz możliwych gałęzi), oraz genhtml, który wyjście z lcov zamienia na przyjemną stronę HTML.

Implementacja

Ponieważ nasza aplikacja to już spory kawałek kodu, powyższą strategię prezentuję poniżej jedynie dla silnika HS/HA, a po jego zrozumieniu każdy inny będziesz mógł wykonać sam.

Na początek kod naszych inteligentnych zaślepek:

Listing 1. Implementacja klas typu mock dla obiektów TcpSocket oraz Epoll

```
class TcpSocketMock: public TcpSocket {
public:
  typedef std::shared_ptr<TcpSocketMock> Ptr;
  TcpSocketMock(int);
  virtual ~TcpSocketMock();
  MOCK_CONST_METHOD0(getDescriptor, Descriptor(void));
  MOCK_CONST_METHOD0(setNonBlocking, void(void));
  MOCK_CONST_METHOD1(read, void(std::string&));
  MOCK_CONST_METHOD1(write, void(const std::string&));
  MOCK_METHOD1(bind, void(int));
  MOCK_CONST_METHOD0(listen, void(void));
  MOCK_CONST_METHOD0(accept, TcpSocket::Ptr(void));
};

TcpSocketMock::TcpSocketMock(int p_fd)
  : TcpSocket(p_fd, sockaddr(), sizeof(struct sockaddr))
{
}

TcpSocketMock::~TcpSocketMock()
{
}

class EpollMock: public Epoll {
public:
  typedef std::shared_ptr<EpollMock> Ptr;
  EpollMock();
  virtual ~EpollMock ();
  MOCK_CONST_METHOD2(add, void(Socket::Ptr, const EventTypes&));
  MOCK_CONST_METHOD2(modify, void(Socket::Ptr, const
  EventTypes&));
  MOCK_CONST_METHOD1(remove, void(Socket::Descriptor));
  MOCK_CONST_METHOD1(wait, void(Events&));
  MOCK_CONST_METHOD0(interrupt, void(void));
};

EpollMock::EpollMock()
  : Epoll()
{
}

EpollMock::~EpollMock()
{
}
```

To właściwie trywialne deklaracje klas `TcpSocketMock` i `EpollMock`! Przy pomocy rodziny makr `MOCK_CONST_METHOD*` deklarujemy (i zarazem implementujemy) metody „constowe", których zachowaniem będziemy chcieli sterować (* jest liczbą argumentów danej metody). Odpowiednikiem rodziny makr dla metod „nie-constowych" jest rodzina `MOCK_METHOD*`.

Zwróćmy uwagę, że podstawą implementacji tych zaślepek jest dziedziczenie – zaślepka dziedziczy interfejs, który ma zastąpić. Znaczy to ni mniej, ni więcej, że przeciążane metody muszą być zadeklarowane jako `virtual` w klasie bazowej, a cały proceder takiego wypełniania oparty jest na polimorfizmie.

Pytanie, które teraz pewnie tkwi w głowie, to: „I co, to już teraz będę miał pełną kontrolę nad tym, jak moje zaślepki będą się zachowywać?". Odpowiedź jest jedna: tak! Czas więc pokazać kod testu:

Listing 2. Klasa testowa wraz z przykładowym przypadkiem testowym dla silnika HSHA

```
class HsHaTests: public testing::Test {
protected:
  virtual void SetUp();
  virtual void TearDown();
};

using namespace ::testing;

void HsHaTests::SetUp()
{
}

void HsHaTests::TearDown()
{
}

TEST_F(HsHaTests, echoSuccess)
{
  int listenerFd = 5;
  int clientFd = 6;
  int port = 5555;
  std::string msg = "msg\n";

  Epoll::EventTypes ets;
  ets.insert(Epoll::EventType::In);
  Epoll::Event sev(ets, listenerFd);
  Epoll::Event cev(ets, clientFd);
  Epoll::Events ses;
  Epoll::Events ces;
  ses.push_back(sev);
  ces.push_back(cev);

  EpollMock::Ptr e(new EpollMock());
  TcpSocketMock::Ptr l(new TcpSocketMock(listenerFd));
  TcpSocketMock::Ptr c(new TcpSocketMock(clientFd));

  EXPECT_CALL(*l, bind(port)).Times(1);
  EXPECT_CALL(*l, listen()).Times(1);
```

```
    EXPECT_CALL(*l, setNonBlocking()).Times(1);
    EXPECT_CALL(*l, getDescriptor()).WillOnce(Return(listenerFd));
    EXPECT_CALL(*l, accept()).WillOnce(Return(c));

    EXPECT_CALL(*c, setNonBlocking()).Times(1);
    EXPECT_CALL(*c, getDescriptor()).WillRepeatedly(Return(client
    Fd));
    EXPECT_CALL(*c, read(_)).WillOnce(SetArgReferee<0>(msg));
    EXPECT_CALL(*c, write(msg)).Times(1);

    EXPECT_CALL(*e, add(_, _)).Times(2);
    EXPECT_CALL(*e, wait(_)).Times(3)
      .WillOnce(SetArgReferee<0>(ses))
      .WillOnce(SetArgReferee<0>(ces))
      .WillOnce(Throw(std::runtime_error("error")));
    EXPECT_CALL(*e, modify(_, _)).Times(2);

    EpollED::Ptr eed(new EpollED(e));
    HSHA::Ptr hsha(new HSHA(eed, 1000));
    ListenerES::Ptr les(new ListenerES(l, port));
    SEH::Ptr seh(new AcceptorSEH(les, *hsha));
    AEH::Ptr aeh;
    HSHA::Handlers ah(seh, aeh);

    hsha->add(ah);

    EXPECT_THROW(hsha->eventLoop(), std::runtime_error);
}
```

Na początek weźmy klasę HhHaTests. To klasa, która zawierać będzie przypadki testowe. Na szczególną uwagę zasługują w niej metody SetUp i TearDown. Pierwsza będzie uruchamiana zawsze przed każdym testem, natomiast druga zawsze po każdym teście (to gwarantuje nam Gtest). W moim wypadku obie mają pustą implementację, i można by ich właściwie nie implementować, ale zrobiłem to, byś wiedział, że są, i mógł sobie wyobrazić, do czego je wykorzystać (bywają naprawdę przydatne).

Linia TEST_F(HsHaTests, echoSuccess) definiuje przy użyciu makra TEST_F przypadek testowy echoSuccess w klasie testów HsHaTests.

Część testu zaznaczona na pomarańczowo to deklaracje i definicje zmiennych, które będą zwracane przez nasze inteligentne zaślepki (zwróćmy uwagę na zmienną c – to jest właśnie obiekt zaślepki, za chwilę się dowiemy, dlaczego jest również na pomarańczowo).

Zielonym kolorem zaznaczyłem to, co w pierwszych częściach serii znajdowało się w funkcji main programu (produkcja i orkiestracja głównych obiektów), zauważmy, że różni się to właściwie tylko dwoma obiektami: zamiast obiektu TcpSocket tworzony jest obiekt TcpSocketMock i zamiast obiektu Epoll produkowany jest EpollMock, reszta pozostała bez zmian.

Fioletowym kolorem zaznaczyłem użycie makra EXPECT_CALL, którym określamy oczekiwania wobec danej zaślepki i jej konkretnej metody.

Na szczególną uwagę zasługują tutaj dwie:

```
EXPECT_CALL(*l, accept()).WillOnce(Return(c));
```

Linia ta znaczy: oczekuję, że metoda accept na obiekcie l zostanie wywołana raz, i ma zwrócić obiekt c (c to nasz TcpSocketMock i właśnie dlatego był zaznaczony na pomarańczowo – bo jest zwracany przez l, czyli naszą zaślepkę gniazda serwerowego).

```
EXPECT_CALL(*e, wait(_)).Times(3)
    .WillOnce(SetArgReferee<0>(ses))
    .WillOnce(SetArgReferee<0>(ces))
    .WillOnce(Throw(std::runtime_error("error")));
```

Ta linia z kolei mówi: spodziewam się na obiekcie e (EpollMock) trzech wywołań funkcji wait ('_' - znaczy, że nie interesuje mnie zawartość argumentu tej metody, FW daje możliwość ich sprawdzania, a zainteresowanych, jak to robić, odsyłam do dokumentacji FW). Za pierwszym razem funkcja wait ma ustawić pierwszy (o indeksie 0) argument tej metody, który jest referencją, tak by odnosił się do obiektu ses (zdarzenia na gnieździe serwerowym), a za drugim do obiektu ces (zdarzenia na gnieździe klienckim). Za trzecim razem funkcja ma rzucić wyjątek (zaraz się dowiemy dlaczego).

Ostatnią interesującą, zaznaczoną na szaro, linią jest:

```
EXPECT_THROW(hsha->eventLoop(), std::runtime_error);
```

Jest to uruchomienie pętli obsługi zdarzeń naszego silnika, które spowoduje lawinę uruchomień poszczególnych funkcji w naszych zaślepkach, a te na podstawie wyżej zdefiniowanych oczekiwań zwrócą/ustawią odpowiednio nakreślone przez nas wartości. Samo EXPECT_THROW w tej linii znaczy, że spodziewamy się zakończenia funkcji eventLoop na obiekcie hsha wyjątkiem std::runtime_error, dokładnie tego, który za trzecim razem rzuca funkcja wait z EpollMock – jest to najwygodniejsze przerwanie właściwie wiecznej funkcji eventLoop. Sprytne, prawda? Jest to gotowy test modułowy, na bazie którego możemy tworzyć kolejne testy, sprawdzając kod i jego obsługę błędów, uzupełniając przy tym brakujące pokrycie kodu. W dowolnej chwili możemy także taki test łatwo rozwinąć i zrobić prawdziwy test wydajnościowy dla naszej aplikacji.

Budowanie, uruchomienie i prezentacja wyników

Na potrzeby automatyzacji budowy programu testowego napisałem taki oto makefile:

Listing 3. Makefile programu testowego

```
CFLAGS = -std=c++11 -Wall -g \
         -fprofile-arcs -ftest-coverage \
         -I ../mocks -I ../.. \
         -I ../../IO -I ../../EchoResponder \
         -I ../../EchoResponder/EventDemultiplexer \
         -I ../../EchoResponder/EventSources
LDFLAGS = -lgcov -lgtest -lgmock -lgmock_main -lpthread
SOURCES = ../../IO/Epoll.cpp \
          ../../IO/Socket.cpp \
          ../../IO/TcpSocket.cpp \
          ../../Threading/ThreadPool.cpp \
          ../../EventEngines/EventDemultiplexer.cpp \
          ../../EventEngines/EventSource.cpp \
          ../../EchoResponder/EventDemultiplexer/EpollED.cpp \
          ../../EchoResponder/EventSources/SocketES.cpp \
          ../../EchoResponder/EventSources/ListenerES.cpp \
          ../../EchoResponder/EventSources/MessageES.cpp \
          ../../EchoResponder/EventHandlers/AcceptorSEH.cpp \
          ../../EchoResponder/EventHandlers/EchoResponderAEH.cpp
\
          ../../EchoResponder/EventHandlers/EchoResponderSEH.cpp
\
          ../mocks/EpollMock.cpp \
          ../mocks/TcpSocketMock.cpp \
          HsHaTests.cpp
NAME = prog
OBJECTS = $(SOURCES:.cpp=.o)
GCNOS = $(SOURCES:.cpp=.gcno)
GCDAS = $(SOURCES:.cpp=.gcda)

all: $(NAME)

$(NAME): $(OBJECTS)
  $(CXX) $(OBJECTS) $(LDFLAGS) -o $(NAME)

%.o: %.cpp
  $(CXX) -c $(CFLAGS) $< -o $@

run:
  ./$(NAME)

coverage: all run
  mkdir -p results/html
  lcov -b . -d ../.. -c -o results/coverage.info
  lcov -r results/coverage.info /usr/\* tst/* -o results/
  coverage.info
  genhtml results/coverage.info --output-directory results/html/
  --ignore-errors source

clean:
  rm -f $(OBJECTS)
  rm -f $(GCNOS)
  rm -f $(GCDAS)
  rm -f $(NAME)
  rm -rf results
```

Pierwsza sprawa to dołożenie odpowiednich flag kompilacji: -fprofile-arcs oraz -ftest-coverage. W wyniku kompilacji z tymi flagami obok plików źródłowych pojawią się pliki *.gcno (pliki do śledzenia przebiegu programu). Druga sprawa to biblioteki dla linkera:

» -lgcov – wymagana do badania pokrycia z użyciem gcov i lcov

- `-lgtest` – biblioteka FW Gtest
- `-lgmock` – biblioteka FW Gmock
- `-lgmock_main` – dołączenie implementacji domyślnej funkcji `main` dla progamu zawierającego nasze testy
- `-lpthread` – standardowa biblioteka wielowątkowości w systemach UNIXowych (wymagana przez Gmock).

Na uwagę zasługuje także element (ang. *target*) `coverage`:

```
coverage: all run
  mkdir -p results/html
  lcov -b . -d ../.. -c -o results/coverage.info
  lcov -r results/coverage.info /usr/\* tst/* -o results/
  coverage.info
  genhtml results/coverage.info --output-directory results/html/
  --ignore-errors source
```

Element ten zależy od elementu `all`, który buduje nasz test, oraz elementu `run`, który uruchamia test. Po uruchomieniu testu obok plików źródeł powstaną dodatkowo odpowiadające im pliki `*.gcda`, które będą zawierać metadane/statystyki dla każdej z linii kodu (m.in. informację, ile razy dana linia w danym pliku źródłowym została odwiedzona).

Zaznaczona na czerwono część odpowiada za uruchomienie narzędzia lcov na głównym katalogu projektu. Lcov odszuka w całej ścieżce wszystkie pliki `*.gcda` i na ich podstawie utworzy tekstowy raport z pokrycia kodu testami w pliku `results/coverage.info`.

Ponieważ lcov działa rekursywnie, weźmie wszystkie pliki łącznie z kodem testującym i bibliotekami standardowymi, więc sekcją zaznaczoną na niebiesko usuwamy dane o tych źródłach z raportu.

W sieci

- Kod źródłowy aplikacji z części I: https://github.com/RomanUlan/ReactorBase
- Kody źródłowe aplikacji z części II: https://github.com/RomanUlan/ReactorInheritance https://github.com/RomanUlan/ReactorTemplates
- Kody źródłowe aplikacji z części III: https://github.com/RomanUlan/ThreadPool
- Kod źródłowy aplikacji z części IV: https://github.com/RomanUlan/HalfSyncHalfAsync
- Kod źródłowy aplikacji z części V: https://github.com/RomanUlan/AllInOne
- Kod źródłowy aplikacji prezentowanej w niniejszej części: https://github.com/RomanUlan/AllInOneTested

Zielona część to zamiana niewygodnego (przynajmniej dla ludzi) tekstowego pliku raportu z lcov na ergonomiczną stronę HTML, której wygląd przedstawia Rysunek 1, i stanowi zwieńczenie naszych starań:

Jak widać, w statystykach już jednym, stosunkowo prostym testem udało nam się pokryć lwią część naszego kodu, a co najważniejsze, to jest to kompleksowy test modułowy.

Zdziwieniem nie może być tu fakt, że komponent `IO` właściwie nie jest pokryty, ponieważ jego główne obiekty zaślepialiśmy. Co należałoby zrobić w sytuacji, gdy klient zażyczy sobie pokrycie i tego komponentu? Nic prostszego – jego klasy pokryłbym osobnymi testami jednostkowymi.

ZAKOŃCZENIE SERII

Podsumujmy całość tego, co wydarzyło się przez te sześć odcinków serii. Przedstawiłem trzy różne wzorce silników obsługi zdarzeń, zaczynając od najprostszego, jednowątkowego *Reactora*, na którego podstawie pokazałem, jak można odseparować logikę wzorca (demultipleksacja i obsługa zdarzeń) od szczegółów domenowych (gniazda i epoll).

W kolejnych numerach omawiałem dwa kolejne podejścia: *Thread Pool* oraz *Half-Sync/Half-Async*. Są one wielowątkowe, znacznie bardziej skalowalne, lecz też dużo bardziej złożone, przez co bardziej podatne na błędy.

Kolejnym etapem była integracja przedstawionych trzech rozwiązań w jedną konfigurowalną i skalowalną aplikację, przy czym wykorzystałem inne użyteczne wzorce, jak choćby *Abstract Factory*.

Całość zakończyliśmy implementacją testów developerskich, gdzie ewidentnie widać, że dzięki dobrze zorganizowanej architekturze obiektowej aplikacji pisanie testów developerskich to przysłowiowa bułka z masłem (oczywiście jeśli się zna przydatne narzędzia). Przy takim podejściu same testy nie są, jak to często się zdarza, problemem, a możemy im w pełni ufać i faktycznie będą nas chronić w fazie utrzymania.

Dodatkowy benefit, który przewijał się przez całą serię, to tak naprawdę systemowe programowanie w Linux. To system, który obecnie święci triumfy i jest stosowany w znacznej większości komercyjnych rozwiązań serwerowych, a komunikacja sieciowa jest ich podstawą.

Na koniec jeszcze raz gorąco zachęcam do korzystania z dobrodziejstw, jakie niesie ze sobą szeroka wiedza z inżynierii oprogramowania, do projektowania rozwiązań na podstawie sprawdzonych mechanizmów zapisanych we wzorcach oraz do pisania kapitalnych testów developerskich, dzięki którym czas maintenance'u może będzie choć trochę mniej smutny.

Dziękuję za uwagę i do zobaczenia wkrótce!

LCOV - code coverage report

Current view:	top level		Hit	Total	Coverage
Test:	coverage.info	**Lines:**	273	418	65.3 %
Date:	2015-06-05	**Functions:**	83	113	73.5 %

Directory	Line Coverage		Functions	
EchoResponder/EventDemultiplexer	79.3 %	23 / 29	75.0 %	6 / 8
EchoResponder/EventHandlers	95.6 %	43 / 45	100.0 %	14 / 14
EchoResponder/EventSources	100.0 %	40 / 40	92.3 %	12 / 13
EventEngines	81.0 %	64 / 79	77.4 %	24 / 31
IO	28.3 %	45 / 159	39.4 %	13 / 33
Threading	87.9 %	58 / 66	100.0 %	14 / 14

Generated by: LCOV version 1.10

Rysunek 1. Raport HTML wygenerowany przez narzędzia lcov i genhtml po wykonaniu omawianego testu modułowego

Roman Ulan roman.ulan@gmail.com

Senior Software Architect w Tieto Poland Sp. z o.o. Trener C++ w Bottega IT Solutions. Specjalizuje się w rozwiązaniach backend'owych w języku C++. Entuzjasta czystego i testowalnego kodu osadzonego w przemyślanej architekturze opartej na sprawdzonych wzorcach. Interesuje go programowanie równoległe i rozproszone oraz zagadnienia „software design".