Obiekt pozorny. A może coś innego?

Kierunek rozwoju testów jednostkowych

Mateusz Bukowski i Paweł Paterek

Mateusz Bukowski jest absolwentem Akademii Górniczo Hutniczej w Krakowie, kierunku Informatyka na Wydziale Elektrotechniki, Automatyki, Informatyki i Elektroniki. Obecnie jest pracownikiem Motorola Polska Electronics Sp. z o.o. gdzie zajmuje się testowaniem systemu bezpieczeństwa publicznego TETRA. Jego zainteresowania to żeglarstwo, cross country i rozwiązywanie łamigłówek logicznych.

Paweł Paterek jest absolwentem Wydziału Elektrotechniki, Automatyki, Informatyki i Elektroniki Akademii Górniczo-Hutniczej w Krakowie, kierunek Elektronika i Telekomunikacja. Obecnie jest pracownikiem firmy Motorola Polska Electronics Sp. z o.o., w której zajmuje się testowaniem oprogramowania systemów czasu rzeczywistego oraz systemów wbudowanych dla stacji bazowych w standardzie TETRA. Jego dziedziną zainteresowań jest modelowanie i ocena efektywności pracy sieci komputerowych.

Obiekt pozorny. A może coś innego?
Kierunek rozwoju testów jednostkowych

Mateusz Bukowski i Paweł Paterek

W dzisiejszym świecie, dostarczenie produktu na czas, nie jest już najważniejszym celem. Dużo większe znaczenie, zaczyna odgrywać jakość dostarczanego oprogramowania. Ponieważ testowanie zabiera coraz więcej czasu i wysiłku, niezbędne jest uproszczenie technik testerskich przy jednoczesnym zachowaniu poziomu znajdowanych defektów. W tym artykule zapoznamy się z obiektami pozornymi i zaślepkami, które wnoszą nową jakość do testów oraz pokażemy, dlaczego warto korzystać z tych pierwszych.

Zanim przedstawimy nowe sposoby testowania, przypomnijmy, czego dotyczą testy jednostkowe oraz czym jest programowanie sterowane testami.

Test jednostkowy (Unit test) to procedura mająca na celu walidację poprawności działania danej jednostki kodu źródłowego. Przez jednostkę kodu źródłowego rozumiemy najmniejszą i możliwą do przetestowania część oprogramowania. W programowaniu obiektowym (ang. *Object Oriented Programming*) taką jednostkę kodu stanowi klasa [8]. Testy jednostkowe odgrywają znaczącą rolę w procesie wytwarzania oprogramowania. Umożliwiają one wykrycie wielu błędów w kodzie już na etapie implementacji lub we wstępnej fazie testów. Pisane są najczęściej przez te same osoby, które zajmują się tworzeniem danego fragmentu oprogramowania.

Testy jednostkowe z założenia są proste i szybkie w uruchomieniu. Testują wyodrębnioną część funkcjonalności w całkowitej izolacji od reszty systemu. Pozwala to na zautomatyzowanie znaczącej części procesu testowania. Tym samym pozwala to na częste sprawdzanie czy wprowadzane zmiany w istniejącym kodzie nie powodują błędów.

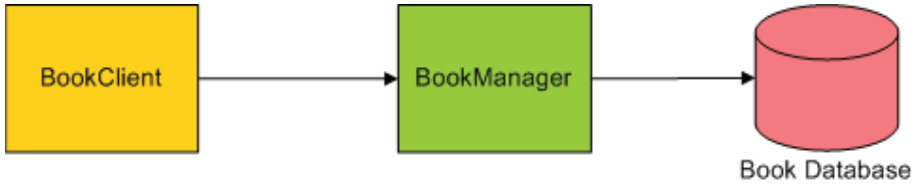
Testy jednostkowe nie weryfikują wszystkich wymagań funkcjonalnych danego systemu. Jest to zadanie testów akceptacyjnych wykonywanych przez odbiorcę oprogramowania [4],[10]. Ten rodzaj testów nie ma za zadania sprawdzać interakcji między różnymi obiektami, co wynika wprost z ich definicji. Nie mogą wobec tego zastąpić testów integracyjnych i systemowych w procesie testowania. Są jednak ich bardzo dobrym uzupełnieniem.

Bardzo często testowane klasy posiadają powiązania do innych klas. Utworzenie referencji do takich obiektów w środowisku testowym niejednokrotnie jest niemożliwe. Jednym z rozwiązań tego problemu są właśnie obiekty zastępcze powszechnie zwane obiektami pozornymi (mock). Są to specjalnie przygotowane przez nas implementacje interfejsów mogące zastąpić problematyczne części kodu oraz ułatwić wykonanie testów.

Programowanie sterowane testami (Test Driven Develomnent TDD) zakłada, że żaden fragment oprogramowania nie powstanie zanim nie napiszemy do niego odpowiedniego testu. Pisany kod musi spełnić wymagania testu, a następnie można przeprowadzić na nim odpowiedni refactoring[1], który pozwoli na ustalenie ostatecznego nazewnictwa metod czy obiektów. Stosowanie tej metodologii ma różne zalety, m.in. pisząc pełny zbiór testów przed napisaniem właściwego kodu piszemy zarazem specyfikację jego działania, po czym sama walidacja jest już automatyczna. Ponieważ przed napisaniem kodu wszystkie testy kończą się błędem, a w miarę jak nasz kod spełnia coraz więcej wymagań tych błędów jest mniej, to otrzymujemy tym samym miarę, w jakim stopniu nasz kod realizuje założenia projektowe [6],[1].

TDD pozwala małymi krokami (poprzez napisanie najmniejszej części kodu, która powoduje pomyślne przejście testu) na sukcesywne i skuteczne realizowanie danego projektu. Jednocześnie tak napisany kod jest mniej skomplikowany, dostajemy wysokie pokrycie testami, a odnajdywanie błędów staje się znacznie prostsze. Jednak, kiedy test dotyczy kodu odwołującego się do baz danych, zdalnych obiektów lub połączeń sieciowych napisanie go może być bardzo problematyczne. Chcąc jednak być w zgodzie z koncepcją metodologii TDD i testować również takie obiekty, musimy zastąpić obiekty, do których odwołuje się testowany kod i zaimitować ich obecność w testowanym kodzie. Jednym z możliwych sposobów testowania takiego kodu jest wykorzystanie wspomnianych wyżej lekkich obiektów pozornych lub dużo cięższych zaślepek [11],[14].

W naszych rozważaniach będziemy posługiwać się następującym przykładem. Mamy bazę danych książek. Każdy element opisany jest następującymi parametrami: autor, tytuł oraz identyfikator. Ponadto w bazie przechowywana jest również liczba dostępnych egzemplarzy każdej książki. Z bazą danych możemy połączyć się poprzez BookManager'a, z którego korzysta BookClient.



Rys. 1 Schemat systemu zamawiania książek

```
public interface BookManager {

    public boolean connect();

    public boolean disconnect();

    public TreeMap<Book, Integer> bookList();

    public boolean order(BookOrder bookOrder) throws BookNotFoundException,
NotEnoughBooksException;

}
public class BookClient {

    public static final String CONNECT_ERROR = "Database connect error";

    public static final String DISCONNECT_ERROR = "Database disconnect error";

    public static final String BOOK_NOT_FOUND_ERROR = "Book not found error";

    public static final String NOT_ENOUGH_BOOKS_ERROR = "Not enough books
error";

    public static final String AVAILABLE_BOOKS = "Available books:";

    public static final String ORDERED_BOOKS = "Ordered books:";

    public static final String BOOK_ORDER_FAIL = "Book order failed error";

    private BookManager bookManager;

    private String message;

    public BookClient() {
        bookManager = new DefaultBookManager();

        message = "";
    }

    public BookClient(BookManager bookManager) {
        this.bookManager = bookManager;

        message = "";
    }

    public String getMessage() {
```

```
        return message;
    }

    public void getBookList() {
        if (!bookManager.connect()) {
            message = CONNECT_ERROR;
            return;
        }

        message = AVAILABLE_BOOKS + bookListToString(bookManager.bookList());

        if (!bookManager.disconnect()) {
            message = DISCONNECT_ERROR;
            return;
        }
    }

    public void orderBooks(BookOrder bookOrder) {
        if (!bookManager.connect()) {
            message = CONNECT_ERROR;
            return;
        }

        try {
            if (bookManager.order(bookOrder)) {
                message = ORDERED_BOOKS +
bookListToString(bookOrder.getBookList());
            } else {
                message = BOOK_ORDER_FAIL;
            }
        } catch (BookNotFoundException e) {
            message = BOOK_NOT_FOUND_ERROR;
        } catch (NotEnoughBooksException e) {
            message = NOT_ENOUGH_BOOKS_ERROR;
        }

        if (!bookManager.disconnect()) {
            message = DISCONNECT_ERROR;
            return;
        }
    }

    public static String bookListToString(TreeMap<Book, Integer> bookList) {
        StringBuilder result;

        result = new StringBuilder();

        for (Book book: bookList.keySet()) {
            result.append("\n");
            result.append(book.getId());
            result.append(" ");
            result.append(book.getAuthor());
            result.append(" ");
            result.append(book.getTitle());
            result.append(" ");
            result.append(bookList.get(book));
        }

        return result.toString();
    }
```

```
}
```

Implementacja pozostałych klas: *Book, BookOrder, DefaultBookManager, NotEnoughBooksException* oraz *BookNotFoundException* nie jest istotna w naszych rozważaniach. Niezbędna jest implementacja funkcji *equals*, *hashCode* oraz *compareTo* w klasie *Book*, ponieważ będziemy używać tych obiektów jako kluczy w *TreeMap'ie*.

Testowaną klasą jest BookClient. BookManager jest interfejsem, który będą implementować konkretne obiekty. Domyślnie jest to klasa DefaultBookManager. Na potrzeby testów dodaliśmy drugi konstruktor, w którym przekazujemy specjalnie spreparowany *BookManager*. Inną stosowaną praktyką jest również korzystanie z metod 'set'.

Wspomniane wcześniej obiekty pozorne nie są jedyną możliwością na zastąpienie prawdziwych obiektów w obiekcie testowanym. Często też bywają mylone z podobnymi do nich w zastosowaniu, lecz różniącymi się funkcjonalnie **obiektami atrapami** (ang. *dummy objects*), **obiektami falsyfikatami** (ang. *fake objects*) oraz **zaślepkami** (ang. *stubs*).

Obiekty atrapy przekazywane są do obiektu testowanego, ale nigdzie nie są używane i zazwyczaj służą jedynie do wypełnienia listy parametrów testowanej metody.

Falsyfikaty (w żargonie informatycznym często nazywane **fake'ami**) posiadają namiastkę działającej implementacji, najczęściej stworzoną ręcznie w ramach kodu testów (na przykład w postaci klasy anonimowej). Implementacja ta jest zazwyczaj skrócona do minimum, co sprawia, że nie nadają się one do użycia w prawdziwym kodzie. Wadami falsyfikatów są: czas poświęcony na ich stworzenie, zwiększenie zawartości całego projektu, a także konieczność ich utrzymywania, w trakcie zmiany interfejsów kodu, z których one korzystają.

Zaślepki mogą być napisane ręcznie lub generowane automatycznie i są najczęściej stosowane wtedy, gdy kod nie jest znany, nie jest gotowy na etapie testowania danego obiektu lub, gdy nie ma możliwości dostępu do danej części kodu, co czasami znacznie ułatwia i przyspiesza proces tworzenia oprogramowania [2], [5]. Zaślepki symulują zachowanie prawdziwego kodu. W naszym przykładzie będzie łączył się z prawdziwą bazą danych. Innym razem w celu dogłębnego przetestowania z użyciem tej techniki musielibyśmy uruchomić serwer http. Zaślepki są bardzo kosztowne w utrzymaniu. Jeśli są stosowane we wczesnych fazach testowania mogą później służyć jako zalążki dla implementacji prawdziwych klas.

Obiektami pozornymi zajmiemy się dokładniej w dalszej części.

Implementacja *BookManager'a* może być jednym z powyższych typów: falsyfikat, zaślepka, obiekt pozorny.

Przyjrzyjmy się teraz konkretnym falsyfikatom i zaślepkom oraz odpowiadającym im testom jednostkowym.

```
public class FakeBookManager implements BookManager {

    private TreeMap<Book, Integer> bookList;

    private int call;

    public FakeBookManager() {
        bookList = new TreeMap<Book, Integer>();

        bookList.put(new Book(101, "Ernest Hemingway", "The Old Man and the
Sea"), 3);
        bookList.put(new Book(102, "Gabriel Garcia Marquez", "One Hundred
Years of Solitude"), 5);
        bookList.put(new Book(103, "Joanne Kathleen Rowling", "Harry Potter
and the Deathly Hallows"), 27);

        call = 0;
    }

    public boolean connect() {
        return true;
    }

    public boolean disconnect() {
        return true;
    }

    public TreeMap<Book, Integer> bookList() {
        return bookList;
    }

    public boolean order(BookOrder bookOrder) throws BookNotFoundException,
NotEnoughBooksException {
        call++;

        switch (call % 8) {
            case 0:
            case 2:
            case 4:
                return true;

            case 1:
            case 3:
            case 5:
                return false;

            case 6:
                throw new BookNotFoundException();

            case 7:
                throw new NotEnoughBooksException();

            default:
                return true;
        }
    }
}
```

Jak widzimy nasz falsyfikat ma zaszytą namiastkę logiki. Wiedząc, który raz wywołujemy metodę *order()*, możemy sprawdzić czy *BookClient* zachowuje się w poprawny sposób.

```
public class BookClientTestFakeBookManager extends TestCase {

    private TreeMap<Book, Integer> bookList;

    private static final Book QUO_VADIS = new Book(1001, "Henryk Sienkiewicz",
"Quo Vadis");

    private static final Book PAN_TADEUSZ = new Book(2001, "Adam Mickiewicz",
"Pan Tadeusz");

    private static final Book CHLOPI = new Book(3001, "Wladyslaw Reymont",
"Chlopi");

    @Override
    protected void setUp() throws Exception {
        super.setUp();

        bookList = new TreeMap<Book, Integer>();

        bookList.put(QUO_VADIS, 2);
        bookList.put(PAN_TADEUSZ, 3);
        bookList.put(CHLOPI, 5);
    }

    public void testOrderGlobal() {
        BookClient client;
        FakeBookManager fake;
        BookOrder bookOrder;
        String message;

        fake = new FakeBookManager();

        client = new BookClient(fake);

        bookOrder = new BookOrder();
        bookOrder.addBook(CHLOPI);

        client.orderBooks(bookOrder);
        message = BookClient.BOOK_ORDER_FAIL;
        assertEquals(message, client.getMessage());

        client.orderBooks(bookOrder);
        message = BookClient.ORDERED_BOOKS +
BookClient.bookListToString(bookOrder.getBookList());
        assertEquals(message, client.getMessage());

        client.orderBooks(bookOrder);
        client.orderBooks(bookOrder);
        client.orderBooks(bookOrder);

        client.orderBooks(bookOrder);
        message = BookClient.BOOK_NOT_FOUND_ERROR;
        assertEquals(message, client.getMessage());

        client.orderBooks(bookOrder);
        message = BookClient.NOT_ENOUGH_BOOKS_ERROR;
        assertEquals(message, client.getMessage());
```

```
    }
}
```

W przykładzie z zaślepką użyliśmy najprostszego rozwiązania. Jako bazy danych użyliśmy Accessa. Używanie tej aplikacji nie wymaga obszernej wiedzy z zakresu zarządzania bazami danych. Access jest łatwy do konfiguracji, prosty w obsłudze i świetnie nadaje się do celów testowych. Gdybyśmy chcieli symulować serwer http nie musimy mieć postawionego Apache'a. Możemy użyć prostego symulatora Jetty, który jest prostą aplikacją javową.

```
public class StubBookManager implements BookManager {

    private Connection conn;

    private String url;

    private String username;

    private String password;

    public StubBookManager(String location) {
        url = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=" +
location;
        username = "";
        password = "";
    }

    public boolean connect() {
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            conn = DriverManager.getConnection(url, username, password);
        } catch (ClassNotFoundException e) {
            return false;
        } catch (SQLException e) {
            return false;
        }

        return true;
    }

    public boolean disconnect() {
        try {
            conn.close();
        } catch (SQLException e) {
            return false;
        }

        return true;
    }

    public TreeMap<Book, Integer> bookList() {
        TreeMap<Book, Integer> result;
        Statement st;
        ResultSet rs;
        Book book;
        int quantity;
```

```
        result = new TreeMap<Book, Integer>();

        try {
            st = conn.createStatement();

            rs = st.executeQuery("SELECT * FROM books");

            while (rs.next()) {
                book = new Book(rs.getInt("id"), rs.getString("author"),
rs.getString("title"));
                quantity = rs.getInt("quantity");

                result.put(book, quantity);
            }
        } catch (SQLException e) {
            return new TreeMap<Book, Integer>();
        }

        return result;
    }

    public synchronized boolean order(BookOrder bookOrder) throws
BookNotFoundException, NotEnoughBooksException {
        TreeMap<Book, Integer> bookList;
        Statement st;
        int quantity;

        bookList = bookList();

        for (Book book: bookOrder.getBookList().keySet()) {
            if (!bookList.containsKey(book)) {
                throw new BookNotFoundException();
            }

            quantity = bookList.get(book) - bookOrder.getBookList().get(book);
            if (quantity < 0) {
                throw new NotEnoughBooksException();
            }

            try {
                st = conn.createStatement();

                st.executeUpdate(
                        "UPDATE books" +
                        " SET quantity = " + quantity +
                        " WHERE id = " + book.getId() +
                        " AND author = '" + book.getAuthor() + "'" +
                        " AND title = '" + book.getTitle() + "'");
            } catch (SQLException e) {
                return false;
            }
        }

        return true;
    }
}
```

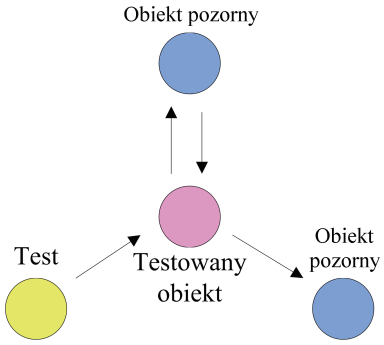
Do poprawnego uruchomienia poniższego testu konieczne jest dodanie katalogu ‘data’ z odpowiednią plikiem bazy danych ‘book.mdb’.

```
public class BookClientTestStubBookManager extends TestCase {

    public void testConnectOk() {
        BookClient client;
        StubBookManager stub;
        String message;

        stub = new StubBookManager("./data/book.mdb");

        client = new BookClient(stub);

        client.getBookList();

        message = BookClient.AVAILABLE_BOOKS;
        assertEquals(message, client.getMessage());
    }

    public void testConnectFail() {
        BookClient client;
        StubBookManager stub;
        String message;

        stub = new StubBookManager("dummy.mdb");

        client = new BookClient(stub);

        client.getBookList();

        message = BookClient.CONNECT_ERROR;
        assertEquals(message, client.getMessage());
    }
}
```

Obiekt pozorny (mock) jest imitacją stworzoną na potrzeby testu jednostkowego w celu sprawdzenia poprawności współpracy testowanego obiektu z jego najbliższym otoczeniem [9]. Bardziej formalne interpretacje tego pojęcia to:

a) obiekt pozorny jest pewną atrapą zastępującą prawdziwy obiekt, który jest niedostępny lub trudny do użycia w scenariuszu testowym
b) obiekt pozorny musi posiadać mechanizm do automatycznej walidacji ustawionych dla niego wcześniej oczekiwanych zachowań [4],[10].

Obiekty te powinny zawsze zwracać z góry określone dane, umożliwiając testowanie logiki aplikacji w jednoznaczny sposób, bez konieczności odwoływania się do prawdziwych obiektów, bądź baz danych. Ich logika oraz implementacja powinna być najprostsza jak to tylko możliwe oraz niezależna od innych obiektów pozornych, bądź fragmentów testowanej aplikacji. Mechanizm automatycznej walidacji powinien zadziałać w momencie wystąpienia błędu, umożliwiając szybkie przerwanie testu i łatwą lokalizację przyczyny problemu [4],[9].



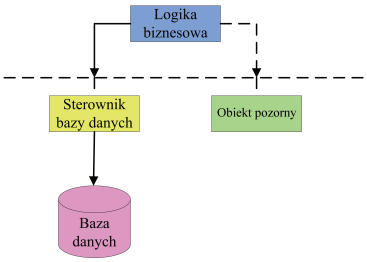
Rys. 2 Wymiana informacji Test – Obiekt – Obiekt Pozorny

Program zorientowany obiektowo jest siecią obiektów wzajemnie ze sobą współpracujących. Aby dobrze przetestować działanie obiektu, należy przetestować poprawność jego współpracy z obiektami, z którymi wymienia informacje w prawdziwej aplikacji. Ponieważ unit test polega na testowaniu obiektu w izolacji od innych obiektów istniejących w aplikacji, obiekty wymieniające informacje z obiektem testowanym muszą zostać podmienione na obiekty pozorne, patrz rys. 2 [6]. W chwili, kiedy na obiekcie zastępczym wystąpi inne wywołanie metody niż oczekiwane, lub wywołanie nastąpi z niewłaściwymi parametrami lub też, gdy wywołanie wystąpi, a nie było ono jawnie oczekiwane test jest automatycznie przerywany i jego wynik ustawiany jest na negatywny [7].

Powody, dla których użycie obiektów pozornych może być nieocenione, są przeróżne. Prawdziwy obiekt wykazuje zachowanie niedeterministyczne. Jest trudno konfigurowalny. Jego działanie jest znacznie wolniejsze (niepotrzebnie wydłużając test). Część jego zdarzeń jest trudna do wywołania (szczególnie sytuacje dotyczące występowania błędów). Prawdziwy obiekt może mieć lub stanowić interfejs użytkownika. Najważniejsze jest to, że prawdziwy obiekt może jeszcze nie być w ogóle stworzony lub test może chcieć wywołać na nim specyficzne zapytanie, które nie jest dostępne w prawdziwym obiekcie w danej chwili [13],[10].

Korzystanie z obiektów pozornych posiada charakterystyczny wzorzec, który można stosować podczas tworzenia testów. Najpierw tworzymy instancję obiektu pozornego, następnie ustawiamy ich stan (parametry i atrybuty używane przez testowany obiekt), a na koniec ustawiamy ich oczekiwane zachowania (oraz ewentualne wartości zwracane), które powinny być wywołane przez testowany obiekt. Testowany kod wywołujemy podając jako parametry wcześniej przygotowane obiekty zastępcze i sprawdzamy czy wszystkie wywołania, które się do nich odnoszą są zgodne z naszymi oczekiwaniami [3].

Praktycznym ułatwieniem stosowania tej techniki może być stworzenie interfejsu opisującego dany obiekt, a następnie implementacja tego interfejsu zarówno dla kodu, jak i w celu stworzenia obiektu zastępczego używanego podczas testu. Jednym z typowych przykładów zastosowania techniki obiektów pozornych, może być testowanie logiki biznesowej, odwołującej się do zdalnej bazy danych (rys. 3). Sterownik bazy danych implementuje najczęściej standardowy interfejs dostępu do bazy danych. Ten sam interfejs można użyć do stworzenia obiektu pozornego, dzięki czemu nie musimy mieć dostępu do prawdziwej bazy danych.



Rys. 3 Przykład zastosowania obiektu pozornego

Ze względu na sposób tworzenia (użytą technikę programowania) obiektów pozornych można podzielić na dwa rodzaje: obiektów pozornych statycznych oraz obiektów pozornych dynamicznych. Obiekty pozorne statyczne możemy tworzyć na dwa różne sposoby, poprzez zwykłą ręczną implementację dodatkowej klasy lub można je wygenerować automatycznie (na etapie kompilacji bądź uruchomienia kodu testu). Obiekty pozorne dynamiczne również można tworzyć na dwa sposoby: za pomocą interfejsu zwanego proxy lub za pomocą bardziej zaawansowanych technik programistycznych. Obiekty pozorne statyczne możemy użyć w przypadku pojedynczych testów, bądź niewielkich i niezbyt skomplikowanych projektów, natomiast w przypadku dużych projektów (duża liczba klas, które musielibyśmy napisać w przypadku pozorantów statycznych), gdzie najczęściej podmieniane obiekty mają rozbudowane funkcjonalności bardziej odpowiednie staje się użycie obiektów pozornych dynamicznych [12].

W odróżnieniu od technik opisanych powyżej obiekty pozorne są najczęściej automatycznie generowanymi obiektami zastępczymi z zaprogramowanymi wcześniej oczekiwaniami wywołań z obiektu testowanego, które są jednocześnie specyfikacją zachowań tego obiektu. Obiekty te mogą zapisywać wykonane wywołania z obiektu testowanego w celu porównania tej informacji z oczekiwaniami wynikającymi ze scenariusza testu. Z wszystkich wymienionych technik tylko obiekty zastępcze weryfikują zachowanie testowanego obiektu, a pozostałe weryfikują jedynie stany, w których znajduje się testowany obiekt [5]. Obiekty pozorne są najczęściej generowani dynamicznie w trakcie uruchamiania testów, a więc nie ma potrzeby tworzenia dodatkowych klas z kodem, a zarazem utrzymywania i aktualizacji ich kodu, co znacznie odróżnia je od pozostałych technik. Całość logiki obiektów zastępczych, a więc zbiór oczekiwanych zachowań jest trzymany razem z kodem testów [2].

Wszystkie przedstawione powyżej techniki mają też wspólne cechy, takie jak: możliwość eliminacji wielu zależności w kodzie, inaczej mówiąc możliwość testowania wybranych fragmentów kodu w izolacji, czyli bez środowiska, w którym ten kod pracuje, możliwość testowania scenariuszy hipotetycznych lub rzadko występujących w prawdziwym działaniu kodu, możliwość znacznego przyspieszenia samego procesu testowania, pozwalając tym samym na częste uruchamianie testów jednostkowych, a jednocześnie na realizację idei testów jednostkowych.

Poniżej przedstawiamy różne podejścia do obiektów pozornych. Na początek obiekt pozorny napisany od początku do końca przez nas.

```
public class MyMockBookManager implements BookManager {

    private TreeMap<Book, Integer> bookList;

    private boolean connect;

    private boolean disconnect;

    private int expectedOrderCount;

    private int orderCount;

    private int expectedConnectCount;

    private int connectCount;

    private int expectedDisconnectCount;

    private int disconnectCount;

    public MyMockBookManager() {
        bookList = new TreeMap<Book, Integer>();

        connect = false;
        disconnect = false;
        expectedOrderCount = 0;
        expectedConnectCount = 0;
        expectedDisconnectCount = 0;
        orderCount = 0;
        connectCount = 0;
        disconnectCount = 0;
    }

    public boolean connect() {
        connectCount++;

        return connect;
    }

    public boolean disconnect() {
        disconnectCount++;

        return disconnect;
    }

    public TreeMap<Book, Integer> bookList() {
        return bookList;
    }

    public boolean order(BookOrder bookOrder) throws BookNotFoundException,
NotEnoughBooksException {
        orderCount++;

        for (Book book: bookOrder.getBookList().keySet()) {
            if (!bookList.containsKey(book)) {
                throw new BookNotFoundException();
            }

            if (bookList.get(book) < bookOrder.getBookList().get(book)) {
                throw new NotEnoughBooksException();
```

```
            }
        }

        return true;
    }

    public void setConnect(boolean connect) {
        this.connect = connect;
    }

    public void setDisconnect(boolean disconnect) {
        this.disconnect = disconnect;
    }

    public void setBookList(TreeMap<Book, Integer> bookList) {
        this.bookList = bookList;
    }

    public void setExpectedConnectCount(int expectedConnectCount) {
        this.expectedConnectCount = expectedConnectCount;
    }

    public void setExpectedDisconnectCount(int expectedDisconnectCount) {
        this.expectedDisconnectCount = expectedDisconnectCount;
    }

    public void setExpectedOrderCount(int expectedOrderCount) {
        this.expectedOrderCount = expectedOrderCount;
    }

    public void verify() {
        if (expectedOrderCount != orderCount) {
            Assert.fail("Wrong number of performed orders");
        }

        if (expectedConnectCount != connectCount) {
            Assert.fail("Wrong number of performed connects");
        }

        if (expectedDisconnectCount != disconnectCount) {
            Assert.fail("Wrong number of performed disconnects");
        }
    }

}


public class BookClientTestMyMockBookManager extends TestCase {

    private TreeMap<Book, Integer> bookList;

    private static final Book QUO_VADIS = new Book(1001, "Henryk Sienkiewicz",
"Quo Vadis");

    private static final Book PAN_TADEUSZ = new Book(2001, "Adam Mickiewicz",
"Pan Tadeusz");

    private static final Book CHLOPI = new Book(3001, "Wladyslaw Reymont",
"Chlopi");

    @Override
```

```
    protected void setUp() throws Exception {
        super.setUp();

        bookList = new TreeMap<Book, Integer>();

        bookList.put(QUO_VADIS, 2);
        bookList.put(PAN_TADEUSZ, 3);
        bookList.put(CHLOPI, 5);
    }

    public void testOrderOk() {
        BookClient client;
        MyMockBookManager mock;
        BookOrder bookOrder;
        String message;

        mock = new MyMockBookManager();

        mock.setBookList(bookList);
        mock.setConnect(true);
        mock.setDisconnect(true);

        mock.setExpectedConnectCount(1);
        mock.setExpectedDisconnectCount(1);
        mock.setExpectedOrderCount(1);

        client = new BookClient(mock);

        bookOrder = new BookOrder();
        bookOrder.addBook(QUO_VADIS);
        bookOrder.addBook(PAN_TADEUSZ);

        client.orderBooks(bookOrder);

        message = BookClient.ORDERED_BOOKS +
BookClient.bookListToString(bookOrder.getBookList());
        assertEquals(message, client.getMessage());

        mock.verify();
    }

    public void testOrderFail() {
        BookClient client;
        MyMockBookManager mock;
        BookOrder bookOrder;
        String message;

        mock = new MyMockBookManager();

        mock.setBookList(bookList);
        mock.setConnect(true);
        mock.setDisconnect(true);

        mock.setExpectedConnectCount(1);
        mock.setExpectedDisconnectCount(1);
        mock.setExpectedOrderCount(1);

        client = new BookClient(mock);

        bookOrder = new BookOrder();
        bookOrder.addBook(QUO_VADIS);
```

```
        bookOrder.addBook(PAN_TADEUSZ);
        bookOrder.addBook(PAN_TADEUSZ);
        bookOrder.addBook(PAN_TADEUSZ);
        bookOrder.addBook(PAN_TADEUSZ);

        client.orderBooks(bookOrder);

        message = BookClient.NOT_ENOUGH_BOOKS_ERROR;
        assertEquals(message, client.getMessage());

        mock.verify();
    }

}
```

Jak łatwo zauważyć klasa *MyMockBookManager* jest dość obszerna. Wyobraźmy sobie ile musielibyśmy napisać takich klas w przypadku dużego systemu. Na szczęście z pomocą przychodzi nam **MockMaker**. Jest to biblioteka służąca do generacji obiektów pozornych z podanego interfejsu. Wygenerowany obiekt pozorny jest bardzo przyjazny dla użytkownika, a testy z jego użyciem pisze się bardzo sprawnie. Co więcej może on być włączony jako wtyczka do eclipse'a. Więcej szczegółów możemy znaleźć na stronie domowej projektu http://mockmaker.sourceforge.net/ oraz w [15].

```
public class MockBookManager implements BookManager{
   private ExpectationCounter myConnectCalls = new
ExpectationCounter("example.book.BookManager ConnectCalls");
   private ReturnValues myActualConnectReturnValues = new ReturnValues(false);
   private ExpectationCounter myDisconnectCalls = new
ExpectationCounter("example.book.BookManager DisconnectCalls");
   private ReturnValues myActualDisconnectReturnValues = new
ReturnValues(false);
   private ExpectationCounter myBookListCalls = new
ExpectationCounter("example.book.BookManager BookListCalls");
   private ReturnValues myActualBookListReturnValues = new
ReturnValues(false);
   private ExpectationCounter myOrderCalls = new
ExpectationCounter("example.book.BookManager OrderCalls");
   private ReturnValues myActualOrderReturnValues = new ReturnValues(false);
   private ExpectationList myOrderParameter0Values = new
ExpectationList("example.book.BookManager example.book.BookOrder");

   public void setExpectedConnectCalls(int calls){
      myConnectCalls.setExpected(calls);
   }

   public boolean connect(){
      myConnectCalls.inc();
      Object nextReturnValue = myActualConnectReturnValues.getNext();
      if (nextReturnValue instanceof ExceptionalReturnValue &&
((ExceptionalReturnValue)nextReturnValue).getException() instanceof
RuntimeException)
         throw
(RuntimeException)((ExceptionalReturnValue)nextReturnValue).getException();
      return ((Boolean) nextReturnValue).booleanValue();
```

```
   }

   public void setupExceptionConnect(Throwable arg){
      myActualConnectReturnValues.add(new ExceptionalReturnValue(arg));
   }

   public void setupConnect(boolean arg){
      myActualConnectReturnValues.add(new Boolean(arg));
   }

   public void setExpectedDisconnectCalls(int calls){
      myDisconnectCalls.setExpected(calls);
   }

   public boolean disconnect(){
      myDisconnectCalls.inc();
      Object nextReturnValue = myActualDisconnectReturnValues.getNext();
      if (nextReturnValue instanceof ExceptionalReturnValue &&
((ExceptionalReturnValue)nextReturnValue).getException() instanceof
RuntimeException)
         throw
(RuntimeException)((ExceptionalReturnValue)nextReturnValue).getException();
      return ((Boolean) nextReturnValue).booleanValue();
   }

   public void setupExceptionDisconnect(Throwable arg){
      myActualDisconnectReturnValues.add(new ExceptionalReturnValue(arg));
   }

   public void setupDisconnect(boolean arg){
      myActualDisconnectReturnValues.add(new Boolean(arg));
   }

   public void setExpectedBookListCalls(int calls){
      myBookListCalls.setExpected(calls);
   }

   public TreeMap<Book,Integer> bookList(){
      myBookListCalls.inc();
      Object nextReturnValue = myActualBookListReturnValues.getNext();
      if (nextReturnValue instanceof ExceptionalReturnValue &&
((ExceptionalReturnValue)nextReturnValue).getException() instanceof
RuntimeException)
         throw
(RuntimeException)((ExceptionalReturnValue)nextReturnValue).getException();
      return (TreeMap<Book,Integer>) nextReturnValue;
   }

   public void setupExceptionBookList(Throwable arg){
      myActualBookListReturnValues.add(new ExceptionalReturnValue(arg));
   }

   public void setupBookList(TreeMap<Book,Integer> arg){
      myActualBookListReturnValues.add(arg);
   }

   public void setExpectedOrderCalls(int calls){
      myOrderCalls.setExpected(calls);
   }

   public void addExpectedOrderValues(BookOrder arg0){
```

```
      myOrderParameter0Values.addExpected(arg0);
   }

   public boolean order(BookOrder arg0) throws BookNotFoundException,
NotEnoughBooksException{
      myOrderCalls.inc();
      myOrderParameter0Values.addActual(arg0);
      Object nextReturnValue = myActualOrderReturnValues.getNext();
      if (nextReturnValue instanceof ExceptionalReturnValue &&
((ExceptionalReturnValue)nextReturnValue).getException() instanceof
BookNotFoundException)
         throw
(BookNotFoundException)((ExceptionalReturnValue)nextReturnValue).getException(
);
      if (nextReturnValue instanceof ExceptionalReturnValue &&
((ExceptionalReturnValue)nextReturnValue).getException() instanceof
NotEnoughBooksException)
         throw
(NotEnoughBooksException)((ExceptionalReturnValue)nextReturnValue).getExceptio
n();
      if (nextReturnValue instanceof ExceptionalReturnValue &&
((ExceptionalReturnValue)nextReturnValue).getException() instanceof
RuntimeException)
         throw
(RuntimeException)((ExceptionalReturnValue)nextReturnValue).getException();
      return ((Boolean) nextReturnValue).booleanValue();
   }

   public void setupExceptionOrder(Throwable arg){
      myActualOrderReturnValues.add(new ExceptionalReturnValue(arg));
   }

   public void setupOrder(boolean arg){
      myActualOrderReturnValues.add(new Boolean(arg));
   }

   public void verify(){
      myConnectCalls.verify();
      myDisconnectCalls.verify();
      myBookListCalls.verify();
      myOrderCalls.verify();
      myOrderParameter0Values.verify();
   }

}

public class BookClientTestMockBookManager extends TestCase {

    private TreeMap<Book, Integer> bookList;

    private static final Book QUO_VADIS = new Book(1001, "Henryk Sienkiewicz",
"Quo Vadis");

    private static final Book PAN_TADEUSZ = new Book(2001, "Adam Mickiewicz",
"Pan Tadeusz");

    private static final Book CHLOPI = new Book(3001, "Wladyslaw Reymont",
"Chlopi");

    private BookClient client;
```

```
    private MockBookManager mock;

    @Override
    protected void setUp() throws Exception {
        super.setUp();

        bookList = new TreeMap<Book, Integer>();

        bookList.put(QUO_VADIS, 2);
        bookList.put(PAN_TADEUSZ, 3);

        mock = new MockBookManager();

        client = new BookClient(mock);
    }

    public void testSecondConnectFail() {
        String message;
        BookOrder bookOrder;

        mock.setExpectedConnectCalls(2);
        mock.setExpectedDisconnectCalls(1);
        mock.setExpectedBookListCalls(1);
        mock.setExpectedOrderCalls(0);

        mock.setupConnect(true);
        mock.setupConnect(false);
        mock.setupDisconnect(true);

        mock.setupBookList(bookList);

        client.getBookList();
        message = BookClient.AVAILABLE_BOOKS +
BookClient.bookListToString(bookList);
        assertEquals(message, client.getMessage());

        bookOrder = new BookOrder();

        client.orderBooks(bookOrder);
        message = BookClient.CONNECT_ERROR;
        assertEquals(message, client.getMessage());

        mock.verify();
    }

    public void testSecondOrderFailWithBookNotFoundException() {
        String message;
        BookOrder first;
        BookOrder second;

        first = new BookOrder();
        first.addBook(QUO_VADIS);
        first.addBook(QUO_VADIS);

        second = new BookOrder();
        second.addBook(CHLOPI);

        mock.setExpectedConnectCalls(2);
        mock.setExpectedDisconnectCalls(2);
        mock.setExpectedOrderCalls(2);
```

```
        mock.setupConnect(true);
        mock.setupConnect(true);

        mock.setupDisconnect(true);
        mock.setupDisconnect(true);

        mock.setupBookList(bookList);

        mock.addExpectedOrderValues(first);
        mock.addExpectedOrderValues(second);

        mock.setupOrder(true);
        mock.setupExceptionOrder(new BookNotFoundException());

        client.orderBooks(first);
        message = BookClient.ORDERED_BOOKS +
BookClient.bookListToString(first.getBookList());
        assertEquals(message, client.getMessage());

        client.orderBooks(second);
        message = BookClient.BOOK_NOT_FOUND_ERROR;
        assertEquals(message, client.getMessage());

        mock.verify();
    }

}
```

Inne podejście do obiektów pozornych prezentują biblioteki takie jak **EasyMock** i **jMock**. Tworzą one pewnego rodzaju proxy na podstawie przedstawionego interfejsu. Mechanizm działania takiego testu wygląda w następujący sposób. Najpierw nagrywamy zachowanie obiektu pozornego:

- jakie metody powinny być uruchomione i ile razy
- jakie wartości powinny być zwrócone lub rzucone wyjątki

Następnie uruchamiamy test i weryfikujemy zachowanie obiektu testowanego.

Na poniższych wydrukach kodu, te przedstawione metody są najbardziej ekonomiczne i przyjazne w użyciu przez użytkownika.

```
 public class BookClientTestEasyMock extends TestCase {

    private TreeMap<Book, Integer> bookList;

    private static final Book QUO_VADIS = new Book(1001, "Henryk Sienkiewicz",
"Quo Vadis");

    private static final Book PAN_TADEUSZ = new Book(2001, "Adam Mickiewicz",
"Pan Tadeusz");

    private static final Book CHLOPI = new Book(3001, "Wladyslaw Reymont",
"Chlopi");
```

```
    private BookClient client;

    private BookManager mock;

    @Override
    protected void setUp() throws Exception {
        super.setUp();

        bookList = new TreeMap<Book, Integer>();

        bookList.put(QUO_VADIS, 2);
        bookList.put(PAN_TADEUSZ, 3);
        bookList.put(CHLOPI, 5);

        mock = EasyMock.createMock(BookManager.class);

        client = new BookClient(mock);
    }

    public void testBookList() {
        String message;

        EasyMock.expect(mock.connect()).andReturn(true);
        EasyMock.expect(mock.bookList()).andReturn(bookList);
        EasyMock.expect(mock.disconnect()).andReturn(true);

        EasyMock.replay(mock);

        client.getBookList();
        message = BookClient.AVAILABLE_BOOKS +
BookClient.bookListToString(bookList);
        assertEquals(message, client.getMessage());

        EasyMock.verify(mock);
    }

    public void testNotEnoughBooks() {
        String message;
        BookOrder bookOrder;

        bookOrder = new BookOrder();
        bookOrder.addBook(QUO_VADIS);

        EasyMock.expect(mock.connect()).andReturn(true);
        EasyMock.expectLastCall().times(3);

        try {
            EasyMock.expect(mock.order(EasyMock.eq(bookOrder))).
                andReturn(true).
                andReturn(true).
                andThrow(new NotEnoughBooksException());
        } catch (BookNotFoundException e) {
        } catch (NotEnoughBooksException e) {
        }

        EasyMock.expect(mock.disconnect()).andReturn(true).times(3);

        EasyMock.replay(mock);

        client.orderBooks(bookOrder);
```

```
        message = BookClient.ORDERED_BOOKS +
BookClient.bookListToString(bookOrder.getBookList());
        assertEquals(message, client.getMessage());

        client.orderBooks(bookOrder);
        message = BookClient.ORDERED_BOOKS +
BookClient.bookListToString(bookOrder.getBookList());
        assertEquals(message, client.getMessage());

        client.orderBooks(bookOrder);
        message = BookClient.NOT_ENOUGH_BOOKS_ERROR;
        assertEquals(message, client.getMessage());

        EasyMock.verify(mock);
    }

}


public class BookClientTestJMock extends TestCase {

    private TreeMap<Book, Integer> bookList;

    private static final Book QUO_VADIS = new Book(1001, "Henryk Sienkiewicz",
"Quo Vadis");

    private static final Book PAN_TADEUSZ = new Book(2001, "Adam Mickiewicz",
"Pan Tadeusz");

    private static final Book CHLOPI = new Book(3001, "Wladyslaw Reymont",
"Chlopi");

    private BookClient client;

    private BookManager mock;

    private Mockery context;

    @Override
    protected void setUp() throws Exception {
        super.setUp();

        bookList = new TreeMap<Book, Integer>();

        bookList.put(QUO_VADIS, 2);
        bookList.put(PAN_TADEUSZ, 3);

        context = new Mockery();

        mock = context.mock(BookManager.class);

        client = new BookClient(mock);
    }

    public void testDisconnectFail() {
        String message;

        context.checking(new Expectations() {{
            one (mock).connect(); will(returnValue(true));
            one (mock).connect(); will(returnValue(true));
```

```
            one (mock).bookList(); will(returnValue(bookList));
            one (mock).bookList(); will(returnValue(bookList));

            one (mock).disconnect(); will(returnValue(true));
            one (mock).disconnect(); will(returnValue(false));
        }});

        client.getBookList();
        message = BookClient.AVAILABLE_BOOKS +
BookClient.bookListToString(bookList);
        assertEquals(message, client.getMessage());

        client.getBookList();
        message = BookClient.DISCONNECT_ERROR;
        assertEquals(message, client.getMessage());

        context.assertIsSatisfied();
    }

    public void testForException() {
        String message;
        final BookOrder bookOrder;

        bookOrder = new BookOrder();
        bookOrder.addBook(CHLOPI);

        try {
            context.checking(new Expectations() {{
                one (mock).connect(); will(returnValue(true));

                one (mock).order(bookOrder); will(throwException(new
BookNotFoundException()));

                one (mock).disconnect(); will(returnValue(true));
            }});
        } catch (BookNotFoundException e) {
        } catch (NotEnoughBooksException e) {
        }

        client.orderBooks(bookOrder);
        message = BookClient.BOOK_NOT_FOUND_ERROR;
        assertEquals(message, client.getMessage());

        context.assertIsSatisfied();
    }

}
```

Główną zaletą użycia techniki obiektów zastępczych w porównaniu z innymi technikami testowania jest możliwość weryfikacji zachowań (zarówno prawidłowości wywołań bądź ich braku, liczby wywołań jak i wartości zwracanych) obiektu testowanego względem środowiska, w którym pracuje, a nie jedynie weryfikacji zmiany samych stanów testowanego obiektu [4]. Dzięki jej zastosowaniu staramy się lepiej zrozumieć wymagania, które mają być realizowane przez nasz kod, a tym samym mamy większą wiedzę na temat tego, co powinniśmy zrobić w najbliższym etapie pracy.

Inne drugoplanowe zalety obiektów pozornych, pozwalające lepiej realizować koncepcję testów jednostkowych to możliwość:

- testowania jeszcze mniejszych części kodu niż w przypadku pozostałych technik,
- tworzenia testów niezależnych od siebie nawzajem,
- późniejszego podjęcia decyzji o pewnych fragmentach infrastruktury oraz zmniejszenie ilości kodu testowego, kosztem nieco większej jego złożoności.

Zbyt ścisłe zrównoleglenie z kodem testowanym i potrzeba synchronizacji pomiędzy nimi jest czasem uznawane za jedna z wad, choć w rzeczywistości zapewnia egzekwowanie przestrzegania ścisłej zgodności z procesem tworzenia testów jednostkowych. Wadą może być za to bardzo wąski obszar danego scenariusza testowego z użyciem mocków, co w przypadku nawet drobnych zmian w testowanym kodzie może prowadzić do tego, że test szybciej stanie się przestarzały i będzie wymagał od nas poprawek [3].

Obiekty zastępcze pozwalają w znacznym stopniu zautomatyzować tworzenie testów jednostkowych (zwłaszcza, że istnieje wiele różnych narzędzi do ich tworzenia w wielu językach programowania), a tym samym zwiększyć wydajność oraz przyspieszyć proces tworzenia testów, powodując, że powstające oprogramowanie będzie znacznie lepszej jakości. W szczególności technika ta pozwoli nam na większe zaufanie, co do jakości indywidualnych części naszego oprogramowania. Pewnymi niedogodnościami mogą być narzędzia do tworzenia obiektów pozornych (szczególnie dla rzadziej używanych języków programowania), ponieważ są stale rozwijane ze względu to, że sama idea powstała w miarę niedawno [2]. Oczywiste jest, że nie ma jedynego słusznego sposobu na wykonanie testów, a tym samym nie zawsze istnieje możliwość skorzystania z tej wybranej techniki, ale tych wyjątków w przypadku obiektów zastępczych jest naprawdę niewiele.

Podsumowując obiekty pozorne są i będą ważną gałęzią testów. Powinny one w niedługim czasie znacznie się rozpowszechnić, co w połączeniu z technikami zwinnymi (agile'owymi) będzie potężnym narzędziem w tworzeniu oprogramowania.

Bibliografia:

[1] 2006. Techniques for Successful Evolutionary/Agile Database Development, artykuł internetowy, http://www.agiledata.org/essays/tdd.html

[2] Bain S. 2006. *Mocks, Fakes, and Stubs*. Net Objectives, Vol. 3, No. 4, pp. 2-14.

[3] Brown M., Tapolcsanyi E. 2003. *Mock Object Patterns*. Proceedings of The 10th Conference on Pattern Languages of Programs, Sept 8-12, USA.

[4] Burke E. M., Coyner B. M. 2003. *Java Extreme Programming Cookbook*. O'Reilly and Associates.

[5] Fowler M. 2007. *Mocks Aren't Stubs*. Artykuł ze strony internetowej autora, http://martinfowler.com/articles/mocksArentStubs.html.

[6] Freeman S., Mackinnon T., Pryce N., Walnes J. 2004. *Mock Roles Not Objects.* Proceedings of 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04), ACM Press, pp. 236–246.

[7] Freeman S., Pryce N. 2006. *Evolving an embedded domain-specific language in Java*. Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA, pp. 855-865.

[8] 1999. *IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987* in IEEE Standards: Software Engineering, Volume Two: Process Standards; The Institute of Electrical and Electronics Engineers, Inc. Software Engineering Technical Committee of the IEEE Computer Society.

[9] Mackinnon T., Freeman S., Craig P. 2001. *Endo-testing: Unit testing with mock objects*. Proceedings of XP2000. Extreme Programming Examined, Addison-Wesley, Boston, MA, pp. 287-301.

[10] Massol V., Husted T. 2003. *JUnit in Action*. Manning Publications.

[11] Ryu H.-Y., Sohn B.-K., Park J.-H. 2005. *Mock objects framework for TDD in the network environment*. Proceedings of the Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05), pp. 430- 434.

[12] Stewart S. 2004. *Approaches to Mocking*. An article from O'Reilly Network Site, http://www.onjava.com/pub/a/onjava/2004/02/11/mocks.html.

[13] Thomas D., Hunt A. 2002. *Mock Objects*. IEEE Software, Vol. 19, No. 3, pp. 22-24.

[14] Wirfs-Brock R.J. 2007. *Driven to ... Discovering Your Design Values*. IEEE Software, Vol. 24, No. 1, pp. 9-11.

[15] Keld H. Hansen *Using Mock Objects in Java*. Artykuł ze strony internetowej, http://javaboutique.internet.com/tutorials/mock_objects/

[16] *EasyMock 2.2 Readme*, http://www.easymock.org/EasyMock2_2_Documentation.html

[17] *The jMock Cookbook*, http://www.jmock.org/cookbook.html