Programowanie sterowane testami w Pythonie

Jak tworzyć skalowalne zestawy testów i aplikacji

Alessandro Molina

Helion

Tytuł oryginału: Crafting Test-Driven Software with Python:
Write test suites that scale with your applications

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-8664-8

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/proste.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/proste
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 2

Dublery używane podczas testów na przykładzie aplikacji czatu

W poprzednim rozdziale wyjaśniłem, że jeśli zbiór testów ma charakteryzować się sensowną niezawodnością, to powinien zawierać różnego rodzaju testy zapewniające pokrycie komponentów na poszczególnych poziomach. Testy — w zależności od liczby komponentów, których dotyczą — są zwykle umieszczane w trzech kategoriach: jednostkowe, integracji i E2E.

Dublery używane podczas testów ułatwiają implementację testów przez usunięcie zależności między komponentami, co pozwala programiście symulować żądane zachowanie.

W tym rozdziale przedstawię najczęściej spotykane rodzaje dublerów używanych podczas testów, omówię ich przeznaczenie, a także pokażę przykłady zastosowania w rzeczywistym kodzie. Po ukończeniu lektury niniejszego rozdziału będziesz mieć wiedzę niezbędną do używania dublerów i umiejętności umożliwiające wykorzystanie tej wiedzy we własnych projektach Pythona.

Dzięki dodaniu do swojego zestawu narzędzi dublerów używanych podczas testów zyskujesz możliwość tworzenia szybciej wykonywanych testów, eliminowania powiązania testowanych komponentów z pozostałą częścią systemu, a także symulowania zachowania, w którym występuje zależność od stanu innych komponentów. Ogólnie rzecz biorąc, usprawnisz własne umiejętności w zakresie tworzenia zbiorów testów.

W tym rozdziale omówię zagadnienia, posługując się podejściem **programowania sterowanego testami** (**TDD**), czyli stylu, w którym praca nad aplikacją mającą zależności zewnętrzne, takie jak sieć lub system zarządzania relacyjnymi bazami danych, opiera się na dublerach testów używanych podczas procesu tworzenia aplikacji i zastępowanych w wewnętrznych warstwach testowych w celu zapewnienia możliwości szybkiego i spójnego wykonywania testów.

Oto tematy, które zostały omówione w tym rozdziale:

- wprowadzenie do dublerów używanych podczas testów;
- aplikacja czatu opracowana z wykorzystaniem modelu TDD;
- stosowanie obiektów imitacji;
- zastępowanie komponentów ich namiastkami;
- sprawdzanie sposobu działania za pomocą szpiegów;
- stosowanie dublerów używanych podczas testów;
- zastępowanie zależności imitacjami;
- poznawanie testów akceptacji i dublerów używanych podczas testów;
- zarządzanie zależnościami za pomocą mechanizmu wstrzykiwania zależności.

Wymagania techniczne

Wymagany jest jedynie funkcjonujący interpreter Pythona.

Przykłady zamieszczone w tym rozdziale zostały utworzone w Pythonie 3.7 i powinny działać w większości nowoczesnych wersji Pythona.

Przykładowe fragmenty kodu omówione w tym rozdziale znajdziesz w materiałach dostępnych pod adresem *https://ftp.helion.pl/przyklady/proste.zip*.

Wprowadzenie do dublerów używanych podczas testów

W modelu programowania sterowanego testami testy mają wpływ na proces tworzenia rozwiązania i jego architekturę. Projekt oprogramowania ewoluuje wraz ze zmianami wprowadzanymi w oprogramowaniu podczas tworzenia nowych testów, a ostateczna postać architektury będzie wynikiem konieczności zaliczenia wszystkich testów.

Testy przyjmują więc rolę arbitra decydującego o przyszłości oprogramowania i deklarującego, że oprogramowanie działa w zamierzony sposób. Istnieją określone rodzaje testów wyraźnie zaprojektowane do informowania, że oprogramowanie działa zgodnie z oczekiwaniami: to testy akceptacji i funkcjonalne.

Wprawdzie istnieją dwa możliwe do zastosowania podejścia w modelu TDD — od początku do końca i od końca do początku (w pierwszym zaczyna się od testów na najwyższym poziomie, w drugim zaś od testów jednostkowych) — ale najlepszym sposobem na uniknięcie błędnej kolejności jest pamiętanie o regułach akceptacji. Z kolei najefektywniejszym sposobem na zapamiętanie tych reguł jest zapisanie ich w postaci testów.

W jaki sposób należy utworzyć test, który zależy od funkcjonowania jeszcze nie utworzonego oprogramowania? Znaczenie kluczowe mają tutaj tzw. **dublery używane podczas testów** (ang. *test doubles*), czyli obiekty, które mogą zastąpić brakujące, niekompletne lub kosztowne fragmenty kodu, wykorzystywane jedynie na potrzeby testów.

Dubler używany podczas testu zajmuje miejsce innego obiektu i udaje, że ma takie same możliwości jak zastąpiony obiekt, choć w rzeczywistości tak nie jest.

Jeżeli dzięki dublerom używanym podczas testów uda się zaliczyć testy, to w jaki sposób uniknąć wydania produktu zawierającego po prostu wiele imitacji encji? Dlatego ważne jest przygotowanie różnych warstw testów — wraz z przechodzeniem do warstw znajdujących się wyżej na stosie warstw zmniejsza się liczba potrzebnych dublerów. Na samej górze stosu mamy testy typu E2E, które w ogóle nie powinny wymagać dublerów.

Model programowania sterowanego testami sugeruje także utworzenie minimalnej ilości kodu niezbędnej do zaliczenia testu. To jest bardzo ważna reguła, ponieważ w przeciwnym razie można bardzo łatwo doprowadzić do tworzenia kodu wymagającego zdefiniowania nowych testów.

W przypadku testów wysokiego poziomu (np. testów akceptacji) będą one od samego początku wymagały wielu dublerów, podczas gdy jeszcze nie będzie kodu tworzonego oprogramowania. Kiedy więc oczekuje się zastąpienia tych dublerów rzeczywistymi obiektami?

Kent Beck, autor książki *TDD. Sztuka tworzenia dobrego kodu*, sugeruje bazowanie na listach rzeczy do zrobienia. Podczas tworzenia kodu należy zapisać wszystko to, co wydaje się niezbędne do usprawnienia, poprawienia lub zastąpienia. Przed przejściem do utworzenia następnego testu akceptacji wszystkie pozycje znajdujące się na liście rzeczy do zrobienia powinny być wykonane.

Na swojej liście rzeczy do zrobienia umieszczasz encje przeznaczone do zastąpienia dublerów testów rzeczywistymi obiektami. W efekcie będą utworzone testy weryfikujące sposób działania tych rzeczywistych obiektów, a co za tym idzie, ich implementacji. Ostatecznie w oryginalnym teście akceptacji dublery używane podczas testów zostaną zastąpione rzeczywistymi obiektami, co pozwoli sprawdzić, czy dany test nadal jest zaliczany.

W tym rozdziale zamierzam pokazać, jak dublery używane podczas testów mogą pomóc w trakcie programowania sterowanego testami. W tym celu posłużę się przykładem aplikacji czatu i wykorzystam najczęściej stosowane rodzaje dublerów testów.

Aplikacja czatu opracowana z wykorzystaniem modelu TDD

Gdy rozpoczynasz pracę nad nową funkcjonalnością, pierwszym testem do utworzenia może być podstawowy test akceptacji — jego przeznaczeniem jest pomoc w zdefiniowaniu tego, co można określić słowami „to jest to, co chcę osiągnąć". Testy akceptacji ujawniają komponenty niezbędne do utworzenia, określają sposoby ich działania oraz pozwalają przejść dalej dzięki opracowaniu testów programistycznych dla tych komponentów, a tym samym zdefiniowaniu testów jednostkowych i testów integracji.

W omawianym przykładzie aplikacji czatu test akceptacji będzie sprawdzał, czy użytkownik może wysłać wiadomość i czy drugi użytkownik może ją otrzymać.

```
import unittest


class TestChatAcceptance(unittest.TestCase):
    def test_message_exchange(self):
        user1 = ChatClient("Jan Kowalski")
        user2 = ChatClient("Harry Potter")
        user1.send_message("Witaj, świecie!")
        messages = user2.fetch_messages()
        assert messages == ["Jan Kowalski: Witaj, świecie!"]

if __name__ == '__main__':
    unittest.main()
```

Na podstawie tego testu jasno wynika, że dwóm egzemplarzom `ChatClient` ma być umożliwiona wymiana wiadomości. Pierwszy egzemplarz będzie wysyłał nową wiadomość, drugi zaś będzie pobierał tę wiadomość i ją odczytywał.

W ten sposób oswajamy się z myślą, że powinny istnieć dwa klienty czatu: jeden będzie wysyłał wiadomości, a drugi będzie je odbierał. Ta uproszczona wizja aplikacji na pewno zostanie w przyszłości zmodyfikowana, natomiast na tym etapie pracy pozwoliła na jasne zdefiniowanie oczekiwań.

Warto w tym miejscu wspomnieć, że klasa `ChatClient` jeszcze nie istnieje. Wprawdzie wiemy, że powinna mieć możliwość wysyłania i odbierania wiadomości, ale nie znamy żadnych związanych z nią szczegółów. Dlatego następnym krokiem jest rozpoczęcie definiowania funkcjonalności, którą ma mieć wymieniona klasa.

Jeżeli uruchomisz test akceptacji zdefiniowany w pliku *01_chat_acceptance.py*, który znajduje się w katalogu zawierającym poprzedni zbiór testów, wówczas wynikiem będzie błąd.

```
$ python 01_chat_acceptance.py TestChatAcceptance
E
======================================================================
ERROR: test_message_exchange (__main__.TestChatAcceptance)
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "01_chat_acceptance.py", line 5, in test_message_exchange
    user1 = ChatClient("Jan Kowalski")
NameError: global name 'ChatClient' is not defined

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (errors=1)
```

Z wygenerowanego komunikatu błędu dotyczącego braku egzemplarza `ChatClient` jasno wynika, że kolejnym krokiem prawdopodobnie powinno być utworzenie klienta.

W ten sposób ustaliliśmy, że należy przystąpić do utworzenia egzemplarza `ChatClient`. Skoro chcemy zapewnić klientowi możliwość użycia nicka użytkownika, trzeba się upewnić, że klasa będzie wiedziała o istnieniu tego nicka. Rozpoczynamy od utworzenia odpowiedniego testu programistycznego.

```
class TestChatClient(unittest.TestCase):
    def test_nickname(self):
        client = ChatClient("Użytkownik 1")

        assert client.nickname == "Użytkownik 1"
```

W tym momencie wiadomo, że oba testy, akceptacji i programistyczne, zakończą się niepowodzeniem, ponieważ nie istnieje jeszcze żadna implementacja, dzięki której test mógłby zostać zaliczony. Sprawdzamy zatem, czy zbiór testów działa zgodnie z oczekiwaniami.

```
$ python 01_chat_acceptance.py TestChatClient
E
======================================================================
ERROR: test_nickname (__main__.TestChatClient)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "01_chat_acceptance.py", line 16, in test_nickname
    client = ChatClient("Użytkownik 1")
NameError: global name 'ChatClient' is not defined

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (errors=1)
```

Test oczywiście kończy się niepowodzeniem z powodu braku egzemplarza `ChatClient`. Przystępujemy więc do implementacji klasy `ChatClient` i zapewnienia jej obsługi nicka.

```
class ChatClient:
    def __init__(self, nickname):
        self.nickname = nickname
```

Teraz ponowne wykonanie testu jednostkowego powinno zakończyć się sukcesem, ponieważ został utworzony egzemplarz `ChatClient`, który potrafi przechowywać wartość nicka przypisaną w aplikacji.

```
$ python 01_dummy.py TestChatClient
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Skoro test jednostkowy został zaliczony, można przejść dalej. Jaki powinien być następny krok? Aby się tego dowiedzieć, należy się cofnąć i ponownie wykonać test akceptacji. Czy ten test został zaliczony? Czy konieczne jest utworzenie jakiejkolwiek innej encji?

```
$ python 01_chat_acceptance.py TestChatAcceptance
E
======================================================================
ERROR: test_message_exchange (__main__.TestChatAcceptance)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "01_chat_acceptance.py", line 8, in test_message_exchange
    user1.send_message("Witaj, świecie!")
AttributeError: ChatClient instance has no attribute 'send_message'

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (errors=1)
```

Ponowne wykonanie testu akceptacji spowodowało wygenerowanie komunikatu błędu dotyczącego braku metody `ChatClient.send_message()`. Ten komunikat wyraźnie wskazuje encję, która powinna być przygotowana jako następna. Tradycyjnie w przypadku stosowania modelu TDD pracę można rozpocząć od zdefiniowania testu jednostkowego.

Zbiór testów `TestChatClient` rozbudowujemy o nowy test zdefiniowany w metodzie o nazwie `test_send_message()`.

```
class TestChatClient(unittest.TestCase):
    def test_nickname(self):
        client = ChatClient("Użytkownik 1")

        assert client.nickname == "Użytkownik 1"

    def test_send_message(self):
        client = ChatClient("Użytkownik 1")
        sent_message = client.send_message("Witaj, świecie!")
        assert sent_message == "Użytkownik 1: Witaj, świecie!"
```

Nowy test powoduje utworzenie klienta dla użytkownika nazwanego `Użytkownik 1`, który następnie wysyła wiadomość w aplikacji czatu. W teście sprawdzamy, czy wiadomość faktycznie została wysłana jako pochodząca od tego użytkownika.

Po powrocie do powłoki i ponownym wykonaniu testów dla komponentu `ChatClient` okazuje się, że należy przystąpić do utworzenia metody pozwalającej zaliczyć nowy test.

```
$ python 01_chat_acceptance.py TestChatClient
.E
======================================================================
ERROR: test_send_message (__main__.TestChatClient)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "01_chat_acceptance.py", line 22, in test_send_message
    sent_message = client.send_message("Witaj, świecie!")
AttributeError: ChatClient instance has no attribute 'send_message'

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (errors=1)
```

Wracamy więc do programowania i przystępujemy do zdefiniowania metody `send_message()` w komponencie. Zgodnie z wcześniejszymi ustaleniami otrzymany komunikat powinien zostać zaakceptowany, poprzedzony prefiksem w postaci nicka nadawcy i prawdopodobnie wysłany do wszystkich pozostałych użytkowników czatu.

```
class ChatClient:
    def __init__(self, nickname):
        self.nickname = nickname

    def send_message(self, message):
        sent_message = "{}: {}".format(self.nickname, message)
        self.connection.broadcast(message)
        return sent_message
```

Teraz trzeba ponownie wykonać testy dla komponentu, aby potwierdzić ich zaliczenie.

```
$ python 01_chat_acceptance.py TestChatClient
.E
======================================================================
ERROR: test_send_message (__main__.TestChatClient)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "01_chat_acceptance.py", line 22, in test_send_message
    sent_message = client.send_message("Witaj, świecie!")
  File "01_chat_acceptance.py", line 32, in send_message
    self.connection.broadcast(message)
AttributeError: ChatClient instance has no attribute 'connection'

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (errors=1)
```

Jednak test ponownie nie został zaliczony — tym razem wygenerowany komunikat wskazuje, że wprawdzie metoda `ChatClient.send_message()` istnieje i test mógł ją wywołać, ale nie zadziałała zgodnie z oczekiwaniami.

Wynika to stąd, że w przypadku klienta wybiegliśmy nieco do przodu — wiadomość została wysłana przez interfejs, zanim został on udostępniony testom. Jednak doskonale wiadomo, że takie rozwiązanie trzeba będzie zastosować, i ten przykład doskonale nadaje się do wprowadzenia pierwszego typu dublerów używanych podczas testów, czyli **obiektów pozornych**.

Stosowanie obiektów pozornych

Obiekt pozorny (ang. *dummy object*) to taki obiekt, który nic nie robi. Jedynym celem istnienia takiego obiektu jest jego przekazywanie jako argumentu, aby uniknąć awarii kodu z powodu braku obiektu. Natomiast implementacja takiego obiektu jest całkowicie pusta i ten obiekt dosłownie nic nie robi.

W omawianej tutaj aplikacji czatu potrzebny jest obiekt połączenia, który będzie miał możliwość przesyłania wiadomości między klientami. Obiekt połączenia jeszcze nie został zaimplementowany i w tym momencie koncentrujemy się na zaliczeniu testu `ChatClient.send_message()`. Jak można zapewnić zaliczenie tego testu, gdy nie mamy działającego obiektu `Connection` wymaganego przez klienta?

W takiej sytuacji przydatny okazuje się obiekt pozorny. Zastępuje on rzeczywisty obiekt i udaje, że wykonuje jego zadanie, choć tak naprawdę nie robi zupełnie nic.

Obiekt pozorny dla klasy `Connection` będzie zdefiniowany za pomocą następującego fragmentu kodu:

```
class _DummyConnection:
    def broadcast(*args, **kwargs):
        pass
```

W ten sposób powstaje obiekt dostarczający metodę `broadcast()` i nie wykonujący absolutnie żadnej operacji. Obiekt pozorny pełni rolę wypełniacza dla argumentów właściwości wymaganych przez inne obiekty. Bardzo często obiekty pozorne nie są nawet używane i jedynie potwierdzają istnienie wymaganego argumentu.

Zdefiniowany wcześniej test `TestChatClient.test_send_message()` można teraz dostosować do użycia obiektu pozornego dla połączenia. W tym celu należy właściwości `client.connection` przypisać wartość `_DummyConnection`. To powinno pozwolić na zaliczenie testu, ponieważ pozbywamy się zależności od rzeczywistego połączenia.

```
class TestChatClient(unittest.TestCase):
    ...
    def test_send_message(self):
        client = ChatClient("Użytkownik 1")
        client.connection = _DummyConnection()
        sent_message = client.send_message("Witaj, świecie!")

        assert sent_message == "Użytkownik 1: Witaj, świecie!"
```

Innym wygodnym sposobem implementacji obiektów pozornych jest wykorzystanie modułu Pythona `unittest.mock`. W podrozdziale „Stosowanie obiektów imitacji” przekonasz się, że

pomimo konkretnej nazwy obiekt `unittest.mock.Mock` można w praktyce wykorzystać we wszystkich sytuacjach, w których są używane dublery testów. Wszystko zależy od funkcjonalności zarówno tej, której trzeba użyć, jak i tej, którą można zignorować.

Powracając do poprzedniego przykładu, `_DummyConnection` można zastąpić obiektem `unittest.mock.Mock` i tym samym uniknąć konieczności implementowania dedykowanej klasy.

```
import unittest.mock

class TestChatClient(unittest.TestCase):
    ...
    def test_send_message(self):
        client = ChatClient("Użytkownik 1")
        client.connection = unittest.mock.Mock()
        sent_message = client.send_message("Witaj, świecie!")

        assert sent_message == "Użytkownik 1: Witaj, świecie!"
```

Po wykonaniu testów `TestChatClient` okaże się, że wreszcie zostały zaliczone.

```
$ python 02_chat_dummy.py TestChatClient
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Czy to oznacza koniec pracy? Jeszcze nie, ponieważ test akceptacji (`TestChatAcceptance`) nadal jest niezaliczony.

```
$ python 02_chat_dummy.py TestChatAcceptance
E
======================================================================
ERROR: test_message_exchange (__main__.TestChatAcceptance)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "02_chat_dummy.py", line 8, in test_message_exchange
    user1.send_message("Witaj, świecie!")
  File "02_chat_dummy.py", line 39, in send_message
    self.connection.broadcast(message)
AttributeError: ChatClient instance has no attribute 'connection'

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (errors=1)
```

Wprawdzie metoda `ChatClient.send_message()` została zaimplementowana i przekazana testowi, ale test akceptacji przypomina o konieczności implementacji obiektu `Connection` używanego jako dublera w teście zdefiniowanym w metodzie `send_message()`.

Obiekt połączenia to kolejny komponent przeznaczony do implementacji. Jednak połączenie musi mieć możliwość prowadzenia komunikacji z serwerem, aby przekazywać wiadomości do

wszystkich połączonych klientów. W takim przypadku przekazanie `DummyConnection` będzie już niewystarczające do zaliczenia testów. Konieczne jest faktyczne przekazywanie wiadomości, co oznacza, że trzeba będzie wykorzystać tzw. **namiastki** (ang. *stubs*).

Zastępowanie komponentów ich namiastkami

W omawianym przykładzie obiekt `Connection` będzie odpowiedzialny za udostępnianie wiadomości pozostałym klientom i prawdopodobnie w przyszłości umożliwi powiadamianie o pojawieniu się nowych wiadomości.

Pierwszym krokiem podczas opracowywania obiektu `Connection` jest utworzenie zbioru testów `TestConnection` i testu w metodzie `test_broadcast()`, aby w ten sposób jasno zdefiniować oczekiwania związane z obiektem.

```
class TestConnection(unittest.TestCase):
    def test_broadcast(self):
        c = Connection(("localhost", 9090))

        c.broadcast("dowolny komunikat")

        assert c.get_messages()[-1] == "dowolny komunikat"
```

Ten test określa, że po przekazaniu wiadomości ostatnim elementem na liście wiadomości w aplikacji powinna być wiadomość przekazana, ponieważ została wysłana jako ostatnia. Oczywiście wykonanie tego testu teraz zakończy się niepowodzeniem, ponieważ obiekt `Connection` w ogóle jeszcze nie istnieje. Przystępujemy więc do jego utworzenia.

Jednym z rozwiązań podczas implementowania komunikacji między klientami jest wykorzystanie egzemplarza `multiprocessing.managers.SyncMaster` i przechowywanie wiadomości na liście dostępnej dla wszystkich klientów.

Konieczne będzie zarejestrowanie w menedżerze pojedynczego identyfikatora `Connection.get_messages`. Celem tego identyfikatora jest zwrot listy wiadomości aktualnie znajdujących na czacie, aby egzemplarz `ChatClient` mógł je odczytywać lub dołączać nowe.

```
from multiprocessing.managers import SyncManager

class Connection(SyncManager):
    def __init__(self, address):
        self.register("get_messages")
        super().__init__(address=address, authkey=b'mychatsecret')
        self.connect()
```

Następnie metoda `Connection.broadcast()` może po prostu pobierać wiadomości za pomocą wywołania `Connection.get_messages()` i dołączać nowe.

```
from multiprocessing.managers import SyncManager

class Connection(SyncManager):
    def __init__(self, address):
        self.register("get_messages")
        super().__init__(address=address, authkey=b'mychatsecret')
        self.connect()

    def broadcast(self, message):
        messages = self.get_messages()
        messages.append(message)
```

W tym momencie obiekt `connection` jest gotowy i dostarcza metodę `broadcast()`. Teraz dzięki ponownemu wykonaniu testu można sprawdzić, czy nowa wiadomość faktycznie jest dodawana do czatu.

```
$ python 03_chat_stubs.py TestConnection
E
======================================================================
ERROR: test_broadcast (__main__.TestConnection)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "03_chat_stubs.py", line 33, in test_broadcast
    c = Connection(("localhost", 9090))
  File "03_chat_stubs.py", line 56, in __init__
    self.connect()
  File "/usr/lib/python3.7/multiprocessing/managers.py", line 532, in connect
    conn = Client(self._address, authkey=self._authkey)
  File "/usr/lib/python3.7/multiprocessing/connection.py", line 492, in Client
    c = SocketClient(address)
  File "/usr/lib/python3.7/multiprocessing/connection.py", line 619, in
SocketClient
    s.connect(address)
ConnectionRefusedError: [Errno 111] Connection refused

----------------------------------------------------------------------
Ran 1 test in 0.003s

FAILED (errors=1)
```

Niestety wykonanie testu zakończyło się niepowodzeniem z powodu braku serwera, co uniemożliwia nawiązanie połączenia. Dopóki nie zostanie udostępniony serwer, metodę `Connection.connect()` można w teście zastąpić jej wersją pozorną i ponownie wykonać test.

```
class TestConnection(unittest.TestCase):
    def test_broadcast(self):
        with unittest.mock.patch.object(Connection, "connect"):
            c = Connection(("localhost", 9090))

        c.broadcast("dowolny komunikat")

        assert c.get_messages()[-1] == "dowolny komunikat"
```

`unittest.mock.patch.object()` to wygodna metoda pozwalająca zastąpić metodę lub obiekt za pomocą egzemplarza `unittest.mock.Mock` na cały czas wykonywania bloku kodu w kontekście. W omawianym przykładzie została wyłączona metoda `Connection.connect()`, aby było możliwe nawiązanie połączenia pomimo braku serwera.

W tym momencie oczekujemy zaliczenia testu. Przystępujemy więc do jego ponownego wykonania.

```
$ python 03_chat_stubs.py TestConnection
F
======================================================================
FAIL: test_broadcast (__main__.TestConnection)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "03_chat_stubs.py", line 36, in test_broadcast
    c.broadcast("dowolny komunikat")
  File "03_chat_stubs.py", line 60, in broadcast
    messages = self.get_messages()
  File "/usr/lib/python3.7/multiprocessing/managers.py", line 724, in temp
    token, exp = self._create(typeid, *args, **kwds)
  File "/usr/lib/python3.7/multiprocessing/managers.py", line 606, in _create
    assert self._state.value == State.STARTED, 'server not yet started'
AssertionError: server not yet started

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Jednak się nie udało. Wprawdzie utworzenie obiektu zakończyło się sukcesem, ale próba pobrania listy wiadomości opublikowanych na czacie w celu dołączenia nowej wiadomości zakończyła się niepowodzeniem z powodu braku serwera, z którym można nawiązać połączenie.

Jednak naprawdę potrzebujemy wiadomości, ponieważ w przeciwnym razie test nie potrafi potwierdzić, czy wiadomość została dodana do istniejących i tym samym wysłana. Co można zrobić w tej sytuacji?

W takich przypadkach z pomocą przychodzą namiastki. Zapewniają one gotowe odpowiedzi, zastępując fragmenty oprogramowania gotowym do użycia stanem lub odpowiedzią, którą otrzymalibyśmy, gdyby dany fragment kodu faktycznie został wykonany. Dlatego wywołanie `Connection.get_messages()` zastępujemy namiastką zwracającą pustą listę i sprawdzamy, czy wszystko działa zgodnie z oczekiwaniami.

```
class TestConnection(unittest.TestCase):
    def test_broadcast(self):
        with unittest.mock.patch.object(Connection, "connect"):
            c = Connection(("localhost", 9090))

        with unittest.mock.patch.object(c, "get_messages", return_value=[]):
            c.broadcast("dowolny komunikat")

            assert c.get_messages()[-1] == "dowolny komunikat"
```

Zwróć uwagę, że po pierwszym wywołaniu `unittest.mock.patch.object()`, w którym metoda `connect()` została zastąpiona jej wersją pozorną, mamy kolejne wywołanie `unittest.mock.patch.object()`. W tym drugim wywołaniu metodę `get_messages()` nowo utworzonego egzemplarza `Connection` zastąpiliśmy odpowiedzią zawierającą pustą listę, a tym samym zasymulowaliśmy wysłanie pierwszej wiadomości do czatu.

Ponowne wykonanie testów pozwala potwierdzić, że metoda `Connection.broadcast()` działa zgodnie z oczekiwaniami.

```
$ python 03_chat_stubs.py TestConnection
.
----------------------------------------------------------------------
Ran 1 test in 0.001s
OK
```

W porządku. Czy skoro testy `ChatClient` i `Connection` zostały zaliczone, naszą pracę można uznać za zakończoną?

Sprawdźmy to za pomocą testu akceptacji.

```
$ python 03_chat_stubs.py TestChatAcceptance
E
======================================================================
ERROR: test_message_exchange (__main__.TestChatAcceptance)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "03_chat_stubs.py", line 10, in test_message_exchange
    user1.send_message("Witaj, świecie!")
  File "03_chat_stubs.py", line 49, in send_message
    self.connection.broadcast(message)
AttributeError: 'ChatClient' object has no attribute 'connection'

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (errors=1)
```

Tak więc to jeszcze nie koniec… Utworzyliśmy obiekt `Connection`, choć zapomnieliśmy o jego dołączeniu do `ChatClient`.

Przechodzimy dalej do połączenia obiektów `ChatClient` i `Connection`. Przy okazji poznasz mechanizm tzw. szpiegów, pozwalający tutaj sprawdzić, czy egzemplarz `ChatClient` używa obiektu `Connection` w oczekiwany sposób.

Sprawdzanie zachowania za pomocą szpiegów

Wiadomo, że w omawianej tutaj aplikacji egzemplarz `ChatClient` musi używać połączenia do wysyłania i odbierania wiadomości. Dlatego kolejnym zadaniem jest upewnienie się, że test zdefiniowany w metodzie `test_message_exchange()` został zaliczony, a tym samym że istnieje połączenie i jest wykorzystane. Jednak nie chcemy, aby połączenie było nawiązywane w trakcie każdej operacji tworzenia egzemplarza `ChatClient`. Rozwiązaniem jest wykorzystanie metody przeznaczonej do nawiązywania połączenia z opóźnieniem, czyli dopiero w chwili jego pierwszego użycia.

Tej metodzie nadajemy nazwę `ChatClient._get_connection()`. Chcemy mieć pewność, że egzemplarz `ChatClient` faktycznie będzie używał połączenia dostarczanego przez tę metodę. Aby to potwierdzić, zdefiniujemy test wykorzystujący tzw. szpiega (ang. *spy*), czyli rodzaj obiektu pozornego, który zamiast nic nie robić rejestruje informacje o jego wywołaniu (o ile to w ogóle nastąpiło) oraz o użytych przy tym argumentach.

Podobnie jak to miało miejsce podczas definiowania namiastek, także tutaj wykorzystamy wywołanie `unittest.mock.patch()` do zastąpienia metody `ChatClient._get_connection()` namiastką, której wartością zwrotną będzie nie połączenie, lecz wspomniany wcześniej szpieg. Następnie za pomocą tego szpiega będzie można sprawdzić, czy do wysłania wiadomości metoda `ChatClient.send_message()` faktycznie użyła otrzymanego połączenia.

```
    def test_client_connection(self):
        client = ChatClient("Użytkownik 1")

        connection_spy = unittest.mock.MagicMock()
        with unittest.mock.patch.object(client, "_get_connection",
                                        return_value=connection_spy):
            client.send_message("Witaj, świecie!")

        # Asercja potwierdzająca wywołanie szpiega
        # razem z oczekiwanymi danymi
        connection_spy.broadcast.assert_called_with(("Użytkownik 1:
                                                     Witaj, świecie!"))
```

Jeżeli teraz ponownie wykonamy test, zakończy się on niepowodzeniem, ponieważ jeszcze nie zdefiniowaliśmy metody `ChatClient._get_connection()` i tym samym nie możemy jej zastąpić namiastką.

```
$ python 04_chat_spies.py TestChatClient
E..
======================================================================
ERROR: test_client_connection (__main__.TestChatClient)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "04_chat_spies.py", line 35, in test_client_connection
    return_value=connection_spy):
  File "/usr/lib/python3.7/unittest/mock.py", line 1319, in __enter__
```

```
    original, local = self.get_original()
  File "/usr/lib/python3.7/unittest/mock.py", line 1293, in get_original
    "%s does not have the attribute %r" % (target, name)
AttributeError: <__main__.ChatClient object at 0x7f962dd8d050> does not
have the attribute '_get_connection'

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (errors=1)
```

Przechodzimy więc do klasy `ChatClient` i zajmiemy się zdefiniowaniem metody `_get_connection()`, której zadaniem jest zwrot nowego obiektu `Connection` razem z podanym numerem portu. Serwer będzie lokalnie nasłuchiwał na tym porcie. (W normalnych warunkach port i host byłyby konfigurowalne, ale skoro to jest tylko przykładowa aplikacja czatu, możemy przyjąć założenie, że serwer nasłuchuje połączeń na znanym adresie hosta i znanym porcie).

```
class ChatClient:
    def __init__(self, nickname):
        self.nickname = nickname

    def send_message(self, message):
        sent_message = "{}: {}".format(self.nickname, message)
        self.connection.broadcast(message)
        return sent_message

    def _get_connection(self):
        return Connection(("localhost", 9090))
```

Doskonale! To powinno pozwolić na zaliczenie testu. Skoro namiastka znajduje się we właściwym miejscu, możemy sprawdzić, jaki będzie wynik wykonania testu.

```
$ python 04_chat_spies.py TestChatClient
E..
======================================================================
ERROR: test_client_connection (__main__.TestChatClient)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "04_chat_spies.py", line 36, in test_client_connection
    client.send_message("Witaj, świecie!")
  File "04_chat_spies.py", line 83, in send_message
    self.connection.broadcast(message)
AttributeError: 'ChatClient' object has no attribute 'connection'

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (errors=1)
```

Znów się nie udało. Wprawdzie metoda `_get_connection()` została zdefiniowana, ale egzemplarz `ChatClient` nigdy jej nie wywołuje. To oznacza, że obiekt wciąż nie ma atrybutu `connection`.

Skoro oczekujemy nawiązania połączenia z opóźnieniem, to musimy zdefiniować właściwość wywołującą metodę `_get_connection()` podczas jej pierwszego użycia.

```
class ChatClient:
    def __init__(self, nickname):
        self.nickname = nickname
        self._connection = None

    def send_message(self, message):
        sent_message = "{}: {}".format(self.nickname, message)
        self.connection.broadcast(message)
        return sent_message

    @property
    def connection(self):
        if self._connection is None:
            self._connection = self._get_connection()
        return self._connection

    @connection.setter
    def connection(self, value):
        if self._connection is not None:
            self._connection.close()
        self._connection = value

    def _get_connection(self):
        return Connection(("localhost", 9090))
```

Teraz po uzyskaniu dostępu do właściwości `ChatClient.connection` okazuje się, że jej wartością jest `None`, więc następuje wywołanie metody `ChatClient._get_connection()` w celu nawiązania nowego połączenia.

Wszystkie elementy układanki powinny już znajdować się na swoim miejscu. Sprawdzamy, czy zdefiniowany wcześniej test wreszcie będzie zaliczony.

```
$ python 04_chat_spies.py TestChatClient
F..
======================================================================
FAIL: test_client_connection (__main__.TestChatClient)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "04_chat_spies.py", line 38, in test_client_connection
    assert connection_spy.broadcast.assert_called_with(("Użytkownik 1: Witaj,
świecie!"))
  File "/usr/lib/python3.7/unittest/mock.py", line 873, in assert_called_with
    raise AssertionError(_error_message()) from cause
AssertionError: Expected call: broadcast('Użytkownik 1: Witaj, świecie!')
Actual call: broadcast('Witaj, świecie!')

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

Niespodziewanie test zakończył się niepowodzeniem. Dobrą wiadomością jest jednak to, że połączenie działa. Test mógł wykorzystać namiastkę i komponent szpiega.

Natomiast złą wiadomością jest to, że w kodzie istnieje błąd, który wcześniej nie został wychwycony przez test zdefiniowany w metodzie `TestChatClient.test_send_message()`. W obecnej implementacji metody `ChatClient.send_message()` tworzona wiadomość zawiera nazwę wysyłającego ją użytkownika, natomiast wiadomość przekazywana dalej jest już pozbawiona tej nazwy. Dlatego żaden z użytkowników korzystających z czatu nie wie, kto jest autorem danej wiadomości.

```
class ChatClient:
    ...
    def send_message(self, message):
        sent_message = "{}: {}".format(self.nickname, message)
        self.connection.broadcast(message)
        return sent_message
```

Trzeba więc zmodyfikować metodę `send_message()` w taki sposób, aby wiadomość była przekazywana wraz z nazwą użytkownika będącego jej autorem. To oznacza użycie w wywołaniu metody `broadcast()` zmiennej `sent_message` zamiast `message`.

```
class ChatClient:
    ...
    def send_message(self, message):
        sent_message = "{}: {}".format(self.nickname, message)
        self.connection.broadcast(sent_message)
        return sent_message
```

Po usunięciu błędu test wreszcie zostaje zaliczony i otrzymujemy potwierdzenie, że egzemplarz `ChatClient` ma dostęp do połączenia i może za jego pomocą prawidłowo wysyłać wiadomości.

```
$ python 04_chat_spies.py TestChatClient
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Następnym krokiem jest jak zwykle powrót do testu akceptacji.

```
$ python 04_chat_spies.py TestChatAcceptance
E
======================================================================
ERROR: test_message_exchange (__main__.TestChatAcceptance)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "04_chat_spies.py", line 10, in test_message_exchange
    user1.send_message("Witaj, świecie!")
  File "04_chat_spies.py", line 58, in send_message
    self.connection.broadcast(sent_message)
  File "04_chat_spies.py", line 64, in connection
    self._connection = self._get_connection()
```

```
  File "04_chat_spies.py", line 74, in _get_connection
    return Connection(("localhost", 9090))
  File "04_chat_spies.py", line 82, in __init__
    self.connect()
  File "/usr/lib/python3.7/multiprocessing/managers.py", line 532, in connect
    conn = Client(self._address, authkey=self._authkey)
  File "/usr/lib/python3.7/multiprocessing/connection.py", line 492, in Client
    c = SocketClient(address)
  File "/usr/lib/python3.7/multiprocessing/connection.py", line 619, in
SocketClient
    s.connect(address)
ConnectionRefusedError: [Errno 111] Connection refused

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Ten test akceptacji potwierdza, że zgodnie z oczekiwaniami klient próbuje nawiązać połączenie z serwerem, co jest doskonałą wiadomością.

Jednak problem polega na tym, że serwer jeszcze nie istnieje. Dlatego test akceptacji nie może zostać zaliczony, ponieważ nie ma możliwości nawiązania połączenia z serwerem i faktycznego potwierdzenia, że klient może wysyłać i otrzymywać wiadomości.

Zanim przejdziemy dalej i zajmiemy się implementacją serwera, warto zapoznać się z koncepcją imitacji, która oferuje wszystkie potężne możliwości wcześniej omówionych dublerów używanych podczas testu.

Stosowanie imitacji

Jak można było wcześniej zauważyć, podczas pracy z obiektami pozornymi, namiastkami i szpiegami ostatecznie zawsze korzystaliśmy z modułu `unittest.mock`. To wynika z tego, że **obiekt imitacji** (ang. *mock object*) może być postrzegany jako obiekt pozorny oferujący pewne namiastki w połączeniu ze szpiegami.

Obiekt imitacji może być przekazywany, zwykle nie wykonuje żadnych działań i zachowuje się praktycznie tak samo jak obiekt pozorny.

Jeżeli istnieje funkcja `read_file()` akceptująca obiekt pliku zawierający zdefiniowaną metodę `read()`, wówczas zamiast rzeczywistego pliku można dostarczyć obiekt `Mock`, a metoda `Mock.read()` nie będzie wykonywała żadnego zadania.

```
>>> def read_file(f):
...     print("ODCZYT ZAWARTOŚCI PLIKU")
...     return f.read()
...
>>> from unittest.mock import Mock
```

```
>>> m = Mock()
>>> read_file(m)
ODCZYT ZAWARTOŚCI PLIKU
```

Natomiast jeśli zamiast powstrzymywać się od działania obiekt ma funkcjonować podobnie do namiastki, to w omawianym przykładzie można przygotować odpowiedź, aby wartością zwrotną wywołania `Mock.read()` był ciąg tekstowy:

```
>>> m.read.return_value = "Witaj, świecie!"
>>> print(read_file(m))
Witaj, świecie!
```

Jeżeli nie chcesz miejsca przeznaczonego dla rzeczywistych obiektów wypełniać jedynie obiektami pozornymi i namiastkami, wówczas imitacje możesz wykorzystać do śledzenia, co tak naprawdę się dzieje. W takim przypadku działanie imitacji będzie podobne do działania szpiega.

```
>>> m.read.call_count
2
```

Jednak cechą charakterystyczną imitacji jest możliwość sprawdzenia sposobu zachowania programu. W przypadku namiastek, szpiegów i obiektów pozornych wszystko wiąże się z informacjami o stanie. Komponenty te dostarczają informacje o stanie działania programu po wstrzyknięciu znanego stanu lub sprawdzeniu stanu, gdy jest używany komponent szpiega.

Natomiast imitacje zwykle monitorują zachowanie. Gdy oprogramowanie nie działa zgodnie z oczekiwaniami, wówczas imitacja przeważnie ulega awarii. Dlatego imitacja jest sprawdzana, czy została użyta w ściśle określony sposób, co pozwala potwierdzić zgodne z oczekiwaniami działanie oprogramowania.

Na przykład można sprawdzić, czy metoda `read()` obiektu `Mock` faktycznie została wywołana.

```
>>> m.read.assert_called_with()
```

Jeżeli zachodzi potrzeba zweryfikowania, czy funkcja `read_file()` wywołała metodę `f.read()` z określonym argumentem, można to sprawdzić za pomocą imitacji. Jeżeli ta metoda nie została wywołana, zostanie wygenerowany komunikat błędu `AssertionError`.

```
>>> m.read.assert_called_with("dowolny argument")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.7/unittest/mock.py", line 873, in assert_called_with
    raise AssertionError(_error_message()) from cause
AssertionError: Expected call: read('dowolny argument')
Actual call: read()
```

Jeżeli wspomniana wcześniej metoda nie została wywołana ze względu na błąd lub niepełną implementację, asercja zostanie wykryta i wówczas można rozwiązać problem przez zmodyfikowanie funkcji `read_file()` w taki sposób, aby działała zgodnie z oczekiwaniami.

Skoro wiesz już nieco o obiektach pozornych, namiastkach, szpiegach i imitacjach, tak naprawdę znasz wiele sposobów na przetestowanie oprogramowania bez konieczności bazowania

na w pełni gotowych i funkcjonujących komponentach. Doskonale wiesz również, że zbiór testów musi zawierać testy szybko wykonywane, łatwe do debugowania i wymagające minimum zależności przy minimalnym wpływie na system zewnętrzny.

Dlatego też rzeczywisty serwer wskazuje na konieczność uruchomienia oddzielnego procesu serwera za każdym razem, gdy zachodzi potrzeba wykonania testów, co niewątpliwie będzie spowalniało testy ze względu na korzystanie z dostępnego połączenia sieciowego.

W kolejnym kroku zamiast implementować rzeczywisty serwer przedstawię koncepcję tzw. atrap i pokażę, jak za pomocą atrapy serwera zaliczyć test akceptacji.

Zastępowanie zależności atrapami

Atrapa to na tyle dobry zamiennik dla rzeczywistej zależności, że sugeruje pracę z rzeczywistym obiektem. Atrapy są często stosowane dla uproszczenia zależności zbioru testów bądź poprawienia jego wydajności działania. Na przykład jeśli działanie oprogramowania zależy od dostępnego w chmurze zewnętrznego API prognozy pogody, wówczas używanie rzeczywistego połączenia sieciowego z zewnętrznym serwerem API nie będzie zbyt wygodnym rozwiązaniem. W najlepszym wypadku będzie to działało bardzo wolno, w najgorszym zaś dostęp do API zostanie ograniczony lub wręcz zablokowany ze względu na wykonywanie zbyt wielu żądań API w krótkim czasie — zbiór testów bardzo łatwo może osiągnąć wielkość setek bądź nawet tysięcy testów.

Najbardziej rozpowszechniony rodzaj atrap to zwykle działające w pamięci bazy danych, ponieważ upraszczają one zadania związane z przygotowaniem rzeczywistego systemu baz danych przed przeprowadzeniem testów i później zadania związane z uprzątnięciem środowiska po testach.

W omawianym przykładzie chcemy uniknąć konieczności uruchomienia serwera czatu za każdym razem, gdy ma zostać wykonany zbiór testów w aplikacji. Dlatego dostarczymy atrapy serwera i połączenia, których zadaniem będzie zastąpienie rzeczywistego połączenia sieciowego.

Skoro istnieje zbiór testów `TestConnection` przeznaczony do weryfikacji połączenia, to może pojawić się pytanie, w jaki sposób po tej drugiej stronie należy sprawdzić działanie serwera.

W tym celu można przeanalizować sposób działania serwera `SyncManager` i dostarczyć atrapę zapewniającą obsługę podstaw protokołu i odpowiedzi udzielanych na żądania. Na szczęście protokół `SyncManager` jest dość prosty. Otrzymuje on polecenia razem ze zbiorem argumentów i udziela odpowiedzi w postaci krotki `("RESPONSE_TYPE", RESPONSE)`, w której `RESPONSE_TYPE` wskazuje, czy udzielona odpowiedź składa się z wartości zwrotnej dla danego polecenia, czy z komunikatu błędu.

To pozwala utworzyć egzemplarz `FakeServer` dostarczający metodę `FakeServer.send()` przechwytującą polecenia wydawane przez klienta i metodę `FakeServer.recv()` udzielającą odpowiedzi klientowi.

```
class FakeServer:
    def __init__(self):
        self.last_command = None
        self.last_args = None
        self.messages = []

    def __call__(self, *args, **kwargs):
        # Spowodowanie, aby egzemplarz SyncManager przyjął założenie, że zostało utworzone nowe
połączenie
        return self

    def send(self, data):
        # Monitorowanie wszelkich poleceń przekazywanych do serwera
        callid, command, args, kwargs = data
        self.last_command = command
        self.last_args = args

    def recv(self, *args, **kwargs):
        # W tym momencie nie obsługujemy jeszcze żadnych poleceń, lecz jedynie komunikat błędu
        return "#ERROR", ValueError("%s - %r" % (
            self.last_command,self.last_args)
        )

    def close(self):
        pass
```

To jest pierwsza wersja bardzo prostej implementacji atrapy serwera, który na wszystkie polecenia będzie udzielał odpowiedzi w postaci komunikatu błędu. Dlatego można śledzić polecenia wysyłane przez klienty do serwera.

Aby przetestować połączenie z serwerem, do zbioru testów `TestConnection` należy dodać metodę `test_exchange_with_server()`, która dostarczony egzemplarz `FakeServer` wykorzysta do powiązania ze sobą dwóch połączeń.

```
class TestConnection(unittest.TestCase):
    def test_broadcast(self):
        ...
    def test_exchange_with_server(self):
         with unittest.mock.patch(
             "multiprocessing.managers.listener_client",
             new={"pickle": (None, FakeServer())}
         ):
            c1 = Connection(("localhost", 9090))
            c2 = Connection(("localhost", 9090))

            c1.broadcast("connected message")
            assert c2.get_messages()[-1] == "connected message"
```

Ten test wymaga wywołania `unittest.mock.patch()` w celu zastąpienia standardowej implementacji kanału komunikacji klient – serwer w `multiprocessing.managers` przygotowanym egzemplarzem `FakeServer`. W praktyce oparty na „pickle" kanał komunikacji zastępujemy innym na czas trwania testu.

Jeżeli teraz test zostanie ponownie wykonany, oczekiwanym wynikiem jest to, że atrapa serwera rozpocznie monitorowanie poleceń wydawanych przez klienty.

```
$ python 05_chat_fakes.py TestConnection
.E
======================================================================
ERROR: test_exchange_with_server (__main__.TestConnection)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "05_chat_fakes.py", line 56, in test_exchange_with_server
    c1 = Connection(("localhost", 9090))
  File "05_chat_fakes.py", line 100, in __init__
    self.connect()
  File "/usr/lib/python3.7/multiprocessing/managers.py", line 533, in connect
    dispatch(conn, None, 'dummy')
  File "/usr/lib/python3.7/multiprocessing/managers.py", line 82, in dispatch
    raise convert_to_error(kind, result)
ValueError: dummy - ()

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (errors=1)
```

Test zakończył się niepowodzeniem ze względu na nierozpoznane polecenie `dummy` (aktualnie nie są rozpoznawane żadne polecenia). Mimo to zyskujemy potwierdzenie, że dostarczona atrapa serwera znajduje się we właściwym miejscu i jest używana przez obiekt `Connection`.

W tym momencie można rozpocząć implementację obsługi polecenia `dummy` (zostało użyte po prostu w celu nawiązania połączenia) i sprawdzenia, jaki będzie wynik.

```
class FakeServer:
    ...
    def recv(self, *args, **kwargs):
        if self.last_command == "dummy":
            return "#RETURN", None
        else:
            return "#ERROR", ValueError("%s - %r" % (
                self.last_command,self.last_args)
            )
```

Wykonanie zbioru testów `TestConnection` spowoduje wywołanie następnego polecenia (tego po zaimplementowanym przed chwilą `dummy`) i wyświetlenie komunikatu błędu informującego o kolejnym brakującym poleceniu.

```
$ python 05_chat_fakes.py TestConnection
...
ValueError: create - ('get_messages',)
```

Wielokrotne wykonanie testów aż do chwili ich zaliczenia pozwala wychwycić wszystkie polecenia, które powinny być obsługiwane przez atrapę `FakeServer` w metodzie `FakeServer.recv()`. W ten sposób będzie można zaimplementować wystarczającą liczbę poleceń, aby otrzymać w miarę pełny egzemplarz `FakeServer`.

```
class FakeServer:
    ...
    def recv(self, *args, **kwargs):
        if self.last_command == "dummy":
            return "#RETURN", None
        elif self.last_command == "create":
            return "#RETURN", ("fakeid", tuple())
        elif self.last_command == "append":
            self.messages.append(self.last_args[0])
            return "#RETURN", None
        elif self.last_command == "__getitem__":
            return "#RETURN", self.messages[self.last_args[0]]
        elif self.last_command in ("incref", "decref",
                                   "accept_connection"):
            return "#RETURN", None
        else:
            return "#ERROR", ValueError("%s - %r" % (
                self.last_command,self.last_args)
            )
```

Na tym etapie zbiór testów powinien zostać zaliczony dzięki istnieniu atrapy serwera i przygotowaniu połączenia między dwoma obiektami `Connection`.

```
$ python 05_chat_fakes.py TestConnection
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Utworzony tutaj egzemplarz `FakeServer` pozwolił potwierdzić, że dwa obiekty `Connection` mogą się ze sobą komunikować i uzyskiwać dostęp do przekazywanych przez nie wiadomości. To okazało się możliwe bez konieczności uruchamiania egzemplarza serwera, nasłuchiwania sieci pod kątem połączeń z czatem itd.

Wprawdzie atrapy są zwykle bardzo wygodne, ale jednocześnie ich implementacja wymaga dość dużego wysiłku. W celu zapewnienia sensownej użyteczności atrapa musi oferować znaczną część funkcjonalności zapewnianych przez rzeczywistą zależność. Ponadto jak można było zobaczyć na omówionym przykładzie, implementacja atrapy może obejmować zastosowanie inżynierii odwrotnej w celu ustalenia sposobu działania komponentu oprogramowania, który próbujemy zastąpić.

Na szczęście dostępne są biblioteki zawierające już gotowe do użycia implementacje dla najczęściej stosowanych atrap serwerów SQL, MongoDB, S3 itd. Wystarczy zainstalować odpowiednią bibliotekę.

Co prawda atrapa przygotowana w omawianym przykładzie działa dobrze, ale najgorszym elementem rozwiązania jest prawdopodobnie sposób użycia modułu `multiprocessing`.

Ten problem wynika z tego, że obiekt `Connection` (bazujący na `SyncManager`) nie zapewnia prawidłowej obsługi mechanizmu wstrzykiwania zależności, która pozwoliłaby na poprawne

wstrzyknięcie opracowanego tutaj kanału komunikacji zamiast konieczności modyfikowania kanału bazującego na „pickle".

Jednak zanim przejdziesz dalej i zobaczysz, jak zapewnić obsługę wstrzykiwania zależności, najpierw dokończymy naszą aplikację czatu, aby został zaliczony test akceptacji.

Testy akceptacji i dublery używane podczas testów

Obiekt `Connection` współdziała z atrapą w postaci egzemplarza `FakeServer`, ale czy to oznacza wreszcie zaliczenie testu akceptacji? Niestety nie. Nadal trzeba dostarczyć serwer (atrapę lub rzeczywisty) i dokończyć implementację klienta.

Zadaniem testów akceptacji jest sprawdzenie, czy po przekazaniu użytkownikom oprogramowanie faktycznie będzie działało zgodnie z oczekiwaniami. Dlatego zwykle dobrym rozwiązaniem jest ograniczenie używania dublerów w kontekście testów akceptacji. Takie testy powinny jak najściślej odzwierciedlać faktyczny sposób użycia danego oprogramowania.

Imitacje, namiastki i obiekty pozorne są rzadko spotykane w testach akceptacji, w przeciwieństwie do atrap, które dość często pojawiają się w tym kontekście. Ponieważ atrapa ma naśladować zachowanie rzeczywistej usługi, z perspektywy oprogramowania nie powinno być różnicy między używaniem atrapy i faktycznej usługi. Jeżeli w testach akceptacji były używane atrapy, warto zdefiniować serię testów systemowych pozwalających na weryfikację oprogramowania, którego działanie zależy od rzeczywistych usług (ze względu na koszt te testy mogą być wykonywane przed wydaniem oprogramowania).

W omawianym przykładzie chcemy, aby test akceptacji działał z rzeczywistym serwerem. Dlatego też zmodyfikujemy go nieco, aby uruchamiał serwer i umożliwiał klientom nawiązywanie połączeń z tym nowym serwerem. Skoro nasz serwer jest implementowany na bazie `SyncManager`, to podobnie jak wszystkie egzemplarze `SyncManager` może być uruchamiany i zatrzymywany za pomocą menedżera kontekstu w konstrukcji `with`.

W kontekście wywołania `with new_chat_server()` serwer będzie uruchomiony, a po opuszczeniu kontekstu serwer zostanie zatrzymany.

```
class TestChatAcceptance(unittest.TestCase):
    def test_message_exchange(self):
        with new_chat_server():
            user1 = ChatClient("Jan Kowalski")
            user2 = ChatClient("Harry Potter")

            user1.send_message("Witaj, świecie!")
            messages = user2.fetch_messages()
            assert messages == ["Jan Kowalski: Witaj, świecie!"]
```

Oczywiście wykonanie testu teraz zakończy się niepowodzeniem z powodu braku funkcji `new_chat_server()`, której zadaniem jest przekazanie serwera przeznaczonego do użycia w teście.

Serwer będzie po prostu podklasą klasy `SyncManager`, dostarczającą listę wiadomości (za pomocą metody `_srv_get_messages()`), aby klienty mogły uzyskać do niej dostęp.

```
_messages = []
def _srv_get_messages():
    return _messages
class _ChatServerManager(SyncManager):
    pass
_ChatServerManager.register("get_messages",
                            callable=_srv_get_messages,
                            proxytype=ListProxy)
def new_chat_server():
    return _ChatServerManager(("", 9090), authkey=b'mychatsecret')
```

W ten sposób zdefiniowaliśmy funkcję `new_chat_server()`, która może być używana do uruchamiania serwera. Następnym krokiem jest jak zwykle sprawdzenie, czy testy zostaną zaliczone.

```
$ python 06_acceptance_tests.py TestChatAcceptance
E
======================================================================
ERROR: test_message_exchange (__main__.TestChatAcceptance)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "06_dependency_injection.py", line 12, in test_message_exchange
    messages = user2.fetch_messages()
AttributeError: 'ChatClient' object has no attribute 'fetch_messages'

----------------------------------------------------------------------
Ran 1 test in 0.011s

FAILED (errors=1)
```

Test nie został zaliczony z powodu braku implementacji ostatniego fragmentu klienta, czyli komponentu odpowiedzialnego za pobieranie wiadomości. Do klienta trzeba dodać metodę `fetch_messages()` i sprawdzić, czy teraz wszystko działa zgodnie z oczekiwaniami.

Pracę jak zwykle rozpoczynamy od zdefiniowania testu dla jednostki `ChatClient.send_message()`, aby móc sprawdzić, czy implementacja działa jak należy.

```
class TestChatClient(unittest.TestCase):
    ...
    def test_client_fetch_messages(self):
        client = ChatClient("Użytkownik 1")
        client.connection = unittest.mock.Mock()
        client.connection.get_messages.return_value = ["message1",
        starting_messages = client.fetch_messages()
        client.connection.get_messages().append("message3")
        new_messages = client.fetch_messages()
```

```
        assert starting_messages == ["message1", "message2"]
        assert new_messages == ["message3"]
```

Ponieważ metoda `ChatClient.fetch_messages()` jeszcze nie istnieje, test natychmiast zakończy się niepowodzeniem.

```
$ python 06_acceptance_tests.py TestChatClient
.E..
======================================================================
ERROR: test_client_fetch_messages (__main__.TestChatClient)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "06_dependency_injection.py", line 46, in test_client_fetch_messages
    starting_messages = client.fetch_messages()
AttributeError: 'ChatClient' object has no attribute 'fetch_messages'

----------------------------------------------------------------------
Ran 4 tests in 0.001s

FAILED (errors=1)
```

Trzeba powrócić do klasy `ChatClient` i zaimplementować metodę `fetch_messages()` w sposób pozwalający na zaliczenie testu.

```
class ChatClient:
    def __init__(self, nickname):
        self.nickname = nickname
        self._connection = None
        self._last_msg_idx = 0

    def send_message(self, message):
        sent_message = "{}: {}".format(self.nickname, message)
        self.connection.broadcast(sent_message)
        return sent_message

    def fetch_messages(self):
        messages = list(self.connection.get_messages())
        new_messages = messages[self._last_msg_idx:]
        self._last_msg_idx = len(messages)
        return new_messages
```

Nowa metoda `ChatClient.fetch_messages()` pobiera wiadomości przechowywane na serwerze i zwraca te, które pojawiły się od chwili wykonania poprzedniej operacji sprawdzenia nowych wiadomości.

Jeżeli implementacja jest poprawna, ponowne wykonanie testu spowoduje jego zaliczenie, co potwierdza działanie metody zgodnie z oczekiwaniami.

```
$ python 06_acceptance_tests.py TestChatClient
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

Skoro to jest ostatni element układanki, test akceptacji powinien zostać teraz zaliczony i potwierdzić poprawność działania aplikacji czatu.

```
$ python 06_acceptance_tests.py TestChatAcceptance
.
----------------------------------------------------------------------
Ran 1 test in 0.016s

OK
```

Hurra! Wreszcie można ogłosić zwycięstwo. Budowana tutaj aplikacja działa z rzeczywistymi klientami i serwerem. Komponenty te mają możliwość nawiązania połączenia i komunikowania się ze sobą, co potwierdza poprawność utworzenia oprogramowania.

Mimo to testy klasy `ChatClient` zawierają dość skomplikowany kod wykorzystujący wywołanie `mock.patchc()` do zastępowania pewnych fragmentów. Konieczne okazało się również zaimplementowanie metody typu setter dla właściwości połączenia, i to tylko w celu uzyskania możliwości użycia dublera testu.

Wprawdzie cel został osiągnięty, ale musi istnieć lepszy sposób na zastosowanie dublerów w kodzie, bez konieczności stosowania w nim wielu wywołań `mock.patch()`.

Zastępowanie na żądanie komponentów systemu to zadanie dla mechanizmu wstrzykiwania zależności. Sprawdzimy więc, czy ten mechanizm pomoże w przełączaniu między atrapami i rzeczywistymi usługami w zbiorze testów.

Zarządzanie zależnościami za pomocą mechanizmu wstrzykiwania zależności

Opracowane tutaj rozwiązanie dotyczące klasy `ChatClient` przeznaczone do nawiązywania połączenia z serwerem jest znacznie bardziej skomplikowane niż to konieczne. Praktycznie jedynym przeznaczeniem metod typu setter, `ChatClient.get_connection()` i `ChatClient.connection()`, jest ułatwienie podczas zastępowania imitacjami połączeń konfigurowanych przez klienta.

To wynika z tego, że egzemplarz `ChatClient` ma zależność od obiektu `Connection`, którą samodzielnie próbuje spełnić. Można to porównać do sytuacji, gdy chcesz coś zjeść. Pierwszą możliwością jest wykorzystanie produktów znajdujących się w lodówce i samodzielne ugotowanie posiłku. Druga możliwość to zamówienie posiłku na wynos w restauracji.

Mechanizm wstrzykiwania zależności można porównać do drugiego z tych rozwiązań (zamówienie posiłku w restauracji). Jeżeli egzemplarz `ChatClient` wymaga połączenia, wówczas zamiast próbować je nawiązać samodzielnie zadanie to powinien zlecić innemu komponentowi i następnie wykorzystać otrzymane połączenie.

W większości mechanizmów wstrzykiwania zależności mamy tzw. **iniektor**, którego zadaniem jest pobranie odpowiedniego obiektu i dostarczenie go klientowi. Klient zwykle nie ma żadnych

informacji dotyczących używanego iniektora. Cały proces najczęściej odbywa się z użyciem dość zaawansowanych frameworków dostarczających rejestr usług i pozwalających klientom rejestrować te usługi. Istnieje także bardzo prosta postać mechanizmu wstrzykiwania zależności, która sprawdza się doskonale i może być stosowana bez żadnych dodatkowych zależności zewnętrznych lub frameworków: wstrzykiwanie zależności podczas tworzenia obiektu.

Wstrzykiwanie zależności podczas tworzenia obiektu oznacza, że usługa wymagana do działania kodu jest dostarczana w postaci parametru podczas tworzenia klasy zależności.

W omawianym przykładzie można dość łatwo przeprowadzić refaktoryzację klasy `ChatClient`, aby akceptował argument `connection_provider`, co pozwoli uprościć implementację `ChatClient` i pozbyć się jej znacznych fragmentów.

```
class ChatClient:
    def __init__(self, nickname, connection_provider=Connection):
        self.nickname = nickname
        self._connection = None
        self._connection_provider = connection_provider
        self._last_msg_idx = 0

    def send_message(self, message):
        sent_message = "{}: {}".format(self.nickname, message)
        self.connection.broadcast(sent_message)
        return sent_message

    def fetch_messages(self):
        messages = list(self.connection.get_messages())
        new_messages = messages[self._last_msg_idx:]
        self._last_msg_idx = len(messages)
        return new_messages

    @property
    def connection(self):
        if self._connection is None:
            self._connection = self._connection_provider(("localhost",
                                                          9090))
        return self._connection
```

Pomimo pozbycia się wywołania `ChatClient.get_connection()` i `connection @property.setter` nie utraciliśmy żadnej funkcjonalności, a także nie zwiększyliśmy poziomu skomplikowania. W większości przypadków egzemplarz klasy `ChatClient` może być używany w dokładnie taki sam sposób jak wcześniej, a kod sam dba o wykorzystanie odpowiedniego obiektu `Connection`.

Jednak zdarzają się sytuacje, w których chcemy przygotować nieco inne rozwiązanie. Wówczas można wstrzykiwać inne rodzaje połączeń.

W teście zdefiniowanym w metodzie `TestChatClient.test_client_connection()` można usunąć trudne w odczycie wywołanie `mock.patch()`, które zostało użyte do skonfigurowania szpiega.

```
class TestChatClient(unittest.TestCase):
    def test_client_connection(self):
        client = ChatClient("Użytkownik 1")
```

```
        connection_spy = unittest.mock.MagicMock()
        with unittest.mock.patch.object
          (client, "_get_connection",return_value=connection_spy):
            client.send_message("Witaj, świecie!")

        connection_spy.broadcast.assert_called_with(("Użytkownik 1:
                                                     Witaj, świecie!"))
```

Zamiast modyfikować implementację klasy `ChatClient` można po prostu dostarczyć szpiega, który następnie zostanie użyty przez egzemplarz `ChatClient`.

```
    def test_client_connection(self):
        connection_spy = unittest.mock.MagicMock()

        client = ChatClient("Użytkownik 1", connection_provider=lambda *args:
                            connection_spy)
        client.send_message("Witaj, świecie!")

        connection_spy.broadcast.assert_called_with(("Użytkownik 1:
                                                     Witaj, świecie!"))
```

Ta wersja kodu jest znacznie łatwiejsza do odczytu i do zrozumienia oraz nie wykorzystuje „magicznych" rozwiązań w stylu modyfikowania obiektów podczas wykonywania kodu.

Dzięki użyciu mechanizmu wstrzykiwania zależności zamiast modyfikować obiekty można uprościć cały zbiór testów `TestChatClient`.

```
class TestChatClient(unittest.TestCase):
    def test_nickname(self):
        client = ChatClient("Użytkownik 1")

        assert client.nickname == "Użytkownik 1"

    def test_send_message(self):
        client = ChatClient("Użytkownik 1", connection_provider=unittest.mock.Mock())
        sent_message = client.send_message("Witaj, świecie!")
        assert sent_message == "Użytkownik 1: Witaj, świecie!"

    def test_client_connection(self):
        connection_spy = unittest.mock.MagicMock()

        client = ChatClient("Użytkownik 1", connection_provider=lambda *args:
                            connection_spy)
        client.send_message("Witaj, świecie!")

        connection_spy.broadcast.assert_called_with(("Użytkownik 1: Witaj, świecie!"))

    def test_client_fetch_messages(self):
        connection = unittest.mock.Mock()
        connection.get_messages.return_value = ["message1", "message2"]

        client = ChatClient("Użytkownik 1", connection_provider=lambda *args:
        ↪connection)
```

```
starting_messages = client.fetch_messages()
client.connection.get_messages().append("message3")
new_messages = client.fetch_messages()

assert starting_messages == ["message1", "message2"]
assert new_messages == ["message3"]
```

Wszystkie wystąpienia trudnych w odczycie wywołań `mock.patch()` zostały zastąpione przez polecenia dostarczające argument `connection_provider` w chwili tworzenia egzemplarza klasy `ChatClient`.

Tak więc mechanizm wstrzykiwania zależności ułatwił pracę podczas testowania, a jednocześnie zapewnił znacznie większą elastyczność utworzonej implementacji.

Załóżmy, że zbudowana tutaj aplikacja czatu ma działać z czymś innym niż `SyncManager`. W takim przypadku wystarczy jedynie przekazać klientom inny rodzaj argumentu `connection_provider`.

Gdy klasy mają zależności od innych budowanych przez nie obiektów, wtedy warto zadać sobie pytanie, czy użycie mechanizmu wstrzykiwania zależności nie będzie lepszym rozwiązaniem i czy te usługi nie mogą zostać wstrzyknięte z zewnątrz zamiast być wbudowane w klasie.

Stosowanie frameworków mechanizmu wstrzykiwania zależności

W Pythonie istnieje wiele frameworków mechanizmów wstrzykiwania zależności. To nie jest łatwa technika do samodzielnej implementacji, stąd istnienie różnych jej odmian w wielu frameworkach. Podstawowym zadaniem frameworka mechanizmu wstrzykiwania zależności jest powiązanie ze sobą obiektów.

We wcześniejszym rozwiązaniu zależności były wyraźnie dostarczane w trakcie każdej operacji tworzenia nowego obiektu (poza domyślną zależnością dostarczoną w postaci argumentu). Framework mechanizmu wstrzykiwania zależności będzie automatycznie wykrywał, że egzemplarz klasy `ChatClient` wymaga obiektu `Connection` i spełni to wymaganie.

W Pythonie jednym z najłatwiejszych w użyciu frameworków mechanizmu wstrzykiwania zależności jest Pinject opracowany przez firmę Google. To doskonały framework, o czym można się przekonać po analizie innych znanych frameworków firmy Google, np. Angulara.

Pinject pozwala zarządzać zależnościami w bardzo prosty i łatwy do opanowania sposób, bazujący na nazwach argumentów metody inicjalizacyjnej i nazwach klas.

Podobnie jak wcześniej także tutaj zakładamy, że istnieją dwie klasy, `ChatClient` i `Connection`. Jednak tym razem egzemplarz klasy `ChatClient` będzie informował, który obiekt `Connection` zostanie użyty. Jedynym celem tego przykładu jest pokazanie, jak Pinject potrafi samodzielnie zająć się obsługą mechanizmu wstrzykiwania zależności.

```
class ChatClient:
    def __init__(self, connection):
        print(self, "GOT", connection)

class Connection:
    pass
```

Teraz można zastosować Pinject do utworzenia wykresu zależności obiektów w naszym przykładzie.

```
import pinject
injector = pinject.new_object_graph()
```

Gdy Pinject ustali zależności dla naszych obiektów (domyślnie odbywa się to przez skanowanie wszystkich klas we wszystkich zaimportowanych modułach, klasy można również wyraźnie wymienić za pomocą argumentu `classes=`), Pinject dostarczy egzemplarz każdej znanej mu klasy i tym samym spełni wszystkie zależności.

```
>>> cli = injector.provide(ChatClient)
<ChatClient object at 0x7fad51469610> GOT <Connection object at 0x7fad51469bd0>
```

W omawianym przykładzie podczas obsługi żądania dotyczącego egzemplarza klasy `ChatClient` Pinject ustalił istnienie klasy `Connection` i zależności od argumentu `Connection`. W tym momencie framework automatycznie nawiązał połączenie i dostarczył je klientowi.

Co można zrobić w sytuacji, gdy zachodzi potrzeba użycia atrap obiektu `Connection` podczas testów? Pinject obsługuje dostarczanie specyfikacji wiązań niestandardowych, więc można wyraźnie wskazać, która klasa spełnia daną zależność.

Jeżeli mamy obiekt `FakeConnection`, można zdefiniować obiekt `pinject.BindingSpec` w celu wskazania, jak należy spełnić zależność `connection`, np.:

```
class FakeConnection:
    pass

class FakedBindingSpec(pinject.BindingSpec):
    def provide_connection(self):
        return FakeConnection()

faked_injector = pinject.new_object_graph(binding_specs=[
    FakedBindingSpec()
])
```

Jeżeli w tym momencie nastąpi próba utworzenia egzemplarza klasy `ChatClient` za pomocą egzemplarza `faked_injector`, wynikiem będzie obiekt `ChatClient` używający atrapy połączenia.

```
>>> cli = faked_injector.provide(ChatClient)
<ChatClient object at 0x7fad513ce350> GOT <FakeConnection object at
0x7fad513d6f90>
```

Trzeba w tym miejscu wspomnieć, że domyślnie framework Pinject pamięta, które obiekty zostały przez niego utworzone. Dlatego po wykonaniu żądania dotyczącego nowego obiektu

ChatClient otrzymasz dokładnie ten sam obiekt połączenia. Takie rozwiązanie okazuje się wygodne podczas tworzenia pełnych programów i zastępowania całych komponentów. Jeżeli chcesz zastąpić warstwę abstrakcji danych i wykorzystać atrapę bazy danych, prawdopodobnie dokładnie tę samą warstwę abstrakcji danych otrzymasz wszędzie tam, gdzie komponenty korzystają z tych samych danych.

Dlatego utworzenie nowego egzemplarza klasy ChatClient powoduje wygenerowanie innego obiektu ChatClient, choć używającego dokładnie tego samego obiektu Connection.

```
>>> cli = faked_injector.provide(ChatClient)
<ChatClient object at 0x7f9878aeb810> GOT <Connection object at 0x7f9878a58f50>
>>> cli2 = faked_injector.provide(ChatClient)
<ChatClient object at 0x7f9878a55fd0> GOT <Connection object at 0x7f9878a58f50>
```

W przypadku klientów budowanej tutaj aplikacji czatu prawdopodobnie lepszym rozwiązaniem będzie, aby każdy z nich używał innego połączenia z serwerem. W tym celu należy użyć egzemplarza BindingSpec i wskazać Pinjectowi, że zwrócona zależność jest prototypem, a nie obiektem typu singleton. Dzięki temu Pinject nie będzie buforować dostarczanych zależności i zawsze będzie zwracać nowe.

```
class PrototypeBindingSpec(pinject.BindingSpec):
    @pinject.provides(in_scope=pinject.PROTOTYPE)
    def provide_connection(self):
        return Connection()

proto_injector = pinject.new_object_graph(binding_specs=[
    PrototypeBindingSpec()
])
```

Jeżeli zmodyfikujemy klasę ChatClient i przystosujemy ją do użycia egzemplarza proto_inject, to zobaczymy, że teraz każdy klient korzysta z oddzielnego obiektu Connection.

```
>>> cli = proto_injector.provide(ChatClient)
<ChatClient object at 0x7fadab060e50> GOT <Connection object at 0x7fadab013910>
>>> cli2 = proto_injector.provide(ChatClient)
<ChatClient object at 0x7fadab060f10> GOT <Connection object at 0x7fadab013850>
```

Jak widzisz, framework mechanizmu wstrzykiwania zależności potrafi znacznie ułatwić pracę. Zastosowanie takiego frameworka zależy przede wszystkim od stopnia skomplikowania zależności w tworzonym oprogramowaniu. Jeżeli się zdecydujesz na wykorzystanie tego typu frameworka, zwykle zapewnisz sobie możliwość szybkiego usunięcia zależności między komponentami, gdy zachodzi taka potrzeba.

Podsumowanie

Zależności między komponentami przeznaczonymi do testowania potrafią bardzo utrudnić pracę programistom. W celu przetestowania czegokolwiek bardziej skomplikowanego niż prosta funkcja narzędziowa trzeba poradzić sobie z dziesiątkami zależności i ich stanem.

Dlatego gdy zautomatyzowane testy stały się rzeczywistością, bardzo szybko narodziła się idea używanych podczas testów dublerów w miejsce faktycznych komponentów. Możliwość zastąpienia komponentów testowanej jednostki atrapami, obiektami pozornymi, namiastkami i imitacjami znacznie ułatwiła pracę programistom, a także pozwoliła na szybsze tworzenie zbiorów testów i ich łatwiejszą obsługę.

Obecnie praktycznie każde oprogramowanie to skomplikowana sieć zależności i dlatego wielu programistów uważa, że testy integracji to najbardziej realistyczna i niezawodna postać testowania. Jednak zarządzanie skomplikowaną siecią może być trudne. Na szczęście mechanizm wstrzykiwania zależności i frameworki mechanizmów wstrzykiwania zależności potrafią bardzo ułatwić pracę pod tym względem.

W tym rozdziale wyjaśniłem, jak tworzyć automatycznie wykonywane zbiory testów, jak używać dublerów do sprawdzania komponentów w izolacji oraz jak weryfikować informacje o stanie komponentów i ich sposób działania. Dzięki temu masz wszystkie narzędzia potrzebne do zagłębienia się w model programowania sterowanego testami, którego omawianie rozpocznę w następnym rozdziale. Z jego lektury dowiesz się m.in., jak tworzyć oprogramowanie, korzystając z modelu TDD.

Najważniejsze jest testowanie kodu — od pierwszej linii!

Spośród wielu koncepcji tworzenia oprogramowania na szczególną uwagę zasługuje model programowania sterowanego testami, znany jako TDD. Technika ta opiera się na integracji procesów projektowania aplikacji i pisania kodu z prowadzeniem testów. Mimo że taki sposób pracy wydaje się dość wymagający dla zespołów deweloperów, łatwo się przekonać, że TDD pozwala na stałe uzyskiwanie dobrych efektów, a opracowane tą metodą aplikacje zaskakują stabilnością i przewidywalnością w środowisku produkcyjnym.

W tej praktycznej książce dokładnie opisano koncepcje przeprowadzania testów oprogramowania, a szczególny akcent położono na model programowania sterowanego testami. Przedstawiono w niej również szeroką gamę przydatnych do testowania narzędzi, takich jak wbudowany w Pythona moduł testów jednostkowych unittest, frameworki pytest i Robot, a także biblioteka webtest. Omówiono też zasady projektowania testów, testowania kodu podczas implementacji nowych funkcjonalności i tworzenia pełnych zbiorów testów. Ponadto dokładnie zaprezentowano najlepsze praktyki związane z testami automatycznymi i modelem programowania TDD. Poszczególne koncepcje zostały zilustrowane praktycznymi przykładami zastosowania narzędzi dostępnych w Pythonie.

W książce między innymi:

- najlepsze praktyki dotyczące projektowania testów
- praca z frameworkiem pytest przeznaczonym do testowania aplikacji
- tworzenie testów funkcjonalnych dla aplikacji WSGI za pomocą biblioteki webtest
- zasady programowania sterowanego testami
- techniki tworzenia niezawodnych aplikacji w Pythonie

Alessandro Molina — od dwóch dekad programuje w Pythonie. Pasjonuje się zastosowaniem tego języka do tworzenia aplikacji internetowych. Obecnie pracuje nad frameworkami TurboGears2 i Beaker. Opracował framework plikowej pamięci masowej DEPOT i prosty interpreter JavaScriptu dla Pythona, nazwany DukPy. Brał też udział w takich projektach Pythona jak FormEncode, ToscaWidgets i Ming MongoDB ORM.

ISBN 978-83-283-8664-8
9 788328 386648
Cena: 79,00 zł

Packt