Testowanie i Jakość Oprogramowania

Akademia Tarnowska

Katedra Informatyki

Tomasz Gądek

Wykład 12: Testy integracyjne, atrapy

Test Doubles (Atrapy)

Test Doubles (Atrapy)

- Dummy (Atrapa)
- Fake (Zaślepka)
- Stub (Podróbka)
- Spy (Szpieg)
- Mock (Imitacja)

Atrapy (Test Doubles) w Testowaniu

> Zastępowanie zależności w testach.

- Obiekty zastępujące rzeczywiste zależności (inne klasy, usługi zewnętrzne, bazy danych).
- Umożliwiają izolowanie testowanego kodu.
- Sprawiają, że testy są szybsze, bardziej przewidywalne i łatwiejsze w kontrolowaniu.

Dummy

> Przekazywany, ale nieużywany.

- Obiekt, który jest przekazywany jako argument, ale jego metody lub właściwości nie są wywoływane w testowanym kodzie.
- Służy do spełnienia sygnatury metody lub konstruktora.
- Nie ma wpływu na logikę testu.

Dummy (Implementacja)

```
class DummyLogger:
    def log(self, message):
        pass

def process_data(data, logger):
    # Dane są przetwarzane, logger nie jest używany
    return len(data)
```

Dummy (Test)

```
import unittest

class TestProcessData(unittest.TestCase):
    def test_process_data_length(self):
        data = [1, 2, 3]

        dummy_logger = DummyLogger()
        result = process_data(data, dummy_logger)

        self.assertEqual(result, 3)
```

Fake

> Działająca, ale uproszczona implementacja.

- Posiada działającą implementację zależności, ale zazwyczaj uproszczoną i działającą w pamięci.
- Łatwiejsza w konfiguracji i kontrolowaniu niż rzeczywista zależność.
- Przykład: repozytorium danych działające na liście zamiast prawdziwej bazy danych.

Fake (Implementacja)

```
class FakeUserRepository:
    def __init__(self):
        self.users = {}

    def get_user(self, user_id):
        return self.users.get(user_id)

    def add_user(self, user_id, username):
        self.users[user_id] = {"username": username}

def get_user_name(user_id, user_repo):
    user = user_repo.get_user(user_id)
    if user:
        return user["username"]
    return None
```

Fake (Test)

```
import unittest

class TestGetUserName(unittest.TestCase):
    def test_get_existing_user(self):
        fake_repo = FakeUserRepository()
        fake_repo.add_user(1, "testuser")

        name = get_user_name(1, fake_repo)

        self.assertEqual(name, "testuser")
```

Stub

> Kontrolowane dane wejściowe.

- Dostarcza zaprogramowane odpowiedzi na wywołania metod.
- Pozwala sprawdzić, jak testowany kod reaguje na różne dane.
- Nie weryfikuje, czy metody na stubie zostały wywołane.

Stub (Implementacja)

```
class StubWeatherService:
    def get_temperature(self, city):
        return 25  # Zawsze zwraca 25 stopni

def should_wear_shorts(city, weather_service):
    temperature = weather_service.get_temperature(city)
    return temperature > 20
```

Stub (Test)

```
import unittest

class TestShouldWearShorts(unittest.TestCase):
    def test_temperature_above_20(self):
        stub_weather = StubWeatherService()

        self.assertTrue(should_wear_shorts("Tarnów", stub_weather))
```

Spy

> Rejestruje informacje o użyciu.

- Atrapa, która rejestruje, jak została użyta (np. ile razy i z jakimi argumentami wywołano jej metody).
- Umożliwia weryfikację interakcji z zależnością.

Spy (Implementacja)

```
class EmailService:
    def send_email(self, recipient, subject, body):
        print(f"Wysyłam e-mail do: {recipient} z tematem: {subject}")

class SpyEmailService:
    def __init__(self):
        self.calls = []

    def send_email(self, recipient, subject, body):
        self.calls.append((recipient, subject, body))

def send_notification(user_email, item_name, email_service):
    subject = f"Powiadomienie: {item_name}"
    body = f"Dostępny jest: {item_name}."
    email_service.send_email(user_email, subject, body)
```

Spy (Test)

```
import unittest

class TestSendNotificationSpy(unittest.TestCase):
    def test_notification_sent(self):
        spy = SpyEmailService()

        send_notification("test@example.com", "Produkt X", spy)

        self.assertEqual(len(spy.calls), 1)
        self.assertEqual(spy.calls[0],
          ("test@example.com", "Powiadomienie: Produkt X", "Dostępny jest: Produkt X."))

    def test_no_notification_sent(self):
        spy = SpyEmailService()

        self.assertEqual(len(spy.calls), 0)
        self.assertEqual(spy.calls, [])
```

Mock (Fake + Spy)

> Oczekiwane interakcje są weryfikowane.

- Atrapa, w której definiujemy oczekiwania dotyczące interakcji (jakie metody, ile razy, z jakimi argumentami).
- Test kończy się niepowodzeniem, jeśli oczekiwania nie zostaną spełnione.
- Służy do weryfikacji zachowania i interakcji.

Mock (Implementacja)

```
class PaymentGateway:
    def charge(self, user_id, amount):
        print(f"Obciążam użytkownika {user_id} kwotą {amount}")

class Order:
    def __init__(self, user_id, total_amount):
        self.user_id = user_id
        self.total_amount = total_amount
        self.paid = False

    def mark_as_paid(self):
        self.paid = True

def process_order(order, payment_gateway):
    payment_gateway.charge(order.user_id, order.total_amount)
    order.mark_as_paid()
```

Mock (Test)

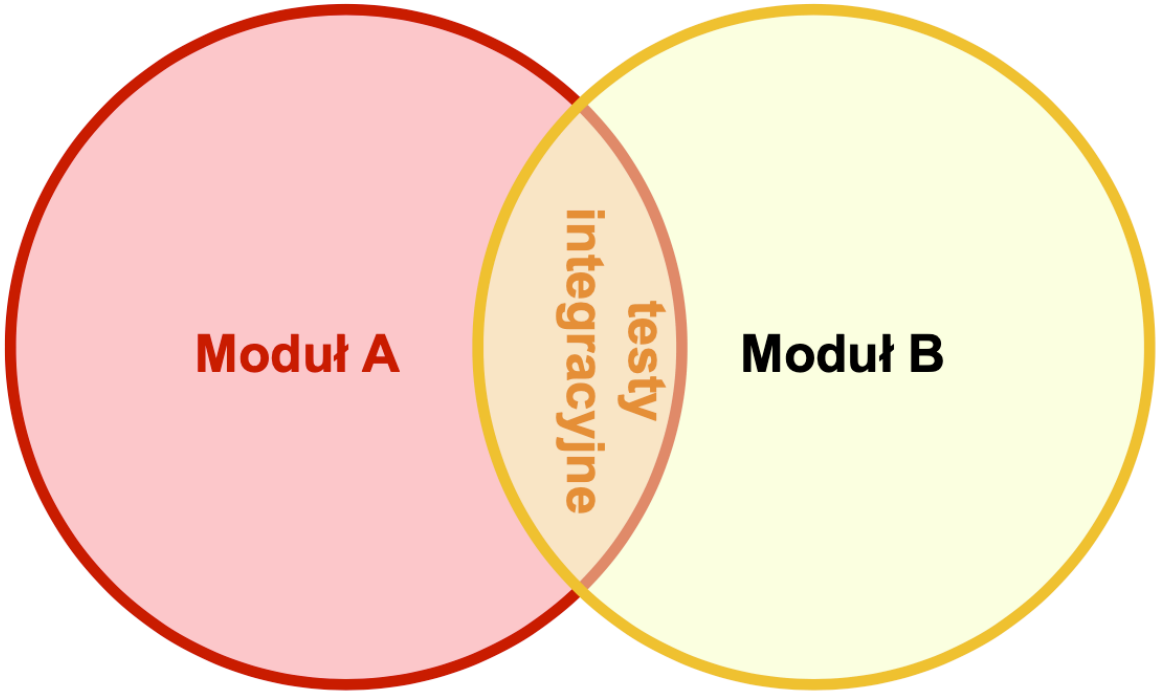
```
from unittest.mock import Mock
import unittest

class TestProcessOrder(unittest.TestCase):
    def test_order_processing(self):
        # Używamy spec, aby powiązać mock z interfejsem PaymentGateway
        mock_payment_gateway = Mock(spec=PaymentGateway)
        order = Order(123, 50.00)

        process_order(order, mock_payment_gateway)

        mock_payment_gateway.charge.assert_called_once_with(123, 50.00)
        self.assertTrue(order.paid)
```

Mock (API)

```python
from unittest.mock import patch, Mock
import unittest, requests

def get_weather_report(city):
    response = requests.get(f"https://api.weather.com/{city}")
    response.raise_for_status()
    return response.json()["temperature"]

class TestGetWeatherReport(unittest.TestCase):
    @patch('requests.get') # mock_get: Argument funkcji testowej, będący zamockowaną wersją requests.get
    def test_successful_api_call(self, mock_get):
        mock_response = Mock()
        mock_response.json.return_value = {"temperature": 28}
        mock_response.raise_for_status.return_value = None
        mock_get.return_value = mock_response

        temperature = get_weather_report("Tarnów")

        self.assertEqual(temperature, 28)
        mock_get.assert_called_once_with("https://api.weather.com/Tarnów")
```

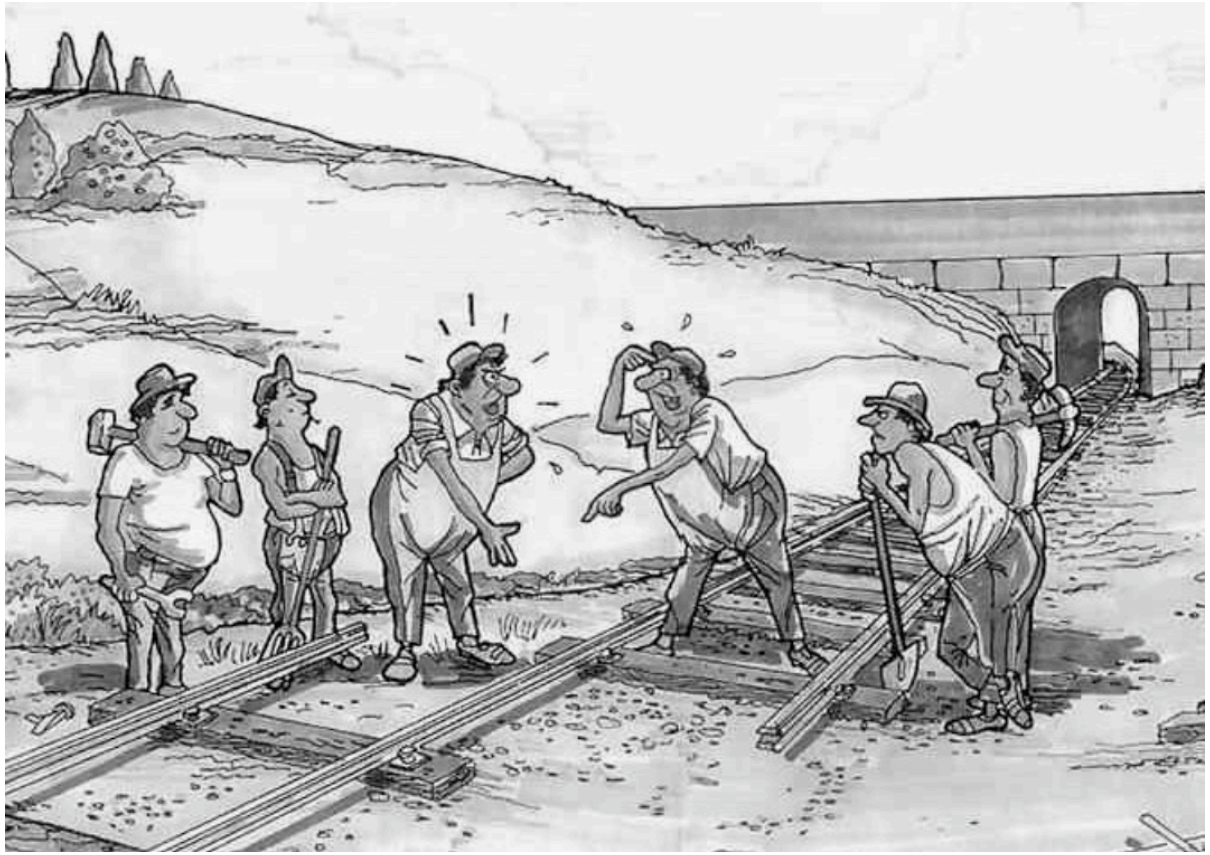
Testy integracyjne

Wykonywane są w celu wykrycia defektów w interfejsach i interakcjach pomiędzy modułami lub systemami.



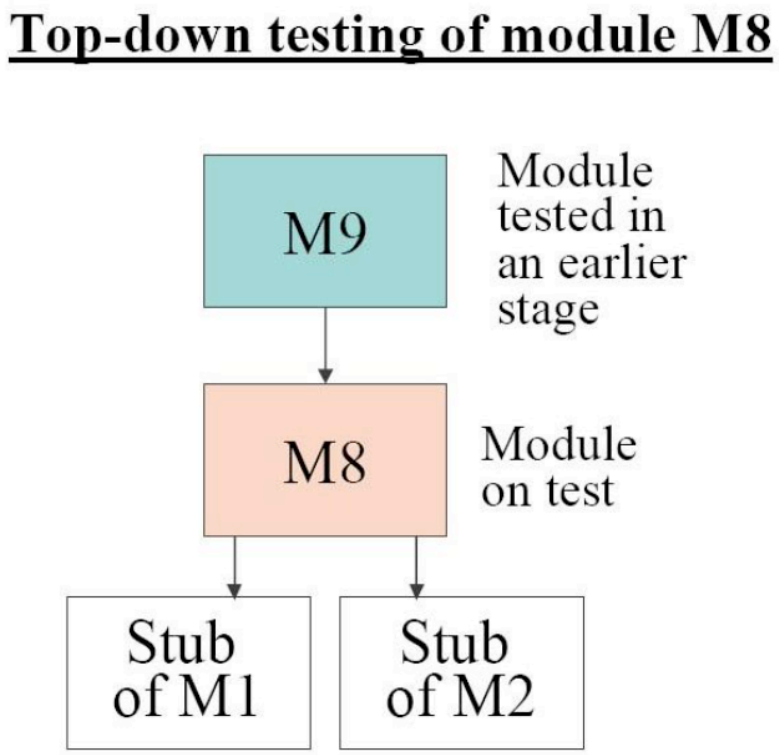
Testy integracyjne...

> posiadają wiele potencjalnych punktów awarii.



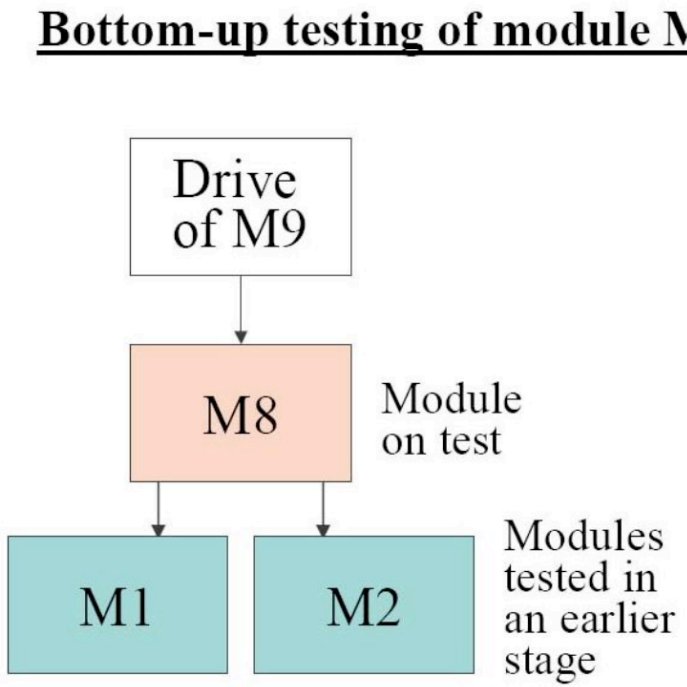
Testy integracyjne zstępujące

> Moduły znajdujące się na najwyższym poziomie są testowane jako pierwsze natomiast te położone niżej w hierarchii są symulowane są przez atrapy.



Testy integracyjne wstępujące

> Najniżej położone komponenty testowane są jako pierwsze.



Testy integracyjne (1/3)

`M1.py`

```
class UserRepository:
    def __init__(self):
        self._users = {}

    def create_user(self, user_id, username):
        self._users[user_id] = {"username": username}

    def get_user(self, user_id):
        return self._users.get(user_id)

    def clear_users(self):
        self._users.clear()
```

Testy integracyjne (2/3)

`M2.py`

```
from M1 import UserRepository

class UserService:
    def __init__(self, user_repository):
        self._user_repository = user_repository

    def register_and_get_username(self, user_id, username):
        self._user_repository.create_user(user_id, username)
        user_data = self._user_repository.get_user(user_id)
        if user_data:
            return user_data["username"]
        return None
```

Testy integracyjne (3/3)

`test_integration.py`

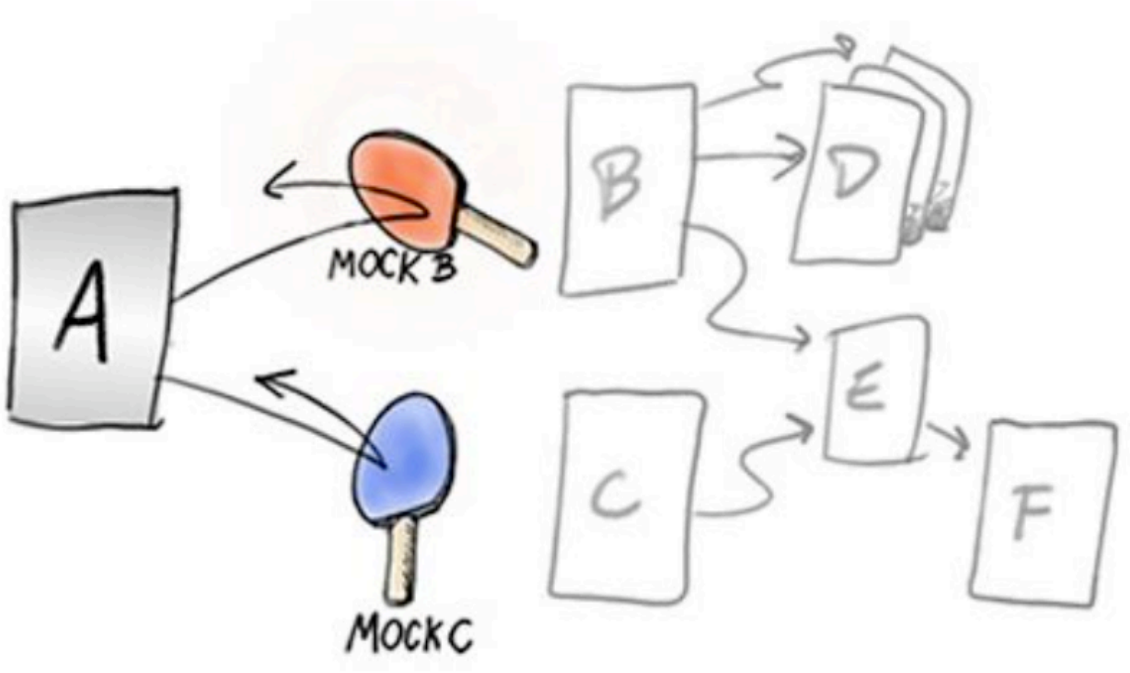
```
import unittest
from M2 import UserService
from M1 import UserRepository

class TestUserServiceIntegration(unittest.TestCase):

    def test_user_registration_and_retrieval(self):
        user_repo = UserRepository()
        user_service = UserService(user_repo)
        user_id = 1
        username = "test_user"

        retrieved_username = user_service.register_and_get_username(user_id, username)
        user_data = user_repo.get_user(user_id)

        self.assertEqual(retrieved_username, username)
        self.assertIsNotNone(user_data)
        self.assertEqual(user_data["username"], username)
```

A gdyby tak...

> UserRepository było atrapą???

Testy jednostkowe, powinny...

> być wyizolowane.



Koniec