Issue 15, 2024

https://doi.org/10.23939/sisn2024.15.210

УДК 004.05

ТЕСТУВАННЯ НА ОСНОВІ КОНТРАКТІВ З ВИКОРИСТАННЯМ ОНТОЛОГІЧНОГО ПІДХОДУ

Дмитро Крупа

Національний університет "Львівська політехніка",
кафедра інформаційних систем та мереж, Львів, Україна
E-mail: dmytro.v.krupa@lpnu.ua, ORCID: 0009-0002-6087-9988

У статті проаналізовано використання контрактного тестування для перевірки сумісності двох компонент, а саме вебсерверів, що використовують прикладний програмний інтерфейс (API) для передавання даних.

Стаття містить також порівняння API та контрактних тестів і описує випадки, коли останні мають перевагу. Описано структуру контракту для контрактного тесту.

Наведено опис онтологічного підходу для порівняння знань про бізнес системи-постачальника, у вигляді онтологій, із програмним поданням системи-споживача, що зберігається у вигляді класів. Розроблено структуру об'єкта, що доповнює контракт, надаючи йому перевірки на рівні бізнес-логіки. Запропоновані базові предикати, що становлять основу цього методу, подано їхній опис. Використання поля mapping уможливлює використання тезаурусів чи словників для автоматизованої заміни понять за допомогою слів-синонімів.

Наведено приклад, що демонструє роботу цього підходу, а саме: спрощене представлення концептів системи-постачальника, спрощена структура класів системи-споживача, приклад API та контракту для нього, розширення із використанням розробленого підходу та результат виконання тестів.

Ключові слова: контрактне тестування; онтологічний підхід; автоматизоване тестування; концепт.

Постановка проблеми

Складні системи, що інтегруються із великою кількість сторонніх систем, можна поділити на ті, що постачають послуги/дані, і на ті, що їх споживають. Наприклад, CRM та ERP системи охоплюють всі процеси роботи компанії та містять багато модулів. Кожний з цих модулів може інтегруватися з однією або багатьма сторонніми системами (наприклад, відображення карти з позначками клієнтів, системи оплати тощо). Відповідно, розроблювана система є "споживачем", а системи, з якими вона інтегрується, – "постачальниками". Це узалежнює систему від роботи "постачальників" і змушує негайно реагувати на зміни API, які відбуваються на боці постачальників.

Під час дослідження, яке здійснила компанія Postman, Inc у 2023 р. (Postman 2023 State of the API Report) [16], 52 % респондентів відповіли, що найбільшим викликом у користуванні API була недостатня кількість документації. Іншими вагомими проблемами були складнощі під час дослідження API (32 %) та нестача часу (27 %). Це означає, що розробники систем-споживачів не завжди можуть отримати необхідну документацію та відомості про особливості роботи з сервісом, з яким вони інтегруються.

Піраміда тестування допомагає розробникам та тестувальникам забезпечувати високу якість розроблюваного продукту. На рис. 1 зображено класичну піраміду автоматизації тестування. Вона відображає охоплення тестами, ціну підтримки та час виконання [3] :

- що ширший рівень, то більше тестів необхідно розробити;
- “вищі” тести складніше підтримувати;
- “вищі” тести виконуються довше.

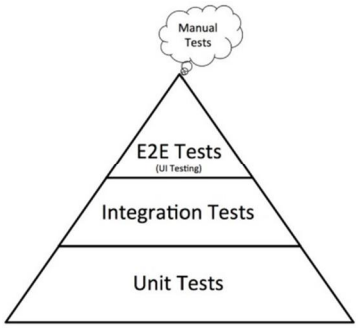

Рис. 1. Піраміда автоматизації тестування

Для перевірки правильності інтеграції із постачальником використовують низку підходів: інтеграційне тестування, API тестування та контрактне тестування.

Усі ці види тестування полягають у перевірці чітко заданих значень, без перевірки самого контексту, в якому використовують об’єкти у запитах між серверами. Якщо розробники системи-постачальника змінюють структуру в бізнес-логіці своєї програми, це може спричинити баги не одразу після наступного релізу, оскільки вони все ж також перевіряють зворотну сумісність та міграції на нову версію, а спричинить їх у майбутньому, можливо, через кілька релізів [4].

Це змусить усіх розробників систем-споживачів терміново вносити виправлення у свою систему. Тобто виникає ситуація, коли, не знаючи про зміну, ми не можемо зробити реакцію проактивною, а змушені реагувати реактивно.

Аналіз останніх досліджень та публікацій

Оскільки підхід до тестування залежить від технологій проєкту, його функціональності та бюджету, розглянемо загальні підходи до тестування, які слугують відправною точкою для розроблення тестів для проєкту.

Етап тестування надзвичайно важливий у структурі SDLC. Його забезпечують команди розробників та тестувальників, щоб досягти якісного та правильного виконання усіх заявлених функцій продукту [1]. Основи тестування, базові визначення та процес власне тестування в описано у багатьох працях. Зазвичай для ознайомлення із основами використовують книгу Р. Савина [2].

Для перевірки сумісності вебсерверів та їх компонент розглянемо перевірку сумісності їх інтерфейсів (API): інтеграційне, API та контрактне тестування.

Інтеграційне тестування допомагає перевірити взаємодію між різними модулями або компонентами системи, щоб визначити, чи вони коректно працюють разом. Цей вид тестування ретельно проаналізовано, виділено основні підходи не лише до створення тестів, але і до їх пріоритезації [5]. Натомість API тестування зосереджене на перевірці API для виявлення помилок та перевірки надійності, безпеки і відповідності очікуваній функціональності [7]. Одним із підвидів API тестів є контрактне тестування – перевірка взаємодії між різними модулями, а саме: чи система-постачальник і система-споживач дотримуються заздалегідь визначеного “контракту”. Контрактні тести стежать, щоб зміни в одному модулі не порушили взаємодію із іншим модулем [6]. Спільними перевагами усіх вищенаведених тестів є можливість рано виявити помилки, виконавши тести. Недолік у них теж спільний – витрата часу на розроблення та підтримку цих тестів.

Згідно із опитуванням Postman, Inc у 2023 р. (Postman 2023 State of the API Report) [16], лише 16 % респондентів відповіли, що використовують контрактне тестування у своїй роботі. Хоча інтеграційне тестування набрало 64 % відповідей і теж може перевірити правильність функціонування API, однак воно ресурсозатратніше порівняно із контрактним тестуванням, а також перевіряє безпеку, кешування, швидкодію та інші нефункціональні вимоги до API, які не є необхідними для правильного функціонування системи-споживача.

Тестування із використанням онтологій не є новим підходом, його досліджували і розвивали впродовж останнього десятиліття. Виконавши систематичний огляд літератури (SLR), після початкової вибірки із 396 досліджень ми виявили 12 онтологій, які належать до домену тестування програмного забезпечення, проте більшість з них мають недоліки (обмежене охоплення, недостатня оцінка тощо) [10].

Окремо розглянуто техніку онтологічного моделювання для IoT проєктів, що продемонструвала потенціал у використанні атрибутів домену в плануванні процесу тестування. Але цю техніку розробили із урахуванням особливостей IoT проєктів, а онтологічний метод не перевіряли на надійність у тестуванні IoT систем [11]. Також розроблено моделі з використанням онтологій для генерування API для мікросервісної архітектури, але вони були обмежені GET та, частково, POST запитами, що дає змогу лише отримувати та зберігати “сирі” дані [12]. Окрім того, розглянуто фреймворк для генерування тестів із використанням онтологій, проте він потребує доопрацювання перед використанням на практиці, а саме: необхідно розробити високорівневу мову для описання тестів, власне онтології для програмних продуктів та визначення критеріїв, які є важливими для предметної області проєктів [13].

Проблему забезпечення якості у системах із незалежною зміною компонентів розглянуто та проаналізовано із використанням онтологій або мереж Петрі, лише із застосуванням класичних підходів до тестування без описання перевірок кожної компоненти та визначення чітких і формальних правил, яким компоненти повинні відповідати [8, 9].

Формулювання цілі статті

Мета статті – проаналізувати можливість покращення інтеграційного процесу за допомогою контрактного тестування, яким керує система-споживач. Передбачено використати онтологічний підхід для перевірки сумісності вебсерверів, порівнявши концепти онтології системи-постачальника із класами системи-споживача, що репрезентують бізнес-логіку.

Крім того, підціль статті – сформувати простий та дієвий підхід до порівняння об’єктів та концептів, що стане основою інтерфейсу системи порівняння.

Виклад основного матеріалу

У статті розробимо загальний підхід до використання онтологій для забезпечення якості роботи програмного забезпечення. Звернемо увагу на API та контактне тестування, розширимо поняття контракту та використаємо онтологічний підхід для перевірки роботи програм.

API-тестування сконцентроване на даних, безпеці та бізнес-логіці систем. Розробники мають змогу перевіряти такі нефункціональні вимоги, як: час опрацювання запиту, HTTP статус-коди, якість даних та коди помилок. Крім того, саме API тести мають змогу перевірити back-end валідації [14].

Недолік API тестів – порівняно велика тривалість виконання кожного тесту, адже необхідно надіслати запит на сервер та очікувати на відповідь на запит, що може зайняти певний час. API тести доповнюють тести на основі контрактів, а саме перевіряють: інтерфейс, передумови, формат даних для кожного поля у відповіді, постумови, коди помилок та, власне, помилки, межі дозволених значень [15].

Для використання онтологічного підходу в перевірці контрактів між системами необхідно розширити контракт, додавши до нього предикати або інші можливі перевірки сумісності структури системи-споживача із онтологією системи-постачальника. Формат є довільним і повністю залежить від реалізації, прийнятої розробниками.

Розгляньмо приклад контракту та його можливе розширення для використання онтологічного підходу. Розроблюваний підхід передбачає створення окремого кроку в специфікації контракту – додавання масиву предикатів, що повинні виконуватися. Фактично, розроблюваний метод – перевірка контракту між онтологією постачальника та структурою класів споживача. Предикати можна зберігати у такій формі для зручного використання їх у перевірках контракту:

[{
“subject”:”…”,
“predicate”:” …”,
“object”:” …”
}]

Тобто розробники повинні створити масив предикатів такого типу, де subject – клас системи-споживача, а object – концепт (або індивід) системи-постачальника. Predicate визначає тип зв’язку між цими поняттями. Після цього, використовуючи перевірки, розробники можуть розробити тести, які обчислюватимуть значення кожного з предикатів true або false, а відтак зможуть приймати рішення, чи потрібно вносити зміни у систему, чи ні. Зазначмо, що деяких системах поріг, який змушує змінювати систему, може становити один провалений тест, а в інших помилки можуть бути допустимими задля забезпечення швидкого розроблення продуктів.

Визначати, які предикати можна вказувати та реалізовувати перевірки на їх основі, повинні розробники системи-споживача. Наведемо можливі правила та їх опис:

- Is – перевіряє, чи subject містить усі обов’язкові поля object-у;
- Not – перевіряє, чи subject не містить жодного обов’язкового поля object-у;
- Has – перевіряє, чи subject – поле класу, яка є об’єктом з усіма обов’язковими полями object.

Зауважмо, що під час реалізації необхідно додати список обов’язкових полів object-у та відповідність полів між ними, якщо їхні назви не збігаються. У результаті отримаємо наступний об’єкт, для якого властивості “requiredFields” і “mapping” є необов’язковими.

[{
“subject”:”…”,
“predicate”:” …”,
“object”:” …”,
“requiredFields”:” …”,
“requiredFields”:” …”
}]

Припустімо, є система-постачальник, що відображає діяльність компанії, яка здає автомобілі в оренду. В оренду можна взяти легковий автомобіль або вантажівку. Для цього необхідно мати відповідну категорію прав та вибрати потрібну місткість автомобіля. Ідея діяльності компанії полягає у тому, що місткість звичайного легкового автомобіля – 0, пікапа – 1, а вантажівки – 6–18 умовних одиниць вантажу. Знання про цю область відображено на рис. 2 у вигляді онтології.

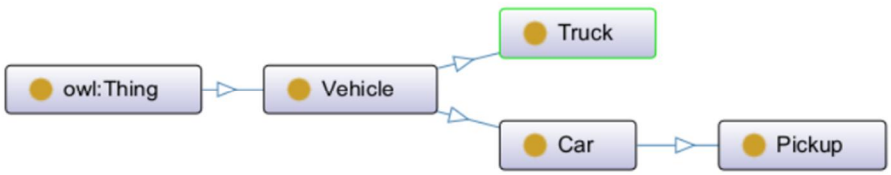

Рис. 2. Концепти системи-постачальника до змін

Окрім цього, створені дві data property: box:int та license:string. Ці властивості належать концепту Vehicle і всім його нащадкам. Box відображає, скільки умовних коробок можна помістити у транспорт, а license відповідає за необхідну категорію у посвідченні водія.

Система-споживач – це сервіс переїзду в інший будинок, одна з функцій якого – вибрати машину власне для переїзду. Ця система використовує постачальника для надання інформації про автомобілі, які можна використати для переїзду. Системи працюють як сервери та обмінюються даними через API. Спрощену структуру класів системи-споживача відображено на рис. 3.

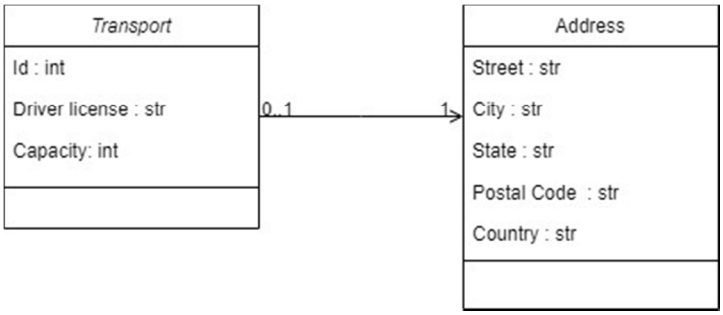

Рис. 3. Спрощена діаграма класів системи-споживача

Контракт запиту на отримання усіх транспортних засобів до системи-постачальника такий [15]:

1. Інтерфейс ендпоінту : /api/vehicle?type=TRUCK, PICKUP.
2. Передумова: type – обов'язковий параметр.
3. Якщо передумови не виконані – відповідь міститиме код помилки 400.
4. Формат відповіді із сервера: масив об'єктів, що містять дані про транспорт: модель, номер реєстрації, тип ліцензії водія, місткість.

Розширимо контракт, додавши до нього логічне (бізнес) значення, яке подамо у вигляді:

1. Transport is Truck.
2. Transport is Pickup.

Тут наведено два предикати, значення яких розробники очікують як true. Ці два твердження означають, що Transport повинен містити усі дані та відповідати водночас класам/концептам Truck та Pickup. Розробники системи-постачальника повинні надати онтологію, що описує їхню систему через чітко визначений ендпоінт для того, щоб розробник міг перевірити істинність цих тверджень автоматизовано або вручну.

Припустімо що бізнес-модель компанії, яка надає послуги з оренди автомобіля, змінилася і вони тепер надають в оренду не саму вантажівку, а вантажівку із водієм. У їхньому розумінні – Vehicle – це транспорт, який може орендувати користувач, а рис. 4 зображена змінена онтологія. Також змін зазнав і сам концепт Truck – як data property він має лише box.

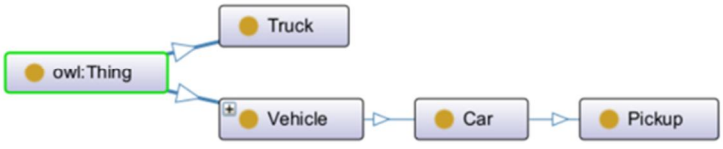

Рис. 4. Концепти системи-постачальника після змін

Перевірка, чи контракт не порушує контекст, у якому працює програма, передбачає усі стандартні перевірки, перераховані вище, а також перевірку предикатів, які визначили розробники. Задані предикати можна переписати у запропонованому вище вигляді:

```
[
   {
      "subject" : "Transport",
      "predicate" : "is",
      "object" : " Truck",
```

```
        “mapping” : [{“capacity”, “box”}]
    },
    {
        “subject” : ” Transport”,
        “predicate” : ”is”,
        “object” : ” Pickup”,
        “mapping” : [{“capacity”, “box”}]
    }
]
```

Розглянемо перевірку з прикладу:

1. **Transport is Truck**: Transport має поля “license” і “capacity”, Truck – лише “box”. Згідно із полем “mapping” зробимо висновок, що “capacity” і є “box”. Але концепт Truck не містить license чи його відповідника. Тому результат такої перевірки – **false**.
2. **Transport is Pickup**: Transport має поля “license” і “capacity”, Pickup – лише “license” і “capacity”. Згідно із полем “mapping” зробимо висновок, що “capacity” і є “box”. Оскільки Transport містить усі поля Pickup, зробимо висновок, що результат такої перевірки – **true**.

Отже, перший предикат не працює (має значення false) і в результаті тестування є один failed-тест. Хоча розробники або DevOps-інженери можуть встановити певне порогове значення, яке означатиме, чи тести пройшли успішно (наприклад, 80 % успішності тестів), рекомендовано переглядати правильність роботи програми та її сумісність із системою-постачальником, якщо є хоча б один неуспішний тест.

Виконаємо експеримент для оцінювання ефективності запропонованого підходу. Для порівняння використаємо інтеграційні тести, які опосередковано перевіряють об’єкти (запити/відповіді) між двома системами. Розрахуємо очікуваний час на виконання одного інтеграційного тесту, не враховуючи часу, що необхідний для створення тестового середовища (ініціалізація тестової бази даних, вебсервера тощо). Час приблизний, адже опрацювання даних та їх передавання варіюється залежно від географічного розміщення серверів, кількості виділених ресурсів та складності логіки, що повинна виконуватися. Оскільки очікуваний час завантаження сторінки вебсайту – 2 секунди (зазвичай варіюється від 1 до 3 секунд і вказується у нефункціональних вимогах до проєкту), то опрацювання даних на сервері триватиме близько 1.8 секунди. Інтеграційний тест складається із підготовки даних та запуску методу (функції), що їх опрацьовує та відповідає API запиту.

Підготовка даних зазвичай не потребує багато часу, тому візьмемо як середній очікуваний час виконання одного інтеграційного тесту 2 секунди. Окремо зауважимо, що для повної перевірки одного об’єкта може знадобитися не один тест, а декілька.

Порівняємо із тестом № 1 (Transport is Truck) із вищенаведеного прикладу. Вилучимо час отримання і зчитування онтології та виконаємо код, що перевірить сумісність об’єктів. Для цього прикладу використані мова програмування Java та бібліотека OWL API. Результат експерименту зображений на рис. 5.

```
Property license doesn't exist

Property capacity exists
Range is the same

Time: 4 ms
```

Рис. 5. Результат опрацювання тесту № 1

Проаналізувавши результати експерименту, можна зробити висновок, що цей підхід є простим способом перевірки відповідності між об’єктами двох систем і швидший приблизно у 500 разів та не потребує затрат на розроблення окремих тестів, а лише на розроблення та підтримку правил перевірки з

боку розробників системи-споживача. Варто зазначити, що експеримент виконано на локальній машині із підготованими даними. Якщо розробити зручний для розробників спосіб визначення полів класу для перевірок, то тесту знадобиться додатковий час на аналіз та відбір полів.

Висновки

У статті запропоноване розширення до класичного контрактного тестування. Сформульовано основний підхід, що ґрунтується на поєднанні онтологій та об'єктно орієнтованого програмування з використанням предикатів для створення чіткої перевірки. Запропоновано основні предикати та структуру об'єкта, що містить інформацію про перевірки.

Наведено приклад і продемонстровано, як саме система має перевіряти дані сумісності із системою-постачальником. Виконано також експеримент на основі поданого прикладу для демонстрації швидкодії розроблюваного підходу.

Онтологічний підхід, розглянутий у цій статті, дозволяє створювати тести, що легко підтримуються, та забезпечує покращення оцінки сумісності компонент між собою з погляду експертів, що розробляють онтологію системи-постачальника.

Список літератури

1. Hossain, M. A. (2023). Software Development Life Cycle (SDLC) Methodologies for Information Systems Project management. *International Journal for Multidisciplinary Research*, 5(5). https://doi.org/10.36948/ijfmr.2023.v05i05.6223

2. Savin, R. (2007). *Тестування DOT COM або Допомога по жорстокому поводженню з багами в інтернет-стартапах*. Print2print

3. Bose, S. (2022). Getting Started with the Test Automation Pyramid – An Ultimate Guide. BrowserSack. https://www.browserstack.com/guide/testing-pyramid-for-test-automation

4. Zarevych, O. (2020). *Як за допомогою тестів пришвидшити реліз*. DOU. https://dou.ua/lenta/articles/how-testing-speed-up-release/

5. Akinsola, J. E. T., Adeagbo, M., Abdul-Yakeen, S. O., & Yusuf, A. (2022). Qualitative comparative analysis of software integration testing techniques. *Journal of Science and Logics in ICT Research* .

https://www.researchgate.net/publication/359857331_Qualitative_Comparative_Analysis_of_Software_Integration_Testing_Techniques

6. Cohen, N. (2023). *Contract Testing: An Introduction and guide*. BlazeMeter. https://www.blazemeter.com/blog/contract-testing

7. (n.d.) *API testing*. Postman. https://www.postman.com/api-platform/api-testing/.

8. Liang, Q., & Huhns, M. N. (2008). Ontology-Based Compatibility Checking for Web Service Configuration Management. 15. https://doi.org/10.1007/978-3-540-89652-4_31

9. Craig, D. C., & Zuberek, W. (2007). Verification of Component Behavioral Compatibility. 2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX '07). https://doi.org/10.1109/DEPCOS-RELCOMEX.2007.53

10. De Souza, E. F., De Almeida Falbo, R., & Vijaykumar, N. L. (2013). Ontologies in software testing: A Systematic Literature Review. ResearchGate. https://www.researchgate.net/publication/282915285_Ontologies_in_software_testing_A_Systematic_Literature_Review

11. Naqvi, M. R., Iqbal, M. W., Ashraf, M., & Ahmad, S. (2023). Ontology Driven Testing Strategies for IoT Applications. Computers, Materials & Continua. https://doi.org/10.32604/cmc.2022.019188

12. Krouwel, M., & Op 't Land, M. (2022). Business Driven Microservice Design – An Enterprise Ontology Based Approach to API Specifications (pp. 95–113). Springer, Cham. https://doi.org/10.1007/978-3-031-11520-2_7

13. Nasser, V., Du, W., & Macisaac, D. (2010). An Ontology-based Software Test Generation Framework. Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering.

https://www.researchgate.net/publication/221390387_An_Ontology-based_Software_Test_Generation_Framework

14. Gillis, A. S. (2023). *API testing*. TechTarget. https://www.techtarget.com/searchapparchitecture/definition/API-testing

15. Braakman, W. (2023, March 30). Introduction to Contract Testing. Medium. https://www.techtarget.com/searchapparchitecture/definition/API-testing

16. (n.d.). 2023 State of the API Report. *Postman*. https://www.postman.com/state-of-api/api-global-growth/#api-global-growth

References

1. Hossain, M. A. (2023). Software Development Life Cycle (SDLC) Methodologies for Information Systems Project management. *International Journal for Multidisciplinary Research*, 5(5). https://doi.org/10.36948/ijfmr.2023.v05i05.6223

2. Savin, R. (2007). *Testing DOT COM or Assistance with harsh treatment of bugs in internet startups*. Print2print

3. Bose, S. (2022). *Getting Started with the Test Automation Pyramid – An Ultimate Guide*. BrowserSack. https://www.browserstack.com/guide/testing-pyramid-for-test-automation

4. Zarevych, O. (2020). *How to speed up the release with the help of tests*. DOU. https://dou.ua/lenta/articles/how-testing-speed-up-release/

5. Akinsola, J. E. T., Adeagbo, M., Abdul-Yakeen, S. O., & Yusuf, A. (2022). Qualitative comparative analysis of software integration testing techniques. *Journal of Science and Logics in ICT*

https://www.researchgate.net/publication/359857331_Qualitative_Comparative_Analysis_of_Software_Integration_Testing_Techniques

6. Cohen, N. (2023). *Contract Testing: An Introduction and guide*. BlazeMeter. https://www.blazemeter.com/blog/contract-testing

7. (n. d.) *API testing*. Postman. https://www.postman.com/api-platform/api-testing/

8. Liang, Q., & Huhns, M. N. (2008). *Ontology-Based Compatibility Checking for Web Service Configuration Management*. 15. https://doi.org/10.1007/978-3-540-89652-4_31

9. Craig, D. C., & Zuberek, W. (2007). Verification of Component Behavioral Compatibility. *2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX '07)*. https://doi.org/10.1109/DEPCOS-RELCOMEX.2007.53

10. De Souza, E. F., De Almeida Falbo, R., & Vijaykumar, N. L. (2013). *Ontologies in software testing: A Systematic Literature Review*. ResearchGate. https://www.researchgate.net/publication/282915285_Ontologies_in_software_testing_A_Systematic_Literature_Review

11. Naqvi, M. R., Iqbal, M. W., Ashraf, M., & Ahmad, S. (2023). Ontology Driven Testing Strategies for IoT Applications. *Computers, Materials & Continua*. https://doi.org/10.32604/cmc.2022.019188

12. Krouwel, M., & Op ‘t Land, M. (2022). *Business Driven Microservice Design – An Enterprise Ontology Based Approach to API Specifications* (pp. 95–113). Springer, Cham. https://doi.org/10.1007/978-3-031-11520-2_7

13. Nasser, V., Du, W., & Macisaac, D. (2010). An Ontology-based Software Test Generation Framework. *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering*.

https://www.researchgate.net/publication/221390387_An_Ontology-based_Software_Test_Generation_Framework

14. Gillis, A. S. (2023). *API testing*. TechTarget. https://www.techtarget.com/searchapparchitecture/definition/API-testing

15. Braakman, W. (2023). *Introduction to Contract Testing*. Medium. https://www.techtarget.com/searchapparchitecture/definition/API-testing

16. (n. d.). *2023 State of the API Report. Postman*. https://www.postman.com/state-of-api/api-global-growth/#api-global-growth

CONTRACT-BASED TESTING USING THE ONTOLOGY APPROACH

Dmytro Krupa

Lviv Polytechnic National University,
Department of Information Systems and Networks, Lviv, Ukraine
E-mail: dmytro.v.krupa@lpnu.ua, ORCID: 0009-0002-6087-9988

This article analyzes the use of contract testing to verify the compatibility of two components, specifically web servers that use an Application Programming Interface (API) for data transmission.

Additionally, the article includes a comparison of APIs and contract tests and describes cases where the latter have an advantage. A contract structure for contract testing is described.

The article contains a description of the ontological approach for comparing knowledge about the business systems of the provider, in the form of ontologies, with the software representation of the consumer system, stored in the form of classes. A structure of the object that supplements the contract by providing it with business logic level checks is developed. Basic predicates that form the basis of this method are proposed and described. The use of the mapping field allows for the use of thesauruses or dictionaries for automated replacement of concepts when using synonym words.

An example is provided that demonstrates the operation of this approach, namely: a simplified representation of the provider system's concepts, a simplified structure of the consumer system's classes, an example of an API and a contract for it, an extension using the developed approach, and the result of test execution.

Key words: contract testing; ontological approach; automated testing; concept.