НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова
праця на правах рукопису

БАЗАКА ЮРІЙ АНАТОЛІЙОВИЧ

УДК 004.05

ДИСЕРТАЦІЯ

МОДЕЛІ, МЕТОДИ ТА ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ТЕСТУВАННЯ РОЗПОДІЛЕНИХ СИСТЕМ

05.13.06 – Інформаційні технології

05 Технічні науки

Подається на здобуття наукового ступеня кандидата технічних наук

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело
_____ Ю.А. Базака

Науковий керівник
Корнага Ярослав Ігорович
доктор технічних наук, доцент

Ідентичність всіх примірників дисертації
ЗАСВІДЧУЮ:
вчений секретар спеціалізованої вченої ради К 58.082.02
Комар М.П. _____

Тернопіль – 2021

АНОТАЦІЯ

Базака Ю.А. Моделі, методи та інформаційна технологія підвищення ефективності тестування розподілених систем. – Рукопис.

Дисертація на здобуття наукового ступеня кандидата технічних наук за спеціальністю 05.13.06 «Інформаційні технології». – Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Західноукраїнський національний університет, Тернопіль, 2021.

Дисертація присвячена вирішенню актуальної задачі розробки моделей і методів тестування компонентів програмного забезпечення шляхом створення інформаційної технології, що дозволяє підвищити ефективність функціонування розподілених систем обробки інформації.

У першому розділі дисертації проведено аналіз механізмів тестування програмного забезпечення систем. Показано, що при тестуванні великих систем, які містять клієнтські частини та серверні, важливо перевіряти не тільки роботу системи при невеликому числі користувачів, а і при максимальних навантаженнях, які можуть привести до непроходження тестів та збоїв у роботі системи. Важливою характеристикою також стає час реагування системи на запити, які до неї відправляються.

Розглянуто піраміду тестування Майка Кона. Згідно механізмів тестування піраміди, варто писати багато тестів з невеликим об’ємом та швидких юніт-тестів. Для написання загальних тестів потрібно використовувати невелику кількість високорівневих наскрізних тестів, які перевіряють програмне забезпечення від початку до кінця. При цьому потрібно слідкувати за тим, щоб в результаті механізми використання піраміди не привели до великих затрат часу.

Розглянуті механізми застосування наскрізних тестів. Показано, що тестувальники, як правило, вибирають Selenium і протокол WebDriver, причому Selenium може застосовуватися для різних версій та типів веб-браузерів. З результатів порівняння способів тестування визначено, що основні потреби

тестування покриваються спеціалізованими продуктами MsBuild і бібліотеками сімейства XUnit. Однак навіть успішне виконання тестових сценаріїв на етапі розробки розподіленої системи не дає достатніх підстав для того, щоб зробити висновок про працездатність системи в реальному середовищі. Потрібно інтегрувати систему підтримки автоматизованого тестування в розроблювану розподілену систему, що дозволить проводити тестування на етапі розгортання та на етапі промислової експлуатації системи.

У другому розділі виконано модифікацію піраміди тестування Майка Кона для адаптації при тестуванні в розподілених системах обробки інформації, що дозволило розширити можливості тестування і застосувати особливості розподілених систем. Розроблено рекомендації щодо подальшого застосування механізмів модифікованої піраміди Майка Кона.

Визначено вимоги для забезпечення безперебійної роботи всіх сервісів розподіленої комп'ютерної системи, в якій розміщується розподілена система обробки інформації, та забезпечення безперервної передачі пакетів запитів із заданою швидкістю.

Розроблена модель тестування програмного забезпечення розподілених систем на основі автономного розгортання програмних компонентів, яка базується на застосуванні механізмів на основі модифікованої піраміди Майка Кона, що дозволяє підвищити достовірність тестування та зменшити кількість об'ємних тестів.

Виконано порівняння схем з різною кількістю сервісів за показником ефективності тестування, в якості якого прийнята апостеріорна імовірність справного стану за умови отримання позитивного результату тестування. Показано, що в системі з роздільним тестуванням одного сервісу апостеріорна імовірність дає найвищий показник, а починаючи з системи тестування двох та більше сервісів, в якій реалізовано ланцюгове тестування, апостеріорна ймовірність зменшується зі збільшенням кількості вузлів.

У третьому розділі показано, що при одноразовому тестуванні розподіленої системи з невеликою кількості вузлів час тестування при

класичному варіанті буде менший в зв’язку з тим, що сума часу розгортання та згортання кожного сервісу буде більшою за розгортання всієї системи. Але, якщо потрібно кілька разів тестувати всю систему, то тестування за допомогою контрактів є більш вигідним, тому що проводяться зміни тільки в кількох сервісах і другий раз уже не потрібно розгортати всю систему, а тільки окремі сервіси.

Удосконалено метод тестування програмного забезпечення вузлів розподіленої системи, який відрізняється від існуючих застосуванням контрактних тестів та аналізом прогнозованого результату поведінки сервісів, що дозволяє зменшити час розгортання компонентів тестового середовища та час проходження тестів.

Показано, що в порівнянні з наскрізним тестуванням інтерфейсів користувача переваги застосування механізмів симуляторів тестування інтерфейсів користувача дозволяють зменшити час, затрачений на тестування будь-якого сервісу інтерфейсу користувача в порівнянні з наскрізним тестування інтерфейсу користувача. Час зменшується за рахунок зменшення кількості одночасних сервісів інтерфейсу користувача.

Удосконалено метод тестування інтерфейсу користувача програмного забезпечення вузлів розподіленої системи, який відрізняється від існуючих механізмом симуляції його роботи, що дозволяє проводити тестування окремих компонентів інтерфейсу системи.

У четвертому розділі описано розроблену за інформаційною технологією інформаційну систему, яка реалізує представлені в роботі моделі та методи і була використана для реалізації експериментальних досліджень тестування сервісів розподілених систем обробки інформації за допомогою механізмів на основі використання контрактів. Під час експерименту проводилось два типи тестування розподіленої системи: наскрізне та контрактне. Для наскрізного тестування використовувалось програмне забезпечення Postman. У Postman потрібно було зазначити всі тестові сценарії запитів, які будуть відсилатися на розподілену систему в рамках наскрізного тестування.

Для реалізації сценаріїв у програмному коді вибрано бібліотеку RestSharp, яка забезпечувала здійснення синхронних та асинхронних запитів на сервіси через HTTP протокол. Запуск тестів відбувається за допомогою бібліотеки XUnit.

Показано, що при невеликій кількості вузлів наскрізне тестування сервісів проходить швидше за контрактне тестування цих же сервісів. При збільшення кількості вузлів час, який необхідний на тестування сервісів розподіленої системи контрактими тестами, стає менший за час, який необхідний для тестування цієї ж системи наскрізним методом. Це пов’язано з тим, що при виникненні негативного результату тесту при наскрізному тестуванні потрібно заново встановлювати всю системи, а при контрактному тестуванні — тільки сервіс, для якого був негативний результат тестування.

Показано, що при невеликій кількості вузлів наскрізне тестування інтерфейсів користувача проходить швидше за симуляційне тестування цих же інтерфейсів користувача. При збільшення кількості вузлів час, який необхідний на тестування сервісів розподіленої системи симуляційними тестами, стає менший за час, який необхідний для тестування цієї ж системи наскрізним методом.

Основні результати роботи використано на кафедрі технічної кібернетики Національного технічного університету України «Київський політехнічний інститут» при викладанні дисциплін навчального плану підготовки бакалаврів за спеціальностями 121 «Інженерія програмного забезпечення».

Ключові слова: тестування програмного забезпечення, розподілена система обробки інформації, інформаційна система.

ANNOTATION

Yu. Bazaka Models, methods and information technology to increase the efficiency of testing distributed systems. – Manuscript.

Dissertation for the degree of candidate of technical sciences in specialty 05.13.06 «Information Technologies». – National Technical University of Ukraine «Kyiv Polytechnic Institute named after Igor Sikorsky», West Ukrainian National University, Ternopil, 2021.

The dissertation is devoted to the decision of an actual problem of development of models and methods of testing of components of the software by creation of information technology that allows to increase efficiency of functioning of the distributed systems of information processing.

Section 1 describes analysis of mechanisms of testing of the software of systems is carried out. It is shown that when testing large systems that contain client parts and servers, it is important to check not only the system with a small number of users, but also at maximum loads, which can lead to failure tests and system failures. An important characteristic is also the response time of the system to requests sent to it.

Mike Kon's testing pyramid is considered. According to the pyramid testing mechanisms, it is worth writing a lot of small-volume tests and quick unit tests. To write general tests, you need to use a small number of high-level end-to-end tests that test the software from start to finish. At the same time it is necessary to watch that as a result mechanisms of use of a pyramid did not lead to big expenses of time.

Mechanisms of application of through tests are considered. It has been shown that testers typically choose Selenium and WebDriver, and Selenium can be used for different versions and types of web browsers. From the results of the comparison of testing methods, it was determined that the main testing needs are covered by specialized MsBuild products and libraries of the XUnit family. However, even the successful execution of test scenarios at the stage of development of a distributed system does not provide sufficient grounds to conclude that the system works in a real

environment. It is necessary to integrate the automated testing support system into the developed distributed system, which will allow testing at the stage of deployment and at the stage of industrial operation of the system.

Section 2 describes modification of the testing pyramid of Mike Kon was performed for adaptation during testing in distributed information processing systems, which allowed to expand the possibilities of testing and apply the features of distributed systems. Recommendations for further use of the mechanisms of the modified Mike Kon pyramid have been developed.

The requirements for ensuring the uninterrupted operation of all services of the distributed computer system in which the distributed information processing system is located and ensuring the continuous transmission of request packets at a given speed are defined.

A model for testing software for distributed systems based on the autonomous deployment of software components has been developed, which is based on the use of mechanisms based on a modified Mike Conn pyramid, which allows to increase the reliability of testing and reduce the number of bulk tests.

The comparison of schemes with different number of services in terms of testing efficiency, which is taken as the a posteriori probability of good condition, provided a positive test result. It is shown that systems with separate testing of one service a posteriori probability gives the highest indicator, and starting with the system of testing two or more services in which chain testing is implemented, a posteriori probability decreases with increasing number of nodes.

Section 3 describes shows that with a single test of a distributed system with a small number of nodes, the test time in the classic version will be shorter due to the fact that the sum of the deployment and collapse time of each service will be greater than the deployment of the entire system. But if you need to test the whole system several times, then testing with contracts is more profitable, because changes are made only in a few services and the second time you no longer need to deploy the entire system, but only individual services.

The method of software testing of distributed system nodes has been improved, which differs from the existing ones by the use of contract tests and analysis of the predicted result of service behavior, which allows to reduce the deployment time of test environment components and test time.

It is shown that in comparison with end-to-end testing of user interfaces, the advantages of using the mechanisms of user interface test simulators allow to reduce the time spent on testing any UI service in comparison with end-to-end testing of user interfaces. The time is reduced by reducing the number of simultaneous user interface services.

The method of testing the user interface of the software of the nodes of the distributed system has been improved, which differs from the existing ones by the mechanism of simulation of its operation, which allows testing the individual components of the system interface.

Section 4 describes the information system developed by information technology, which implements the models and methods presented in the work and was used to implement experimental studies of testing services of distributed information processing systems using mechanisms based on the use of contracts. During the experiment, two types of testing of the distributed system were performed: through and contract. Postman software was used for end-to-end testing. In Postman, it was necessary to specify all test scenarios of queries that will be sent to the distributed system as part of end-to-end testing.

To implement the scripts in the program code, the RestSharp library was selected, which provided synchronous and asynchronous requests for services via the HTTP protocol. Tests are run using the XUnit library.

It is shown that with a small number of nodes, end-to-end testing of services is faster than contract testing of the same services. As the number of nodes increases, the time required to test the services of a distributed system by contract tests becomes less than the time required to test the same system through the method. This is due to the fact that if there is a negative test result in end-to-end testing, you need to reinstall the

entire system, and in contract testing only the service for which there was a negative test result.

It is shown that with a small number of nodes, end-to-end testing of user interfaces is faster than simulation testing of the same user interfaces. As the number of nodes increases, the time required to test the services of a distributed system by simulation tests becomes less than the time required to test the same system through the method.

The main results of the work were used at the Department of Technical Cybernetics of the National Technical University of Ukraine "Kyiv Polytechnic Institute" in teaching the disciplines of the curriculum for bachelors in specialties 121 "Software Engineering".

Key words: software testing, distributed information processing system, information system.

Список опублікованих праць за темою дисертації

Праці, в яких опубліковані основні наукові результати дисертації:

1. Mukhin V., Kornaga Ya., Mostovyi Y., Bazaka Y. A Model For Events Monitoring Heterogeneous Distributed Databases Based on Vector-matrix Operations. The Far East Journal of Electronics and Communications, 2016. Vol. 16, Issue 3. P. 645 – 656. (This journal is indexed in Elsevier Scopus).

2. Zhenbing H., Mukhin V., Kornaga Ya., Herasymenko O., Bazaka Y. The scheduler for the grid system based on the parameters monitoring of the computer components. Eastern European Journal of Enterprise Technologies, 2017. № 1 (2-85). P. 31 – 39. (This journal is indexed in Elsevier Scopus).

3. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Метод автоматизації тестування розподіленої системи з використанням контрактів. Sciences of Europe, 2020. № 55. С. 45 – 49.

4. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Метод тестування інтерфейсу користувача з використанням імітаторів. The

scientific heritage, 2020. № 51. С. 54 – 57.

5. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Method of Protecting Software Code from Reengineering Using Virtual Machines. Sciences of Europe, 2020. № 58. С. 59 – 62.

6. Корнага Я.І., Базака Ю.А., Базалій М.Ю. Захист програмного забезпечення за допомогою заплутуючих перетворень. The scientific heritage, 2020. № 54. С. 72 – 75.

7. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Використання баз даних тамов програмування у високоточних обчисленнях чисел із плаваючою точкою великої розрядності. Вчені записки Таврійського національного університету імені В.І. Вернадського, 2020. Том 31(70), №5. С. 82 – 88.

Праці, що засвідчують апробацію матеріалів дисертації:

8. Mukhin V., Kornaga Ya., Yakovleva A., Herasymenko O., Bazaca Yu., Bazaliy M. The model for distributed systems testing based on the control services. XXIV Міжнародна конференція з Автоматичного Управління “АВТОМАТИКА – 2017”. 13 – 15 вересня 2017 р., м. Київ., 2017. С. 45 – 46.

9. Мухін В.Є., Корнага Я.І., Яковлєва А.П., Базалій М.М., Базака Ю.А. Модифікований метод обфускації програмного кода за допомогою вставки інструкцій. Міжнародна наукова конференція “Інтелектуальні системи прийняття рішень та проблеми обчислювального інтелекту” (ISDMCI’2018), 21 – 27 травня 2018 р., с. Залізний порт. – Херсон: Видавництво ПП Вишемирський, 2018.– С. 67 – 68.

10. Mukhin V., Kornaga Ya., Bazaka Yu., Bazaliy M., Yakovleva A. Modified Method of Software Testing for Distributed Computer System. 2018 IEEE First International Conference on System Analysis & Intelligent Computing (SAIC). 8 – 12 October 2018, Kiev, Ukraine, 2018. P. 1 – 4. (This conference is indexed in Elsevier Scopus).

11. Mukhin V.Ye., Kornaga Ya.I., Abdullayev S.H., Gerasymenko O.Yu.,

Bazaka Yu.A. Method of the reserve resources determination for the distributed computer systems with the network-centric resource control. 6th International Conference on Control and Optimization with Industrial Applications (COIA). 11 – 13 July 2018. Baku, Azerbaijan, 2018. P. 226 – 231. (This conference is indexed in Web of Science).

12. Mukhin V., Vyshnivskyi V., Kornaga Ya., Herasymenko O., Bazaka Yu., Bazaliy M. Study of the functioning of the distributed computer system with a resource control mechanism based on a network-centric approach. 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2019. 18 – 21 September 2019, Metz, France. – 2019. P. 100 – 105. (This conference is indexed in Elsevier Scopus, Web of Science).

13. Mukhin V., Kornaga Y., Tkach M., Herasymenko O., Bazaka Y., Mukhin O. Subtask Prioritization on Workflow Execution in Distributed Wireless Computer System with Network-Centric Approach to Resource Control. 5th IEEE International Symposium on Smart and Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems, IDAACS-SWS 2020, 17 September 2020, Dortmund, Germany. – 2020. (This conference is indexed in Elsevier Scopus, Web of Science).

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

РС – розподілена система

КТ – контрактне тестування

СТ – симуляційне тестування

НТ – наскрізне тестування

API – application programming interface

ПЗ – програмне забезпечення

ІС – інтерфейс сервісу

ЮТ – юніт - тести

БД – база даних

ВСТУП

Актуальність теми. Сучасний рівень організації та управління розподіленими інформаційними системами передбачає використання різних механізмів оброблення та зберігання даних. Одним із них є забезпечення користувачів розподілених систем можливістю необмеженого доступу до масивів даних за умови високої швидкості оброблення даних у системах з постійним масштабуванням обсягу оброблюваної інформації. Проведенням перевірки на працездатність таких систем займаються багато програмістів.

У наш час розподілені системи розвиваються надзвичайно швидко завдяки використанню мікросервісних архітектур та необмеженого доступу до ресурсів мережі інтернет. За останні три роки об’єми інформації у глобальних мережах збільшилися в кільканадцять разів та становлять понад 40 трильйонів гігабайт. Перші дослідження у сфері тестування розподілених систем обробки інформації розпочалися у кінці 1990-х рр., після того як глобальна мережа стала доступною більшості людей. Основним завданням досліджень стали можливості автоматизованого тестування та перевірка розподілених систем, що характеризуються обмеженням часу доступу та кількістю ресурсів. У свою чергу, це стало передумовою появи великої кількості можливостей застосування нових засобів та механізмів у практичному використанні.

На сьогодні тестування розподілених систем обробки інформації вважають одним із найбільш важливих завдань під час розроблення будь-якого програмного забезпечення, а з появою різних технологій побудови розподілених систем методи тестування дозволяють перевіряти великі масиви інформації за відповідних заданих параметрів на вузли системи. Ці параметри є важливими для впровадження автоматизованого тестування й досягнення мети підвищення швидкості та якості роботи програмних продуктів.

Питанням побудови механізмів тестування розподілених систем обробки інформації присвячено велику кількість наукових робіт українських та зарубіжних вчених В.Є. Мухіна, Ю.В. Кравченка, Г.А. Кучука, І.Ю. Субача, а

також F. Eliassen, A. Polini, J. Wildstrom, P. Stone, E. Witchel, R. Mooney, M. Dahlin та ін. Питання забезпечення надійності роботи розподілених систем досліджено у роботах О.Г. Додонова, Д.В. Ланде, Ю.В. Журавського, І.В. Рубана.

Незважаючи на значні успіхи у вирішенні проблем тестування розподілених систем обробки інформації, далеко не всі завдання у цій сфері можна вважати вирішеними. Актуальними дослідженнями є системи з динамічною структурою середовища зберігання даних в умовах впливу внутрішніх і зовнішніх дестабілізуючих факторів. Виходячи з цього, недосконаліть та обмеженість відомих наукових методів тестування розподілених систем обробки інформації, зокрема у бездротових мобільних мережах, не дозволяє забезпечити повноцінне виявлення несправностей та своєчасне їх усунення.

Таким чином, дослідження моделей, методів та інформаційних технологій підтримки ефективного тестування програмного забезпечення на сьогодні є актуальним науковим завданням.

Зв'язок роботи з науковими програмами, планами та темами. Тематика дисертаційної роботи й отримані результати безпосередньо відповідають пріоритетності розвитку інформаційних та комунікаційних технологій в Україні до 2020 р. згідно із Законом України «Про пріоритетні напрями розвитку науки і техніки» від 11.07.2001 р., № 2623-III, зі змінами, внесеними згідно із Законом України «Про наукову та науково-технічну діяльність» від 26.11.2015 р., № 848-VIII.

Дисертаційна робота виконана відповідно до планів наукової і науково-технічної діяльності Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» і є частиною досліджень у межах науково-дослідних робіт:

– «Антропоморфний роботизований транспортний засіб для розвантаження людини в умовах підвищеного ризику та невизначеності рельєфу місцевості» (Державний реєстраційний № 0117U001179, КПІ ім. Ігоря Сікорського, м. Київ),

яку виконував Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського» у 2017–2018 рр.;

– «Оптимізація роботи веб-орієнтованих систем з великим набором даних» (державний реєстраційний № 0117U004913, КПІ ім. Ігоря Сікорського, м. Київ), яку виконує кафедра технічної кібернетики Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» з 2018 р.

Особисто автором у першій науково-дослідній роботі запропоновано механізми тестування вузлів зв'язку з антропоморфним роботизованим транспортним засобом, у другій – методи автоматизованого тестування інформації великого обсягу на основі модифікації механізмів обміну даними.

Мета і задачі дослідження. Метою дисертаційної роботи є підвищення ефективності функціонування розподілених систем обробки інформації на основі розроблення моделей і методів тестування компонентів таких систем.

Для досягнення поставленої мети в роботі вирішуються такі **основні завдання**:

1. Аналіз проблеми використання наявних підходів до тестування в архітектурі розподілених систем.

2. Розроблення модифікованої «піраміди» тестування програмного забезпечення в архітектурі систем розподіленої обробки інформації.

3. Розроблення моделі тестування сервісів розподіленої системи обробки інформації.

4. Удосконалення методу тестування програмного забезпечення розподілених систем з використання імітаторів.

5. Удосконалення методу тестування інтерфейсів користувача розподіленої системи з використанням симуляційних тестів.

6. Розроблення інформаційної технології для підвищення ефективності тестування програмного забезпечення розподілених систем.

7. Розроблення спеціалізованого середовища для моделювання та порівняння результатів експериментальних досліджень запропонованих методів тестування розподілених систем.

Об’єкт дослідження – процеси тестування систем розподіленої обробки інформації.

Предмет дослідження – моделі, методи та інформаційна технологія для ефективного тестування розподіленого програмного забезпечення.

Методи дослідження. Дисертаційне дослідження ґрунтується на системному аналізі результатів сучасних теоретичних і прикладних розробок вітчизняних і зарубіжних вчених у сфері аналізу програмного коду. Під час виконання дослідження було використано логіко-імовірнісні та статистичні методи, методи теорії обробки спостережень для аналізу експериментальних даних, методи формалізації даних, методи аналітичного моделювання.

Наукова новизна отриманих результатів:

1. Уперше розроблено модель тестування програмного забезпечення розподілених систем на основі автономного розгортання програмних компонентів, яка ґрунтується на застосуванні механізмів на основі модифікованої піраміди Майка Кона, що дозволяє підвищити ефективність тестування та зменшити кількість об’ємних тестів.

2. Удосконалено метод тестування програмного забезпечення вузлів розподіленої системи, який різниться від наявних застосуванням контрактних тестів та аналізом прогнозованого результату поведінки сервісів, що дозволяє зменшити час розгортання компонентів тестового середовища та час проходження тестів.

3. Удосконалено метод тестування інтерфейсу користувача програмного забезпечення вузлів розподіленої системи, який різниться від наявних підходом до симуляції його роботи, що дозволяє проводити тестування окремих компонентів інтерфейсу системи.

4. Набула подальшого розвитку інформаційна технологія, яка ґрунтується на комплексному застосуванні запропонованих моделі та методів, що дозволяє

підвищити ефективність тестування за рахунок прискорення процесу тестування програмного забезпечення розподілених систем.

Практичне значення одержаних результатів. Створена за запропонованою інформаційною технологією інформаційна система, що реалізує представлені в роботі моделі та методи, може бути використана для вирішення завдань забезпечення тестування розподілених систем обробки інформації. Проведені експериментальні дослідження тестування інформаційної системи підтверджують можливість її застосування для розподілених систем обробки інформації. Результати виконаних розрахунків та оцінок можуть стати основою для підготовки рішень з управління розробкою програмного забезпечення.

Практичне значення результатів роботи підтверджено актами впровадження інформаційної системи ТОВ «Нафтогазбудінформатика» та у конструкторському бюро інформаційних систем КПІ ім. Ігоря Сікорського, довідкою про застосування результатів роботи у навчальному процесі кафедри технічної кібернетики Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

Особистий внесок здобувача. Усі наукові результати дисертації одержано автором самостійно. У друкованих працях, опублікованих у співавторстві, йому належить таке: аналіз методів моніторингу роботи гетерогенних розподілених баз даних [1]; обґрунтування вимог до вибору параметрів, за якими тестують комп’ютерні компоненти [2]; метод тестування з використанням контрактів [3]; метод тестування інтерфейсів з використанням симуляторів [4]; дослідження питань побудови моделі захищеного коду з використанням віртуальних машин [5]; тестування обфускації програмного коду [6]; дослідження використання в мовах програмування високоточних обчислень [7]; модель тестування розподілених систем на основі служб управління [8]; підхід до методу тестування програмного коду після обфускації вставкою інструкцій [9]; розроблення методу тестування програмного забезпечення розподілених комп’ютерних систем [10]; побудова спрощеної математичної

моделі тестування ресурсів розподіленої системи [11]; аналіз використання мережоцентричного підходу в розподіленій комп'ютерній системі [12]; тестування бездротової комп'ютерної системи з мережоцентричним управлінням [13].

Апробація результатів дисертації. Основні результати дисертаційних досліджень доповідалися й обговорювалися на таких конференціях і семінарах:

– XXIV Міжнародна конференція з Автоматичного Управління «АВТОМАТИКА 2017» (Київ, 2017).

– Міжнародна наукова конференція «Інтелектуальні системи прийняття рішень та проблеми обчислювального інтелекту» (ISDMCI) (Залізний порт, 2018).

– 2018 IEEE First International Conference on System Analysis & Intelligent Computing (SAIC) (Kyiv, 2018).

– 6th International Conference on Control and Optimization with Industrial Applications (COIA) (Baku, 2018).

– 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS) (Metz, 2019).

– 5th IEEE International Symposium on Smart and Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS 2020) (Dortmund, Germany).

Публікації. Результати дисертації опубліковано в 13 друкованих працях. Статей – 7, з яких 1 стаття – у науковому фаховому виданні України з Переліку, затвердженого МОН України; 4 статті в наукових журналах, включених до міжнародних наукометричних баз; 2 статті в наукових журналах, включених до Scopus; 6 публікацій у працях і тезах доповідей міжнародних та всеукраїнських наукових конференцій, з яких 3 публікації в наукових конференціях, включених до Scopus та/або Web of Science.

Структура та обсяг роботи. Дисертаційна робота складається зі вступу, 4 розділів, висновків, списку використаних джерел (153 найменування на

18 сторінках), 3 додатків (на 13 сторінках). Основний текст роботи викладено на 129 сторінках. Загальний обсяг роботи становить 176 сторінок.

РОЗДІЛ 1
ПОРІВНЯЛЬНИЙ АНАЛІЗ МЕХАНІЗМІВ ТЕСТУВАННЯ ДАНИХ В РОЗПОДІЛЕНИХ СИСТЕМАХ

1.1 Аналіз організації, зберігання, обробки та тестування даних в сучасних системах

1.1.1. Аналіз видів тестування програмного забезпечення систем

В даний час при розробці програмного забезпечення додатки, що розробляються, реалізовані на різних мовах програмування та складаються з різних компонентів, а також застосовуються різні технології для різних платформ. В таких системах при взаємодії між її компонентами зростає складність тестування[123, 78]. Візьмемо наприклад додаток, який складається з серверної частини, клієнтської частини. При чому клієнтська частина може бути орієнтована на різні види додатків: веб-додаток, мобільний додаток, Linux додаток, Windows додаток та MacOS додаток. Серверна частина буде складатися з бази даних та серверних засобів, які працюють за різними протоколами обміну даними[99,137].

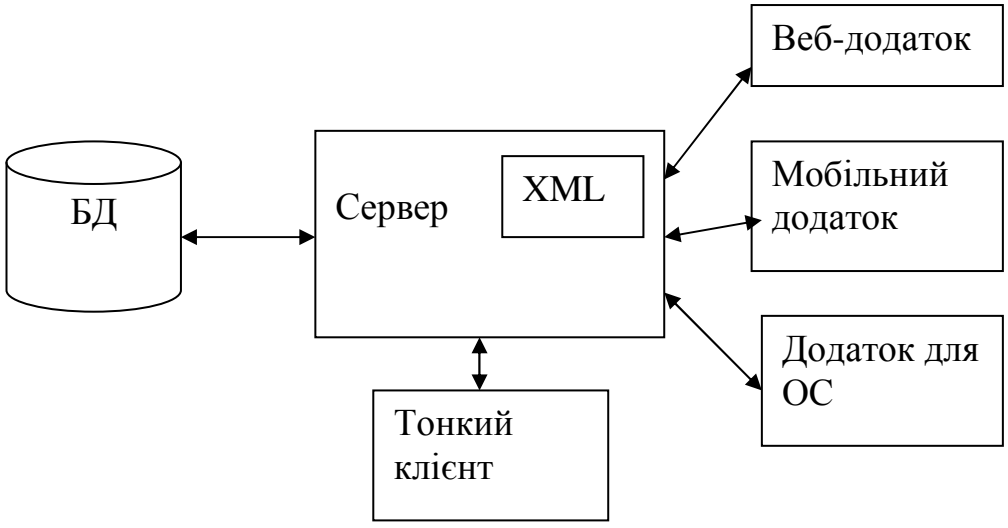


Рис. 1.1. Типова система з різними видами додатків

Зі збільшенням складності програмного забезпечення зростає складність його розробки та значно зростає складність його тестування в зв’язку з підвищеним рівнем вимог до якості програмного забезпечення. При розробці таких складних систем проводиться його обов’язкове тестування не тільки на фінальному етапі розробки, а і при розробці кожного модуля та сервісу. Фінальне тестування проводиться після компіляції всього готового продукту і, як показує практика, успішне фінальне тестування залежить від кількості тестувань на початку та під час розробки[28,80].

В зв’язку з тим, що для забезпечення мети підвищення якості програмного забезпечення на сьогоднішній день єдина правильна методика є тестування. Вона була вибрана шляхом великих об’ємів досліджень та застосування знань багатьох науковців. Особливо активно застосовуються методики тестування для розробки комерційних додатків, причому при їх розробці частину процесів тестування виконують працівники, які пишуть код, а іншу – спеціальна група тестувальників[33, 143]. Як правило, в одному проекті співвідношення тестувальників до розробників – як один до трьох.

Існують наступні види тестування[19, 111]:

- тестування певної одиниці модуля програми: класів, методів або інших компонентів забезпечує блокове тестування. Даний вид тестування забезпечується самим програмістами як одним, так і групою. Воно виконується окремо від інших елементів модулю;
- тестування певної одиниці програми: модуля, пакету або іншої частини, що виконується окремо від інших частин, забезпечує компонентне тестування;
- для забезпечення взаємодії мж пакетами, компонентами або підсистемами програмного забезпечення використовують інтеграційне тестування. Воно проводиться групою програмістів або тестувальників при умові, що у повністю готові компоненти, які тестуються та прописана взаємодія між ними;

- при розробці програмного забезпечення завжди виникають випадки, коли потрібно виконувати тести по кілька разів для забезпечення повного виявлення дефектів в програмі.

На верхньому рівні тестування це процес виконання програмного забезпечення при повній функціональності та інтеграції з іншими програмними системами[95, 139]. Тому такі параметри тестування, як безпека, продуктивність, синхронізацію та інші тестують тільки після повної компіляції програмного продукту[19].

Всі рівні тестування застосовуються при розробці програмного забезпечення, не виділяючи їх в окрему фазу життєвого циклу розробки. По мірі зростання складності розроблюваного програмного продукту для гарантованого тестування потрібно проводити все більше тестів, тому компанії збільшують штат працівників – тестувальників [58, 129]. Але при розробці великих проектів все рівно виникає необхідність застосовувати засоби автоматизованого тестування програмних продуктів, які базуються на основі сучасних методологій тестування програмного забезпечення[97, 117].

Для розробки розподілених систем оброблення інформації основна увага приділяється взаємодії між компонентами розподіленої системи. Дана взаємодія будується по різних протоколах та різних каналах зв’язку. При виникненні ситуації, коли при передачі пакетів порушується логіка, тоді потрібно перевіряти наявність тих компонентів, які працюють некоректно. Некоректна робота перевіряється співпадінням очікуваної відповіді на тест системи, причому для різних моментів часу і ситуацій відповідь на тест може бути різною[71, 139]. При тестуванні великих систем, які містять клієнтські частини та серверні, важливо перевіряти не тільки роботу системи при невеликому числі користувачів, а і при максимальних навантаженнях, які можуть привести до непроходження тестів та збоїв у роботі розподіленої системи. Важливою характеристикою також стає час реагування системи на запити, які до неї відправляються[14, 18].

1.1.2 Аналіз підходів до тестування програмного забезпечення в сучасних системах

Сучасні системи поділяються на різні види і для різних видів використовується різний вид тестування. Для різних рівнів систем знаходження помилок в роботі програмного забезпечення призводить до розробки механізмів автоматизації[65, 122]. У цьому допомагає безперервне тестування, яке гарантує дотримання швидкості розробки та забезпечує потрібну якість. При безперервному тестуванні використовується конвеєрний тип складання автоматизованих тестів і застосування їх до розгорнених середовищ. Збірка, тестування і розгортання середовища при постійно зростаючій кількості вузлів системи за допомогою ручних методів неефективні [32, 49].

При автоматизованому тестуванні працівники використовують основну концепцію, яка виражена в піраміді тестів. ЇЇ запропонував Майк Кон у своїй книзі «Scrum: гнучка розробка ПО» (Succeeding With Agile. Software Development Using Scrum). В його піраміді існує залежність між рівнями тестування та обсягами тестів, які потрібно виконати (рис. 1.2.)

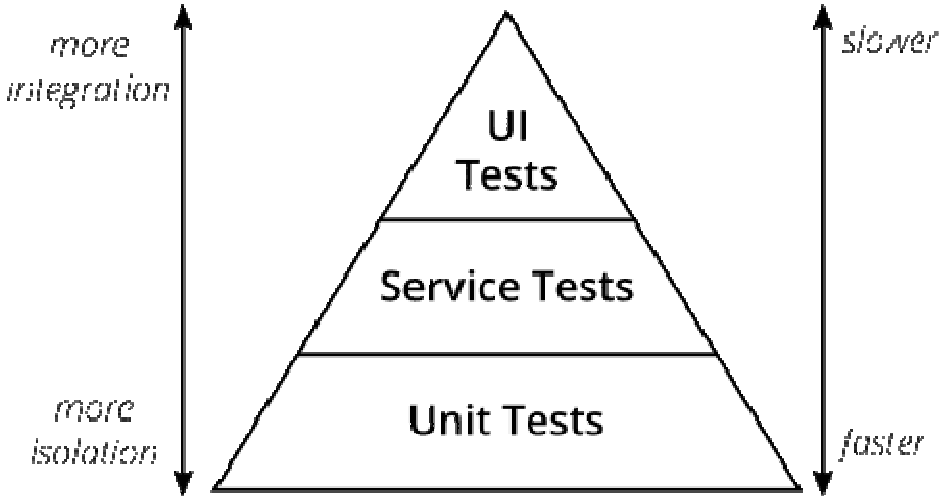


Рис. 1.2. Піраміда тестування Майка Кона

Оригінальна піраміда тестів Майка Кона складається з трьох рівнів (знизу до верху)[141, 153]:

- юніт-тести;
- сервісні тести (API тестування);
- наскрізні тести призначеного для користувачів інтерфейсу.

При застосуванні піраміди Майка Кона швидкість та ізольованість тестів змінюється за наступним законом[86, 125]:

- швидкість виконання тестів зменшується знизу вгору (від юніт – до тестів призначеного для користувача інтерфейсу);
- ізольованість тестованих об'єктів зменшується знизу вгору (від юніт - до тестів призначеного для користувача інтерфейсу).

Через свою простоту у використанні тестова піраміда є тим механізмом, який застосовували при створенні наборів тестів. Можна зробити висновок, що існує два принципи [21, 47]:

- писати тести різної деталізації;
- чим вище рівень, тим менше тестів.

Згідно механізмів тестування піраміди, варто писати багато тестів з невеликим об’ємом та швидких юніт-тестів. Для написання більш загальних тестів потрібно використовувати зовсім мало високорівневих наскрізних тестів, які перевіряють програмне забезпечення від початку до кінця[44, 147]. При цьому потрібно слідкувати за тим, щоб в результаті механізми використання піраміди не стали надто повільні та такі тести буде складно реалізовувати та вони не даватимуть точного результату[83, 151].

1.2 Дослідження особливостей побудови тестів різних рівнів програмного забезпечення сучасних систем

Для тестування різних компонентів програмного забезпечення використовуються два різних типи тестових дублерів, а саме: імітації (mocks) та заглушки (stubs). При використанні в тестуванні ці терміни є взаємозамінними. Заглушки замінюють реальний компонент системи (наприклад, клас, модуль або функцію) підробленою копією цього компонента[67]. Даний механізм дає

можливість викликати компоненти, які виглядають як їх оригінали та давати відповіді цим компонентам, які абсолютно аналогічні оригінальним (наприклад, такі ж відповіді на ті ж виклики методів), але це заздалегідь встановлені еталонні відповіді, які тестувальник сам визначає для юніт-тесту[127-129].

Тестові дублери використовуються не тільки в юніт-тестах. Більш складні механізми застосовуються для контрольованої імітації цілих частин системи, що тестується. Однак в юніт-тестах використовується особливо багато імітацій і заглушок тому, що безліч сучасних мов програмування та бібліотек дозволяють легко і зручно їх створювати та застосовувати[121].

Незалежно від обраної технології, в стандартних бібліотеках мови програмування або при застосуваннях популярних сторонніх бібліотек вже є способи налаштування цих імітацій[93].

Швидкість виконання юніт-тестів залежить від напряму, від швидкості роботи програмного забезпечення, а те в свою чергу залежить від параметрів віртуальних серверів, на яких вони встановлені. Таким чином, можна прогнати тисячі модульних тестів за кількох хвилин. Відповідно тестові сценарії повинні бути узагальнені: заглушка зовнішніх учасників, настройка вхідних даних, виклик тестового суб'єкта та перевірка, що повертає значення відповідає очікуваному, тобто еталонному[144, 152]. Розробка програмного забезпечення через таке тестування може допомогти створити добре підтримувану архітектуру, яка автоматично застосовує повністю автоматизований набір таких тестів[15].

Юніт-тести застосовуються для всіх класів коду при різних етапах розробки незалежно від їх набору функціональності або того, до якого рівня внутрішньої структури вони належать. Такі тести підходять для контролерів, репозиторіїв, класів предметної області або програм зчитування та запису файлів. При застосуванні таких тестів працює правило: один тестовий клас на один клас розробки[112, 152].

Юніт-тести забезпечують тестування відкритого інтерфейсу класу. При цьому немає можливості для тестувальника тестувати закриті методи, тому що

їх не можна викликати з іншого тестового класу. Для тестування таких захищених або доступних методах в межах пакету (package-private) або методах, доступних з тестового класу, потрібно проводити зміну коду, що може не дати правильного тестування[38, 127]. Вони повинні гарантувати, що перевірені всі нетривіальні шляхи коду, в тому числі і сценарій за замовчуванням і ситуації-екстремуми будуть відтестовані. У той же час вони не повинні бути прив'язані до реалізації програмного забезпечення[44, 67].

Для застосування структури всіх тестів (не тільки модульних) повинна виконуватися наступна методика[79, 81]:

- налаштування тестових даних;
- виклик тестованого методу;
- перевірка еталонних результатів тестування.

Застосування цієї методики приводить до використання структури трьох елементів Arrange, Act, Assert. Також можна застосовувати іншу концепцію, яка заснована на описі поведінки програмного коду, де «дано» відображає налаштування, «коли» – виклик методу, а «тоді» - твердження, яке потрібно перевірити[75, 123].

Цей шаблон можна застосувати не тільки для цих тестів, а і до інших видів більш високорівневих тестів. У кожному разі він гарантує, що тести виконуються досить швидко та на їх написання не потрібно витрачати багато часу[61, 148].

Всі нетривіальні додатки програмного забезпечення, що інтегровані з деякими іншими частинами, тобто базами даних, файловими системи, мережевими викликами до іншої системи, можуть бути тестовані із застосуванням вищезгаданих механізмів. Для юніт-тестів, які імітуються для кращої ізоляції і підвищення швидкості тестування програмного забезпечення, потрібно проводити аналіз взаємодії з іншими частинами[18, 46]. Для таких цілей використовуються інтеграційні тести, які проводять перевірку інтеграції додатків з усіма компонентами, які використовуються за їх межами із зовнішніх систем[19, 23].

Для автоматизованих тестів це означає, що потрібно запустити не тільки власний додаток, а й інтегрований компонент. Якщо тестується інтеграція з базами даних, то при виконанні тестів треба встановлювати та запускати служби системи управління баз даних[25, 137]. Щоб перевірити читання файлів з диска, потрібно зберегти файл на диск і завантажити його в інтеграційний тест. Разом з контрактним тестуванням і виконанням контрактних тестів на імітаторах і реальних реалізаціях потрібно складати інтеграційні тести, які швидші, але більш незалежні та зазвичай простіші в написанні[55, 67].

Інтеграційні тести – це тести, які виконують функцію тестування, що призводить до інтеграції із зовнішньою частиною (файлової системою, базою даних, окремим сервісом).

Тест інтеграції бази даних з системним додатком виглядає наступним чином[113, 129]:

- запуск бази даних;
- підключення додатку до БД;
- запуск функції в коді, яка записує дані в БД;
- перевірка, що очікувані дані записані в базу шляхом їх читання з бази даних.

При написанні інтеграційних тестів потрібно локально запускати зовнішні залежності: локальну базу даних, локальне сховище файлів і т.д. При проходженні інтеграції з окремою службою екземпляр цієї служби запускається локально або створюється і запускається аналогічна версія, яка імітує поведінку реальної служби[88, 141].

При відсутності можливості локально запустити сторонню службу проводиться запуск виділеного тестового об’єкту та проведення тестування, підключивши інтеграційні тести на цю службу. В автоматизованих тестах потрібно уникати інтеграції з реальною розроблювальною системою. Для тестування інтеграції з сервісом по мережі також використовуються інтеграційні тести, але для такого варіанту тестування написання цих тестів

займає багато часу та вони будуть виконуватися повільніше по відношенню до наскрізних тестів[48, 124].

Як видно з піраміди тестування, інтеграційні тести знаходяться на більш високому рівні, ніж модульні тести. Інтеграція файлових систем і баз даних зазвичай набагато повільніша, ніж виконання юніт-тестів з їх виконуваними імітаціями. Також збільшується складність у написанні в порівнянні з модульними тестами, які писати значно легше[53, 67]. Потрібно визначати механізми для зовнішньої частини тесту. Але при цьому у них є перевага, яка полягає у тому, що після тестування програми з усіма зовнішніми частинами програмне забезпечення буде працювати вірно.

При присутності в проектів користувацького інтерфейсу, такого наприклад, як веб-інтерфейс, потрібні тести, які будуть перевіряти його роботу. Тести користувацького інтерфейсу перевіряють правильність роботи призначеного для користувача інтерфейсу додатку. При вирішення задачі тестування вони повинні ініціювати правильні події в інтерфейсі, при яких дані повинні представлятися користувачеві, та стан користувацького інтерфейсу повинен змінюватися очікуваним чином[151]. Абсолютно аналогічним чином працюють наскрізні тести. Повне тестування додатків означає проходження через призначений для користувача інтерфейс всіх необхідних тестів.

Для тестування користувацького інтерфейсу традиційних веб-додатків призначені спеціальні інструменти тестування, наприклад, Selenium. Якщо інтерфейсом користувача є інтерфейс REST API, то розробляються правильні наскрізні тести за допомогою принципів цього API[92, 136-138].

У веб-інтерфейсах потрібно перевіряти наступні тести для користувацького інтерфейсу: тести поведінки інтерфейсу, тести правильної верстки для різних браузерів, тести зручності користування, тести дотримання стилю, обраного для даного продукту, та ін.[41, 87].

Тестування поведінки користувацького інтерфейсу проводиться за допомогою сучасних фреймворків для односторінкових додатків (react, vue.js, Angular та інші), які містять у собі інструменти для ретельного тестування цих

взаємодій на досить низькому рівні (в юніт-тесті). При написанні власної реалізації фронтенду, наприклад, на мові програмування JavaScript, все одно можна використовувати звичайні інструменти тестування, такі, як Jasmine і Mocha. Для традиційних додатків з рендерингом на стороні сервера найкращим вибором стануть тести на основі Selenium або Cypress[48, 112].

Тестування цілісності верстки веб-додатку перевіряти складніше в порівняні з тестуванням поведінки користувацького інтерфейсу. В залежності від програми та потреб користувачів виникає необхідність додатково перевіряти, що при внесені змін до програмного забезпечення не порушується верстка веб-додатку[53, 128].

Існує багато інструментів для автоматизованої перевірки дизайну веб-додатку в готовому програмному забезпеченні. Більшість з них використовують Selenium для відкриття веб-додатків в різних браузерах і форматах, створення скріншотів і порівняння з раніше зробленими скріншотами. Якщо старий і новий скріншоти відрізняються, то дані програмні інструменти видають на виході повідомлення тестувальнику про невідповідність у користувацькому інтерфейсі[47, 106].

Тестування розгорнутого додатку через користувацький інтерфейс – це тестування, яке найбільше займає часу та проводиться при розробленні програмного забезпечення розподілених систем обробки інформації. Описані вище тести користувацького інтерфейсу проводяться наскрізним способом через WebDriver[124- 125]. Наскрізні тести дають максимальне покриття множини відповідей та визначають, в кінцевому варіанті, працює програмне забезпечення чи ні. Selenium і протокол WebDriver дозволяють автоматизувати тести шляхом автоматичного надсилання headless-браузера на розгорнуті сервіси для виконання натискань, введення даних і перевірки стану користувацького інтерфейсу[20, 98].

Проте застосування наскрізних тестів для тестування розподілених систем приводить до неповного покриття всіх можливих варіантів поведінки програми та можуть виникати проблеми у роботі програмного забезпечення з невідомих і

непередбачуваних причин, наприклад, це хибнопозитивні проблеми у роботі додатків. Чим складніший користувацький інтерфейс, тим більша ймовірність невірного тестування. Велика кількість браузерів, проблеми з синхронізацією даних, анімація і несподівані спливаючі діалоги в веб-додатках призводять до збільшення часу написання тестів та часу їх проходження[81, 95]. Для уникнення таких ситуацій тестування головних «маршрутів», по яких ходять запити від користувачів, потрібно проводити автоматизацію таких тестів. Наприклад, при тестуванні інтернет-магазину найдовшим і найскладнішим тестом буде тест, який стосується пошуку продукту, переміщення його в кошик та оформлення відповідного замовлення[117-119]. При коректній роботі за таким варіантом тестування наскрізний тест проходитиме вдало, якщо працюють всі ланки тестування, а якщо якась з них не працює, то тестувальник не може сказати, яка ланка працює неправильно.

При використанні піраміди тестів тестувальник пише багато низькорівневих тестів, які забезпечують тестування всіх варіантів граничних варіантів роботи та інтеграції з іншими частинами системи. Тоді необхідність повторювати ці тести на більш високому рівні не потрібна. При таких тестах потрібно постійно забезпечувати якісне технічне обслуговування та все рівно багато тестів, які виявлять помилки, призведуть до сповільнення тестування програмного забезпечення системи[74, 137]. Для проведення наскрізних тестів тестувальники як правило вибирають Selenium і протокол WebDriver, причому Selenium може застосовуватися для різних версій та типів веб-браузерів.

Більш новий підхід полягає у використанні headless-браузера (тобто браузера без користувальницького інтерфейсу) для тестів WebDriver. До недавнього часу найчастіше для автоматизації браузерних завдань використовувався PhantomJS. Але коли Chromium і Firefox впровадили headless-режим, PhantomJS перестав бути актуальним[29, 57]. Самим надійним способом протестувати сайт є тестування за допомогою реального браузера, який використовується користувачами (наприклад, Firefox чи Chrome), а не з імітатору браузера.

1.3 Механізми побудови систем тестування в сучасних розподілених системах

Представлені в цьому розділі механізми юніт-тестів (модульних тестів) можуть бути застосовані тільки на низькорівневому тестування і їх як правило при розробці програмного забезпечення проводять самі розробники, а не окремі тестувальники. Інтеграційне тестування проводиться уже тестувальниками при готовності самих пакетів програмного забезпечення. А наскрізне тестування користувацького інтерфейсу проводиться тільки при готовому програмному забезпеченні[34, 150]. Загалом архітектори процесів тестування визначають терміни, як правило, в межах однієї компанії розробки. Терміни узгоджуються в межах команди. Загалом стандарт International Software Testing Qualifications Board (ISTQB) є рекомендованим ресурсом для опису прийнятної термінології тестування в світі, а для України не існує стандарту , який описував тільки тестування, але існує державний стандарт України, що описує весь життєвий цикл програмного забезпечення[37, 127].

При використанні безперервної інтеграції програмного забезпечення тестування повинно проводитися автоматизовано і запускатися при внесенні змін. Цей процес розділяється на кілька етапів, які поступово тестують всі компоненти програмного забезпечення та приводять до встановлення його в робочому середовищі[87, 98]. Інформація про виконання тестів повинна надходити протягом декількох секунд та записуватися у базу знань. Етапи тестування безперервної інтеграції програмного забезпечення визначаються не типами тестів, а їх швидкістю і областю дії, тому потрібно розміщати в разі потреби деякі з інтеграційних тестів на ту ж стадію, що і юніт-тести, для забезпечення більш швидкого зворотного зв'язку[100-102].

Для тестувальника дуже важливо уникати дублювання тестів на різних рівнях піраміди. Кожен тест в тестовому наборі – додатково затрачений час на

написання та тестування. У контексті реалізації піраміди тестів виникають два емпіричних правила по уникненню дублювання[59, 130]:

- якщо в тесті більш високого рівня виявлена помилка, а в тестах більш низького рівня немає, то необхідно писати тест нижчого рівня;
- потрібно зміщувати тести якомога нижче за рівнями піраміди.

При першому правилі тести більш низького рівня краще дозволяють звузити область застосування та ізольовано відтворити помилку. Вони працюють швидше і займають менше місця, що допомагає при налагодженні вручну. Друге правило важливе для швидкого виконання набору тестів. Якщо повністю покрити всі варіанти на тестах нижчого рівня, то в тесті більш високого рівня немає необхідності. Набір тестів буде працювати повільніше, і при зміні коду потрібно буде змінити більше тестів[49, 117].

Для розподілених систем обробки інформації тест більш високого рівня дає більше впевненості, що система працює правильно. Написання модульного тесту для класу контролера допомагає перевірити логіку всередині самого контролера. Але неможливо підтвердити, чи дійсно кінцева точка REST, яку надає цей контролер, відповідає на запити HTTP. Таким чином, відбувається рух вгору по піраміді тестів і додається тест, який повинен перевіряти цю ситуацію[47-50].

Правила виключення тестів, що є надлишковими для покривання певної області тестування програмного забезпечення[86, 122]:

- видаляти високорівневі тести, які вже покриті на більш низькому рівні (враховуючи, що вони не дають додаткової цінності);
- замінювати тести більш високого рівня тестами нижчого рівня, якщо це можливо.

У традиційному підході до розробки програмного забезпечення вважається, що завдання автоматичного тестування і моніторингу програмного забезпечення не перетинаються. Але в розподілених системах обробки інформації тестування та моніторинг програмного забезпечення проводиться

одночасно в рамках завдання управління відмовами при розробці компонентів програмного забезпечення[133].

Під управлінням відмовами ми будемо розуміти визначення, реєстрацію, повідомлення персоналу і (якщо це можливо) автоматичне усунення несправності з тим, щоб дозволити розподіленій системі обробки інформації виконувати функції по її прямому призначенню. Незважаючи на те, що дане визначення часто застосовується в області мережевих технологій, воно також може бути застосовано і щодо розробки розподіленого програмного забезпечення[24, 78].

Спроби створення інструментарію, придатного для вирішення завдань управління відмовами складних програмних систем, орієнтовані на роботу з програмними продуктами певних постачальників і не є універсальними. Основні підходи до вирішення завдання управління відмовами в разі розробки розподіленого програмного забезпечення – це автоматизоване тестування та моніторинг[66, 134].

Проведемо аналіз кількох технічних рішень, які використовуються при розробці системи моніторингу розподіленої системи обробки інформації. Зокрема, застосування напівструктурованих документів в форматі XML для вирішення проблеми зберігання, забезпечення прозорого доступу та візуалізації відомостей довільної структури, які зберігаються в технологічному журналі системи[41, 87]. При розробці ПО неминуче виникає завдання управління відмовами, що відповідає специфіці розроблюваної системи.

При варіанті, коли програмна система нерозподілена, завдання управління відмовами часто звужується до коректної обробки виняткової ситуації, видачі відповідного повідомлення про помилку користувачеві і/або занесення повідомлення в журнал. Після чого функціонування програмної системи або продовжується, або (в разі критичної ситуації) припиняється. Тестування програмних компонентів може проводитися без урахування їх оточення і без належної верифікації заданих параметрів конфігурації системи[50, 98].

При варіанті, коли програмна система розподілена, подібний підхід до управління відмовами неприйнятний. По-перше, в розподілених програмних комплексах значна частина критичних ситуацій буває обумовлена помилками розгортання або конфігурації. По-друге, система збору даних про події, що відбулися у виняткових ситуаціях, та відмовах повинна також мати властивості розподіленої системи. Також повинен існувати єдиний інтерфейс подання інформації про виниклі проблеми[95, 105].

Проведемо порівняльний аналіз функціональності систем моніторингу та тестування додатків. Вимоги до функціональності розділені на загальні вимоги (що відносяться як до тестування, так і до моніторингу), вимоги до систем тестування і вимоги до систем моніторингу додатків. В даному порівняльному аналізі були розглянуті чотири системи: MsBuild, XUnit, IBM Autonomic Computing Toolkit (далі ACTK) і MS AsmL. Система IBM ACTK складається з безлічі компонентів, однак у відкритому доступі є тільки Log & Trace Analyzer.

IBM ACTK – система моніторингу і аналізу функціонування розподілених систем, призначена для розробки засобів автоматизації адміністрування великих програмно-апаратних комплексів.

MS AsmL – розробка корпорації Microsoft в області тестування програмних компонентів, що відрізняється від типових способів автоматизованого тестування (Unit-тестування) можливістю побудови моделі виконуваної програми.

XUnit - де-факто індустріальний стандарт в області автоматичного модульного тестування для платформи .NET.

MsBuild - система забезпечення постійної готовності останніх збірок і їх автоматичного тестування для платформи .NET.

Перші дві системи забезпечують безперервний аналіз роботи готових програмних компонентів. Решта продукти призначені для багаторазового збирання та тестування програмного забезпечення. Система MsBuild була обрана в силу початкової орієнтації на роботу з додатками для платформи .NET. Вибрані системи покривають основні потреби автоматизованого тестування та

покривають потреби моніторингу. У таблиці 1.1 наведені результати порівняльного аналізу чотирьох обраних систем на предмет задоволення ними потреб автоматизованого тестування.

Таблиця 1.1.

Характеристики систем тестування

Характеристики	IBM ACTK	MS Asml	MsBuild	XUnit
Виконання тестового сценарію	-	+	+	+
Інтеграція з вбудованою системою безпеки	-	-	+	+
Тестування оточення і конфігурації	-	-	-	+
Ініціалізація оточення системи	-	-	+	-
Збір інформації про результати тестування різних компонентів	-	-	+	+
Пошук і фільтрація результатів тестування	-	-	+	-

З результатів порівняння добре видно, що основні потреби тестування покриваються спеціалізованими продуктами MsBuild і бібліотеками сімейства XUnit. Однак навіть успішне виконання тестових сценаріїв на етапі розробки розподіленої системи не дає достатніх підстав для того, щоб зробити висновок про працездатність системи в реальному середовищі. Тому потрібно інтегрувати систему підтримки автоматизованого тестування в розроблювану розподілену систему. Це дозволить проводити тестування на етапі розгортання та на етапі промислової експлуатації системи[35, 147].

1.4 Огляд сучасних засобів для тестування розподілених систем обробки інформації

Проведемо оцінку сучасних засобів для тестування розподілених систем розробки інформації. Для цього розглянемо основні фреймворки, які застосовуються для тестування, а саме: Jepsen, QuickCheck, Catcher.

Спочатку розглянемо використання Jepsen, який є фреймворком для функціонального наскрізного тестування розподілених баз даних, черг, кешів та ін.. Написаний на мові програмування Clojure[90, 140].

Jepsen імітує мережеві помилки, потім генерує випадкові операції до розподіленої системі. Далі перевіряє, яким чином ці операції були застосовані до розподіленої системі, до еталонної її поведінки, до моделі цієї розподіленої системи, і виявляє проблеми у системі.

Якщо при тестуванні Jepsen знайшов якусь проблему, то він шукає для розподіленої системи контрприклади її роботи та застосовує їх. Також, Jepsen-тести носять імовірнісний характер, якщо по результатам тестів нічого не знайдено, то це не значить, що розподілена система працює без помилок. На рисунку 1.3. зображена структурна схема тестування розподілених систем обробки інформації.

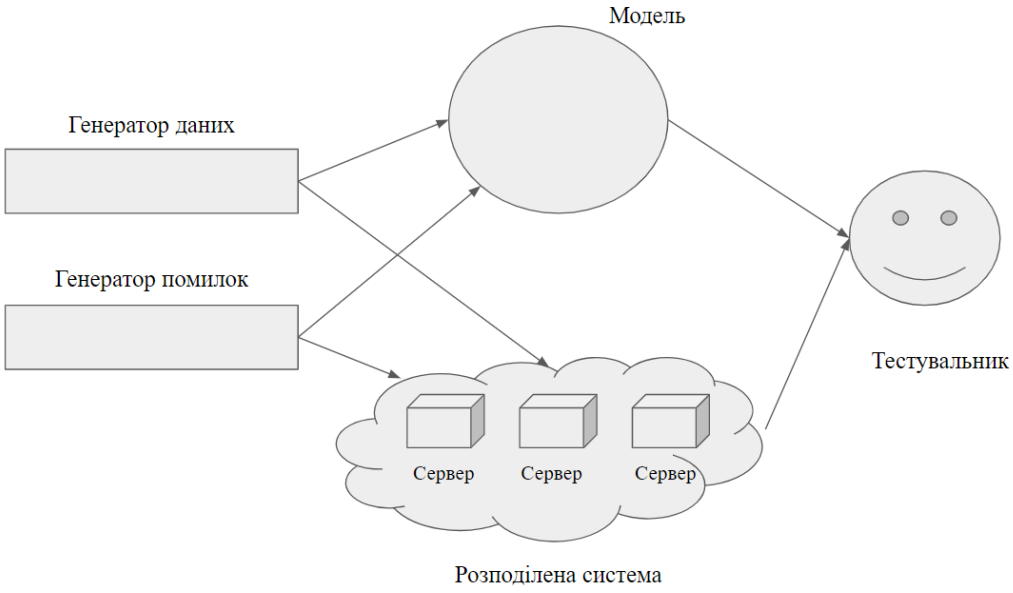


Рис. 1.3. Структурна схема тестування за допомогою Jepsen

В даній схемі відбувають наступні етапи: генератор генерує випадкові операції, які застосовуються до розподіленої системи, далі розподілена система може бути на реальних серверах або розгорнута в Docker, тобто на віртуальних серверах. Використовується еталонна модель, яка описує поведінку розподіленої системи. З результатами поведінки еталонної моделі порівнюються отримані результати тестів та приймається рішення про правильність роботи системи[58, 113].

У Jepsen є готові механізми тестування для різних баз даних, кешів і тд. Також можна створювати свої механізми тестування. Готові механізми тестування даних в Jepsen існують для наступних технологій:

- Consul.
- Mongodb-smartos.
- Rabbitmq.
- Aerospilke.
- Crate.
- Elasticsearch.
- Mysql-cluster.
- Rethinkdb.
- Postgres-rds.
- Zookeeper.
- Chronos.
- Disque.
- Etcd.
- Logcabin.
- Percona.
- Robuscitc.
- Cockroachdb.
- Doc.

- Galera.
- Mongodb-rocks.
- Postgres-rds.

Основна проблема роботи з Jepsen заключається в тому, що він написаний на мові програмування Lisp, точніше на її діалекті Clojure, тому тести також потрібно писати на Clojure. В зв'язку з тим, що мова програмування Clojure появилася недавно та є складною мовою для вивчання, то станом на сьогоднішній час знайти програміста, який володів нею, практично неможливо.

Перевагою застосування Jepsen є універсальність. Це пов'язано з тим, що цю технологію можна використовувати для будь-яких баз даних. Проте недоліками є те, що тести носять імовірносний характер, тести потрібно запускати багато разів, а це призводить до втрати часу, та спеціалістів, які б проводили розробку та модифікацію на мові програмування, на якому розроблений Jepsen, є дуже мало[27, 69].

Наступним розглянемо тестовий фреймворк QuickCheck, який дозволяє визначати інваріанти або властивості коду. Тестування відбувається за допомогою визначення властивостей програмного коду та перебором всіх можливих варіантів поведінки цих властивостей. При запуску він генерує безліч випадкових вхідних даних, на яких запускає код і перевіряє, чи функціонал не порушено. Основною перевагою такого способу є велика кількість вхідних даних та висока ймовірність охопити граничні випадки, які міг не розглядати розробник чи тестувальник[17, 29].

QuickCheck дозволяє налаштувати кількість запущених тестів для кожного масиву вхідних даних. Для розподіленої системи з величезною кількістю комбінацій, розкладів, типу команд, відключень мережі, затримок мережі, тайм-аутів, помилка може бути знайденою в будь-якому місці.

Переваги QuickCheck є:

- відкритий код;
- міжплатформовий фреймворк;
- можливість паралельного виконання тестів.

Проте у QuickCheck є кілька недоліків, а саме:

- складна діагностика непройдених тестів;
- складне програмування скріптів виконання.

Тестовий фреймворк Catcher – гнучкий наскрізний інструмент тестування, який можна використовувати для автоматизованого тестування розподілених API. Це допомагає перевірити взаємодію одного сервісу служби або всіх сервісів системи від інтерфейсу до інтерфейсу. Catcher – працює по HTTP протоколу, а також може перевіряти різні сервіси, такі як Kafka, Postgres, CouchBase, Mongodb, Elastic, S3, електронні листи та інші[74,146].

Catcher виконує сценарії, які пишуться в json/yaml. Сценарій складається з послідовних кроків, наприклад:

```
steps:
  - http:
      post:
        url: '127.0.0.1/save_data'
        body: {key: '1', data: 'foo'}
  - postgres:
      request:
        conf: 'dbname=test user=test host=localhost password=test'
        query: 'select * from test where id=1'
```

Також можна запускати одні скріпти з інших скріптів, що сприяє повторному використанню коду (включаючи запуск тільки частини кроків через систему тегів, відкладений запуск і тд).

В фреймворку Catcher є наступні переваги:

- не потрібно писати код, тільки сценарії;
- переключення на різні середовища роботи системи через конфігураційний файл;
- можна писати власні модулі на будь-якій мові програмування.

Проте недоліком використання фреймворку Catcher є складний для читання синтаксис (в порівнянні з іншими BDD інструментами).

Таблиця 1.2.

Порівняльна таблиця фреймворків для наскрізного тестування

Властивість	Jepsen	QuickCheck	Catcher
Простота написання тестів	-	-	+
Тестування розподілених БД	+	+	+
Тестування розподілених кешів	+	+	-
Тестування розподілених API	-	-	+
Простота діагностики результатів	+	-	-
Імовірнісний характер тестів	+	-	-
Міжплатформовість	-	+	+

Результати порівняння показали, що для тестування розподілених БД та кешів добре підходять Jepsen та QuickCheck, а для тестування розподілених API краще використати Catcher. Проте, Jespen – єдиний фреймворк, який дозволяє легко діагностовувати результати проходження тестів. Оскільки Jespen містить імовірнісний характер тестів, то він може знайти ті помилки в роботі програми, які пропустив тестувальник при плануванні тестового сценарію, але при цьому Jespen тести потрібно запускати кілька разів для кращого аналізу роботи системи. Фреймворк Catcher є найзручнішим для написання тестів, адже там не потрібно писати код, лише сценарії.

Для тестування інтерфейсів користувачів найпопулярнішим фреймворком є Selenium WebDriver, який пропонує інструмент для автоматизованого керування веб-браузерами, що використовується для тестування веб-додатків. Він дозволяє писати тести на різних мовах програмування. Selenium WebDriver працює з браузеразми Google Chrome, Firefox, Internet Explorer, Opera, Safari, а також з мобільними платформами iOS та Android.

Переваги Selenium WebDriver:

- відкритий код;
- може застосовуватись для веб-додатків, написаних на різних технологіях;
- можливість виконувати скрипти в різних браузерах та операційних системах;
- підтримка мобільних пристроїв;
- тести виконуються автоматично і без участі тестувальника.

Недоліки Selenium WebDriver:

- підтримує тільки тестування веб-додатків;
- не підтримує розробку через сценарії та не має внутрішнього сховища об’єктів;
- не має власного середовища для розробки, що впливає на швидкість розробки тестів;
- відсутній доступу до кнопок навігації та функціоналу браузера;
- не містить генераторів звіту по результатам тести;
- параметризація залежить від мови програмування, на якій розробляються тести.

Розглянуті фреймворки для тестування розподілених систем не в повній мірі задовольняють роботу тестувальника програмного забезпечення, якому приходиться вивчати нові технології або писати код в закритих системах.

1.5 Постановка задачі дослідження

Проведений у підрозділах 1.1–1.4 аналіз наявних методів тестування програмного забезпечення систем дає змогу зробити висновок про те, що для забезпечення тестування розподілених систем необхідно розробити інформаційну технологію, що дозволяє оперативно проводити тестування програмного забезпечення при розробці та модифікації систем розподіленої обробки інформації.

Відомі методи і алгоритми не повною мірою використовують ознакову

інформацію для ототожнення проходження тестування, а при тестуванні розподілених систем обробки інформації, використовують не всю сукупність переваг таких систем.

Недоліками розглянутих механізмів тестування програмного забезпечення, які були запропоновані в піраміді Майка Кона, є відсутність можливості тестування окремих сервісів розподіленої системи обробки інформації та застосування наскрізних тестів тестування користувацького інтерфейсу, які приводять до збільшення об’єму написання тесту та зменшення швидкості тестування.

На сьогодні не існує вузької спеціалізованої інформаційної системи тестування розподілених систем з використанням API. Тому розробка моделей, методів та інформаційної технології для підвищення ефективного тестування програмного забезпечення є актуальною задачею.

Для проведення аналізу роботи розроблених сервісів розподілених систем обробки інформації, тестування окремих сервісів та тестування користувацьких інтерфейсів без застосування наскрізного тестування необхідна наявність інформаційної технології, що дозволяє оперативно створювати тести при зміні програмного коду в частині сервісів розподіленої системи.

Виходячи із цього, метою дисертаційної роботи є підвищення ефективності функціонування розподілених систем обробки інформації на основі розробки моделей і методів тестування компонентів таких систем.

Для досягнення мети необхідно вирішити наукове завдання – розробити моделі, методи та інформаційну технологію підтримки ефективного тестування програмного забезпечення.

Об’єктом дослідження є процеси тестування систем розподіленої обробки інформації., а предметом дослідження – моделі, методи та інформаційна технологія підтримки ефективного тестування програмного забезпечення.

Рішення сформульованого вище наукового завдання доцільно розбити на ряд взаємозалежних складових частин, що визначають часткові завдання досліджень. Основними з цих завдань є:

1. Аналіз проблеми використання існуючих підходів до тестування в архітектурі розподілених систем;

2. Розробка модифікованої «піраміди» тестування програмного забезпечення в архітектурі систем розподіленої обробки інформації;

3. Розробка моделі системи тестування сервісів розподіленої системи обробки інформації;

4. Удосконалення методу тестування програмного забезпечення розподілених систем з використання контрактів;

5. Удосконалення методу тестування інтерфейсів користувача розподіленої системи з використанням симуляційних тестів;

6. Розробка спеціалізованого середовища для моделювання та порівняння експериментальних досліджень запропонованих методів тестування розподілених систем.

7. Розробка рекомендацій з використання запропонованих методів.

Рішення показаних вище взаємозалежних часткових завдань визначило структуру та зміст дисертаційної роботи.

1.7 Висновки до розділу 1

Проведено аналіз механізмів тестування програмного забезпечення систем. Показано, що при тестуванні великих систем, які містять клієнтські частини та серверні, важливо перевіряти не тільки роботу системи при невеликому числі користувачів, а і при максимальних навантаженнях, які можуть привести до непроходження тестів та збоїв у роботі системи. Важливою характеристикою також стає час реагування системи на запити, які до неї відправляються.

Розглянуто піраміду тестування Майка Кона. Згідно механізмів тестування піраміди, варто писати багато тестів з невеликим об’ємом та швидких юніт-тестів. Для більш написання загальних тестів потрібно використовувати зовсім мало високорівневих наскрізних тестів, які перевіряють

програмне забезпечення від початку до кінця. При цьому потрібно слідкувати за тим, щоб в результаті механізми використання піраміди не привели до великих затрат часу.

Розглянуті механізми застосування наскрізних тестів. Показано, що тестувальники як правило вибирають Selenium і протокол WebDriver, причому Selenium може застосовуватися для різних версій та типів веб-браузерів. З результатів порівняння способів тестування визначено, що основні потреби тестування покриваються спеціалізованими продуктами MsBuild і бібліотеками сімейства XUnit. Однак навіть успішне виконання тестових сценаріїв на етапі розробки розподіленої системи не дає достатніх підстав для того, щоб зробити висновок про працездатність системи в реальному середовищі. Потрібно інтегрувати систему підтримки автоматизованого тестування в розроблювану розподілену систему, що дозволить проводити тестування на етапі розгортання та на етапі промислової експлуатації системи.

Розглянуті фреймворки для тестування розподілених систем не в повній мірі задовольняють роботу тестувальника програмного забезпечення, якому приходиться вивчати нові технології або писати код в закритих системах.

Аналіз, здійснений у цьому розділі, ґрунтовано на результатах, які автор опублікував у працях [3, 4, 5].

РОЗДІЛ 2
ДОСЛІДЖЕННЯ ТА РОЗРОБКА МОДЕЛЕЙ ТА ЗАСОБІВ ТЕСТУВАННЯ В АРХІТЕКТУРІ РОЗПОДІЛЕНИХ СИСТЕМ

2.1. Розробка підходу до тестування на основі модифікації піраміди тестування Майка Кона

Піраміда тестування Майка Кона була розглянута у Розділі 1 та складається з наступних компонентів: блокові тести (тестуються певні ділянки програмного коду, виклик методів), тести сервісу (тестують системні компоненти в обхід інтерфейсу) і наскрізні тести інтерфейсу користувача (тестують всю систему з використанням графічного інтерфейсу). У напрямку "знизу вгору" область дії тестів збільшується, оскільки спочатку тестуються найпростіші компоненти системи і її окремі функції, потім взаємодія її компонентів, і наступним етапом тестується робота всієї системи. При цьому розмір тестів, складність їх формування і час, витрачений на виконання тестів, також зростають. Відповідно, в напрямку "зверху вниз" знижується складність пошуку причини невиконання тестів: місце збою блокових тестів можна визначити досить точно, в деяких випадках аж до знаходження помилкового рядка в програмному коді, а встановити причину невиконання наскрізних тестів буває дуже складно у зв'язку з їх розміром і складністю формування.

У зв'язку з тим, що піраміда Кона розроблялася для тестування не розподілених систем, виконаємо її модифікацію таким чином, щоб вона повною мірою використовувала архітектуру систем розподіленої обробки даних. Для цього розглянемо різні види тестів і визначимо ефективність їх реалізації.

Блокові тести проводяться на рівні фрагментів коду, тому замінювати їх тестами іншого типу немає сенсу. У модифікованій піраміді Кона, яка зображена на рис. 2.1, нижній рівень залишається без змін. При цьому тести сервісу і наскрізні тести замінюються з урахуванням архітектури розподілених систем.

Рис. 2.1. Модифікована піраміда Майка Кона для тестування розподілених систем обробки інформації

Очевидно, що використання наскрізних тестів в архітектурі розподілених систем є неефективним підходом, оскільки потрібна гарантія, що при розгортанні нової підсистеми в реальних додатках внесені зміни не внесуть конфлікти в роботу інших підсистем. Одним із способів реалізувати це без використання реальних підсистем є використання так званих "контрактів", складених на основі запитів від підсистеми. Під контрактом розуміється код тесту, який запускається в режимі постачальника.

При використанні контрактів необхідно визначити очікування споживача від даного сервісу (постачальника). Ці тести повинні виконуватися по відношенню до окремо взятого постачальника, який знаходиться в ізоляції, тому такі тести будуть значно швидші і надійніші, ніж звичайні наскрізні тести для тестування сервісів API.

Тести сервісу є довготривалими в зв'язку з тим, що їх формування вимагає знань структури всієї системи і структури інтерфейсу. Пропонується частково

замінити їх контрактними тестами, а частково тестами інтерфейсу, які будуть проводити тестування взаємодії інтерфейсу з системою обробки даних.

Для тестування розподіленої системи з 5 вузлів в класичному випадку треба протестувати, як показано на рис. 2.2, три рази шлях по три вузла: вузол 1 - вузол 2 - вузол 3; вузол 1 - вузол 2 вузол 4; вузол 1 - вузол 2 вузол 5. Для цього треба по черзі для кожного з трьох варіантів встановлювати компоненти з трьох вузлів, що вимагає додаткових витрат часу, крім цього, треба встановлювати вузол 1 - 3 рази, вузол 2 - 3 рази. З цього можна зробити висновок, що така система тестування не ефективна для розподілених систем, тому більш ефективним буде використання контрактів, яке дозволить встановлювати вузли всього один раз і при цьому повністю протестувати їх функціонал за допомогою тестів споживача і постачальника.

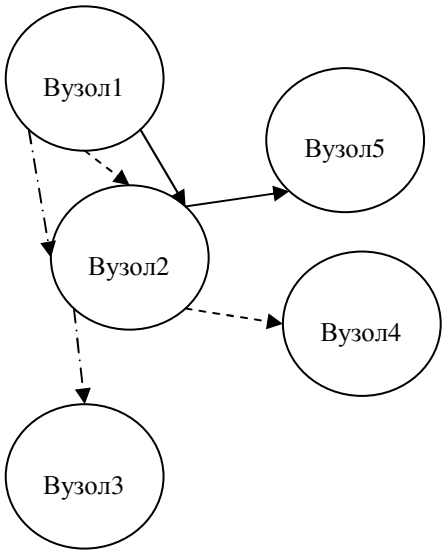


Рис. 2.2. Розподілена система обробки інформації, яка складається з 5 вузлів та зв’язків між ними

Тобто, при тестуванні розподілених систем обробки інформації потрібно проводити модифікацію класичного методу тестування на основі піраміди тестування Майка Кона, що дозволить розширити можливості тестування і застосувати особливості розподілених систем. Розроблено рекомендації щодо подальшого застосування механізмів модифікованої піраміди Майка Кона.

2.2 Формальне представлення розподіленої системи обробки інформації

Розподілена система обробки інформації включає в себе багато підсистем (сервісів), які зв’язані між собою багатьма функціями. Кожна з таких підсистем підтримує свою множину запитів та множину відповідей. Дані підсистеми працюють для вирішення багатьох задач, які ставляться перед розподіленою системою обробки даних.

На рисунку 2.3 представлено приклад розподіленої системи обробки інформації, яка складається з десяти сервісів. На схемі чорним кружком зображено сервіс № 1, через який відбувається доступ до даних, як правило це інтерфейс веб додатка чи мобільного додатка. Сірими кружками позначають сервіси (№ 2), через які користувачі також можуть підключатися до системи так само, як і через сервіс № 1. Сервіси з сірим ободком та білою серединою (№№3,4) – це сервіси, в яких встановлено систему управління базами даних та розміщено відповідні дані, а в сервісах (№№5,7) з сірим ободком та білою сірою серединою встановлено API для роботи з базами даних. В пустих кружках розміщені проміжні сервіси, які виконують функції забезпечення приймання, обробки та передачі запитів від зовнішніх сервісів (фронтенд) до внутрішніх (бекенд).

Дана розподілена система обробки інформації забезпечує доступ до даних шляхом перенаправлення запитів по мережі, яка зображена відповідними лініями. Канал, з якого надходять ці запити до розподіленої системи, позначений стрілкою. В кожного сервісу є своя система обробки даних (запитів) та відповідає мінімальним вимогам до продуктивності вузлів, а канали зв’язку забезпечують передачу пакетів з запитами з швидкістю, яка більша за мінімально допустиму. Від цих двох параметрів залежить швидкість прийняття, обробки та передачі пакету з відповідним запитом до сервісів, в тому числі і сервісів з базами даних та пакетів з підготовленими відповідями до користувача, який звертався до розподіленої системи обробки інформації.

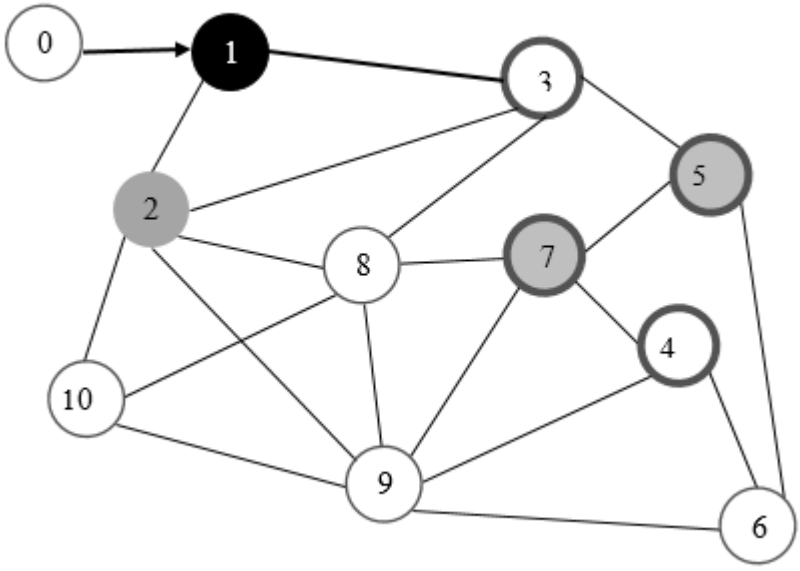


Рис. 2.3. Приклад розподіленої системи обробки інформації з десяти сервісів

Розподілена система обробки представляє собою набір підсистем (сервісів), які забезпечують підготовку, обробку та передачу даних:

$$DS \in \{S_1, S_2, ..., S_n\}, \tag{2.1}$$

де DS – розподілена система обробки інформації, S_i – i сервіс, який входить в розподілену систему, n – кількість сервісів в розподіленій системі обробки інформації.

Кожний сервіс може містити певний набір функцій (параметрів), які забезпечують роботу даного вузла розподіленої системи:

$$S_i \in \{s_i^1, s_i^2, ..., s_i^k\}, \tag{2.2}$$

де s_i^j – j параметр i сервісу, k – кількість параметрів в i сервісу.

Перелік можливих параметрів підсистеми S_i:

1. Персональна конфігурація;
2. Алгоритм авторизації;

3. База даних;
4. Вхідні контролери для доступу зовні;
5. Вихідні контролери для доступу до інших сервісів;
6. Алгоритми шифрування даних для сервісу та для бази даних;
7. Множина станів, у яких може перебувати сервіс;
8. Моделі вигляду, для представлення свого функціоналу користувачу.

Кожна підсистема комунікує з іншими підсистемами через канали зв'язку:

$$C \in \{C_1, C_2, ..., C_z\}, \tag{2.3}$$

де C – множина каналів зв'язку розподіленої системи обробки інформації, C_i – i канал зв'язку між сервісами, що входять в розподілену систему, z – кількість каналів зв'язку в розподіленій системі обробки інформації.

Зв'язки C_i реалізовані у вигляді основних REST методів для комунікації між підсистемами: GET, POST, PUT, DELETE, тощо.

Зв'язок представляє собою запит R від сервісу i до сервісу j та містить певний набір параметрів:

$$C_{ij} \in \{c_{ij}^1, c_{ij}^2, ..., c_{ij}^m\}, \tag{2.4}$$

де c_{ij}^o – o параметр каналу зв'язку між сервісом i та сервісом j, m – кількість параметрів в каналі зв'язку між сервісом i та сервісом j.

Перелік можливих параметрів:

1. Протокол передачі;
2. Параметри авторизації;
3. Заголовки запиту (Headers);
4. Тип запиту: GET, POST, PUT, DELETE і т.д.;
5. Адреса кінцевого сервісу та його порт;
6. Кінцева точку входу, або назва контролера, на який буде йти запит;

7. Тіло запиту;

8. Формат передачі даних.

Причому максимальна кількість зв'язків C_i може бути вирахувана з твердження, що граф повнозв'язний:

$$C_{\max} = \frac{n*(\mathrm{n}-1)}{2}, \qquad (2.5.)$$

де n – кількість сервісів в розподіленій системі обробки інформації.

Для повного опису зв'язків між сервісами будується таблиця 2.1 залежності.

Таблиця 2.1.

Таблиця залежності сервісів розподіленої системи

Сервіси	S_1	S_2	…	S_n
S_1	-	$C_{12} \in \{c_{12}^1, c_{12}^2, ..., c_{12}^m\}$	-	$C_{1n} \in \{c_{1n}^1, c_{1n}^2, ..., c_{1n}^m\}$
S_2	$C_{21} \in \{c_{21}^1, c_{21}^2, ..., c_{21}^m\}$	-	-	-
…	-	-	-	-
S_n	$C_{n1} \in \{c_{n1}^1, c_{n1}^2, ..., c_{n1}^m\}$	-	-	-

Як видно з таблиці 2.1, зв'язки існують між сервісами, але не у вигляді повнозв'язного графу, тому кількість зв'язків та їх параметри залежить від логіки побудови розподіленої системи.

Для забезпечення безперебійної роботи всіх сервісів потрібно, щоб розподілена комп'ютерна система, в якій розміщується розподілена система обробки інформації, відповідала всім вимогам та забезпечувала безперервну передачу з заданою швидкістю пакетів із запитами.

Для тестування роботи всієї розподіленої системи потрібно розробити методику тестування та провести розрахунок вибору методу тестування згідно

модифікованої піраміди Майка Кона та розробити методи для проведення верхніх рівнів тестування.

2.3. Модель тестування системи розподіленої обробки даних

Для вибору методу тестування потрібно визначити спосіб тестування сервісів розподіленої системи обробки інформації, для цього виберемо схему, коли тестування проводиться на цілій системі, що складається з двох сервісів. Перша схема тестування буде відповідати системі, в якій запит від тестувальника буде проходити через два сервіси одночасно, а друга схема – коли тестувальник буде тестувати два сервіси окремо один від одного. При другій схемі будемо вважати, що тестувальнику відомо всі параметри сервісів, які потрібно протестувати.

Розглянемо схему №1 про тестування 2 сервісів, які працюють послідовно, та будемо тестувати послідовно (рис. 2.4).

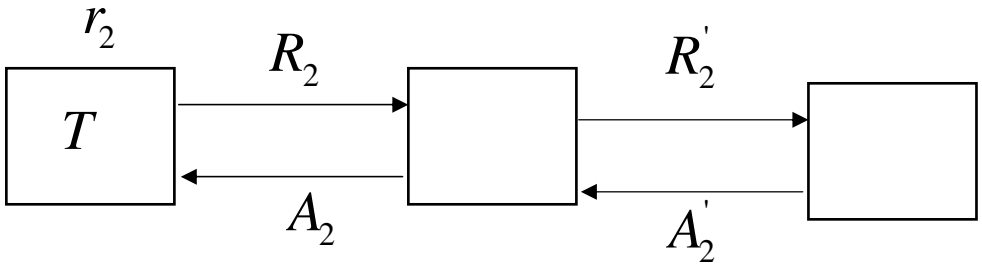


Рис. 2.4. Схема №1 тестування двох сервісів одночасно

В даній схемі тестувальник:

1) відправляє тест R_2 через сервіс S_1 на сервіс S_2;

2) якщо сервіс S_1 справний, то він передає тест R_2 без спотворень;

3) відповідь A_2 на тест R_2 передається також через сервіс S_1, який в разі справності передає її без спотворень;

4) тестувальник приймає відповідь A_2 та видає результат r_2 за формулою:

$$r_2 = \begin{cases} 0, \text{ якщо } A_2 = A_{etal}, \\ 1, \text{ якщо } A_2 \neq A_{etal}, \end{cases} \quad (2.6)$$

де A_{etal} – еталонна відповідь для перевірки запиту до сервісу. Якщо результат $r_2 = 0$, то тестувальник вважає сервіс S_2 справним, і навпаки, якщо результат $r_2 = 1$, то тестувальник вважає сервіс S_2 не справним.

Розглянемо схему №2 про тестування 2 сервісів, які ми будемо тестувати паралельно, а яким чином вони зв'язані між собою, тестувальник знає тільки з точки зору того, які запити та відповіді відправляються (рис. 2.5).

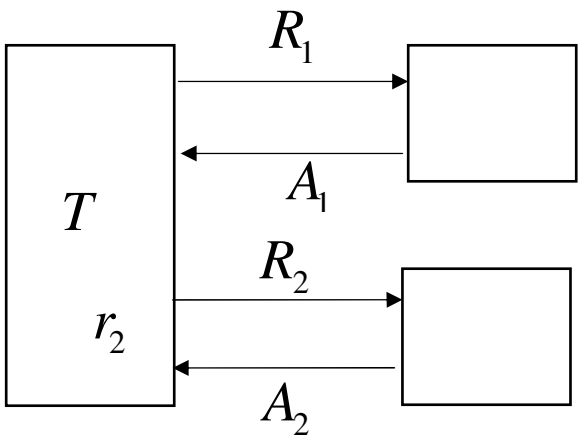


Рис. 2.5. Схема №2 тестування двох сервісів паралельно

В даній схемі тестувальник:

1) відправляє тест R_1 на сервіс S_1;

2) якщо сервіс S_1 справний, то він передає відповідь A_1 тестувальнику;

3) тестувальник приймає відповідь A_1 та видає результат r_1 за формулою:

$$r_1 = \begin{cases} 0, \text{ якщо } A_1 = A_{etal}, \\ 1, \text{ якщо } A_1 \neq A_{etal}, \end{cases} \quad (2.7)$$

де A_{etal} – еталонна відповідь для перевірки запиту до сервісу. Якщо результат $r_1 = 0$, то тестувальник вважає сервіс S_1 справним, і навпаки, якщо результат $r_1 = 1$, то тестувальник вважає сервіс S_1 не справним.

Аналогічним чином тестувальник проводить тестування другого сервісу схеми №2.

Для порівняння схеми №1 та схеми №2 будемо вираховувати P^* справного стану сервісу 2, де P^* – апостеріорна імовірність справного стану сервісу 2 за умови, що результат тестування $r_2 = 0$.

Характеристика P^* буде враховувати припущення:

1) справність сервісів 1,2;

2) надійність тестувальника;

3) імовірність правильного оцінювання відповіді на тест несправним тестувальником (якщо тестувальник несправний, то він видає результат перевірки навмання 0 або 1. Будемо вважати, що з імовірністю $P_r = 0,5$ видає 0 і з імовірністю $P_r = 0,5$ видає 1);

4) систему оцінювання:

$$r = \begin{cases} 1, \text{ якщо } \textit{тестувальник та перевіряємий модуль справні}; \\ 0, \textit{якщо тестувальник справний, а перевіряємий модуль несправний}; \\ 0 \vee 1, \textit{якщо тестувальник несправний}. \end{cases}$$

Інші припущення будуть описані нижче.

Тоді за схемою № 2 тестувальник T мають наступні показники найдійності:

1) апріорна до випробування імовірність справного стану тестувальника P_T, тоді ймовірність відмови вираховується як $q_T = 1 - P_T$;

2) апріорна імовірність P_2 справного сервісу 2, тоді ймовірність відмови вираховується як $q_2 = 1 - P_2$.

Можливі 4 гіпотези H_i справного і несправного станів тестувальника та сервісу 2:

1 – тестувальник написав правильний тест та відповідь з сервісу 2 співпала з еталонною відповіддю;

2 – тестувальник написав правильний тест, але відповідь не співпала з еталонною;

3 – тестувальник написав неправильний тест, але відповідь співпала з еталонною;

4 – тестувальник написав неправильний тест та відповідь не співпала з еталонною.

Для обчислення апостеріорної імовірності будемо використовувати теорему Байєса.

Тоді для кожної з гіпотез опишемо наступні твердження:

1 – $\mathrm{P}(H_1) = P_T * P_2$, $\mathrm{P}(\mathrm{A}/\, H_1) = 1$.

2 – $\mathrm{P}(H_2) = P_T * \mathrm{q}_2$, $\mathrm{P}(\mathrm{A}/\, H_2) = 0$.

3 – $\mathrm{P}(H_3) = q_T * P_2$, $\mathrm{P}(\mathrm{A}/\, H_3) = 0{,}5$.

4 – $\mathrm{P}(H_4) = q_T * q_2$, $\mathrm{P}(\mathrm{A}/\, H_4) = 0{,}5$.

В зв'язку з тим, що гіпотези складають повну групу подій, то сума їх імовірностей:

$$\sum_{i=1}^{4} P(H_i) = 1. \qquad (2.8)$$

В якості події A приймається подія отримання результату $r_2 = 0$, тому очевидно, що коли тестувальник та сервіс 2 справні, то результат $r_2 = 0$ буде отримано з імовірністю $\mathrm{P}(\mathrm{A}/\, H_1) = 1$. Якщо тестувальник справний, а сервіс 2 несправний, то результат $r_2 = 0$ буде отримано з ймовірністю $\mathrm{P}(\mathrm{A}/\, H_2) = 0$. В інших двох варіантах тестувальник несправний, тому він «угадує» результат з імовірністю $P_r = 0{,}5$, тому відповідно $\mathrm{P}(\mathrm{A}/\, H_3) = \mathrm{P}(\mathrm{A}/\, H_4) = 0{,}5$.

Повна ймовірність $\mathrm{P}(\mathrm{A})$ буде вираховуватися за формулою:

$$P(A)=\sum_{i=1}^{4}P(H_i)\cdot P(A/H_i)=P_TP_2+P_rq_TP_2+P_rq_Tq_2=P_TP_2+P_rq_T. \quad (2.9)$$

Тоді за теоремою Байєса апостеріорні імовірності гіпотез вираховуються:

$$\begin{aligned}
P(H_1/A)&=\frac{P(H_1)\cdot P(A/H_1)}{P(A)}=\frac{P_TP_2}{P_TP_2+q_TP_r},\\
P(H_2/A)&=\frac{P(H_2)\cdot P(A/H_2)}{P(A)}=\frac{P_Tq_2\cdot 0}{P_TP_2+q_TP_r}=0,\\
P(H_3/A)&=\frac{P(H_3)\cdot P(A/H_3)}{P(A)}=\frac{q_TP_2P_r}{P_TP_2+q_TP_r},\\
P(H_4/A)&=\frac{P(H_4)\cdot P(A/H_4)}{P(A)}=\frac{q_Tq_2P_r}{P_TP_2+q_TP_r}.
\end{aligned} \quad (2.10)$$

Звідси апостеріорна імовірність достовірності тестування сервісу 2 схеми 2 буде розраховуватися за формулою:

$$P_2^*=P(\mathrm{H}_1/\mathrm{A})+P(\mathrm{H}_3/\mathrm{A})=\frac{P_TP_2+q_TP_2P_r}{P_TP_2+q_TP_r} \quad (2.11)$$

Виведемо формулу для розрахунку апостеріорної імовірності для схеми № 1. В разі, якщо в схемі №1 буде тільки один вузол, то вона тестується по схемі №2.

Тоді за схемою № 1 тестувальник T має наступні показники найдійності:

1) апріорна до випробування імовірність справного стану тестувальника P_T, тоді ймовірність відмови вираховується як $q_T=1-P_T$;

2) апріорна імовірність P_1 справного сервісу 1, тоді ймовірність відмови вираховується як $q_1=1-P_1$;

3) апріорна імовірність P_2 справного сервісу 2, тоді ймовірність відмови вираховується як $q_2=1-P_2$.

Можливі 8 гіпотез H_i справного і несправного станів тестувальника та сервісу 2:

1 – тестувальник написав правильний тест, сервіс 1 працює справно та передав тест на сервіс 2. Сервіс 2 працює та він відповів сервісу 1, а той в свою чергу передав відповідь тестувальнику;

2 – тестувальник написав правильний тест, сервіс 1 працює справно та передав тест на сервіс 2. Сервіс 2 працює несправно та він відповів сервісу 1, а той в свою чергу передав несправну відповідь тестувальнику;

3 – тестувальник написав правильний тест, сервіс 1 працює несправно та передав тест на сервіс 2. Сервіс 2 працює справно та він відповів сервісу 1, а той в свою чергу передав відповідь тестувальнику;

4 – тестувальник написав правильний тест, сервіс 1 працює несправно та передав тест на сервіс 2. Сервіс 2 працює несправно та він відповів сервісу 1, а той в свою чергу передав відповідь тестувальнику;

5 – тестувальник написав неправильний тест, сервіс 1 працює справно та передав тест на сервіс 2. Сервіс 2 працює справно та він відповів сервісу 1, а той в свою чергу передав відповідь тестувальнику;

6 – тестувальник написав неправильний тест, сервіс 1 працює справно та передав тест на сервіс 2. Сервіс 2 працює несправно та він відповів сервісу 1, а той в свою чергу передав відповідь тестувальнику;

7 – тестувальник написав неправильний тест, сервіс 1 працює несправно та передав тест на сервіс 2. Сервіс 2 працює справно та він відповів сервісу 1, а той в свою чергу передав відповідь тестувальнику;

8 – тестувальник написав неправильний тест, сервіс 1 працює несправно та передав тест на сервіс 2. Сервіс 2 працює несправно та він відповів сервісу 1, а той в свою чергу передав відповідь тестувальнику.

Тоді для кожної з гіпотез опишемо наступні твердження:

1 – $\mathrm{P}(H_1) = P_T * P_1 * P_2$, $\mathrm{P}(\mathrm{A}/\, H_1) = 1$.

2 – $\mathrm{P}(H_2) = P_T * P_1 * \mathrm{q}_2$, $\mathrm{P}(\mathrm{A}/\, H_2) = 0$.

3 – $\mathrm{P}(H_3) = P_T * q_1 * P_2$, $\mathrm{P}(\mathrm{A}/\, H_3) = 0{,}5$.

4 – $\mathrm{P}(H_4) = P_T * q_1 * q_2$, $\mathrm{P}(\mathrm{A}/\ H_4) = 0{,}5$.

5 – $\mathrm{P}(H_5) = q_T * P_1 * P_2$, $\mathrm{P}(\mathrm{A}/\ H_5) = 0{,}5$.

6 – $\mathrm{P}(H_6) = q_T * P_1 * \mathrm{q}_2$, $\mathrm{P}(\mathrm{A}/\ H_6) = 0{,}5$.

7 – $\mathrm{P}(H_7) = q_T * q_1 * P_2$, $\mathrm{P}(\mathrm{A}/\ H_7) = 0{,}5$.

8 – $\mathrm{P}(H_8) = q_T * q_1 * q_2$, $\mathrm{P}(\mathrm{A}/\ H_8) = 0{,}5$.

В якості події A приймається подія отримання результату $r_2 = 0$, тому очевидно, що коли тестувальник, сервіс 1 та сервіс 2 справні, то результат $r_2 = 0$ буде отримано з імовірністю $\mathrm{P}(\mathrm{A}/\ H_1) = 1$. Якщо тестувальник справний, а сервіс 2 – несправний, то результат $r_2 = 0$ буде отримано з імовірністю $\mathrm{P}(\mathrm{A}/\ H_2) = 0$. В інших шести варіантах тестувальник несправний або несправний сервіс 1, тому він «угадує» результат з імовірністю $P_r = 0{,}5$, тому відповідно:

$$\mathrm{P}(\mathrm{A}/\ H_3) = \mathrm{P}(\mathrm{A}/\ H_4) = \mathrm{P}(\mathrm{A}/\ H_5) = \mathrm{P}(\mathrm{A}/\ H_6) = \mathrm{P}(\mathrm{A}/\ H_7) = \mathrm{P}(\mathrm{A}/\ H_8) = 0{,}5. \quad (2.12)$$

Повна ймовірність $\mathrm{P}(\mathrm{A})$ буде вираховуватися за формулою:

$$\mathrm{P}(\mathrm{A}) = \sum_{i=1}^{8} (\mathrm{P}(H_i)\,\mathrm{P}(\mathrm{A}/\ H_i)) =$$

$$P_T P_1 P_2 + P_r P_T q_1 P_2 + P_r P_T q_1 q_2 + P_r q_T P_1 P_2 + P_r q_T P_1 q_2 + P_r q_T q_1 P_2 + P_r q_T q_1 q_2 = \quad (2.13)$$

$$P_T P_1 P_2 + P_r P_T q_1 + P_r q_T P_1 + P_r q_T q_1$$

Тоді за теоремою Байєса апостеріорні імовірності гіпотез вираховуються за формулою:

$$P(H_1/A)=\frac{P(H_1)\cdot P(A/H_1)}{P(A)}=\frac{P_T P_1 P_2}{P_T P_1 P_2+P_r P_T q_1+P_r q_T P_1+P_r q_T q_1},$$

$$P(H_2/A)=\frac{P(H_2)\cdot P(A/H_2)}{P(A)}=\frac{P_T P_1 q_2\cdot 0}{P_T P_1 P_2+P_r P_T q_1+P_r q_T P_1+P_r q_T q_1}=0,$$

$$P(H_3/A)=\frac{P(H_3)\cdot P(A/H_3)}{P(A)}=\frac{P_r P_T q_1 P_2}{P_T P_1 P_2+P_r P_T q_1+P_r q_T P_1+P_r q_T q_1},$$

$$P(H_4/A)=\frac{P(H_4)\cdot P(A/H_4)}{P(A)}=\frac{P_r P_T q_1 q_2}{P_T P_1 P_2+P_r P_T q_1+P_r q_T P_1+P_r q_T q_1},$$

$$P(H_5/A)=\frac{P(H_5)\cdot P(A/H_5)}{P(A)}=\frac{P_r q_T P_1 P_2}{P_T P_1 P_2+P_r P_T q_1+P_r q_T P_1+P_r q_T q_1},$$

$$P(H_6/A)=\frac{P(H_6)\cdot P(A/H_6)}{P(A)}=\frac{P_r q_T P_1 q_2}{P_T P_1 P_2+P_r P_T q_1+P_r q_T P_1+P_r q_T q_1},$$

$$P(H_7/A)=\frac{P(H_7)\cdot P(A/H_7)}{P(A)}=\frac{P_r q_T q_1 P_2}{P_T P_1 P_2+P_r P_T q_1+P_r q_T P_1+P_r q_T q_1},\qquad (2.14)$$

$$P(H_8/A)=\frac{P(H_8)\cdot P(A/H_8)}{P(A)}=\frac{P_r q_T q_1 q_2}{P_T P_1 P_2+P_r P_T q_1+P_r q_T P_1+P_r q_T q_1}.$$

Звідси апостеріорна імовірність достовірності тестування сервісу 2 за схемою 1 буде розраховуватися за формулою:

$$P_1^* = P(\mathrm{H}_1/\mathrm{A})+P(\mathrm{H}_3/\mathrm{A})+P(\mathrm{H}_5/\mathrm{A})+P(\mathrm{H}_7/\mathrm{A})=$$
$$\frac{P_T P_1 P_2+P_r P_T q_1 P_2+P_r q_T P_1 P_2+P_r q_T q_1 P_2}{P_T P_1 P_2+P_r P_T q_1+P_r q_T P_1+P_r q_T q_1} \qquad (2.15)$$

Отже, за показником ефективності тестування, в якості якого прийнята апостеріорна імовірність справного стану сервісу 2, було запропоновано дві схеми для її розрахунків.

Виведемо формулу для розрахунку апостеріорної імовірності для схеми з n вузлів, які розміщені послідовно та тестується останній вузол (рис. 2.6.).

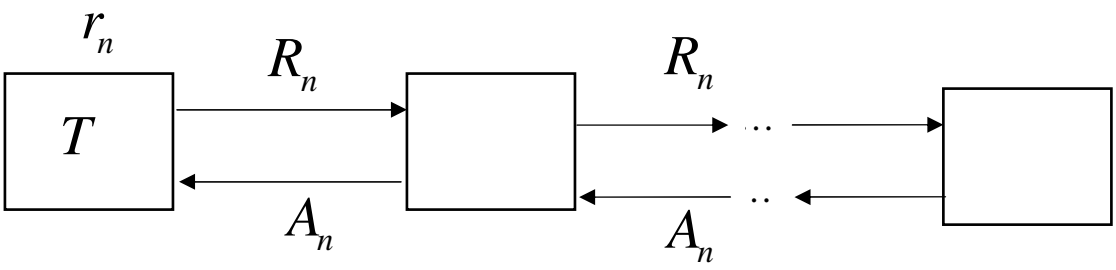


Рис. 2.6. Схема тестування n сервісу

Тоді за розширеною схемою тестувальник T має наступні показники найдійності:

1) апріорна до випробування імовірність справного стану тестувальника P_T, тоді ймовірність відмови вираховується як $q_T = 1 - P_T$;

2) апріорна імовірність P_1 справного сервісу 1, тоді ймовірність відмови вираховується як $q_1 = 1 - P_1$;

…

n) апріорна імовірність P_n справного сервісу n, тоді ймовірність відмови вираховується як $q_n = 1 - P_n$.

Можливі 2^n гіпотез H_i справного і несправного станів тестувальника та сервісу n:

1 – тестувальник написав правильний тест, сервіс 1 працює справно та передав тест на сервіс 2. … Сервіс n працює та він відповів сервісу $n-1$, а той в свою чергу передав відповідь через інші сервіси тестувальнику;

2 – тестувальник написав правильний тест, сервіс 1 працює справно та передав тест на сервіс 2. … Сервіс n працює несправно та він відповів сервісу $n-1$, а той в свою чергу передав відповідь через інші сервіси тестувальнику;

…

2^n – тестувальник написав неправильний тест, сервіс 1 працює несправно та передав тест на сервіс 2. … Сервіс n працює несправно та він відповів

сервісу $n-1$, а той в свою чергу передав відповідь через інші сервіси тестувальнику.

Тоді для кожної з гіпотез опишемо наступні твердження:

$1-\ \mathrm{P}(H_1) = P_T * P_1 * ... * P_n,\ \mathrm{P}(\mathrm{A}/\ H_1) = 1.$

$2-\ \mathrm{P}(H_2) = P_T * P_1 * ... * P_{n-1} * \mathrm{q}_n,\ \mathrm{P}(\mathrm{A}/\ H_2) = 0.$

…

$2^n-\ \mathrm{P}(H_{2^n}) = q_T * q_1 * ... * q_n,\ \mathrm{P}(\mathrm{A}/\ H_{2^n}) = 0{,}5.$

В якості події A приймається подія отримання результату $r_2 = 0$, тому очевидно, що коли тестувальник та сервіси справні, то результат $r_2 = 0$ буде отримано з імовірністю $\mathrm{P}(\mathrm{A}/\ H_1) = 1$. Якщо тестувальник справний, а сервіс n несправний, то результат $r_2 = 0$ буде отримано з ймовірністю $\mathrm{P}(\mathrm{A}/\ H_2) = 0$. В інших $2^n - 2$ варіантах тестувальник несправний або несправний сервіс n, тому він «угадує» результат з імовірністю $P_r = 0{,}5$, тому відповідно:

$$\mathrm{P}(\mathrm{A}/\ H_3) = \mathrm{P}(\mathrm{A}/\ H_4) = ... = \mathrm{P}(\mathrm{A}/\ H_{2^n}) = 0{,}5. \qquad (2.16)$$

Повна ймовірність $\mathrm{P}(\mathrm{A})$ буде вираховуватися за формулою:

$$\mathrm{P}(\mathrm{A}) = \sum_{i=1}^{2^n} (\mathrm{P}(H_i)\,\mathrm{P}(\mathrm{A}/\ H_i)). \qquad (2.17)$$

Тоді за теоремою Байєса апостеріорні імовірності гіпотез вираховуються за формулою:

$$P(H_1/A)=\frac{P(H_1)\cdot P(A/H_1)}{P(A)}=\frac{P_T P_1 ... P_n}{\sum_{i=1}^{2^n}(\mathrm{P}(H_i)\mathrm{P}(\mathrm{A}/H_i))},$$

$$P(H_2/A)=\frac{P(H_2)\cdot P(A/H_2)}{P(A)}=\frac{P_T P_1 ... q_n \cdot 0}{\sum_{i=1}^{2^n}(\mathrm{P}(H_i)\mathrm{P}(\mathrm{A}/H_i))}=0,$$

$$P(H_3/A)=\frac{P(H_3)\cdot P(A/H_3)}{P(A)}=\frac{P_r P_T P_1 ... q_{n-1} P_n}{\sum_{i=1}^{2^n}(\mathrm{P}(H_i)\mathrm{P}(\mathrm{A}/H_i))}, \quad (2.18)$$

...

$$P(H_{2^n}/A)=\frac{P(H_{2^n})\cdot P(A/H_{2^n})}{P(A)}=\frac{P_r q_T q_1 ... q_n}{\sum_{i=1}^{2^n}(\mathrm{P}(H_i)\mathrm{P}(\mathrm{A}/H_i))}.$$

Звідси апостеріорна імовірність достовірності тестування сервісу n за схемою з n сервісами буде розраховуватися за формулою:

$$P^* = P(\mathrm{H}_1/\mathrm{A}) + P(\mathrm{H}_3/\mathrm{A}) + ... + P(\mathrm{H}_{2^n-3}/\mathrm{A}) + P(\mathrm{H}_{2^n-1}/\mathrm{A}) = \sum_{i=1}^{2^n-1} P(\mathrm{H}_i/\mathrm{A}) \quad (2.19)$$

Отже, за показником ефективності тестування, в якості якого прийнята апостеріорна імовірність справного стану сервісу n, було запропоновано формули для розрахунку апостеріорної імовірності та розроблена модель тестування програмного забезпечення.

2.4 Порівняльні аналітичні оцінки моделі тестування програмного забезпечення розподілених систем

Проведемо аналітичну оцінку запропонованих розрахунків в пункті 2.3 для системи тестування з одним сервісом. Для знаходження апостеріорної імовірності дано:

1) тестувальник відправляє тест R_1 на сервіс S_1 та отримує відповідь A_1 від сервісу;

2) тестувальник приймає відповідь A_1 та видає результат r_1 за формулою:

$$r_1 = \begin{cases} 0, \text{ якщо } A_1 = A_{etal}, \\ 1, \text{ якщо } A_1 \neq A_{etal}, \end{cases}$$

де A_{etal} – еталонна відповідь для перевірки запиту до сервісу. Якщо результат $r_2 = 0$, то тестувальник вважає сервіс S_1 справним, і навпаки, якщо результат $r_2 = 1$, то тестувальник вважає сервіс S_1 несправним;

3) апріорна (до випробування) імовірність справного стану тестувальника $P_T = 0{,}95$, імовірність відмови $q_T = 1 - P_T = 0{,}05$;

4) апріорна імовірність справного стану сервісу S_1: $P_1 = 0{,}95$ та відповідно $q_1 = 1 - P_1 = 0{,}05$.

Можливі 4 гіпотези справного і несправного станів тестувальника та сервісу S_1 представлено в вигляді:

1 – $P(H_1) = P_T P_1 = 0{,}95 * 0{,}95 = 0{,}9025$;

2 – $P(H_2) = P_T q_1 = 0{,}95 * 0{,}05 = 0{,}0475$;

3 – $P(H_3) = q_T P_1 = 0{,}05 * 0{,}95 = 0{,}0475$;

4 – $P(H_4) = q_T q_1 = 0{,}05 * 0{,}05 = 0{,}0025$.

Імовірність $P(A/H_i)$ вираховується для різних гіпотез по різному, а саме:

1 – $P(A/H_1) = 1$, тому що і тестувальник написав правильний запит і сервіс дав правильну відповідь;

2 – $P(A/H_2) = 0$, тому що тестувальник написав правильний запит, але сервіс дав неправильну відповідь;

3 – $P(A/H_3) = 0{,}5$, тому що тестувальник написав неправильний запит, але сервіс працює справно;

4 – $P(A/H_4) = 0{,}5$, тому що тестувальник написав неправильний запит, але сервіс дав неправильну відповідь.

Звідси повна імовірність розраховується як:

$$P(\mathrm{A}) = \sum P(\mathrm{H}_i) * P(\mathrm{A}/\mathrm{H}_i) = 0,9025 * 1 + 0,0475 * 0,5 + 0,0025 * 0,5 = 0,9275$$

За теоремою Байєса апостеріорні імовірності гіпотез:

$$P(\mathrm{H}_1/\mathrm{A}) = \frac{P(\mathrm{H}_1) \cdot P(\mathrm{A}/\mathrm{H}_1)}{P(\mathrm{A})} = 0,973,$$

$$P(\mathrm{H}_2/\mathrm{A}) = \frac{P(\mathrm{H}_2) \cdot P(\mathrm{A}/\mathrm{H}_2)}{P(\mathrm{A})} = 0,$$

$$P(\mathrm{H}_3/\mathrm{A}) = \frac{P(\mathrm{H}_3) \cdot P(\mathrm{A}/\mathrm{H}_3)}{P(\mathrm{A})} = 0,0256,$$

$$P(\mathrm{H}_4/\mathrm{A}) = \frac{P(\mathrm{H}_4) \cdot P(\mathrm{A}/\mathrm{H}_4)}{P(\mathrm{A})} = 0,0013.$$

Апостеріорна імовірність справного стану сервісу:

$$P_1^* = P(\mathrm{H}_1/\mathrm{A}) + P(\mathrm{H}_3/\mathrm{A}) = 0,973 + 0,0256 = 0,9986$$

Таким чином, обчислено P_1^* для схеми з одним сервісом.

Проведемо аналітичну оцінку запропонованих розрахунків в пункті 2.2.2 для системи тестування з двома сервісами. Для знаходження апостеріорної імовірності дано:

1) тестувальник відправляє тест R_2 на сервіс S_1, який пересилає його на сервіс S_2;

2) сервіс S_2 обробляє тест та відправляє відповідь A_2 через сервіс S_1 тестувальнику;

3) тестувальник приймає відповідь A_2 та видає результат r_2 за формулою:

$$r_2 = \begin{cases} 0, \text{ якщо } A_2 = A_{etal}, \\ 1, \text{ якщо } A_2 \neq A_{etal}. \end{cases}$$

де A_{etal} – еталонна відповідь для перевірки запиту до сервісу. Якщо результат $r_2 = 0$, то тестувальник вважає сервіс S_2 справним, і навпаки, якщо результат $r_2 = 1$, то тестувальник вважає сервіс S_2 несправним;

3) апріорна (до випробування) імовірність справного стану тестувальника $P_T = 0,95$, імовірність відмови $q_T = 1 - P_T = 0,05$;

4) апріорна імовірність справного стану сервісів S_1 та S_2: $P_1 = 0,95$ та відповідно $q_1 = 1 - P_1 = 0,05$, $P_2 = 0,95$ та відповідно $q_2 = 1 - P_2 = 0,05$.

Можливі 8 гіпотез справного і несправного станів тестувальника та сервісів S_1 та S_2 представлено у вигляді:

1 – $P(H_1) = P_T P_1 P_2 = 0,95 * 0,95 * 0,95 = 0,8573$;

2 – $P(H_2) = P_T P_1 q_2 = 0,95 * 0,95 * 0,05 = 0,0451$;

3 – $P(H_3) = P_T q_1 P_2 = 0,95 * 0,05 * 0,95 = 0,0451$;

4 – $P(H_4) = P_T q_1 q_2 = 0,95 * 0,05 * 0,05 = 0,0024$;

5 – $P(H_5) = q_T P_1 P_2 = 0,05 * 0,95 * 0,95 = 0,0451$;

6 – $P(H_6) = q_T P_1 q_2 = 0,05 * 0,95 * 0,05 = 0,0024$;

7 – $P(H_7) = q_T q_1 P_2 = 0,05 * 0,05 * 0,95 = 0,0024$;

8 – $P(H_8) = q_T q_1 q_2 = 0,05 * 0,05 * 0,05 = 0,0001$.

Імовірність $P(A/H_i)$ вираховується для різних гіпотез по різному, а саме:

1 – $P(A/H_1) = 1$, тому що і тестувальник написав правильний запит і сервіси працюють правильно;

2 – $P(A/H_2) = 0$, тому що тестувальник написав правильний запит, перший сервіс працює правильно, але другий сервіс дав неправильну відповідь;

3 – $P(A/H_3) = 0,5$, тому що тестувальник написав правильний запит, перший сервіс працює неправильно, а другий сервіс дав правильну відповідь;

4 – $P(A/H_4) = 0,5$, тому що тестувальник написав правильний запит, перший сервіс працює неправильно та другий сервіс дав неправильну відповідь;

5 – $P(A/H_5) = 0,5$, тому що тестувальник написав неправильний запит, перший сервіс працює правильно та другий сервіс дав правильну відповідь;

6 – $P(A/H_6) = 0,5$, тому що тестувальник написав неправильний запит, перший сервіс працює правильно, але другий сервіс дав неправильну відповідь;

7 – $P(\text{A}/\text{H}_7)=0,5$, тому що тестувальник написав неправильний запит, перший сервіс працює неправильно, але другий сервіс дав правильну відповідь;

8 – $P(\text{A}/\text{H}_8)=0,5$, тому що тестувальник написав неправильний запит, перший сервіс працює неправильно та другий сервіс дав неправильну відповідь.

Звідси повна імовірність розраховується як:

$$P(\text{A})=\sum P(\text{H}_i)*P(\text{A}/\text{H}_i)=0,9061$$

За теоремою Байєса апостеріорні імовірності гіпотез:

$$P(\text{H}_1/\text{A})=\frac{P(\text{H}_1)\cdot P(\text{A}/\text{H}_1)}{P(\text{A})}=0,9462,$$

$$P(\text{H}_2/\text{A})=\frac{P(\text{H}_2)\cdot P(\text{A}/\text{H}_2)}{P(\text{A})}=0,$$

$$P(\text{H}_3/\text{A})=\frac{P(\text{H}_3)\cdot P(\text{A}/\text{H}_3)}{P(\text{A})}=0,0249,$$

$$P(\text{H}_4/\text{A})=\frac{P(\text{H}_4)\cdot P(\text{A}/\text{H}_4)}{P(\text{A})}=0,0013,$$

$$P(\text{H}_5/\text{A})=\frac{P(\text{H}_5)\cdot P(\text{A}/\text{H}_5)}{P(\text{A})}=0,0249,$$

$$P(\text{H}_6/\text{A})=\frac{P(\text{H}_6)\cdot P(\text{A}/\text{H}_6)}{P(\text{A})}=0,0013,$$

$$P(\text{H}_7/\text{A})=\frac{P(\text{H}_7)\cdot P(\text{A}/\text{H}_7)}{P(\text{A})}=0,0013,$$

$$P(\text{H}_8/\text{A})=\frac{P(\text{H}_8)\cdot P(\text{A}/\text{H}_8)}{P(\text{A})}=0,00006.$$

Апостеріорна імовірність справного стану сервісу:

$$P_2^*=P(\text{H}_1/\text{A})+P(\text{H}_3/\text{A})+P(\text{H}_5/\text{A})+P(\text{H}_7/\text{A})=0,9973$$

Таким чином, обчислено P_2^* для схеми з двома сервісами.

Також проведемо аналітичну оцінку запропонованих розрахунків в пункті 2.2 для системи тестування з трьома сервісами.

Візьмемо, що:

1) апріорна (до випробування) імовірність справного стану тестувальника $P_T=0,95$, імовірність відмови $q_T=1-P_T=0,05$;

2) апріорна імовірність справного стану сервісів S_1, S_2 та S_3: $P_1 = 0,95$ та відповідно $q_1 = 1 - P_1 = 0,05$, $P_2 = 0,95$ та відповідно $q_2 = 1 - P_2 = 0,05$, $P_3 = 0,95$ та відповідно $q_3 = 1 - P_3 = 0,05$.

Дані для розрахунку знаходяться в таблиці 2.2.

Таблиця 2.2.

Значення розрахунку апостеріорної імовірності для трьох послідовних вузлів

№	$P(H_i)$	$P(A/H_i)$	$P(H_i/A)$
1	0,81450625	1	0,919495382
2	0,04286875	0	0
3	0,04286875	0,5	0,024197247
4	0,00225625	0,5	0,001273539
5	0,04286875	0,5	0,024197247
6	0,00225625	0,5	0,001273539
7	0,00225625	0,5	0,001273539
8	0,00011875	0,5	6,70284E-05
9	0,04286875	0,5	0,024197247
10	0,00225625	0,5	0,001273539
11	0,00225625	0,5	0,001273539
12	0,00011875	0,5	6,70284E-05
13	0,00225625	0,5	0,001273539
14	0,00011875	0,5	6,70284E-05
15	0,00011875	0,5	6,70284E-05
16	6,25E-06	0,5	3,52781E-06

Звідси повна імовірність розраховується як:

$$P(A) = \sum P(H_i) * P(A/H_i) = 0,8858.$$

Апостеріорна імовірність справного стану сервісу:

$$P_3^* = P(\mathrm{H}_1/\mathrm{A}) + P(\mathrm{H}_3/\mathrm{A}) + P(\mathrm{H}_5/\mathrm{A}) + P(\mathrm{H}_7/\mathrm{A}) +$$
$$P(\mathrm{H}_9/\mathrm{A}) + P(\mathrm{H}_{11}/\mathrm{A}) + P(\mathrm{H}_{13}/\mathrm{A}) + P(\mathrm{H}_{15}/\mathrm{A}) = 0,9959.$$

Таким чином, обчислено P_3^* для схеми з трьома сервісами.

Також проведемо аналітичну оцінку запропонованих розрахунків в пункті 2.2.2 для системи тестування з чотирма сервісами.

Візьмемо, що:

1) апріорна (до випробування) імовірність справного стану тестувальника $P_T = 0,95$, імовірність відмови $q_T = 1 - P_T = 0,05$;

2) апріорна імовірність справного стану сервісів S_1, S_2, S_3 та S_4: $P_1 = 0,95$ та відповідно $q_1 = 1 - P_1 = 0,05$, $P_2 = 0,95$ та відповідно $q_2 = 1 - P_2 = 0,05$, $P_3 = 0,95$ та відповідно $q_3 = 1 - P_3 = 0,05$, $P_4 = 0,95$ та відповідно $q_4 = 1 - P_4 = 0,05$.

Дані для розрахунку знаходяться в таблиці 2.3.

Таблиця 2.3.

Значення для розрахунку апостеріорної імовірності для трьох послідовних вузлів

№	$P(\mathrm{H}_i)$	$P(\mathrm{A}/\mathrm{H}_i)$	$P(\mathrm{H}_i/\mathrm{A})$
1	0,77378094	1	0,89296723
2	0,04072531	0	0
3	0,04072531	0,5	0,02349914
4	0,00214344	0,5	0,0012368
5	0,04072531	0,5	0,02349914
6	0,00214344	0,5	0,0012368
7	0,04072531	0,5	0,02349914
8	0,00214344	0,5	0,0012368
9	0,00214344	0,5	0,0012368
10	0,00011281	0,5	6,5095E-05
11	0,00214344	0,5	0,0012368
12	0,00011281	0,5	6,5095E-05

13	0,00214344	0,5	0,0012368
14	0,00011281	0,5	6,5095E-05
15	0,00011281	0,5	6,5095E-05
16	5,9375E-06	0,5	3,426E-06
17	0,04072531	0,5	0,02349914
18	0,00214344	0,5	0,0012368
19	0,00214344	0,5	0,0012368
20	0,00011281	0,5	6,5095E-05
21	0,00214344	0,5	0,0012368
22	0,00011281	0,5	6,5095E-05
23	0,00011281	0,5	6,5095E-05
24	0,00011281	0,5	6,5095E-05
25	0,00011281	0,5	6,5095E-05
26	5,9375E-06	0,5	3,426E-06
27	0,00011281	0,5	6,5095E-05
28	5,9375E-06	0,5	3,426E-06
29	0,00214344	0,5	0,0012368
30	5,9375E-06	0,5	3,426E-06
31	5,9375E-06	0,5	3,426E-06
32	3,125E-07	0,5	1,8032E-07

Звідси повна імовірність розраховується як:

$$P(\mathrm{A}) = \sum P(\mathrm{H}_i) * P(\mathrm{A}/\mathrm{H}_i) = 0{,}8665 .$$

Апостеріорна імовірність справного стану сервісу:

$$P_4^* = P(\mathrm{H}_1/\mathrm{A}) + P(\mathrm{H}_3/\mathrm{A}) + P(\mathrm{H}_5/\mathrm{A}) + P(\mathrm{H}_7/\mathrm{A}) +$$
$$P(\mathrm{H}_9/\mathrm{A}) + P(\mathrm{H}_{11}/\mathrm{A}) + P(\mathrm{H}_{13}/\mathrm{A}) + P(\mathrm{H}_{15}/\mathrm{A}) +$$
$$P(\mathrm{H}_{17}/\mathrm{A}) + P(\mathrm{H}_{19}/\mathrm{A}) + P(\mathrm{H}_{21}/\mathrm{A}) + P(\mathrm{H}_{23}/\mathrm{A}) +$$
$$P(\mathrm{H}_{25}/\mathrm{A}) + P(\mathrm{H}_{27}/\mathrm{A}) + P(\mathrm{H}_{29}/\mathrm{A}) + P(\mathrm{H}_{31}/\mathrm{A}) = 0{,}9946$$

Таким чином, обчислено P_4^* для схеми з чотирма сервісами.

Порівнюючи схеми з різною кількістю сервісів за показником ефективності тестування, в якості якого прийнята апостеріорна імовірність справного стану за умови отримання позитивного результату тестування, схема 1 з роздільним тестуванням одного сервісу дає найвищу апостеріорну імовірність, а починаючи з схеми 2, в якій реалізовано ланцюгове тестування, апостеріорна ймовірність зменшується зі збільшенням кількості вузлів. Для роздільного тестування $P_1^* = 0,9987$, а для ланцюгового тестування $P_2^* = 0,9973$, $P_3^* = 0,9959$, $P_4^* = 0,9946$.

Для порівняння проведемо аналітичні розрахунки для різних початкових величин апріорної імовірності. Нехай апріорні імовірності будуть рівні:

1. Апріорна (до випробування) імовірність справного стану тестувальника $P_T = 0,9$, імовірність відмови $q_T = 1 - P_T = 0,1$, апріорна імовірність справного стану сервісів: $P_i = 0,9$ та відповідно $q_i = 1 - P_i = 0,1$.

2. Апріорна (до випробування) імовірність справного стану тестувальника $P_T = 0,85$, імовірність відмови $q_T = 1 - P_T = 0,15$, апріорна імовірність справного стану сервісів: $P_i = 0,85$ та відповідно $q_i = 1 - P_i = 0,15$.

3. Апріорна (до випробування) імовірність справного стану тестувальника $P_T = 0,8$, імовірність відмови $q_T = 1 - P_T = 0,2$, апріорна імовірність справного стану сервісів: $P_i = 0,8$ та відповідно $q_i = 1 - P_i = 0,2$.

Після цього проведемо розрахунок апостеріорних імовірностей для цих всіх варіантів, визначених апріорною імовірністю. Дані занесемо в таблицю та побудуємо відповідний графік (рис. 2.7).

Таблиця 2.4.

Визначення апостеріорних імовірностей при відомих значеннях апріорних

Кількість вузлів	Апріорна імовірність $P_i = 0,95$	Апріорна імовірність $P_i = 0,9$	Апріорна імовірність $P_i = 0,85$	Апріорна імовірність $P_i = 0,8$
1	0.9986522911	0.9218328841	0.8477088949	0.7762803235
2	0.9973099738	0.8988826045	0.8079045386	0.7239619258
3	0.9959747691	0.8783399539	0.7744283184	0.6827581828
4	0.9946483613	0.8600358687	0.7464880491	0.6506888664

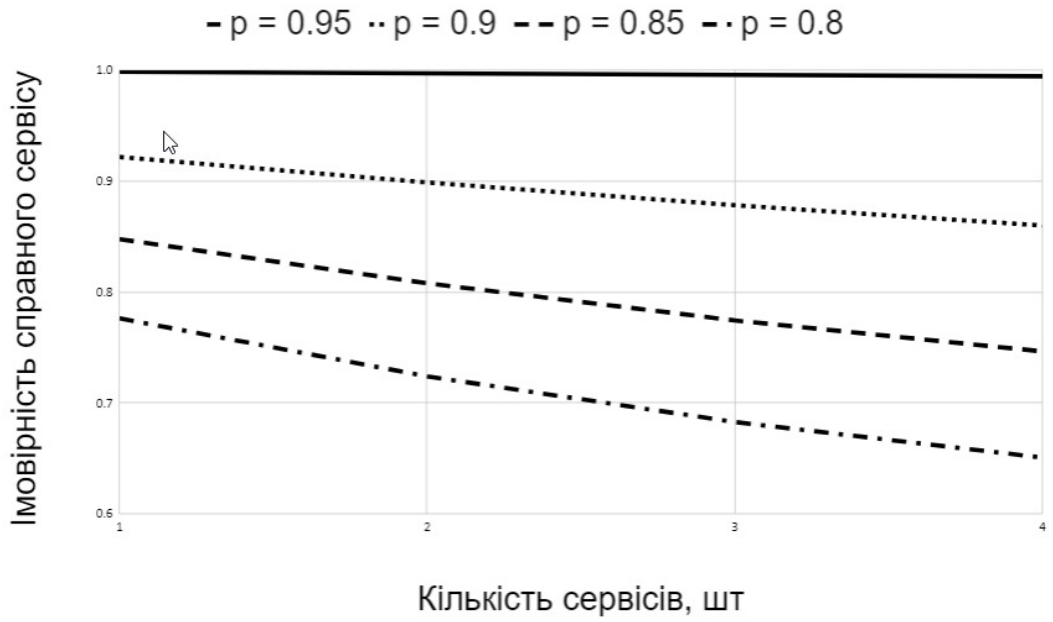


Рис. 2.7. Визначення апостеріорних імовірностей справного стану сервісу

Аналітичні оцінки застосування експерименту для чотирьох різних апріорних імовірностей показали, що зі зменшенням апріорної імовірності графік залежностей спадає з меншим відхиленням.

2.5 Висновки до розділу 2

Модифіковано піраміду тестування Майка Кона для адаптації при тестуванні в розподілених систем обробки інформації, що дозволило розширити можливості тестування і застосувати особливості розподілених систем. Розроблено рекомендації щодо подальшого застосування механізмів модифікованої піраміди Майка Кона.

Визначено вимоги для забезпечення безперебійної роботи всіх сервісів розподіленої комп’ютерної системи, в якій розміщується розподілена система обробки інформації та забезпечення безперервної передачі пакетів запитів із заданою швидкістю.

Розроблена модель тестування програмного забезпечення розподілених систем на основі автономного розгортання програмних компонентів, яка базується на застосуванні механізмів на основі модифікованої піраміди Майка

Кона, що дозволяє підвищити ефективність тестування та зменшити кількість об’ємних тестів.

Виконано порівняння схем з різною кількістю сервісів за показником ефективності тестування, в якості якого прийнята апостеріорна імовірність справного стану за умови отримання позитивного результату тестування. Показано, що системи з роздільним тестуванням одного сервісу апостеріорна імовірність дає найвищий показник, а починаючи з системи тестування двох та більше сервісів, в якій реалізовано ланцюгове тестування, апостеріорна ймовірність зменшується з збільшенням кількості вузлів. Для роздільного тестування $P_1^* = 0,9987$, а для ланцюгового тестування $P_2^* = 0,9973$, $P_3^* = 0,9959$, $P_4^* = 0,9946$.

Результати, подані в розділі, автор опублікував у працях [6, 7, 8, 9, 10].

РОЗДІЛ 3
МЕТОДИ ТЕСТУВАННЯ СИСТЕМИ РОЗПОДІЛЕНОЇ ОБРОБКИ ІНФОРМАЦІЇ В АРХІТЕКТУРІ РОЗПОДІЛЕНИХ СИСТЕМ

3.1 Удосконалення методу тестування розподіленої системи з використанням контрактів

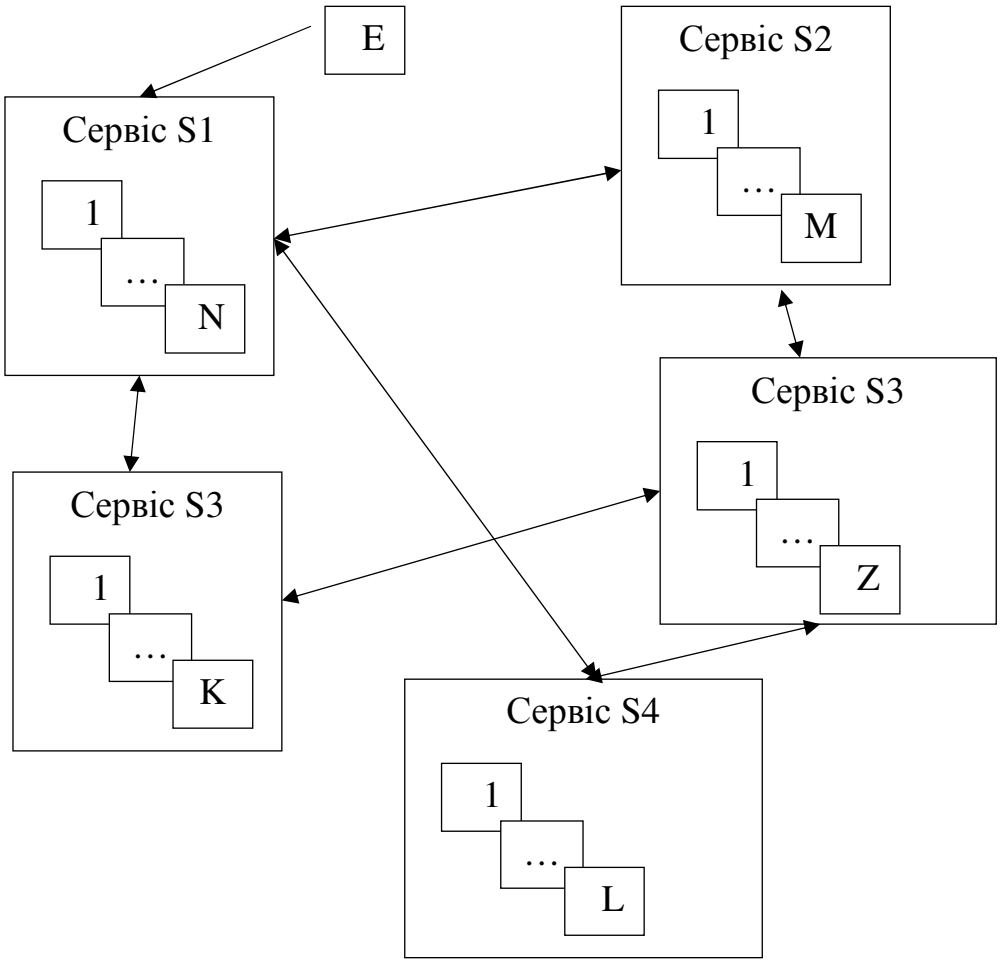
3.1.1 Механізм підміни функціоналу для тестування сервісів розподіленої системи

Перевагою архітектури розподілених систем обробки даних у порівняні з монолітними архітектурами є те, що кожна частина (сервіс) є незалежною одиницею, у якого є свій API та який може підключатися до інших API. Виникає потреба в розробці механізму тестування для сервісів розподіленої системи обробки даних, використовуючи метод підміни функціоналу. Даний механізм допоможе значно скоротити час на перевірку компонентів сервісів, оскільки не потрібно установлювати повністю всю систему, а достатньо установити один модуль інтерфейсу та протестувати його. Це також зменшить ресурси дискового простору та пам’яті, які потрібні на віртуальних машинах. Тому потрібно розробити механізм тестування сервісів розподіленої системи обробки даних на основі контрактів.

Опишемо детальніше реалізацію механізму контрактних тестів у системах розподіленої обробки даних. Щоб протестувати ту частину функціоналу, яка стосується окремої підсистеми, потрібно ізолювати функціонал та залежності, які стосуються інших підсистем. Для цього необхідно зімітувати всіх інших взаємодіючих співучасників процесу. Це можна зробити, якщо під час розгортання середовища для тестування будуть запущені своєрідні сервіси-імітатори, які потрібно налаштувати на відправку зворотних відповідей з ціллю імітації роботи справжнього сервісу. Наприклад, при запиті на сервіс-імітатор він може повертати заздалегідь відоме значення відповіді, при цьому перевіряючи правильність запиту, який надійшов з сервісу, що тестується. При

цьому на сервіс, що тестується, також відправляються запити і порівнюються відповіді з етелонними.

Візьмемо фрагмент розподіленої системи обробки даних, у якій є, крім звичайних сервісів, ще і сервіси, які відповідають за обмін інформацією з користувачами (рисунок 3.1).



Сервіси S_1, S_2, S_3, S_4, S_5 містять деяку кількість методів обробки інформації N, M, K, Z, L відповідно. E — точка входу в систему з зовнішнього середовища, через яку користувач має можливість почати роботу з системою.

Нехай час $tt(\mathrm{i})$ – час проходження тесту до сервісу S_1, де $i = 1,2,...,N$ кількість методів, а $t(\mathrm{i})$ – час розгортання сервісу S_1, $t(s)$ – час розгортання тестового середовища.

При тестування методів сервісів потрібно також врахувати, що етап розгортання тестового середовища буває дуже часозатратним у великих системах.

Відповідно, загальний час T, який буде затрачений на тестування усіх компонентів розподіленої системи, буде рівний сумі часу розгортання середовища, часу розгортання сервісів, часу завантаження методів сервісів, часу тестування сервісів, а також часу згортання всього середовища, тобто:

$$T = \sum_{p=1}^{5} \sum_{i=1}^{N} \mathrm{tt}_p(\mathrm{i}) + \sum_{i=1}^{N} \mathrm{t(i)} + \mathrm{t(s)}. \qquad (3.1)$$

де p – кількість сервісів.

Для удосконалення роботи традиційного методу тестування сервісів введемо поняття сервіс-імітатор (контракт) – це окремий сервіс, який має свою власну адресу, протокол та порт передачі даних, які користувач може налаштувати власноручно. Тоді процес тестування сервісом-імітатором розподіленої системи проходить у режимах споживача та постачальника. Режим споживача дозволяє обробити ті запити, які приходять з сервісу, а режим постачальника дозволяє відправляти заготовки запитів на сервіс та порівнювати відповідь з еталонним значенням. Ці два режими дозволяють модифікувати традиційне тестування в послідовній схемі на тестування в паралельній схемі.

Режим споживача (рис 3.2) приймає запит від сервісу за допомогою модулю «Обробка даних запит», який розділяє запит на компоненти та виділяє з нього параметри запиту. Параметри запиту передаються в модуль «Система пошуку», який проводить пошук в модулі «Контракти» такого ж очікуваного запиту по масиву предикатів. Модуль «Контракти» віддає в модуль «Система пошуку» всі значення по предикатам і знаходиться відповідний запит. Пошук відбувається по відповідності типу запиту, точці контролеру, заголовках та тіла запиту. Якщо прийняті дані співпадають, на модуль «Обробка результатів пошуку» посилається заготовка відповіді на запит. Відповідь зі знайденої пари

“предикат-відповідь” передається за допомогою модулю «Відповідь». Масив відправлених запитів та отриманих відповідей на сервісі-імітаторі записується у контракт, який являється документацією для тестування у режимі постачальника.

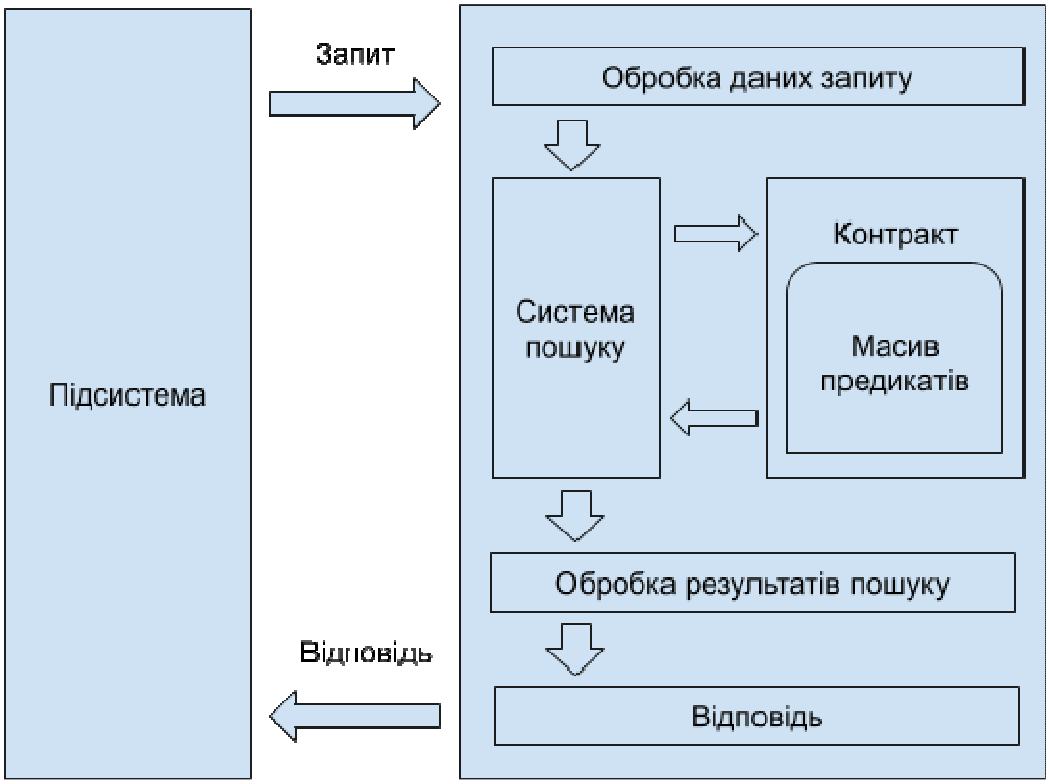


Рис 3.2. Схема роботи режиму споживача

Дана схема тестування дозволяє розгорнути окремий сервіс, в якому будуть визначені масив предикатів, по яких проводиться тестування підсистеми (сервісу). Якщо в масиві немає відповідних параметрів запиту, то тестувальник повинен додати ці запити в модуль «Контракти».

Режим постачальника (рис 3.3) працює за своїм алгоритмом роботи. Модуль «Вибір предикату з контракту» проводить вибір предикатів, по якому повинно відбуватися тестування підсистеми (сервісу) за допомогою модулю «Контракт». Далі модуль «Контракт» формує запит, який за допомогою модулю «Запит на сервіс» відправляє запит з контракту на сервіс, який повертає відповідь. Модуль «Отримання відповіді» приймає відповідь та за допомогою модуля «Порівняння відповіді» проводить порівняння відповіді по предикатам з

еталонною. Тест порівнює відповідь підсистеми зі значенням пари “предикат-відповідь” та відправляє тестувальнику через модуль «Результат» результат проходження тестування. Пошук відбувається по відповідності тіла та статус коду відповіді.

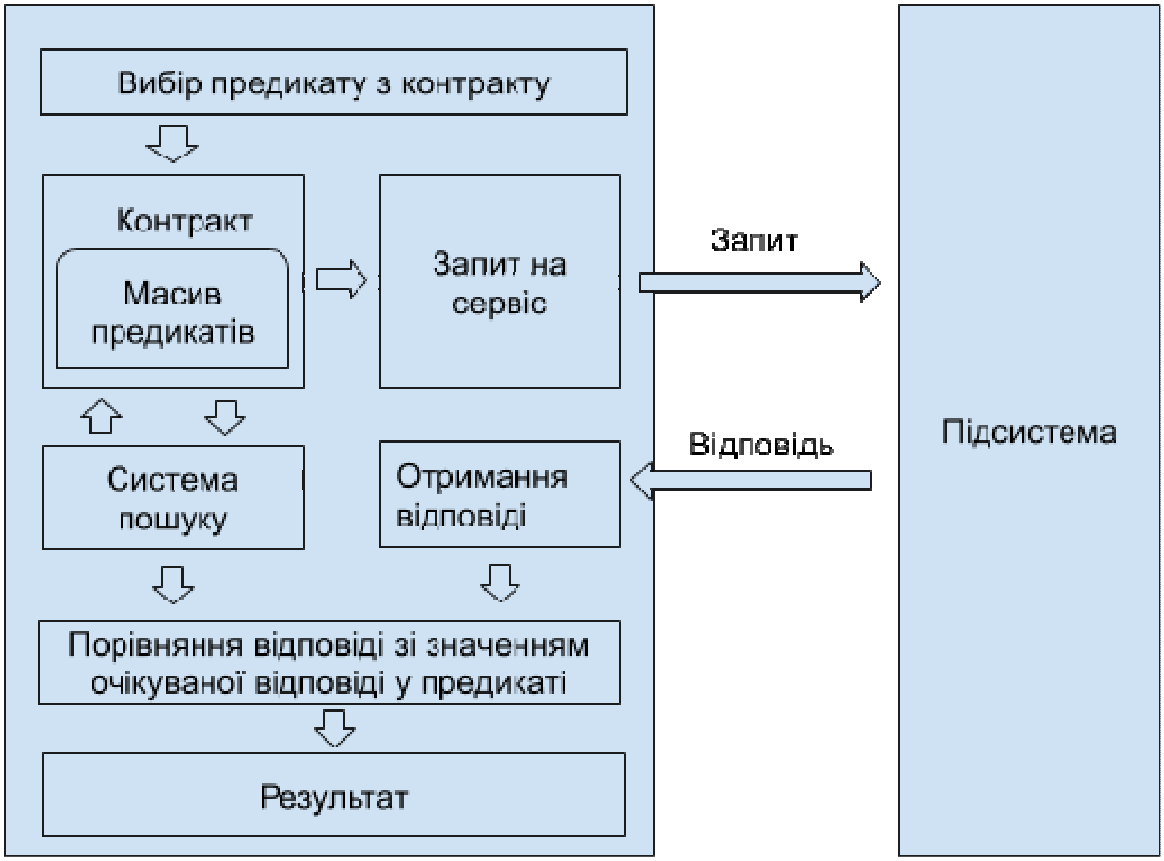


Рис 3.3. Схема роботи режиму постачальника

В результаті вищесказаного в режимі споживача система створює контракти на основі запитів, які проходили через сервіс-імітатор. В режимі постачальника ці контракти використовується як документація. Масив вхідних даних відправляється на підсистему та тести успішно проходять, якщо отриманий результат співпадає з очікуваним результатом у контракті.

Нехай час $tt(\text{i})$ – час проходження тесту до сервісу S_1, де $i = 1,2,...,N$ кількість методів, $to(\text{i})$ – час розгортання сервісу S_1, $tk(\text{i})$ – час розгортання контракту сервісу S_1, $t(s)$ – час розгортання тестового середовища.

При тестуванні сервісу за допомогою модифікованого механізму потрібно також врахувати ще важливий параметр: час розгортання контракту *tk* сервісу для тестування. Цей етап в порівнянні з класичним тестуванням займає набагато менше часу та місця на віртуальних машинах.

Відповідно, загальний час T, який буде затрачений на тестування усіх компонентів розподіленої системи, буде рівний сумі часу розгортання середовища, часу розгортання сервісу, часу завантаження методів сервісів, часу тестування сервісів, а також часу згортання сервісу, тобто:

$$T = \sum_{j=1}^{5}(\sum_{i=1}^{N}(t_j(\mathrm{i}) + to_j(\mathrm{i}) + tt_j(\mathrm{i})) + tk)\,. \qquad (3.2)$$

При одноразовому тестування розподіленої системи з невеликою кількості вузлів час тестування при класичному варіанті буде менший в зв'язку з тим, що сума часів розгортання та згортання кожного сервісу буде більшою за розгортання всієї системи. Але, якщо потрібно кілька разів тестувати всю систему, то тестування за допомогою контрактів є більш вигідним, тому що проводяться зміни тільки в кількох сервісах і другий раз уже не потрібно розгортати всю систему, а тільки окремі сервіси.

3.1.2 Удосконалений метод підміни функціоналу для тестування сервісів розподіленої системи

В порівнянні з наскрізним тестуванням, переваги підходу застосування контрактів для тестування сервісів розподілених систем очевидні. Час, затрачений на тестування будь-якого сервісу, буде значно менший, ніж при наскрізному тестування цього ж сервісу. Наприклад, розподіленій системі, яка була розглянута для тестування компонентів сервісу S_1, нам не потрібно розгортати всі п'ять сервісів та чекати на завантаження компонентів всіх сервісів.

Рис 3.4. Схема методу тестування сервісів розподіленої системи за допомогою наскрізних тестів

Також, для процесу тестування потрібно виділити значно менші ресурси на дисковий простір та оперативну пам'ять. Для початку тестування сервісів розподіленої системи наскрізними тестами необхідно розгорнути тестове середовище (рис. 3.4). Для цього на віртуальних машинах, де запускаються тести, запускають скрипти, які виконують установку пакетів для тестування сервісів розподіленої системи обробки інформації.

Далі проходить встановлення усіх компонентів розподіленої системи обробки інформації та всіх програмних пакетів, які необхідні для запуску цих компонентів. При встановленні системи вона тестується на наявність помилок та при виникненні помилки процес тестування негайно зупиняється та проводиться опис помилки. Якщо тестування встановлення розподіленої системи пройшло успішно, то механізм тестування оснований на відповідному фреймворку ініціалізує масив тестів та створює чергу їх запуску у порядку відповідно до режиму тестування цими тестами: послідовного або паралельного. Потім перед запуском кожного тесту виконується операція стосовно тестових передвимог та післявимог. По завершенню формування запускається тест, який перевіряє компоненти об’єкта, що тестуються. Якщо компоненти об’єкта містять помилки, то тест вважається непройденим, тестове середовище робить запис помилки, яка виникла, та процес переходить до наступного тесту. Під час кожної помилки наповнюється база знань помилок, яка потім аналізується розробниками. Після проходження останнього тесту тестове середовище згортається та завершує свою роботу. По завершенні проходження всіх тестів вся інформація про проходження успішних та неуспішних тестів з бази знань стає доступним користувачу та виводиться на засоби оцінки помилок, які виникли під час тестування.

Розглянемо схему роботи тестування сервісів розподіленої системи із застосуванням контрактних тестів (рис. 3.5, 3.6)

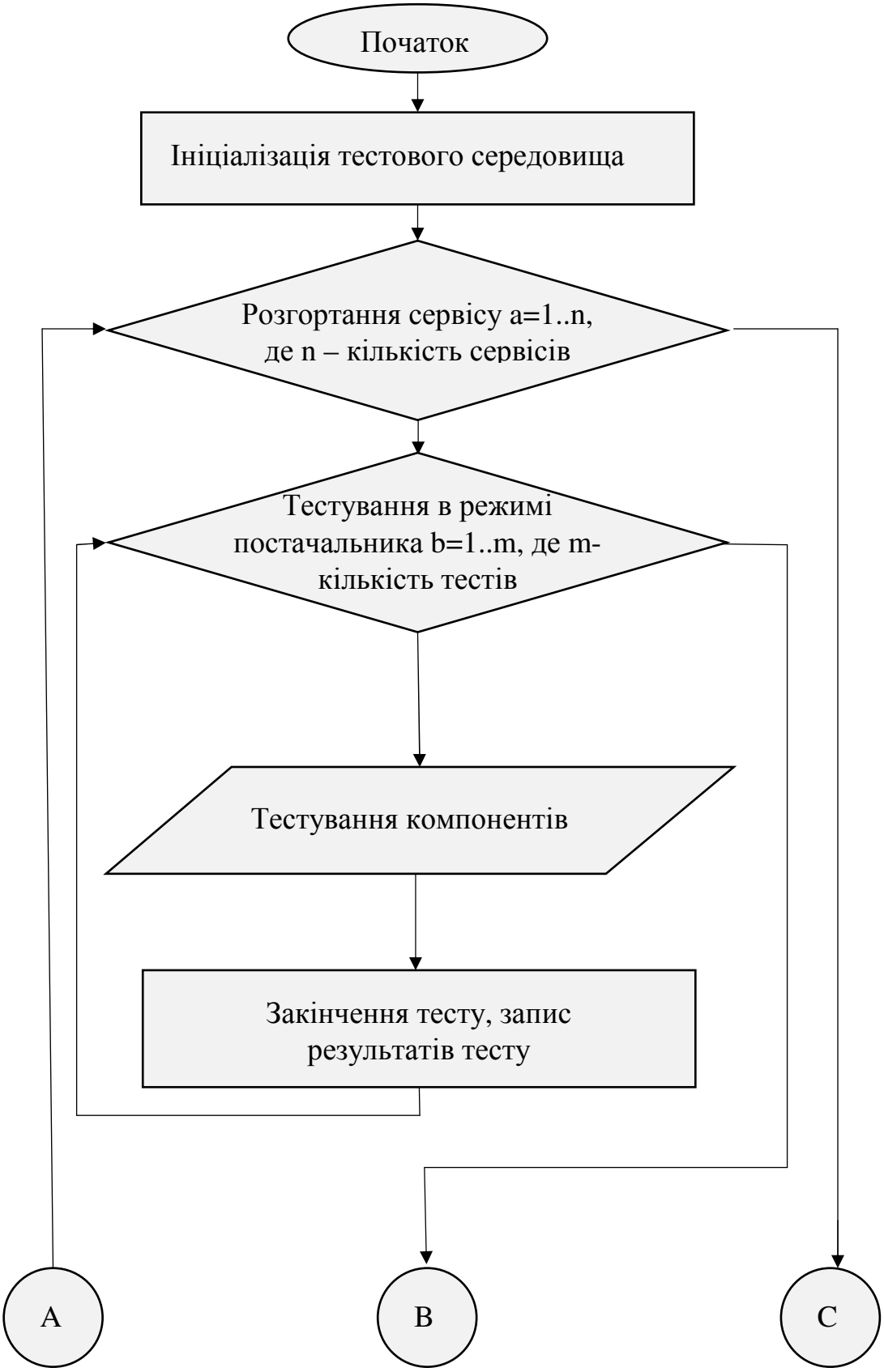


Рис 3.5. Схема методу тестування сервісів розподіленої системи за допомогою контрактних тестів (перша частина)

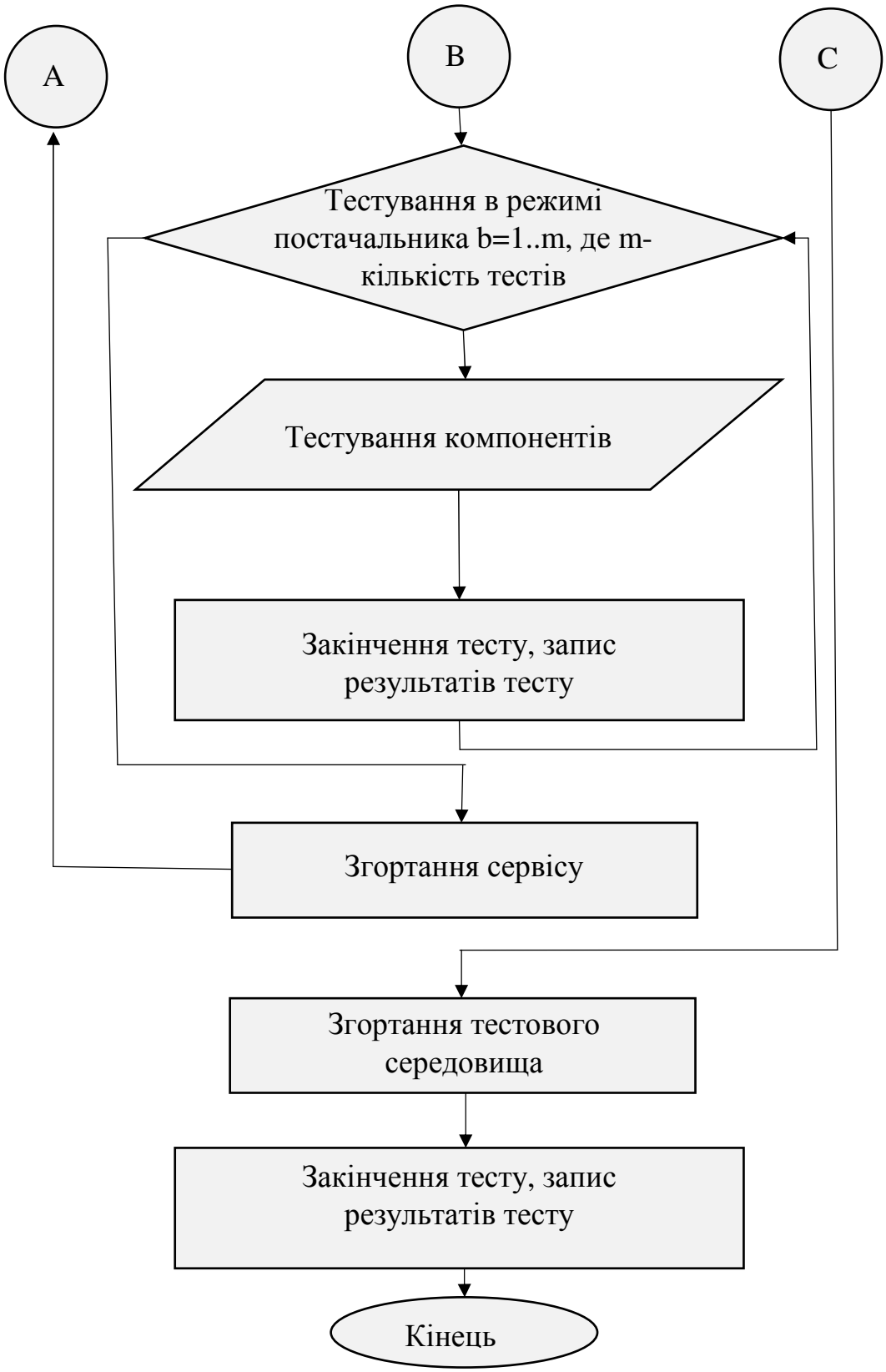


Рис 3.6. Схема методу тестування сервісів розподіленої системи за допомогою контрактних тестів (друга частина)

На першому етапі запуску тестів з використанням контрактів – проводиться розгортання тестового середовища. Тестове середовище складається з спеціальних програм-імітаторів, які працюють таким чином, що сервіси при роботі з ними обмінюються запитами, як з реальним сервісом. В тестовому середовищі проводиться встановлення програмного забезпечення цих імітаторів. Для кожного з цих імітаторів тестувальнику потрібно задати інформацію щодо очікуваної поведінки сервісів, тобто сформувати масив даних всіх варіантів запитів та відповідей сервісу (контракти) згідно зі спроектованими раніше тестовими сценаріями. Ці контрактні тести надсилаються на імітатори, а самі імітатори встановлюються на локальній машині. Кожен імітатор має свій унікальний порт, по якому при тестуванні сервісів проводяться запити на нього та отримається від нього відповідь, яка ідентична тій, що відправляється на самі сервіси.

Після встановлення тестового середовища починається процес тестування по черзі усіх сервісів розподіленої системи обробки інформації. Сервіси тестуються у послідовному порядку та за принципом того, що вони не мають залежності один від одного.

Після розгортання сервісу, який буде тестуватись, всі зв’язки з іншими сервісами, які мав тестований сервіс, заміняються вищевизначеними імітаторами шляхом підміни адрес реальних сервісів на адреси імітаторів (локальна машина localhost + унікальний порт).

Коли сервіс та його імітатори успішно розгорнені, запускається тестування у режимі постачальника. У режимі постачальника тестуються випадки, коли тестований сервіс відправляє запити на інші сервіси. Оскільки сервіси підмінені імітаторами, які мають масив очікуваних на них запитів, якщо на імітатор приходить очікуваний запит, то він віддає очікувану відповідь. Відповідь обробляється у сервісі, який у свою чергу повертає результат своєї роботи. Якщо результат його роботи співпадає з очікуваним результатом, то тест вважається пройденим. Якщо імітатор не знайшов такого запита у своєму масиві очікуваних запитів, то значить, на імітатор прийшов некоректний запит і

сервіс працює неправильно. Якщо імітатор віддав сервісу очікувану відповідь, але результат роботи сервісу виявився відмінним від очікуваного, це значить, що сервіс невірно обробив відповідь від імітатора і, відповідно, працює неправильно.

По закінченні проходження тестів в режимі постачальника результати тесту записуються в базу знань і процес тестування переходить у тестування в режимі споживача.

Режим споживача – це коли перевіряються усі випадки, коли інші сервіси відправляють запити на сервіс, який тестується. Всі імітатори сервісів, які надсилають запити на тестований сервіс, беруть всі очікувані запити зі свого заготовленого масива та відправляють ці запити на сервіс. Запити обробляються всередині сервісу і сервіс повертає результат своєї роботи. Якщо цей результат співпадає з очікуваним результатом роботи сервісу, то тест вважається пройденим. Якщо результат не співпадає, то тест невірно обробляє запити та містить помилку. В такому випадку тест вважається не пройденим. По завершенні тестування в режимі споживача результати тестування та роботи сервісу під час тестування записуються в базу знань.

Після закінчення тестування в режимі споживача сервіс згортається. Далі процес тестування переходить до наступного сервісу, який розгортається разом з імітаторами, які приймають або відправляють на нього запити.

По закінченню тестування всіх сервісів тестове середовище згортається, результати проходження кожного тесту записуються в базу знань та виводяться на засоби оцінки помилок, які виникли під час тестування.

Сформуємо кроки удосконаленого методу тестування із застосуванням механізмів контрактних тестів:

Крок 1. Проводиться встановлення середовища для тестування.

Крок 2. Відбувається розгортання сервісу, який буде тестуватися.

Крок 3. Проводиться тестування в режимі постачальника, при цьому система тестування моделює собою сервіс, який відправляє запити та отримує відповіді.

Крок 4. Проводиться тестування в режимі споживача, при цьому система тестування приймає запит від сервісу та моделює відповідь на нього.

Крок 5. Відбувається запис результату проведення тесту в базу знань.

Крок 6. Відбувається згортання сервісу та очищення пам’яті в тестовому середовищі.

Крок 7. Кроки 2 – 6 повторюються до тих пір, поки не будуть протестовані всі сервіси розподіленої системи обробки інформації.

Крок 8. Відбувається згортання тестового середовища.

Крок 9. Проводиться оцінка результатів тестування та підготовка рекомендацій розробникам.

3.2 Удосконалення методу тестування інтерфейсу розподіленої системи з використанням симуляційних тестів

3.2.1 Механізм підміни функціоналу для тестування сервісів розподіленої системи

Перевагою архітектури розподілених систем обробки даних у порівняні з монолітними архітектурами є те, що кожна частина (сервіс) є незалежною одиницею, у якого є свій API та який може підключатися до інших API. Використовуючи метод підміни функціоналу імітаторами, можуть тестуватися функціонали сервісів, використовуючи контрактні тести. Виникає потреба в застосуванні цього механізму тестування для інтерфейсів користувачів. Даний механізм допоможе значно скоротити час на перевірку компонентів інтерфейсу, оскільки не потрібно установлювати повністю всю систему, а достатньо установити один модуль інтерфейсу та протестувати його. Це також зменшить ресурси дискового простору та пам’яті, які потрібні на віртуальних машинах.

Візьмемо фрагмент розподіленої системи обробки даних, у якій є, крім звичайних сервісів, ще і сервіси, які відповідають за обмін інформацією з користувачами (рисунок 3.7).

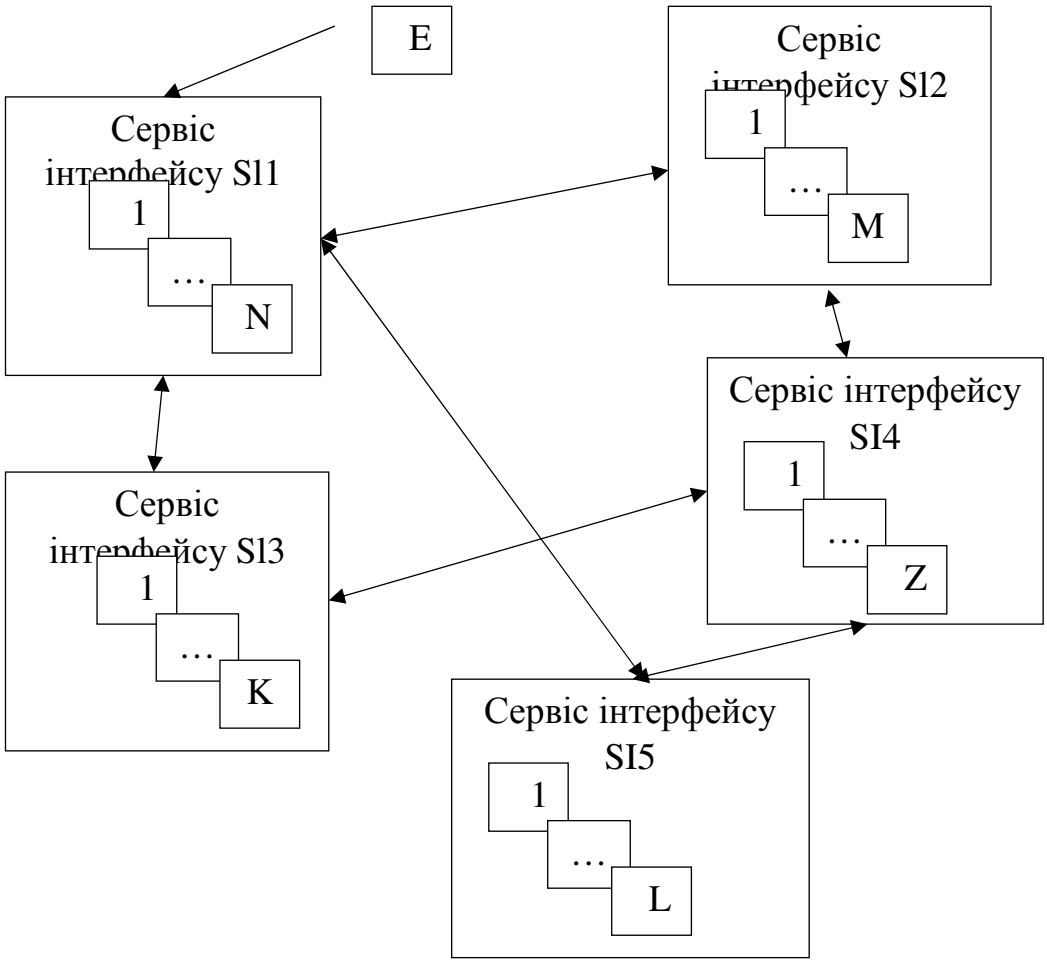


Рис. 3.7. Розподілена система сервісів інтерфейсів користувача

Сервіси інтерфейсів SI_1, SI_2, SI_3 містять деяку кількість компонентів N, M, K відповідно. E — точка входу в систему з зовнішнього середовища, через яку користувач має можливість почати роботу з системою. Сервіси інтерфейсу SI_4, SI_5 містять деяку кількість компонентів Z та L відповідно.

Нехай час $to_1(i)$ – час розгортання інтерфейсу сервісу SI_1, де $i = 1,2,\ldots,N$ кількість компонентів, $tl_1(i)$ – час завантаження компонентів інтерфейсу сервісу SI_1, $tt_1(i)$ – час проходження тестування сервісу інтерфейсу SI_1. Аналогічно для інших сервісів інтерфейсів розподіленої системи.

При тестування компонентів інтерфейсу користувача потрібно також врахувати ще важливий параметр: час розгортання ts середовища тестування, оскільки цей етап буває дуже часозатратним у великих системах.

Відповідно, загальний час T, який буде затрачений на тестування усіх компонентів розподіленої системи, буде рівний сумі часу розгортання середовища, часу розгортання сервісів та сервісів інтерфейсів, часу завантаження інтерфейсів, часу тестування сервісів та сервісів інтерфейсів, а також часу згортання всього середовища, тобто:

$$T = \sum_{i=1}^{N}(ti1(i) + tl1(i) + tt1(i)) + \sum_{i=1}^{M}(ti2(i) + tl2(i) + tt2(i)) + \sum_{i=1}^{L}(ti3(i) + tl3(i) + tt3(i)) + \sum_{i=1}^{K}(ti4(i) + tl4(i) + tt4(i)) + \sum_{i=1}^{Z}(ti5(i) + tl5(i) + tt5(i)) + t(s) \quad (3.3)$$

При наскрізному тестуванні інтерфейсу розподілених систем складнощі з'являються, коли у нас є два або більше сервісів. Наприклад, для того, щоб протестувати інтерфейс сервісу $SI1$, нам потрібно спочатку розгорнути усі сервіси, а потім пройти через завантаження елементів інтерфейсу сервісів.

Та час T_{SI2}, який буде затрачено на тестування сервісу інтерфейсу $SI2$, буде часом тестування всієї системи, тому що потрібно встановлювати всі сервіси для повного наскрізного тестування. Тому дана формула позує, що використання наскрізних тестів в архітектурі розподілених систем приводить до необхідності встановлювати всю систему та, відповідно, витрачати час та ресурси на це.

Розробимо новий механізм для використання його у методах тестування, який заснований на використанні симуляторів. Симулятор (заглушка) – це сервіс, який встановлюється разом з сервісом інтерфейсу та дозволяє тестувати його.

Добавимо в загальну систему імітатори для тестування сервісів інтерфейсів. В даному випадку ми будемо використовувати два типи імітаторів: імітатор входу V та імітатор виходу W. Імітатори входу подіють на сервіс

інтерфейсу заготовлену інформацію, яка необхідна для завантаження усіх елементів інтерфейсу користувача з боку звичайного сервісу. Причому може виникнути ситуація, коли на вхід потрібно подати кілька імітаторів входу для покриття різних видів запитів до інтерфейсу. Імітатор виходу працює у протилежному напрямку, тобто перевіряє відповіді, які відправляє сервіс інтерфейсу. Сервіс інтерфейсу бачить імітатори входу та виходу як сервіси, з якими він обмінюється інформацією (рисунок 3.8).

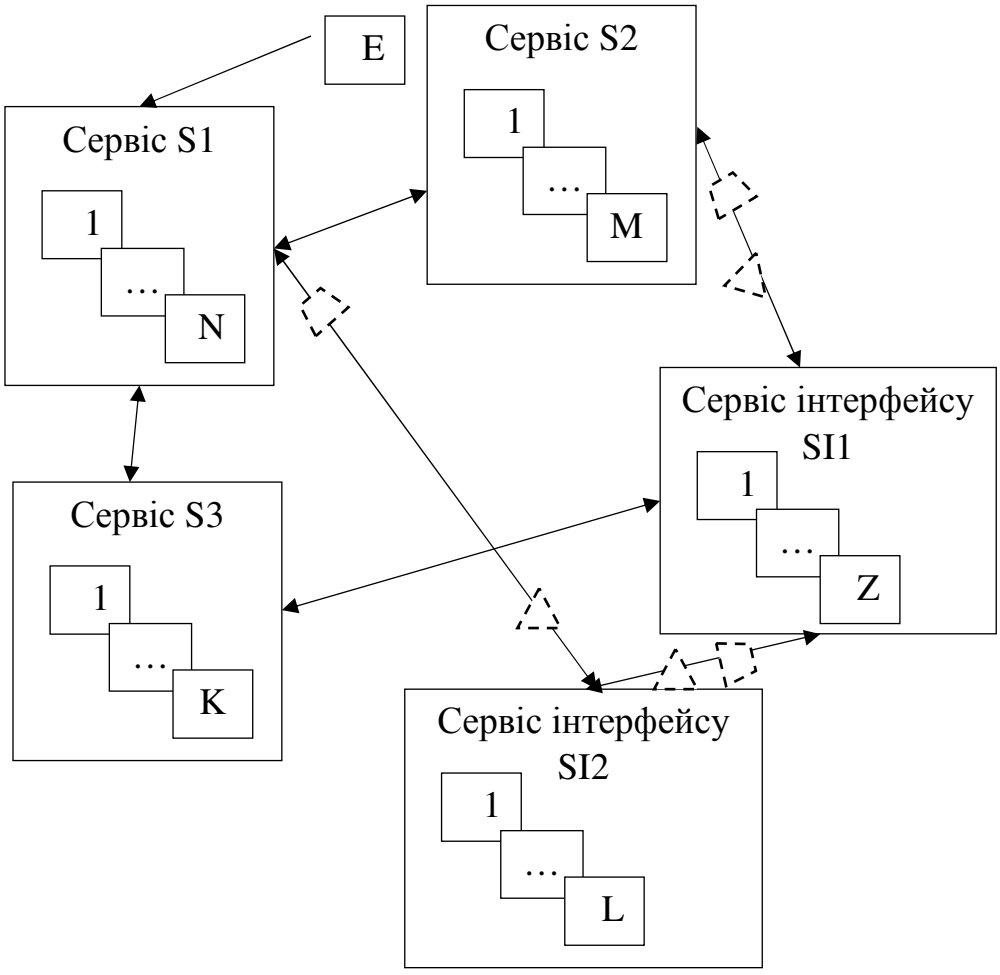


Рис. 3.8. Розподілена система сервісів інтерфейсів користувача із застосуванням симуляторів

Параметри імітаторів входу та виходу складається з масиву пар “запит-відповідь”, кожен елемент якого містить наступні параметри:

1. Запит містить: заголовки запиту (Headers), тип запиту (POST, PUT, DELETE), кінцева точка входу (контролер, на який буде іти запит), тіло запиту.

2. Відповідь містить: статус (код) відповіді, тіло відповіді.

Незважаючи на те, що імітатори входу та виходу мають однакові параметри, їхній функціонал суттєво відрізняється один від одного.

Імітатор входу надсилає на сервіс, який тестується, одразу весь свій масив запитів. Запит надсилається автоматично перед запуском тестів. Відповідно сервіс, який тестується, ще до початку тестування має можливість коректно завантажити всі елементи інтерфейсу користувача, для повноцінної роботи яких потрібна інформація з інших сервісів.

Імітатор виходу містить масив очікуваних запитів та відповідей на них. Коли сервіс, який тестується, відправляє запит на імітатор, імітатор проводить пошук аналогічного запиту у масиві своїх очікуваних запитів. Коли аналогічний запит знайдено, імітатор надсилає заготовлену відповідь на сервіс. Якщо відповідь не знайдено, імітатор відправляє відповідь з кодом 404 (Не знайдено) та інформує тестувальника про цю відповідь. Пошук причини такої поведінки є відповідальністю тестувальника, та може бути зв'язаний з некоректною роботою системи, тобто є цінним сигналом про наявність помилок у сервісі.

Тоді час тестування одного сервісу інтерфейсу $SI1$ буде рівний:

$$T = \sum_{i=1}^{N} (to1(i) + tl1(i) + tt1(i) + tsi2(i) + tsi4(i) + tso5(i)) + t(s) \qquad (3.4)$$

де $tsi2(i)$ – час розгортання стимулятора входу від сервісу інтерфейсу SI_2, $tsi4(i)$ – час розгортання стимулятора входу від сервісу інтерфейсу SI_4, $tso5(i)$ – час розгортання стимулятора виходу від сервісу інтерфейсу SI_5.

В порівнянні з наскрізним тестуванням, переваги даного підходу очевидні. Час, затрачений на тестування будь-якого сервісу інтерфейсу, буде значно менший, ніж при наскрізному тестування цього ж сервісу. Наприклад, в розподіленій системі, яка була розглянута для тестування компонентів сервісу інтерфейсу SI_1, нам не потрібно розгортати чотири сервіси та чекати на

завантаження компонентів сервісів SI_2, SI_3, SI_4, SI_5, а для тестування компонентів сервісу інтерфейсу $SI2$ нам не потрібно розгортати усі п’ять сервісів та чекати на завантаження компонентів сервісів інтерфейсу SI_1, SI_3, SI_4, SI_5 і т.д.

3.2.2 Удосконалення методу тестування інтерфейсу розподіленої системи з використанням симуляційних тестів

В порівнянні з наскрізним тестуванням інтерфейсу користувача, переваги підходу застосування симуляційних тестів для тестування сервісів інтерфейсів розподілених систем очевидні. Час, затрачений на тестування будь-якого сервісу інтерфейсу, буде значно менший, ніж при наскрізному тестування інтерфейсу користувача цього ж сервісу. Наприклад, розподіленій системі, яка була розглянута для тестування компонентів сервісу інтерфейсу SI_1, нам не потрібно розгортати всі п’ять сервісів інтерфейсів та чекати на завантаження компонентів всіх сервісів інтерфейсів. Також, для процесу тестування потрібно виділити значно менші ресурси на дисковий простір та оперативну пам'ять.

Розглянемо схему тестування наскрізними тестами інтерфейсу користувачів розподіленої системи обробки інформації (рис. 3.9, 3.10).

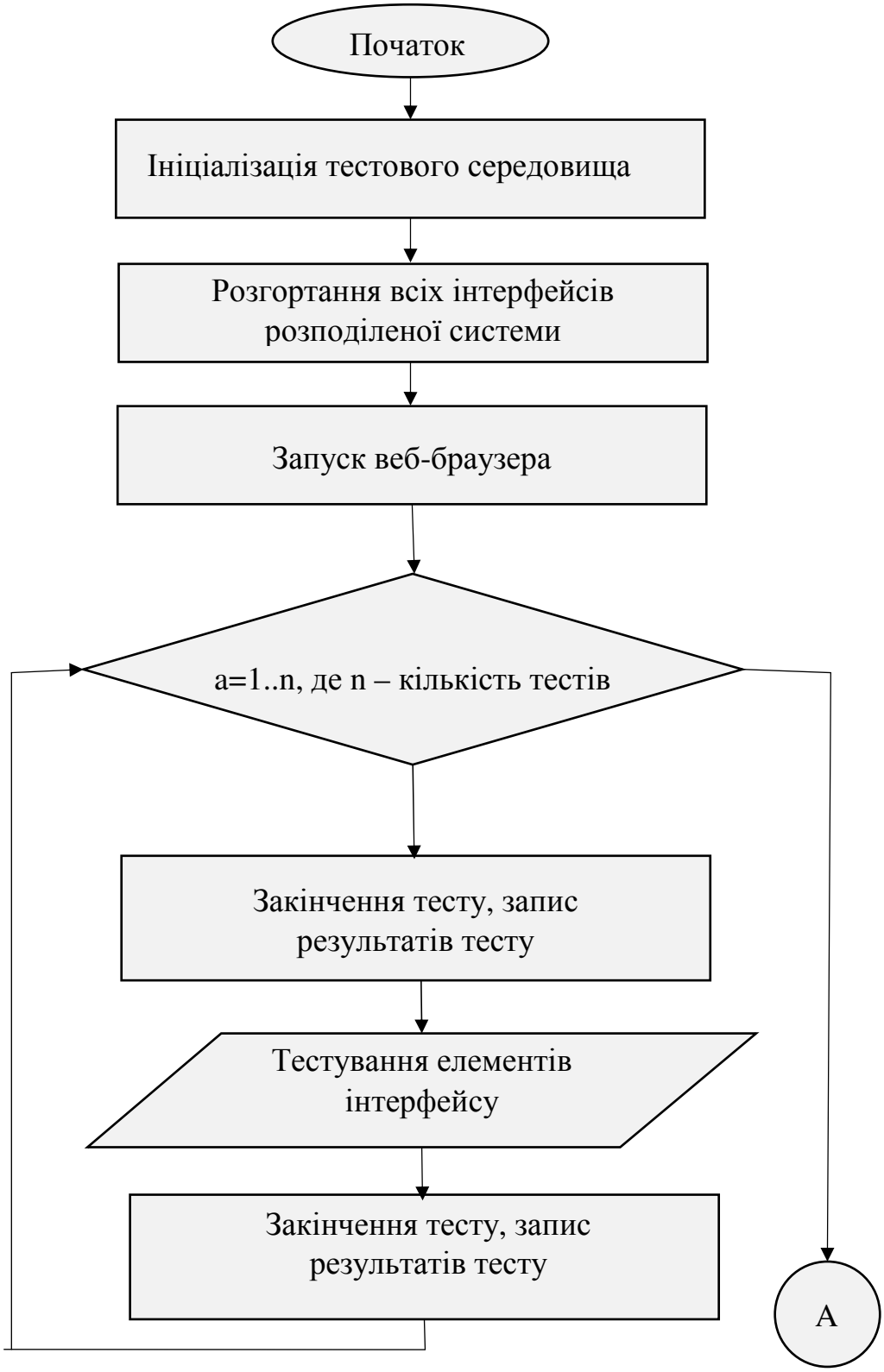


Рис 3.9. Схема методу тестування інтерфейсів користувачів розподіленої системи за допомогою наскрізних тестів (перша частина)

Рис 3.10. Схема методу тестування інтерфейсів користувачів розподіленої системи за допомогою наскрізних тестів (друга частина)

Для початку тестування інтерфейсів наскрізним способом потрібно встановлювати тестове середовища розподіленої обробки інформації та ініціалізувати його. При запуску тестів на локальній машині потрібно провести встановлення різних версій та типів веб-браузерів. Далі проходить встановлення усіх сервісів інтерфейсів системи та інсталяція необхідних пакетів для її роботи. Якщо встановлення розподіленої системи обробки інформації пройшло

успішно, тестовий фреймворк ініціалізує масив тестів та розставляє їх у порядку відповідно до режиму запуску цих тестів: послідовного або паралельного. Перед запуском кожного з тестів відкривається відповідна версія та тип веб-браузера. Кожен тест починається з завантаження головної сторінки інтерфейсу розподіленої системи обробки інформації. Далі послідовно тест переходить шляхом до необхідного інтерфейсу, який потрібно буде тестувати. Якщо завантаження необхідного елементу не відбулося протягом заданого в тестах параметру часу, то тест завершується з помилковим результатом та повідомленням користувачу, що необхідний елемент не було знайдено. Коли необхідний інтерфейс завантажений, то відбувається тестування компонентів його функціоналу. Один тест – це один блок функціоналу інтерфейсу. Якщо якийсь функціонал поводить себе невідповідно очікуваному результату, то тест вважається не пройденим. А якщо функціонал інтерфейсу поводить себе згідно очікуваних результатів, тест вважається пройденим і проводиться його завершення. При будь-якому результаті завершення тестів його результат записується в базу знань і тестове середовище переходить до наступного тесту інтерфейсу. Після закінчення проходження всіх тестів для всіх видів та типів веб-браузера тестувальник завершує свою роботу, а всі результати тестів з бази знань виводяться на засоби оцінки помилок, які виникли під час тестування, та відбувається згортання тестового середовища.

Розглянемо схему роботи удосконаленого методу тестування інтерфейсу користувача вузлів розподіленої системи з використанням стимуляторів (рис. 3.11, 3.12).

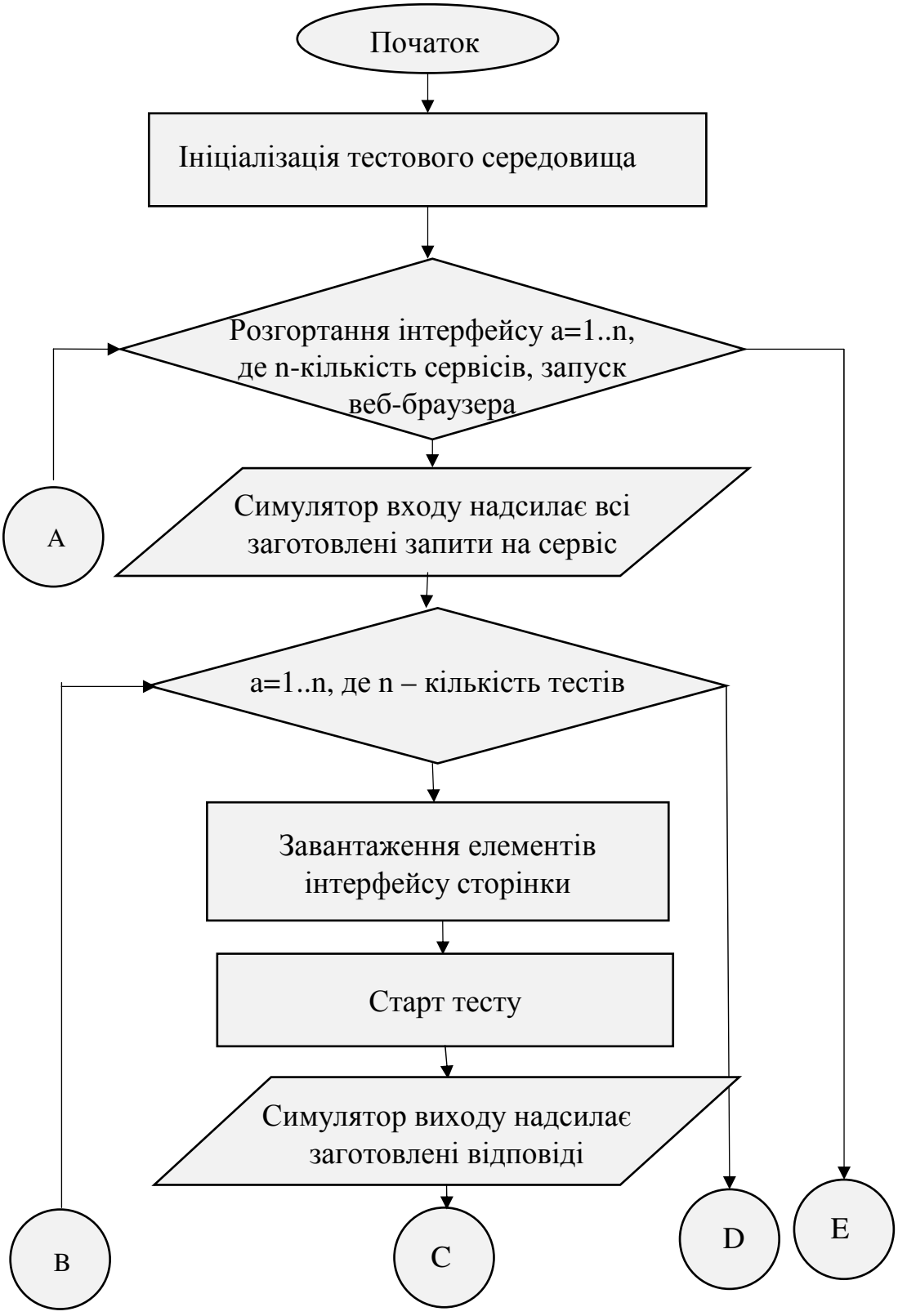


Рис 3.11. Схема методу тестування інтерфейсів користувачів розподіленої системи за допомогою симуляційних тестів (перша частина)

Рис 3.12. Схема методу тестування інтерфейсів користувачів розподіленої системи за допомогою симуляційних тестів (друга частина)

Розгортання тестового середовища включає у себе розгортання симуляторів входу і виходу та ініціалізація пакетів, необхідних для їх роботи.

Симулятори підключені до кожного інтерфейсу за допомогою програмного коду в якості частини тестового сценарію, який повинен буде виконуватися. На кожен симулятор подаються необхідні елементи інтерфейсу, які під час тесту будуть передаватися на інтерфейс, який тестується. Симулятори представляють собою масиви елементів інтерфейсу, які потрібні для коректного завантаження інтерфейсу, який буде перевірятися у тесті. Це можуть бути не тільки запити, а і рисунки, картинки, таблиці, посилання та інші елементи інтерфейсу. Після розгортання тестового середовища починається розгортання цільового інтерфейсу, який потрібно тестувати, а також відбувається завантаження різних видів та типів веб-браузера, для яких проводиться тестування. Після завантаження цільового інтерфейсу симулятор входу подає на інтерфейс всі необхідні елементи зв'язаних з ним інтерфейсів, які потрібні для коректної роботи інтерфейсу, який тестується. Таким чином відбувається повне завантаження сторінки, яка складається з усіх зв'язаних між собою інтерфейсів. Відбувається запуск тесту і у веб-браузері проходить перевірка коректності роботи елементів інтерфейсу. Якщо під час тестування інтерфейсу з'являється необхідність завантаження елементів з іншого сервісу інтерфейсу, симулятор виходу відправляє ці значення зі свого масиву елементів на інтерфейс, який перевіряється. Якщо поведінка інтерфейсу, який тестується, задовольняє очікуваний результат з масиву результатів, то такий тест проходить. Якщо якийсь компонент не завантажився за заданий у параметрах тестів проміжок часу, робота інтерфейсу вважається помилковою і тест завершується з негативним результатом. Всі результати проходження тесту записуються у базу знань. Після завершення тесту запускається наступний тест і симулятори входу і виходу передають необхідні значення для завантаження та коректної роботи інтерфейсу. Після проходження всіх тестів завершується робота веб-браузерів, а результати тесту виводяться з бази знань на засоби оцінки помилок, які виникли під час тестування, та відбувається згортання тестового середовища.

Сформуємо кроки удосконаленого методу тестування інтерфейсів із застосуванням механізмів симуляційних тестів:

Крок 1. Проводиться встановлення середовища для тестування інтерфейсів користувачів.

Крок 2. Відбувається розгортання інтерфейсу користувача, який буде тестуватися.

Крок 3. Проводиться завантаження симулятора входу.

Крок 4. Відбувається завантаження сторінки інтерфейсу користувача.

Крок 5. Проводиться завантаження симулятора виходу.

Крок 6. Відбувається тестування елементів інтерфейсу користувача.

Крок 7. Результати тестування записуються у базу знань.

Крок 8. Відбувається згортання інтерфейсу користувача.

Крок 9. Кроки 2 – 8 повторюються до тих пір, поки не будуть протестовані всі інтерфейси користувачів.

Крок 10. Відбувається згортання тестового середовища.

Крок 11. Проводиться оцінка результатів тестування та розробляються рекомендації розробникам програмного забезпечення.

3.3 Висновки до розділу 3

Показано, що при одноразовому тестування розподіленої системи з невеликою кількості вузлів час тестування при класичному варіанті буде менший в зв’язку з тим, що сума часів розгортання та згортання кожного сервісу буде більшою за розгортання всієї системи. Але, якщо потрібно кілька разів тестувати всю систему, то тестування за допомогою контрактів є більш вигідним, тому що проводяться зміни тільки в кількох сервісах і другий раз уже не потрібно розгортати всю систему, а тільки окремі сервіси.

Удосконалено метод тестування програмного забезпечення вузлів розподіленої системи, який відрізняється від існуючих застосуванням контрактних тестів та аналізом прогнозованого результату поведінки сервісів, що дозволяє зменшити час розгортання компонентів тестового середовища та час проходження тестів.

Показано, що в порівнянні з наскрізним тестуванням інтерфейсів користувача переваги застосування механізмів симуляторів тестування інтерфейсів користувача дозволяють зменшити час, затрачений на тестування будь-якого сервісу інтерфейсу користувача в порівнянні з наскрізним тестування інтерфейсу користувача. Час зменшується за рахунок зменшення кількості одночасних сервісів інтерфейсу користувача.

Удосконалено метод тестування інтерфейсу користувача програмного забезпечення вузлів розподіленої системи, який відрізняється від існуючих механізмом симуляції його роботи, що дозволяє проводити тестування окремих компонентів інтерфейсу системи.

Результати, подані в розділі, автор опублікував у працях [1, 2, 3].

РОЗДІЛ 4
ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ПОРІВНЯЛЬНА ОЦІНКА КЛАСИЧНИХ ТА РОЗРОБЛЕНИХ МЕТОДІВ ТЕСТУВАННЯ СИСТЕМ РОЗПОДІЛЕНОЇ ОБРОБКИ ІНФОРМАЦІЇ

4.1. Розробка спеціалізованого фреймворку для дослідження параметрів та способів тестування програмного забезпечення розподілених систем

Для застосування розробленої моделі, яка запропонована у другому розділі, та двох удосконалених методів, запропонованих у третьому розділі була розроблена інформаційна технологія та інформаційна система для підвищення ефективності тестування програмного забезпечення розподілених систем.

Інформаційна технологія забезпечує проведення тестування сервісів API та інтерфейсів розподілених систем (рис. 4.1). Вхідними даними є: набір сервісів, набір тестів, тестові дані та тип тестування. Після проходження ініціалізації тестового фреймворку та розгортання тестового середовища вібувається тестування удосконаленими методами, схеми роботи яких описані у другому розділі. На виході отримуємо наступні показники: час тестування кожного сервісу, помилки, які виникли при тестуванні сервісів, список сервісів, які працюють з помилками та непрацюючі компоненти (методи) цих сервісів.

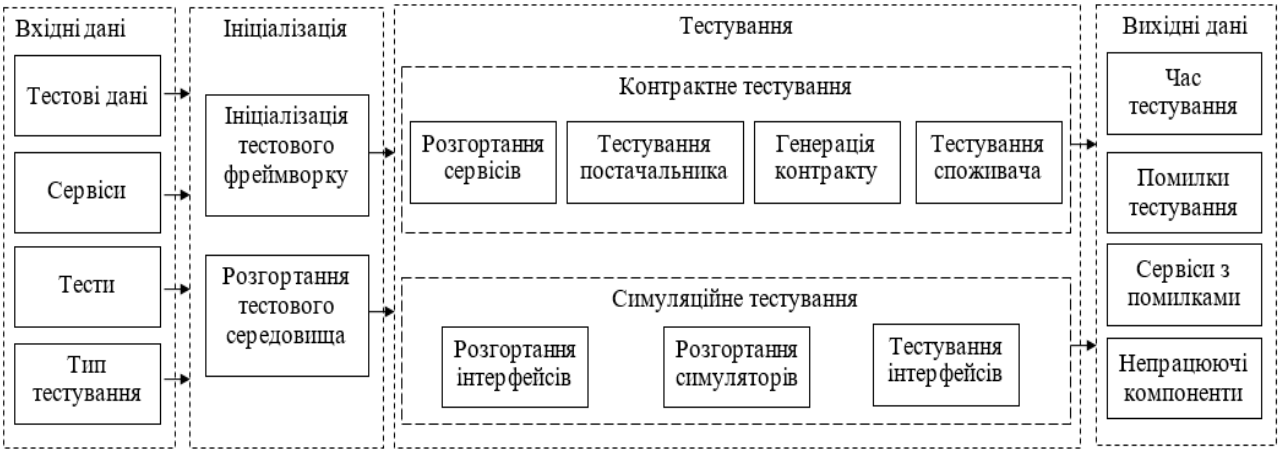


Рис. 4.1. Схема інформаційної технології тестування розподілених API та інтерфейсів

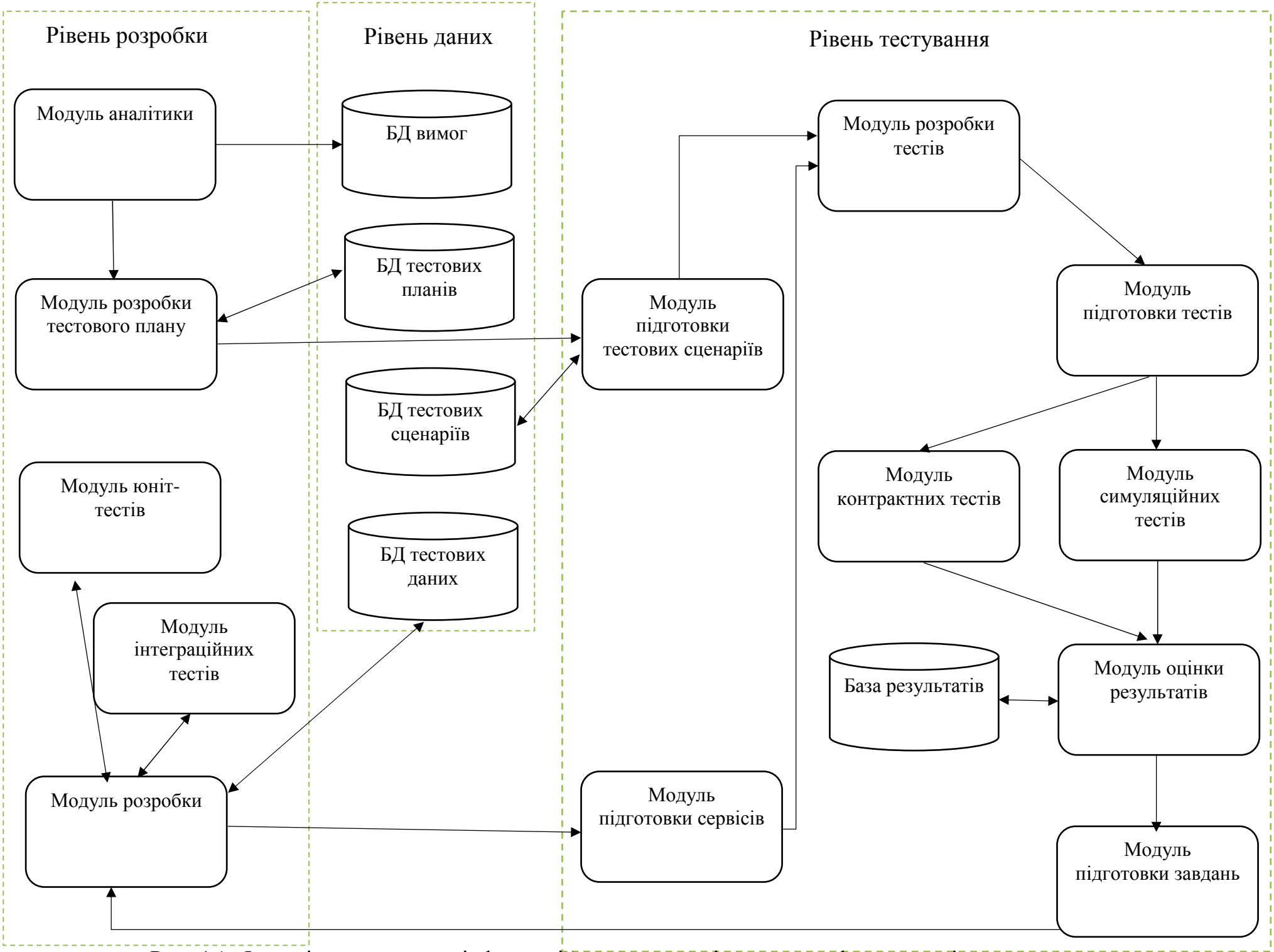


Рис. 4.1. Функціональна схема інформаційної системи підвищення ефективності тестування

На основі інформаційної технології було подано інформаційну систему, яка складається з трьох рівнів виконання: рівень розробки, рівень даних та рівень тестування (рис. 4.2). Рівень даних відповідає за збереження даних у базах даних та містить наступні сховища: база даних вимог, база даних тестових планів, база даних тестових сценаріїв та база даних тестових даних.

На рівні розробки модуль аналітики відповідає за формування вимог до розробки продукту та запису цих вимог в базу даних. Зазвичай, це обов'язки бізнес-аналітика у команді розробників програмного забезпечення. Розробка тестового плану проводиться на основі аналізу вимог до розробки програмного продукту. По готовності тест-план відправляється до тестувальника для підготовки тестових сценаріїв.

Модуль розробки відповідає на розробку продукту командою програмістів. Створюючи сервіси розподіленої системи, розробники забов'язані покривати розроблений функціонал юніт-тестами та інтеграційними тестами в зв'язку з тим, що це займає менше часу, ніж коли це виконує тестувальник.

Рівень тестування відповідає за процес тестування програмного забезпечення розподілених сервісів. Для початку тестування розробник тестів готує тестові сценарії, які за допомогою фреймворку реалізує їх у вигляді програмного коду, який тестує розроблений сервіс.

Тестувальник проводить розробку тестів для тестування сервісу та підготовку та запуск контрактних та симуляційних тестів на розроблених сервісах. Отримані результати зберігаються у базі знань. На основі знайдених помилок тестувальник проводить їх опис та передає команді розробників, які після отримання нових завдань для розробки проводять модифікацію програмного продукту.

Для проведення експериментальних досліджень тестування розподілених систем обробки інформації було розроблено фреймворк, за допомогою якого можна проводити контрактне тестування API сервісів розподіленої системи. Для розробки фреймворку використовувалась платформа .NET Core 3.1, яка сумісна з різними операційними системами, такими, як операційною системою Windows

та Linux. Мовою програмування було обрано C#, так як вона активно розвивається та забезпечує відповідний набір бібліотек для розробки фреймоворків. Розробка проводилася в середовищі Visual Studio 2019, яке дозволяє створювати готові контейнерні зборки. Також для тестування було використано у якості бібліотеки для запуску тестів інструмент, код якого можна модифіковувати – Xunit.

Для запуску системи тестування за механізмом, який використовує контракти, у вигляді імітатора робочого сервісу було використано інструмент підміни Mountebank, який дозволяє створювати віртуальні HTTP сервіси, прописувати їм різні методи для роботи, задавати параметри цих методів, які використовуються для тестування, та розміщувати їх на різних портах локальної машини чи віртуальної машини.

Для встановлення тестового середовища по проведенню експериментів, встановлення у ньому сервісів на основі контрактів та API сервісів розподіленої системи обробки інформації було обрано програмне забезпечення, яке дозволяє автоматизувати розгортання, управління та згортання середовища – Docker. Завдяки йому потрібно розгорнути всі необхідні компоненти в контейнерах, що дозволить тестовому фреймворку не залежати від операційної системи.

Опишемо структуру нашого фреймворку та його методи, які використовуються для тестування (рис. 4.3). В основний функціонал фреймворку входять два класи: ContractBuilder та ContractExecuter.

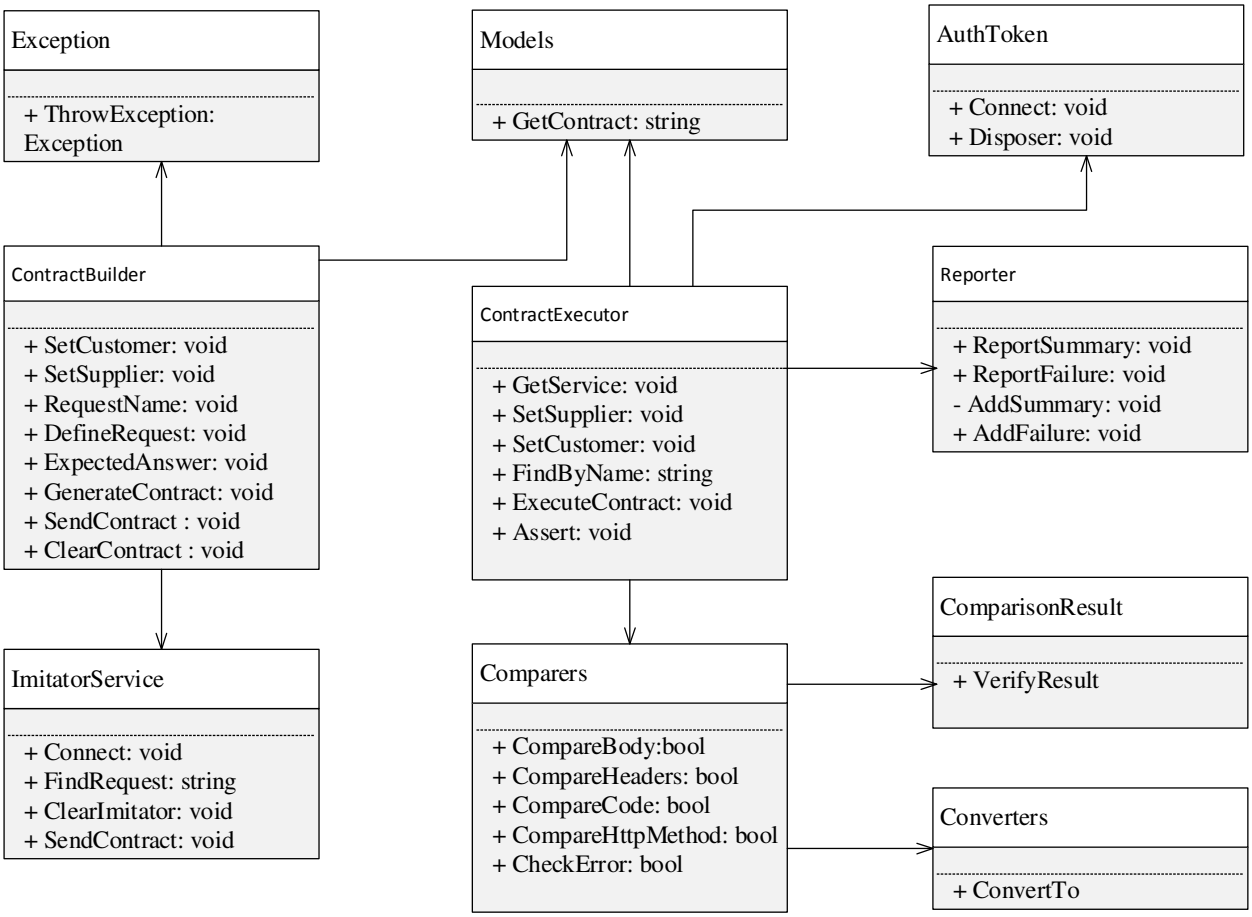


Рис. 4.3. Діаграма класів розробленого фреймворку

Клас ContractBuilder відповідає за визначення очікуваної поведінки для сервісу та оформлення її у вигляді контракту. Контракт формується у форматі JSON. Також цей клас відповідає за передачу контракту від тесту до сервісу імітатора, який працює за допомогою інструмента Mountebank. Масив очікуваної поведінки сервісу імітатора зберігається на сервісі, і при очікуваному запиті на цей сервіс імітатор поверне очікувану відповідь.

В класі ContractBuilder є наступні методи:

1) SetCustomer — задати ім’я сервіса-споживача;

2) SetSupplier — задати ім’я сервіса-постачальника;

3) RequestName — задати назву запиту, який буде відбуватися на інший сервіс;

4) DefineRequest — задати параметри запиту, який надсилатиметься, його http-метод, шлях, тіло запиту на хедери;

5) ExpectedAnswer — задати параметри очікуваної відповіді, яка повинна прийти від сервісу;

6) GenerateContract — створити контракт, в якому буде записана очікувана поведінка сервісу у методах, описаних вище;

7) SendContract — надіслати згенерований контракт на сервіс-імітатор, створений за допомогою Mountebank;

8) ClearContract — видалити масив контрактів з сервіса-імітатора.

Клас ContractExecuter відповідає за пошук необхідного для тестування контракту у сервісі імітаторі, формування з даних у контракті тіла запиту, який буде відправлятися на сервіс в режимі постачальника, та перевірку відповіді на цей запит від сервісу та порівняння її з очікуваною відповіддю. Якщо очікувана відповідь співпадає з еталонною для даного тесту, то тест вважається пройденим.

Розглянемо методи класу ContractExecuter:

1) GetService — вказати адресу сервіса-імітатора, на якому розміщений контракт;

2) SetSupplier — задати ім’я сервіса-постачальника;

3) SetCustomer — задати ім’я сервіса-споживача;

4) FindByName — пошук необхідного запиту в масиві контрактів за його іменем;

5) ExecuteContract — відправка обраного запису з контракту на сервіс-постачальник;

6) Assert — перевірка отриманого результату з очікуваним.

Також було розроблено інші допоміжні класи розробленого фреймворку:

• Comparers - містить функції для порівняння різних складових відповіді від сервісу: тіло запиту, хедери, методи, коди відповіді, повідомлення про помилки;

• Reporter — відповідає за генерацію результату проходження тестів та дозволяє налаштовувати формат запису результатів тестування у базу знань;

• ImitatorService — відповідає за встановлення з’єднання з сервісом імітатором та передачу контрактів на нього;

• AuthToken — налаштування параметрів авторизації на сервіс, який тестується;

• ComparisonResult — верифікація результатів запиту на сервіс;

• Models — містить допоміжні моделі, необхідні для розробки фреймворку;

• Converter — дозволяє конвертувати моделі з однієї структури даних в іншу;

• Exception — опис помилок, які можуть статися при роботі над фреймворком.

Для проведення процедури генерування контракту для проведення тестування сервісу потрібно виконати операції по підготовці до обробки файлу. Розроблений фреймворк генерує контракт у вигляді JSON-файлу. Контракт містить дані про споживача та постачальника, опис запиту, його тіло та очікувану відповідь.

Наведемо приклад згенерованого контракту, в якому є два запити – GET і POST – на отримання id користувача та створення користувача. Для кожного запиту є очікувана відповідь від сервісу.


```
{
  "Supplier": "Service2",
  "Customer": "Service1",
  "ImitatorHost": "localhost:4684",
  "Contract": [
    {
      "Name": "get user by id",
      "Request": {
        "Method": "get",
        "Path": "/id/a250"
      },
```

```
      "ExpectedAnswer": {
        "StatusCode": 200,
        "Body": {
          "Id": "a250",
          "Name": "Main User",
          "Email": "main@test.com"
        }
      }
    },

    {
      "Name": "create new user",
      "Request": {
        "Method": "post",
        "Path": "/new",
        "Body": {
          "Name": "Second User",
          "Email": "second@test.com"
        }
      },
      "ExpectedAnswer": {
        "StatusCode": 200,
        "Body": "User has successfully created"
      }
    }
  ]
}
```


Після виконання процедури генерування відбувається запуск сервісу імітатора та установка контракту, який дозволить провести тестування розподіленої системи.

За допомогою контейнерів Docker створюється сервіс-імітатор, на який фреймворк надсилає контракт. Створення контейнера з імітатором:

```
serviceimitator2:
   image: mountebank
   ports:
    - "2525:2525" // Порт для Mountebank
    - "4684:4684" // Зовнішній порт сервісу імітатору
```

Після того, як фреймворк згенерує контракт та відправить його на mountebank, можемо спостерігати запущений сервіс у системі mountebank:

```
{
 "imposters": [
  {
    "protocol": "http",
    "port": 4684,
    "_links": {
      "self": {
        "href": "http://localhost:2525/imposters/4684"
      }
    }
  }
 ]
}
```

Запуск тестів за допомогою контейнерів у програмному забезпеченні Docker потрібно провести:

Запуск тестів, а для цього необхідно:

- Створити Dockerfile для запуску сервісу;

- Створити Dockerfile для запуску тестів;
- Створити docker-compose файл для розгортання сервіса-імітатора, сервіса який тестується та тестового проекту.

Dockerfile для сервісу:

```
FROM microsoft/aspnetcore:3.1.0
COPY /publish /app
WORKDIR /app
ENV ASPNETCORE_URLS http://*:5000
EXPOSE 5000
ENTRYPOINT dotnet Service1.dll
```

Файл зберігається з іменем DockerfileService1 з типом File.

Dockerfile для тестового проекту:

```
FROM microsoft/aspnetcore:3.1.0
COPY . /TestProject
WORKDIR /TestProject
RUN dotnet restore --no-cache
```

Файл зберігається з іменем TestDockerfile з типом File

Docker-compose файл для розгортання імітатора, сервісу та тестового проекту:

```
version: '1'
services:
  #Розгортаємо контейнер з тестовим проектом за його Dockerfile
  testproject:
```

```
  build:
    dockerfile: TestDockerfile
  #Запускаємо тести
  command: dotnet test
  links:
    - service1
    - serviceimitator2

#Розгортаємо контейнер з сервісом, який буде тестуватися
service1:
  build:
    dockerfile: TestDockerfile
  ports:
    - "5010:5000"

#Розгортаємо сервіс імітатор
serviceimitator2:
  image: mountebank
  ports:
    - "2525:2525"
    - "4684:4684"
```

Таким чином запускаються всі необхідні компоненти для тестування у контейнерах незалежно від операційної системи машини чи її налаштованого середовища.

Docker-compose файл зберігається у форматі yml. Запустити цей файл потрібно за допомогою команди у консолі:

```
docker-compose up --build --no-cache
```

Для проведення тестування у режимі споживача потрібно визначити контракт та його поведінку, використовуючи розроблений фреймворк. Потім написати тест, який викликатиме методи сервісу, який тестується. При запитах тестованого сервісу на сервіс-імітатор, він отримуватиме відповіді, визначені у контракті. Отримана від виклику методу сервісу відповідь в тесті порівнюється з очікуваною відповіддю (рис. 4.4).

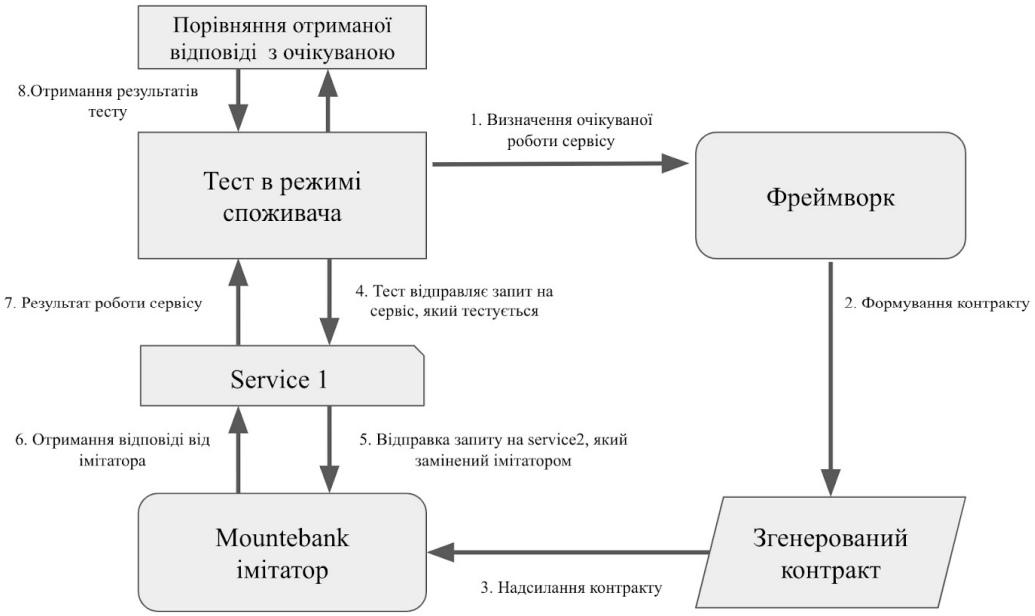


Рис. 4.4. Тестування у режимі споживача

Визначення контракту у тестах за допомогою розробленого фреймворку наведено у додатку А. Цей контракт забезпечує тестування всіх методів сервісу. Коли цей сервіс намагається зробити запит на інший сервіс, імітатор відправляє сервісу очікувану відповідь у випадку, якщо на імітатор прийшов правильний запит. Якщо на імітатор прийде запит, якого немає у масиві даних, то тест вважається непройденим, адже на імітатор не прийшло правильної відповіді. Якщо імітатор віддає правильну відповідь, яка потім неправильно обробиться у сервісі, який тестується, то цей сервіс поверне не ту відповідь, яку очікував тестувальник і, відповідно, тест вважатиметься непройденим.

Тест для перевірки методу сервісу визначений у додатку А. Результат проходження тесту в режимі споживача.

Build started, please wait...

Build completed.

Starting test execution, please wait...

Send request to service1

Method: POST; Path: “/registration/finish”; Body: { “Name”: "Second User", “Email”: "second@test.com", “PhoneNumber”: "0981234567", “UserExist” : false, “RegistrationDone” : true }; Encoding: UTF8; Content-Type: "application/json"

Send request to serviceimitator2

Mountebank:

POST /new HTTP/1.1

Body: { “Name”: "Second User", “Email”: "second@test.com"}

Host: localhost:4684

Content-Type: application/json

GET /imposters/2525 HTTP/1.1

Host: localhost:4684

Content-Type:: application/json

HTTP/1.1 200 OK

Vary: Accept

Content-Type: application/json; charset=utf-8

Connection: keep-alive

{

"protocol": "http",

"port": 2525,

"numberOfRequests": 1,

"recordRequests": false,

"requests": [

{

```
    "requestFrom": "service1:5010",
    "method": "POST",
    "path": "/new",
    "query": {},
    "headers": {
      "Content-Type": "application/json",
      "Connection": "keep-alive"
    },
    "body": { “Name”: "Second User", “Email”: "second@test.com"}
  }
 ],
 "stubs": [
  {
    "predicates": [
      {
        "equals": {
          "method": "POST",
          "path": "/new"
        }
      }
    ],
    "responses": [
      {
        "is": {
          "statusCode": 200,
          "body": "User has successfully created"
        }
      }
    ],
  }
```

}

Response has been received from serviceimitator2

Result:

Status Code: 200; Body: { "User has successfully created"}

Send response from POST "/registration/finish

Status Code: 200; Body: "Registration is finished and user has been created"

Test successfully finished

Тест проходить успішно, це значить – протестований метод даного сервісу працює коректно та віддає правильну відповідь на запит.

Для тестування в режимі постачальника (рис. 4.5) потрібно розгорнути сервіс 2, який в попередньому режимі був замінений імітатором з описаним контрактом та у тесті необхідно викликати метод ContractExecuter розробленого фреймворку, в якому вибрати запит, який буде тестуватися на цьому сервісі. Фреймворк відправить обраний запит сервіс, а відповідь від сервісу буде порівняна з очікуваною відповіддю, яка вказана у контракті. Після порівняння фреймворк видасть результат проходження тесту.

Рис. 4.5. Тестування в режимі постачальника

Код тесту в режимі постачальника винесено у додатку А. Результати проходження тесту:

Build started, please wait...

Build completed.

Starting test execution, please wait…

Mountebank:

GET /imposters/2525 HTTP/1.1

Host: localhost:4684

Content-Type:: application/json

Finding request in contract array...

Request is found...

Send request to service2

Method: POST; Path: “/new”; Body: {"Name": "Second User", "Email": "second@test.com" }; Encoding: UTF8; Content-Type"application/json"

Response has been received

Status Code: 200; Body: { "User has successfully created"}

Verifying with expected response...

Test successfully passed

Очікуваний результат співпав з записаним у контракті результатом, значить, тест пройшов успішно.

Таким чином, було розроблено фреймворк для тестування сервісів розподілених систем обробки інформації за допомогою механізмів на основі використання контрактів.

4.2. Експериментальне визначення середнього часу проходження тестів в архітектурі розподілених систем

При тестуванні сервісів розподілених систем за допомогою механізму контрактного тестування ми тестуємо кожен сервіс ізольовано. Тестування відбувається на предмет правильності надісланого запиту та отримання правильної відповіді. При наскрізному тестуванні потрібно розгорнути разом одразу кілька сервісів та запустити тести по відношенню до цих сервісів. Очевидно, що область дії такого тесту ширша. В той же час, такі тести виконуються повільніше та визначити джерело помилки в ході їх проведення значно складніше.

Для експерименту було побудовано дві різні системи з різною кількістю підсистем.

Система №1 (рис. 4.6) являє собою поштовий сервіс, який не потребує реєстрації (на прикладі сервісу mailforspam.com). Система складається з трьох підсистем (сервісів) та бази даних: A - сервіс авторизації, B - сервіс керування листами, C - сервіс керування користувачами.

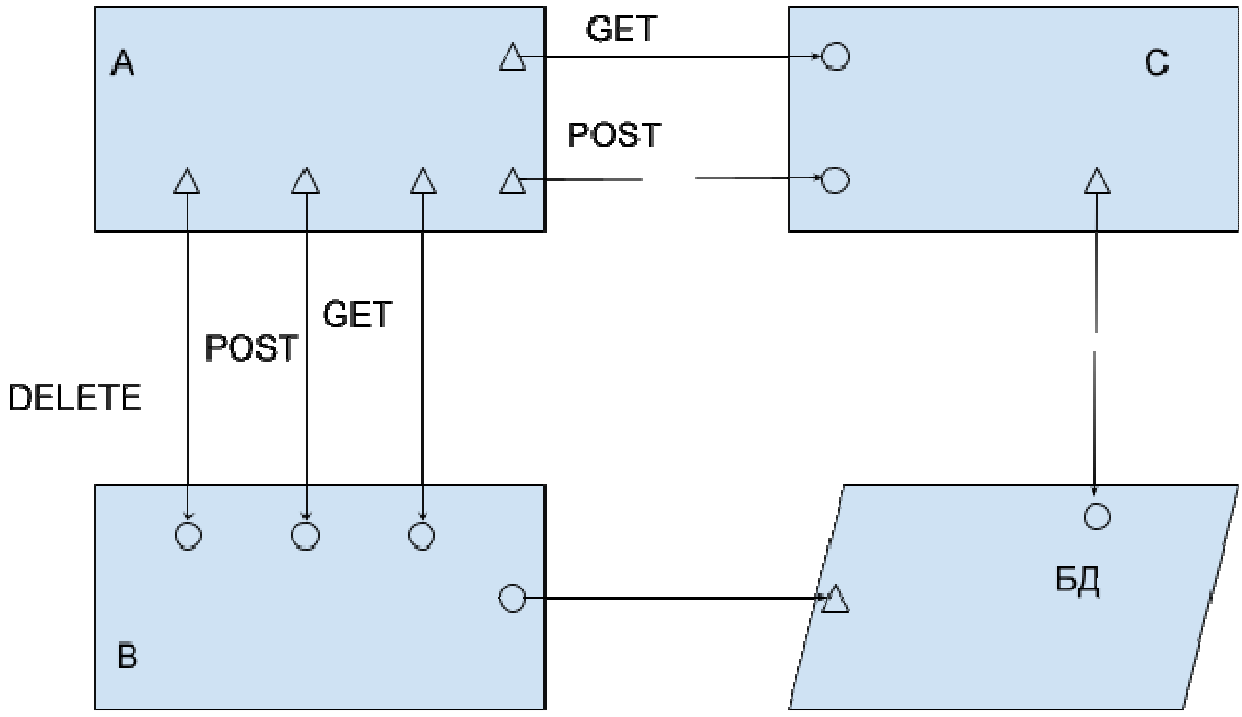


Рис. 4.6. Розподілена система №1

Сервіс А містить веб-інтерфейс користувача. При запуску усієї системи користувач бачить саме цей інтерфейс. Користувачу пропонується ввести адресу електронної скриньки. Після вводу адреси сервіс А надсилає GET запит на сервіс С, який у свою чергу перевіряє наявність даної адреси в БД. Якщо GET запит повертає успішний статус код 200, значить, користувач вже існує. Якщо отриманий статус код помилковий, сервіс А відправляє на сервіс С POST запит, який у свою чергу створить нового користувача в БД.

Після отримання успішного результату від сервісу С, сервіс А відправляє GET запит на сервіс В, який у свою чергу підтягує вміст поштової скриньки користувача. Користувачу може здійснити наступні операції з листами:

- відкрити лист (POST запит, який до того ж відправляє інформацію, чи був лист прочитаний раніше, чи ні);
- видалити лист (DELETE запит).

Після відправлення вищеописаний запитів з сервісу А на сервіс В, сервіс В робить відповідні запити у БД та повертає результат користувачу.

В даній системі можливі наступні коди відповідей від сервісу:

- 200, успішно
- 201, створено
- 204, успішно, вміст відсутній
- 400, невірний запит
- 404, не знайдено
- 500, внутрішня помилка серверу

Кожен HTTP метод може повертати деякі з вищеперечислених кодів:

- GET (отримати контент): 200, 404, 500
- POST (оновлення контенту): 200, 201, 400, 404, 500
- DELETE (видалити вказаний контент): 204, 404, 500

Напишемо тестовий план для наскрізного тестування даної системи №1:

Номер тесту	Дія	Очікуваний результат
1	Створити новий поштовий ящик	Поява листа в БД
2	Відкрити існуючий поштовий ящик	Відображений вміст поштової скриньки
3	Відкрити новий лист.	Відображено вміст листа. Лист відмічений як прочитаний у БД та на веб-інтерфейсі.
4	Перевірити правильність відображення вмісту поштової скриньки.	Кількість відображених листів на веб-інтерфейсі дорівнює кількості листів у БД.
5	Видалити лист	Лист відсутній у БД та на веб-інтерфейсі
6	Реєстрація користувача з невалідною адресою	Помилка реєстрації
7	Спроба прочитати вже видалений лист	Повідомлення про те, що лист вже видалено
8	Спроба повторного видалення листа	Повідомлення про те, що лист вже видалено

А тепер напишемо список тестів, якими потрібно покрити систему у випадку контрактного тестування.

Коли сервіс А - споживач, а сервіс С - постачальник:

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST	201, 400, 500

Як бачимо, всього у нас є 6 різних тестових випадків. Оскільки, у контрактному тестуванні кожен сервіс тестується ізольовано за допомогою сервіса-імітатора, загальна сума тестів у зв'язках "споживач – сервіс-імітатор" та "сервіс-імітатор – постачальник" буде вдвічі більшою – 12.

Наступний випадок, коли сервіс А – споживач, а сервіс В – постачальник:

Запит	Варіанти кодів відповідей
GET	200, 404, 500
DELETE	204, 404, 500
POST	200, 400, 404, 500

Кількість тестових випадків – 10. Кількість тестів згідно підходу контрактного тестування – 20. Кількість контрактних тестів, потрібних для покриття усієї системи: 20+12 = 32.

Відповідно, щоб покрити дану систему контрактними тестами, потрібно 32 тести, щоб покрити її наскрізними, потрібно 8 тестів.

Системи №2 (рис. 4.7) являє собою повноцінний поштовий сервіс з функціями реєстрацій та відновлення пароля. Скринька та профіль користувача мають власні налаштування. Повідомлення можна надсилати, приймати та керувати отриманими повідомленнями.

Система складається з наступних підсистем:

- Сервіс А. Сервіс аутентифікації, який також містить інтерфейс користувача. Інтерфейс дозволяє користувачу керувати усією системою.

- Сервіс B. Сервіс керування листами, який зчитує, або відправляє листи на поштовий сервіс C та зберігає листи користувача у базу даних.

- Сервіс C. Поштовий (транспортний) сервіс, який отримує та відправляє листи на поштовий сервер.

- Сервіс D. Сервіс(керування базоб даних), який, взаємодіючи з базою даних керує інформацією про усіх користувачів системи.

- Сервіс E. Сервіс реєстрації, який дозволяє зареєструвати нового користувача.

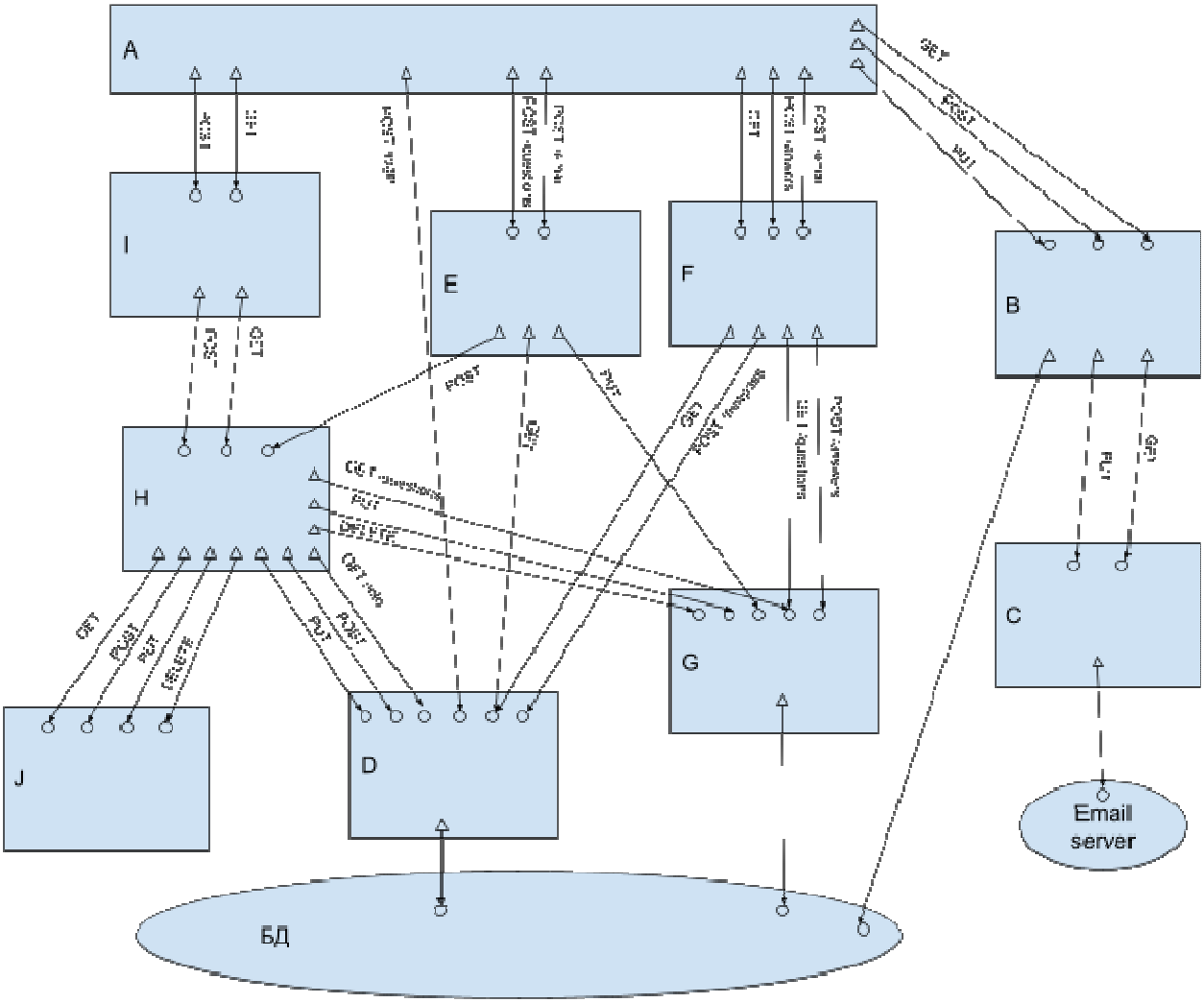


Рис. 4.7. Розподілена система №2

- Сервіс F. Сервіс відновлення паролю. Дозволяє користувачу відновити втрачений доступ до облікового запису.

- Сервіс G. Сервіс секретних запитань. Зберігає та керує секретними запитаннями користувача, які потрібні для відновлення доступу для втраченого облікового запису.

- Сервіс H. Сервіс керування користувачами. Сервіс містить функціонал для створення та редагування інформації про користувача.

- Сервіс I. Сервіс профілю користувача. Повертає користувачу інформацію про нього.

- Сервіс J. Сервіс збереження фотографій Дозволяє додавати, редагувати, видаляти фото користувача. Взаємодіє з базою даних.

Сервіс A являється найбільшим сервісом системи. Сервіс A відправляє сервісу B GET, POST та PUT запити для здійснення операцій перегляду, надсилання, керування та видалення листа.

Сервіс B відповідає за керування листами. Сервіс зберігає листи користувача у базу даних. Сервіс B у свою чергу може надсилати GET та PUT запити на сервіс C для отримання та надсилання листів. Сервіс C взаємодіє з поштовим сервером.

Коли користувач вводить свій логін та пароль, сервіс A надсилає POST запит на сервіс D, щоб перевірити наявність користувача в системі та правильність введеного логіна та паролю. Сервіс D отримує цю інформацію з базі даних.

При реєстрації нового користувача сервіс A відправляє всю надану інформацію про нового користувача у вигляді POST запиту на сервіс E. Сервіс E перевіряє існування введеної поштової адреси користувача у базі даних за допомогою GET запиту на сервіс D. Якщо поштова адреса користувача не є зайнятою, вся інформація про нового користувача відправляється POST запитом на сервіс H, який відправляє PUT запит на створення нового користувача на сервіс D. Також сервіс H відправляє завантажене фото користувача на сервіс J для подальшого збереження у базі даних. В наступному кроці користувачу пропонується обрати та дати відповідь на секретні запитання, які спочатку відправляються POST запитом з сервісу A на сервіс E, обробляються та

посилаються у вигляді PUT запиту на сервіс G, для створення та шифрування інформації про секретні запитання та відповіді у базі даних.

При відновлені втраченого паролю сервіс A надсилає POST запит з введеною поштовою адресою на сервіс F. Сервіс F надсилає GET запит на сервіс D для перевірки існування введеної поштової адреси користувача. Якщо адреса правильна, сервіс A надсилає GET запит на сервіс F для отримання секретних запитань. Сервіс F відправляє у свою чергу запит на сервіс G, який дістає секретні запитання з бази даних, дешифрує їх та повертає ці дані сервісу F. Сервіс F повертає дешифровані дані користувачу на сервіс A. Після того як користувач введе відповіді на секретні запитання, вони відправляються на сервіс F за допомогою POST запиту. Сервіс F надсилає ці дані POST запитом на сервіс G для шифрації та перевірки правильності відповідей у базі даних. Якщо відповіді на запитання введені невірно 5 разів, профіль користувача блокується.

Авторизований користувач може змінювати інформацію свого профілю. Для перегляду та внесення змін профілю користувача з сервісу A на сервіс I надсилаються GET та POST запити відповідно. Сервіс I отримує інформацію про користувача за допомогою GET запиту на сервіс H. Зміна інформації відбувається за допомогою POST запиту на сервіс H.

Сервіс H надсилає на сервіс J запити GET, POST, PUT та DELETE для перегляду, створення, заміни або видалення фотографії профілю користувача у базі даних. Також сервіс H дозволяє редагувати дані про секретні запитання за допомогою GET та PUT запитів на сервіс G. А також редагувати та створювати інформацію про користувача, відправивши POST та PUT запити на сервіс D відповідно.

В даній системі можливі наступні коди відповідей від сервісу:

- 200, успішно
- 201, створено
- 204, успішно, вміст відсутній
- 400, невірний запит
- 404, не знайдено

- 409, конфлікт даних
- 500, внутрішня помилка серверу

Кожен HTTP метод може повертати деякі з вищеперечислених кодів:

- GET (отримати контент): 200, 404, 500
- POST (оновлення контенту): 200, 201, 400, 404, 500
- PUT (створеня контенту): 200, 201, 400, 404, 409, 500
- DELETE (видалити контент): 204, 404, 500

Напишемо тестовий план для наскрізного тестування даної системи

Номер тесту	Дія	Очікуваний результат
1	Введення коректного логіна та пароля	Успішний вхід
2	Введення некоректного логіна	Повідомлення про помилку
3	Введення некоректного паролю	Повідомлення про неправильний пароль
4	Відправка листа зі сторонньої адреси	Поява листа у поштовій скринці
5	Відкрити новий лист.	Відображено вміст листа. Лист відмічений як прочитаний у БД та на веб-інтерфейсі.
6	Перевірка правильності відображення вмісту листа	Отриманий лист відповідає надісланому оригіналу
7	Перевірити правильність відображення вмісту поштової скриньки.	Кількість відображених листів на веб-інтерфейсі дорівнює кількості листів у БД.
8	Перевірити правильність	Кількість відображених листів

	відображення вмісту поштової скриньки після отримання нових та видалення старих листів.	на веб-інтерфейсі дорівнює кількості листів у БД.
9	Видалити лист	Лист відсутній у БД та на веб-інтерфейсі
10	Спроба двічі видалити один і той же лист	Повідомлення про помилку. Можлива подальна робота з програмою
11	Відправка нового листа на сторонню пошту	Лист з’явився у поштовій скриньці отримувача
12	Відправка нового листа	Лист з’явився у базі даних
13	Відправка листа без адресата	Повідомлення про помилку
14	Реєстрація нового користувача з секретними запитаннями	Новий користувач з’явився у базі даних. Секретні запитання зберігаються у базі даних в зашифрованому вигляді
15	Реєстрація користувача з вже існуючою поштовою адресою	Повідомлення про те, що адреса вже існує
16	Введення невалідних даних при реєстрації	Повідомлення про некоректні дані
17	Спроба введення невалідних відповідей на секретні запитання при реєстрації	Повідомлення про неправильне введення
18	Спроба введення однакових секретних запитань при	Повідомлення про некоректне введення

	реєстрації	
19	Перервати реєстрацію, спроба зареєструвати користувача з такою ж поштою	Реєстрація успішна
20	Відновлення акаунту з правильними відповідями на секретні запитання	Користувач може увійти у систему
21	Спроба відновлення акаунту з неіснуючою адресою	Повідомлення про неіснуючу адресу
22	Відновлення акаунту з неправильними відповідями на секретні запитання	Повідомлення про неправильні відповіді
23	Відновлення акаута з 5-ма спробами введення неправильних відповідей на секретні запитання	Користувач відмічений як заблокований у базі даних
24	Перегляд профілю користувача	Введені під час реєстрації дані та фото коректно відображаються в профілі користувача
25	Редагування профілю користувача	Нові дані відображаються у профілі та у базі даних
26	Введення невалідних даних у профіль користувача	Повідомлення про помилку
27	Видалення секретних запитань	Запитання видалені з бази даних

28	Введення нових секретних запитань	Запитання занесені в базу даних
29	Перегляд фото користувача	Фото відображається на екрані
30	Завантаження нового фото користувача	Фото відображається у профілі та у базі даних
31	Заміна фото користувача	Нове фото відображається у профілі та у базі даних
32	Видалення фото користувача	Фото не відображається у профілі користувача та не існує в базі даних

Відповідно, щоб покрити дану систему наскрізними тестами, потрібно 32 тести.

Тепер покриємо дану систему контрактними тестами.

Сервіс A - споживач, сервіс C - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST	200, 400, 500
DELETE	204, 404, 500

Сервіс B - споживач, сервіс C - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
PUT	201, 400, 500

Сервіс A - споживач, сервіс D - постачальник

Запит	Варіанти кодів відповіді
POST	200, 400, 404, 500

Сервіс A - споживач, сервіс E - постачальник

Запит	Варіанти кодів відповіді
POST (/email)	200, 400, 500
POST (/questions)	200, 400, 500

Сервіс E - споживач, сервіс D - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500

Сервіс E - споживач, сервіс G - постачальник

Запит	Варіанти кодів відповіді
PUT	200, 400, 404, 500

Сервіс E - споживач, сервіс H - постачальник

Запит	Варіанти кодів відповіді
POST	200, 400, 404, 500

Сервіс A - споживач, сервіс F - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST (/email)	200, 400, 404, 500
POST (/answers)	200, 400, 404, 500

Сервіс F - споживач, сервіс D - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST	200, 400, 404, 500

Сервіс F - споживач, сервіс G - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST	200, 400, 404, 500

Сервіс A - споживач, сервіс I - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST	200, 400, 404, 500

Сервіс I - споживач, сервіс H - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST	200, 400, 404, 500

Сервіс H - споживач, сервіс J - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST	200, 400, 404, 500
PUT	200, 201, 400, 404, 409, 500
DELETE	200, 404, 500

Сервіс H - споживач, сервіс D - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST	200, 400, 404, 500
PUT	200, 400, 404, 409, 500

Сервіс H - споживач, сервіс G - постачальник

Запит	Варіанти кодів відповіді
GET	200, 404, 500
POST	200, 400, 404, 500
DELETE	204, 404, 500

Під час експерименту проводилось два типи тестування розподіленої системи: наскрізне та контрактне. Для наскрізного тестування використовувалось програмне забезпечення Postman. У Postman потрібно було зазначити всі тестові сценарії запитів, які будуть відсилатися на розподілену систему в рамках наскрізного тестування.

Під час експерименту проводилось два типи тестування розподіленої системи: наскрізне та контрактне. Для проведення наскрізного тестування було використано фреймворк Catcher, який був розглянутий у першому розділі. Мовою програмування було обрано C#, запуск тестів відбувається за допомогою бібліотеки XUnit. Для контрактного тестування використовувався розроблений фреймворк, описаний вище.

Для проведення експерименту було обрано 4 типи розподілених систем (таблиця 4.1): розподілена база даних, розподілена система кешів, розподілені API сервіси та розподілена система інтерфейсів. Для простоти експерименту кожна розподілена система мала 5 вузлів.

Для тестування розподілених систем використовувались описані у роботі методи: удосконалений метод тестування за допомогою контрактів та удосконалений метод тестування за допомогою симуляційних тестів, а також фреймворки Jepsen, Catcher та QuickCheck, описані у розділі 1.

Для проведення тестування розподіленої системи БД можна застосувати фреймворки Jepsen, Catcher, QuickCheck та метод тестування з використанням контрактів. За результатами тестування час тестування фреймворками Catcher та QuickCheck виявився найвищим, 431 та 416 секунд відповідно. Причиною являється велика кількість тестів та їх проходження та тривалий час розгортання всієї розподіленої системи. Тестування за допомогою Jepsen потребує найменшої кількості тестів, які є достатньо швидкими. Загальний час тестування за допомогою Jepsen — 160 секунд. Найшвидший час проходження тесту показав удосконалений метод з використанням контрактів — 9 секунд. Але через більшу кількість тестів, ніж у Jepsen, та дещо довший час розгортання

тестового середовища, час тестування за допомогою контрактів виявився на 1.3% повільнішим, ніж час тестування за допомогою Jepsen.

Таблиця 4.1

Результати тестування систем розподіленої обробки даних

Тип системи	Характеристика	Jepsen	QuickCheck	Catcher	Удосконалений метод тестування	Відсоток порівняння часу тестування системи
Розподілена система БД. Кількість вузлів = 5	Кількість тестів	10	23	18	14	
	Час проходження тесту	12	15	16	9	
	Час розгортання сервісу	16	39	20	5	
	Час розгортання тестового середовища	24	32	33	31	
	Час тестування системи	**160**	**416**	**341**	**162**	-1.3%
Розподілена система кешів Кількість вузлів = 5	Кількість тестів	7	17	19	11	
	Час проходження тесту	5	12	10	4	
	Час розгортання сервісу	17	25	23	5	
	Час розгортання тестового середовища	10	23	25	14	
	Час тестування системи	**62**	**252**	**238**	**63**	-1.6%
Розподілена система API Кількість вузлів = 5	Кількість тестів	—	20	18	16	
	Час проходження тесту		4	3	2	
	Час розгортання сервісу		16	15	5	
	Час розгортання тестового середовища		9	8	35	
	Час тестування системи		**105**	**77**	**72**	6.5%

Схожі результати показало тестування розподіленої системи кешів. Фреймворки Catcher, QuickCheck виявились значно повільнішими, 238 та 252 секунди відповідною. Причиною є велика кількість тестів та час їх проходження (19 тестів по 10 секунд кожен та 17 тестів по 12 секунд кожен, відповідно).

Удосконалений метод за допомогою контрактних тестів та фреймворк Jepsen виявились значно швидшими. Для тестування системи за допомогою Jepsen було необхідно всього 7 тестів по 5 секунд кожен. На розгортання системи пішло 17 секунд, а на розгортання тестового середовища — 10 секунд. Загальний час тестування системи за допомогою фреймворку Jepsen — 62 секунди. Для тестування цієї ж розподіленої системи кешів удосконаленим методом з використанням контрактів потрібно було 11 тестів, кожен з яких проходив по 4 секунди. Незважаючи за невеликий сумарний час розгортання сервісів розподіленої системи, через більшу кількість тестів загальний час тестування розподіленої системи кешів за допомогою удосконаленого методу тестування контрактами, загальний час проходження тестів вийшов 63 секунди, що на 1.6% довше, ніж час тестування за допомогою Jepsen.

Для тестування розподіленої системи API фреймворк Jepsen застосувати неможливо. Тому тестування відбувалось з допомогою Catcher, QuickCheck та удосконаленим методом з використанням контрактів. Усі три фреймворки видали достатньо швидкий час тестування — 2-4 секунди. Удосконалений метод з використанням контрактів потребував найменшу кількість тестів — 16, найменший час розгортання сервісів — 5 секунд, та значно довший час розгортання тестового середовища — 35 секунд у порівнянні з 8 секунд для Catcher та 9 секунд для Jepsen. В результаті, час тестування розподіленої системи API фреймворками Quickcheck склала 105 секунд, Catcher — 77 секунд та тестами з використанням контрактів — 72 секунди. Удосконалений метод з використанням контрактів виявився швидшим за існуючі засоби на 6.5%.

Тестування обома методами було проведено на розподілених системах з 2, 5, 10, 20,40 та 50 вузлами. Для кожного методу було виміряно час проходження тестів та розраховано середній час, який записаний у таблицю 4.2 та зображено на рисунку 4.8.

Таблиця 4.2

Результати проходження тестів

Кількість вузлів	Catcher, с	Удосконалений метод тестування, с	%
2	40	45	-12,5
5	77	72	6,5
10	110	101	8,2
20	180	160	11,2
40	360	315	12,5
50	480	412	14,2

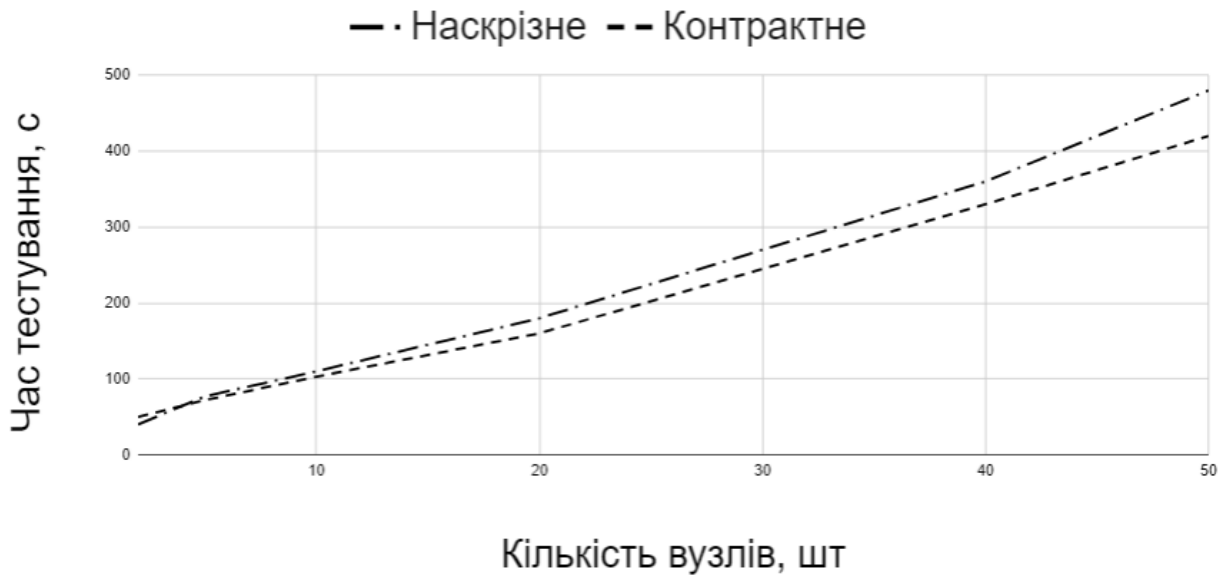


Рис. 4.8. Порівняння середнього часу тестування різними методами

При невеликій кількості вузлів наскрізне тестування сервісів проходить швидше за контрактне тестування цих же сервісів. При збільшення кількості вузлів час, який необхідний на тестування сервісів розподіленої системи контрактими тестами, стає менший за час, який необхідний для тестування цієї ж системи наскрізним методом. Це пов’язано з тим, що при виникненні

негативного результату тесту при наскрізному тестуванні потрібно заново встановлювати всю системи, а при контрактному тестуванні – тільки сервіс, для якого був негативний результат тестування.

Як показав експеримент, при визначенні середнього часу тестування при трьох та менше вузлах тестування контрактними тестами воно являється неефективним. А при збільшенні кількості вузлів час тестування зменшується.

4.3. Експериментальне визначення середнього часу проходження тестів інтерфейсу користувача з використанням симуляторів входу-виходу

Для проведення експерименту було обрано веб-сайт з розподіленою системою його інтерфейсів. Для розробки інтерфейсів використовувався фреймворк VueJs, мова програмування – TypeScript. Середовище розробки – Visual Studio Code 1.53.

Для розгортання розподіленої системи статичні файли інтерфейсу надсилаються на Amazon S3 та CloudFront.

Під час експерименту проводилось два типи тестування: наскрізне та симулятивне тестування. Для обох типів було використано веб-браузер Google Chrome.

Для взаємодії програмного коду з браузером під час наскрізного тестування використовувався фреймворк Selenium WebDriver, який дозволяє відтворювати поведінку користувача у браузері за допомогою команд з коду та був розглянутий у розділі 1. Під час наскрізного тестування розгорталися всі інтерфейси розподіленої системи, запускався браузер та запускався масив усіх тестів.

Для симуляційного тестування для роботи симулятора входу використано бібліотеку shallowMount, а для роботи симулятора виходу – бібліотеку Jest.

Таблиця 4.3

Результати тестування інтерфейсів розподіленої системи

Тип системи	Характеристика	Catcher	Удосконалений метод тестування інтерфейсів	Відсоток порівняння часу тестування системи
Розподілена система інтерфейсів Кількість вузлів = 5	Кількість тестів	12	18	
	Час проходження тесту	21	12	
	Час розгортання інтерфейсів	14	30	
	Час завантаження компонентів інтерфейсу	7	4	
	Час розгортання тестового середовища	13	15	
	Час тестування системи	**286**	**265**	7.3%

Для тестування розподіленої системи інтерфейсів, описаної у розділі 1 фреймворків, можна застосувати лише Catcher та удосконалений метод з використанням симуляційних тестів. Для тестування даної системи за допомогою Catcher потрібно було всього 12 тестів, але час проходження одного тесту дорівнював 21 секунді. Для тестування цієї ж системи за допомогою симуляційних тестів знадобилося 18 тестів по 12 секунд кожен. Оскільки під час симуляційного тестування кожен сервіс розгортається окремо, сумарний час розгортання сервісів під час виконання симуляційних тестів дорівнював 30 секунд, в той час для як для розгортання всієї системи під час тестування фреймворком Catcher знадобилося 14 секунд. Загальний час, необхідний для тестування всієї розподіленої системи інтерфейсів за допомогою Catcher, склав 286 секунд, тоді як час тестування цієї ж системи з використанням симуляційних тестів склав 265 секунд. Відповідно, удосконалений метод симуляційного тестування виявився швидшим за існуючий фреймворк Catcher на 7.3%.

Тестування обома методами було проведено на розподілених системах з 2, 5, 10, 15 та 20 вузлами. Для кожного методу було виміряно середній час

проходження тестів та розраховано середній час, який записаний у таблицю 4.4 та зображено на рисунку 4.9.

Таблиця 4.4

Результати проходження тестів

Кількість вузлів	Selenium WebDriver	Симуляційне	%
2	120	150	-25
5	286	265	7,3
10	915	830	9,3
15	1812	1556	14,2
20	3633	2930	19,4

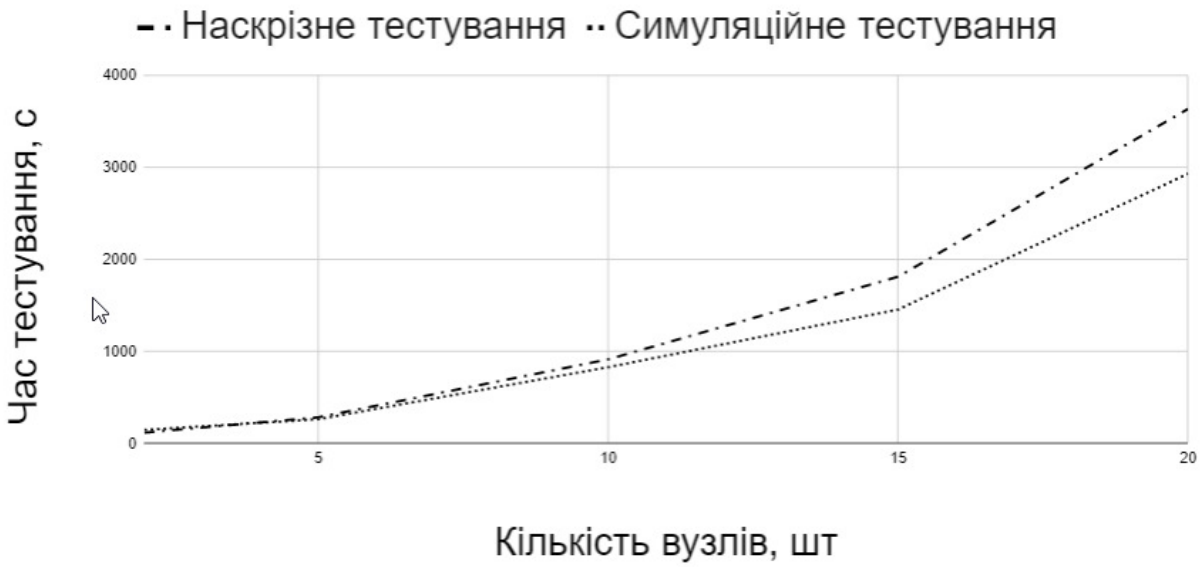


Рис. 4.9. Порівняння середнього часу тестування інтерфейсів різними методами

При двох сервісах інтерфейсу наскрізне тестування проходить швидше за стимуляційне, а при збільшенні кількості вузлів час, необхідний на тестування сервісів розподіленої системи інтерфейсів симуляційними тестами, стає менший за час, який необхідний для тестування цієї ж системи наскрізним методом.

4.4. Розробка рекомендацій по впровадженню запропонованих моделей та методів

При тестуванні розподілених систем обробки інформації, які використовуються для роботи підприємств крім традиційних завдань (технічного і програмного тестування), виникає ряд специфічних завдань, витікаючих не тільки з територіальної розподіленості структур організацій, а і з використанням різних програм для забезпечення стабільної роботи підприємств, тобто використання гетерогенних програм, що використовують гетерогенні СУБД: забезпечення оперативного управління розподіленими базами даних гетерогенного характеру, вирішення аналітичних завдань – оперативна оцінка (моніторинг) обробки запитів, забезпечення контролю над діяльністю підрозділів та служб підприємств, забезпечення надійного обміну інформацією з іншими установами та організаціями.

Ці завдання виходять за рамки традиційних завдань тестування даних в розподілених системах обробки інформації та вимагають додаткових затрат як для програмної, так і для технічної компоненти програмних систем.

Наочна область побудови системи автоматизованого тестування має складну структуру і тому на першому етапі вона розбивається на декілька задач, які надалі вирішуються окремо, але в цілому забезпечують єдину задачу:

- створення системи моніторингу подій на основі сенсорів подій в розподілених системах обробки інформації;

- створення засобів для забезпечення підвищення швидкості тестування програмного забезпечення;

- створення засобів для забезпечення підвищення швидкості розробки нових та модифікації старих тестів.

Формування та використання інформаційних ресурсів – одна з ключових проблем створення єдиного інформаційного середовища організації, що вирішується при побудові розподілених систем обробки інформації. Джерелом даних для створення тестів служить автоматизована система управління

процесами моніторингу різних систем, що надає інформацію про операції та ресурси, які використовуються при обробці інформації в розподілених системах.

Однією з основних особливостей розподілених систем обробки інформації є їх територіальна розміщеність. У структурі розробки програмного забезпечення в зв'язку з цим виділимо ієрархію території діяльності – множину віддалених об'єктів, на яких розміщені організації (рис. 4.10).

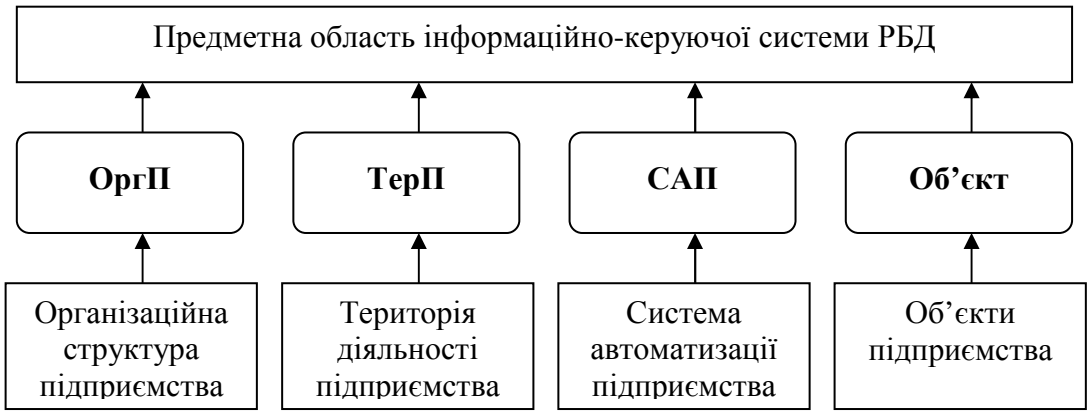


Рис. 4.10. Структура предметної області підприємства

Структура ієрархічної будови інформаційно-керуючої системи підприємства представлена наступними елементами:

- центральний сервер, що здійснює єдине керівництво в цілях оптимального розподілу і використання ресурсів;
- управління технічними, матеріальними, фінансовими і людськими ресурсами;
- локальні сервери, що розташовані у підрозділах, які здійснюють частину функцій загального управління;
- комп'ютери користувачів, які працюють з розподіленими системами обробки даних.

У організаційній структурі підприємства виділяють наступні рівні ієрархії управління: управління, відділи, служби, головні менеджери, старші фахівці, оператори, працівники, що показано на рисунку 4.11. На

підставі опитування експертів у сфері обслуговування розподілених систем і узгодження їх оцінок були виділені основні відносини в області.

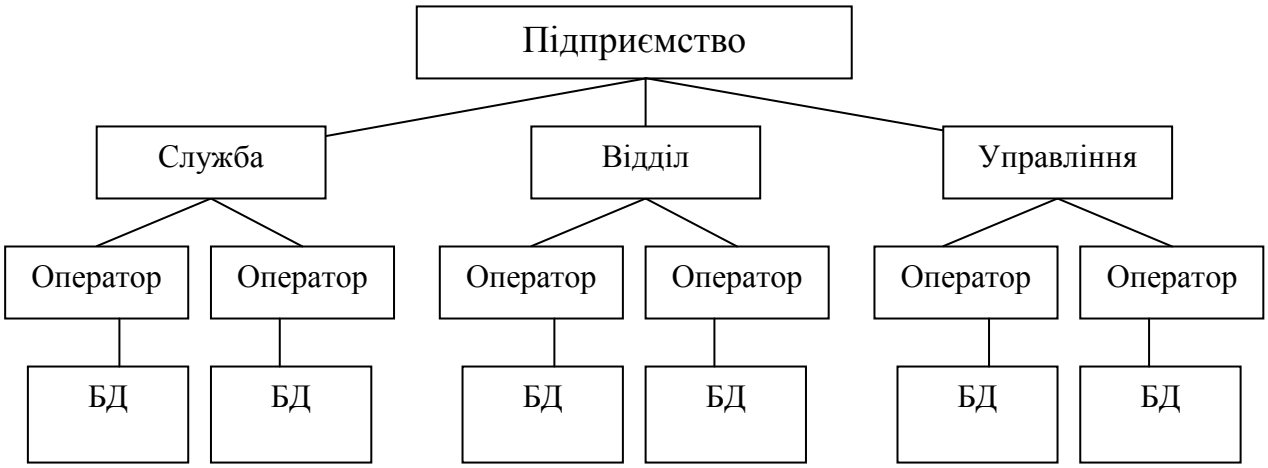


Рис. 4.11. Структура інформаційних об’єктів підприємства

Апаратне забезпечення системи автоматизації тестування можна представити як сукупність бази знань тестів та бази знань еталонних відповідей на них. Також потрібно враховувати тестування роботи програмного забезпечення з сервісами для друку та сканування, які в останній час активного розвиваються. Введення нових засобів призводить до нових видів логічних зв’язків у розподіленій системі, які також треба враховувати при тестуванні.

4.6 Висновки до розділу 4

Для проведення експериментальних досліджень моделей, методів та інформаційної технології було розроблено фреймворк для тестування сервісів розподілених систем обробки інформації за допомогою механізмів на основі використання контрактів. Під час експерименту проводилось два типи тестування розподіленої системи: наскрізне та контрактне. Для наскрізного тестування використовувався фреймворк Catcher. Запуск тестів відбувається за допомогою бібліотеки XUnit.

Розроблено інформаційну технологію та подано інформаційну систему для підвищення ефективності тестування програмного забезпечення розподілених систем.

Показано, що при невеликій кількості вузлів наскрізне тестування сервісів проходить швидше за контрактне тестування цих же сервісів. При збільшенні кількості вузлів час, який необхідний на тестування сервісів розподіленої системи контрактими тестами, стає менший за час, який необхідний для тестування цієї ж системи наскрізним методом. Це пов’язано з тим, що при виникненні негативного результату тесту при наскрізному тестуванні потрібно заново встановлювати всю системи, а при контрактному тестуванні – тільки сервіс, для якого був негативний результат тестування.

Проведено експериментальне дослідження застосування методу тестування програмного забезпечення вузлів розподіленої системи при визначенні середнього часу тестування. Для трьох та менше вузлів системи тестування контрактними тестами неефективне в порівнянні з механізмом наскрізного тестування, а в іншому випадку середній час тестування зменшується. Для 5 вузлів розподіленої системи час тестування API зменшився на 6,5%.

Показано, що при невеликій кількості вузлів наскрізне тестування інтерфейсів користувача фреймворком Selenium WebDriver проходить швидше за симуляційне тестування цих же інтерфейсів користувача. При збільшення кількості вузлів час, який необхідний на тестування сервісів розподіленої системи симуляційними тестами, стає менший за час, який необхідний для тестування цієї ж системи наскрізним методом.

Проведено експериментальне дослідження застосування методу тестування інтерфейсу користувача програмного забезпечення вузлів розподіленої системи, який показав, що при збільшенні кількості вузлів час, який необхідний на тестування сервісів розподіленої системи інтерфейсів симуляційними тестами, стає менший за час, який необхідний для тестування

цієї ж системи наскрізним методом. Для 5 інтерфейсів розподіленої системи час тестування зменшився в порівнянні з Selenium WebDriver на 7,3%.

Результати, подані в розділі, автор опублікував у працях [11, 12, 13].

ВИСНОВКИ

У дисертаційній роботі подано результати, які відповідно до поставленої мети є рішенням актуального науково-практичного завдання розроблення моделей, методів та інформаційної технології підтримки ефективного тестування програмного забезпечення.

Одержано такі нові теоретичні та практичні результати:

1. Проведено аналіз механізмів тестування програмного забезпечення систем. Показано, що під час тестування великих систем, які містять клієнтські та серверні частини, важливо перевіряти не лише роботу системи за невеликої кількості користувачів, а й за максимальних навантажень, що може спричинити непроходження тестів та збої у роботі системи. Важливою характеристикою також стає час реагування системи на запити, які до неї відправляються.

2. Модифіковано піраміду тестування Майка Кона. Показано, що для написання загальних тестів потрібно використовувати зовсім мало високорівневих наскрізних тестів, які перевіряють програмне забезпечення від початку до кінця. При цьому потрібно пильнувати, щоб механізми використання піраміди не призвели до великих затрат часу.

Результати порівняння фреймворків тестування показали, що для тестування розподілених баз даних та кешів підходять Jepsen та QuickCheck, а для тестування розподілених API та інтерфейсів краще використати Catcher. Проте Jespen – єдиний фреймворк, який дозволяє легко діагностовувати результати проходження тестів. Фреймворк Catcher є найзручнішим для написання тестів, причому не потрібно писати код, а лише сценарії.

3. Уперше розроблено модель тестування програмного забезпечення розподілених систем на основі автономного розгортання програмних компонентів, яка ґрунтується на застосуванні механізмів на основі модифікованої піраміди Майка Кона, що дозволяє підвищити ефективність тестування та зменшити кількість об’ємних тестів.

4. Удосконалено метод тестування програмного забезпечення вузлів розподіленої системи, який різниться від наявних застосуванням контрактних тестів та аналізом прогнозованого результату поведінки сервісів, що дозволяє зменшити час розгортання компонентів тестового середовища та час проходження тестів. Проведено експериментальне дослідження застосування методу тестування програмного забезпечення вузлів розподіленої системи з визначенням часу тестування. Для трьох та менше вузлів системи тестування контрактними тестами неефективне порівняно з механізмом наскрізного тестування, а в іншому випадку час тестування зменшується. Для п’яти вузлів розподіленої системи час тестування API зменшився на 6,5 %.

5. Удосконалено метод тестування інтерфейсу користувача програмного забезпечення вузлів розподіленої системи, який різниться від наявних підходом до симуляції його роботи, що дозволяє проводити тестування окремих компонентів інтерфейсу системи. Проведено експериментальне дослідження застосування методу тестування інтерфейсу користувача програмного забезпечення вузлів розподіленої системи, який показав, що зі збільшенням кількості вузлів час, потрібний на тестування сервісів розподіленої системи інтерфейсів симуляційними тестами, стає меншим за час, потрібний для тестування цієї ж системи наскрізним методом. Для п’яти сервісів інтерфейсів розподіленої системи час тестування зменшився порівняно з фреймворком Catcher на 7,3 %.

6. Розроблено інформаційну технологію, що реалізує подані в роботі моделі та методи. Інформаційну систему, створену за такою інформаційною технологією, було використано для реалізації експериментальних досліджень – вона підвищує ефективність тестування програмного забезпечення розподілених систем обробки інформації.

7. Розроблені моделі, методи та інформаційна технологія підвищення ефективності тестування програмного забезпечення розподілених систем обробки інформації апробовані, застосовані та впроваджені під час розробки інформаційних систем у ТОВ «Нафтогазбудінформатика» та у

конструкторському бюро інформаційних систем КПІ ім. Ігоря Сікорського для вирішення завдання зменшення часу тестування програмного забезпечення, що підтверджено відповідними актами впровадження та довідкою про застосування результатів роботи у навчальному процесі кафедри технічної кібернетики Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Mukhin V., Kornaga Ya., Mostovyi Y., Bazaka Y. A Model For Events Monitoring Heterogeneous Distributed Databases Based on Vector-matrix Operations. The Far East Journal of Electronics and Communications, 2016. Vol. 16, Issue 3. P. 645 – 656. (This journal is indexed in Elsevier Scopus).

2. Zhenbing H., Mukhin V., Kornaga Ya., Herasymenko O., Bazaka Y. The scheduler for the grid system based on the parameters monitoring of the computer components. Eastern European Journal of Enterprise Technologies, 2017. № 1 (2-85). P. 31 – 39. (This journal is indexed in Elsevier Scopus).

3. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Метод автоматизації тестування розподіленої системи з використанням контрактів. Sciences of Europe, 2020. № 55. С. 45 – 49.

4. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Метод тестування інтерфейсу користувача з використанням імітаторів. The scientific heritage, 2020. № 51. С. 54 – 57.

5. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Method of Protecting Software Code from Reengineering Using Virtual Machines. Sciences of Europe, 2020. № 58. С. 59 – 62.

6. Корнага Я.І., Базака Ю.А., Базалій М.Ю. Захист програмного забезпечення за допомогою заплутуючих перетворень. The scientific heritage, 2020. № 54. С. 72 – 75.

7. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Використання баз даних тамов програмування у високоточних обчисленнях чисел із плаваючою точкою великої розрядності. Вчені записки Таврійського національного університету імені В.І. Вернадського, 2020. Том 31(70), №5. С. 82 – 88.

8. Mukhin V., Kornaga Ya., Yakovleva A., Herasymenko O., Bazaca Yu., Bazaliy M. The model for distributed systems testing based on the control services. XXIV Міжнародна конференція з Автоматичного Управління “АВТОМАТИКА

– 2017”. 13 – 15 вересня 2017 р., м. Київ., 2017. С. 45 – 46.

9. Мухін В.Є., Корнага Я.І., Яковлєва А.П., Базалій М.М., Базака Ю.А. Модифікований метод обфускації програмного кода за допомогою вставки інструкцій. Міжнародна наукова конференція “Інтелектуальні системи прийняття рішень та проблеми обчислювального інтелекту” (ISDMCI’2018), 21 – 27 травня 2018 р., с. Залізний порт. – Херсон: Видавництво ПП Вишемирський, 2018.– С. 67 – 68.

10. Mukhin V., Kornaga Ya., Bazaka Yu., Bazaliy M., Yakovleva A. Modified Method of Software Testing for Distributed Computer System. 2018 IEEE First International Conference on System Analysis & Intelligent Computing (SAIC). 8 – 12 October 2018, Kiev, Ukraine, 2018. P. 1 – 4. (This conference is indexed in Elsevier Scopus).

11. Mukhin V.Ye., Kornaga Ya.I., Abdullayev S.H., Gerasymenko O.Yu., Bazaka Yu.A. Method of the reserve resources determination for the distributed computer systems with the network-centric resource control. 6th International Conference on Control and Optimization with Industrial Applications (COIA). 11 – 13 July 2018. Baku, Azerbaijan, 2018. P. 226 – 231. (This conference is indexed in Web of Science).

12. Mukhin V., Vyshnivskyi V., Kornaga Ya., Herasymenko O., Bazaka Yu., Bazaliy M. Study of the functioning of the distributed computer system with a resource control mechanism based on a network-centric approach. 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2019. 18 – 21 September 2019, Metz, France. – 2019. P. 100 – 105. (This conference is indexed in Elsevier Scopus, Web of Science).

13. Mukhin V., Kornaga Y., Tkach M., Herasymenko O., Bazaka Y., Mukhin O. Subtask Prioritization on Workflow Execution in Distributed Wireless Computer System with Network-Centric Approach to Resource Control. 5th IEEE International Symposium on Smart and Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems,

IDAACS-SWS 2020, 17 September 2020, Dortmund, Germany. – 2020. (This conference is indexed in Elsevier Scopus, Web of Science).

14. Азарова А. О., Ткачук Л. М., Нікіфорова Л. О., Шиян А. А., Хошаба О. М. Публічне управління та адміністрування в контексті захисту його інформаційного простору. Вісник ЖДТУ, 2019. № 2. С. 149–155.

15. Артюхова А.С. Исследование зависимости оптимальности (эффективности) тест-кейса от различных показателей. Известия Южного федерального университета. Технические науки, 2018. № 4 (198). С.210–223.

16. Балыбердин В. А., Белевцев А. М., Степанов О. А. Анализ основных процессов обеспечения надежности программных средств АСУ специального назначения. Известия ЮФУ. Технические науки, 2015. № 3 (164). С. 62–70.

17. Балыбердин В. А., Белевцев А. М., Степанов О. А. О тестировании программных средств мобильных АСУ специального назначения. Известия Южного федерального университета. Технические науки, 2018. № 3 (197). С. 209–219.

18. Барабан М. В., Довгалець К. С., Щиров О. С. Розробка програмного забезпечення для знаходження шляху між двома точками за допомогою google maps. Вісник Вінницького політехнічного інституту. 2018. № 6 С. 90–94.

19. Барабаш О. В., Куліковська Ю. А. Аналіз експлуатаційних проблем розподілених систем Наукові записки Українського науково-дослідного інституту зв'язку, 2015. №4(38) С. 86–94.

20. Барабаш О.В.,. Бодров С.В., Мусієнко А.П. Методика накопичення діагностичної інформації в системах інтелектуального відеоконтролю. Системи управління, навігації та зв'язку, 2015. Вип. 1 (33). С. 118 – 121.

21. Баранов Г.Л., Комісаренко О.С. Технологія інтеграції гетерогенних процесів моделювання формотворення матеріалів для майбутніх транспортних систем. Серія «Технічні науки», 2018. Вип. 1 (40). С. 24–33.

22. Безуглий Д. Г. Маркетингові стратегії при розробці та реалізації проекту. Вісник НТУ «ХПІ». Серія: Стратегічне управління, управління портфелями, програмами та проектами, 2015. № 1 (1110). С. 34–38.

23. Березко О., Ковалик Л. Аналіз алгоритму двостороннього сліпого рецензування в контексті впровадження принципів відкритої науки. Вісник Національного університету “Львівська політехніка”. Серія: Інформатизація вищого навчального закладу, 2018. Вип. 903. С. 58–66.

24. Білощицький А.О., Кучанський О.Ю., Безмогоричний Д.М., Пида С.В., Кузьомко А.С. Формування концепцій побудови інфраструктури GameHub в українських університетах. Управління розвитком складних систем, 2016. № 26. С. 163–170.

25. Богдан И.В. Верификация моделей объектно-ориентированных программ: проверка на непротиворечивость и согласованность. Технические науки и технологии, 2017. № 2(8). С. 110–116.

26. Бойко Н. І. Перспективні технології дослідження великих даних у розподілених інформаційних системах. Радіоелектроніка, інформатика, управління, 2017. №4(43). С.66–76.

27. Болдуєв М.В. Розвиток методичного забезпечення проведення комп’ютерного аудиту. Держава та регіони, серія: Економіка та підприємництво, 2016. №3 (90). С. 35–39.

28. Бородина Е.А., Алёшин С.П., Гафияк А.М., Куприенко С.В. Тестирование программного обеспечения и его составляющие. Инновационная наука, образование, производство и транспорт: Техника и технологии, 2018. С. 196-203.

29. Будасов І.О. Багатоканальна доставка даних, як основний напрямок оптимізації доставки інформаційних масивів даних в комп'ютерних мережах. «Young Scientist», 2015. № 8 (23), Part 2. С. 11–14.

30. Бурков В. Н., Бушуев С. Д., Возный А. М. Управление ресурсами распределенных проектов и программ. Монография, 2015. С. 386.

31. Буров Є., Микіч Х., Верес О. Система підтримки ситуаційної обізнаності у процесі тестування програмного забезпечення. Вісник Національного університету "Львівська політехніка". Інформаційні системи та мережі, 2020. №7. С. 59–69.

32. Буров, Є. В., Микіч, Х. І., Верес, О. М., & Литвин, В. В. Система ідентифікації проблемних ситуацій тестування програмного забезпечення. Вісник НУ “Львівська політехніка”. Серія : “Інформаційні системи та мережі”,2019. №5, С.78–89.

33. Бурый А. С., Балванович А. В. Научные коммуникации как распределенные информационные структуры. Правовая информатика, 2020. №2. С.17–27.

34. Бушуєв С., Мінаєва Т. Управління програмою розвитку для міжнародних компаній у турбулентному середовищі. Вісник НТУ «ХПІ». Серія: Стратегічне управління, управління портфелями, програмами та проектами, 2015. №1(1110). С. 3–11.

35. Величко О. М.,. Грабовський О. В, Гордієнко Т. Б. Особливості застосування v-моделі при розробленні та оцінюванні якості програмного забезпечення засобів вимірювальної техніки, Збірник наукових праць Одеської державної академії технічного регулювання та якості, 2019. Вип. 1 (14). С. 6–11.

36. Величко, О. М.; Грабовський, О. В.; Гордієнко, Т. Б. Особливості застосування спіральної моделі для випробувань програмного забезпечення засобів вимірювальної техніки. Збірник наукових праць Одеської державної академії технічного регулювання та якості, 2019. С.42–49.

37. Возний О. М., Кошкін К. В., Книрік Н. Р. Імітаційне моделювання ІТ-проектів на основі мереж Петрі. Вісник НТУ «ХПІ». Серія: Стратегічне управління, управління портфелями, програмами та проектами, 2015. № 1 (1110). С. 24–28.

38. Гаращук О., Куценко В. Інформаційно-інноваційні технології – дієвий механізм та інструмент управління фундаменталізацією знань. Інноваційні комп'ютерні технології у вищій школі: матеріали 8-ої Науково-практичної конференції, 2016. С.52–57.

39. Герасименко І. Досвід співпраці ВНЗ з ІТ-компаніями. Інноваційні комп'ютерні технології у вищій школі : матеріали 8-ої Науково-практичної конференції, 2016. С.57–61.

40. Глушенкова А.А. Структура інноваційного потенціалу підприємств сфери телекомунікацій та інформатизації. Економіка. Менеджмент. Бізнес, 2016. Вип.4(18). С. 100–107.

41. Гожий В. Нечіткий когнітивний аналіз ризиків при тестуванні програмного забезпечення. Вісник Національного університету "Львівська політехніка", 2015. №826. С.372–379.

42. Гонтар Ю. М. Розробка розподіленої системи обробки бізнес-інформації з використанням агентного підходу. Системи обробки інформації. Information processing systems, 2016. Вип. 4 (141). С. 137–142.

43. Гончаренко С.Г., Соменко О.О. Захист електронних даних в інформаційних системах. І міжнародної науково-практичної конференції «Актуальні питання права та соціально-економічних відносин», 2018. С. 170.

44. Гречко В., Бабенко Т., Мирутенко Л. Безпечне програмне забезпечення, що розробляє рекомендації. Кібербезпека: освіта, наука, техніка, 2019. Вип. 2, вип. 6. С. 82-93.

45. Гук О.М., Чередниченко О.Ю., Штонда Р. М., Диба І.О. Дії в кіберпросторі під час підготовки та ведення мережецентричної війни. Сучасні інформаційні технології у сфері безпеки та оборони, 2017. No 2. С. 107–111.

46. Гура О. Підготовка майбутніх інженерів-програмістів до тестування програмного забезпечення в умовах неформальної освіти: проблеми та шляхи успішної реалізації. Педагогічні науки: теорія, історія, інноваційні технології, 2020. № 7. С. 55–62.

47. Данилина Г.В. Уровни взаимодействия и конфликтного управления в защищенных информационных системах с псевдосервисами. Телекомунікаційні та інформаційні технології, 2016. №3. С. 48–54.

48. Даниліна Г. В., Андрусевич Н. В. Псевдосервіси та теорія конфлікту в стохастичному управлінні інформаційними мережами. ІІІ-я Міжнародна конференція «Виробництво & Мехатронні Системи 2019», 2019. С. 106 – 109.

49. Даниліна Г.В., Жукова Л.Л., Андрусевич Н.В., Цвіркун С.Л., Стохастичне керування параметрами уразливостей з використанням

ескалаційних пасток. Вісник Криворізького національного університету,2019. Вип. 48. С. 114–121.

50. Дмитренко Т.А., Деркач Т.М., Рогачов О.В. Розробка программного забезпечення через тестування – test-driven development. Новітні інформаційні системи та технології - Modern information system and technologies , 2015. № 2.

51. Дубовой В.М., Пилипенко І.В. Моделювання процесу тестування програмного забезпечення як розгалужено-циклічного технологічного процесу. Автоматизація технологічних і бізнес-процесів, 2015. Том 7. № 4. С.55 – 64.

52. Дубовой В.М., Пилипенко І.В. Побудова імітаційної моделі процесу тестування програмного забезпечення в пакеті Simulink. Журнал інженерних наук, 2016. Вип.3, №2. С. Н1–Н6.

53. Єгорова О.В., Бичок В. Програмні засоби для тестування програмного забезпечення. Молодий вчений, 2019. № 11 (75). С. 680-684.

54. Жирова Т.О., Котенко Н.О. Застосування Scrum при розробці та тестуванні Web-додатків. Тези доповідей п’ятої міжнародної науково-практичної конференції «Управління розвитком технологій». Тема: Інформаційні технології розвитку освіти, 2018. С. 69 – 71.

55. Жирова Т.О., Котенко Н.О. Проблеми тестування інтерфейсу Web-додатків. Збірник матеріалів IX Міжнародної конференції молодих вчених «Молоді вчені 2018 – від теорії до практики», 2018. С. 184 – 188.

56. Зибін В. І., Лютенко І. В. Проектування інформаційного забезпечення для оцінки якості пз вбудованих систем. Вісник Національного технічного університету «ХПІ». Серія: Системний аналіз, управління та інформаційні технології, 2020. №1 (3) С. 124–130.

57. Ісаков О. С., Чередніченко О. Ю., Мозгін В. В., Янголенко О. В. Дослідження моделей процесів тестування зручності використання програмних продуктів. Вісник Національного технічного університету «ХПІ», 2018. №2 (1278). С. 73–80.

58. Калбертсон, Кобб Б. Быстрое тестирование, 2017. С.374.

59. Кальніченко О. В., Морозов В. В., Хрутьба А. С. Використання антисипативного управління проектами при створенні розподілених інформаційних систем. Вісник Національного технічного університету "ХПІ". Серія: Стратегічне управління, управління портфелями, програмами та проектами, 2018. № 2 (1278). С. 15–21.

60. Ковтун О. В., Медведєва С. О. Software testing types. Матеріали XLV Науково-технічної конференції ВНТУ, 2016. С. 63–64.

61. Кононенко І. В., Харазій А. В. Розробка програмного забезпечення для багатокритеріальної оптимізації змісту проекту за допомогою методу поступок. Вісник НТУ «ХПІ». Серія: Стратегічне управління, управління портфелями, програмами та проектами, 2015. № 1 (1110). С. 11–24.

62. Конюхов С., Храпач О. Огляд основних понять інженерії якості програмного забезпечення. Інформаційно-комунікаційні технології навчання, 2016. С. 83–88.

63. Коркошко А.В., Черних О.П. Исследование методов построения приложений электронной коммерции с высокой скоростью и надёжностью. Матеріали IX Університетської науково-практичної студентської конференції магістрів НТУ "ХПІ", 2015. С. 74.

64. Коркошко А.В., Черних О.П. Розробка програмного модулю для підвищення продуктивності взаємодії між кластерами сервера. Матеріали XXIV Міжнародної науково-практичної конференції "Інформаційні технології: наука, техніка, технологія, освіта, здоров’я", 2016. Ч. IV. С.148.

65. Косенко В. В. Принципи і структура методології ризик-адаптивного управління параметрами інформаційно-телекомунікаційних мереж систем критичного застосування. Сучасний стан наукових досліджень та технологій в промисловості, 2017. № 1 (1). С. 46–52.

66. Котенко Н. О., Жирова Т. О., Кулеба М. Б. Дослідження особливостей тестування мобільних додатків. Управління розвитком складних систем. Київ 2020. Вип. 41, С.55–60.

67. Котляров В.П. Основы тестирования программного обеспечения: учебное пособие, 2016. С.348.

68. Кравчук С.О. Проблеми автоматизації тестування програмного забезпечення. Актуальні задачі сучасних технологій: матеріали VII міжнар. наук.-техн. конф. мол. учен. та студ., 2018. С. 95.

69. Круглик В.С., Осадчий В.В. Аналіз змісту професійної підготовки інженерів-програмістів. Проблеми інженерно-педагогічної освіти, 2016. Вип. 52-53. С. 101-110.

70. Кузьмінська Н. Л. Прикладне програмне забезпечення. Конспект лекцій, 2020. С.89.

71. Куликов С. С. Тестирование программного обеспечения: учебное пособие, 2019. С.276.

72. Куліков С. Тестирование программного обеспечениюю Базовый курс. EPAM Systems, 2017. С. 296.

73. Купріянов Є. Тлумачний словник іспанської мови як інструмент для лінгвістичних досліджень. Наукові записки, 2019. №173. С. 767–773.

74. Литвинов В. В., Богдан И. В. Автоматизированная система верификации моделей объектно-ориентированного программного обеспечения. ВісникЧернігівського державного технологічного університету. Серія «Технічні науки», 2015. № 1(77). С. 83–90.

75. Ліщина Н.М., Ліщина В.О., Повстяна Ю.С. Підходи до підготовки фахівців з розробки та тестування програмного забезпечення у вищих навчальних закладах. Науковий журнал "Комп’ютерно-інтегровані технології: освіта, наука, виробництво", 2017. Випуск №27. С. 130–134.

76. Лысенко И. А., Смирнов А. А. Информационная технология проектирования тестовых наборов на основе требований к программному обеспечению. Системи управління, навігації та зв'язку, 2017. Вип. 4(44). С. 112–115.

77. Мелешко Ю. Методи оцінки якості роботи рекомендаційних систем. Системи управління, навігації та зв'язку. Збірник наукових праць, 2018. Т. 5 (51). С. 92– 97.

78. Мелкозерова О., Рассомахін, С. Тестування продуктивності програмного забезпечення. Вісник Харківського національного університету імені В.Н. Каразіна, серія «Математичне моделювання. Інформаційні технології. Автоматизовані системи управління», 2020. №45. С. 56–66.

79. Мелкозерова О., Нарежний О., Малахов С. Аналіз інструментів для автоматизованого тестування програмного забезпечення. Комп'ютерні науки та кібербезпека. Міжнародний електронний науково-теоретичний журнал, 2019. №1. С. 75 – 84.

80. Мелкозьорова О., Гайкова В., Малахов С. Опис схеми API тестування програмного забезпечення. Комп'ютерні науки та кібербезпека, 2020. № 4. С.38–45.

81. Микусь С. А. Стратегія побудови функціонально стійких інформаційно- телекомунікаційних систем. Сучасні інформаційні технології у сфері безпеки та оборони, 2017. № 2 (29) С. 42–45.

82. Мислович М.В., Сисак Р.М., Остапчук Л.Б., Гижко Ю.І, Герцик С.М. Алгоритми функціонування та програмне забезпечення багаторівневої системи моніторингу стану та технічного діагностування обладнання об'єктів електроенергетики. Технічна електродинаміка, 2016. № 4. С. 86–88.

83. Мищенко Е. Ю., Соколов А. Н. Алгоритмы реализации методов обезличивания персональных данных в распределенных информационных системах. Доклады Томского государственного университета систем управления и радиоэлектроники, 2019. Vol. 22, №1. P. 66–70.

84. Міхалевська Г.І., Міхалевський В. Ц. Окремі питання комунікаційного середовища як чинник розвитку інформаційного суспільства. Вісник Хмельницького національного університету. Технічні науки, 2018. № 3. С. 264–269.

85. Мнушка О. В., Савченко В. М. Формування та керування командою розробників програмного забезпечення. Вісник НТУ "ХПІ". Серія: Інформатика та моделювання,2020. №1 (3). С. 99 – 112.

86. Муравецький С. А., Крамський С. О. Планування процесів забезпечення якості у великих та географічно розподілених гібридних ІТ–проектах. Вісник Нац. техн. ун-ту "ХПІ"Стратегічне управління, управління портфелями, програмами та проектами, 2016. № 1 (1173). С. 106–109.

87. Наумук О.В. Стан та перспективи впровадження засобів віртуалізації у процес вивчення дисципліни «Адміністрування комп'ютерних мереж». Ukrainian Journal of Educational Studies and Information Technology, 2016. Т. 3, №1. С. 49-54.

88. Новикова К.В., Люта М.В., Розломій І.О. Дослідження методу тестування програмного забезпечення «білий ящик». «Молодий вчений», 2017. № 9 (49). С.470 – 473.

89. Огнєвий О. В. , Огнева А. М. , Коваль В. А. , Присяжнюк В. В. Особливості управління інформаційними ресурсами в корпоративних інформаційних системах. Вісник Хмельницького національного університету. Технічні науки, 2018. №5 С.17–20.

90. Пех П.А., Тихомиров В.В. Методи формування системних вимог до А3 для громіздких 3d програм. Комп'ютерно-інтегровані технології: освіта, наука, виробництво, 2018. Вип. №30-31. С. 120–125.

91. Писаренко А. Метрики та критерії тестування програмного забезпечення. VIII Всеукраїнська студентська науково - технічна конференція "Природничі та гуманітарні науки. Актуальні питання", 2015. С. 118–119.

92. Поліщук Л.І. Інформаційна технологія автоматизації проектування та тестування об'єктно – орієнтованого програмного забезпечення. Автоматика та комп'ютерно-інтегровані технології у промисловості, телекомунікаціях, енергетиці та транспорті, 2017. С. 204–206.

93. Постіл С. Д. CASE-технології. Міждисциплінарне інформаційне моделювання: навч. посіб. Ун-т ДФС України, 2018. С.304.

94. Протасов В. И., Потапова З. Е. Методика кардинального снижения вероятности принятия ошибочных решений в системах коллективного интеллекта. Современные информационные технологии и ИТ-образование, 2019. Т. 15, №3. С.588–601.

95. Ромашкина П. П. Использование термина облачные технологии в правовом поле. E-Scio, 2020. №3 (42). С.361–365.

96. Саланда І.П. Аналіз псевдорегулярних структур розподілених інформаційних систем за показником функціональної стійкості. Системи обробки інформації, 2015. №11 (136) С. 63–67.

97. Самойленко Е.С., Проніна О.І. Моделювання нечіткого виведення для перевірки якості програмного продукту. Науковий журнал "Комп'ютерно-інтегровані технології: освіта, наука, виробництво", 2019. Випуск №36. С. 63–68.

98. Семахин А. М. Методы верификации и оценки качества программного обеспечения: учебное пособие, 2018. С.150.

99. Сергієнко І.В. Математичне та програмне моделювання складних систем з використанням суперкомп'ютерних технологій. Вісник Національної академії наук України, 2018. № 3. С. 39–48.

100. Скарга-Бандурова І.С., Коваленко Я.П. Технологічні аспекти статичного аналізу коду програмного забезпечення систем залізничної автоматизації. Радіоелектронні і комп'ютерні системи, 2016. Вип. 6. С. 94–100.

101. Скиба О.В., Руденко О.В., Доманов И.А., Троцик С.Н., Кравченко В.С. Формування концепції тестування спеціального програмного забезпечення для збройних сил України. Збірник наукових праць ВІТІ, 2020. № 2. С. 98–107.

102. Скиба, О., Троцик, С. Застосування техніки негативного тестування спеціального програмного забезпечення при випробуванні зразків озброєння та військової техніки. Наукові праці Державного науково-дослідного інституту випробувань і сертифікації озброєння та військової техніки, 2020. (4). С.118-128.

103. Стрюк А.М. Становлення та розвиток інженерії програмного забезпечення як галузі знань. Інформаційні технології в освіті, 2018. № 4(37). С. 103–135.

104. Сугоняк І. І., Ковальчук А. М., Данильченко А. О. Професійна сертифікація як засіб підвищення рівня практичної підготовки студентів ІТ-спеціальностей. Інформаційні технології і засоби навчання, 2016. Т. 53, вип. 3. С.109-122.

105. Сулим, А., Сіора О., Мельник О., Хозя П., Третяк Е. Розробка алгоритмічного та програмного забезпечення для визначення раціонального режиму ведення поїзда метрополітену. Вісник східноукраїнського національного університету імені Володимира Даля, 2020. Вип. 5(261). С. 57–66.

106. Табунщик Г. В., Каплиенко Т. И., Шитикова Е. В. Модель верификации систем с ограниченными ресурсами. Радіоелектроніка, інформатика, управління, 2017. №4(43). С.162–167.

107. Теремінко Л.Г. Особливості професійної діяльності фахівців з інженерії програмного забезпечення. Збірник наукових праць «Педагогічні науки», 2018. Випуск 80, Т. 3. С. 217-223.

108. Теремінко Л.Г. Професійна мобільність фахівця з інженерії програмного забезпечення: дефінітивний аналіз. Вісник Національного авіаційного університету. Серія: Педагогіка, Психологія, 2016. Вип. 9. С. 160–166.

109. Теремінко Л.Г. Формування готовності до професійної мобільності як актуальна проблема професійної підготовки майбутніх фахівців з інженерії програмного забезпечення. Вісник Національного авіаційного університету. Серія: Педагогіка, Психологія, 2017. В. 10. С. 139–145.

110. Тіменко, А. В. Щодо аспектів специфікації і перевірки протоколів інтернету речей On the aspects of iot protocols specification and checking Shipbuilding & Marine Infrastructure, 2019. № 2 (12). С. 35–41.

111. Трофімук А. В., Федін С. С. Оцінка якості програмного забезпечення за показниками надійності. Наукові розробки молоді на сучасному етапі, 2018. Т. 2. С. 369 – 370.

112. Троян А.М., Ю. Б. Моденов Доцільність автоматизованого тестування для забезпечення якості програмних продуктів. Проблеми інформатизації та управління, 2017. Том 1, № 57-58. С. 86 – 89.

113. Федасюк Д. В., Яковина В. С., Сердюк П. В., Нитребич О. О. Метод побудови сценаріїв тестування програмного забезпечення на основі аналізу його змінних. Інформаційні технології та комп'ютерна інженерія, 2014. № 2 (30). С. 50–58.

114. Харди Б., Филлипс Б. Android программирование для профессионалов. СПб., Питер Пресс, 2016.

115. Харитонов Д.А. Моделювання організаційних хвороб в управлінні проектами. Вісник НТУ «ХПІ». Серія: Стратегічне управління, управління портфелями, програмами та проектами, 2015. № 1 (1110). С. 39–54.

116. Чернега В.М. Математична модель процесу обслуговування замовлень в розподіленій базі даних інформаційної системи в умовах впливу кібератак. Сучасні інформаційні технології у сфері безпеки та оборони, 2016. № 1 (25) С.124–129.

117. Черних О.П., Коркошко А.В. Розробка програмного забезпечення для підвищення продуктивності взаємодії між кластерами сервера. Вісник Національного технічного університету "ХПІ", 2016, № 44 (1216) С. 171–181.

118. Чухрай Н.І., Щербата Т.С. Співробітництво між підприємствами-виробниками інформаційно-технологічного продукту та ВНЗ. Маркетинг і менеджмент інновацій, 2016. № 3. С. 161–169.

119. Шапорин В. О., Тишин П. М., Шапорин Р. О. Лингвистическая оценка активов сложной компьютерной системы для анализа рисков информационной безопасности. Электротехнические и компьютерные системы, 2015. № 18. С. 28–32.

120. Шаров С. В., Левада В. Р. Напрямки використання інформаційних систем у навчальному процесі. Інформаційні технології в освіті та науці, 2017. №9. С.161–165.

121. Шевчук І. Б., Яцев В. В. Розробка програмного забезпечення для аналізу діяльності банків на ринку валютно-обмінних операцій. Modern Economics, 2018. № 8. С.218–226.

122. Шедько В. В. Тестирование, верификация и аттестация программного обеспечения. Метод. рекомендации, 2020. 30, [1] С.28–30.

123. Яшина К.В., Ялова К.М. Розробка програмного забезпечення для інтенсифікації процесів трансферу знань. Комп'ютерні науки та інформаційні технології, 2019. Т. 2 № 35. С. 87–92.

124. Agile Processes in Software Engineering and Extreme Programming. Lecture Notes in Business Information Processing: Springer International Publishing, 2018. № 314. С.314.

125. Akour M., Falah B., Al-Zyoud A. A., Bouriat S., Alemerien K. Mobile Software Testing: Thoughts, Strategies, Challenges, and Experimental Study. International Journal of Advanced Computer Science and Applications, 2016. Vol. 7. №6. P. 12 – 19.

126. Almeida D. R., Machado P.D.L., Wilkerson L. A. Testing tools for Android context-aware applications: a systematic mapping. Journal of the Brazilian Computer Society, 2019. №12.

127. Atamanyuk I., Kondratenko Y. Computer's Analysis Method and ReliabilityAssessment of fault-tolerance Operation ofInformation Systems. Intern. Conf. ICTERI, 2015. Vol.1356. P.507–522.

128. Boyko N. I. Advanced technologies of big data research in distributed information systems. Radio Electronics, Computer Science, Control, 2018. №4. С.66–76.

129. Chernov, K., Yeromin, Y., Mariia, P., Oleksiy, S., & Kotukh, Y. Automated software vulnerability testing using in-depth training methods. Computer Science and Cybersecurity, 2019. (4). P.36-42.

130. Chohan A. H., Affandi H. M., Awad J., Che-Ani A. I. A Methodology to Develop a Mobile Application Model to Appraise Housing Design Quality. International Journal of Interactive Mobile Technologies, 2017. Vol. 11. N6. P. 4 – 17.

131. Emiratli A., Marchuk M., Osadcha K. The problem of forming skills from academic writing for future programmers. Ukrainian Journal of Educational Studies and Information Technology, 2017. V.5, N.3. P. 28–36.

132. Garousia V., Feldererbe M., Karapıçakc Ç. M., Yılmazd U. Testing embedded software: A survey of the literature. Information and Software Technology, 2018. Vol. 104. P.14–45.

133. Hahanov V., Litvinova E., Chumachenko S. Maevsky D., Bojko A. Cyber PhysicalComputing for IoT-driven Services. Publ. Springer, 2018. №79. P.6.

134. Hill M. D. et al. On the Spectre and Meltdown Processor Security Vulnerabilities. IEEE Micro, 2019. T. 39. No. 2. C. 9–19.

135. Kiv A. E., Semerikov S. O., Soloviev V. N., Striuk A. M. First student workshop on computer science & software engineering. Computer Science & Software Engineering: Proceedings of the 1st Student Workshop, 2018. P. 1-10.

136. Komar M., Sachenko A., Kochan V., Ababii V. Improving the Security ofIntrusion Detection System. Proc. of 13thInternational Conference on Development andApplication Systems, 2016. P. 19–21.

137. Kononenko I.V., Aghaee A. Model and Method for Synthesis of Project Management Methodology With Fuzzy Input Data. Bulletin of NTU" KhPI". Ser.: Strategic Management, Portfolio, Program and Project Management, 2016, №1 (1173). P. 9–13.

138. Kosenko V., Persiyanova E., Belotskyy O., Malyeyeva O. Methods of managing traffic distribution in information and communication networks of critical infrastructure systems", innovative technologies and scientific solutions for industries, 2017. №2 (2). P. 48–55.

139. Lishchyna, N., Lishchyna, V., Povstyana, J. Approaches and algorithms for processing and image recognition of complex structure. Computer-integrated technologies: education, science, production, 2020. №(38), С.5–9.

140. Martsenyuk V., Didmanidze I., Sverstiuk A., Andrushchak I., Rud K. Automated method of building exploites in analysis software testing. Computer-integrated technologies: education, science, production, 2020. №39. С.146–150.

141. Martynyuk O. N., Ahmesh T., Drozd O. V., Stepova H. S. Checkability of hierarchical transmitions for behavioral check. Systems and Technologies, 2018. V. 1, n. 56. P. 30–40.

142. Martynyuk O. N., Ahmesh T., Thuong B. V., Drozd O. V. Multilevel behavioral testing of distributed information systems. Herald of Advanced Information Technology, 2019. Vol.2 No.4. C. 298–309.

143. Mehmood N., Petersen M. K., Börstler J., Wnuk K. Regression testing for large-scale embedded software development – Exploring the state of practice. Information and Software Technology, 2020. Vol. 120. Article 106243.

144. Melkozerova, O., Rassomakhin, S. Software performance testing. Bulletin of V.N. Karazin Kharkiv National University, Series «Mathematical Modeling. Information Technology. Automated Control Systems», 2020. №45 P.56–66.

145. Melkozorova O., Haikova V., Malakhov S. Circuit description of API interface for software testing. Computer Science and Cybersecurity, 2020. (4). P.38-45.

146. Opanasenko V., Kryvyi S. Synthesis of Multilevel Structures with MultipleOutputs. CEUR Workshop Proc. of the 10thInternational Conference of Programming, UkrPROG, 2016. Vol.1631, Code 122904. P.32–37.

147. Schmidt C. Agile Software Development Teams: the Impact of Agile Development on Team Performance. Progress in IS, 2016. P.184.

148. Siavvas M. G., Chatzidimitriou K. C., Symeonidis A. L. QATCH An adaptive framework for software product quality assessment. Expert Systems with Applications, 2017. Vol. 86. P. 350–366.

149. Teslia Yu., Khlevnyi A., Khlevna I. Control of informational Impacts on project management. Proceedings of the 1th IEEE International Conference on Data Stream Mining & Processing, 2016. P. 387–391.

150. Velychko O., Gaman V., Gordiyenko T., Hrabovskyi O. Testing of measurement instrument software with the purpose of conformity assessment.Eastern-European Journal of Enterprise Technolo-gies. Information and controlling systems, 2019. No1/9(97). P. 19–26.

151. Velychko O., Gordiyenko T., Hrabovskyi O. Testing of measurement instrumentsoftware on the national level. Eastern-European Journal of Enterprise Technologies. Information and controlling systems, 2018. No 2/9(92). P. 13–20.

152. Yaskevych V., Klochko O., Методи підвищення відмовостійкості інтернет сервісів. Кібербезпека: освіта, наука, техніка, 2019. Вип. 3. С. 104–111.

153. Zeina S. Salleha N., Grundyb J. A systematic mapping study of mobile application testing techniques. Journal of Systems and Software, 2016. Vol. 117. P. 334 – 335.

Додаток А. Коди розробленого фреймворку

Визначення контракту у тестах за допомогою розробленого фреймворку:

```
public void defineService2Behaviour()
{
    var contractBuilder = new ContractBuilder();

    contractBuilder
        .ClearContract()
        .SetCustomer("Service1")
        .SetSupplier("Service2")
        .RequestName("create new user")
        .DefineRequest(new
        {
            Method = HttpVerb.Post,
            Path = "/new",
            Body = new
            {
                Name = "Second User",
                Email = "second@test.com"
            }
        })
        .ExpectedAnswer(new
        {
            StatusCode = 200,
            Body = "User has successfully created"
        })
        .GenerateContract()
        .SendContract("serviceimitator2", "4684:4684");
    contractBuilder
```

```
        .ClearContract()
        .SetCustomer("Service1")
        .SetSupplier("Service2")
        .RequestName("get user by id")
        .DefineRequest(new
        {
           Method = HttpVerb.Get,
           Path = "/id/a250"
        })
        .ExpectedAnswer(new
        {
           StatusCode = 200,
           Body = new
           {
              Id = "a250",
              Name = "Main User",
              Email = "main@test.com"
           }
        })
        .GenerateContract()
        .SendContract("serviceimitator2", "4684:4684");
}
```

Тест для перевірки методу сервісу:

```
public class CustomerTest
   {
      private readonly HttpClient _client;
      public CustomerTest(HttpConnection client, CreateContract contract)
      {
```

```
        contract.defineService2Behaviour();
        _client = client.Client;
    }
    [Fact]
    public void WhenRegistrationIsFinished_ThenUserIsCreated()
    {
        var user = new UserModel
        {
            Name = "Second User",
            Email = "second@test.com",
            Id = "0981234567",
            UserExist = false,
            RegistrationDone = true
        };
        var expectedResponse = "Registration is finished and user has been created";
        // Посилаємо запит на сервіс, який тестується
        var response = _client.PostAsync("/registration/finish",
            new StringContent(JsonConvert.SerializeObject(user), Encoding.UTF8, "application/json"));
        var status = response.Result.StatusCode;
        // Перевірка отримання успішної відповіді від сервісу
        Assert.Equal(HttpStatusCode.OK, status);
        var content = response.Result.Content.ReadAsStringAsync().Result;
        var result = !string.IsNullOrEmpty(content)
            ? content.ToString()
            : null;
        // Порівнюємо очікуваний запит з отриманим
        Assert.Equal(expectedResponse, result);
    }
```

```
}
```

Код тесту в режимі постачальника :

```
public class UserTests
{
    private readonly ContractExecutor _contractExecutor;
    public AuthenticationTest(ContractExecutor executor)
    {
        _contractExecutor = executor;
    }
    [Fact]
    public void CreateNewUser()
    {
        _contractExecutor
            .GetService("localhost:2525/imposters/4684")
            .SetSupplier("Service2")
            .SetCustomer("Service1")
            .FindByName("create new user")
            .ExecuteContract()
            .Assert();
    }
}
```

Додаток Б. Акти впровадження

АКТ

про впровадження результатів дисертаційної роботи Базаки Юрія Анатолійовича поданої на здобуття ступеня кандидата технічних наук за спеціальністю 05.13.06 «Інформаційні технології» на тему «Моделі, методи та інформаційна технологія підвищення ефективності тестування розподілених систем»

Комісія зі складу фахівців ТОВ «Нафтогазбудінформатика»:
Голова:
Виконавчий директор Волкова Н.П.

Члени комісії:
Головний спеціаліст Куцан Н.В.
Провідний спеціаліст Глущенко Г.Л.

розглянула матеріали звітів та результати роботи під час впровадження в 2017-2019 роках інформаційної розподіленої системи обробки даних і встановила наступне.

В процесі розробки інформаційної системи розподіленої системи обробки даних з використанням гетерогенних баз даних на основі мережоцентричного підходу впроваджені наступні результати дисертаційної роботи Базаки Ю.А. «Моделі, методи та інформаційна технологія підвищення ефективності тестування розподілених систем», які отримані ним особисто:

- Розроблено модель тестування програмного забезпечення розподілених систем на основі автономного розгортання програмних компонентів, яка базується на застосуванні механізмів на основі модифікованої піраміди Майка Кона, що дозволяє підвищити ефективність тестування та зменшити кількість об'ємних тестів. Дана модель є основою запропонованих методів і дозволяє забезпечити механізми тестування розподілених систем.

Даний акт не є підставою для фінансових взаєморозрахунків.

Голова комісії:
Виконавчий директор Волкова Н.П.

Члени комісії:
Головний спеціаліст Куцан Н.В.
Провідний спеціаліст Глущенко Г.Л.

«ЗАТВЕРДЖУЮ»
Директор конструкторського бюро
інформаційних систем
КПІ ім. Ігоря Сікорського

_____ О.Д. Фіногенов
“ 23 ” _____ 2022 р.

АКТ

про використання результатів дисертаційної роботи
Базаки Юрія Анатолійовича на тему «Моделі, методи та інформаційна технологія підвищення ефективності тестування розподілених систем», поданої на здобуття вченого ступеня кандидата технічних наук
при розробці інформаційних систем

Комісія в складі: начальника відділу супроводження порталу університету конструкторського бюро інформаційних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського», Цуріна Н.О. – голова комісії; начальник відділу розробки та підтримки програмного забезпечення Габзовська О.Б. і провідний інженер відділу розробки та підтримки програмного забезпечення Ромашкевич Я.О. – члени комісії; розглянула матеріали звітів та результати роботи під час розробки та впровадження в 2015-2020 роках інформаційних систем та встановила наступне:

В процесі розробки інформаційних систем впроваджені наступні результати дисертаційної роботи Базака Ю.А. «Моделі, методи та інформаційна технологія підвищення ефективності тестування розподілених систем», які отримані ним особисто:

- розроблено механізм тестування програмного забезпечення розподілених систем на основі автономного розгортання програмних компонентів, яка базується на застосуванні механізмів на основі модифікованої піраміди Майка Кона, що дозволяє

підвищити достовірність тестування та зменшити кількість об'ємних тестів;

- удосконалено тестування програмного забезпечення вузлів розподіленої системи, який відрізняється від існуючих застосуванням контрактних тестів та аналізом прогнозованого результату поведінки сервісів, що дозволяє зменшити час розгортання компонентів тестового середовища та час проходження тестів.

Тестові випробування і практичне застосування інформаційних систем підтвердили коректність та ефективність запропонованих проектних рішень та достовірність відповідних положень дисертаційної роботи.

Голова комісії Н.О. Цуріна

Члени комісії: О.Б. Габзовська

Я.О. Ромашкевич

«ЗАТВЕРДЖУЮ»
Декан факультету інформатики
та обчислювальної техніки
КПІ ім. Ігоря Сікорського

_____ Сергій ТЕЛЕНИК
«__13__» ____липня____ 2021 р.

АКТ

про впровадження в навчальний процес кафедри технічної кібернетики факультету інформатики та обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» результатів дисертаційної роботи Базаки Юрія Анатолійовича «Моделі, методи та інформаційна технологія підвищення ефективності тестування розподілених систем», поданої на здобуття наукового ступеня кандидата технічних наук.

Ми, що нижче підписалися: завідувач кафедри технічної кібернетики, д.т.н. проф. Пархомей І.Р., заступник завідувача кафедри, к.т.н., доц. Ткач М.М., вчений секретар кафедри, к.т.н., доц. Ліхоузова Т.А. розглянувши матеріали навчально-методичного забезпечення курсів: «Основи програмування» та «Якість програмного забезпечення та тестування», які підготовлені та викладаються асистентом Базакою Ю.А., що в навчальному процесі на лабораторних заняттях, а також в процесі підготовки бакалаврських атестаційних робіт використовуються такі наукові результати, отримані дисертантом особисто:

- алгоритм застосування формального апарату опису предметної області, що дозволяє переносити його до інформаційної системи та використовувати при розробці програмного забезпечення;

- механізми проведення тестування при розробці програмного забезпечення розподілених систем;
- інформаційна технологія для автоматизації процесів тестування програмних систем.

Впровадження результатів дисертаційної роботи Базаки Ю.А. в навчальний процес дозволило підвищити якість підготовки студентів бакалаврату спеціальності 121 «Інженерія програмного забезпечення» на кафедрі технічної кібернетики КПІ ім. Ігоря Сікорського.

Завідувач кафедри
технічної кібернетики,
д.т.н., проф. Ігор ПАРХОМЕЙ

Заст. завідуючого кафедри,
к.т.н., доц. Михайло ТКАЧ

Вчений секретар кафедри
технічної кібернетики,
к.т н., доц. Тетяна ЛІХОУЗОВА

Додаток В. Список публікацій здобувача за темою дисертації

Праці, в яких опубліковані основні наукові результати дисертації:

1. Mukhin V., Kornaga Ya., Mostovyi Y., Bazaka Y. A Model For Events Monitoring Heterogeneous Distributed Databases Based on Vector-matrix Operations. The Far East Journal of Electronics and Communications, 2016. Vol. 16, Issue 3. P. 645 – 656. (This journal is indexed in Elsevier Scopus).

2. Zhenbing H., Mukhin V., Kornaga Ya., Herasymenko O., Bazaka Y. The scheduler for the grid system based on the parameters monitoring of the computer components. Eastern European Journal of Enterprise Technologies, 2017. № 1 (2-85). P. 31 – 39. (This journal is indexed in Elsevier Scopus).

3. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Метод автоматизації тестування розподіленої системи з використанням контрактів. Sciences of Europe, 2020. № 55. С. 45 – 49.

4. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Метод тестування інтерфейсу користувача з використанням імітаторів. The scientific heritage, 2020. № 51. С. 54 – 57.

5. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Method of Protecting Software Code from Reengineering Using Virtual Machines. Sciences of Europe, 2020. № 58. С. 59 – 62.

6. Корнага Я.І., Базака Ю.А., Базалій М.Ю. Захист програмного забезпечення за допомогою заплутуючих перетворень. The scientific heritage, 2020. № 54. С. 72 – 75.

7. Корнага Я.І., Герасименко О.Ю., Базака Ю.А., Базалій М.Ю., Мухін О.В. Використання баз даних тамов програмування у високоточних обчисленнях чисел із плаваючою точкою великої розрядності. Вчені записки Таврійського національного університету імені В.І. Вернадського, 2020. Том 31(70), №5. С. 82 – 88.

Праці, що засвідчують апробацію матеріалів дисертації:

8. Mukhin V., Kornaga Ya., Yakovleva A., Herasymenko O., Bazaca Yu., Bazaliy M. The model for distributed systems testing based on the control services. XXIV Міжнародна конференція з Автоматичного Управління "АВТОМАТИКА – 2017". 13 – 15 вересня 2017 р., м. Київ., 2017. С. 45 – 46.

9. Мухін В.Є., Корнага Я.І., Яковлєва А.П., Базалій М.М., Базака Ю.А. Модифікований метод обфускації програмного кода за допомогою вставки інструкцій. Міжнародна наукова конференція "Інтелектуальні системи прийняття рішень та проблеми обчислювального інтелекту" (ISDMCI'2018), 21 – 27 травня 2018 р., с. Залізний порт. – Херсон: Видавництво ПП Вишемирський, 2018.– С. 67 – 68.

10. Mukhin V., Kornaga Ya., Bazaka Yu., Bazaliy M., Yakovleva A. Modified Method of Software Testing for Distributed Computer System. 2018 IEEE First International Conference on System Analysis & Intelligent Computing (SAIC). 8 – 12 October 2018, Kiev, Ukraine, 2018. P. 1 – 4. (This conference is indexed in Elsevier Scopus).

11. Mukhin V.Ye., Kornaga Ya.I., Abdullayev S.H., Gerasymenko O.Yu., Bazaka Yu.A. Method of the reserve resources determination for the distributed computer systems with the network-centric resource control. 6th International Conference on Control and Optimization with Industrial Applications (COIA). 11 – 13 July 2018. Baku, Azerbaijan, 2018. P. 226 – 231. (This conference is indexed in Web of Science).

12. Mukhin V., Vyshnivskyi V., Kornaga Ya., Herasymenko O., Bazaka Yu., Bazaliy M. Study of the functioning of the distributed computer system with a resource control mechanism based on a network-centric approach. 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2019. 18 – 21 September 2019, Metz, France. – 2019. P. 100 – 105. (This conference is indexed in Elsevier Scopus, Web of Science).

13. Mukhin V., Kornaga Y., Tkach M., Herasymenko O., Bazaka Y.,

Mukhin O. Subtask Prioritization on Workflow Execution in Distributed Wireless Computer System with Network-Centric Approach to Resource Control. 5th IEEE International Symposium on Smart and Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems, IDAACS-SWS 2020, 17 September 2020, Dortmund, Germany. – 2020. (This conference is indexed in Elsevier Scopus, Web of Science).

Таблиця В.1

Відомості про апробацію результатів дисертації

№ з/п	Назви конференції, конгресу, симпозіуму, семінару, школи	Місце проведення	Дата проведення	Форма участі
1.	XXIV Міжнародна конференція з Автоматичного Управління «АВТОМАТИКА 2017»	м. Київ	13-15 вересня 2017 р.	очна
2.	Міжнародна наукова конференція «Інтелектуальні системи прийняття рішень та проблеми обчислювального інтелекту» (ISDMCI) (Залізний порт, 2018)	с. Залізний порт	21-23 травня 2018 р.	очна
3.	2018 IEEE First International Conference on System Analysis & Intelligent Computing (SAIC)	м. Київ	8-12 жовтня 2018 р.	очна
4.	6th International Conference on Control and Optimization with Industrial Applications (COIA)	м. Баку	11-13 липня 2018 р.	очна
5.	10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)	м. Метц	18-21 вересня 2019 р.	очна
6.	5th IEEE International Symposium on Smart and Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS 2020)	м. Дортмунд	17 вересня 2020 р.	заочна