Київський національний університет імені Тараса Шевченка
Факультет радіофізики, електроніки та комп’ютерних систем
Кафедра комп’ютерної інженерії

МЕТОДИ АНАЛІЗУ ТА ПІДВИЩЕННЯ ШВИДКОДІЇ WEB-ЗАСТОСУНКУ НА ПРИКЛАДІ СИСТЕМИ ІНФОПАКЕТ

Дипломна робота магістра
студента 2 року навчання
спеціальність: 123 «Комп'ютерна інженерія»
Миколи ДЕРКАЧА

Науковий керівник
асистент кафедри комп'ютерної інженерії
Юрій ЮРЧИК

Рецензент
канд. фіз.-мат. наук Денис ЛЕБЕДЕВ,
доцент факультет електроніки
київського політехнічного інституту імені Ігоря Сікорського

До захисту допускаю: Завідувач кафедрою

Юрій БОЙКО

Ухвалено на засіданні кафедри “____” _____ 2022 р., протокол № ___

Київ - 2022

РЕФЕРАТ

Обсяг роботи магістра за об’ємом складає 41 сторінка, містить 18 рисунків, 4 додатки, 7 використаних джерел.

Інформаційний пакет – це каталог курсів Київського Національного Університету імені Тараса Шевченка, який вміщує інформацію про навчальні програми університету, відповідних навчальних дисциплін, викладачів та структурних підрозділів.

Актуальність роботи базується, на необхідності пришвидшення роботи каталогу курсів університету та підвищення рівня індексування пошуковими сервісами.

Предметом роботи є каталог курсів Київського Національного Університету імені Тараса Шевченка, тобто інформаційний пакет.

У рамках роботи було використано систему оновлення даних про систему, яка була створена у веб-програмі. Закритий програмний комплекс надає можливість вносити інформацію про редакцію каталогів університету.

Для пришвидшення програмного забезпечення було використано мови програмування C# від компанії Microsoft; використовуються програмні бібліотеки MVC, Entity Framework. Підходи SOLID, DRY та шабони проєктування програмного продукту.

Ключові слова: MVC, EntityFramework, .NET Framework, MSSQL Server.

Зміст

ВСТУП

Інформаційний пакет – це каталог курсів Київського Національного Університету імені Тараса Шевченка, який вміщує інформацію про навчальні програми університету, відповідних навчальних дисциплін, викладачів та структурних підрозділів.

Наявність такого каталогу та публічного доступу до нього є необхідною умовою для вищого навчального закладу згідно положень закону України про вищу освіту. Це робить можливим участь навчального закладу у програмах академічної мобільності, визнання дипломів іншими університетами, коректної підготовки додатків дипломів європейського зразку, акредитації освітніх програм у НАЗЯВО та участі університету у рейтингах освітніх закладів[1].

Метою даної роботи є пошук шляхів пришвидшення роботи каталогу курсів університету. Пошук та виправлення коду, відповідно до загальноприйнятих норм написання.

РОЗДІЛ І. РЕФАКТОРИНГ.

Рефакторинг коду (англ. Code refactoring) – це перевірка та правка написаного коду на відповідність загальноприйнятим нормам за для покращення швидкості виконання, легкості читання та повторного використання написаного коду. Під час рефакторингу виконується перевірка написаного коду відповідності принципам SOLID та DRY.

1.1 Шаблони (патерни) проектування.

Шаблони (патерни) проектування – це найкращі практики щодо вирішення проблем та задач, які часто зустрічаються в ході програмування. Використання патернів допомагає уникнути можливості допускання помилок, покращує читабельність коду, та пришвидшує роботу програми. Шаблони не прив’язані до мов програмування, тому їх реалізація на різних мовах може відрізнятися.

Концепція патернів прийшла в програмування з будівництва, коли в 1977-му році архітектор Крістофер Александр опублікував книгу «Мова шаблонів. Міста. Будинки. Будівництво». В книзі описано 253 шаблони проектування міст. Автор підмічає найкращі практики в будівництві а також часті помилки в проектуванні та методи їх розв’язання[2]. Дана ідея не отримала широкої підтримки в сфері будівництва, на відміну від сфери програмування.

В 1994 році, натхненні ідеями Александра чотири програміста: Ерік Гамма (Erich Gamma), Річард Хелм (Richard Helm), Ральф Джонсон(Ralph Johnson) і Джон Влісседс (John Vlissides) видають книгу «Прийоми об'єктно-орієнтованого проектування. Патерни проектування» (англ. Design Patterns: Elements of Reusable Object-Oriented Software). Назва книги виявилася завеликою, тому в простонародді книгу почали називати «книга

банди чотирьох» («book by the gang of four»), а пізніше назву взагалі скоротили до «GoF book» (GoF – від gang of four). Завдяки цій книзі шаблони проектування набули такої популярності в програмуванні.

В книгу увійшли 23 архітектурні шаблони проектування. В книзі дані патерни розділялися на три категорії:

- **Породжуючі** (абстрагують процес створення об’єктів та класів)
- **Структурні** (розглядають як класи та об’єкти об’єднуються в більш складні структури)
- **Поведінкові** (визначають алгоритми взаємодії між класами та об’єктами)

На сьогоднішній день було розроблено десятки інших шаблонів та додано різні категорії, далі в дипломній роботі буде розглянуто патерни, описані в книзі «Прийоми об'єктно-орієнтованого проектування. Патерни проектування».

1.1.1.Породжуючі шаблони

Породжуючі патерни абстрагують процес створення об’єктів та класів, роблячи даний процес більш зручним та безпечним. В «книзі банди чотирьох» розрізняють наступні п’ять породжуючих шаблонів:

- **Фабричний метод** (Factory method) – це породжуючий патерн проектування, що визначає інтерфейс для створення об’єктів деякого класу, але саме створення об'єктів доручає підкласам, які в свою чергу можуть змінювати деякі частини класу, для реалізації специфічних потреб. Даний шаблон використовують, коли заздалегідь невідомо, які специфічні потреби можуть знадобитися в тому, чи іншому випадку, але при цьому основні функції мають виконуватися у всіх випадках. Недоліками можна вважати те,

що для кожного нового продукту потрібно створювати свою «фабрику», тобто свій клас.

- **Абстрактна фабрика** (Abstract factory) – це породжуючий патерн проектування, що дозволяє створювати сімейства об’єктів, без залежності від конкретних класів, створюваних об’єктів. Даний шаблон використовують, коли створювані об’єкти використовуються разом і мають бути взаємозалежними. Серед недоліків можна виділити те, що код програми може сильно ускладнитися, через створення додаткових класів.

- **Будівельник** (Builder) – це породжуючий патерн проектування, що дає можливість створювати складні об’єкти покроково, додаючи потрібні властивості об’єкта по ходу його створення. Будівельник дає можливість використовувати один і той же код для створення різних об’єктів. Недоліком даного патерна є те, код програми може сильно ускладнитися, через створення додаткових класів.

- **Прототип** (Prototype) – це породжуючий патерн проектування, що дає можливість копіювати об’єкти, не вдаючись в деталі їх реалізації. Коли об'єкти програми містять сотні полів і тисячі можливих конфігурацій, прототипи можуть служити своєрідною альтернативою створенню підкласів. Зазвичай, реалізація даного шаблону схожа у всіх мовах програмування. Клас, який потрібно копіювати підписують на реалізацію інтерфейсу, який, зазвичай має тільки один метод – Clone(). В даному методі описується як саме потрібно клонувати даний клас, всі його приватні поля і розпутуються внутрішні залежності. Перевагами даного шаблону є те, що він надає можливість клонувати об’єкти без прив’язки до конкретних класів, зменшує повторення коду, пришвидшує створення об’єктів та є альтернативою створенню підкласів складних об’єктів. Недоліком є те, що важко реалізовувати інтерфейс, що буде клонувати об’єкт, який має посилання на інші об’єкти.

- **Одинак** (Singleton) – це породжуючий шаблон проектування, який гарантує, що у класу буде тільки один екземпляр і забезпечує для нього

глобальну точку доступу. Одинака часто використовують для створення єдиного доступу для підключення до бази даних. Оскільки кожен клас може використовувати потрібні для нього дані з БД, то замість того, щоб кожен клас мав поле підключення до бази, можна винести це поле в клас одинака. Тоді не буде створюватися новий об’єкт в динамічній пам’яті при ініціалізації класу. Недоліками одинака є те, що він порушує прицип єдиного обов’язку.

1.1.2. Структурні шаблони

Структурні шаблони – це шаблони проектування, які розглядають як класи та об’єкти об’єднуються в більш складні структури. В «книзі банди чотирьох» розрізняють наступні сім структурних шаблона:

- **Адаптер** (Adapter) – це структурний шаблон проектування, який дозволяє взаємодію між об’єктам з несумісними інтерфейсами. Даний шаблон схожий на перехідник між розеткою та вилкою різних країн, для їх взаємодії потрібен спеціальний адаптер.
- **Міст** (Bridge) – це структурний патерн проектування, він покликаний розділяти інтерфейси і реалізації і застосовувати їх незалежно один від одного. Даний шаблон реалізує принцип відкритості/закритості SOLID (Open/Closed Principle): система має бути розширюємою, але закритою для змінення вже існуючого коду.
- **Компонувальник** (Composite) – це структурний шаблон проектування, який групує об’єкти в дерево і дає можливість працювати з ними, як одиничним об’єктом. Даний патерн також інколи називають деревом. Він полегшує додавання нових компонентів, а також спрощує взаємодію з існуючими об’єктами, але створює занадто загальний дизайн класів.

- **Декоратор** (Decorator) – це структурний патерн проектування, який дозволяє додавати допоміжний функціонал до об’єкту, шляхом «обгортання» об’єкту. Через це декоратор також інколи називають обгорткою. Проводячи аналогію з реальним світом, декоратор нагадує одяг. Людина може одягнути на себе куртку і отримати додатковий функціонал захисту від холоду, дощу та снігу. Декоратор використовують у випадку, коли наслідування є неприйнятним або не бажаним, але об’єкту потрібно додати функціонал. Серед переваг також те, що обгортку можна додавати динамічно. Серед недоліків є те, що декоратор – це по суті невеликий клас і коли є потреба додавати різний функціонал може з’являтися велика кількість крихітих класів. Крім того, важко конфігурувати об’єкти, які загорнуто в декілька обгорток.

- **Фасад** (Facade) – це структурний шаблон проектування, який надає простий і зрозумілий інтерфейс для складної системи. Він дозволяє закрити від клієнтської частини непотрібний функціонал і залишити лише необхідний. Недоліком є те, що при невмілому використанні фасад ризикує перетворитися на «божественний об’єкт», тобто на об’єкт, який володіє занадто великим функціоналом.

- **Легковаговик** (Flyweight) – це структурний шаблон проектування, який покликаний заощадити оперативну пам’ять, шляхом винесення спільного стану великої кількості об’єктів у окремий клас. Його часто використовують разом з фабричним методом, який займається обчисленням контексту для конкретних об’єктів. Даний патерн слід використовувати тільки в тому випадку, коли об’єктів дійсно велика кількість, оскільки легковаговик економить тільки оперативну пам’ять, за рахунок обчислювальних потужностей, які будуть витрачатися на обчислення контексту. Також він ускладнює код програми внаслідок введення додаткових класів.

- **Замісник** (Proxy) – це структурний шаблон проектування, суть якого полягає в створенні об’єкту-замісника. Даний об’єкт може

використовуватися для того, щоб кешувати запити до об’єкту-оригіналу, захищати об’єкт-оригінал від прямого доступу, для пришвидшення взаємодії, якщо об’єкт-оригінал є важким та з ним важко взаємодіяти, або для локального доступу, якщо об’єкт-оригінал знаходиться на віддаленому сервері. Часто замісник використовують з віддаленим сервером або базою даних.

1.1.3. Поведінкові шаблони

Поведінкові шаблони – це шаблони проектування, які визначають поведінку класів і об’єктів між собою. В «книзі банди чотирьох» розрізняють наступні одинадцять поведінкових паттернів:

- **Ланцюг обов’язків** (Chain of Responsibility) – це поведінковий шаблон проектування, який дає можливість опрацьовувати запит чергою обробників. Даний підхід дає можливість використовувати необхідний функціонал тільки тоді, коли він потребується. Недоліком даного шаблону є те, що запит може бути пропущений кожною ланкою і не обробитися взагалі. Chain of Responsibility використовують, коли один запит потрібно обробити динамічно, не знаючи, як саме його прийдеться оброблювати.
- **Команда** (Command) – це поведінковий шаблон проектування, який дає можливість перетворити оброблення запиту на об’єкт. Це, в свою чергу, дає можливість ставити запити в чергу та виконувати їх тоді, коли нам зручно, скасовувати або повторювати їх. Аналогом шаблону Command в мові C# є делегати та події.
- **Ітератор** (Iterator) – це поведінковий шаблон проектування, що дає можливість обходу складних структур даних в необхідному для нас порядку, без розкриття їх внутрішньої структури. Якщо ми маємо специфічну, власну структуру даних, яка передбачає, що її будуть обходити,

ми можемо використати шаблон ітератор та описати власні способи їх обходу. Даний шаблон є необхідним лише у випадку, коли потрібну для нас структуру не можна обійти простим циклом, інакше паттерн може ускладнити існуючий код.

● **Посередник** (Mediator) – це поведінковий шаблон проектування, який виконує функцію зменшення зв'язності між класами шляхом перенесення зв'язків від взаємодіючих між собою класів до спеціального класу посередника. Даний паттерн використовують для розплутування складних зв'язків між класами і централізації керування цими класами. Також посередник Спрощує взаємодію між компонентами і дає можливість їх повторного використання. Серед недоліків можливість того, що посередник сильно роздується і потребуватиме окремого рефакторингу.

● **Знімок** (Memento) – це поведінковий шаблон проектування, який покликаний зберігати минулі стани об'єкту (знімки) дотримуючись принципу інкапсуляції. Недоліками даного паттерну є те, що він потребує великої кількості пам'яті, якщо потрібно буде зберігати велику кількість станів. Також якщо правильно та вчасно не звільняти пам'ять від не потрібних знімків, то шаблон може спричинити додаткові витрати пам'яті. До того ж не всі мови програмування можуть забезпечувати правильну інкапсуляцію даних (PHP, Python, JavaScript).

● **Спостерігач** (Observer) – це поведінковий паттерн проектування, що надає механізм стеження одними об'єктами за подіями в інших об'єктах. Реалізація даного шаблону: об'єкт, який має спостерігати (спостерігач) звертається до того, за ким він спостерігає (спостережуваний) і просить його сповіщати тоді, коли з ним відбувається якась подія. Даний шаблон схожий на паттерн Посередник, але їх різниця полягає в тому, що Спостерігач має на меті забезпечити динамічний односторонній зв'язок, а Посередник прибирає залежності.

● **Стан** (State) – це поведінковий шаблон проектування, що дає можливість змінювати поведінку об'єкту, в залежності, від його

внутрішнього стану. Внутрішні стани об’єкту є скінченний автоматом. Стан можна застосовувати тоді, коли в коді присутня велика кількість умовних операторів. В такому випадку гілки умовних операторів переносяться в окремі класи-стани. Даний паттерн покликаний спростити код шляхом позбавлення від великих умовних операторів або винесення дублювання коду в окремі класи. Але якщо станів мало і вони змінюються рідко, використання шаблону невиправдано ускладнить код.

- **Стратегія** (Strategy) – це поведінковий шаблон проектування, який визначає деякий набір схожих алгоритмів, інкапсулює їх та забезпечує їх взаємозамінність. Дані інкапсульовані в класи алгоритми і називаються стратегіями. Для взаємозамінності всі класи матимуть однаковий інтерфейс. Перевагами паттерну є його динамічність, реалізація інкапсуляції алгоритмів, заміна спадкування делегуванням. Серед недоліків – ускладнення коду внаслідок додавання нового функціоналу. А також той факт, що клієнт повинен розбиратися в представлених стратегіях, щоб обрати необхідну для нього.

- **Шаблонний метод** (Template method) – це поведінковий шаблон проектування, покликаний прибрати дублювання коду. Для цього спільний код з декількох класів переноситься в батьківський абстрактний клас. Методи, які є спільними отримують реалізацію, а методи, які відрізняються залишаються без змін. Крім того існують ще хуки. Це необов’язкові методи, які можуть взагалі не містити коду. В необхідних місцях хуки отримують реалізацію, таким чином створюючи можливість вклинюватися в хід виконання. Недоліками даного паттерну є обмеженість скелетом існуючого алгоритму. Крім того, шаблон стає важко підтримувати, якщо кількість кроків збільшується.

- **Відвідувач (Visitor)** – це поведінковий шаблон проектування, який покликаний доповнити функціонал класу, не змінюючи його. Для виконання поставленої задачі паттерн пропонує створити новий клас «відвідувач», який буде виконувати необхідні дії над об’єктом класу, якому

необхідно додати функціоналу. Шаблон може прибирати дублювання коду, шляхом концентрації спільних операцій в класі відвідувачі. Також спрощується процес додавання операцій, у випадку складної структури даних. Але паттерн може призвести до порушення інкапсуляції. А також його використання не покращить роботу, якщо код має змінну ієрархію елементів[3].

1.2. Принципи SOLID

SOLID – це принципи об’єктно орієнтованого програмування, запропоновані Робертом Мартіном на початку 2000-х років[4]. Назва SOLID є акронімом від 5ти принципів:

- S – Single responsibility principle (Принцип одної відповідальності) – Один об’єкт має свій власний унікальний єдиний обов’язок.
- O – Open/close principle (Принцип відкритості/закритості) – Програма має бути відкритою для розширення, але закритою для змін.
- L – Liskov substitution principle (Принцип підстановки Лісков) – У всіх місцях в програмі, де використовується батьківський об’єкт, може бути використаний наслідуємий об’єкт, без зміни коду програми, але не навпаки.
- I – Interface segregation principle (Принцип розділення інтерфейсу) – Інтерфейси мають спеціалізуватися на конкретній задачі.
- D – Dependency inversion principle (Принцип інверсії залежностей) – Конкретні реалізації мають бути залежними від абстракцій а не навпаки. Модулі нищих рівнів мають залежати від модулів вищих рівнів. Для реалізації даного принципу потрібно відмовитися від конкретних класів та використовувати абстрактні класи та інтерфейси.

1.3. Принципи DRY

DRY (від англ. Don`t repeat yourself) – Принцип розробки програмного забезпечення, який полягає в повторному використанні написаного коду. Даний принцип покликаний зменшити кількість написаного коду для полегшення його читання і пошуку помилок, а також для спрощення написання нового функціоналу.

Якщо потрібно додати новий функціонал, легше робити це в одному, відповідальному за це місці, а не додавати його всюди, де функціонал викликається. Не дивлячись на логічність даного підходу, на практиці розробнику легше писати для конкретної задачі свою унікальну реалізацію. Даний принцип зобов’язує робити пошук схожого функціоналу та об’єднувати його абстракцією, що є складнішою задачею, ніж написання унікальних реалізацій.

Порушення принципу DRY називають WET (від англ. “Wright everything twice”, або “We love typing”) – даний акронім є грою слів (від англ. dry – сухий, wet – вологий). Такий підхід може викликати важкість читання написаного коду, а також проблеми з роботою з вже існую функціональністю.

1.4. Рефакторинг коду в інфопакет та застосування шаблонів проєктування

Рефакторинг написаного коду є частиною роботи з проєктом інфопакет. Під час написання програмного продукту виконується перевірка на відповідність Принципам SOLID, DRY та використання шаблонів проєктування.

Найбільш росповсюдженим шаблоном проєктування в системі інфопакет є шаблон замісник (proxy). Його використання полягає у заміні взаємодії з базою даних напряму на взємодію з об'єктом замісником. Задля використання цього шаблону було введено в проєкт бібліотеку Entity Framework. Дана бібліотека дозволяє абстрагуватися від прямих запитів до бази даних і працювати з нею як з екземпляром класу. Такий підхід не тільки дає змогу працювати з програмним кодом C# замість запитів SQL, але і забезпечує додаткову безпеку, оскільки написання строкових запитів можуть містити небезпечні помилки.

```
private nr12_informpaketEntities db = new nr12_informpaketEntities();
var selectedCourses = db.Courses
    .Where(m => m.FacultyID == userFacultyId)
    .Where(m => m.YearID == years);
return View(selectedCourses.ToList());
```

Рис. 1. Приклад використання шаблону замісник в системі інфопакет

Застосування кешування, яке буде детальніше розглянуто в наступних розділах роботи, є нічим іншим, як застосуванням шаблону знімок (memento).

Простим рішенням в деяких випадках рефакторингу може бути винесення спільного коду в окремі методи та викликання їх з місць, де раніше використовувався винесений код. В необхідному місці метод визивається з відповідними параметрами. Такий підхід домагає в підтримці існуючого коду. Для прикладу, якщо потрібно змінтити загальний функціонал у всіх місцях, в

яких він викликається, то це потрібно буде робити тільки в одному місці, а якщо потрібно буде додати функціоналу в окремому місці, можна додати його перед, або після виклику методу. Такий підхід є прикладом використання паттерну template method (шаблонний метод).

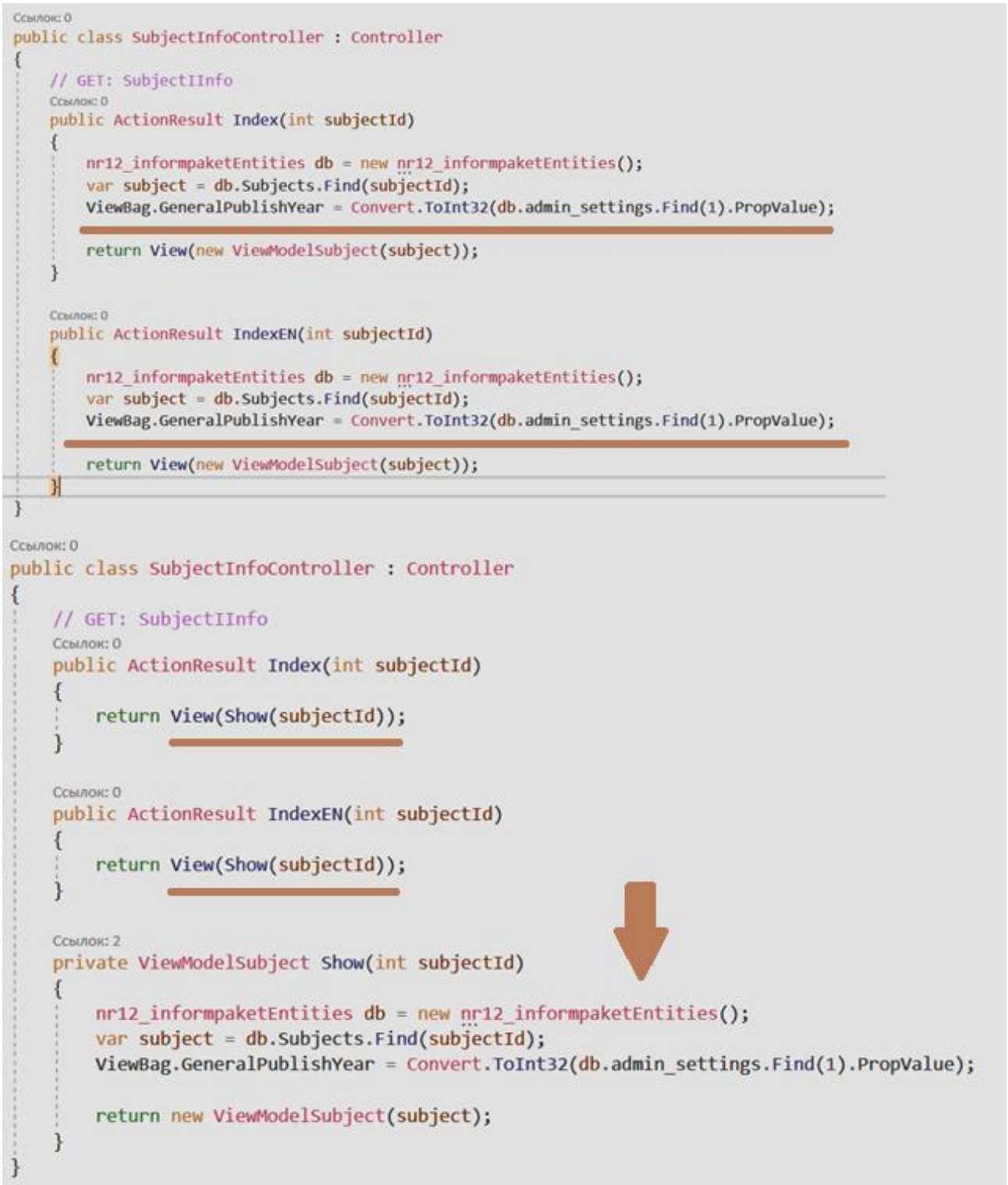

Рис. 2. Приклад використання шаблонного методу в системі інфопакет.

Крім того гарним тоном під час написання програмного коду є використання тернарних виразів.

Тернарний вираз (від лат. ternarius – потрійний) – це вираз, що повертає свій другий або третій операнд в залежності від значення першого, який є логічним виразом. Такі вирази дають змогу значно скоротити кількість строк програмного коду та, в деяких випадках, покращити читабельність коду.

```
roles = user != null && user.role != null? new string[] { user.role.Name } : roles;
```

```
if (user != null && user.role != null)
{
    roles = new string[] { user.role.Name };
}
```

Рис. 3. Порівняння використання тернарного виразу і оператору розгалудження.

РОЗДІЛ ІІ. ПРОФІЛЮВАННЯ

Виходячи з вимог до розробленого програмного продукту, було проведено профілювання та пришвидшення роботи продукту.

Для пришвидшення роботи спочатку потрібно з’ясувати слабкі та повільні місця програми. В даній роботі було використано інструменти розробника google chrome та дебагер microsoft visual studio.

2.1 Введення метрики оцінювання оптимізації

Одним із місць, які можуть бути пришвидшені є взаємодія із базою даних. Оскільки сервер постійно звертається до БД за для того, щоб добути різного роду інформацію, пришвидшивши хоча б на декілька відсотків доступ до БД ми значно скоротимо час очікування для користувача.

Для того, щоб оцінити пришвидшення роботи, спочатку потрібно оцінити поточний стан системи та швидкість опрацювання запитів. Оскільки розробка програмного забезпечення робота складна, потрібно об’єктивно оцінювати вплив введених змін на швидкодію програмного продукту. Деякі зміни, покликані покращити швидкодію, при неправильному застосуванні можуть навпаки зменшити ефективність роботи системи. Тому необхідним є введення метрик оцінювання за їх замір.

Метриками пришвидшення можна обрати час відгуку мілісекундах (ms) та об’єм переданих даних в кілобайтах (kB). Зменшення об’єму переданих даних пришвидшить роботу системи, оскільки проводити необхідні операції потрібно для кожної одиниці даних. Також можна пришвидшити роботу за допомогою оптимізації написаного коду, чи за допомогою введення нового необхідного функціоналу.

Оцінювання оптимізації в даній роботі було виконано за допомогою інструментів розробника google chrome. Обраний інструмент показує загальний

час відгуку від серверу і більше концентрується на замірюванні швидкості відрисовки інтерфейсу. Даний засіб краще підходить для оптимізації frontend розробки, ніж для оптимізації запитів до БД, чи backend розробки. Але такий підхід достатній для базової оцінки оптимізації backend розробки.

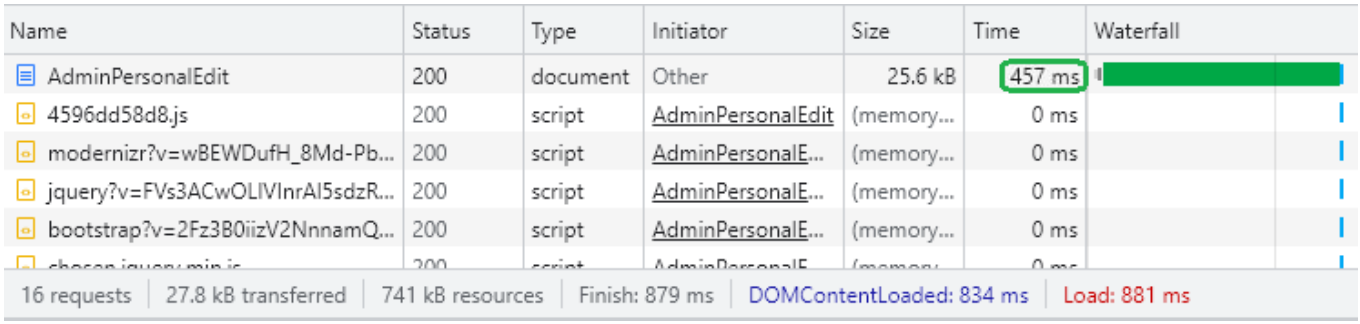

Рис. 4. Час очікування відклику сторінки персоналу до оптимізації.

2.2 Пришвидшення роботи шляхом застосування Eager loading та введення асинхронних запитів.

Після заміру необхідних даних, можна починати проводити оптимізацію. Для оптимізації роботи, систему було переведено на жадібне завантаження а також додано кешування даних з БД.

Якщо потрібно завантажувати додаткову інформацію з інших таблиць з даними, можна використовувати три підходи:

- Eager loading (жадібне завантаження) – завантажує всю обрану інформацію при запиті до основної сутності.
- Explicit loading (явне завантаження) – завантажує інформацію явно, в місцях де викликається спеціальний метод Load().
- Lazy loading (ліниве завантаження) – завантажує інформацію після того, як відбувається запит до конкретної додаткової сутності.

Розглянемо сутність навчального предмета. При завантаженні сторінки предмету завантажуються додаткові сутності, які містяться в інших таблицях бази даних, серед яких: список рівнів кваліфікацій, список галузей знань, можливі форми навчання, можливі гаранти освітньої програми та інші. При лінивому завантаженні підвантаження даних сутностей відбудеться тоді, коли вони потребуються. При Explicit loading завантаження сутності відбудеться саме тоді, коли буде потрібно, тобто це ручне керування завантаженням даних. При жадібному завантаженні обрані пов’язані з навчальним предметом сутності будуть завантажені тоді ж, коли завантажується сама сутність предмета.

На перший погляд здається, що використання лінивого завантаження найкращий вибір, оскільки ми не завантажуємо непотрібні сутності, таким чином зменшуючи об’єм завантажувальних даних, але на справді, при відрисовці інтерфейсу сторінка вже має отримати всі необхідні дані, включаючи додаткові сутності, тому, щоб не створювати додаткові окремі

запити, краще використати саме жадібне завантаження та добути всю необхідну інформацію за один запит.

Для застосування Lazy loading використовується ключове слово Find(), в нашому ж випадку потрібно використовувати ключове слово Include(), щоб явно вказати, що потрібно використовувати жадібне завантаження.

```
        private ViewModelSubject Show(int subjectId)
        {
            nr12_informpaketEntities db = new nr12_informpaketEntities();
-           var subject = db.Subjects.Find(subjectId);
+           var subject = db.Subjects
+               .Include("lector_assignments")
+               .Include("Course")
+               .Include("ECTSCredit")
+               .Include("education_cycle")
+               .Include("semester1")
+               .Include("StudyYear")
+               .FirstOrDefault(x => x.ID == subjectId);
            ViewBag.GeneralPublishYear = Convert.ToInt32(db.admin_settings.Find(1).PropValue);

            return new ViewModelSubject(subject);
```

Рис. 5. Заміна лінивого завантаження на жадібне завантаження в проєкті.

Крім того можливим способом пришвидшення системи є використання асинхронних запитів до сервісів чи бази даних.

Асинхронні запити використовуються для виконання процесів в паралельному режмі. Таким чином додатковий потік продовжує обробку інформації. В мові C# асинхронні запити позначаються ключовими словами async та await.

В системі інфопакет такі запити використовуються під час отримання інформації з бази даних та в зверненні до зовнішніх сервісів. Відклик сервісу та запити до бази даних можуть займати досить велику кількість часу, в цей час сторінка очікує. Інший потік може відмальовувати сторінку, поки відбуваються запити інший потік видобуває необхідну інформацію.

```
if(progProfile != null)
{
    string progProfileFileName = s3FolerName + "/"
   + sysABBR + "/"
   + "onp" + course.ID + "/"
   + "EduProgProfile" + course.ID + ".pdf";
    await S3FileUpload(progProfile, progProfileFileName);

    course.ProgramProfileURL = s3ServerUrl
        + progProfileFileName;
}
private async static Task S3FileUpload(HttpPostedFileBase file, string filePath)
{
    if (file.ContentLength > 0)
    {
        // Make a bucket on the server, if not already present.
        bool found = await minio.BucketExistsAsync(bucketName);
        if (!found)
        {
            await minio.MakeBucketAsync(bucketName, location);
        }
        // Upload a file to bucket.
        await minio.PutObjectAsync(bucketName, objectName,
              file.InputStream, file.ContentLength, contentType);
    }
    return;
}
```

Рис. 6.. Приклад асинхронного запиту до серверу зберігання фотографій

```
// Save form fields to as a DB record.
db.Entry(courses).State = EntityState.Modified;
await db.SaveChangesAsync();
```

Рис. 7.. Приклад асинхронного запиту до бази даних.

Попри те, що асинхронність може пришвидшити швидкодію, інколи її введення не дає очікуваних результатів. В деяких випадках виконання програмного засобу має відбуватися почергово, другий потік має спочатку отримати результат виконання першого потоку, тому в будь-якому випадку буде очікуват, поки він відпрацює. Таким чином, навіть якщо було введено в роботу асинхронність, виконання все одно буде почерговим.

Але при цьому асинхронний функціонал може робити програмний код більш заплутаним та збільшити важкість налагодження. Крім того не правильне використання асинхронних запитів може призводити до виникнення непередбачуваних результатів, коли для прикладу може створюватися декілька екземплярів об’єкту, який має бути єдиним в системі. Тому важливо розуміти концепт використання даного функціоналу.

2.3. Використання кешування для прискорення завантаження сторінки

Кеш (з англійської cache – схованка, ховати) – це проміжний буфер (швидкісна пам’ять або частина оперативної пам’яті) із можливістю швидкого доступу до вмісту. Кеш в пам’яті може зберігати будь-який об’єкт.

Окрім вказання явного способу завантаження та асинхронних запитів в проєкт було додано серверне кешування. Серверне кешування раз в 1740 хвилин записує в пам’ять серверу дані з БД, при зверненні до цих даних. Тобто, якщо сторінка є частовідвідуємою, то сервер запише до пам’яті дані, які були видобуті з бази даних і при повторному зверненні до них не буде відправляти запит до бази даних, а видобуде їх зі своєї пам’яті. Таким чином можна замінити повільні запити до бази даних на більш оперативну роботу з кешованою пам’яттю, оскільки звернення до БД завжди потребує більше часу, ніж робота з кешем.

Для застосування кешування в проєкті використовується клас memoryCache, який є частиною бібліотеки System.Runtime.Caching. Даний клас являє собою список з ключем, тобто клас Dictionary[5].

В системі інфопакет відбувається перевірка, чи існує запит в кеші, якщо його немає, то дані отримані з БД передаються до кешу, звідки його можливо буде отримати за спеціальним ключем в разі наступного звернення за тією ж інформацією.

```
MemoryCache memoryCache = MemoryCache.Default;
IEnumerable<SelectListItem> result = memoryCache.Get("12") as IEnumerable<SelectListItem>;

if (result == null)
{
    result = new SelectList(db.ECTSCreditsCourses, "ID", "Value").OrderBy(a => a.Text);
    memoryCache.Add("12", result, DateTime.Now.AddMinutes(1740));
}
return result;
```

Рис. 8. Додавання кешування в проєкт.

На жаль, якщо сторінка є рідковідвідуємою, то даний спосіб лише зменшить швидкість виконання програми, саме кешування займає час під час першого відвідування сторінки. Тому використовуватиданий підхід потрібно лише в потрібних місцях, які є частовідвідуємими. Для виявлення таких місць потрібно розуміти який функціонал системи є головним, а який будуть застосовувати рідше.

Крім того, об’єм кешованої пам’яті має обмеження, тобто неможливо передати до кешу абсолютно всю інформацію, яку коли небудь потрібно буде добути з бази даних. Найкраще застосовувати кешування для інформації, яка рідко змінюється. Та обов’язково потрібно обмежувати кількість записів до кешу, оскільки його переповнення може призвести до збою в системі.

2.4. Оцінка результату роботи.

Після введення необхідних змін необхідно оцінити проведену роботу в цілому в системі, оцінити пришвидшення завантаження сторінок.

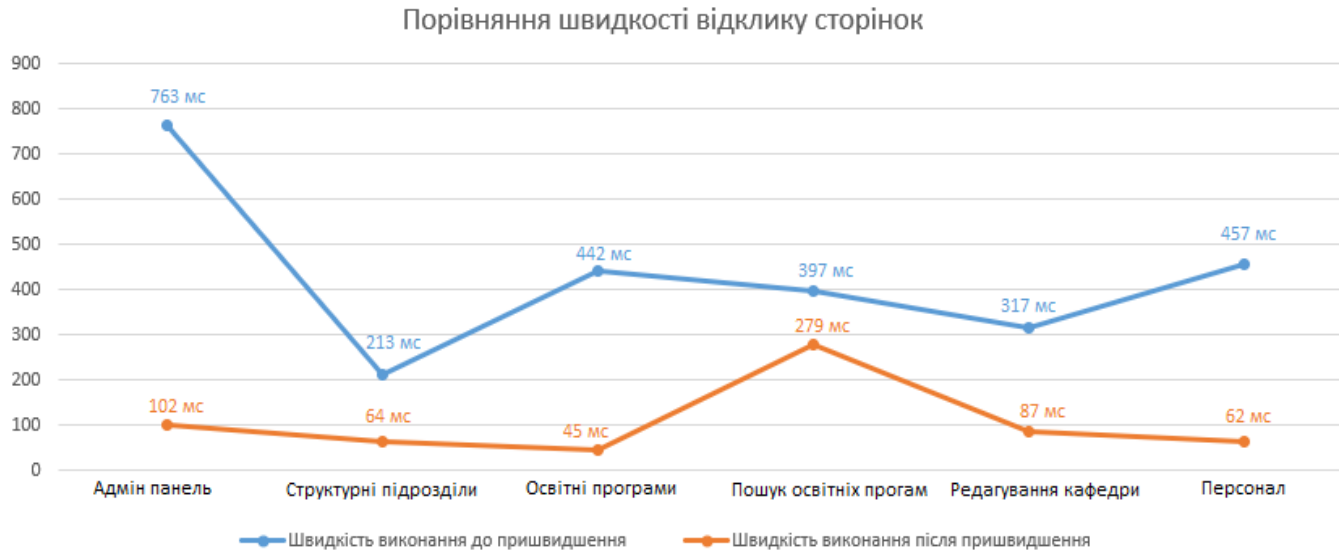

Рис. 9. Час очікування сторінок відгуку після оптимізації.

Під час заміру результатів пришвидшення роботи було проведено оцінку роботи шести сторінок системи. В середньому система була пришвидшена в 5 разів. Пришвидшення деяких сторінок дала значний приріст швидкості, як на приклад сторінки адміністративної панелі, яку було пришвидшено в 7.5 разів, але сторінка пошуку освітніх програм не була пришвидшена так сильно. Результат залежить від особливостей функціоналу та даних, якими оперує сторінка. Сторінки, які застосовують кешування, використовують асинхронні запити значно покращили свою швидкодію. В свою чергу сторінки, які більше опиралися на обробку отриманих результатів, ніж на їх отримання, такі як пошук освітніх програм було прискорено в 1.4 рази.

РОЗДІЛ ІІІ. АНАЛІЗ КОРЕКТНОСТІ ВИКОНАННЯ ТА ВПРОВАДЖЕННЯ ПРОДУКТУ

Під час розробки програмного продукту важливою частиною процесу є тестування розробленого продукту на відповідність поставлених вимог до програмного забезпечення та вимог написання програмного коду. Процес розробки зазвичай включає в себе інформацію щодо того яким чином має відбуватися тестування та які засоби при цьому будуть застосовуватися.

3.1. Тестування програмного забезпечення

Тестування програмного забезпеценння – це процес дослдження відповідності програмного продукту до його вимог та правильність виконання всіх поставлених до нього задач. Тестування виконуються в ізольованому від випливу інших факторів середовищі за допомогою певного набору тестів, обраних відповідно до тестувального функціоналу[6].

Тести можуть включати, але не обов’язково обмежуватися, наступними діями:

- Тестування правильності виконання програмного продукту.
- Тестування швидкодії та оптимальності написаного коду.
- Тестування відповідності між поставленою задачою та реальною роботою ПЗ.
- Перевірка можливих збоїв та поведінки при цих збоях.
- Процес нагляду за роботою програми в спеціальному середовищі з попередньо визначеними тестами.
- Оцінка програмного засобу додатковими сервісами, що оцінюють оптимальність виконання.

- Оскільки процес тестування може бути нескінченно довгим, навідь для нескладних систем, тому оцінювання системи відбувається за заздалегідь визначеним планом виходячи з наявних вимог та ресурсів.

3.1.1. Тестування методом більої скриньки

Тестування методом більої скриньки – це процес дослідження, коли тестувальнику відомий процес розробки ПЗ, при цьому тестуються тільки окремі ізольовані частини системи. При даному тестуванні досліджуються результати роботи окремих методів, їх справність, тобто внутрішня інфраструктура[7].

Переваги використання методу білої скриньки:

- Дослідження відбувається беспосередньо на найнижчому рівні, що гарантує перевірку правильності роботи структурних елементів.
- Перевіряються всі результати операторів розгалудження.
- Перевіряються справність системи при можливих небезпечних діапазонів значень.

Недоліки використання методу білої скриньки:

- Перевірка даним методом не гарантує правильності кінцевого результату, що відобразиться в інтерфейсі користувача.
- Важкість тестування всіх випадків, оскільки їх може бути занадто велика кількість для тестування.

Під час проведення тестування методом білої скриньки в каталозі курсів інфопакет було виявлено декілька небезпечних діапазонів значень об’єктів. Прикладом таких значень може бути значення NULL. При отримання цього значення система поводилася некоректно. Обраний метод

надав можливість знайти та виправити помилку небезпечного діапазону ще на стадії розробки.

```
<div class="card-personel">
    <div class="card-title">
        @item.personel_data.LastOrDefault().FName
        @item.personel_data.LastOrDefault().MName
        @item.personel_data.LastOrDefault().LName
    </div>
    <div class="card-container">
        Посада: @item.personel_data.LastOrDefault().Rank,
                Email: @item.Email , Телефон: @item.TellNom
@if (item.personel_data.LastOrDefault() != null)
 {
     <div class="card-personel">
         <div class="card-title">
             @item.personel_data.LastOrDefault().FName
             @item.personel_data.LastOrDefault().MName
             @item.personel_data.LastOrDefault().LName
         </div>
         <div class="card-container">
             Посада: @item.personel_data.LastOrDefault().Rank,
                     Email: @item.Email , Телефон:@item.TellNom
         </div>
```

Рис. 10. Додання перевірки на нульові значення

3.1.2. Тестування методом сірої скриньки

Тестування методом сірої скриньки – це процес дослідження, коли тестувальнику відомий процес роботи ПЗ і тестування відбувається за допомогою інтерфейсу. В цьому випадку відбувається дослідження коректності внутрішніх елементів та їх взаємодія між собою.

Часто за для таких перевірок можливе використання спеціального адміністративного інтерфейсу. Тоді тестувальник може бачити те, чого не буде бачити простий користувач. Такий підхід надає більш детальну інформацію про систему для інтерфейсу розробника і полегшує проведення тестуавння.

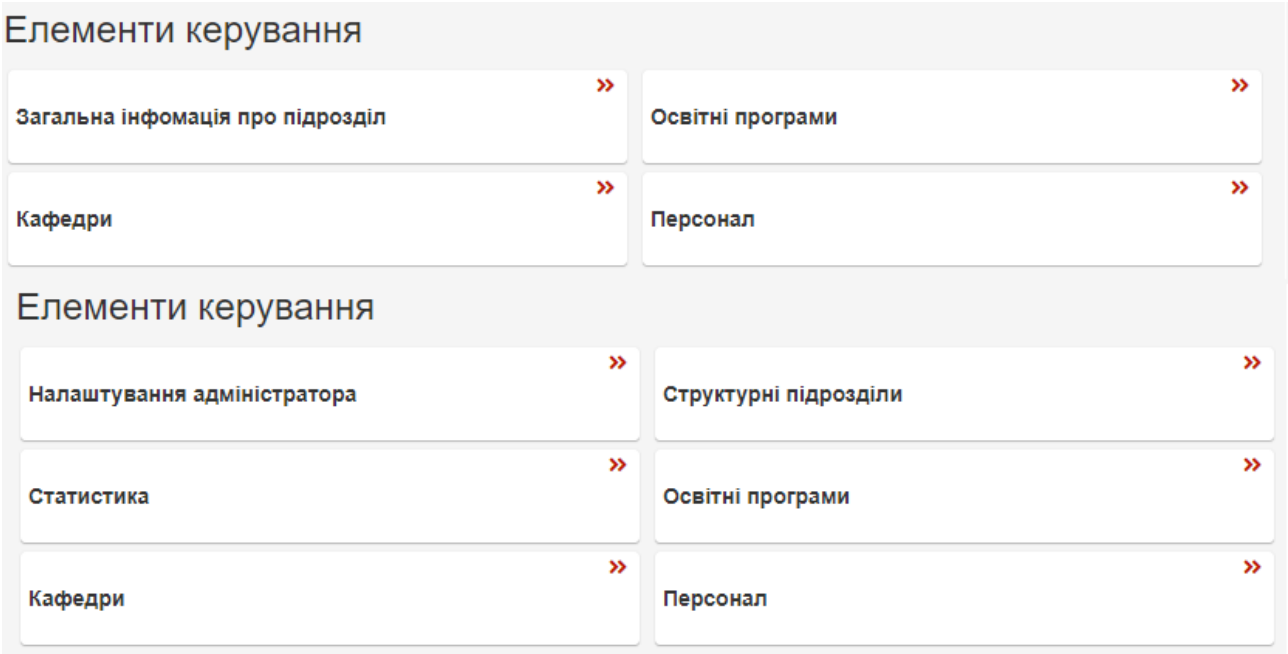

Рис. 11. Різниця інтерфейсів адміністративної панелі для адміністратора та суперадміністратора. Зверху інтерфейс адміністратора, Знизу інтерфейс суперадміністратора.

На рисунку 11 та 12 наведена різниця між інтерфейсами користувача системи, а саме адміністратора, який буде вводити інформацію для конкретного структурного підрозділу та суперадміністратору, який має слідкувати за корректністю роботи програмного продукту. Суперадміністратор має доступ до будь-якого структурного підрозділу та розширені можливості для редагування інформації.

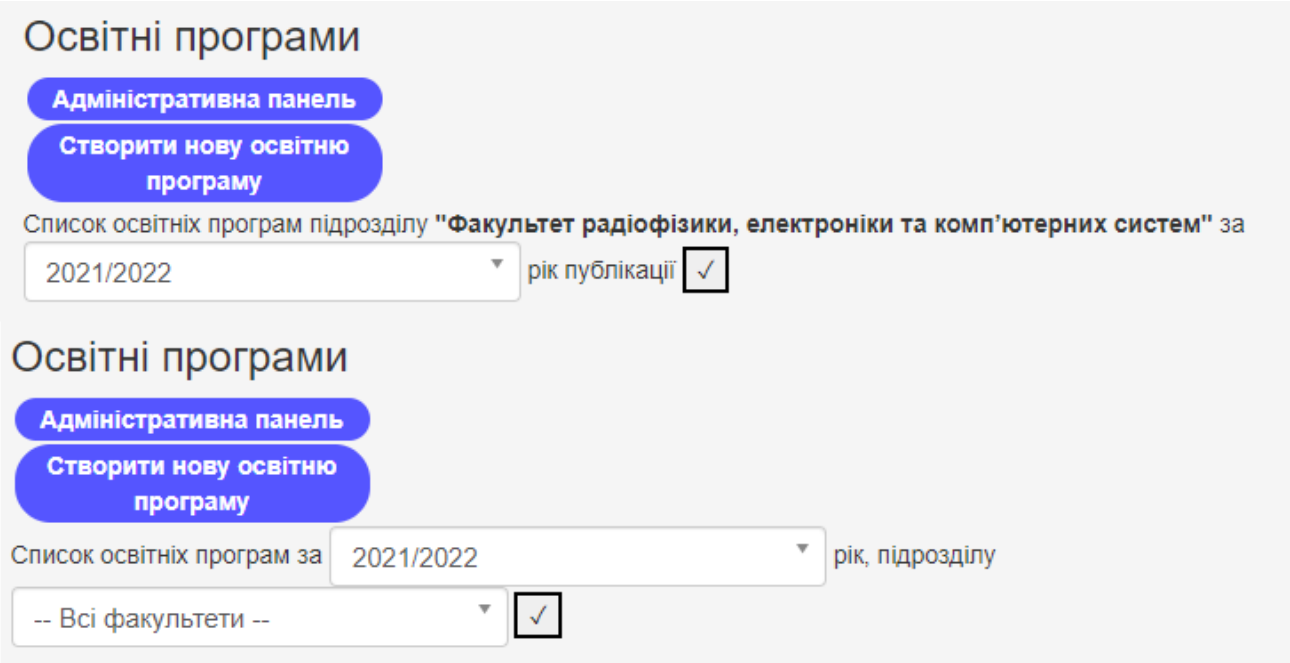

Рис. 12. Різниця інтерфейсів редагування освітніх програм для адміністратора та суперадміністратора. Зверху інтерфейс адміністратора, Знизу інтерфейс суперадміністратора.

Таким чином, знаючи деталі реалізації програмного продукту а також маючи доступ до адміністративного інтерфейсу можливо проводити тестування функціоналу системи методом сірої скриньки.

Такий підхід має ряд переваг та недоліків:

Переваги використання методу сірої скриньки:

- Даний підхід дає можливість знайти помилки, знаючи слабкі місця системи.
- Можливість виявлення та виправлення помилок ще на стадії розробки, даючи можливість знаходження більш глибоких помилок на стадії подальшого тестування.

Недоліки використання методу сірої скриньки:

- Знаючи як правильно має працювати система тестувальник може проводити заздалегідь упереджені тести, не очікуючи інших варіантів поводження.
- Не маючи адміністративного інтерфейсу тестувальник потребує значно більшого часу для тестування застосовуючи різні випадки дослідження.

Для прикладу розглянемо одну з помилок, виявлену на стадії розробки ПЗ під час тестування методом сірої скриньки.

При дослідженні інтерфейсу створення персоналу було знайдено вразливість. Адміністратор мав змогу обрати будь-який структурний підрозділ, таким чином додавати персонал до інших підрозділів. Користувач міг подумати, що функціонал працює некоректно, оскільки додавання викладачів до його підрозділу не відбувалося, але на справді сторінка додавалася б до інших факультетів, чи інститутів, до яких адміністратор не має доступу.

```
ViewData["personel.DepartmentID"] = new SelectList(db.Departments, "ID", "Name")
                                                        .OrderBy(a => a.Text);

if (User.IsInRole("admin"))
{
    int? faculId = db.users.FirstOrDefault(m => m.Email == User.Identity.Name).FaculID;
    ViewData["personel.DepartmentID"] = new SelectList(db.Departments
        .Where(x => x.FacultyID == faculId), "ID", "Name").OrderBy(a => a.Text);
}
else if (User.IsInRole("superadmin"))
{
    ViewData["personel.DepartmentID"] = new SelectList(db.Departments, "ID", "Name")
                                                        .OrderBy(a => a.Text);
}
```

Рис. 13. Розв'язання вразливості при створенні сторінок викладачів.

Даний метод допоміг знайти та виправити помилку ще на стадії розробки, скориставшись інтерфейсом та заздалегідь знаючи, яка поведінка мала бути у функціоналу.

3.1.3. Тестування методом чорної скриньки

Тестування методом чорної скриньки – це процес дослідження програмного продукту, коли тестувальнику невідомий процес роботи програми, тестувальник бачить та може взаємодіяти лише з інтерфейсом системи. Таким чином користувач не знає як має прцювати засіб, але користується ним та може проводити ряд тестів спрямованих на результат роботи. Даний метод названий чорною скринькою, оскільки тестувальник не знає яким чином дані мають оброблятися та зберігатися, а тільки який результат очікується наприкінці тестування.

Такий підхід не дає можливості повної перевірки всього функціоналу, оскільки якщо, для прикладу, продукт має 10 тестувальних форм і для кожної потрібно 10 прикладів вхідних даних, то кількість тестів, необхідних для тестування всіх варіантів становитиме 10^{10} тестових зразків. Отже метод чорної скриньки зосереджується не на перевірці всіх можливих вхідних даних, а на перевірці потенційно небезпечних місць в системі.

Переваги використання методу чорної скриньки:

- Такий підхід дає можливість проводити тестування без упередженості знаючи тільки який результат очікується в результаті виконання.
- Надає можливість виявлення помилок в користувацькому інтерфейсі, роблячи його зручним для використання.
- Тестувальник не обов’язково повинен мати технічні знання для виконання тестувань

Недоліки використання методу чорної скриньки:

- Неможливо виконати тестування в повному обсязі, оскільки для перевірки всіх тестувальних випадків потребується занадто багато тестів.

Під час розробки каталогу курсів університету Шевченка тестування проводилося за участю факультету соціології Київського Національного Університету імені Тараса Шевченка. Комунікація відбувалася за допомогою системи Zammad. Тестувальники використовували метод чорної скриньки, оскільки вони не мали доступу до програмного коду системи.

Прикладом знайденої та вирішеної проблеми може слугувати покращення інтерфейсу користувача. Тестувальники звернули увагу, що при редагуванні дисципліни рік програми не зберігається і при поверненні потрібно знову виставляти поточний рік. Зберігання року в пам’яті системи могло б додатково обтяжити систему, тому як вирішення задачі було обрано споіб виставлення поточного року, в такий спосіб користувачі, які зазвичай редагують у великому об’ємі саме поточний рік не отримували б ускладнень, а система не була б переобтяжена потребою в збереженні надлишкової інформації.

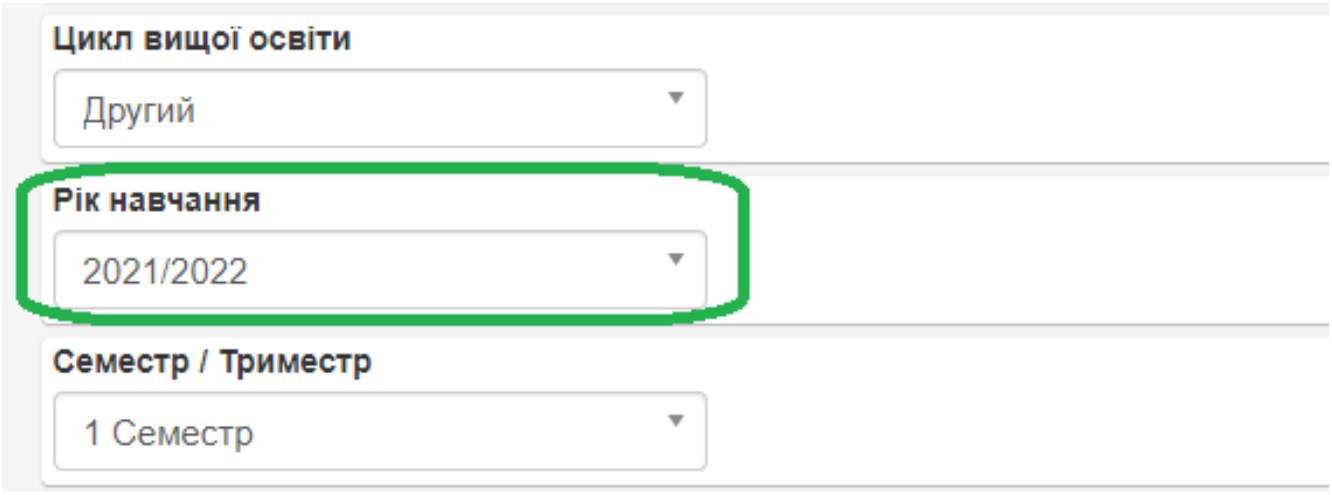

Рис. 14. Оновлений інтерфейс редагування дисциплін.

```
<div class="card-container">
    @Html.DropDownList("StudyYearID", Model.ListForYearOfStudy, htmlAttributes: new { @class = "form-
    control list" })
    @Html.DropDownList("StudyYearID", Model.ListForYearOfStudy.OrderByDescending(x => x.Value),
     htmlAttributes: new { @class = "form-control list" })
</div>
```

Рис. 15. Оновлений програмний код для редагування дисциплін. Червоним показано програмний код, що використовувався, а зеленим – поточний.

Ще один приклад вирішення помилки, який було виявлено під час тестування методо чорної скриньки. Тестувальники не могли з точністю знати, який саме формат файлів повинен використовуватися для структурно-логічної схеми освітньої програми, бо інструкції для заповнення могли відрізнятися від одного структурного підрозділу до іншого, тому намагалися завантажувати файли в різних форматах. Під час таких завантажень було виявлено некоректну роботу системи, яка призводила до помилок в подальшому.

За для розв’язання виявлених помилок було обмежено можливість завантаження файлів з різними форматами та обмежено можливістю використання тільки документів в форматі PDF.

```
<div class="card-container">
    <input type="file" name="progDiagram" />
    <input type="file" accept="application/pdf" name="progDiagram" />
</div>
```

Рис. 16. Обмеження можливостей завантаження невідповідних форматів. Червоним показано програмний код, що використовувався, а зеленим – поточний.

Даний метод допоміг виявити та вирішити проблеми, не передбачені під час розробки програмного засобу, але які могли викликати проблеми під час впровадження системи в подальшому. Саме такі проблеми покликаний вирішувати метод чорної скриньки. Неупередженість до системи тестувальниками надають їм змогу дати правильну оцінку програмному продукту та виявити ряд недоліків.

3.2. Впровадження системи

Тестування програмного засобу є важливою складовою розробки якісного продукту, але способи тестування не є досконалими. Під час тестування неможливо виявити абсолютно всі недоліки системи, частина з них може проявитися під час впровадження системи, особливо на ранніх стадіях.

Під час впровадження системи інфопакет було виявлено та вирішено ряд недоліків системи. Комунікація з адміністраторами структурних підрозділів відбувалася за допомогою системи Zammad. За для безпечності роботи над системою було прийнято рішення прибрати можливість видалення співробітників, оскільки навіть після звільнення співробітника в системі мають зберігатися дані про його роботу. Тому було додано можливість архівувати сторінку співробітника, але даний функціонал міг використовувати лише суперадміністратор, бо безконтрольне архівування могло нашкодити цілісності системи. Але під час впровадження було отримано велику кількість запитів на надання цієї можливості для адміністраторів. Користувачі випадково дублювали сторінки з викладачами та не мали можливості видалення. Тому було прийнято рішення повернути можливість архівування викладачів та додано можливість їх поновлення.

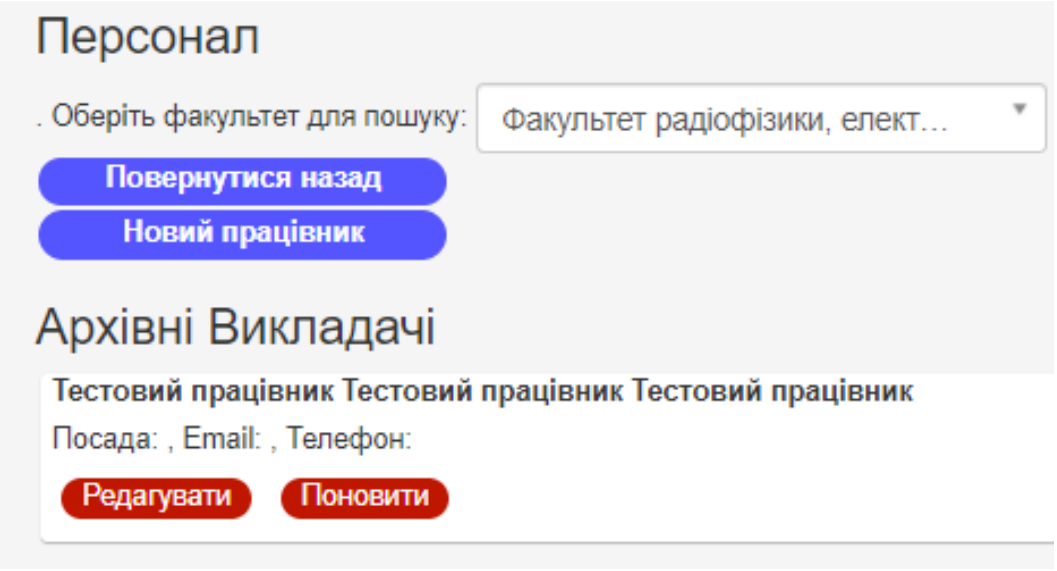

Рис. 17. Інтерфейс архівних викладачів, видимий лише для суперадміністраторів

Видалені сторінки не були видимі для адміністраторів, їм не потрібно було мати можливість роботи з архівними викладачами, а їх поновленням могли займатися суперадміністратори.

```
    if (personel != null)
    {
        personel.Archived = false;
        db.Entry(personel).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return RedirectToAction("Index");
```

```
<a href="@Url.Action("Delete", new { id = item.PersonelID })">
    <div class="card-button">
        Видалити
    </div>
</a>
```

```
@if (User.IsInRole("superadmin"))
{
    <a href="@Url.Action("Delete", new { id = item.PersonelID })">
    <a href="@Url.Action("Restore", new { id = item.PersonelID })">
        <div class="card-button">
            Видалити
            Поновити
        </div>
    </a>
}
```

Рис. 18. Програмний код для архівування та відновлення сторінок персоналу.

Під час впровадження системи було отримано та виправлено 86 запитів.

ВИСНОВКИ

Потреба в оптимізації та пришвидшенні роботи програмного продукту є життєво важливою для великої кількості систем і інфопакет не є виключенням. За допомогою введення нового потрібного функціоналу було досягнуто пришвидшення роботи системи а також зменшення повторюваності коду та збільшення використання головних принципів розробки програмного забезпечення.

Вдалося добитися збільшення швидкодії виконання шести досліджуємих сторінок в середньому в 5 разів.

Під час розробки системи було виконано аналіз та тестування системи на справність.

Розроблений програмний засіб був введений в дію з використанням ресурсів інформаційно-обчислювальному центру Київського національного університету імені Тараса Шевченка.

На даний момент створена система перебуває на стадії заповнення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Про вищу освіту: Закон України від 01.07.2014 р. № 1556-VII. Дата оновлення: 28.09.2017. Відомості Верховної Ради України. 2014. № 37-38. стор. 2716, стаття 2004;
2. Кристофер А. Язык шаблонов. Города, здания, строительство / Александер Кристофер. – Москва: Второе издание, 2020. – 1096 с. – (Издательство Студии Артемия Лебедева). – (ISBN: 978-5-98062-121-6).
3. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования = PHP Objects, Patterns and Practice, Third Edition. — 3-е издание. — М.: «Вильямс», 2015. — С. 368. — (ISBN 978-5-496-00389-6).
4. Martin, Robert C. (2000). "Design Principles and Design Patterns". Режим веб доступу до ресурсу: https://web.archive.org/web/20150906155800/http:/www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
5. Microsoft Docs. Cache in-memory in ASP.NET Core. Режим веб доступу до ресурсу: https://docs.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-6.0
6. ISO/IEC TR 19759:2015 (SWEBOOK)
7. "SOA Testing Tools for Black, White and Gray Box". Режим веб доступу до ресурсу: https://web.archive.org/web/20181001010542/http:/www.crosschecknet.com:80/soa_testing_black_white_gray_box.php

ДОДАТКИ

Додаток 1. Програмний код сторінки налаштувань персоналу (контролер)

```
nr12_informpaketEntities db = new nr12_informpaketEntities();
      [Authorize]
      public ActionResult Index(int? facultyId, int? departmentId)
      {
         ViewBag.facultyId = new SelectList(db.Faculties, "ID", "Name").OrderBy(a => a.Text);

         if (User.IsInRole("admin"))
         {
            int? userFacultyId = db.users.FirstOrDefault(m => m.Email == User.Identity.Name).FaculID;
            ViewBag.departmentId = new SelectList(db.Departments.Where(m => m.FacultyID == userFacultyId), "ID",
"Name").OrderBy(a => a.Text);
            if (departmentId != null)
            {
               var selectedPersonnel = db.personel_data
                  .Where(m => m.personel.Department.FacultyID == userFacultyId)
                  .Where(m => m.personel.DepartmentID == departmentId)
                  .Where(m => m.Year == db.personel_data.Where(b => b.PersonelID == m.PersonelID).Select(d =>
d.Year).Max());
               return View(selectedPersonnel.ToList());
            }
            else
            {
               var selectedPersonnel = db.personel_data
                  .Where(m => m.personel.Department.FacultyID == userFacultyId)
                  .Where(m => m.personel.DepartmentID == departmentId)
                  .Where(m => m.Year == db.personel_data.Where(b => b.PersonelID == m.PersonelID).Select(d =>
d.Year).Max());
               return View(selectedPersonnel);
            }
         }

         if (User.IsInRole("superadmin"))
         {
            ViewBag.departmentId = new SelectList(db.Departments, "ID", "Name").OrderBy(a => a.Text);
            if (facultyId != null && departmentId != null)
            {
               ViewBag.departmentId = new SelectList(db.Departments.Where(m => m.FacultyID == facultyId), "ID",
"Name").OrderBy(a => a.Text);
               var selectedPersonnel = db.personel_data
                  .Where(m => m.personel.Department.FacultyID == facultyId)
                  .Where(m => m.personel.DepartmentID == departmentId)
                  .Where(m => m.Year == db.personel_data.Where(b => b.PersonelID == m.PersonelID).Select(d =>
d.Year).Max());
               return View(selectedPersonnel.ToList());
            }
            else if (facultyId == null && departmentId != null)
            {
               var selectedPersonnel = db.personel_data
                  .Where(m => m.personel.DepartmentID == departmentId)
                  .Where(m => m.Year == db.personel_data.Where(b => b.PersonelID == m.PersonelID).Select(d =>
d.Year).Max());
               return View(selectedPersonnel.ToList());
            }
            else if (facultyId != null && departmentId == null)
            {
```

```
                ViewBag.departmentId = new SelectList(db.Departments.Where(m => m.FacultyID == facultyId), "ID",
"Name").OrderBy(a => a.Text);
                var selectedPersonnel = db.personel_data
                   .Where(m => m.personel.Department.FacultyID == facultyId)
                   .Where(m => m.Year == db.personel_data.Where(b => b.PersonelID == m.PersonelID).Select(d =>
d.Year).Max());
                return View(selectedPersonnel.ToList());
            }
            else
            {
                var selectedPersonnel = db.personel_data
                   .Where(m => m.personel.DepartmentID == departmentId)
                   .Where(m => m.Year == db.personel_data.Where(b => b.PersonelID == m.PersonelID).Select(d =>
d.Year).Max());
                return View(selectedPersonnel);
            }
        }
        return View();
          }
```

Додаток 2. Програмний код сторінки налаштувань персоналу (представлення)

```
@model IEnumerable<InfPack_EF_DBF.Models.personel_data>
@{
    ViewBag.Title = "Адмін. Персонал";
}

<div class="main-content">
    <h3>Персонал</h3>

    @if (User.IsInRole("superadmin"))
    {
        <form class="list-filter-input" action="@Url.Action("Index")" method="get">
            <div>. Оберіть факультет для пошуку: </div>
            @Html.DropDownList("facultyId", null, "--Виберіть структурний підрозділ--", htmlAttributes: new { @class =
"list" })
            <div>Оберіть кафедру для пошуку:</div>
            @Html.DropDownList("departmentId", null, "--Виберіть Кафедру--", htmlAttributes: new { @class = "list" })
            <input class="input-submit" type="submit" value="✓" />
        </form>
    }
    @if (User.IsInRole("admin"))
    {
        <form class="list-filter-input" action="@Url.Action("Index")" method="get">
            <div>Оберіть кафедру для пошуку:</div>
            @Html.DropDownList("departmentId", null, "--Виберіть Кафедру--", htmlAttributes: new { @class = "list" })
            <input class="input-submit" type="submit" value="✓" />
        </form>
    }

    <!--breadcrumbs-->
    <div style="max-width:205px" class="ui-data-block">
        <a class="ref-block-link" href="@Url.Action("Index", "AdminPanel")">
            <div class="page-button">
                Повернутися назад
            </div>
        </a>
    </div>
    <!--Controll buttons-->
    <div style="max-width:205px" class="ui-data-block">
        <a class="ref-block-link" href="@Url.Action("Create")">
            <div class="page-button">
                Новий працівник
            </div>
        </a>
    </div>
    <!--List of tutors-->
    <div class="ui-data-block">
        @foreach (var item in Model.Where(n => n.personel.Archived == false))
        {
            <div class="card-large card-large-personel">
                <div class="card-image">
                    @if (item.personel.Image != null)
                    {
                        <img class="image-personel" src="@item.personel.Image" />
                    }
                </div>
                <div class="card-personel">
                    <div class="card-title">
                        @item.FName
                        @item.MName
                        @item.LName
                    </div>
                    <div class="card-container">
```

```
                Посада: @item.Rank, Email: @item.personel.Email , Телефон: @item.personel.TellNom
            </div>

            <div class="card-container">
                <a href="@Url.Action("Edit", new {id = item.PersonelID})">
                    <div class="card-button">
                        Редагувати
                    </div>
                </a>
                <a href="@Url.Action("Delete", new { id = item.PersonelID })">
                    <div class="card-button">
                        Видалити
                    </div>
                </a>
            </div>
        </div>
    </div>
}
@if (User.IsInRole("superadmin"))
{
    <h3>Архівні Викладачі</h3>
    foreach (var item in Model.Where(n => n.personel.Archived == true))
    {
        <div class="card-large card-large-personel">
            <div class="card-image">
                @if (item.personel.Image != null)
                {
                    <img class="image-personel" src="@item.personel.Image" />
                }
            </div>
            <div class="card-personel">
                <div class="card-title">
                    @item.FName
                    @item.MName
                    @item.LName
                </div>
                <div class="card-container">
                    Посада: @item.Rank, Email: @item.personel.Email , Телефон: @item.personel.TellNom
                </div>

                <div class="card-container">
                    <a href="@Url.Action("Edit", new {id = item.PersonelID})">
                        <div class="card-button">
                            Редагувати
                        </div>
                    </a>
                    @if (User.IsInRole("superadmin"))
                    {
                        <a href="@Url.Action("Restore", new { id = item.PersonelID })">
                            <div class="card-button">
                                Поновити
                            </div>
                        </a>
                    }
                </div>
            </div>
        </div>
    }
}
</div>
</div>

@section scripts
{
    @if (User.IsInRole("admin"))
    {
```

```
    <script>
        $(function () {
            $("#departmentId").chosen({ width: '320px' });
        })
    </script>
}

@if (User.IsInRole("superadmin"))
{
    <script>
        $("#departmentId").chosen({ width: '280px' });

        $("#facultyId").chosen({ width: '280px' }).change(function () {
            if ($('#facultyId').find("option:selected").text() == "--Виберіть структурний підрозділ--") {
                $("#departmentId option").remove();
                $("#departmentId").append("<option>--Виберіть Кафедру--</option>");
                $("#departmentId").trigger("chosen:updated");
            }
            else {
                $.get("/AdminPersonalEdit/GetDepsList", { facultyID: $('#facultyId').val() }, function (data) {
                    $("#departmentId").empty();
                    $("#departmentId").append("<option>--Виберіть Кафедру--</option>");
                    $.each(data, function (index, row) {
                        $("#departmentId").append("<option value='" + row.ID + "' >" + row.Name + "</option>");
                    })
                    $("#departmentId").trigger("chosen:updated");
                });
            }
        });

    </script>
}


<script src="~/Scripts/jquery-ui.min.js"></script>
<script>
$(function () {
$("#tabs").tabs({active: @ViewBag.stCode});

    });
</script>
        }
```

Додаток 3. Програмний код сторінки налаштувань персоналу (модель)

```
namespace InfPack_EF_DBF.Models
{
    using System;
    using System.Collections.Generic;

    public partial class personel_data
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2214:DoNotCallOverridableMethodsInConstructors")]
        public personel_data()
        {
            this.courses = new HashSet<Course>();
        }

        public int ID { get; set; }
        public string FName_EN { get; set; }
        public string MName { get; set; }
        public string MName_EN { get; set; }
        public string LName { get; set; }
        public string LName_EN { get; set; }
        public string Rank { get; set; }
        public string Rank_EN { get; set; }
        public Nullable<int> Year { get; set; }
        public int PersonelID { get; set; }
        public string FName { get; set; }

        public virtual personel personel { get; set; }
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Course> courses { get; set; }
    }
            }
```

Додаток 4. Програмний код для кешування рівнів кваліфікації предмету

```
public IEnumerable<SelectListItem> ListForQualificationID
    {
      get
      {
        MemoryCache memoryCache = MemoryCache.Default;
        IEnumerable<SelectListItem> result = memoryCache.Get("cacheQualification") as IEnumerable<SelectListItem>;
        if (result == null)
        {
          result = new SelectList(db.QualificationLists, "ID", "Name").OrderBy(a => a.Text);
          memoryCache.Add("cacheQualification", result, DateTime.Now.AddMinutes(1740));
        }
        return result;
      }
    }
```