МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**Український державний університет
науки і технологій**

Кафедра «Автоматика
та телекомунікації»

В авторській редакції

ОСНОВИ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-методичні рекомендації
до виконання індивідуального завдання

Електронне видання

ДНІПРО
2026

УДК 004.9(076.1)
О 75

Упорядники:
Р. В. Рибалка, К. В. Гончаров, Л. С. Тимошенко

Електронне видання

Схвалено Групами забезпечення якості освітніх програм
«Автоматика та автоматизація на транспорті» спеціальності
G7 «Автоматизація, комп'ютерно-інтегровані технології та
робототехніка» протокол № 4 від 30.04.2026
«Системи керування рухом поїздів» спеціальності J7
«Залізничний транспорт» протокол № 6 від 23.04.2026

О 72 Основи інформаційних технологій : навчально-методичні рекомендації до виконання індивідуального завдання / упоряд. Р. В. Рибалка, К. В. Гончаров, Л. С. Тимошенко; Укр. держ. ун-т науки і технологій. – Електрон.вид. – Дніпро : УДУНТ, 2026. – 58 с.

Навчально-методичні рекомендації призначено для використання здобувачами вищої освіти, які здобувають освітній ступінь «бакалавр» за освітньо-професійними програмами «Автоматика та автоматизація на транспорті» та «Системи керування рухом поїздів» під час виконання індивідуального завдання з навчальної дисципліни «Основи інформаційних технологій». Навчально-методичні рекомендації містять основні теоретичні відомості, постановку задачі, приклад виконання індивідуального завдання з поясненнями та питання для самоконтролю.

Іл. 9. Бібліогр.: 6 назв.

ЗМІСТ

ВСТУП

Характеристика місця та значення навчальної дисципліни для підготовки фахівця. Ці навчально-методичні рекомендації (далі – НМР) до індивідуального завдання складено для опанування навчальної дисципліни «Основи інформаційних технологій» (далі – Дисципліна), яка викладається відповідно до освітньо-професійних програм (далі – ОПП) «Автоматика та автоматизація на транспорті» (далі – ААТ) спеціальності G7 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка» та «Системи керування рухом поїздів» (далі – СКРП) спеціальності J7 «Залізничний транспорт».

Опанування Дисципліни дозволяє здобувачу, який здобуває освіту за ОПП ААТ, отримати навики використання інформаційних і комунікаційних технологій, здатність вільно користуватись сучасними комп'ютерними та інформаційними технологіями для вирішення професійних завдань, програмувати та використовувати прикладні та спеціалізовані комп'ютерно-інтегровані середовища для вирішення задач автоматизації, робототехніки та зв'язку. Опанування Дисципліни дозволяє здобувачу, який здобуває освіту за ОПП СКРП, отримати навики використання інформаційних і комунікаційних технологій; розвинути здатність до абстрактного мислення, аналізу та синтезу; створити основу для застосування сучасних програмних засобів для розробки проектно-конструкторської та технологічної документації зі створення, експлуатації, ремонту та обслуговування систем керування рухом поїздів, пристроїв залізничної автоматики та їх елементів, а також для організації дії системи звітності та обліку (управлінського, статистичного, технологічного) роботи систем керування рухом поїздів та пристроїв залізничної автоматики, здійснювати діловодство, документування та управління якістю згідно нормативно-правових актів, інструкцій та методик.

Мета індивідуального завдання з Дисципліни для цих НМР – сформувати у здобувача освіти (далі – Здобувач) навики: відповідно очікуваним результатам навчання, визначеним в робочій програмі Дисципліни, розробляти програму мовою програмування C# шляхом застосування основ об'єктно-орієнтованого програмування, зокрема шляхом реалізації сутностей з використанням відношень «успадкування» та «агрегація».

Вимоги до попередніх знань та умінь: знання типових операцій з файлами та папками (створення, перейменування, копіювання тощо), навики програмування мовою C# з використанням об'єктно-орієнтованого програмування, основ роботи у Visual Studio (або в іншому подібному середовищі). Використання технічних засобів: індивідуальне завдання (далі – ІЗ) виконується за допомогою комп'ютера.

Вимоги до оформлення звіту з ІЗ (далі – Звіт):

- Допустимі види оформлення Звіту: електронна, паперова.
- Формат аркушу Звіту: А4.

- У разі оформлення Звіту:
  - З використанням засобів комп’ютерної техніки: гарнітура основного тексту – Times New Roman; гарнітура тексту програм – Consolas, Cascadia або інша моноширинна; розмір шрифту – мінімум 10 пт.
  - Власноруч: допускається прикріплення додатків (рисунок, таблиця, текст програми тощо), які може бути надруковано за допомогою комп’ютерної техніки.
- Звіт повинен містити інформацію, яка однозначно ідентифікує його виконавця: прізвище та ім’я Здобувача, номер академічної групи, шифр Здобувача (номер індивідуального навчального плану), номер варіанту.
- Наповнення Звіту повинне відповідати вимогам розділу «Зміст звіту».

Розподіл роботи над виданням між співавторами:

- Рибалка Р. В. – вступ, розробка розділів «Основні теоретичні відомості», «Постановка задачі», третьої частини прикладу вирішення задачі, «Контрольні запитання», дод. 2 (разом з Гончаровим К. В.), 6;
- Гончаров К. В. – розробка розділу «Загальні пояснення до прикладу виконання індивідуального завдання», першої та другої частини прикладу вирішення задачі, дод. 2 – 5, 8;
- Тимошенко Л. С. – розробка розділу «Зміст звіту», четвертої частини прикладу вирішення задачі, дод. 1, 7, 9.

Визначення номера варіанту завдання

Номер варіанту індивідуального завдання позначається цілим числом, яке дорівнює остачі від ділення суми двох останніх цифр у шифрі Здобувача (номер індивідуального навчального плану) на загальну кількість варіантів завдань N, якщо не вказано іншого. *Приклад*:

- якщо сума двох останніх цифр у шифрі Здобувача дорівнює 18 і варіанти завдань мають номери в інтервалі 0…6 (загальна кількість $N = 7$);
- то номер варіанту дорівнює 4, що є остачею від ділення 18 на 7.

ВИМОГИ ДО ТЕХНІЧНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Індивідуальне завдання виконується з використанням комп’ютера. Вимоги до комп’ютера – установлено:

- інтегроване середовище розробки Visual Studio версії не раніше 2008 року з англійською локалізацією. Visual Studio Community 2022 доступно для завантаження за посиланням https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=Community

&channel=Release&version=VS2022&source=VSLandingPage&cid=2030&passive=false;

– .NET Framework версії не раніше 4.0 (доступно для завантаження на сторінці за посиланням https://dotnet.microsoft.com/en-us/download/dotnet-framework) або .NET (доступно для завантаження на сторінці за посиланням https://dotnet.microsoft.com/en-us/download).

ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Під час розроблення проекту за допомогою об'єктно-орієнтованого програмування розробники, як правило, намагаються створити проект зі слабким сполученням (loosely coupled design) між сутностями. У разі потреби внести зміни до такого проекту (наприклад, після надходження нових вимог, необхідності виправити дефекти тощо) це дозволяє зменшити витрати часу на внесення вказаних змін та зменшити ризик виникнення нових дефектів в програмі. Існують різні способи розроблення проекту зі слабким сполученням, наприклад: використання інкапсуляції, інтерфейсів, організації певних відношень між класами тощо. *Класом* (class) у мові програмування називається *шаблон* (тип даних) для об'єктів, що визначає внутрішню структуру й набір операцій для екземплярів таких об'єктів [1]. *Інтерфейс* C# (interface) визначає контракт [2].

Серед принципів розроблення програм, які допомагають створити проект зі слабким сполученням, є DRY (Don't Repeat Yourself – не повторювати себе), що заохочує розробників уникати створення дублікатів коду в проекті [3]. Тобто певну поведінку потрібно розмістити в тексті програми тільки в одному місці уникнувши дублювання. Це дозволяє створювати компоненти, код яких може бути використаним повторно (code reuse), наприклад, підпрограми чи модулі.

Принцип DRY може бути реалізованим, наприклад, за допомогою встановлення між сутностями відношення «*успадкування*» (inheritance), тобто копіювання всієї або частини внутрішньої структури й набору операцій з базового класу до похідного класу [1]). Це дозволяє розмістити у базовому класі певні можливості, які є загальними для похідних класів. У разі потреби змінити реалізацію можливості, яку декларовано у базовому класі, можна перевизначити (override) цю можливість у похідних класах, реалізувавши поліморфну поведінку.

Часто відношення «успадкування» є гарним вибором. Проте в загальному випадку воно призводить до створення єрархії з сильно сполучених класів (tight coupling). *Приклад*: існує єрархія, що складається з базового класу `A` та похідних від нього класів `B`, `C` та `D` (рис. 1). Можливості, які декларовано в

класі A, можуть бути доступними в класах B, C та D (наприклад, використовуючи модифікатори C# protected або public). Т.ч. можливості класу A є *загальними* для класів B, C та D.

Припустимо, що в класах C та D є можливості, які є однаковими (ідентичними за назвою та реалізацією). Ці можливості є *частково* загальними для єрархії на рис. 1, оскільки їх не декларовано в класі B. Переміщувати частково загальні можливості до базового класу A може бути недоцільним.

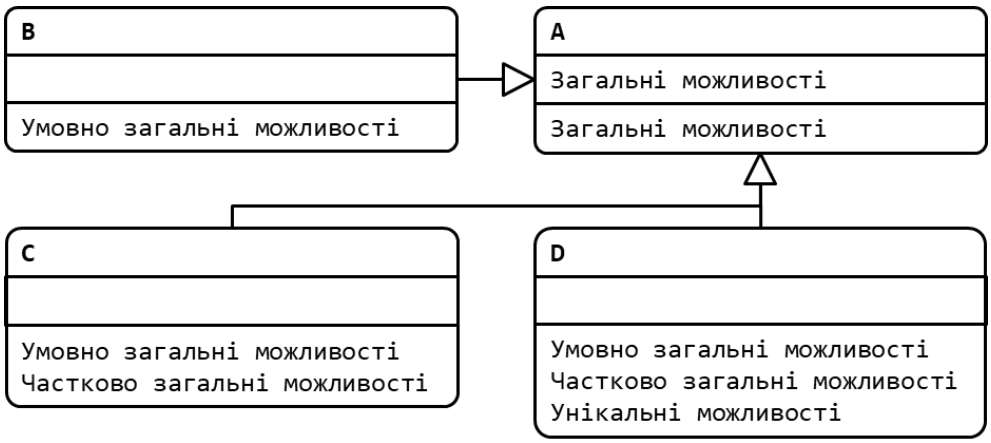


Рис. 1. UML діаграма сутностей під час успадкування

Якщо все ж перемістити їх до базового класу A, то в класі B доведеться реалізовувати «заглушки», тобто елементи, які можна викликати, але вони не міститимуть жодної поведінки (тексту програми), генеруватимуть виняток (exception) [4] тощо. Це рішення призведе до порушення принципу підстановки Ліскова (Liskov Substitution Principle) з переліку принципів SOLID [5], оскільки частково загальні можливості (рис. 1) мають смисл не для всіх похідних класів (для B – не мають смислу).

В єрархії на рис. 1 показано, що класи B, C та D можуть мати *умовно* загальні можливості, які ідентичні за назвою, але їх реалізація може відрізнятись (для кожної сутності – власна реалізація). На прикладі класу D показано (рис. 1), що можуть існувати й *унікальні* можливості, які відсутні в інших сутностях.

Для того, щоб створювати нові класи шляхом компонування можливостей інших класів, можна використати відношення «агрегація» із застосуванням впровадження залежностей. *Впровадження залежності* (dependency injection) – це шаблон проектування, який допомагає розробляти програму зі слабким сполученням, яку легко підтримувати. Найбільш часто залежності впроваджують в конструкторі. У цьому шаблоні, як правило, декларовано тільки один параметризований конструктор (конструктора за промовчанням немає), який очікує на об’єкт певного типу, що передано через параметр цього конструктора.

Якщо під час розроблення програми використано шаблон впровадження залежності через конструктор, то можна (див. рис. 2):

1. До параметрів конструкторів класів `B`, `C` та `D` передати екземпляр класу `A`, в якому декларовано можливості, що є *загальними* для класів `B`, `C` та `D`.

2. *Частково* загальні можливості класів `C` та `D` перемістити до нової сутності, наприклад, класу `E`.

3. До параметрів конструкторів класів `C` та `D` окрім екземпляру класу `A` передати екземпляр класу `E`, в якому декларовано *частково* загальні можливості, що є загальними для класів `C` та `D` і відсутні в класі `B`.

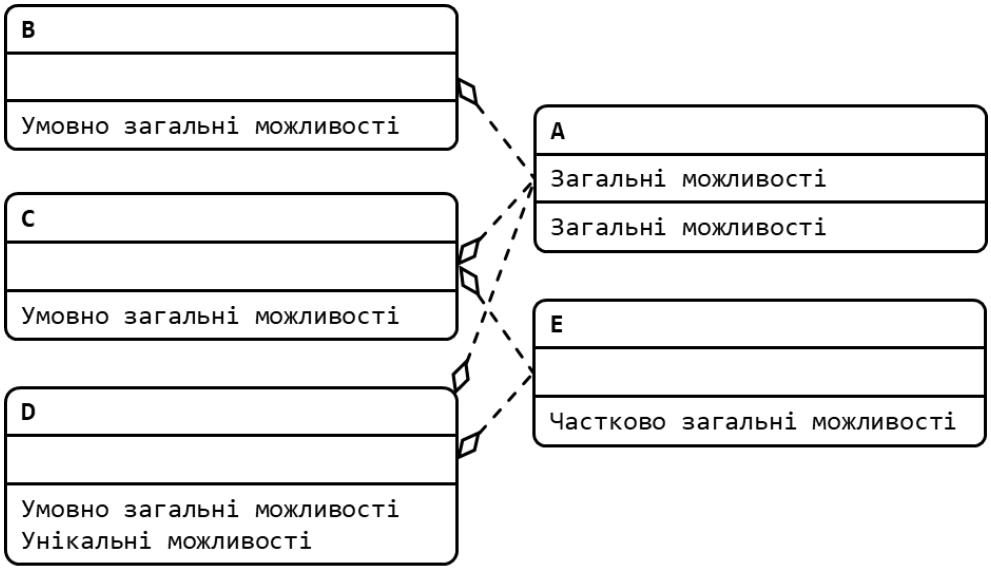


Рис. 2. UML діаграма сутностей під час агрегації

Т.ч. до конструкторів:

- класу `B` впроваджено залежність від класу `A`;
- класів `C` та `D` (в кожному) впроваджено залежності від класів `A` та `E`.

Це дозволяє:

- Не створювати в класі `B` «заглушки» для можливостей, які йому не притаманні (оскільки мають смисл тільки для класів `C` та `D`). Тобто клас `B` не вимушений залежати від інтерфейсу, якого він не використовує.
- Декларувати нові класи, до яких може бути впроваджено залежності від класів `A` та (або) `E`, ніби «набираючи» (компонуючи) потрібні можливості з цих класів.

Щоб додатково послабити сполучення між класами `B` та `A`, `C` та (`A`, `E`), `D` та (`A`, `E`) можна використати інтерфейси:

1. Декларувати інтерфейси для класів `A` та `E`, наприклад, `IA` та `IE`.

2. Змінити заголовки класів `A` та `E` так, щоб вони реалізовували інтерфейси `IA` та `IE`, відповідно.

3. Замінити в конструкторах класів:

3.1. B, C та D тип параметру A на IA.

3.2. C та D тип параметру E на IE.

Використання інтерфейсів під час впровадження залежностей дозволяє прибрати жорстку залежність від конкретних класів. Тепер до параметру(ів) конструкторів класів

- B, C та D можна передавати *будь-який* клас, що реалізує інтерфейс IA.
- C та D можна передавати *будь-який* клас, що реалізує інтерфейс IE.

В цьому проекті дотримано такі принципи SOLID [5]:

- Принцип єдиної відповідальності (Single Responsibility Principle) [4]. Адже, наприклад, до класу A переміщено можливості, що є загальними для класів B, C та D. Т.ч. клас A відповідає тільки за ці загальні можливості і не відповідає за інші можливості (наприклад, декларовані в класі E). Тому клас A має тільки одну причину для змін – якщо потрібно змінити можливості, що є загальними для класів B, C та D.
- Принцип відкритості/закритості (Open/Closed Principle). Припустимо, що потрібно під час створення об'єкта класу (наприклад, B), змінити реалізацію можливостей, які декларовано в класі A (екземпляр A передається до параметра конструктора B). Тоді можна до параметру конструктора класу B замість екземпляра класу A (що реалізує інтерфейс IA) передати екземпляр якогось нового класу (нехай це буде G), що також реалізує інтерфейс IA. Т.ч. функціонал класу B розширено шляхом додавання нового класу G без зміни існуючих класів A та B.
- Принцип розділення інтерфейсів (Interface Segregation Principle). Адже клас A не зобов'язаний реалізовувати інтерфейс IE, який декларує контракт, що визначає можливості, які не мають смислу для класу A. Аналогічно клас E не зобов'язаний реалізовувати інтерфейс IA.
- Принцип інверсії залежності (Dependency Inversion Principle). Адже, наприклад, до параметру типу IA конструктора класу B, не обов'язково передавати екземпляр класу A. Можна передати екземпляр *будь-якого* класу, що реалізує інтерфейс IA.

Відношення «успадкування» можна перетворити на відношення «агрегація». Деякі розробники надають перевагу саме агрегації. Процес реструктуризації існуючого коду (тексту програми) без зміни його зовнішньої поведінки називається *рефакторинг* коду (code refactoring) [6]. Мета рефакторингу коду – покращити структуру та (або) реалізацію програми зберігаючи її функціональність.

ПОСТАНОВКА ЗАДАЧІ

У цих НМР розглядаються реалізації трьох сутностей (`B`, `C` та `D`) та взаємодія з ними. На основі умови задачі за варіантом дод. 2 Здобувач повинен самостійно обрати конкретні найменування сутностей, що реалізуються (див. рядок «Назва»), та їх можливостей. В НМР припускається, що ці сутності реалізуються як класи (мова програмування C#). У кожної з сутностей (`B`, `C` та `D`) відповідно варіанту дод. 2 визначено перелік можливостей (features), які поділено на:

- Загальні – іменування та реалізації можливостей є ідентичними для всіх сутностей.
- Умовно загальні – іменування можливостей є ідентичними для всіх сутностей, а їх реалізації можуть відрізнятись.
- Частково загальні – можливості існують (мають смисл) не для всіх сутностей (тільки для двох з трьох), їх іменування та реалізації – ідентичні.
- Унікальні – можливості існують (мають смисл) не для всіх сутностей (тільки для однієї з трьох).

Для спрощення реалізації можливостей сутностей (дод. 2) їх функціонал полягає у виведенні на екран консольного додатку текстового повідомлення. Це дозволяє імітувати поведінку відповідних можливостей та відображати результат їх виконання. Фрагменти текстових повідомлень в дод. 2, що подано курсивом, позначають місця в текстовому рядку, в яких потрібно забезпечити виведення значень, що відповідають цим фрагментам.

ІЗ складається з чотирьох частин. Здобувачу потрібно розробити консольний додаток мовою програмування C#, в якому:

1. Реалізувати сутності (`B`, `C` та `D`) відповідно умові задачі дод. 2 з використанням відношення «успадкування» (рис. 1).

2. Виконати певні дії (дод. 3) з використанням реалізованих сутностей.

3. Перетворити відношення «успадкування» між сутностями на відношення «агрегація» з використанням впровадження залежностей (dependency injection) до їх конструкторів (рис. 2).

4. Внести необхідні зміни до тексту програми, який реалізує певні дії (дод. 3) з використанням реалізованих сутностей.

Для того, щоб фіксувати основні етапи вирішення ІЗ, Здобувачу потрібно розробити рішення (solution) .NET, яке містить два проекти виду «консольний додаток» мовою програмування C#. У кожному з проектів зберігатиметься текст програми, що відповідає вирішенню задачі:

1. В першому проекті (назву обрати самостійно) – з використанням відношення «успадкування».

2. В другому проекті (назву обрати самостійно) – з використанням відношення «агрегація» шляхом впровадження залежностей (dependency injection) до конструкторів сутностей (дод. 2).

ЗМІСТ ЗВІТУ

1. Титульний аркуш (дод. 1).

2. Номер варіанту, вихідні дані (дод. 2), постановка задачі (див. розділ «Постановка задачі», у т.ч. дод. 3).

3. Перша частина ІЗ (див. розділ «Постановка задачі»):

3.1. Назва цієї частини постановки задачі.

3.2. Діаграма, на якій відображено усі сутності, за допомогою яких забезпечено реалізацію сутностей за варіантом дод. 2 з використанням відношення «успадкування».

3.3. Відповідний текст програми (з коментарями) усіх сутностей, за допомогою яких забезпечено реалізацію задачі за варіантом дод. 2.

4. Друга частина ІЗ (див. розділ «Постановка задачі»):

4.1. Назва цієї частини постановки задачі.

4.2. Скриншот(и) вікна консольного додатку, на якому відображено результат виконання проекту, який реалізує задачу (дод. 3) за умови використання відношення «успадкування».

4.3. Відповідний текст програми (з коментарями), який реалізує задачу (дод. 3) за умови використання відношення «успадкування».

5. Третя частина ІЗ (див. розділ «Постановка задачі»):

5.1. Назва цієї частини постановки задачі.

5.2. Діаграма, на якій відображено усі сутності, за допомогою яких забезпечено реалізацію сутностей за варіантом дод. 2 з використанням відношення «агрегація».

5.3. Відповідний текст програми (з коментарями) усіх сутностей, за допомогою яких забезпечено реалізацію задачі за варіантом дод. 2.

6. Четверта частина ІЗ (див. розділ «Постановка задачі»):

6.1. Назва цієї частини постановки задачі.

6.2. Скриншот(и) вікна консольного додатку, на якому відображено результат виконання проекту, який реалізує задачу (дод. 3) за умови використання відношення «агрегація».

6.3. Відповідний текст програми (з коментарями), який реалізує задачу (дод. 3) за умови використання відношення «агрегація».

7. Обґрунтовані висновки щодо:

7.1. Відповідності можливостей розроблених проектів поставленим задачам.

7.2. Порівняння результатів виконання проектів, розроблених за умови використання відношення «успадкування» та «агрегація», між собою.

ЗАГАЛЬНІ ПОЯСНЕННЯ ДО ПРИКЛАДУ ВИКОНАННЯ ІНДИВІДУАЛЬНОГО ЗАВДАННЯ

У цих НМР як приклад подано порядок вирішення задачі за умови реалізації сутностей з варіанту «Приклад» дод. 2, що є подібними до інших задач в дод. 2. Взаємодія із сутностями, що реалізовано, відповідає задачі в дод. 3. Інтегроване середовище розробки – Visual Studio Community 2022 з англійською локалізацією. Реалізація .NET – .NET 8.0, що відповідає версії мови програмування C# 12.

Якщо Здобувачем обрано реалізацію .NET (і відповідно версію мови C#), що є більш ранньою за C# 12, то до тексту програму може бути потрібно внести зміни. Наприклад, якщо обрано .NET 5.0, то в кожному файлі, де викликається метод `Console.WriteLine()`, на початку файлу потрібно вказувати відповідну директиву `using`, тобто:

```
using System;
```

Про це свідчитиме наявність помилки на етапі компіляції. Для швидкого вирішення помилки:

1. В редакторі коду розмістити курсор на назві класу `Console`.
2. Викликати контекстне меню `Quick actions and Refactorings`… натиснути комбінацію клавіш `Alt` + `Enter`, або `Ctrl` + `.`, або правою кнопкою миші (далі – ПКМ) та в контекстному меню ЛКМ по `Quick actions and Refactorings`….
3. В контекстному меню, натиснути лівою кнопкою миші (далі – ЛКМ) по `using System`.

Інший приклад: під час створення запиту LINQ (п. 4 дод. 3) в .NET 5.0 потрібно доповнити перелік директив `using` таким рядком:

```
using System.Linq;
```

Для зменшення кількості рядків коду в тексті програм у цих НМР прийнято розміщувати фігурну дужку, що відкриває (тобто {), в кінці того ж рядка, в якому розміщено заголовок типу (клас, інтерфейс), його функціонального елемента (метод, властивість) або зіставних операторів (оператори циклу). За промовчанням у VS символ { розміщено на новому (окремому) рядку. Відповідні налаштування автоматичного форматування тексту програми можна змінити: у VS меню `Tools` \ `Options` \ у вікні `Options` розкрити список `Text Editor` \ `C#` \ `Code Style` \ `Formatting` \ `New Lines` і в налаштуваннях відмітити випадки, в яких символ { буде автоматично розміщено на новому рядку або зняти відмітку, для розміщення на тому ж рядку.

ЧАСТИНА № 1
РЕАЛІЗАЦІЯ СУТНОСТЕЙ З ВИКОРИСТАННЯМ ВІДНОШЕННЯ «УСПАДКУВАННЯ»

Приклад виконання

Відповідно до розділу «Постановка задачі» Здобувачу потрібно розробити консольний додаток мовою програмування C#. Для фіксації основних етапів вирішення ІЗ Здобувачу потрібно розробити рішення (solution) .NET, яке містить проект, в якому зберігатиметься текст програми, що відповідає вирішенню задачі з використанням відношення «успадкування».

Створимо рішення для цього ІЗ з назвою `IndividualTask`. Обґрунтування назви – рішення призначене для реалізації задач з ІЗ. Рішення `IndividualTask` міститиме в собі проект консольного додатку для цієї частини задачі ІЗ з назвою `LampInheritance`. Обґрунтування назви – відповідно до варіанту «Приклад» дод. 2 потрібно моделювати різні види ламп. Для створення вказаних рішення та проекту:

1. Створити у Visual Studio (VS) рішення (solution) `IndividualTask` з проектом `LampInheritance` (розташування та назва елементів інтерфейсу користувача VS залежить від версії VS):

1.1. Запустити VS.

1.2. Натиснути ЛКМ по кнопці `Create a new project`.

1.3. З випадних списків обрати мову `C#`, платформу – `Windows`, тип проекту – `Console`.

1.4. ЛКМ по `Console Application` (або `Console App`). Натиснути кнопку `Next` (`OK`, або `Create`).

1.5. В полі `Project Name` ввести `LampInheritance`, а в полі `Solution name` ввести `IndividualTask`.

1.6. Задати розташування (поле `Location`) до місця на локальному комп'ютері папки, в якому зберігатиметься це рішення, натиснути на кнопку `Next`.

1.7. З випадного списку обрати реалізацію .NET, але не раніше за .NET Framework 4.0 (залежить від обраної реалізації .NET та версії VS).

1.8. Якщо VS надає можливість *не* використовувати «оператори верхнього рівня» (“Do not use top-level statements”), то активувати цю можливість (залежить від обраної реалізації .NET та версії VS).

1.9. Натиснути кнопку `Next` (`OK`, або `Create`).

2. Виконати початкові налаштування VS: відобразити вікна (меню `View`) `Solution Explorer`, `Error List`, `Properties Window`.

У першій частині задачі (розділ «Постановка задачі») потрібно реалізувати сутності (`B`, `C` та `D`) відповідно дод. 2 з використанням відношення «успадкування». За вихідними даними варіанту «Приклад» дод. 2 потрібно реалізу-

вати сутності «лампа розжарювання», «лампа світлодіодна», «лампа світлодіодна кольорова». Представимо ці сутності як класи, нехай їх відповідні назви будуть такі: `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`. Обґрунтування вибору назв класів – відповідають версіям назв цих сутностей англійською мовою. Декларуємо ці класи:

1. В редакторі коду відкрити файл `Program.cs`: у вікні `Solution Explorer` подвійний ЛКМ по назві файлу `Program.cs`.

Залежно від реалізації .NET текст програми у файлі може мати такий вигляд:

```
namespace LampInheritance
{
    internal class Program {
        static void Main(string[] args) {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Якщо в тілі методу `Main` містяться якісь рядки коду, то вилучити їх, залишивши тіло методу порожнім. Для того, щоб швидко вилучити рядок коду: розмістити курсор на потрібному рядку, натиснути комбінацію клавіш `Shift` + `Delete`.

2. В тілі простору імен `LampInheritance` (але поза межами класу `Program`) ввести код, після виконання якого буде декларовано класи `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`. Для швидкого створення відповідного коду:

2.1. Доповнити простір імен `LampInheritance` словом `class` і натиснути (можливо, двічі) на клавішу `Tab`, або натиснути комбінацію клавіш `Ctrl` + `Пробіл` і з контекстного меню (рис. 1.1) ЛКМ по `class`. Це доповнить текст програми в редакторі коду певним попередньо підготовленим фрагментом тексту – “code snippet” (економить час на друкування тексту розробником власноруч).

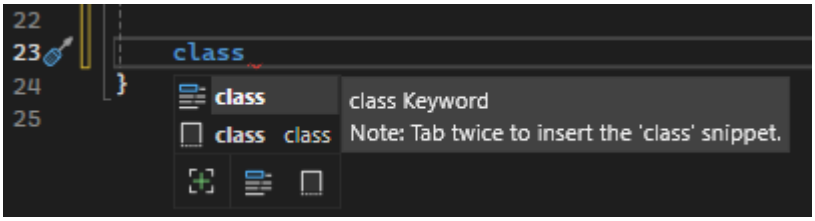


Рис. 1.1. Контекстне меню під час вставлення фрагменту коду для класу

2.2. У доданому коді замінити назву класу на `IncandescentLamp`. Результуючий код:

```
class IncandescentLamp { }
```

2.3. Виконати пункти 2.1 і 2.2 для створення класів `LEDLamp` та `LEDLampColoured`. Результуючий код:

```
class LEDLamp { }
class LEDLampColoured { }
```

3. Перемістити кожен з класів до окремого файлу. Для швидкого виконання:

3.1. В редакторі коду розмістити курсор у заголовку класу `IncandescentLamp` (наприклад, на назві класу).

3.2. Викликати контекстне меню `Quick actions and Refactorings…` (`Alt` + `Enter`).

3.3. В контекстному меню ЛКМ по `Move type to IncandescentLamp.cs` (рис. 1.2).

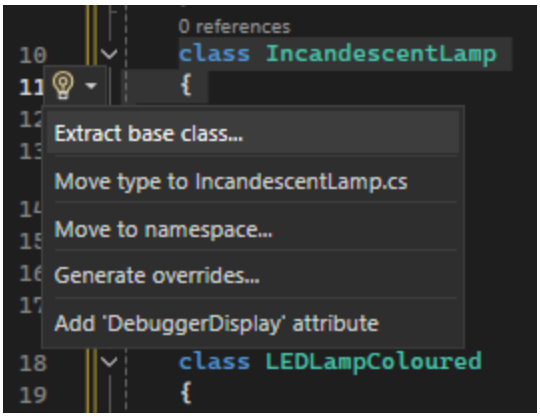


Рис. 1.2. Контекстне меню `Quick actions and Refactorings…` для класу

Після цього у вікні `Solution Explorer` буде створено елемент `IncandescentLamp.cs`, який представляє однойменний файл, що містить код класу `IncandescentLamp`.

3.4. Повторити пункти 3.1, 3.2, 3.3 для класів `LEDLamp` та `LEDLampColoured`. Назви файлів при цьому будуть `LEDLamp.cs` та `LEDLampColoured.cs`, відповідно (рис. 1.3).

Реалізуємо можливості класу `IncandescentLamp`. За вихідними даними варіанту «Приклад» дод. 2 в сутності «лампа розжарювання» потрібно реалізувати можливості одноразово (тільки під час створення об’єкта) записувати значення номінальної потужності (Вт) і багаторазово його читати. Реалізуємо це в класі `IncandescentLamp` як публічну властивість `WattsNominal` типу `double`, яка доступна тільки для читання:

1. В редакторі коду відкрити файл `IncandescentLamp.cs`: у вікні `Solution Explorer` подвійний ЛКМ по файлу `IncandescentLamp.cs`.

Примітки: швидко перейти до файлу, якщо він є відкритим у вкладці редактора коду, можна за допомогою комбінації клавіш `Ctrl` + `Tab`.

2. В редакторі коду в тілі класу `IncandescentLamp` декларувати відповідну властивість. Для швидкого створення відповідного коду:

2.1. Розмістити курсор в тілі класу `IncandescentLamp`.

2.2. В тілі класу надрукувати слово `prop` і натиснути (можливо, двічі) клавішу `Tab` (“code snippet” автоматично реалізуємої властивості, рис. 1.4).

2.3. Якщо в редакторі коду виділено місце для типу даних, то написати `double`.

2.4. Перейти до назви властивості: натиснути клавішу `Tab` (можливо, двічі). Або просто перемістити курсор.

2.5. Надрукувати назву властивості `WattsNominal`.

2.6. Якщо у властивості є методи доступу окрім `get`, то вилучити їх.

Результуючий код:

```
public double WattsNominal { get; }
```

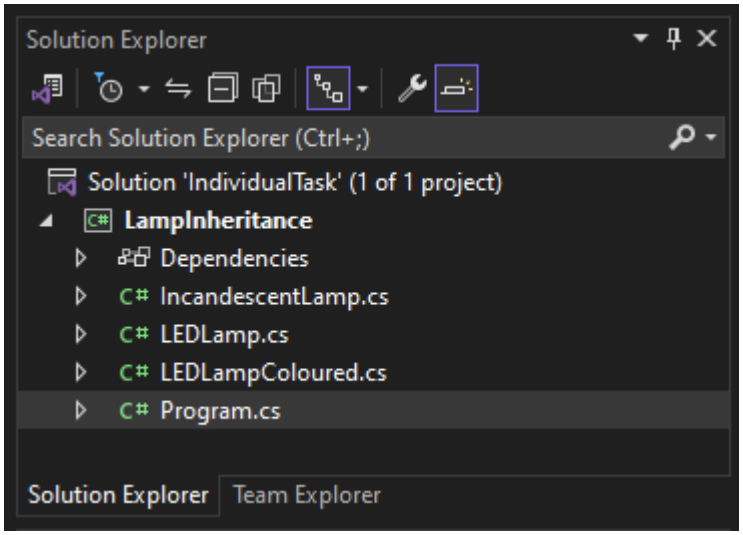


Рис. 1.3. Вікно `Solution Explorer` з рішенням `IndividualTask` та проектом `LampInheritance`

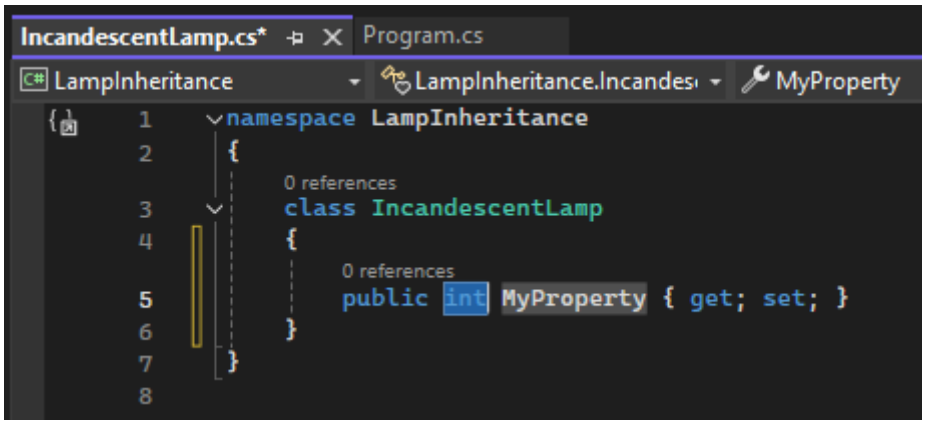


Рис. 1.4. Фрагмент коду автоматично реалізуємої властивості, який вставлено до тіла класу `IncandescentLamp` після застосування “code snippet”для `prop`

Властивість WattsNominal дозволить багаторазово читати значення номінальної потужності, Вт. Тепер реалізуємо можливість одноразово (тільки під час створення об'єкта) записувати значення номінальної потужності, Вт. Для цього декларуємо конструктор в класі IncandescentLamp, який приймає параметр wattsNominal типу double та зберігає його у приватному полі тільки для читання _wattsNominal типу double. Для швидкого створення відповідного коду:

1. В редакторі коду в тілі класу IncandescentLamp надрукувати слово ctor і натиснути (можливо, двічі) клавішу Tab ("code snippet" конструктора без параметрів).

В результаті тіло класу IncandescentLamp доповнено таким кодом:

```
public IncandescentLamp() {
}
```

Доповнити список параметрів цього конструктора параметром wattsNominal типу double. Після цього заголовок конструктора виглядає так

```
public IncandescentLamp(double wattsNominal) {
```

Доповнимо клас IncandescentLamp кодом, який зберігає параметр wattsNominal у приватному полі тільки для читання _wattsNominal типу double:

1. В редакторі коду в тілі класу IncandescentLamp декларувати поле _wattsNominal так

```
double _wattsNominal;
```

2. Доповнити тіло конструктора класу IncandescentLamp кодом

```
_wattsNominal = wattsNominal;
```

Внесемо зміни до властивості WattsNominal, щоб вона повертала значення поля _wattsNominal. Після внесення змін властивість подано таким кодом

```
public double WattsNominal { get => _wattsNominal; }
```

що еквівалентно

```
public double WattsNominal {
    get { return _wattsNominal; }
}
```

За вихідними даними варіанту «Приклад» дод. 2 в сутності «лампа розжарювання» потрібно реалізувати можливість багаторазово записувати та читати значення електричної напруги, В. Реалізуємо це в класі IncandescentLamp як публічну властивість Voltage типу double. Оскільки за умовою задачі жодної додаткової поведінки реалізовувати не потрібно, то властивість Voltage реалізуємо як автоматичну реалізовану властивість (auto-implemented property). Для швидкого створення відповідного коду:

1. В редакторі коду в тілі класу IncandescentLamp надрукувати слово prop і натиснути (можливо двічі) клавішу Tab (“code snippet” автоматично реалізуємої властивості).

2. Надрукувати тип даних double, назву властивості – Voltage.

В результаті тіло класу IncandescentLamp доповнено таким кодом:

```
public double Voltage { get; set; }
```

За вихідними даними варіанту «Приклад» дод. 2 в сутності «лампа розжарювання» потрібно реалізувати можливість перетворювати цей об’єкт на рядок, що має певну структуру. Для цього в класі IncandescentLamp перевизначимо метод ToString(), який декларовано в класі Object як віртуальний. Метод ToString() доступний для перевизначення в усіх класах, оскільки в .NET усі класи походять від класу Object. Доповнимо тіло класу IncandescentLamp кодом:

```
public override string ToString() {
    return $"{this.GetType()} : ..."
        + Environment.NewLine
        + $"номінальна потужність, Вт : {WattsNominal}"
        + Environment.NewLine
        + $"електрична напруга, В : {Voltage}";
}
```

Якщо в редакторі коду вказано повідомлення про помилку «назва ‘Environment’ не існує в поточному контексті», то для швидкого виправлення:

1. В редакторі коду розмістити курсор на Environment.

2. Викликати контекстне меню Quick actions and Refactorings… (Alt + Enter).

3. В контекстному меню ЛКМ по “using System;”. Після цього поточний файл буде доповнено відповідною директивою using.

Відповідно до вихідних даних варіанту «Приклад» дод. 2 в сутності «лампа розжарювання» потрібно реалізувати можливість перетворювати напругу та потужність на світло. Реалізуємо це в класі IncandescentLamp як метод з назвою Light, що не приймає параметрів і має повертаєме значення void: доповнити тіло класу IncandescentLamp таким кодом

```
public void Light() {
    Console.WriteLine($"{this.GetType()} " +
        $"перетворює напругу {Voltage} (В) " +
        $"на світло з {WattsNominal} (Вт)... Виконано.");
}
```

Примітки: для швидкого введення Console.WriteLine() в редакторі коду достатньо надрукувати слово cw і натиснути (можливо двічі) клавішу Tab (відповідний “code snippet”).

Т.ч. всі можливості класу IncandescentLamp реалізовано. Тепер реалізуємо можливості класу LEDLamp. За вихідними даними варіанту «Приклад»

дод. 2 в сутності «лампа світлодіодна» певні можливості є *загальними* (колонка «Вид можливості» дод. 2), тобто ідентичними за назвою та реалізацією можливостями сутності «лампа розжарювання»:

- одноразово (тільки під час створення об’єкта) записувати значення номінальної потужності, Вт, і багаторазово читати це (ці) значення;
- багаторазово записувати та читати значення електричної напруги, В.
- перетворювати цей об’єкт на рядок.

Тому достатньо виділити певні елементи класу `IncandescentLamp` (поле `_wattsNominal`, властивість `WattsNominal`, властивість `Voltage`, конструктор `IncandescentLamp`, метод `ToString`), скопіювати їх, відкрити в редакторі коду файл, що містить клас `LEDLamp` та вставити скопійовані фрагменти коду до тіла класу `LEDLamp`. Змінити назву конструктора на `LEDLamp`. Після цього код класу `LEDLamp`:

```
class LEDLamp {
    double _wattsNominal;
    public double WattsNominal { get => _wattsNominal; }
    public LEDLamp(double wattsNominal) {
        _wattsNominal = wattsNominal;
    }
    public double Voltage { get; set; }
    public override string ToString() {
        return $"{this.GetType()} : ..."
            + Environment.NewLine
            + $"номінальна потужність, Вт : {WattsNominal}"
            + Environment.NewLine
            + $"електрична напруга, В : {Voltage}";
    }
}
```

Якщо в редакторі коду вказано повідомлення про помилку «назва ‘`Environment`’ не існує в поточному контексті», то див. рішення вище (для класу `IncandescentLamp`).

За вихідними даними варіанту «Приклад» дод. 2 в сутності «лампа світлодіодна» є можливість перетворювати напругу та потужність на світло, яка має назву, що ідентична можливості сутності «лампа розжарювання», але для її реалізації потрібно спочатку *випрямити* напругу. Це *умовно* загальна можливість (колонка «Вид можливості» дод. 2). Реалізуємо в класі `LEDLamp` можливість випрямляти напругу як метод з назвою `RectifyVoltage`, що не приймає параметрів і має повертаємо значення `void`: доповнити тіло класу `LEDLamp` таким кодом

```
public void RectifyVoltage() {
    Console.WriteLine($"{this.GetType()} випрямляє " +
        $"напругу {Voltage} (В)... Виконано.");
}
```

Тепер реалізуємо в класі LEDLamp можливість перетворювати напругу та потужність на світло. Для цього:

1. Скопіювати метод Light() з класу IncandescentLamp та вставити його до тіла класу LEDLamp.

2. Внести зміни до тіла методу Light() в класі LEDLamp: перед виведенням на екран консольного вікна викликати метод RectifyVoltage(). В результаті метод Light() в класі LEDLamp має такий код:

```
public void Light() {
    RectifyVoltage();
    Console.WriteLine($"{this.GetType()} " +
        $"перетворює напругу {Voltage} (В) " +
        $"на світло з {WattsNominal} (Вт)... Виконано.");
}
```

З аналізу елементів класів IncandescentLamp та LEDLamp видно, що в них декларовано *загальні* можливості (номінальна потужність, електрична напруга, перетворення об'єкту на рядок), які є ідентичними. Наявність такого "copy-paste" коду, як правило, свідчить про недолік у проектуванні сутностей (т.з. "code smell") і порушення принципу DRY. Для виправлення доцільно перемістити код, який повторюється у цих класах, до нової сутності та пов'язати її з існуючими класами.

Відповідно до розділу «Постановка задачі» в першій частині потрібно реалізувати сутності з використанням відношення «успадкування». Тому декларуємо новий клас, який буде базовим для існуючих класів IncandescentLamp та LEDLamp. Оскільки за вихідними даними варіанту «Приклад» дод. 2 потрібно реалізувати сутності, які представляють різні види ламп, то декларуємо базовий клас з назвою Lamp, від якого будуть походити існуючі класи IncandescentLamp та LEDLamp. Для швидкого створення відповідного коду:

1. В редакторі коду відкрити файл IncandescentLamp.cs.

2. В редакторі коду розмістити курсор у заголовку класу IncandescentLamp.

3. Викликати контекстне меню Quick actions and Refactorings… (Alt + Enter).

4. В контекстному меню ЛКМ по Extract base class… (рис. 1.2).

5. У вікні Extract Base Class (рис. 1.5):

5.1. В полі New Type Name ввести Lamp.

5.2. Відмітити "New file name" і у текстовому полі поруч ввести Lamp.cs.

5.3. В списку Select members встановити позначки ліворуч усіх елементів.

5.4. В колонці Make abstract встановити позначку біля методу Light().

5.5. Натиснути кнопку OK.

Після цього у вікні Solution Explorer буде створено елемент Lamp.cs, який представляє однойменний файл, що містить код класу Lamp. Метод

`Light()` у базовому класі `Lamp` буде декларовано як *абстрактний* (без реалізації), оскільки в `IncandescentLamp` та `LEDLamp` декларовано метод з ідентичною назвою але з *різними* реалізаціями. Це дозволить викликати метод `Light()` навіть з екземпляра базового класу `Lamp`.

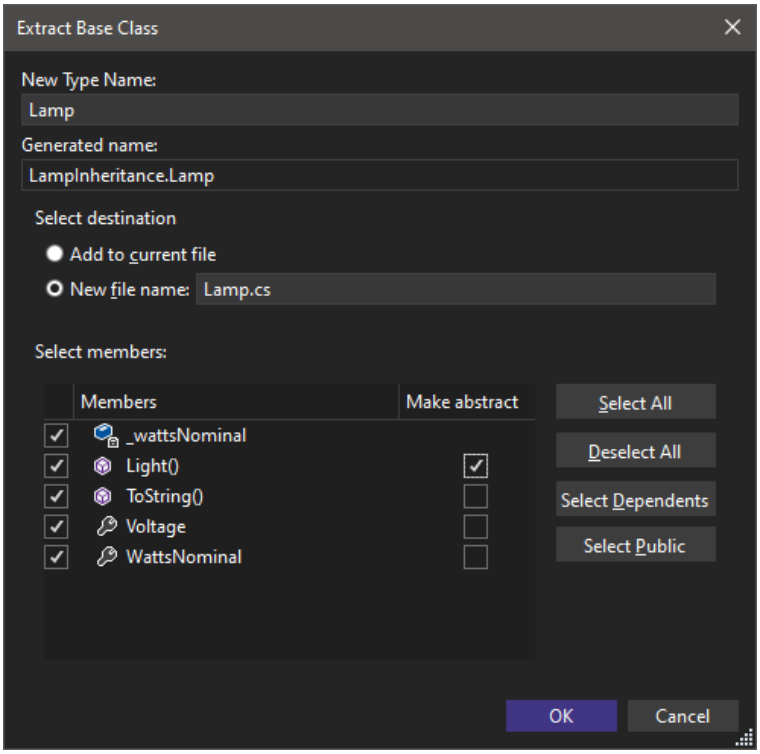


Рис. 1.5. Вікно `Extract Base Class`, викликане для класу `IncandescentLamp`

В проекті буде створено новий файл, який містить абстрактний клас `Lamp` з таким кодом:

```
internal abstract class Lamp {
    double _wattsNominal;
    public double Voltage { get; set; }
    public double WattsNominal { get => _wattsNominal; }
    public abstract void Light();
    public override string ToString() {
        return $"{this.GetType()} : ..."
            + Environment.NewLine
            + $"номінальна потужність, Вт : {WattsNominal}"
            + Environment.NewLine
            + $"електрична напруга, В : {Voltage}";
    }
}
```

Т.ч. в класі `Lamp` декларовано реалізацію усіх *загальних* можливостей та декларовано абстрактний метод, що відноситься до *умовно* загальних можливостей.

Клас `IncandescentLamp` матиме такий вигляд:

```
class IncandescentLamp : Lamp {
    public IncandescentLamp(double wattsNominal) {
        _wattsNominal = wattsNominal;
    }

    public override void Light() {
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Voltage} (В) " +
            $"на світло з {WattsNominal} (Вт)... Виконано.");
    }
}
```

В заголовку класу `IncandescentLamp` автоматично встановлено відношення «успадкування» від базового класу `Lamp`.

Після переміщення поля `_wattsNominal` з класу `IncandescentLamp` до базового класу, в конструкторі `IncandescentLamp` з'явилась помилка на етапі компіляції. Для її виправлення потрібно забезпечити присвоєння значення параметру `wattsNominal` цьому полю. Для цього:

1. В класі `Lamp` декларувати конструктор

```
protected Lamp(double wattsNominal) {
    _wattsNominal = wattsNominal;
}
```

2. В класі `IncandescentLamp` змінити конструктор на такий:

```
public IncandescentLamp(double wattsNominal) : base(wattsNominal) { }
```

Це дозволить передавати значення параметру `wattsNominal` полю `_wattsNominal`, через виклик конструктора базового класу `Lamp`.

Внести аналогічні зміни до класу `LEDLamp`:

1. Встановити клас `LEDLamp` похідним від класу `Lamp` – змінити його заголовок на такий

```
class LEDLamp : Lamp {
```

2. Вилучити елементи класу `LEDLamp`, які ідентичні тим, що переміщено до базового класу `Lamp`.

3. Замінити конструктор на такий

```
public LEDLamp(double wattsNominal) : base(wattsNominal) { }
```

4. Доповнити заголовок методу `Light()` модифікатором `override`.

Після цього код класу `Lamp` матиме такий вигляд:

```
class LEDLamp : Lamp {
    public LEDLamp(double wattsNominal) : base(wattsNominal) { }
    public void RectifyVoltage() {
        Console.WriteLine($"{this.GetType()} випрямляє " +
            $"напругу {Voltage} (В)... Виконано.");
    }
    public override void Light() {
        RectifyVoltage();
```

```
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Voltage} (В) " +
            $"на світло з {WattsNominal} (Вт)... Виконано.");
    }
}
```

Тепер реалізуємо можливості класу `LEDLampColoured`. За вихідними даними варіанту «Приклад» дод. 2 в сутності «лампа світлодіодна кольорова» певні можливості є *загальними* (колонка «Вид можливості» дод. 2), тобто ідентичними за назвою та реалізацією можливостями сутностей «лампа розжарювання» та «лампа світлодіодна», які переміщено до базового класу – `Lamp`. За аналогією з класами `IncandescentLamp` та `LEDLamp` змінимо декларацію класу `LEDLampColoured` так, щоб він походив від `Lamp`:

```
class LEDLampColoured : Lamp {
```

Це дозволяє отримати доступ до можливостей, що є загальними для класів `IncandescentLamp`, `LEDLamp` і `LEDLampColoured` та уникнути непотрібного дублювання коду. Тобто дотримати принцип DRY.

Після встановлення відношення «успадкування» між похідним класом `LEDLampColoured` та базовим `Lamp`, вікно `Error List` (та редактор коду) VS сповіщає про помилку етапу компіляції: в класі `LEDLampColoured` не реалізовано абстрактний метод `Light()`. Для швидкого усунення помилки:

1. В редакторі коду розмістити курсор на назві класу `LEDLampColoured`.
2. Викликати контекстне меню `Quick actions and Refactorings…` (`Alt` + `Enter`).
3. В контекстному меню ЛКМ по “`Implement abstract class`”.

Після цього тіло класу `LEDLampColoured` доповнено перевизначенням методу `Light()`, яке є «заглушкою» тому, що в його тілі викликається виняток:

```
throw new System.NotImplementedException();
```

Це дозволяє розробнику під час налагоджування програми:

- швидко декларувати методи без витрати часу на реалізацію їх поведінки
- і отримувати виняток `System.NotImplementedException()`, якщо цей метод все ж було викликано але так і не було доповнено відповідною поведінкою.

Залишилась ще одна помилка етапу компіляції: відсутній аргумент, який відповідає потрібному формальному параметру `wattsNominal` конструктора `Lamp`. Дійсно, у базовому класі `Lamp` декларовано конструктор із заголовком

```
protected Lamp(double wattsNominal)
```

Але в тілі класу `LEDLampColoured` цей конструктор не викликається. Для виправлення: доповнити тіло класу `LEDLampColoured` конструктором, що є аналогічним конструкторам класів `IncandescentLamp` та `LEDLamp`:

```
public LEDLampColoured(double wattsNominal) : base(wattsNominal) { }
```

За вихідними даними варіанту «Приклад» дод. 2 в сутності «лампа світлодіодна кольорова» є *частково* загальна можливість випрямляти напругу, яка є ідентичною за назвою та реалізацією можливості сутності «лампа світлодіодна». Тому: скопіювати та вставити метод `RectifyVoltage()` з класу `LEDLamp` до тіла класу `LEDLampColoured`.

Примітки: для швидкого виділення блоку коду:

1. Розмістити курсор в тексті потрібного блоку – тіло методу `RectifyVoltage()`.

2. Натиснути комбінацію клавіш `Alt` + `Shift` + `=` стільки разів, скільки потрібно для збільшення обсягу блоку, який виділено, до потрібного (заголовок класу разом з тілом класу). Якщо потрібно зменшити обсяг виділеного коду, то натиснути комбінацію клавіш `Alt` + `Shift` + `–`.

За вихідними даними варіанту «Приклад» дод. 2 в сутності «лампа світлодіодна кольорова» є *умовно* загальна можливість перетворювати напругу та потужність на світло. Її назва ідентична можливості сутності «лампа світлодіодна», але для її реалізації потрібно не тільки випрямити напругу, а й *зафарбувати* світло. Для цього реалізуємо можливість зафарбовувати світло як метод з назвою `ColourTheLight`, що не приймає параметрів і має повертаєме значення `void`:

```
public void ColourTheLight() {
    Console.WriteLine($"{this.GetType()} зафарбовує " +
        $"світло... Виконано.");
}
```

Замінимо в класі `LEDLampColoured` метод-«заглушку» `Light()` кодом

```
public override void Light() {
    RectifyVoltage();
    ColourTheLight();
    Console.WriteLine($"{this.GetType()} " +
        $"перетворює напругу {Voltage} (В) " +
        $"на світло з {WattsNominal} (Вт)... Виконано.");
}
```

Тіло методу `Light()` класу `LEDLampColoured` відрізняється від тіла методу `Light()` класу `LEDLamp` тільки викликом методу `ColourTheLight()` після виклику методу `RectifyVoltage()`.

Т.ч. створення коду для вирішення першої частини задачі завершено. Остаточний текст програми подано в дод. 4. Якщо читати текст програми (дод. 4) згори донизу, то код ніби «розповідає історію». Наприклад, у методі `Light()` класу `LEDLampColoured` послідовно викликаються методи `RectifyVoltage()` та `ColourTheLight()`. Після тіла методу `Light()` спочатку реалізовано метод `RectifyVoltage()`, потім – `ColourTheLight()`.

Для того, щоб виявити помилки на етапі компіляції доцільно скомпілювати проект: натиснути на комбінацію клавіш `Ctrl` + `B`. У разі наявності помилок на етапі компіляції – внести відповідні виправлення.

З аналізу елементів класів LEDLamp та LEDLampColoured видно, що можливість випрямляти напругу (метод RectifyVoltage()) є *частково* загальною, тобто *ідентичною* за назвою та реалізацією у цих двох класах і відсутньою в класі IncandescentLamp. Це порушує принцип DRY. Для виправлення доцільно перемістити код, який повторюється у цих класах, до нової сутності та пов'язати її з існуючими класами.

Відповідно до розділу «Постановка задачі» в першій частині задачі потрібно реалізувати сутності з використанням відношення «успадкування». Якщо перемістити можливість випрямляти напругу до базового класу Lamp, то це матиме смисл для похідних від нього класів LEDLamp та LEDLampColoured, але *не* матиме смислу для класу IncandescentLamp. Адже за вихідних даних варіанту «Приклад» дод. 2 цієї можливості немає в класу IncandescentLamp. У цьому випадку в класі IncandescentLamp довелось би створювати метод-«заглушку», що не завжди є найкращою практикою. Існують різні способи включити до тіла класу елементи, які належать різним єрархіям класів. Серед них – використання відношення «агрегація». Цьому присвячено третю частину розділу «Постановка задачі».

Для того, щоб засобами VS створити діаграму класів (п. 3.2 розділу «Зміст звіту») Lamp, IncandescentLamp, LEDLamp та LEDLampColoured:

1. Додати елемент Class Diagram до проекту LampInheritance:

1.1. У вікні Solution Explorer натиснути ПКМ по назві проекту.

1.2. В контекстному меню ЛКМ по Add \ New Item \ Class diagram, в полі Name ввести назву файлу ClassDiagram1.cd, натиснути на кнопку Add.

2. Розмістити класи для створення діаграми у вікні файлу ClassDiagram1.cd. Для цього у вікні Solution Explorer:

2.1. Відкрити файл ClassDiagram1.cd – подвійний ЛКМ по цьому елементу.

2.2. Затиснути ЛКМ на файлі Lamp.cs, перетягнути його з вікна Solution Explorer до вікна ClassDiagram1.cd та відпустити ЛКМ.

2.3. Повторити п. 2.2 для файлів IncandescentLamp.cs, LEDLamp.cs та LEDLampColoured.cs.

Результуючу діаграму класів, створену засобами Class designer, подано в дод. 8.

Якщо в переліку шаблонів відсутній Class diagram, то потрібно інсталювати компонент Class designer:

1. Зберегти поточний проект, закрити VS.

2. Запустити на виконання інсталяційний файл VS (Visual Studio Installer). На вкладці Installed, натиснути кнопку Modify.

3. Перейти на вкладку Individual Components, у полі пошуку ввести Class Designer.

4. В списку Code tools відмітити Class Designer, натиснути на кнопку Modify.

5. Дочекатись завершення оновлення VS, закрити вікно Visual Studio Installer.

ЧАСТИНА № 2
ВЗАЄМОДІЯ З СУТНОСТЯМИ, ЯКІ РЕАЛІЗОВАНО З ВИКОРИСТАННЯМ ВІДНОШЕННЯ «УСПАДКУВАННЯ»

Приклад виконання

Відповідно до другої частини задачі (розділ «Постановка задачі») потрібно виконати певні дії (дод. 3) з використанням сутностей, що реалізовано в першій частині. Відповідно дод. 3 потрібно декларувати колекцію `X` загальною кількістю не менше п'яти елементів. Причому, колекція `X` повинна дозволяти збереження екземплярів всіх класів, що реалізовано відповідно дод. 2 (`IncandescentLamp`, `LEDLamp` та `LEDLampColoured`).

Колекцію `X` призначено для збереження екземплярів різних ламп, тому нехай вона має назву `lamps`. Оскільки за умовою задачі жодних вимог для доступу до елементів колекції чи їх збереження немає, то оберемо колекцію, яка дозволяє довільний доступ до елементів. Серед таких колекцій є масив (`Array`) та список (`ArrayList` і `List<T>`). Для того, щоб колекція `lamps` могла зберігати екземпляри класів `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`, в якості типу елементів колекції визначимо базовий клас `Lamp`. Тому `ArrayList` є не найкращим кандидатом, отже потрібно обрати між `Array` та `List<T>`. В умові задачі нічого не вказано стосовно можливості зміни кількості елементів в колекції. Припустимо, що вона може змінюватись, тому оберемо загальну (generic) колекцію «список», тобто `List<T>`, де `T` – тип-параметр. В якості типу-параметра списку `lamps` визначимо базовий клас `Lamp`. Декларуємо в тілі методу `Main` класу `Program` змінну `lamps`:

```
List<Lamp> lamps;
```

Якщо в редакторі коду вказано повідомлення про помилку «назва '`List<>`' не існує в поточному контексті», то для швидкого виправлення:

1. В редакторі коду розмістити курсор на `List<Lamp>`.
2. Викликати контекстне меню `Quick actions and Refactorings…` (`Alt` + `Enter`).
3. В контекстному меню ЛКМ по "`using System.Collections.Generic;`". Після цього поточний файл буде доповнено відповідною директивою `using`.

Відповідно п. 1 дод. 3 до елементів колекції `lamps` потрібно записати екземпляри класів, що реалізовано відповідно дод. 2 (`IncandescentLamp`, `LEDLamp` та `LEDLampColoured`), не менше ніж по одному екземпляру кожного класу. Ініціювати стан цих екземплярів (їх поля, властивості) довільними значеннями. Замінимо декларацію `lamps` в тілі методу `Main` класу `Program`, наприклад, таким кодом:

```
List<Lamp> lamps = new List<Lamp> {
    new IncandescentLamp(100) { Voltage = 1 },
```

```
    new LEDLamp(8) { Voltage = 2 },
    new LEDLampColoured(7) { Voltage = 2 },
    new LEDLamp(12) { Voltage = 5 },
    new LEDLamp(7.5) { Voltage = 5 }
};
```

Відповідно дод. 3 для кожного елементу колекції `lamps` потрібно викликати можливість, що в дод. 2 віднесено до *умовно* загальних. Такою можливістю є перетворювати напругу та потужність на світло (реалізовано методом `Light()`). Оскільки потрібно ітерувати (переглянути) кожен елемент колекції і при цьому їх не потрібно змінювати, то оберемо цикл `foreach`. Для швидкого створення відповідного коду в тілі методу `Main` класу `Program`:

1. Доповнити метод `Main` класу `Program` надрукувавши слово `foreach` і натиснути (можливо двічі) клавішу `Tab`. Відповідний код (“code snippet”):

```
foreach (var item in collection) {
}
```

2. Якщо виділено слово `var`, то залишити його без змін і натиснути клавішу `Tab`.

3. Тепер виділено слово `item`. Залишити його без змін і натиснути клавішу `Tab`.

4. Тепер виділено слово `collection`. Замінити його на `lamps`.

5. Доповнити тіло циклу `foreach` кодом, що для кожного `item` викликає метод `Light()`. Остаточний вигляд циклу `foreach`:

```
foreach (var item in lamps) {
    item.Light();
    Console.WriteLine();
}
```

Відповідно дод. 3 потрібно створити запит LINQ, в якому сортувати за *спаданням* елементи колекції `lamps` за одним з параметрів (обрати самостійно), що в дод. 2 віднесено до загальних можливостей, які можна багаторазово записувати та читати. Для варіанту «Приклад» дод. 2 існує тільки один такий параметр, який доступний через властивість `Voltage`. У тому ж запиті LINQ потрібно сортувати за зростанням за одним з параметрів (обрати самостійно), що в дод. 2 віднесено до загальних можливостей, які можна одноразово записувати та багаторазово читати. Для варіанту «Приклад» дод. 2 існує тільки один такий параметр, який доступний через властивість `WattsNominal`. Доповнимо тіло методу `Main` класу `Program` декларацією змінної запиту з назвою `query`:

```
var query = lamps
    .OrderByDescending(x => x.Voltage)
    .ThenBy(x => x.WattsNominal);
```

Якщо в редакторі коду вказано повідомлення про помилку «назва ‘List<Lamp>’ не містить визначення ‘OrderByDescending’», то доповнити перелік директив using таким рядком:

```
using System.Linq;
```

Відповідно дод. 3 потрібно вивести на екран консольного додатку рядкове подання елементів результату виконання створеного запиту LINQ. Оскільки за умовою задачі відсутня вимога до збереження результатів запиту в якійсь структурі даних, то достатньо його ітерувати (переглянути). Для цього використаємо цикл foreach. Доповнимо тіло методу Main в класі Program відповідним кодом:

```
Console.WriteLine("Результат виконання запиту:");
foreach (var item in query) {
    Console.WriteLine(item.ToString());
    Console.WriteLine();
}
```

Т.ч. створення коду для вирішення другої частини задачі завершено. Остаточний текст програми подано в дод. 5.

Для отримання скриншоту(ів) вікна консольного додатку, на якому відображено результат виконання проекту LampInheritance (п. 4.2 розділу «Зміст звіту»), потрібно скомпілювати цей проект та запустити на виконання (доцільно – в режимі налагодження): меню Debug\ Start Debugging.

Якщо у вікні консольного додатку не всі літери українського алфавіту відображено коректно (наприклад, замість літери «і» відображено символ «?»), то доповнити метод Main класу Program перед першим виведенням на екран консольного вікна таким рядком коду

```
Console.OutputEncoding = Encoding.Unicode;
```

Якщо в редакторі коду вказано повідомлення про помилку «назва ‘Encoding’ не існує в поточному контексті», то для швидкого виправлення:

1. В редакторі коду розмістити курсор на Encoding.
2. Викликати контекстне меню Quick actions and Refactorings… (Alt + Enter).
3. В контекстному меню ЛКМ по using System.Text;. Після цього поточний файл буде доповнено відповідною директивою using.

ЧАСТИНА № 3
ПЕРЕТВОРЕННЯ ВІДНОШЕННЯ «УСПАДКУВАННЯ» НА ВІДНОШЕННЯ «АГРЕГАЦІЯ»

Приклад виконання

Відповідно до розділу «Постановка задачі» для фіксації основних етапів вирішення ІЗ Здобувачу потрібно в існуючому рішенні `IndividualTask` створити проект (консольний додаток), в якому зберігатиметься текст програми, що відповідає вирішенню задачі з використанням відношення «агрегація» шляхом впровадження залежностей (dependency injection) до конструкторів сутностей. Створимо в рішенні `IndividualTask` новий проект консольного додатку для цієї частини задачі ІЗ з назвою `LampAggregation`. Обґрунтування вибору назви – потрібно моделювати різні види ламп за допомогою відношення «агрегація».

Для створення проекту `LampAggregation` у рішенні `IndividualTask`:

1. Відкрити рішення `IndividualTask` з проектом `LampInheritance`.

2. У вікні `Solution Explorer` натиснути ПКМ по назві рішення (`IndividualTask`), `Add\ New Project…`, `Console Application` (або `Console App`). Натиснути кнопку `Next` (`OK`, `Create` тощо).

3. В полі `Project Name` ввести `LampAggregation`. Якщо VS надає можливість *не* використовувати «оператори верхнього рівня» («do not use top-level statements»), то активувати цю можливість (залежить від обраної реалізації .NET та версії VS). Натиснути кнопку `Next` (`OK`, `Create` тощо).

Відповідно до третьої частини задачі (розділ «Постановка задачі») потрібно перетворити відношення «успадкування» між класами `Lamp`, `IncandescentLamp`, `LEDLamp` та `LEDLampColoured` на відношення «агрегація» з використанням впровадження залежностей (dependency injection) до їх конструкторів (виконати “refactoring”). Скопіюємо існуючі файли з класами з проекту `LampInheritance` до проекту `LampAggregation`:

1. У вікні `Solution Explorer` серед елементів проекту `LampInheritance` затиснути ЛКМ по файлу `Lamp.cs`, перетягнути вказівник миші на назву проекту `LampAggregation`, відпустити ЛКМ.

2. Повторити п. 1 для файлів `IncandescentLamp.cs`, `LEDLamp.cs` та `LEDLampColoured.cs`.

3. В редакторі коду закрити всі відкриті вкладки з кодом (для зручності). У вікні `Solution Explorer` в проекті `LampAggregation` відкрити файли `Lamp.cs`, `IncandescentLamp.cs`, `LEDLamp.cs` та `LEDLampColoured.cs`.

4. Замінити назву простору імен в кожному файлі (п. 3) на назву простору імен проекту `LampAggregation`, тобто – `LampAggregation`.

Для зручності у вікні `Solution Explorer` доцільно згорнути проект `LampInheritance`.

Вилучимо відношення «успадкування» в існуючій єрархії класів – змінимо заголовки класів IncandescentLamp, LEDLamp та LEDLampColoured на такі:

```
class IncandescentLamp {
class LEDLamp {
class LEDLampColoured {
```

Тепер єрархії з базового класу Lamp та похідних від нього класів IncandescentLamp, LEDLamp та LEDLampColoured не існує. Всі ці класи неявно походять від класу Object. Тому в редакторі коду відображено помилки етапу компіляції. Для їх виправлення:

1. Видалити з класів IncandescentLamp, LEDLamp та LEDLampColoured їх конструктори (ті, що приймали в списку параметрів wattsNominal).

2. Видалити із заголовків методів Light() модифікатор override. Оскільки немає базового класу, що містив абстрактний метод Light(), який можна перевизначити.

3. Замінити в заголовку конструктора класу Lamp модифікатор protected на public.

Виконаємо перетворення тексту програми без зміни функціональності, тобто – “refactoring”. Мета: переробити класи IncandescentLamp, LEDLamp та LEDLampColoured так, щоб вони в параметрі своїх конструкторів отримували екземпляр класу Lamp, замість того, щоб походити від нього, як це було під час використання відношення «успадкування».

Щоб мати можливість створювати екземпляр класу Lamp, видалимо з його заголовку модифікатор abstract. Оскільки після цього в класі не може бути абстрактних елементів, то видалимо абстрактний метод Light(). Остаточний код класу Lamp:

```
internal class Lamp {
    double _wattsNominal;
    public double Voltage { get; set; }
    protected Lamp(double wattsNominal) {
        _wattsNominal = wattsNominal;
    }
    public double WattsNominal { get => _wattsNominal; }
    public override string ToString() {
        return $"{this.GetType()} : ..."
            + Environment.NewLine
            + $"номінальна потужність, Вт : {WattsNominal}"
            + Environment.NewLine
            + $"електрична напруга, В : {Voltage}";
    }
}
```

В класах LEDLamp та LEDLampColoured залишилось дублювання реалізації можливості випрямляти напругу (метод RectifyVoltage()). Вона є *частково* загальною для класів LEDLamp та LEDLampColoured і відсутня в класі IncandescentLamp. Щоб дотримати принцип DRY, доцільно перемістити

код, який повторюється у цих класах, до нової сутності та пов'язати цю сутність з існуючими класами. Безпосередньо в єрархії, яку було створено в першій частині задачі, це було проблематично. Проте під час використання відношення «агрегація» це реалізувати значно легше. Для цього декларуємо новий клас `VoltageRectifier` (з англ. – випрямляч напруги) та перемістимо до нього метод `RectifyVoltage()` з класів `LEDLamp` та `LEDLampColoured`:

1. В редакторі коду відкрити файл `LEDLamp.cs`.

2. В тілі простору імен декларувати клас `VoltageRectifier`.

3. *Перемістити* метод `RectifyVoltage()` з тіла класу `LEDLamp` до тіла класу `VoltageRectifier`. Для швидкого переміщення цього методу:

3.1. Виділити метод `RectifyVoltage()` (заголовок та тіло методу).

3.2. Натиснути комбінацію клавіш `Alt` + ↓ (або `Alt` + ↑), щоб перемістити фрагмент коду на один рядок донизу (чи догори). Виконати це стільки разів, скільки потрібно для розміщення методу `RectifyVoltage()` в тілі класу `VoltageRectifier`.

4. Перемістити клас `VoltageRectifier` до нового файлу. Для швидкого виконання:

4.1. В редакторі коду розмістити курсор в заголовку класу `VoltageRectifier` (наприклад, на назві класу).

4.2. Викликати контекстне меню `Quick actions and Refactorings…` (`Alt` + `Enter`).

4.3. В контекстному меню ЛКМ по `Move type to VoltageRectifier.cs`.

Остаточний код класу `VoltageRectifier`:

```
class VoltageRectifier {
    public void RectifyVoltage() {
        Console.WriteLine($"{this.GetType()} випрямляє " +
            $"напругу {Voltage} (В)... Виконано.");
    }
}
```

5. В редакторі коду відкрити файл `LEDLampColoured.cs`.

6. Видалити з класу `LEDLampColoured` метод `RectifyVoltage()`. Для швидкого видалення рядка коду: розмістити курсор на потрібному рядку, натиснути комбінацію клавіш `Shift` + `Delete`.

Т.ч. в класах `LEDLamp` та `LEDLampColoured` вилучено дублювання коду – в них відсутні методи `RectifyVoltage()`. За забезпечення частково загальної можливості (випрямляти напругу) тепер відповідає клас `VoltageRectifier`.

Після виконання попередніх дій в проекті `LampAggregation` існують такі класи:

- `IncandescentLamp`, `LEDLamp` та `LEDLampColoured` – містять *умовно* загальні можливості, які мають *різну* реалізацію залежно від класу.
- `Lamp` – містить *загальні* можливості для класів `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`.

– `VoltageRectifier` – містить *частково* загальні можливості – тільки для двох класів – `LEDLamp` та `LEDLampColoured`.

Впровадимо до класів `IncandescentLamp`, `LEDLamp` та `LEDLampColoured` залежності (через параметри їх конструкторів) так, щоб вони залежали від класів `Lamp` та, у разі потреби, від `VoltageRectifier`.

За вихідними даними варіанту «Приклад» дод. 2 в класі `IncandescentLamp` не вистачає можливостей декларованих в класі `Lamp`. Впровадимо відповідну залежність в конструкторі класу `IncandescentLamp`:

1. В редакторі коду відкрити файл `IncandescentLamp.cs`.

2. Декларувати в тілі класу `IncandescentLamp` конструктор, який в якості параметра приймає об’єкт з назвою `lamp` типу `Lamp`. Відповідний код:

```
public IncandescentLamp(Lamp lamp) { }
```

3. Цей параметр (п. 2) потрібно зберегти в класі `IncandescentLamp` з доступом «тільки для читання». Для швидкого створення відповідного коду:

3.1. В редакторі коду розмістити курсор на назві параметру `lamp` в списку параметрів конструктора (п. 2).

3.2. Викликати контекстне меню `Quick actions and Refactorings…` (`Alt` + `Enter`).

3.3. В контекстному меню ЛКМ по `Create and assign property ‘Lamp’` (рис. 3.1). Публічну властивість тільки для читання обрано через те, що відповідно до п. 3 дод. 3 потрібно забезпечити доступ (у т.ч. поза межами класу `IncandescentLamp`) до загальних можливостей, які розміщено в класі `Lamp`.

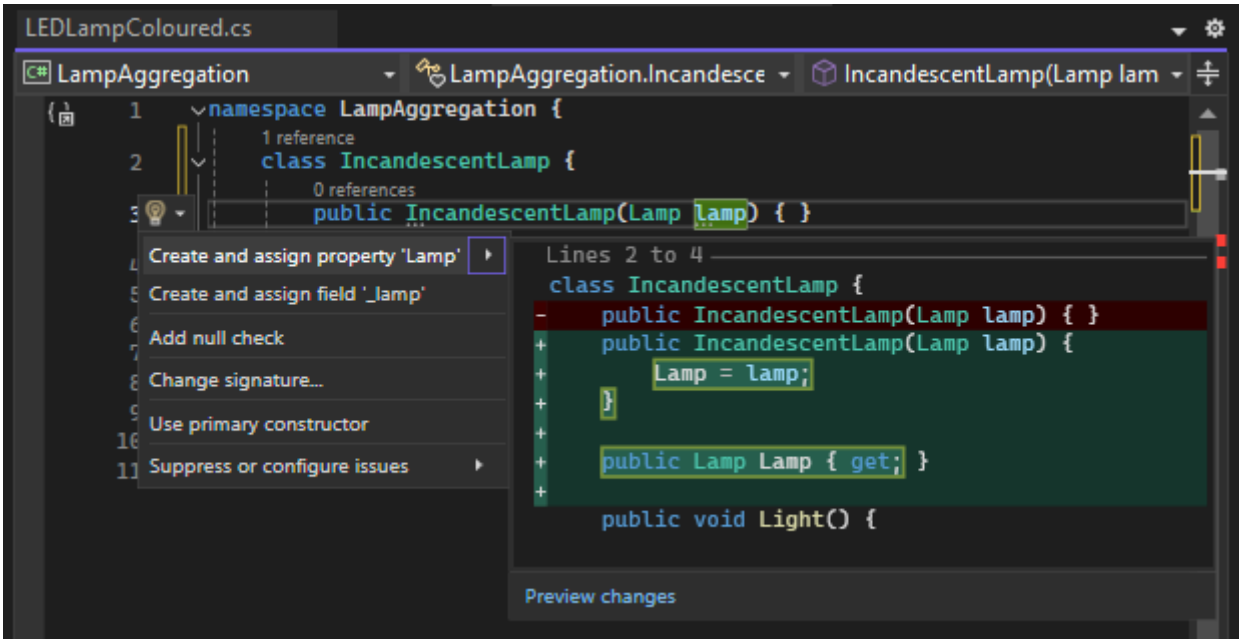


Рис. 3.1. Контекстне меню `Quick actions and Refactorings…` для параметру `lamp` в списку параметрів конструктора класу `IncandescentLamp`

Код класу IncandescentLamp:

```
class IncandescentLamp {
    public IncandescentLamp(Lamp lamp) {
        Lamp = lamp;
    }

    public Lamp Lamp { get; }

    public void Light() {
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Voltage} (В) " +
            $"на світло з {WattsNominal} (Вт)... Виконано.");
    }
}
```

За вихідними даними варіанту «Приклад» дод. 2 в класі LEDLamp не вистачає можливостей декларованих в класах Lamp та VoltageRectifier. Впровадимо відповідні залежності в конструкторі класу LEDLamp:

1. В редакторі коду відкрити файл LEDLamp.cs.

2. Декларувати в тілі класу LEDLamp конструктор, який в якості параметра приймає два об'єкти – з назвами lamp типу Lamp та rectifier типу VoltageRectifier. Відповідний код:

```
public LEDLamp(Lamp lamp, VoltageRectifier rectifier) { }
```

3. Ці параметри (п. 2) потрібно зберегти в класі LEDLamp з доступом «тільки для читання». Для швидкого створення відповідного коду:

3.1. В редакторі коду розмістити курсор на назві параметру lamp в списку параметрів конструктора (п. 2).

3.2. Викликати контекстне меню Quick actions and Refactorings… (Alt + Enter).

3.3. В контекстному меню ЛКМ по Create and assign property 'Lamp'. Публічну властивість тільки для читання обрано з тих же причин, що і в класі IncandescentLamp.

3.4. Розмістити курсор на назві параметру rectifier в списку параметрів конструктора (п. 2).

3.5. Виконати п. 3.2.

3.6. В контекстному меню ЛКМ по Create and assign field '_rectifier'. Приватне поле обрано через те, що відповідно до дод. 3 *не* вимагається забезпечити доступ (у т.ч. поза межами класу LEDLamp) до можливості, яку розміщено в класі VoltageRectifier. В іншому випадку можна реалізувати як публічну властивість тільки для читання, подібно до збереження параметру lamp.

Код класу LEDLamp:

```
class LEDLamp {
    private readonly VoltageRectifier _rectifier;
```

```
    public LEDLamp(Lamp lamp, VoltageRectifier rectifier) {
        Lamp = lamp;
        _rectifier = rectifier;
    }

    public Lamp Lamp { get; }

    public void Light() {
        RectifyVoltage();
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Voltage} (В) " +
            $"на світло з {WattsNominal} (Вт)... Виконано.");
    }
}
```

За вихідними даними варіанту «Приклад» дод. 2 в класі `LEDLampColoured` не вистачає можливостей, які декларовано в класах `Lamp` та `VoltageRectifier`. Для виправлення: виконати ті ж самі дії, за допомогою яких впроваджено залежності до класу `LEDLamp`. Остаточний код, яким доповнено тіло класу `LEDLampColoured`, є ідентичним коду, яким доповнено тіло класу `LEDLamp`, за виключенням назви конструктора – `LEDLampColoured`.

В класах `IncandescentLamp`, `LEDLamp` та `LEDLampColoured` в методі `Light()` виконується звертання до `Voltage` та `WattsNominal`. Ці властивості переміщено до класу `Lamp`, екземпляр якого доступний у відповідних класах через властивість `Lamp`. Для виправлення: замінити в класах `IncandescentLamp`, `LEDLamp` та `LEDLampColoured` звернення до `Voltage` та `WattsNominal` на **`Lamp.`**`Voltage` та **`Lamp.`**`WattsNominal`, відповідно.

В класах `LEDLamp` та `LEDLampColoured` в методі `Light()` виконується виклик методу `RectifyVoltage()`. Цей метод переміщено до класу `VoltageRectifier`. Для виправлення: замінити в класах `LEDLamp` та `LEDLampColoured` виклики цього методу на **`_rectifier.`**`RectifyVoltage()`.

В класі `VoltageRectifier` в методі `RectifyVoltage()` виконується звертання до `Voltage`. Але цю властивість переміщено до класу `Lamp`. Щоб не впроваджувати залежність класу `VoltageRectifier` від `Lamp`: змінити сигнатуру методу `RectifyVoltage()` так, щоб він приймав як параметр значення `Voltage`. За погодженнями щодо іменування сутностей в C#, локальні параметри іменуються починаючи з малої літери (“camel casing”), тому замість `Voltage` надрукуємо `voltage` типу `double`. Відповідно перейменуємо сутність в тілі методу `RectifyVoltage()`. Код класу `VoltageRectifier`:

```
class VoltageRectifier {
    public void RectifyVoltage(double voltage) {
        Console.WriteLine($"{this.GetType()} випрямляє " +
            $"напругу {voltage} (В)... Виконано.");
    }
}
```

Оскільки сигнатуру методу RectifyVoltage() класу VoltageRectifier змінено, то: внести зміни до рядків класів LEDLamp та LEDLampColoured, в яких викликається цей метод. Після цього виклик методу

```
_rectifier.RectifyVoltage();
```

замінено на

```
_rectifier.RectifyVoltage(Lamp.Voltage);
```

Код для вирішення третьої частини задачі можна покращити. З аналізу елементів класу Lamp видно, що після видалення методу Light() у цьому класі відсутні можливості характерні саме для *лампи*. Залишились можливості, які притаманні для будь-якого електричного пристрою. Тому доцільно всюди в коді проекту LampAggregation змінити назву Lamp на ElectricDevice (з англ. – електричний пристрій). Для швидкої зміни назви сутності у всьому проекті:

1. В редакторі коду відкрити файл Lamp.cs проекту LampAggregation.
2. Розмістити курсор на назві класу – Lamp.
3. Викликати меню для перейменування: натиснути комбінацію клавіш Ctrl + R + R.
4. Замість Lamp ввести нову назву – ElectricDevice. Натиснути клавішу Enter (або кнопку Apply).

Т.ч. створення коду для вирішення третьої частини задачі завершено. Остаточний текст програми зі змінами, які внесено після виконання четвертої частини задачі, подано в дод. 6.

Клас IncandescentLamp агрегує клас ElectricDevice. Класи LEDLamp та LEDLampColoured агрегують класи ElectricDevice та VoltageRectifier. Це досягнуто шляхом впровадження залежностей через параметри конструкторів класів IncandescentLamp, LEDLamp та LEDLampColoured. Якщо б C# дозволяв множинне успадкування, то класи ElectricDevice та VoltageRectifier можна було б декларувати як базові. Від класу ElectricDevice походили би класи IncandescentLamp, LEDLamp та LEDLampColoured, а від VoltageRectifier – тільки LEDLamp та LEDLampColoured.

Впровадження залежностей дозволяє створювати нові класи шляхом впровадження до них можливостей інших класів. Наприклад, потрібно створити класи «Радіоприймач» та «Powerbank», в кожному з яких повинні бути можливості, які вже реалізовано в класах ElectricDevice та VoltageRectifier. Достатньо до параметрів конструкторів класів «Радіоприймач» та «Powerbank» передати два об’єкти – типу ElectricDevice та типу VoltageRectifier. Це дозволяє делегувати відповідальність за опрацювання вказаних можливостей класам ElectricDevice та VoltageRectifier. Т.ч. нові класи створено як «набори» з існуючих елементів – класів ElectricDevice та VoltageRectifier. При цьому нові класи можуть належати будь-якій єрархії класів.

Для того, щоб виявити помилки на етапі компіляції доцільно скомпілювати проект. У разі наявності помилок – внести відповідні виправлення.

ЧАСТИНА № 4
ВНЕСЕННЯ НЕОБХІДНИХ ЗМІН ДО ВЗАЄМОДІЇ З СУТНОСТЯМИ, ЯКІ РЕАЛІЗОВАНО ЗА ДОПОМОГОЮ ВІДНОШЕННЯ «АГРЕГАЦІЯ»

Приклад виконання

Відповідно до четвертої частини задачі (розділ «Постановка задачі») потрібно виконати ті ж самі дії (дод. 3), що і в другій частині задачі, але з використанням сутностей, які реалізовано в третій частині (відношення «агрегація»). Скопіюємо код методу `Main` класу `Program` проекту `LampInheritance` до методу `Main` класу `Program` проекту `LampAggregation`:

1. У вікні `Solution Explorer` в проекті `LampInheritance` відкрити файл `Program.cs`. Виділити та скопіювати тіло методу `Main` класу `Program`.

2. В редакторі коду перейти до класу `Program` проекту `LampAggregation`. Замінити тіло методу `Main` класу `Program` на скопійований код (п. 1).

3. Закрити файл `Program.cs` проекту `LampInheritance`.

Для зручності у вікні `Solution Explorer` доцільно згорнути проект `LampInheritance`. Оскільки у даній частині буде виконано зміни в коді проекту `LampAggregation`, то доцільно встановити цей проект стартовим: у вікні `Solution Explorer` ПКМ по назві проекту `WinApp`, `Set As Startup Project`. Відкрити файл `Program.cs` проекту `LampAggregation`. Скомпілювати цей проект. У вікні `Error List` повинен відобразитися перелік помилок в методі `Main` класу `Program` проекту `LampAggregation`. Вони обумовлені тим, що в третій частині задачі виконано заміну відношення «успадкування» на відношення «агрегація» шляхом впровадження залежностей. Почнемо їх виправлення. Серед помилок є те, що класу `Lamp` не існує. Дійсно, в третій частині його було перейменовано на `ElectricDevice`. Тому замінимо у фрагменті коду, в якому декларовано список `lamps`, тип параметр `Lamp` на `ElectricDevice`:

```
List<ElectricDevice> lamps = new List<ElectricDevice> {
```

Наступна помилка – некоректно викликано конструктори під час створення екземплярів класів `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`. Дійсно, в третій частині конструктори цих класів було змінено так, щоб вони в якості параметру(ів) приймали екземпляри інших класів, замість того, щоб *походити* (успадковуватись) від них. Для виправлення: змінити виклики конструкторів під час ініціювання елементів списку `lamps` з таких

```
new IncandescentLamp(100) { Voltage = 1 },
```

```
new LEDLamp(8) { Voltage = 2 },
new LEDLampColoured(7) { Voltage = 2 },
new LEDLamp(12) { Voltage = 5 },
new LEDLamp(7.5) { Voltage = 5 }
```

на такі:

```
new IncandescentLamp(new ElectricDevice(100) { Voltage = 1 }),
new LEDLamp(new ElectricDevice(8) { Voltage = 2 },
    new VoltageRectifier()),
new LEDLampColoured(new ElectricDevice(7) { Voltage = 2 },
    new VoltageRectifier()),
new LEDLamp(new ElectricDevice(12) { Voltage = 5 },
    new VoltageRectifier()),
new LEDLamp(new ElectricDevice(7.5) { Voltage = 5 },
    new VoltageRectifier())
```

Наступна помилка – елементами списку `lamps` не можуть бути екземпляри класів `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`. Дійсно, в другій частині задачі код працював тому, що типом-параметром списку `lamps` було вказано базовий клас `Lamp`, від якого походили класи `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`. Перетворення типів вгору за єрархією успадкування завжди успішно, тому типи екземплярів похідних класів неявно перетворювались на тип `Lamp` і могли бути елементами списку `lamps`. Але у даному проекті такої єрархії не існує. Тому потрібно створити якусь сутність, на яку можуть бути перетворені типи екземплярів класів `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`, щоб це дозволило екземплярам цих класів бути присвоєними елементам списку `lamps`.

Відповідно дод. 3 для списку `lamps` (елементи якого ініційовано екземплярами класів `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`) потрібно:

- за п. 3 дод. 3 – для кожного елементу списку `lamps` викликати метод `Light()`;
- за п. 4 дод. 3 – створити запит LINQ, в якому сортувати елементу списку `lamps` за спаданням за властивістю `Voltage`, а потім сортувати за зростанням за властивістю `WattsNominal`;
- за п. 5 дод. 3 – вивести на екран консольного додатку рядкове подання елементів результату виконання запиту LINQ викликавши метод `ToString()`.

З аналізу вимог дод. 3 видно, що через класи `IncandescentLamp`, `LEDLamp` та `LEDLampColoured` доступні такі елементи:

- Метод `Light()` – безпосередньо з тіл цих класів.
- Властивості `Voltage` та `WattsNominal` – через властивість типу `ElectricDevice`.

Тому створимо інтерфейс, який би було реалізовано класами `IncandescentLamp`, `LEDLamp` та `LEDLampColoured`, щоб:

– цей інтерфейс можна було вказати як тип-параметр списку `lamps` замість `ElectricDevice`
– і отримати доступ до методу `Light()` та властивостей `Voltage` і `WattsNominal`.

Обрано використати саме інтерфейс, а не базовий клас, оскільки інтерфейс не зобов'язує сутність, яка його використовує, перебувати у відношенні «успадкування» та може належати іншій єрархії класів. Т.ч. потрібен інтерфейс, який би дозволив надати доступ до методу `Light()` та екземпляру класу `ElectricDevice`. Оскільки цей інтерфейс відповідає за можливість світити та звертатися до електричного пристрою, то нехай його назвою буде `IElectricallyLightable` (з англ. – здатний електрично світитися). Для швидкого створення відповідного коду:

1. В редакторі коду відкрити файл `IncandescentLamp.cs` в проекті `LampAggregation`.

2. Розмістити курсор на назві цього класу.

3. Викликати вікно `Extract interface` (рис. 4.1): натиснути комбінацію клавіш `Ctrl` + `R`, `Ctrl` + `I`.

4. У вікні `Extract interface`:

4.1. В полі `New Type Name` ввести назву інтерфейсу `IElectricallyLightable`.

4.2. В полі `Select public members to form instance` відмітити елементи: метод `Light()`, властивість `Lamp`.

4.3. Натиснути на кнопку `OK`.

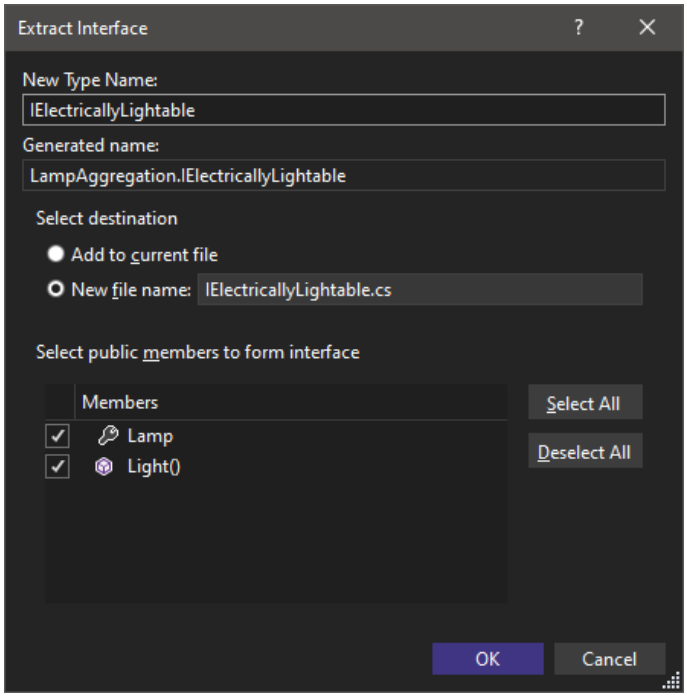


Рис. 4.1. Вікно `Extract Interface`, викликане для класу `IncandescentLamp`

Після цього буде створено новий файл IElectricallyLightable.cs, який містить однойменний інтерфейс.

```
internal interface IElectricallyLightable {
    ElectricDevice Lamp { get; }
    void Light();
}
```

Заголовок класу IncandescentLamp буде змінено так, щоб клас реалізовувати інтерфейс IElectricallyLightable, тобто:

```
class IncandescentLamp : IElectricallyLightable {
```

Замінимо в тілі методу Main класу Program проекту LampAggregation тип-параметр списку lamps на IElectricallyLightable, тобто:

```
List<IElectricallyLightable> lamps =
    new List<IElectricallyLightable> {
```

Якщо зараз скомпілювати проект, то у фрагментах коду, де елементи списку lamps ініційовано екземплярами класу IncandescentLamp, помилка зникне. Для того, щоб помилка також зникла у фрагментах коду, де елементи списку lamps ініційовано екземплярами класів LEDLamp та LEDLampColoured, змінити заголовки цих класів так, щоб вони реалізовували інтерфейс IElectricallyLightable, тобто:

```
class LEDLamp : IElectricallyLightable {
class LEDLampColoured : IElectricallyLightable {
```

Остання помилка етапу компіляції – у фрагменті коду, де декларовано запит LINQ (п. 4 дод. 3). Оскільки властивості Voltage і WattsNominal доступні через властивість Lamp типу ElectricDevice, то для виправлення – змінити запит на такий:

```
var query = lamps
    .OrderByDescending(x => x.Lamp.Voltage)
    .ThenBy(x => x.Lamp.WattsNominal);
```

Якщо скомпілювати код проекту LampAggregation та запустити на виконання, то під час виконання запиту (виклик методу ToString() відповідно п. 5 дод. 3) завжди виводитиметься назва тих класів, екземплярами яких є відповідні елементи списку lamps. Це реалізація методу ToString() за промовчанням. Дійсно, перевизначений метод ToString() переміщено до класу ElectricDevice і саме цей клас відповідає за опрацювання властивостей Voltage і WattsNominal.

Потрібно реалізувати загальну можливість перетворювати цей об’єкт на рядок (дод. 2). Можливо, найшвидше цього можна досягти так: доповнити тіла класів IncandescentLamp, LEDLamp та LEDLampColoured перевизначенням методу ToString(), який викликатиме метод ToString() з властивості Lamp:

```
public override string ToString() {
```

```
    return $"{this.GetType()} : ..." + Environment.NewLine
        + Lamp.ToString();
}
```

Додавання такого однакового фрагменту коду до класів `IncandescentLamp`, `LEDLamp` та `LEDLampColoured` порушує принцип DRY. Але обов'язкове його виконання може призвести до створення нових, часом надлишкових, рівнів абстракції [3]. Тому принцип DRY, як і будь-які інші принципи, потрібно використовувати тоді, коли це доцільно.

Якщо скомпілювати код проекту, запустити на виконання і порівняти результат тексту, виведеного до консольного вікна, з результатом, який було виведено після виконання проекту `LampInheritance`, то вони майже ідентичні. Відмінність: в частині рядків, де виведено інформацію після виконання запиту LINQ (п. 5 дод. 3) доповнено рядками з назвою класу `ElectricDevice` через те, що в цьому класі у перевизначеному методі `ToString()` міститься виведення значення `this.GetType()`.

Т.ч. створення коду для вирішення четвертої частини задачі завершено. Остаточний текст методу `Main` класу `Program` проекту `LampAggregation` подано в дод. 7, а сутності цього проекту зі змінами, які внесено під час виконання четвертої частини задачі, подано в дод. 6. Відповідну діаграму типів, створену засобами `Class designer`, подано в дод. 9.

Для того, щоб додатково зменшити сполучення з класами `ElectricDevice` та `VoltageRectifier` (зробити цей проект зі слабким сполученням – "loosely coupled") можна:

1. Декларувати для цих класів інтерфейси `IElectricDevice` та `IVoltageRectifier`, відповідно.
2. Змінити заголовки цих класів так, щоб кожен з них реалізовував відповідний інтерфейс.
3. Замінити в класах `IncandescentLamp`, `LEDLamp` та `LEDLampColoured` клас `ElectricDevice` на інтерфейс `IElectricDevice`.
4. Замінити в класах `LEDLamp` та `LEDLampColoured` клас `VoltageRectifier` на інтерфейс `IVoltageRectifier`.

Контрольні запитання

1. Клас. Відношення «успадкування».
2. Інтерфейс. Реалізація інтерфейсу в класі.
3. Шаблон проектування «впровадження залежності» через конструктор.
4. Проект зі слабким сполученням. Способи розроблення таких проектів.
5. Рефакторинг коду. Вилучення базового класу та інтерфейсу з класу засобами Visual Studio.

ДОДАТОК 1

ФОРМА ТИТУЛЬНОГО АРКУШУ ЗВІТУ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
УКРАЇНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
НАУКИ І ТЕХНОЛОГІЙ

КАФЕДРА: «Автоматика та телекомунікації»

ЗВІТ
З ІНДИВІДУАЛЬНОГО ЗАВДАННЯ

ТЕМА: «Відношення між класами»
НАВЧАЛЬНА ДИСЦИПЛІНА: «Основи інформаційних технологій»

ВИКОНАВ (*ВИКОНАЛА*):
студент (*студентка*) групи _____
Ім'я ПРІЗВИЩЕ
шифр – _____

ПЕРЕВІРИВ (*ПЕРЕВІРИЛА*):
посада
Ім'я ПРІЗВИЩЕ

20__ р.

ДОДАТОК 2

ВИХІДНІ ДАНІ ВАРІАНТІВ ЗАДАЧ В ЧАСТИНІ РЕАЛІЗАЦІЇ СУТНОСТЕЙ

№ вар.	Вид можливості	Сутності		
		B[1]	C[1]	D[1]
Приклад	Назва	Лампа розжарювання	Лампа світлодіодна	Лампа світлодіодна кольорова
	Загальні[2]	Одноразово (тільки під час створення об'єкта) записувати: значення номінальної потужності, Вт. Багаторазово читати це значення. Багаторазово записувати та читати: значення електричної напруги, В. Перетворювати цей об'єкт на рядок – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта*; номінальна потужність, Вт : *значення номінальної потужності*; електрична напруга, В : *значення електричної напруги*»		
	Умовно загальні[3]	Перетворювати напругу та потужність на світло – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* перетворює напругу *значення електричної напруги* (В) на світло з *значення номінальної потужності* (Вт)… Виконано.».	Перетворювати напругу та потужність на світло – випрямити напругу, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності B.	Перетворювати напругу та потужність на світло – випрямити напругу, потім зафарбувати світло, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності B.
	Частково загальні[4]	–	Випрямляти напругу – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* випрямляє напругу *значення електричної напруги* (В)… Виконано.».	
	Унікальні[5]	–	–	Зафарбовувати світло – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* зафарбовує світло… Виконано.»

№ вар.	Вид можливості	Сутності		
		B[1]	C[1]	D[1]
0	Назва	Вольтметр постійної напруги	Вольтметр змінної напруги	Вольтметр форми сигналу
	Загальні[2]	Одноразово (тільки під час створення об'єкта) записувати: нижню та верхню границі шкали, В. Багаторазово читати ці значення. Багаторазово записувати та читати: електричний сигнал (без одиниць виміру). Перетворювати цей об'єкт на рядок – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта*; нижня та верхня границі шкали, В : *нижня границя шкали, верхня границі шкали*; електричний сигнал : *значення електричного сигналу*»		
	Умовно загальні[3]	Перетворювати сигнал на результат вимірювання – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* повертає виміряне значення постійної електричної напруги, яке дорівнює *значення електричного сигналу* Вольт.».	Перетворювати сигнал на результат вимірювання – обчислити середньоквадратичне значення електричної напруги, потім вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* повертає виміряне значення змінної електричної напруги, яке дорівнює *значення електричного сигналу* Вольт.».	Перетворювати сигнал на результат вимірювання – обчислити середньоквадратичне значення електричної напруги, потім виміряти амплітуду змінної напруги, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності C.
	Частково загальні[4]	–	Обчислювати середньоквадратичне значення електричної напруги – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* обчислює середньоквадратичне значення змінної електричної напруги на основі *значення електричного сигналу*, Вольт.».	
	Унікальні[5]	–	–	Виміряти амплітуду змінної напруги – вивести на екран консольного вікна повідомлення «*назва типу*

№ вар.	Вид можливості	Сутності		
		B[1]	C[1]	D[1]
				цього об'єкта обчислює амплітуду змінної електричної напруги на основі *значення електричного сигналу*, Вольт.».
1	Назва	Резистор сталого опору	Резистор змінного опору	Фоторезистор
	Загальні[2]	Одноразово (тільки під час створення об'єкта) записувати: значення номінального опору, Ом. Багаторазово читати це значення. Багаторазово записувати та читати: значення електричного струму, що протікає через резистор, А. Перетворювати цей об'єкт на рядок – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта*; номінальний опір, Ом : *значення номінального опору*; струм, А : *значення електричного струму*»		
	Умовно загальні[3]	Створювати необхідний електричний опір в електричному колі – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* створює електричний опір *значення номінального опору* Ом під час протікання електричного струму *значення електричного струму* Ампер… Виконано.».	Створювати необхідний електричний опір в електричному колі – визначити коефіцієнт перетворення опору з урахуванням положення повзунка резистора, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності B.	Створювати необхідний електричний опір в електричному колі – виміряти рівень освітлення фоторезистора, потім визначити коефіцієнт перетворення опору з урахуванням положення повзунка резистора, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності B.
	Частково загальні[4]	–	Визначати коефіцієнт перетворення опору з урахуванням положення повзунка резистора – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* створює електричний опір з урахуванням положення повзунка резистора… Виконано.».	

№ вар.	Вид можливості	Сутності		
		B[1]	C[1]	D[1]
	Унікальні[5]	–	–	Виміряти рівень освітлення фоторезистора – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* створює електричний опір з урахуванням рівня освітлення… Виконано.».
2	Назва	Коаксіальний кабель	Кручена пара	Оптоволокно
	Загальні[2]	Одноразово (тільки під час створення об'єкта) записувати: максимальну пропускну здатність лінії, біт/с. Багаторазово читати це значення. Багаторазово записувати та читати: поточну швидкість передавання повідомлення, біт/с. Перетворювати цей об'єкт на рядок – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта*; максимальна пропускна здатність, біт/с : *значення максимальної пропускної здатності*; поточна швидкість передавання повідомлення, біт/с : *значення поточної швидкості передавання повідомлення*»		
	Умовно загальні[3]	Передавати повідомлення – передати електричну енергію, потім вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* передає повідомлення на швидкості *значення поточної швидкості передавання повідомлення*, біт/с… Виконано.».	Передавати повідомлення – ідентично сутності B.	Передавати повідомлення – не відчувати впливу електромагнітних завад, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності B.

№ вар.	Вид можливості	Сутності		
		B[1]	C[1]	D[1]
	Частково загальні[4]	Передавати електричну енергію – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* передає електричну енергію… Виконано.».		–
	Унікальні[5]	–	–	Не відчувати впливу електромагнітних завад – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* не відчуває впливу електромагнітних завад… Виконано.».
3	Назва	Мобільний зв'язок 2G	Мобільний зв'язок 3G	Мобільний зв'язок 4G
	Загальні[2]	Одноразово (тільки під час створення об'єкта) записувати: ширину смуги частот каналу, кГц. Багаторазово читати це значення. Багаторазово записувати та читати: поточну швидкість передавання повідомлення, біт/с. Перетворювати цей об'єкт на рядок – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта*; ширина смуги частот каналу, кГц : *значення ширини смуги частот каналу*; поточна швидкість передавання повідомлення, біт/с : *значення поточної швидкості передавання повідомлення*»		
	Умовно загальні[3]	Передавати голосові повідомлення – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* передає голосові повідомлення зі швидкістю *значення поточної швидкість передавання повідомлення* біт/с… Виконано.».	Передавати голосові повідомлення – передати сигнал з використанням широкої смуги частот, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності B.	Передавати голосові повідомлення – передати сигнал з використанням широкої смуги частот, потім використати технологію LTE, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності B.

№ вар.	Вид можливості	Сутності		
		B[1]	C[1]	D[1]
	Частково загальні[4]	–	Передавати сигнал з використанням широкої смуги частот – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* передає сигнал з використанням широкої смуги частот… Виконано.».	
	Унікальні[5]	–	–	Використовувати технологію LTE – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* реалізує технологію Long-Term Evolution… Виконано.».
4	Назва	Аналоговий сигнал	Дискретний сигнал	Цифровий сигнал
	Загальні[2]	Одноразово (тільки під час створення об'єкта) записувати: мінімальний та максимальний рівні амплітуди сигналу, В. Багаторазово читати ці значення. Багаторазово записувати та читати: тривалість сигналу, с. Перетворювати цей об'єкт на рядок – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта*; мінімальний рівень амплітуди сигналу, В : *значення мінімального рівня амплітуди сигналу*; максимальний рівень амплітуди сигналу, В : *значення максимального рівня амплітуди сигналу*; тривалість сигналу, с : *значення тривалості сигналу*»		
	Умовно загальні[3]	Записувати на цифровий носій даних – дискретизувати за часом, квантувати за амплітудою та кодувати, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності D.	Записувати на цифровий носій даних – квантувати за амплітудою та кодувати, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності D.	Записувати на цифровий носій даних – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* записує сигнал тривалістю *значення тривалості сигналу*, с на цифровий носій даних… Виконано.».

№ вар.	Вид можливості	Сутності		
		B[1]	C[1]	D[1]
	Частково загальні[4]	Квантувати за амплітудою та кодувати – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* виконує квантування (округлення) дискретних відліків сигналу за амплітудою та їх кодування… Виконано.».		–
	Унікальні[5]	Дискретизувати за часом – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* виконує дискретизацію неперервного сигналу за часом… Виконано.».	–	–
5	Назва	Пасажирський поїзд	Швидкісний поїзд	Вантажний поїзд
	Загальні[2]	Одноразово (тільки під час створення об'єкта) записувати: значення максимальної швидкості, км/год. Багаторазово читати це (ці) значення. Багаторазово записувати та читати: поточну швидкість, км/год. Перетворювати цей об'єкт на рядок – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта*; поточна швидкість, км/год : *значення поточної швидкості*; максимальна швидкість, км/год : *значення максимальної швидкості*»		
	Умовно загальні[3]	Виконати перевезення – виконати посадку пасажирів, потім вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* виконує перевезення на швидкості *значення поточної швидкості*, км/год… Виконано.».	Виконати перевезення – ідентично сутності B.	Виконати перевезення – завантажити вантаж, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності B.

№ вар.	Вид можливості	Сутності		
		B[1]	C[1]	D[1]
	Частково загальні[4]	Виконати посадку пасажирів – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* виконує посадку пасажирів… Виконано.».		–
	Унікальні[5]	–	–	Завантажити вантаж – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* завантажує вантаж… Виконано.».
6	Назва	Стрілочний привід з двигуном МСТ-0,3 на стрілці з маркою хрестовини 1/11	Стрілочний привід з двигуном МСП-0,15 на стрілці з маркою хрестовини 1/11	Стрілочний привід з двигуном МСП-0,25 на стрілці з маркою хрестовини 1/11
	Загальні[2]	Одноразово (тільки під час створення об'єкта) записувати: максимальний час переводу стрілки, с. Багаторазово читати це значення. Багаторазово записувати та читати: час робочого переводу стрілки, с. Перетворювати цей об'єкт на рядок – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта*; максимальний час переводу стрілки, с : *значення максимального часу переводу стрілки*; час робочого переводу стрілки, с : *значення часу робочого переводу стрілки*»		
	Умовно загальні[3]	Переводити гостряки стрілочного переводу – використати енергію змінного струму, потім вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* виконує переведення стрілки за *значення часу робочого переводу стрілки*, с… Виконано.».	Переводити гостряки стрілочного переводу – використати енергію постійного струму, потім вивести на екран консольного вікна повідомлення, яке ідентичне тому, що в сутності B.	Переводити гостряки стрілочного переводу – ідентично тому, що в сутності C.

№ вар.	Вид можливості	Сутності		
		B[1]	C[1]	D[1]
	Частково загальні[4]	–	Використовувати енергію постійного струму – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* використовує електричну енергію постійного струму.».	
	Унікальні[5]	Використовувати енергію змінного струму – вивести на екран консольного вікна повідомлення «*назва типу цього об'єкта* використовує електричну енергію змінного струму.».	–	–

[1] Фрагменти, подані курсивом, позначають те, що під час реалізації відповідних можливостей в тексті програми потрібно забезпечити виведення значень, які відповідають цим фрагментам.

[2] Іменування та реалізації ідентичні для всіх сутностей.

[3] Іменування ідентичні для всіх сутностей, а реалізації можуть відрізнятись.

[4] Існують не для всіх сутностей (тільки для двох з трьох), іменування та реалізації ідентичні.

[5] Існують не для всіх сутностей (тільки для однієї з трьох).

ДОДАТОК З

ПОСТАНОВКА ЗАДАЧІ В ЧАСТИНІ ВЗАЄМОДІЇ З РЕАЛІЗОВАНИМИ СУТНОСТЯМИ

1. Декларувати колекцію `X` загальною кількістю не менше п'яти елементів. Причому, колекція `X` повинна дозволяти збереження екземплярів всіх класів, що реалізовано відповідно дод. 2 (`B`, `C` та `D`).

2. Записати до елементів колекції `X` (п. 1) екземпляри класів, які реалізовано відповідно дод. 2 (`B`, `C` та `D`), не менше ніж по одному для кожної з сутностей. Ініціювати стан цих екземплярів (їх поля, властивості) довільними значеннями.

3. Для кожного елементу колекції `X` викликати можливість, що в дод. 2 віднесено до умовно загальних.

4. Створити запит LINQ, в якому сортувати за спаданням елементи колекції `X` за одним з параметрів (обрати самостійно), що в дод. 2 віднесено до загальних, які можна багаторазово записувати та читати. А потім сортувати за зростанням за одним з параметрів (обрати самостійно), що в дод. 2 віднесено до загальних, які можна одноразово записувати та багаторазово читати.

5. Вивести на екран консольного додатку рядкове подання елементів результату виконання запиту LINQ (п. 4).

ДОДАТОК 4

ТЕКСТ ПРОГРАМИ ЗА РЕЗУЛЬТАТОМ ВИКОНАННЯ ПЕРШОЇ ЧАСТИНИ ЗАДАЧІ (У ПРОСТОРІ ІМЕН `LampInheritance`)

```
internal abstract class Lamp {
    double _wattsNominal;
    public double Voltage { get; set; }
    protected Lamp(double wattsNominal) {
        _wattsNominal = wattsNominal;
    }
    public double WattsNominal { get => _wattsNominal; }
    public abstract void Light();
    public override string ToString() {
        return $"{this.GetType()} : ..."
            + Environment.NewLine
            + $"номінальна потужність, Вт : {WattsNominal}"
            + Environment.NewLine
            + $"електрична напруга, В : {Voltage}";
    }
}
class IncandescentLamp : Lamp {
    public IncandescentLamp(double wattsNominal) : base(wattsNominal)
    { }
    public override void Light() {
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Voltage} (В) " +
            $"на світло з {WattsNominal} (Вт)... Виконано.");
    }
}
class LEDLamp : Lamp {
    public LEDLamp(double wattsNominal) : base(wattsNominal)
    { }
    public override void Light() {
        RectifyVoltage();
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Voltage} (В) " +
            $"на світло з {WattsNominal} (Вт)... Виконано.");
    }
    public void RectifyVoltage() {
        Console.WriteLine($"{this.GetType()} випрямляє " +
            $"напругу {Voltage} (В)... Виконано.");
    }
}
class LEDLampColoured : Lamp {
    public LEDLampColoured(double wattsNominal) : base(wattsNominal)
    { }
    public override void Light() {
        RectifyVoltage();
```

```
        ColourTheLight();
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Voltage} (В) " +
            $"на світло з {WattsNominal} (Вт)... Виконано.");
    }
    public void RectifyVoltage() {
        Console.WriteLine($"{this.GetType()} випрямляє " +
            $"напругу {Voltage} (В)... Виконано.");
    }
    public void ColourTheLight() {
        Console.WriteLine($"{this.GetType()} зафарбовує " +
            $"світло... Виконано.");
    }
}
```

ДОДАТОК 5

ТЕКСТ ПРОГРАМИ ЗА РЕЗУЛЬТАТОМ ВИКОНАННЯ ДРУГОЇ ЧАСТИНИ ЗАДАЧІ (У ПРОСТОРІ ІМЕН `LampInheritance`)

```
internal class Program {
    static void Main(string[] args) {
        Console.OutputEncoding = Encoding.Unicode;

        List<Lamp> lamps = new List<Lamp> {
            new IncandescentLamp(100) { Voltage = 1 },
            new LEDLamp(8) { Voltage = 2 },
            new LEDLampColoured(7) { Voltage = 2 },
            new LEDLamp(12) { Voltage = 5 },
            new LEDLamp(7.5) { Voltage = 5 }
        };

        foreach (var item in lamps) {
            item.Light();
            Console.WriteLine();
        }

        var query = lamps
            .OrderByDescending(x => x.Voltage)
            .ThenBy(x => x.WattsNominal);

        Console.WriteLine("Результат виконання запиту:");
        foreach (var item in query) {
            Console.WriteLine(item.ToString());
            Console.WriteLine();
        }
    }
}
```

ДОДАТОК 6

ТЕКСТ ПРОГРАМИ ЗА РЕЗУЛЬТАТОМ ВИКОНАННЯ ТРЕТЬОЇ ЧАСТИНИ ЗАДАЧІ (У ПРОСТОРІ ІМЕН `LampAggregation`)

```
class IncandescentLamp : IElectricallyLightable {
    public IncandescentLamp(ElectricDevice lamp) {
        Lamp = lamp;
    }

    public ElectricDevice Lamp { get; }

    public void Light() {
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Lamp.Voltage} (В) " +
            $"на світло з {Lamp.WattsNominal} (Вт)... Виконано.");
    }
    public override string ToString() {
        return $"{this.GetType()} : ..." + Environment.NewLine
            + Lamp.ToString();
    }
}
class LEDLamp : IElectricallyLightable {
    private readonly VoltageRectifier _rectifier;
    public LEDLamp(ElectricDevice lamp, VoltageRectifier rectifier) {
        Lamp = lamp;
        _rectifier = rectifier;
    }
    public ElectricDevice Lamp { get; }
    public void Light() {
        _rectifier.RectifyVoltage(Lamp.Voltage);
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Lamp.Voltage} (В) " +
            $"на світло з {Lamp.WattsNominal} (Вт)... Виконано.");
    }
    public override string ToString() {
        return $"{this.GetType()} : ..." + Environment.NewLine
            + Lamp.ToString();
    }
}
class LEDLampColoured : IElectricallyLightable {
    private readonly VoltageRectifier _rectifier;
    public ElectricDevice Lamp { get; }
    public LEDLampColoured(ElectricDevice lamp,
        VoltageRectifier rectifier) {
        Lamp = lamp;
        _rectifier = rectifier;
    }
    public void Light() {
```

```
        _rectifier.RectifyVoltage(Lamp.Voltage);
        ColourTheLight();
        Console.WriteLine($"{this.GetType()} " +
            $"перетворює напругу {Lamp.Voltage} (В) " +
            $"на світло з {Lamp.WattsNominal} (Вт)... Виконано.");
    }
    public void ColourTheLight() {
        Console.WriteLine($"{this.GetType()} зафарбовує " +
            $"світло... Виконано.");
    }
    public override string ToString() {
        return $"{this.GetType()} : ..." + Environment.NewLine
            + Lamp.ToString();
    }
}
internal class ElectricDevice {
    double _wattsNominal;
    public double Voltage { get; set; }
    public ElectricDevice(double wattsNominal) {
        _wattsNominal = wattsNominal;
    }
    public double WattsNominal { get => _wattsNominal; }
    public override string ToString() {
        return $"{this.GetType()} : ..."
            + Environment.NewLine
            + $"номінальна потужність, Вт : {WattsNominal}"
            + Environment.NewLine
            + $"електрична напруга, В : {Voltage}";
    }
}
class VoltageRectifier {
    public void RectifyVoltage(double voltage) {
        Console.WriteLine($"{this.GetType()} випрямляє " +
            $"напругу {voltage} (В)... Виконано.");
    }
}
internal interface IElectricallyLightable {
    ElectricDevice Lamp { get; }
    void Light();
}
```

ДОДАТОК 7

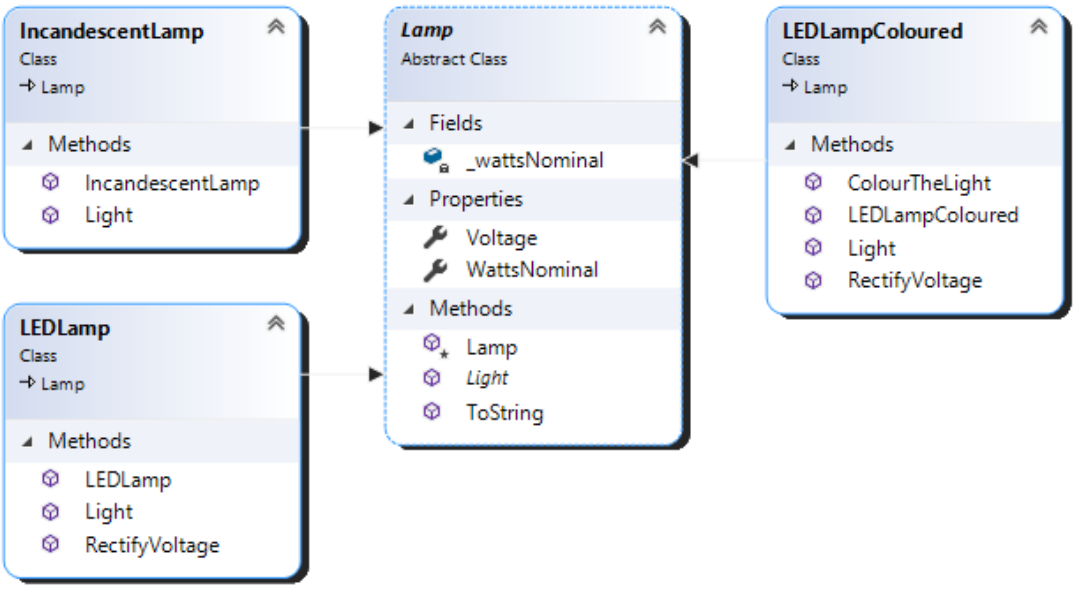
ТЕКСТ ПРОГРАМИ ЗА РЕЗУЛЬТАТОМ ВИКОНАННЯ ЧЕТВЕРТОЇ ЧАСТИНИ ЗАДАЧІ (У ПРОСТОРІ ІМЕН `LampAggregation`)

```
internal class Program {
    static void Main(string[] args) {
        Console.OutputEncoding = Encoding.Unicode;
```

```
        List<IElectricallyLightable> lamps =
            new List<IElectricallyLightable> {
                new IncandescentLamp(
                    new ElectricDevice(100) { Voltage = 1 }),
                new LEDLamp(
                    new ElectricDevice(8) { Voltage = 2 },
                    new VoltageRectifier()),
                new LEDLampColoured(
                    new ElectricDevice(7) { Voltage = 2 },
                    new VoltageRectifier()),
                new LEDLamp(
                    new ElectricDevice(12) { Voltage = 5 },
                    new VoltageRectifier()),
                new LEDLamp(
                    new ElectricDevice(7.5) { Voltage = 5 },
                    new VoltageRectifier())
        };

        foreach (var item in lamps) {
            item.Light();
            Console.WriteLine();
        }

        var query = lamps
            .OrderByDescending(x => x.Lamp.Voltage)
            .ThenBy(x => x.Lamp.WattsNominal);

        Console.WriteLine("Результат виконання запиту:");
        foreach (var item in query) {
            Console.WriteLine(item.ToString());
            Console.WriteLine();
        }
    }
}
```

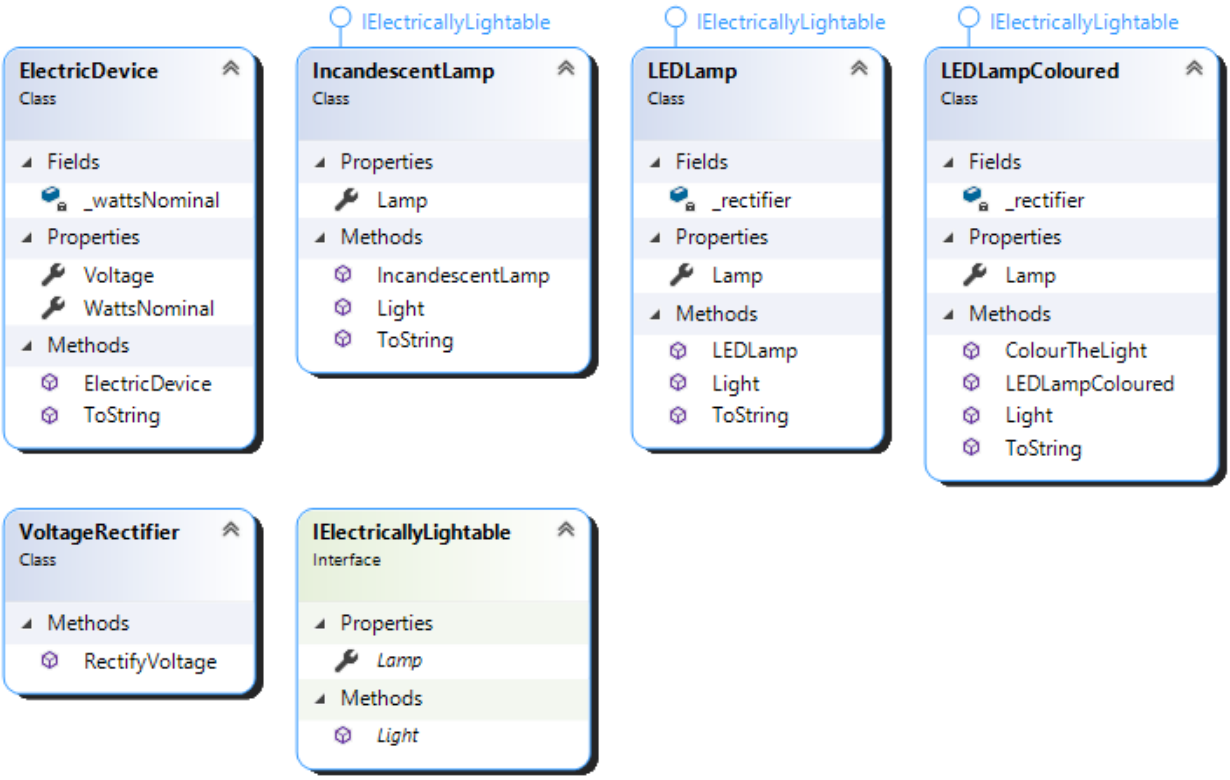
ДОДАТОК 8

ДІАГРАМА ТИПІВ ЗА РЕЗУЛЬТАТОМ ВИКОНАННЯ ПЕРШОЇ ЧАСТИНИ ЗАДАЧІ



ДОДАТОК 9

ДІАГРАМА ТИПІВ ЗА РЕЗУЛЬТАТОМ ВИКОНАННЯ ТРЕТЬОЇ ЧАСТИНИ ЗАДАЧІ



СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. ДСТУ ISO/IEC 2382:2017. Інформаційні технології. Словник термінів. Чинний від 2019–01–01. Вид. офіц. Київ : УкрНДНЦ. 2020. 464 с.
2. Microsoft Learn. URL: https://learn.microsoft.com/en-gb (дата звернення: 27.07.2023).
3. Don't repeat yourself (DRY) in Software Development. URL: https://www.geeksforgeeks.org/dont-repeat-yourselfdry-in-software-development/ (дата звернення: 15.08.2024).
4. Malshika A. Yohan. Mastering SOLID Principles in C#: A Practical Guide. URL: https://www.syncfusion.com/blogs/post/mastering-solid-principles-csharp (дата звернення: 15.08.2024).
5. Мартін Р. Чистий код: створення і рефакторинг за допомогою Agile / пер. з англ. І. Бондар-Терещенко. – Харків : Вид-во «Ранок»: Фабула, 2023. –448 с.
6. Code refactoring. URL: https://en.wikipedia.org/wiki/Code_refactoring (дата звернення: 15.08.2024).

Навчально-методичне видання

Рибалка Роман Володимирович,
Гончаров Костянтин Вікторович,
Тимошенко Людмила Сергіївна

ОСНОВИ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-методичні рекомендації
до виконання індивідуального завдання

Електронне видання

В авторській редакції
Комп'ютерна верстка Р. В. Рибалка

Зареєстровано НМВ УДУНТ (№ 1.866 від 29.04.2026)

Формат 60х84 $^{1/16}$. Ум. друк. арк. 3,41. Обл.-вид. арк. 3,36.
Зам. № 45

Видавець: Український державний університет науки і технологій
вул. Лазаряна, 2, ауд. 2216, м. Дніпро, 49010.
Свідоцтво суб'єкта видавничої справи ДК № 7709 від 14.12.2022

Адреса видавця та дільниці оперативної поліграфії:
вул. Лазаряна, 2, Дніпро, 49010