

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО

В.о. завідувача кафедри

Олександр КОВАЛЬ

«__» _____ 2025 р.

Дипломна робота

на здобуття ступеня бакалавра

**за освітньо-професійною програмою «Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення**

**на тему: «Програмне забезпечення оптимальної комутації різномірних
сонячних панелей в сонячну електростанцію»**

Виконав:

студент IV курсу, групи ТВ-11

Андрійчук Антон Юрійович

(прізвище, ім'я, по батькові)

_____ (підпис)

Керівник:

доктор технічних наук, професор, професор
кафедри ІПЗЕ Барабаш О. В.

_____ (посада, науковий ступінь, вчене звання, прізвище та ініціали)

_____ (підпис)

Рецензент:

доктор технічних наук, професор, професор
кафедри ЦТЕ Отрох С. І.

_____ (посада, науковий ступінь, вчене звання, прізвище та ініціали)

_____ (підпис)

Засвідчую, що у цій дипломній роботі
немає запозичень із праць інших авторів
без відповідних посилань.

Студент

_____ (підпис)

Київ – 2025

**Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»**

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці»

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

_____ Олександр КОВАЛЬ

«___» _____ 2025р.

ЗАВДАННЯ

на дипломну роботу студенту

_____ Андрійчуку Антону Юрійовичу

(прізвище, ім'я, по батькові)

1. Тема роботи: Програмне забезпечення оптимальної комутації різнорідних
сонячних панелей в сонячну електростанцію

керівник роботи: д-р техн. наук, проф., проф. кафедри ІПЗЕ Барабаш О.В.

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «02» червня 2025р. №1875-с

2. Строк подання студентом роботи «09» червня 2025р.

3. Вихідні дані до роботи: мова програмування Python, фреймворк Qt,
модуль bigtree

4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити): Розробка математичної моделі; проєктування та реалізація графічного інтерфейсу; налагодження сумісності систем та модулів проєкту; тестування.

5. Перелік ілюстративного матеріалу: діаграми прецедентів, класів; супровідні приклади та схеми.

6. Дата видачі завдання «31» жовтня 2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	30.10.2024	Виконано
2	Дослідження предметної області	31.10.2024 – 01.11.2024	Виконано
3	Дослідження існуючих рішень	02.11.2024 – 01.12.2024	Виконано
4	Постановка вимог до проєктування системи	02.12.2024 – 12.01.2025	Виконано
5	Розробка програмного продукту	13.01.2025 – 11.05.2025	Виконано
6	Тестування	12.05.2025 – 15.05.2025	Виконано
7	Захист програмного продукту	12.05.2025 – 15.05.2025	Виконано
8	Оформлення дипломної роботи	19.05.2025 – 01.06.2025	Виконано
9	Передзахист	02.06.2025 – 06.06.2025	Виконано
10	Захист	16.06.2025 – 27.06.2025	

Студент

(підпис)

Антон АНДРІЙЧУК

(ім'я, прізвище)

Керівник роботи

(підпис)

Олег БАРАБАШ

(ім'я, прізвище)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Робота містить 54 сторінки, 24 рисунки, 2 додатки та 6 посилань.

Метою роботи є розробка програмного застосунку для побудови оптимальних схем комутації різнорідних сонячних панелей із використанням архітектурних засобів фреймворку Qt. Основну увагу зосереджено на проєктуванні вигідної математичної моделі, оптимального алгоритму автоматичного формування дерева комутації, оптимізованого евристичними прийомами; зручного графічного інтерфейсу, а також забезпеченні коректної взаємодії між проєктними компонентами за допомогою можливостей обраного інструментарію.

Практичне значення одержаних результатів полягає в тому, що створений інструмент дозволяє швидко й ефективно формувати життєздатні електротехнічні схеми з сонячних панелей різного типу, враховуючи технічні параметри напруги, струму та потужності. Такий підхід може бути використаний як у навчальних середовищах для вивчення роботи сонячних систем, так і в прикладних задачах проєктування малої енергетики. Гнучка архітектура застосунку дозволяє розширювати його функціональність або інтегрувати з іншими проєктами.

Ключові слова: сонячна панель, інвертор, різнорідний, Qt, графічний інтерфейс, евристичний алгоритм, комутаційна схема, дерево з'єднань, сигнал, слот, віджет.

ABSTRACT

Structure and scope of the thesis. The work contains 54 pages, 24 figures, 2 appendices and 6 references.

The purpose of the work is to develop a software application for constructing optimal connection schemes for heterogeneous solar panels using the architectural tools of the Qt framework. The main focus was on designing a favorable mathematical model, an optimal algorithm for automatic connection tree formation, optimized by heuristic techniques; a user-friendly graphical interface, as well as ensuring correct interaction between project components using the capabilities of the chosen tools.

The practical value of the obtained results lies in the fact that the created tool allows one to quickly and efficiently form viable electrical circuits from solar panels of various types, taking into account the technical parameters of voltage, current and power. This approach can be used both in educational environments for studying the operation of solar systems and in applied tasks for designing small-scale energy systems. The flexible architecture of the application allows its functionality to be expanded or integrated with other projects.

Keywords: solar panel, inverter, heterogeneous, Qt, graphical interface, heuristic algorithm, connection circuit, connection tree, signal, slot, widget.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ.....	7
ВСТУП.....	8
1 ПОСТАНОВКА ЗАДАЧІ.....	10
1.1 Дослідження предметної області	10
1.2 Формування вимог до програмної системи	15
1.3 Обрані методи та інструменти	18
Висновки до розділу 1	20
2 РОЗРОБКА МАТЕМАТИЧНОЇ МОДЕЛІ.....	21
2.1 Вибір та впровадження структури даних.....	21
2.2 Алгоритм побудови дерева з'єднань.....	23
Висновки до розділу 2	25
3 РОЗРОБКА ГРАФІЧНОГО ІНТЕРФЕЙСУ.....	26
3.1 Дизайн віджетів.....	26
3.1.1 Основи розробки віджетів засобами Qt.....	26
3.1.2 Головне вікно застосунку	28
3.1.3 Плаваючі вікна Dock Widgets.....	30
3.1.4 Віджет інформації про об'єкт.....	31
3.1.5 Віджет інвентаря елементів	32
3.1.6 Діалогові вікна.....	33
3.2 Сигнали і слоти	35
3.3 Програмна реалізація віджетів.....	37
3.3.1 MainWindow – головне вікно застосунку	37
3.3.2 Плаваюче вікно DataWidget.....	39
3.3.3 Плаваюче вікно InventoryWidget	41
3.3.4 Діалогові вікна.....	43
Висновки до розділу 3	45
4 ТЕСТУВАННЯ ТА ВАЛІДАЦІЯ.....	47
4.1 Тестування розробленої системи	47
4.2 Користувачський огляд.....	48
Висновки до розділу 4	51
ВИСНОВКИ.....	52
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	54
ДОДАТОК А Лістинг розробленої системи.....	55
ДОДАТОК Б Презентація.....	68

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

UI	Інтерфейс користувача
UX	Враження користувача
API	Application Programming Interface – інтерфейс, який дозволяє різним програмним додаткам/модулям взаємодіяти між собою
Віджет	Окремий елемент графічного інтерфейсу
Фреймворк	Структурований набір компонентів, бібліотек та інших інструментів, які допомагають створювати програмні продукти
СЕС	Сонячна електростанція
MPP	Maximum power point – точка максимальної потужності
Фотовольтаїчний	Такий, що перетворює світло в електрику

ВСТУП

У сучасних умовах стрімкого зростання енергетичних потреб, виснаження традиційних джерел енергії та посилення глобальних викликів у сфері екологічної безпеки, все більшої актуальності набуває впровадження відновлюваних джерел енергії. Енергетична трансформація до екологічно чистих технологій є не лише загальносвітовим трендом, а й нагальною потребою для забезпечення сталого розвитку суспільства. Одним із найперспективніших напрямів у цій сфері є сонячна енергетика – галузь, що базується на використанні сонячного випромінювання як джерела для генерації електричної енергії.

Сонячні електростанції (СЕС), які перетворюють енергію сонця на електричну за допомогою фотоелектричних елементів, вирізняються низкою переваг: екологічністю, відсутністю шкідливих викидів, можливістю автономного використання, простотою масштабування та гнучкістю в плануванні. СЕС можуть бути встановлені як на промислових об'єктах, так і в особистій власності, що робить їх універсальним рішенням для енергозабезпечення. У контексті зростання тарифів на електроенергію та зацікавленості громадян у зниженні залежності від централізованих джерел, попит на сонячну енергетику зростає і в Україні, і в світі загалом.

Проте, попри очевидні переваги, проєктування та експлуатація СЕС супроводжуються низкою технічних викликів. Однією з найважливіших задач є підбір оптимального поєднання сонячних панелей з урахуванням їхніх електричних характеристик, таких як потужність, напруга та струм у точці максимальної потужності (МРР), струм короткого замикання, напруга відкритого кола тощо. Невдале компонування модулів може призвести до зниження ККД загальної системи, перевантаження інверторів, порушення температурного режиму обладнання, коротких замикань, перегріву кабелів і навіть пожеж. Це вказує на необхідність володіти інструментами, які дозволяють точно і зручно моделювати, аналізувати та перевіряти різні схеми підключення ще на етапі проєктування СЕС.

Ручний підрахунок та проєктування великих або складних систем є трудомістким, схильним до помилок процесом, що вимагає значного вихідного досвіду та інвестицій, грошових і часових. Тому автоматизація цього процесу є важливим кроком до підвищення ефективності та безпеки проєктування СЕС майбутнього. Створення інтуїтивно зрозумілої програмної системи для моделювання та візуалізації роботи сонячної електростанції дозволяє забезпечити точні розрахунки, виявляти помилки на ранніх етапах, зменшити час на аналіз варіантів і, зрештою, обрати найефективнішу конфігурацію обладнання.

Розробка подібного інструменту є актуальною для інженерів-проєктувальників, монтажників та науковців, що займаються дослідженням ефективності відновлюваних джерел енергії, а також для ентузіастів, які прагнуть встановити сонячну станцію у власному домогосподарстві. Доступність такого програмного забезпечення сприятиме популяризації сонячної енергетики, зменшенню вхідного порогу для впровадження нових СЕС і забезпеченню сталого розвитку енергетичної сфери.

Мета даної роботи полягає у створенні програмного застосунку для моделювання роботи сонячних електростанцій з акцентом на інтуїтивно-зрозумілий інтерфейс, візуалізацію результатів у реальному часі та автоматичний аналіз параметрів з'єднань. Результатом дослідження і виконаної роботи стане ефективний інструмент, що дозволяє зменшити кількість помилок при проєктуванні, підвищити якість технічних рішень і зробити процес створення СЕС доступнішим для широкого кола користувачів.

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Дослідження предметної області

Сонячні елементи безпосередньо перетворюють сонячну енергію в електричну за допомогою фотоелектричного ефекту. У напівпровідниковій структурі фотовольтаїчних модулів фотони сонячного світла збуджують електрони, які переходять у зону провідності р-п переходу, створюючи електричний струм, як показано на схемі на рисунку 1.1.^[3]

Inside a photovoltaic cell

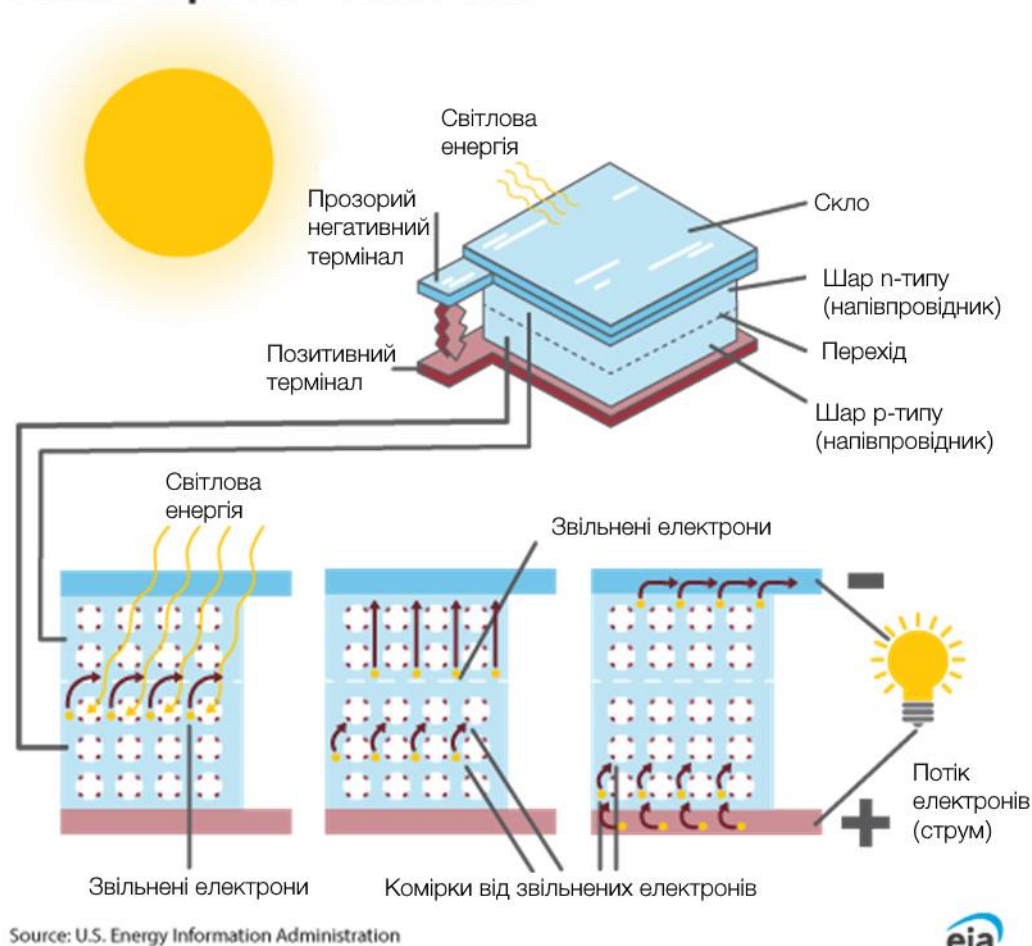
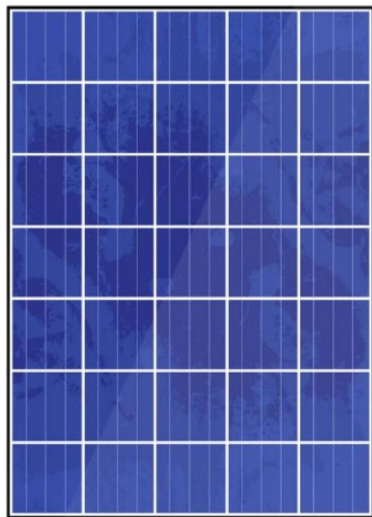


Рисунок 1.1 – Принцип роботи фотовольтаїчної комірки

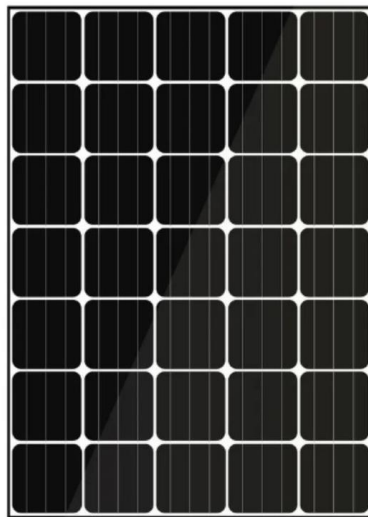
Сформований постійний струм збирається контактами на поверхні панелі і, за потреби, перетворюється інвертором в змінний для живлення електромережі чи акумуляторів. Сучасні інвертори з функцією максимальної точки потужності

автоматично регулюють напругу на вході, щоб підтримувати роботу панелей у їхній максимальній потужності.^[3]

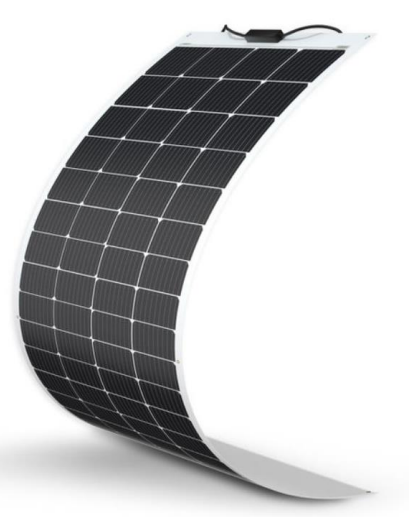
На ринку використовують три основні типи PV-модулів: монокристалічні, полікристалічні та тонкоплівкові. Монокристалічні панелі виготовляють із єдиного кристалу кремнію. Вони мають найвищу ефективність (приблизно 17–22 %), вони компактні та чорного кольору, але дорожчі. Полікристалічні панелі складаються з багатокристалічного кремнію (охоłodжений розплав, що містить багато кристалів). Вони мають середню ефективність (15–17 %) і нижчу ціну. Тонкоплівкові панелі (наприклад, з аморфного кремнію a-Si, CIGS, CdTe) наносять тонкою плівкою на скляну чи іншу основу. Вони значно легші й гнучкі, але мають найнижчий ККД (10–13 %). Загалом, монокристалічні панелі забезпечують найкращу продуктивність і компактність, полікристалічні – оптимальні за вартістю до енергоефективності, а тонкоплівкові використовують там, де дозволені великі площі під панелі чи потрібні гнучкі рішення (наприклад, у застосуваннях, інтегрованих в архітектуру будівель).^[4] На рисунку 1.2 зображено зовнішній вигляд трьох типів сонячних панелей.



Полікристалічний



Монокристалічний



Тонкоплівковий

Рисунок 1.2 – Типи сонячних панелей

Стрімкому розвитку будь-якої індустрії зазвичай характерна широка низка абсолютно різних стандартів, методологій та ідей. Це не оминуло і сферу сонячної

енергетики – на будь-якому публічному ринку можна знайти абсолютно різні сонячні елементи за типами, розмірами, характеристиками, цінами, як видно із рисунку 1.3. Це створює певний головний біль для новачків та для тих, хто не має можливості вільно вибирати яких і скільки фотоелементів придбати і встановити для власних потреб – чи то з фінансових причин, чи із потреби обходитися тим, що є в наявності.

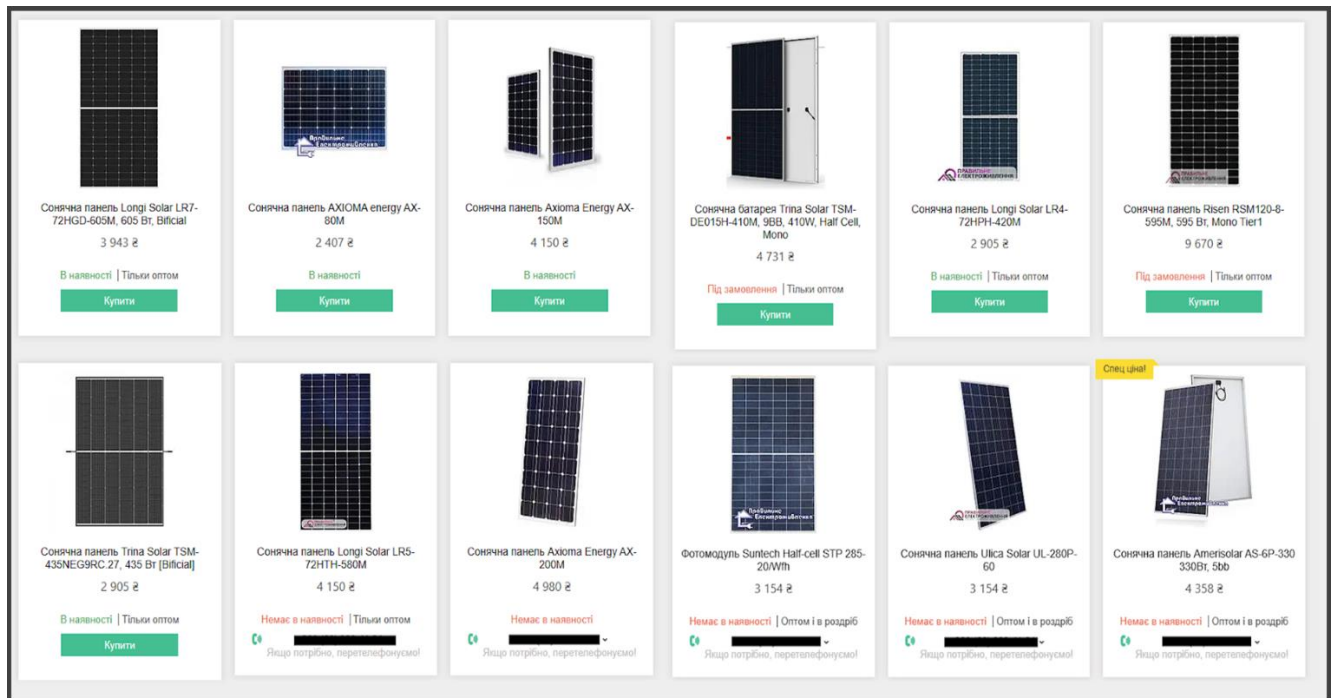


Рисунок 1.3 – Типовий торгівельний майданчик сонячних панелей

Для з'єднання сонячних панелей існує три способи: послідовний, паралельний та змішаний. При послідовному з'єднанні плюсовий контакт кожної наступної панелі з'єднується з мінусовим попередньої. У такій схемі струм у всіх панелях однаковий, а напруги сумуються. Перевагою є те, що зростає загальна напруга системи. Це дозволяє вигідно працювати на великих відстанях без значних втрат. Недоліком є чутливість до затінення чи несправності: при послідовному з'єднанні одна «слабка» панель обмежує струм всього ланцюга, а при повному «обриві» її ланцюг вимикається.

За паралельного з'єднання плюсові контакти всіх панелей з'єднуються разом, і мінусові разом. Напруга на виході залишається такою ж, як у одній панелі, а загальний струм – сумою струмів всіх панелей. Це означає, що потужності панелей

додаються без збільшення напруги. Наприклад, дві ідентичні панелі 100 Вт на 18.9 В при паралельному з'єднанні дадуть загальну силу струму 10.58 А та потужність 200 Вт. Перевага паралельного з'єднання це відносна стійкість: падіння потужності чи вихід з ладу однієї панелі вплине на загальну потужність, але не вплине на інші паралельні панелі, вони продовжать працювати. Недоліком є велике значення сили струму, що протікає, тому необхідні більш потужні кабелі і встановлення запобіжників. При потужностях на рівні кіловатів струм може перевищувати десятки ампер, що вимагає значних перерізів проводу за товщиною та розгалужувачів для підключення до енергосистеми.

Змішане з'єднання застосовується для більших систем, де необхідно одночасно збільшити і напругу, і струм. Панелі групуються в послідовні ряди, які потім з'єднуються паралельно. Для окремо взятих паралельних та послідовних з'єднань діють попередньо описані принципи. Наприклад, з'єднання двох рядів панелей по чотири панелі в кожній матиме вдвічі більшу силу струму і в чотири рази більшу напругу, ніж одна сонячна панель. Змішане з'єднання дає перевагу в гнучкості конструкції: дозволяє підвести параметри масиву у відповідність до бажаних, або до характеристик контролера/інвертора (не перевищувати їх максимальні вхідні напругу та струм).

Змішування панелей різних моделей чи потужностей у ланцюгах загалом не рекомендується. Якщо різні панелі мають відмінні характеристики, то виникає невідповідність. У послідовному з'єднанні це означає, що струм всього ланцюга обмежується найменшим значенням будь-якої панелі. У паралельному ж ланцюгу, навпаки, напруга вирівнюється за найнижчим значенням серед панелей. Принцип поєднання різнорідних сонячних панелей зображено на схемі на рисунку 1.4.

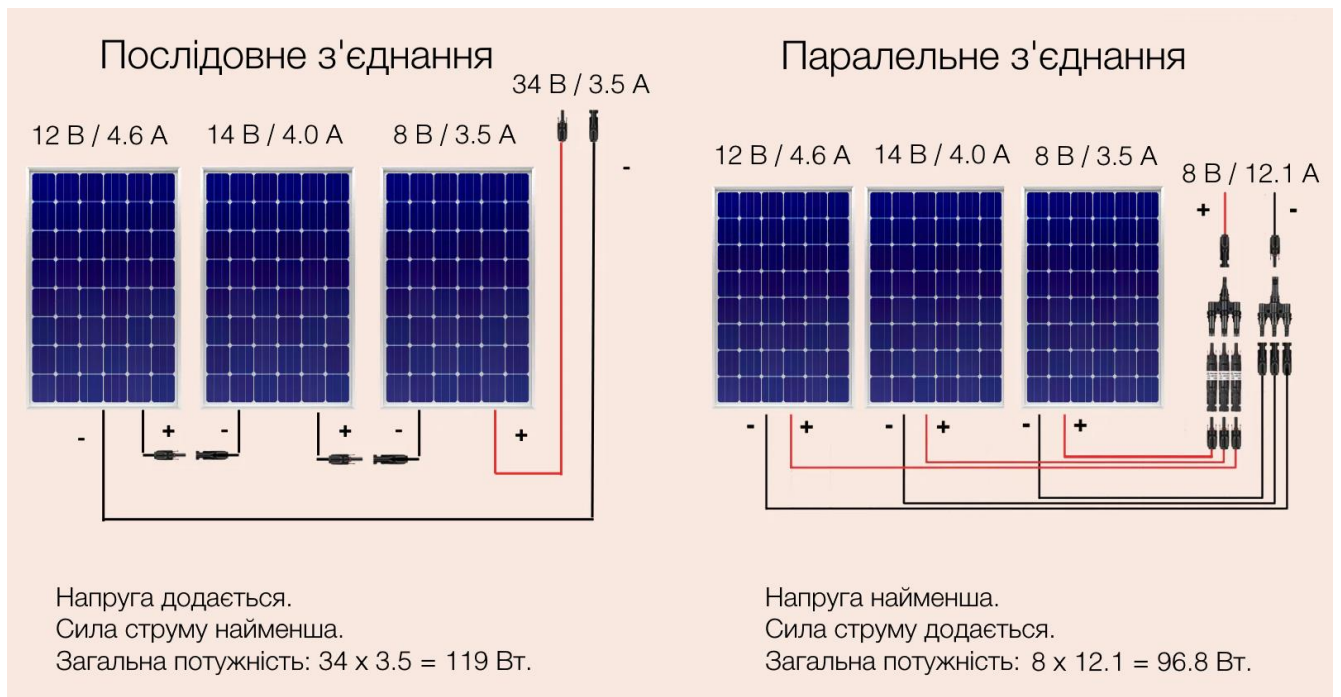


Рисунок 1.4 – Правила поєднання різномірних сонячних панелей.

Простими словами, система вироблятиме набагато менше електроенергії, ніж її окремі елементи могли до з'єднання. Комбінуючи фотоелементи в різний спосіб, можна досягти різних значень загальної потужності. Тому при проектуванні систем із невідповідністю слід намагатися використовувати в одному рядку панелі з однаковим чи дуже близьким параметром сили струму, а в паралельних групах – однаковим чи близьким параметром напруги. Метою цієї роботи є пошук такого оптимального рішення та презентація його користувачеві в наочному вигляді.

В ідеальних умовах потужність сонячної електростанції дорівнює сумі потужностей всіх сонячних панелей, але на практиці справжня потужність буде залежати від низки факторів. Серед таких факторів є втрати при передачі струму, перетворенні, затіненні, невідповідності панелей одна одній. Також слід зазначити зміну ККД перетворення сонячної енергії на електричну в залежності від температури. Номінальні параметри сонячних панелей задаються за стандартних умов та в точці максимальної потужності. Для температури, зокрема, це 25 °С. Враховуючи всі фактори, окрім невідповідності, загальне ККД системи зазвичай на 5-15% менше від теоретичного. Невідповідність сонячних панелей може знизити його набагато сильніше.^[5] Фізичні параметри, такі як кут падіння сонячних

променів, розміщення сонячних панелей, затінення, температура тощо, не розглядатимуться в межах даної роботи.

В цьому підрозділі проаналізували фізичні явища, що супроводжують предметну область, а саме принципи роботи сонячних панелей, а також правила їх поєднання, зокрема поєднання різних за характеристиками сонячних панелей.

1.2 Формування вимог до програмної системи

У процесі розробки будь-якого прикладного програмного забезпечення одним із найважливіших етапів є формування чітких і послідовних вимог до функціоналу, логіки роботи, інтерфейсу користувача та технічної реалізації. У рамках цього проєкту мета полягає у створенні програмного застосунку для моделювання сонячних електростанцій, зокрема конфігурацій сонячних панелей та їх підключення до інвертора. Застосунок повинен стати ефективним інструментом для інженерів-проєктувальників, дослідників, студентів та ентузіастів у галузі відновлюваної енергетики. Формування вимог до програмної системи поділяється на кілька категорій: функціональні вимоги, вимоги до інтерфейсу, алгоритмічні вимоги, а також вимоги до сумісності та розширюваності.

Функціональність системи охоплює повний цикл роботи користувача з конфігурацією сонячної електростанції – від введення даних до отримання результатів аналізу. Було визначено такі основні функціональні вимоги:

- Введення та редагування даних про сонячні панелі.
- Побудова схеми з'єднання.
- Автоматичний аналіз конфігурації.
- Візуалізація конфігурації з'єднань.

Користувач має змогу вручну додавати технічні параметри панелей або імпортувати їх з попередньо підготовлених файлів. Параметри включають потужність, напругу і струм в точці максимальної потужності, напругу холостого ходу тощо. Система повинна мати можливості для об'єднання панелей у послідовні

та паралельні групи, з формуванням ієрархічної структури. В основі схеми лежить структура дерева або орієнтованого графа, що відображає зв'язки між елементами.

Після формування схеми система має провести автоматичні обчислення напруги, струму та потужності, перевіряти відповідність допустимим значенням та виявляти потенційні проблеми. Система повинна відображати побудовану конфігурацію у графічному вигляді з можливістю перегляду параметрів кожного вузла.

Інтерфейс системи повинен бути зручним, інтуїтивним та адаптованим для користувачів з різним рівнем підготовки. Програма має автоматично виявляти критичні помилки конфігурації, інформувати користувача про небезпечні ситуації або невідповідності, та пропонувати варіанти виправлення.

Основні визначені вимоги до інтерфейсу включають:

- Чітку логічну структуру з відображенням усіх етапів роботи – від введення даних до візуалізації результатів. Повинна бути реалізована панель, що показує загальні параметри всієї системи, а також повідомляє користувача про порушення обмежень чи некоректні конфігурації.
- Доступність усіх основних дій через зрозумілі кнопки, панелі, контекстні меню.
- Можливість швидкого додавання, видалення, переміщення елементів у схемі.
- Адаптивність для екранів різного розміру.

Слід реалізувати алгоритм автоматичної побудови з'єднань на основі заданих вхідних параметрів. Алгоритм має враховувати не лише обмеження інвертора, а й прагнути до досягнення певних критеріїв оптимальності: максимальна сукупна потужність, мінімальні втрати потужності, збалансоване навантаження. До алгоритму автоматичної побудови було сформовано наступні вимоги:

- Евристичні підходи та частковий перебір. Повний перебір недоцільно використовувати для великих систем через високу часову складність.

- Масштабованість розрахунків. Алгоритм повинен працювати коректно та швидко навіть при значній кількості панелей.

Для цього доцільно кешувати результати, спиратися на структури даних та уникати надлишкових перерахунків.

Застосунок повинен бути реалізований у вигляді десктопної програми, що забезпечує максимальну стабільність та зручність при роботі з великими схемами. Основні технічні вимоги:

- Підтримка популярних ОС: Windows, Linux та MacOS.
- Мінімальні вимоги до апаратного забезпечення – можливість запуску на ПК середнього рівня продуктивності.
- Відсутність залежності від сторонніх серверів або підключення до Інтернету під час роботи.
- Простота встановлення (без необхідності окремого встановлення додаткового ПЗ).

Обрання десктопної архітектури пояснюється складністю і незручністю візуального моделювання та аналізу, що неефективно реалізується на мобільних пристроях або в браузерях. Також це дозволяє краще контролювати ресурси системи та забезпечити стабільність.

Програмна архітектура має бути побудована з урахуванням можливості подальшого розвитку. Зокрема, важливо забезпечити модульність та інкапсуляцію – розділення на незалежні блоки, що спрощує тестування, обслуговування та розширення функціональності. Також важливо закласти можливість інтеграції з іншими інструментами проєктування або базами даних компонентів у майбутньому. Поставлені вимоги до програмної системи було викладено на діаграмі прецедентів, що зображена на рисунку 1.5.

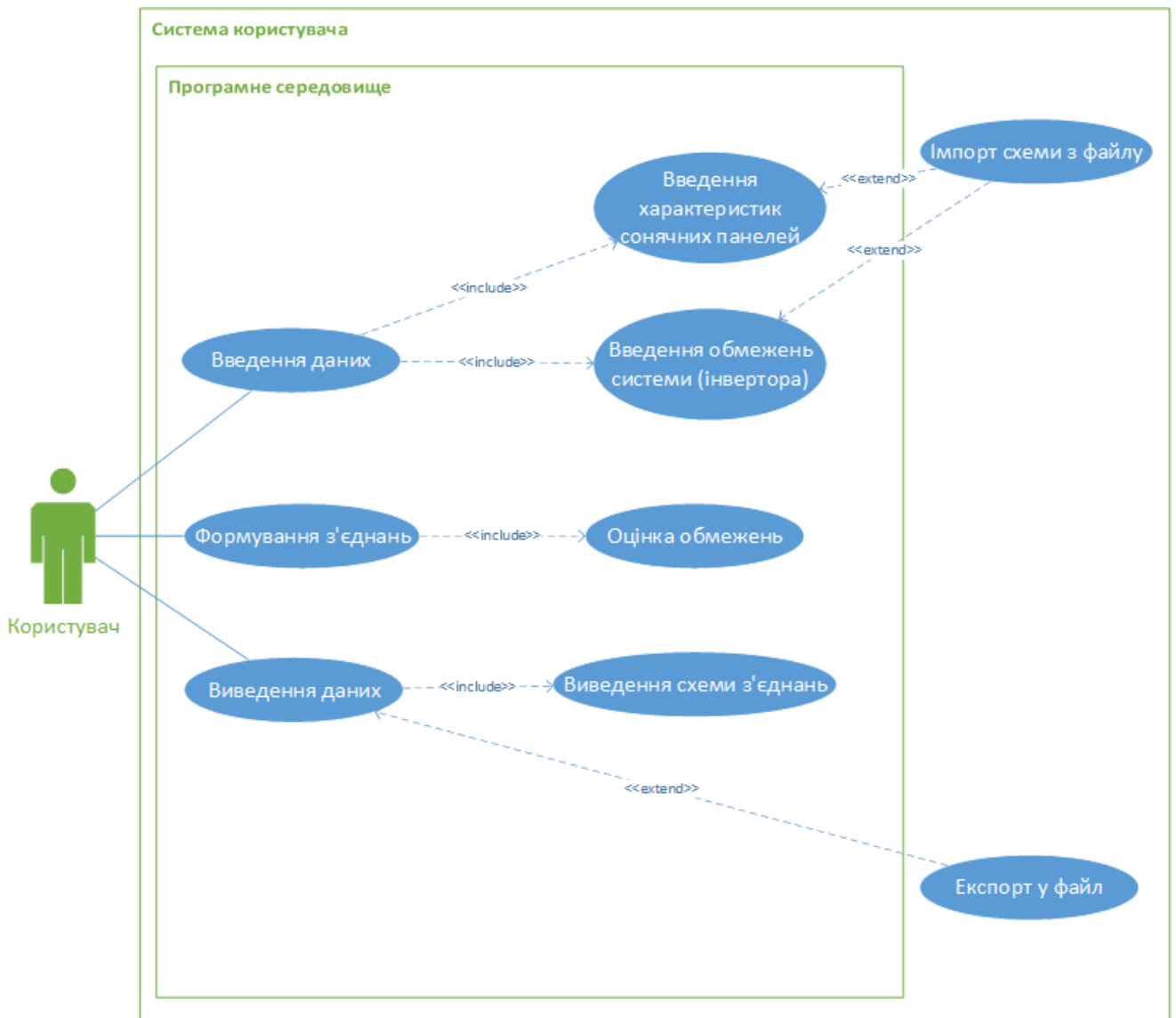


Рисунок 1.5 – Діаграма прецедентів програмної системи

У цьому підрозділі було сформульовано вимоги до програмної системи, які охоплюють усі аспекти, необхідні для забезпечення функціональності, зручності, ефективності й надійності роботи застосунку.

1.3 Обрані методи та інструменти

Враховуючи попередньо описані вимоги до програмного продукту, зокрема сучасність, інтуїтивність інтерфейсу та кросплатформність, було прийнято рішення використати для розробки графічний фреймворк користувацьких інтерфейсів Qt. Фреймворк представлений для мов програмування C++ та Python. Розробка велась

мовою Python, для якої фреймворк Qt підтримується двома проєктами – PySide та PyQt. Обидва проєкти є подібними за продуктивністю, синтаксисом, набором інструментів та робочим процесом розробки. Однак перевагу було віддано PySide через те, що він офіційно підтримується розробниками Qt Company, коли PyQt розробляється та підтримується спільнотою. Остання версія фреймворку мовою Python на момент розробки даного продукту – PySide 6.9.0.

Qt також пропонує зручний дизайнер віджетів Qt Designer, що дозволяє наочно та швидко реалізовувати попередній дизайн графічних віджетів. Збережені файли дизайну потім можна скомпілювати в програмний код мовою Python за допомогою інструменту Qt User Interface Compiler (UIC). Скомпільований програмний код дизайнів можна надалі використовувати в якості основи для розробки віджетів інтерфейсу програмного продукту.

Також Qt пропонує компілятор ресурсів Qt Resource Compiler (RCC), що дозволяє інтегрувати графічні ресурси, що використовуються застосунком, в якості модуля програми. Це робить застосунок більш компактним та кросплатформним.

Проєкт можна за потреби перекомпілювати під операційні системи Windows, Linux, MacOS не вносячи жодних змін, оскільки фреймворк Qt є кросплатформним в основі. Також за потреби можна перекомпілювати проєкт і під мобільні операційні системи, але для цього рекомендується явно реалізувати в модулях продукту підтримку та інтерпретацію сенсорної взаємодії та жестів, щоб запобігти можливій небажаній поведінці графічного користувацького інтерфейсу.

Для створення і зображення дерева було обрано бібліотеки `bigtree` та `graphviz`. Вони містять зручний функціонал для побудови, обходу та зображення дерева графів.

В цьому підрозділі було описано обраний інструментарій для реалізації поставлених завдань, були аргументовані причини та фактори, що вплинули на вибір саме цих інструментів для розробки.

Висновки до розділу 1

У першому розділі було охоплено три ключові теми: фізичні основи предметної області, вимоги до функціональності та реалізації системи, а також обраний технічний інструментарій. Було досліджено предметну область, сформульовано науково-технічне підґрунтя процесів, що пов'язані із роботою та використанням сонячних панелей, зокрема панелей з різними характеристиками. Також було сформовано повний перелік вимог до системи. На основі сформованих вимог було аргументовано вибір засобів реалізації програмного продукту. Обрані інструменти відповідають вимогам проєкту й дозволяють реалізувати як основну, так і допоміжну функціональність.

Сукупність проведених досліджень і сформульованих положень створює міцну концептуальну й інженерну основу для наступних етапів проєкту, що стосуються безпосередньої розробки програмного застосунку. Завдяки чіткості вимог, зваженості архітектурних рішень і врахуванню специфіки предметної області очікується створення ефективного, гнучкого й корисного інструменту для проєктування сучасних систем сонячної енергетики.

2 РОЗРОБКА МАТЕМАТИЧНОЇ МОДЕЛІ

2.1 Вибір та впровадження структури даних

Успішне виконання поставлених завдань та вимог зумовлює розробку структури даних, яка б обслуговувала дані про сонячні панелі, електричний інвертор та процес їх створення при виконанні застосунку. Також необхідно впровадити підхід до побудови та виведення схеми з'єднань.

Основним принципом моделювання схеми з'єднань стала деревовидна структура, в якій компоненти системи репрезентуються у вигляді вузлів дерева. Інвертор буде коренем дерева, а панелі – гілками. Таке представлення дозволяє природно відтворити каскадну природу електричних з'єднань та забезпечити гнучкість у зміні топології під час пошуку оптимальних рішень.

Вузли дерева реалізовані у вигляді класів `SolarPanelNode` та `InverterNode`, які успадковують функціональність від класу `Node` бібліотеки `bigtree`, що спеціалізується на управлінні деревовидними структурами. У той же час ці класи успадковують відповідні фізичні характеристики компонентів від класів `SolarPanel` та `Inverter`. Ці класи було інкапсульовано для зручності, вони лише необхідні поля для відповідних необхідних технічних характеристик. Для сонячних панелей це максимальна потужність, напруга та струм у точці максимальної потужності, а також напруга холостого ходу. Для інверторів, у свою чергу, задаються максимальна вхідна потужність, мінімальна та максимальна вхідна напруги, граничне значення струму та параметр автономності. Параметр автономності вказує, чи під'єднана система до більшої електричної мережі, чи є повністю відокремленою.

Щоб гарантувати унікальність кожного вузла в моделі та забезпечити коректне зіставлення між візуальним інтерфейсом користувача та внутрішніми даними, було реалізовано клас `NodeFactory`. Він відповідає за централізоване створення відповідних об'єктів та автоматичне призначення унікального числового

ідентифікатора (ui_id) кожному новоствореному вузлу. Це досягається завдяки глобальному відслідковуванню останнього використаного ідентифікатора. Кожен новий об'єкт просто отримує наступне число в якості ідентифікатора. Це дозволяє однозначно ідентифікувати кожен елемент у моделі та підтримувати узгодженість даних між числовим і графічним рівнями застосунку. Автоматичність даного процесу дозволяє звільнити користувача від клопоту керування назвами елементів у застосунку і зосередити його увагу лише на введенні технічних характеристик. На рисунку 2.1 зображено діаграму всіх класів, що згадуються в цьому підрозділі, із полями та методами, важливими для програмної системи.

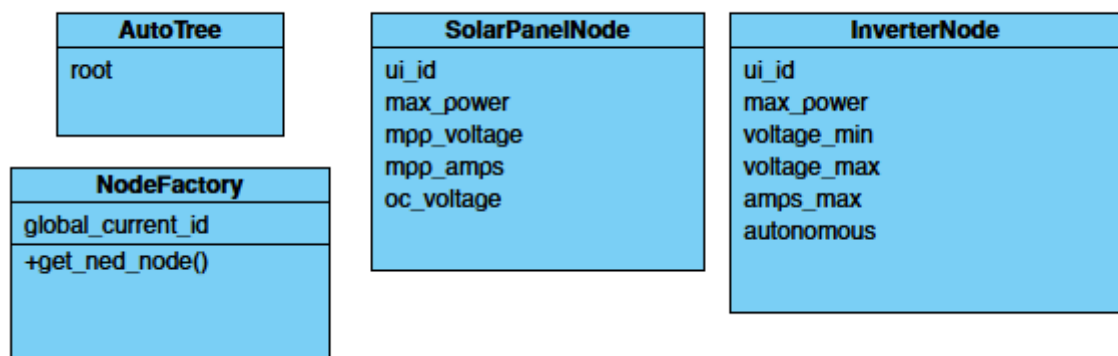


Рисунок 2.1 – Діаграма класів, що складають модель даних

Володіння фабрикою нод буде логічно передати тому віджету інтерфейсу, що відповідає за відображення всіх елементів системи, подібно до інвентарю елементів. Отже, на цей віджет буде покладено відповідальність за весь життєвий цикл елементів як математичної, так і візуальної систем, і, що не менш важливо, узгодженість між цими системами.

Деревовидна структура в бібліотеці bigtree визначається лише за коренем. Для зручності було створено клас-обгортку AutoTree, який містить вказівку root на об'єкт кореневого вузла типу InverterNode. Усі сонячні панелі, підключені до цього інвертора, є дочірніми вузлами в дереві. Оскільки InverterNode та SolarPanelNode успадковують поведінку Node, вони містять поле зі списком вказівників на власні дочірні об'єкти, якщо такі є. Кожній ноді також відомий свій батьківський елемент. Очевидно, що дочірність одного елемента іншому вказує на послідовний характер

з'єднання, за виключенням електричного інвертора. Тобто інвертор та сонячні панелі можуть мати довільну кількість дочірніх елементів, а такі елементи, що є дочірніми на одному рівні до одного і того самого елемента, вважаються під'єднаними паралельно відносно один одного.

Отже, в цьому підрозділі обґрунтували обрані патерни моделювання даних програмної системи, що забезпечують можливість формування як простих, так і складних схем із довільною структурною шириною, глибиною та кількістю елементів.

2.2 Алгоритм побудови дерева з'єднань

Щоб реалізувати задачу автоматичного структурування різнорідних сонячних панелей у межах допустимих характеристик енергосистеми було розроблено алгоритмічну модель, яка забезпечує ітеративне формування схеми електричних з'єднань. Очевидно, що повний перебір усіх можливих варіантів та вибір найкращого займатиме занадто довгий період часу при збільшених кількостях елементів, тому перевагу було віддано пошуку результату, наближеного до найкращого. Також було впроваджено кілька евристичних скорочень для оптимізації алгоритму.

Центральним об'єктом цієї моделі є клас `AutoTreeBuilder`, який виконує побудову дерева підключень шляхом групування панелей у послідовні стрічки та паралельні масиви, враховуючи обмеження інвертора. Допоміжну роль складають дві структури даних: `SeriesString` як представлення послідовного з'єднання панелей, та `ParallelArray` як сукупність таких стрічок, з'єднаних паралельно. Клас `SeriesString` є представленням ряду сонячних панелей, з'єднаних послідовно. Кожна послідовна стрічка описується списком панелей, для яких розраховуються загальні електричні характеристики: загальна напруга в точці максимальної потужності, напруга холостого ходу, обмежуючий струм (мінімальний струм серед усіх панелей) та ефективна потужність стрічки панелей. Розрахунок цих значень та

їх збереження важливо для швидкої оцінки допустимості або ефективності стрічки в межах системи під час процесу побудови та оцінки. Діаграми класу AutoTreeBuilder, допоміжних структур даних SeriesString та ParallelArray із доречними для завдання полями та методами, зображено на рисунку 2.2.

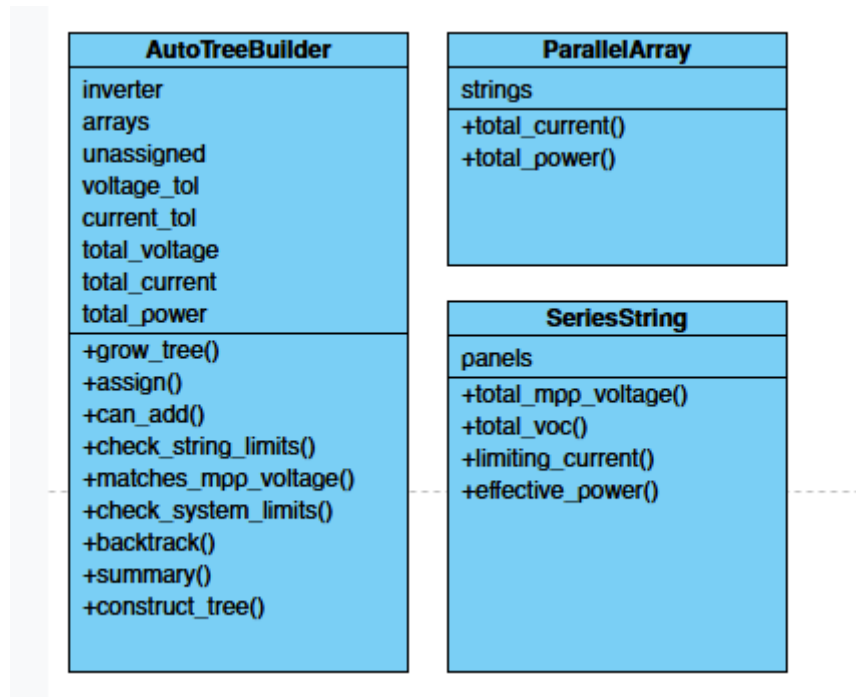


Рисунок 2.2 – Класи AutoTreeBuilder, ParallelArray, SeriesString

Процес побудови виконується методом `grow_tree()`, який проходить усі невикористані панелі по черзі, намагаючись додати їх до вже існуючих стрічок, де б вони пасували найкраще, або створити нові стрічки й масиви. Рішення про додання стрічки приймається з урахуванням параметру максимального відхилення струмів `current_tol`, що за замовчуванням встановлено на 1%, та враховуючи, що додання панелі до стрічки не призведе до перевищення можливостей інвертора. Якщо панель не може бути ефективно інтегрована у поточну структуру, вона відкладається. Пізніше відбувається процес повернення (`backtracking`), під час якого схема перебудовується із сортуванням панелей за спаданням потужності – це дозволяє збалансувати попереднє рішення з урахуванням сильніших панелей на початку.

Загальні обмеження системи, які включають перевищення максимальної потужності або струму інвертора, перевіряються на кожному кроці методом

`check_system_limits()`. Також слід відмітити важливий метод оцінки сумісності стрічок у межах одного паралельного масиву. Метод `matches_mpp_voltage()` забезпечує, що нова стрічка матиме середню напругу в межах заданого відхилення `voltage_tol` від середньої напруги інших стрічок у паралельному масиві. Це забезпечує більшу узгодженість умов для оптимального паралельного підключення.

У разі успішної побудови структур даних, схема формалізується у вигляді дерева інструментами бібліотеки `bigtree` з використанням операторів з'єднання між вузлами типу `SolarPanelNode` та `InverterNode`, де перша панель у кожній стрічці безпосередньо підключається до інвертора, а наступні панелі в стрічці з'єднуються між собою послідовно. Бібліотека `bigtree` рекомендує використання операторів бітового зсуву (`>>` та `<<`) як один із способів з'єднання вузлів дерева.

В цьому підрозділі описано роботу алгоритму автоматичної побудови оптимальної схеми з'єднань різнорідних сонячних панелей. Його можна вважати евристичним, оскільки в процесі перебору та оцінки враховується загальна логіка системи, але без гарантії знаходження глобального оптимуму – натомість досягається практичний, хоч і субоптимальний результат.

Висновки до розділу 2

У другому розділі було обґрунтовано обрану модель даних, яка надає можливості створення як простих, так і складних конфігурацій сонячних електростанцій довільної ширини та глибини. Описано роботу розробленого евристичного алгоритму, що формує часткові конструкції з урахуванням обмежень системи та поєднує їх у сонячну електростанцію, знаходячи наближений до найкращого результат.

3 РОЗРОБКА ГРАФІЧНОГО ІНТЕРФЕЙСУ

3.1 Дизайн віджетів

3.1.1 Основи розробки віджетів засобами Qt

Віджети є основними елементами для створення користувацьких інтерфейсів у Qt. Вони можуть відображати дані та інформацію про стан, отримувати вхідні дані від користувача та слугувати контейнером для інших віджетів, які слід згрупувати разом. Віджет, який не вбудований у батьківський віджет, називається вікном.^[6]

Як було зазначено в підрозділі 1.3, для розробки попереднього вигляду віджетів графічного інтерфейсу, фреймворк Qt пропонує дизайнер віджетів Qt Designer. Робота із Qt Designer є дуже схожою на роботу із Windows Forms для .NET у середовищі розробки Visual Studio від Microsoft. Вигляд головного вікна інструменту Qt Designer зображено на рисунку 3.1.

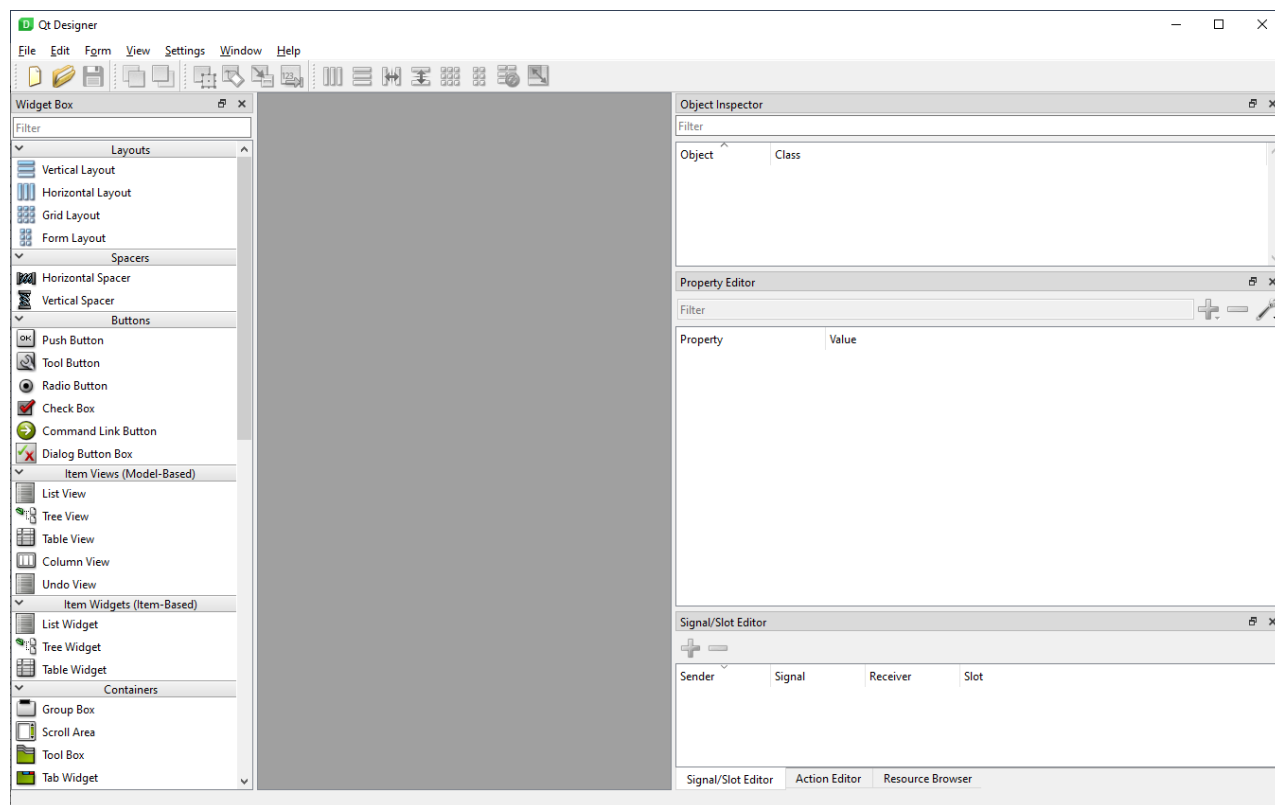


Рисунок 3.1 – Вікно інструменту Qt Designer

Однак значною відмінністю є те, що основну поведінку віджетів не можна задавати прямо безпосередньо під час дизайну. Щоб реалізовувати поведінку віджетів, збережені файли дизайну із розширенням «.ui» необхідно скомпілювати в програмний код Python. Інструмент компілює дизайни у код класів на основі базових Q-класів віджетів, що є у фреймворку Qt. Формується конструктор класу, що програмно ініціалізує попередньо розроблений дизайн, зі всіма вбудованими у віджет дочірніми віджетами та їх параметрами. Класи віджетів, що безпосередньо використовуються в програмному коді, успадковують та ініціалізують дизайн-класи, додаючи поведінку та взаємодію між віджетами.

Отже, створення дизайну за допомогою фреймворку Qt полягає у створенні та використанні класів графічних віджетів та впровадженні коректної взаємодії між ними. Класи графічних віджетів не є окремими одиницями, вони успадковують поля та методи певних базових класів віджетів, що описані у фреймворку. Взаємодія між віджетами зазвичай відбувається через систему сигналів та слотів. Про систему сигналів та слотів буде наведено в наступних підрозділах.

У директорії даної роботи класи, що відповідають за графічний інтерфейс зберігаються у папці «ui». Там безпосередньо розміщені «.ui» файли дизайну, а також файли скомпільованого програмного коду цих дизайнів. За шляхом «ui/widgets» містяться класи безпосередніх віджетів, розроблених на основі цих дизайнів. За шляхом «ui/res» тимчасово містяться графічні ресурси, які використовує застосунок. Вони не входять до структури проєкту, оскільки вони вже були категоризовані у файлі resources.qrc в папці «ui» та скомпільовані в об'єкт ресурсів resources.rc. Для класів, що відповідають за обробку даних, тобто математичну модель програмної системи, відведено директорію «data». Також було створено директорію для зберігання модульних тестів, але вона не включена в структуру проєкту, і не є необхідною для виконання застосунку.

3.1.2 Головне вікно застосунку

Для графічного інтерфейсу користувача було обрано модульну віконну структуру. Головне вікно забезпечує основу для побудови користувацького інтерфейсу програми. Фреймворк Qt має клас QMainWindow та пов'язані з ним класи для управління головним вікном. QMainWindow має власний макет, до якого можна додати QToolBar, QDockWidget, QMenuBar та QStatusBar, про які наведено нижче. Макет головного вікна має центральну область, яку може займати будь-який тип віджета. Схема макету головного вікна на рисунку 3.2.^[6]

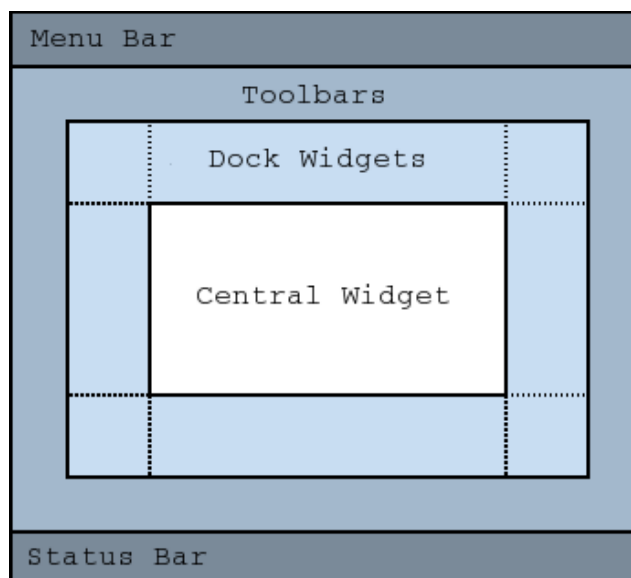


Рисунок 3.2 – Макет головного вікна QMainWindow

Центральним віджетом для головного вікна застосунку в цій роботі є QGraphicsView. QGraphicsView надає широку низку інструментів для візуалізації вмісту певної сцени у вікні перегляду, яке можна прокручувати. Цей віджет буде представляти візуальний вигляд схеми, із якою працює користувач.

Dock Widgets – плаваючі вікна, які можуть динамічно прикріплюватися до головного вікна або бути окремими вікнами. Таких вікон заплановано два – вікно інформації про об'єкт/систему і вікно, що показує список всіх елементів. Про плаваючі вікна буде детальніше наведено в наступних пунктах.

Вгорі вікна знаходиться QMenuBar, що містить різні дії, доступні користувачеві. Дії виражені класами QAction, та зазвичай розсортовані по

категоріях, що представлені класом QMenu. В межах даного проекту було виділено такі категорії: File, Edit, View. Всі дії меню головного вікна зображено на рисунку 3.3.

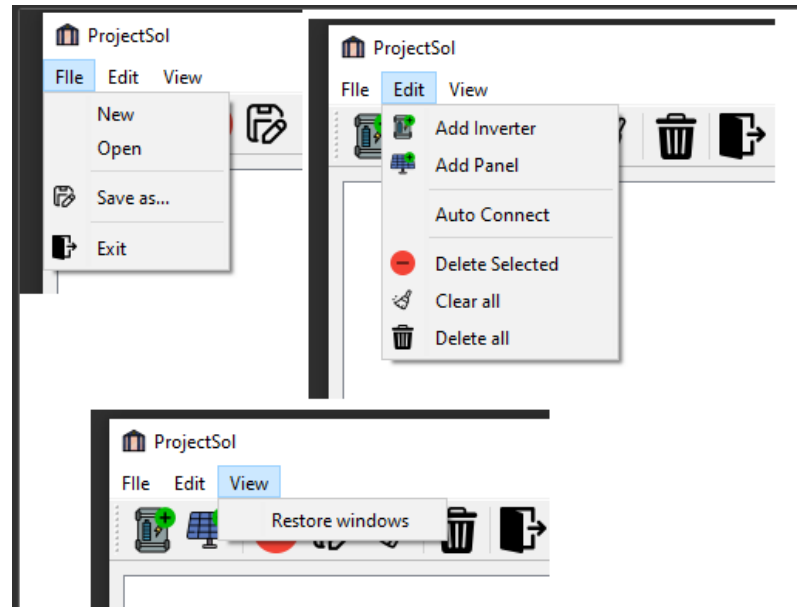


Рисунок 3.3 – Дії меню головного вікна

Серед дій QAction є такі, як додання панелей/інверторів, видалення елементів, очищення вікна, вихід тощо. Нижче QMenuBar міститься стрічка інструментів QToolBar, що дублює в собі основні дії із QMenu для швидкого доступу. Повний дизайн головного вікна, але без плаваючих віджетів, зображено на рисунку 3.4.

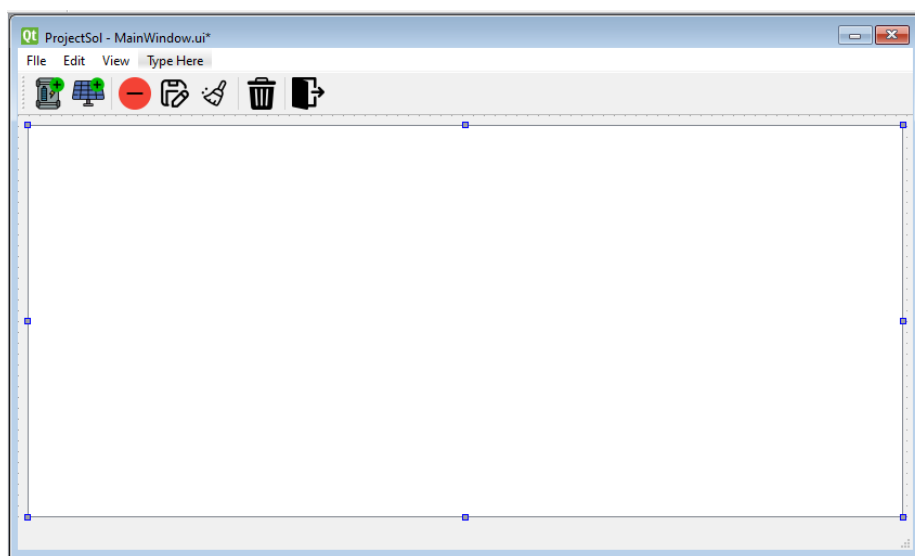


Рисунок 3.4 – Дизайн головного вікна без інших модулів

Клас `QStatusBar` надає горизонтальну панель в нижній частині головного вікна, яка підходить для відображення різноманітної інформації про стан об'єктів, дії що відбулися або інших повідомлень для користувача. В межах цієї роботи `QStatusBar` не використовуватиметься, його функціональні можливості будуть реалізовані іншими шляхами.

3.1.3 Плаваючі вікна `Dock Widgets`

Клас `QDockWidget` надає функціонал віджету, який можна закріпити всередині `QMainWindow` або розмістити як окреме вікно верхнього рівня на робочому столі. `QDockWidget` реалізує концепцію док-віджетів, також відомих як палітри інструментів, вікна утиліт або плаваючі вікна. Док-віджети – це вторинні вікна, розміщені в області `Dock Widgets` навколо центрального віджета в `QMainWindow`.^[6]

Плаваючі вікна можуть бути переміщені всередині їхньої поточної області, переміщені в нові області або від'єднані користувачем. API класу `QDockWidget` дозволяє розробнику обмежити можливість переміщення, виведення на екран та закриття віджетів док-станції, а також області, в яких вони можуть бути розміщені.^[6] В межах даної дипломної роботи було вирішено не обмежувати користувача в можливостях рухати та закривати плаваючі вікна, натомість давши можливість користувачеві повернути їх в початкове положення в будь-який момент опцією меню. Початковим ж положенням вважатимемо положення плаваючих вікон, визначене при старті застосунку, а також їх повна видимість.

`QDockWidget` є потужним засобом для побудови модульного та інтерактивного інтерфейсу у застосунках, що особливо актуально для програм з великою кількістю панелей управління або додаткових вікон. Але в основі `DockWidget` мусить знаходитися певний уже існуючий віджет, якому і буде надано функціонал плаваючого вікна.

3.1.4 Віджет інформації про об'єкт

Для відображення інформації про поточний об'єкт, тобто такий об'єкт, на який користувач натиснув востаннє, розроблено просте вікно, дизайн якого зображено на рисунку 3.5.

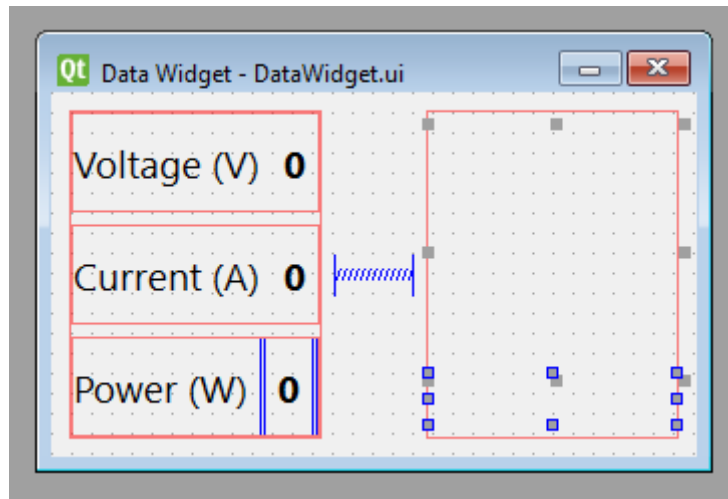


Рисунок 3.5 – Вікно інформації про об'єкт

Вікно відображатиме інформацію про три технічні характеристики, що важливі як для сонячної панелі, так і для інвертора, а також для всієї системи загалом. Відведено поля під напругу, струм та потужність із відповідними підписами, що реалізується класами QLabel. Для сонячних панелей це напруга, струм та потужність точки максимальної потужності із технічного паспорту. Для інвертора це максимальна напруга, максимальний струм та максимальна потужність із технічного паспорту. Якщо користувач клікнув десь у пустому місці, передбачено виведення технічної інформації про загальну схему. На віджеті також для зручності міститься зображення поточно обраного об'єкта, та його ідентифікатор. Всі елементи згруповано у горизонтальні QHBoxLayout та вертикальні QVBoxLayout макети із використанням пропусків Spacer, щоб досягти вигляду віджета, як на рисунку. Також така парадигма в дизайні віджетів гарантує адаптивність дизайну до зміни розміру віджетів. Віджет інформації про об'єкт за замовчуванням розміщено в правій частині головного вікна.

3.1.5 Віджет інвентаря елементів

Для відображення інвентаря об'єктів розроблено простий віджет, у якому центральним елементом є список із іконок, реалізований за допомогою QListWidget. Усього віджет містить один вертикальний QVBoxLayout, до якого додано єдиний дочірній елемент – саме цей список. За своєю розміткою, InventoryWidget має невелику висоту та фіксовану її політику, що означає незмінність розміру при зміні розміру головного вікна, в яке віджет за замовчуванням вкладається знизу. Передбачається, що віджет відображатиме вміст списку в один ряд в горизонтальному положенні, із можливістю перегортання.

Список у режимі IconMode розташовує елементи зліва направо, не допускаючи переносів на інші ряди, що за наявності такої функції автоматично б додавало більше рядів та конфліктувало із початковою ідеєю сталості висоти віджету. Для елементів передбачено сітку із невеликими відступами між елементами. Параметр рухливості елементів «movement=Snap» гарантує, що кожен елемент прилипає до позицій сітки та їх можна вільно переставляти. Кожному елементу передбачений уніфікований дизайн із іконкою, що зображує тип об'єкта (панель чи інвертор), а також підпис, що містить унікальний ідентифікатор об'єкта, який репрезентовано. Крім того, заборона редагування елементів робить список виключно панеллю перегляду, у якій користувач може лише обирати, переглядати чи переставляти об'єкти, але не змінювати їхній вміст. Дизайн віджета інвентаря зображено на рисунку 3.6.

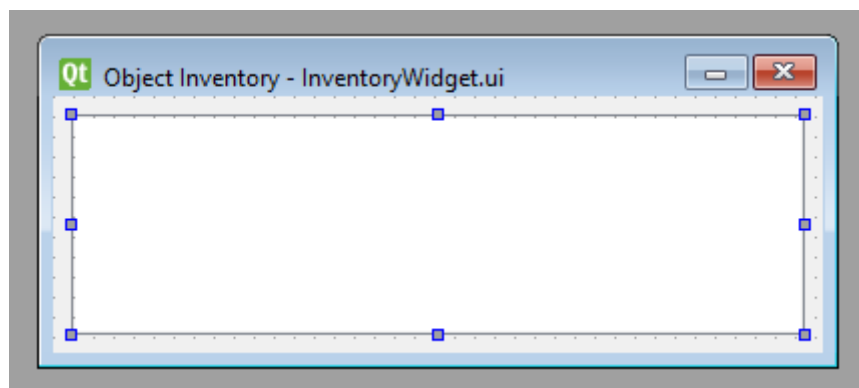


Рисунок 3.6 – Віджет інвентаря об'єктів

Цей віджет за замовчуванням вбудовується у нижню частину головного вікна і аналогічно є адаптивним в запланованих межах. Віджет можна відкріпити та розтягувати або стискати в горизональному напрямку. Завдяки такому дизайну користувач отримує інтуїтивно зрозумілу панель інвентаря, де кожен об'єкт представлений великою іконкою із ідентифікатором та може бути швидко знайдений і обраний.

3.1.6 Діалогові вікна

У межах цієї роботи діалогові вікна виконують ключову роль у зборі і валідації вхідних даних перед створенням об'єктів в системі, що представляють реальні фізичні об'єкти та явища. За допомогою модальних діалогів, створення інвертора та сонячної панелі в застосунку реалізовано централізовано, що одразу перевіряє формат введених даних і заповнення усіх полів. Це дозволяє запобігти неповним або несумісним даним в моделі. На рисунку 3.7 зображено дизайн обох діалогових вікон.

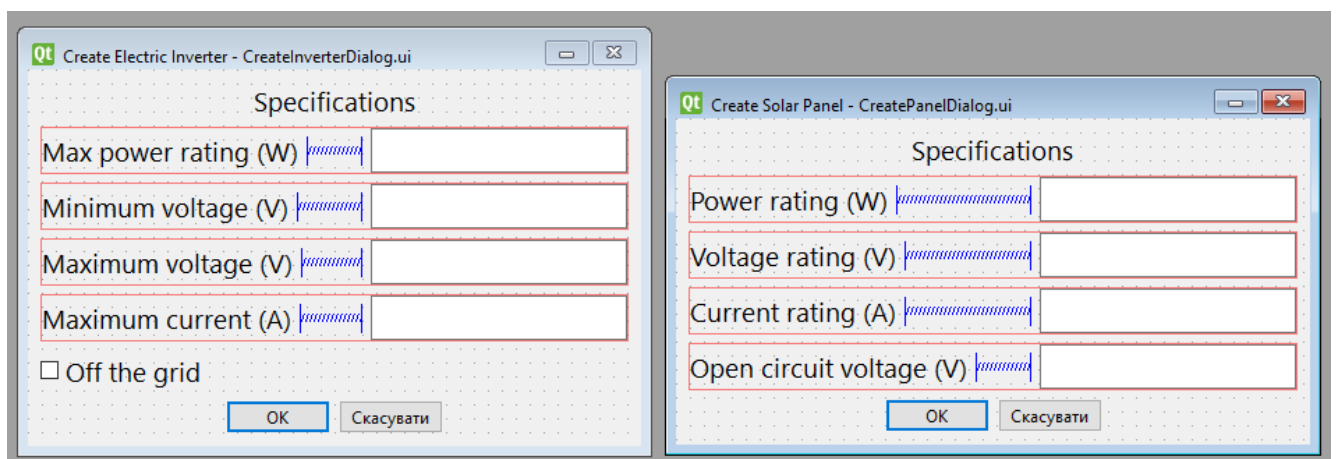


Рисунок 3.7 – Дизайн діалогових вікон

З точки зору дизайну обидва діалогові вікна наслідують від QDialog і побудовані за однаковою схемою. Верхня частина кожного вікна містить заголовок «Specifications», який одразу інформує користувача про контекст введення технічних характеристик. Основний блок реалізовано через вертикальний QVBoxLayout, у який послідовно додані горизонтальні ряди QHBoxLayout: ліворуч

розміщено QLabel із назвою певного параметра, справа – рядок введення QLineEdit із політикою обмеження довжини введення до 10 символів. Між ними вставлені динамічні пропуски, що гарантують рівномірні відступи й вирівнювання.

У діалозі Create Inverter передбачено поля для максимальної потужності, мінімальної й максимальної вхідної напруги, максимальної сили струму та прапорця «Off the grid», що вказує на параметр автономності. Прапорець реалізовано через QCheckBox, який також розміщено в окремому горизонтальному ряді з відповідним підписом. У діалозі Create Solar Panel аналогічним чином організовано чотири рядки для введення номінальної потужності, напруги, струму та напруги холостого ходу.

Нижня частина кожного діалогу містить групу кнопок QDialogButtonBox зі стандартними кнопками Ok і Cancel, які генерують сигнали accepted() та rejected(), які будуть використані при програмуванні поведінки віджетів. Використання цієї стандартної компоненти полегшує обробку результату в коді й гарантує однотипну поведінку: після підтвердження дані з усіх полів читаються та передаються до фабрики вузлів, у разі відміни – вікно закривається без змін моделі. Якщо користувач намагається відправити неповні або некоректні дані, передбачено інформування користувача про це. Для діалогових вікон передбачена фіксована політика розмірів та чітко налаштовані відступи, уникаючи небажаного розтягування або зсуву.

У цьому підрозділі було описано процес розробки концептуального дизайну віджетів графічного користувацького інтерфейсу та аргументовано дизайнерські рішення, що були прийняті на цьому етапі. У наступних підрозділах буде детально розглянуто механізм сигналів і слотів у Qt, а також описано, як разом із концептуальними дизайнами цей механізм використовувався для організації взаємодії між реальними компонентами інтерфейсу в даному проєкті.

3.2 Сигнали і слоти

У фреймворку PySide6, що є офіційним Python-зв'язком до фреймворку Qt, ключовим механізмом для організації взаємодії між елементами інтерфейсу є сигнали і слоти. Ця концепція реалізує принцип реактивного програмування, де об'єкти можуть «реагувати» на дії користувача або інші події без жорсткої прив'язки один до одного. Такий підхід значно спрощує побудову інтерфейсів з чітким поділом між логікою та відображенням.

Через особливості фреймворку Qt, всі QObject потребують способу комунікації, і саме тому цей механізм є центральною функцією Qt. Простіше кажучи, розуміти сигнали та слоти можна, як взаємодію зі світлом у домі. Натискання на вимикач світла – сигнал, продукує результат – слот, який може бути увімкненням або вимкненням лампочок. Під час розробки інтерфейсів найбільш базовим прикладом є натискання кнопки: «клік» буде сигналом, а слотом буде те, що відбувається при натисканні цієї кнопки, наприклад, закриття вікна, збереження документа тощо.^[6]

Всі класи, що успадковують QObject або один з його підкласів, наприклад QWidget, можуть містити сигнали та слоти. Сигнали випромінюються об'єктами, коли вони змінюють свій стан таким чином, що це може бути цікаво для інших об'єктів. Це все, що робить об'єкт для комунікації. Він не знає і не дбає про те, чи хтось отримує сигнали, які він випромінює. Це справжня інкапсуляція інформації, яка гарантує, що об'єкт може бути використаний як програмний компонент.^[6]

Слоти можуть використовуватися для прийому сигналів, але вони також є звичайними функціями-членами. Так само, як об'єкт не знає, чи хтось приймає його сигнали, слот не знає, чи є до нього підключені якісь сигнали. Це гарантує, що за допомогою Qt можна створювати справді незалежні компоненти.^[6]

До одного слоту можна підключити стільки сигналів, скільки необхідно. Аналогічно, один сигнал можна підключити до стількох слотів, скільки потрібно. Можна навіть підключити сигнал безпосередньо до іншого сигналу. Це призведе

до негайного випромінювання другого сигналу щоразу, коли випромінюється перший.^[6]

Тобто сигнали – це події, які надсилаються об’єктом, коли відбувається певна дія (натиснення кнопки, зміна значення поля тощо). Слоти – це методи або функції, які викликаються у відповідь на сигнали. Наприклад, коли користувач натискає кнопку ОК у діалоговому вікні створення панелі, об’єкт діалогового вікна генерує сигнал `submit_form` із параметром типу `dict` (словник Python). Параметр – це певна інформація, що випромінюється разом із сигналом. Оскільки параметр є словником, слот, що реагує на сигнал, може зручно обробити такий пакет даних та отримати необхідні відомості.

Сигнал `submit_form` можна «під’єднати» за допомогою методу `connect()` до будь-якого слоту, який має необхідну логіку – проаналізувати параметр та згідно нього оновити текст або перерахувати дані. В даній роботі з’єднання сигналів зі слотами зазвичай робиться у контексті головного вікна, оскільки воно є батьківським елементом і для джерела сигналу, і для необхідного слоту. А необхідним слотом у попередньому прикладі є функція, що обробляє створення нового об’єкту сонячної панелі у контексті віджета інвентаря.

Крім вбудованих сигналів, PySide6 дозволяє створювати нові сигнали за допомогою класу `Signal` з модуля `PySide6.QtCore`. Це особливо корисно для організації взаємодії між власними класами, які не є стандартними віджетами Qt. Попередній приклад було реалізовано саме завдяки власноруч створеному сигналу `submit_form`.

Приклади деяких сигналів, що використовуються в проєкті:

- `QAction.triggered` для всіх дій в меню головного вікна.
- `InventoryWidget.submit_item_data` для передачі даних про об’єкт із віджета інвентаря об’єктів на віджет, що відображає інформацію про об’єкт.
- `InventoryWidget.tree_updated` для оновлення зображення дерева підключень.

В цьому підрозділі було розібрано систему сигналів і слотів. Вона є одним із найпотужніших інструментів у PySide6, що дозволяє побудувати масштабовані, зручні у підтримці застосунки із графічним інтерфейсом користувача з чітко розмежованими компонентами та зонами відповідальності.

3.3 Програмна реалізація віджетів

3.3.1 MainWindow – головне вікно застосунку

У межах даної роботи клас `MainWindow` виконує роль центрального віджета графічного інтерфейсу, комбінуючи макет із механізмом взаємодії між різними компонентами. Успадковуючи від `QMainWindow` та від згенерованого за допомогою `Qt Designer` макета `Ui_MainWindow`, він ініціалізує головне вікно застосунку, підключає головні дії меню до відповідних методів, що їх обробляють та координує роботу плаваючих вікон, діалогів і області відображення графічного зображення дерева підключень.

У конструкторі `__init__()` відбувається виклик `setupUi()`, що монтує елементи форми згідно макету, після чого через сигнали `triggered` головних дій меню здійснюється прив'язка таких операцій, як вихід із програми, відновлення видимості док-панелей, створення нових панелей і інверторів або видалення обраного елемента. Таким чином, клас забезпечує реактивність інтерфейсу на вибір пунктів меню безпосередньо в головному вікні.

Метод `set_dockwidgets()` призначає два док-вікна: одне для відображення інформації про поточний об'єкт через `DataWidget`, інше – для списку наявних елементів через `InventoryWidget`. Обидва вікна прикріплюються до правої та нижньої областей головного вікна за допомогою `addDockWidget()`. В цьому ж контексті встановлюються релевантні міжвіджетні зв'язки через сигнали й слоти: при взаємодії із елементом з інвентаря оновлюється інформаційна панель за

сигналом `submit_item_data`, а кнопки меню активують методи видалення всіх елементів, конкретного елемента або автоматичного побудови дерева з'єднань.

Метод `mousePressEvent()` доповнює функціонал головного вікна, дозволяючи при кліці мишею у вільному місці оновлювати інформаційний віджет, вказуючи в ньому системні технічні характеристики схеми. Це дає змогу отримати оцінку стану системи без введення додаткових кнопок.

Метод `updateTreeImage()`, підписаний на сигнал `tree_updated` від інвентарного вікна, відповідає за візуалізацію схеми комутацій при оновленні: він створює нову `QGraphicsScene` замість старої, завантажує попередньо збережений растровий файл з образом дерева комутації і поміщає його у `QGraphicsView`. За відсутності актуальної сформованої структури сцена просто очищується, що гарантує коректну реакцію інтерфейсу на різні стани даних інформаційної системи на різних етапах.

Методи `open_create_panel_dialog()`, `open_create_inverter_dialog()` оперують модальними діалоговими вікнами створення панелі та інвертора. Вони зв'язуються з інвентарним віджетом за допомогою сигналу `submit_form`, що дозволяє, при підтвердженні введених у діалоговому вікні параметрів, передати зібраний пакет даних у фабрику вузлів та створити за ними новий вузол у моделі даних. Методи `quit()` та `view_restore()` забезпечують завершення роботи застосунку й відновлення прихованих плаваючих вікон відповідно. Діаграма класу `MainWindow` зображена на рисунку 3.8.

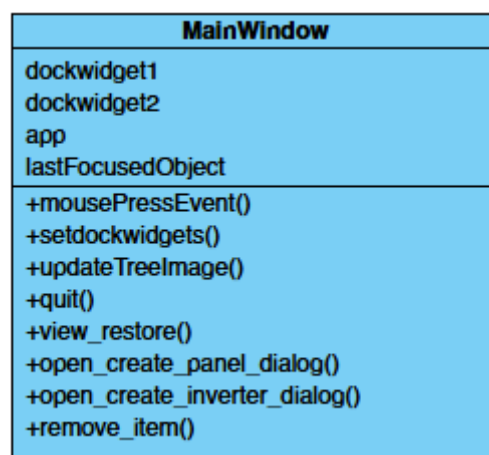


Рисунок 3.8 – Клас головного вікна `MainWindow`

MainWindow виступає в проєкті об'єднаним центром координації дочірніх UI-компонентів, забезпечує інтеграцію створених діалогів, док-панелей і графічного відображення структури в єдиний взаємопов'язаний інтерфейс, що гарантує узгодженість і зручність роботи користувача з моделлю комутації сонячних панелей.

3.3.2 Плаваюче вікно DataWidget

Клас DataWidget у цьому проєкті виконує функцію динамічного інформаційного вікна, що призначене для відображення актуальних параметрів як окремих елементів (сонячних панелей або інверторів), так і всієї системи загалом. Цей клас наслідує від QWidget та класу інтерфейсного макету Ui_DataWidget, згенерованого за допомогою Qt Designer. Основна мета реалізації цього класу – забезпечити реактивне оновлення графічного інтерфейсу на основі вхідних структурованих даних, що надходять від інших компонентів системи у вигляді словника datadict.

На рівні реалізації, ключовим методом, окрім __init__(), є updateData(), який приймає словник параметрів, що описують поточний фокус системи: це може бути окрема сонячна панель, інвертор, вся система або стан «нічого не вибрано». У випадку, коли тип об'єкта (параметр kind) дорівнює "panel" або "inverter", віджет переходить у режим фокусу на конкретному об'єкті, встановлюючи focusObjectId та встановлюючи прапорець realObjectInFocus у значення True. Відповідно до типу, встановлюється графічне представлення компонента: для панелей – іконка панелі із файлу solar_panel.png, для інверторів – іконка electric_invertor.png. Зображення масштабуються під доступну область вікна з використанням режиму KeepAspectRatioByExpanding, що гарантує коректне та адаптивне відображення незалежно від пропорцій. Параметри напруги, струму та потужності відображаються у відповідних полях типу QLabel. Якщо тип об'єкта встановлено на "tree", тобто в фокусі перебуває загальна структура комутаційного дерева, тоді

зображення відсутнє, а числові поля заповнюються схемовими технічними характеристиками. Замість ідентифікатора об'єкта відображається підпис "Connection Tree", що дає зрозуміти користувачеві, що йдеться про агреговані характеристики всієї системи.

Окремо передбачено обробку випадку `kind == 0`, який означає відсутність вибору – наприклад, коли користувач натиснув поза будь-якими об'єктами, але дерево ще не сформовано. У цьому випадку `focusObjectId` скидається до -1, усі числові поля заповнюються нулями, прапорець `realObjectInFocus` переводиться у стан `False`, а графічне поле очищується. Це дозволяє системі реагувати на відсутність даних для відображення, не вводити користувача в оману неактуальною інформацією або, у гіршому випадку, створювати виключну ситуацію. Діаграма класу `DataWidget` зображена на рисунку 3.9.

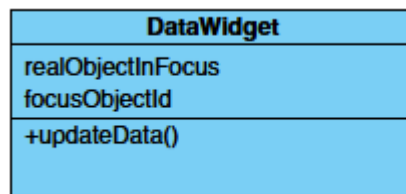


Рисунок 3.9 – Клас відображення інформації про об'єкт

Клас `DataWidget` реалізує універсальний інтерфейс для представлення технічних характеристик будь-яких об'єктів системи у зрозумілій, візуально структурованій формі. Завдяки використанню єдиного поліморфного методу `updateData()`, який обробляє всі можливі стани інтерфейсу через просту логіку розгалуження, досягається ефективна підтримка відгуку інтерфейсу на зміну фокусу. Крім того, наявність параметрів `focusObjectId` та `realObjectInFocus`, що ідентифікують стан системи, дозволяє іншим компонентам (наприклад, `MainWindow`) перевіряти поточний контекст віджета та ухвалювати відповідні рішення – викликати оновлення даних, здійснювати видалення об'єкта.

3.3.3 Плаваюче вікно InventoryWidget

Клас `InventoryWidget` виконує роль інтерактивної панелі візуалізації елементів енергосистеми – інверторів і сонячних панелей. Він є одним з двох плаваючих віджетів головного вікна та відповідає за зберігання, відображення, видалення та групове об'єднання компонентів у логічну структуру – схему комутації. Клас реалізовано як підклас `QWidget`, а також підклас макету `Ui_InventoryWidget`, отриманий через `Qt Designer`, і містить графічний елемент `QListWidget` для представлення списку компонентів системи у вигляді іконок із підписами в горизонтальному розміщенні.

Основне призначення `InventoryWidget` – відображення та контроль об'єктів, що включає створення та видалення об'єктів. Панелі та інвертори створюються з використанням фабрики `NodeFactory` і зберігаються у внутрішньому списку `dataInventoryList`. Кожен новий об'єкт, після створення через відповідне діалогове вікно, додається до інвентаря через метод `new_item()`, який викликає `add_to_inventory()`. Тут створюється графічне представлення у вигляді елемента `QListWidgetItem` списку `QListWidget`, асоційованого з відповідною іконкою сонячної панелі чи інвертора.

Клас визначає два основні сигнали: `submit_item_data` для передачі технічної інформації про вибраний компонент у віджет даних `DataWidget`, та `tree_updated`, що інформує систему про зміну структури дерева. При натисканні на елемент списку метод `get_item_info()` витягує об'єкт з `dataInventoryList` за унікальним ідентифікатором `ui_id`, формує словник параметрів та випромінює сигнал `submit_item_data` разом із словником. `DataWidget` отримує сигнал та обробляє його параметр для коректного відображення. У цьому полягає ключовий механізм взаємодії між плаваючими віджетами на рівні інтерфейсу користувача.

Для видалення одного компонента з системи використовується метод `full_item_remove()`, який синхронно оновлює як внутрішню модель `dataInventoryList`, так і графічний список `QListWidget`. У випадку, якщо видаляється

інвертор, який колись було збережено як корінь поточної побудованої структури `auto_tree.root`, вся модель дерева анулюється, і сигнал `tree_updated` повідомляє про це інші частини системи, що могли б бути зацікавлені в цьому. Повне очищення списку передбачено методом `remove_all_items()`, який також обнуляє глобальний лічильник ідентифікаторів, гарантуючи чистий стартовий стан системи в усіх аспектах без перезапуску застосунку.

Ключова функція `auto_build_tree()` реалізує поведінку алгоритмічного модуля побудови дерева `AutoTreeBuilder`. Метод вибирає найбільш потужний інвертор серед наявних як корінь дерева та формує список сонячних панелей. Далі викликається `grow_tree()` об'єкта `AutoTreeBuilder`, який на основі розробленого алгоритма групує панелі в послідовно-паралельні структури з урахуванням обмежень інвертора. Після побудови дерева, корінь зберігається для зручності в об'єкт `AutoTree`, а також графічно виводиться у файл `tree.png` засобами бібліотеки `bigtree` – методом `tree_to_dot()`. Цей файл використовується для оновлення зображення у головному вікні. У разі помилки (наприклад, рекурсивного виснаження через неможливість з'єднати панелі оптимальним шляхом) користувач отримує відповідне повідомлення через `QMessageBox`.

Останній метод `get_sys_stats()` використовується головним вікном для отримання агрегованих характеристик усієї побудованої системи – загальної напруги, потужності та струму. Якщо дерево вже збудовано, ці значення випромінюються сигналом `submit_item_data` із пакетом даних, що вказує `kind = "tree"` та містить детальну інформацію для відображення. В іншому ж випадку пакет подається зі вказанням порожнього вибору, тобто `kind = 0`. Діаграма класу `InventoryWidget` зображена на рисунку 3.10.

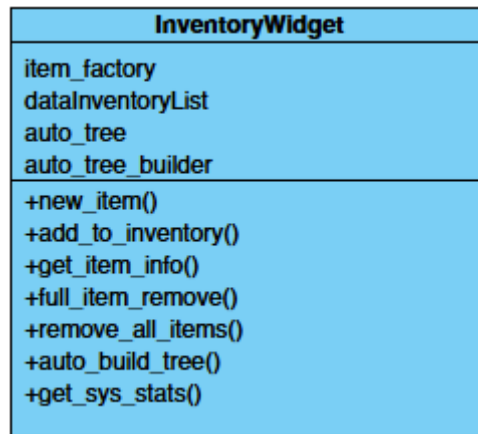


Рисунок 3.10 – Клас інвентарного віджета

InventoryWidget є ключовим поєднувачем графічної інтерфейсної частини застосунку зі структурою даних. Він забезпечує інтерактивне управління складовими частинами системи, формування структури з'єднань та координацію з іншими вікнами через сигнально-слотову модель. Клас є реалізацією патерну Model View Controller: він виконує роль контролера та частково представлення, інтегруючи модель даних dataInventoryList з графічним інтерфейсом QListWidget і алгоритмічною логікою AutoTreeBuilder.

3.3.4 Діалогові вікна

Класи CreatePanelDialog та CreateInverterDialog реалізують спеціалізовані діалогові вікна для створення нових компонентів енергетичної системи: сонячних панелей та інверторів відповідно. Обидва класи наслідують від QDialog і відповідного інтерфейсного класу макета, згенерованого з .ui файлів за допомогою Qt Designer. Їх призначення полягає в тому, щоб забезпечити користувацький інтерфейс для ручного введення технічних характеристик елементів перед їх додаванням до системної моделі.

Основна мета діалогових вікон полягає у зборі структурованих числових даних, необхідних для створення вузлів типу SolarPanelNode або InverterNode, що надалі використовуються при формуванні дерева з'єднань. Користувач вводить ці значення вручну, після чого діалог перевіряє коректність введених параметрів і, у

разі успішного підтвердження, випромінює сигнал `submit_form` із підготовленим словником параметрів. Таке інкапсульоване введення є ключовим для підтримки консистентності системи, оскільки унеможливорює створення елементів з неповними або некоректними технічними характеристиками. Зібрані у такий спосіб дані отримуються інвентарним віджетом, де буде створено відповідний об'єкт у моделі та відображенні.

У конструкторі кожного класу викликається `setupUi()`, що ініціалізує макет інтерфейсу віджета, після чого перевизначаються стандартні з'єднання кнопок `Ok` та `Cancel`, надані елементом `QDialogButtonBox`. Замість типових слотів використовуються власні методи: `accept_and_emit()` при підтвердженні введення та `rejected_close()` при скасуванні.

Метод `accept_and_emit()` діалогового вікна створення інвертора намагається зчитати п'ять числових параметрів: максимальну потужність, мінімальну та максимальну напругу, максимальний струм, а також логічний стан прапорця автономності (`off-grid`). Зібрані дані упаковуються у словник `form_data`, до якого додається ключ `"kind": "inverter"`, та випромінюються у зовнішню систему через сигнал `submit_form`. У випадку помилки чи виключної ситуації, наприклад, порожнє або нечислове поле, відображається діалог критичної помилки `QMessageBox.critical`, а введення не приймається. Аналогічна логіка реалізована і для сонячних панелей. Користувач вводить чотири параметри: максимальну потужність, напругу `MPP`, струм `MPP` і напругу холостого ходу. Після перевірки і валідації вони передаються у вигляді словника з ключем `"kind": "panel"` через сигнал `submit_form`. Метод `rejected_close()`, у свою чергу, просто скасовує діалогове вікно.

Кожен із діалогів не виконує жодних змін самостійно, а натомість передає валідовані дані у вигляді сигналу із параметром, що реалізує архітектурні принципи інверсії керування та впровадження залежностей. Це дозволяє легко масштабувати або замінювати ці компоненти, не змінюючи інші частини застосунку. У головному вікні `MainWindow` сигнал `submit_form` від діалогу

пов'язаний зі слотом `new_item()` інвентарного віджета, що забезпечує логічне завершення процесу створення нового вузла після отримання та валідації даних.

Класи `CreatePanelDialog` і `CreateInverterDialog` реалізують діалогову підсистему збору параметрів, критично важливу для формування стійкої та достовірної структури енергетичної системи. Вони чітко розмежовують відповідальність між інтерфейсом введення, перевіркою вхідних даних і передачею в модель, що забезпечує модульність, масштабованість і надійність у взаємодії між компонентами застосунку. Класова діаграма `CreatePanelDialog` та `CreateInverterDialog` відображена на рисунку 3.11.

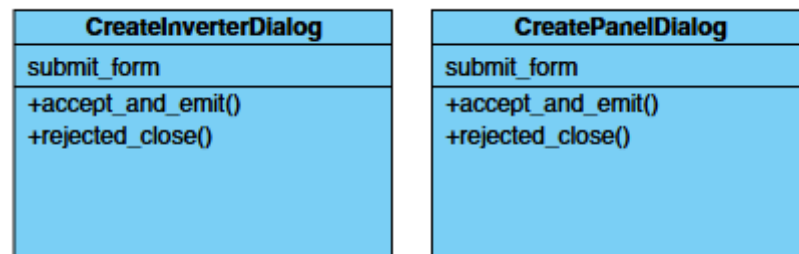


Рисунок 3.11 – Класи `CreateInverterDialog` та `CreatePanelDialog`

У цьому підрозділі було розглянуто реалізацію основного вікна застосунку, плаваючих віджетів – панелі даних і інвентаря, а також діалогових вікон створення нових елементів. Розглянуто, як ці компоненти взаємодіють між собою через сигнали й слоти для підтримки динамічного та узгодженого інтерфейсу.

Висновки до розділу 3

У третьому розділі було описано процес розробки концептуального дизайну віджетів графічного користувацького інтерфейсу та аргументовано прийняті дизайнерські рішення. Детально розглянуто систему сигналів і слотів у PySide6, яка є потужним інструментом для створення масштабованих і зручних у підтримці застосунків із чітко розмежованими компонентами та зонами відповідальності. Описано, як на основі реалізованих попередньо макетів, спираючись на механізм сигналів і слотів, розробити класи для реальних компонентів графічного

інтерфейсу, організувати взаємодію між компонентами, та користувача із компонентами, забезпечити динамічний і узгоджений інтерфейс користувача.

Класу `MainWindow` виконує роль головного інтегратора графічного інтерфейсу: керує відображенням, взаємодією користувача з меню, відкриттям діалогових вікон та координацією зв'язків між віджетами. Його структура забезпечує логічну організацію елементів на екрані та дозволяє реагувати на дії користувача в реальному часі.

`InventoryWidget` реалізує інтерфейс керування всіма наявними об'єктами системи. Він відповідає за збереження, відображення, видалення елементів і побудову дерева з'єднань на основі вхідних параметрів. Завдяки сигналам `submit_item_data` та `tree_updated`, цей компонент динамічно синхронізований з іншими частинами інтерфейсу.

`DataWidget` відповідає за виведення параметрів активного об'єкта або всієї системи у графічному вигляді. Він реагує на зміну фокусу через простий механізм обробки пакету даних сигналу, змінюючи відповідні поля та об'єкти.

Діалогові вікна `CreatePanelDialog` та `CreateInverterDialog` надають користувачеві зручний інтерфейс для введення технічних параметрів нових компонентів. Ці діалогові вікна реалізують інкапсульовану логіку перевірки цілісності та відповідності даних і відправляють результат за допомогою сигналу. Їх інтеграція дозволяє підтримувати єдиний сценарій взаємодії між користувачем та логікою застосунку.

4 ТЕСТУВАННЯ ТА ВАЛІДАЦІЯ

4.1 Тестування розробленої системи

Для забезпечення коректної та надійної роботи розробленого застосунку було проведено комплексне тестування, яке включало модульне тестування основних функціональних компонентів та тестування користувацького інтерфейсу. Такий підхід дозволив виявити і усунути потенційні дефекти на різних рівнях архітектури застосунку на різних стадіях розробки.

Модульне тестування здійснювалося з використанням модуля unittest мови Python, що забезпечило автоматизовану перевірку логіки окремих класів і функцій застосунку. Основними об'єктами тестування були:

- Алгоритми формування та оптимізації деревовидної структури з'єднань сонячних панелей.
- Точки створення нових об'єктів у програмній системі.
- Логіка перевірки входних параметрів, контроль цілісності значень та коректності типів даних.

Були розроблені тести, що охоплювали типові та негативні сценарії. Було досягнуто високий рівень покриття тестами, що забезпечило високий рівень довіри до коректності виконання основних обчислювальних функцій застосунку.

Для забезпечення високої якості користувацького досвіду та коректності взаємодії користувача із застосунком було проведено тестування інтерфейсу. Основні напрямки тестування UI/UX включали:

- Функціональність основних елементів управління: кнопки керування, панелі, діалогові вікна.
- Перевірку коректності відображення інформації при різних станах застосунку, включно з пустими та граничними наборами даних.

- Реакцію інтерфейсу на некоректні дії користувача: введення недопустимих значень, спроби додати конфліктні параметри, пусті параметри тощо.

- Стабільність роботи при послідовному виконанні типових сценаріїв: створення, видалення елементів, формування схем комутації, очищення, перевірка.

В цьому підрозділі було описано процес тестування застосунку, розробленого в межах даної роботи, описані методи та їх аргументація. Результати тестування підтвердили, що інтерфейс є інтуїтивно зрозумілим, відсутні критичні помилки, а застосунок коректно обробляє некоректні вхідні дані та реагує на динамічні зміни стану. Крім того перевірено взаємодію між модулями: коректну передачу даних між моделлю даних та відображенням, а також коректність оновлення показників.

4.2 Користувацький огляд

Після запуску вікно програми приймає вигляд як на рисунку 4.1, з віджетами розміщеними на головному вікні програми. Їх можна вільно відкріплювати та переміщувати, як на рисунку 4.2.

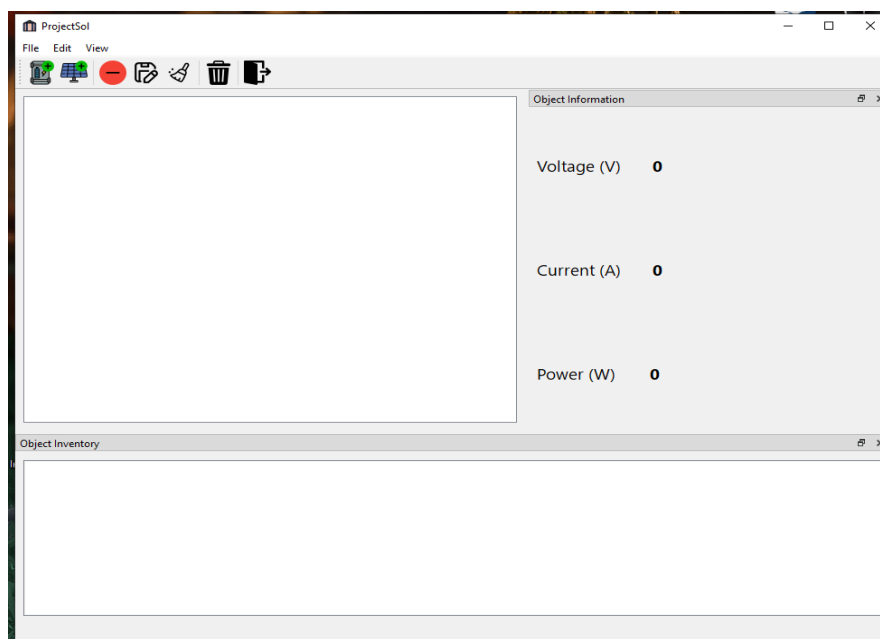


Рисунок 4.1 – Головне робоче вікно застосунку

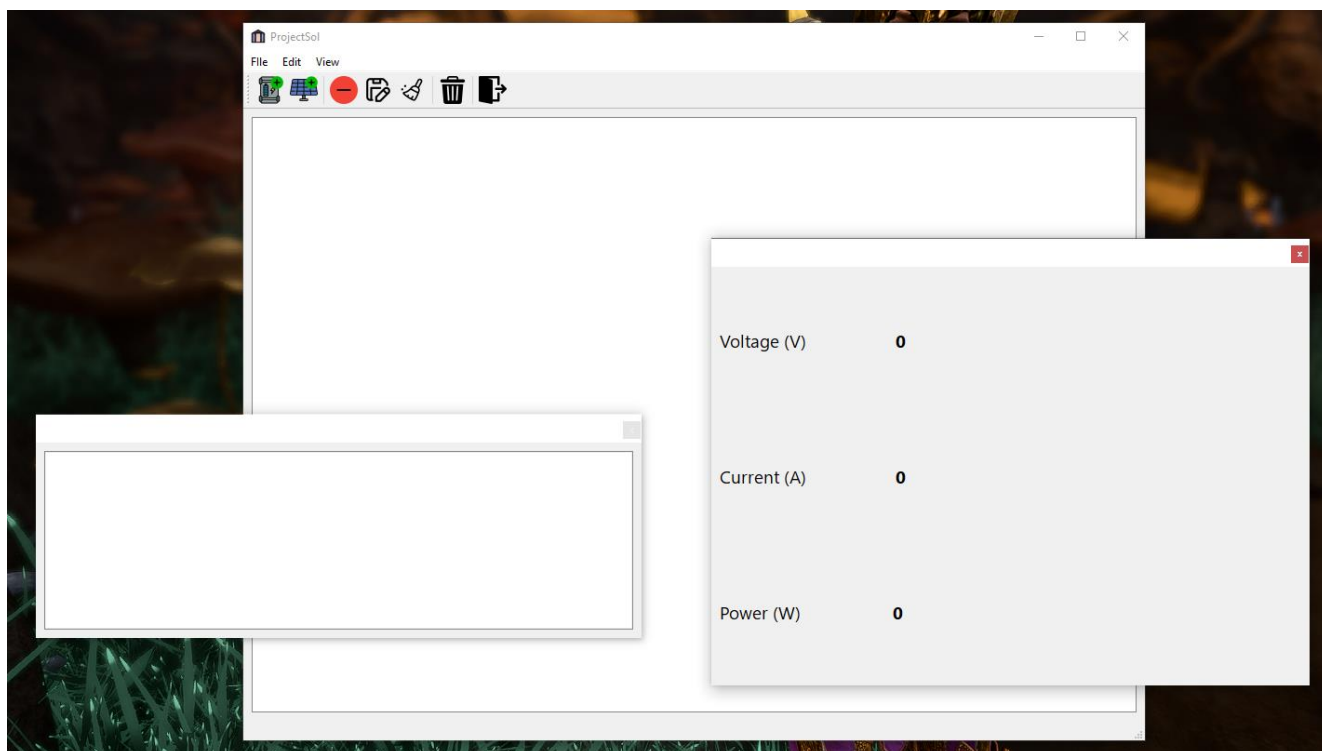


Рисунок 4.2 – Можливість вільно адаптувати модулі за потреби

Меню у верхній частині дозволяє виконувати різноманітні операції, можливості меню зображені на рисунку 4.3.

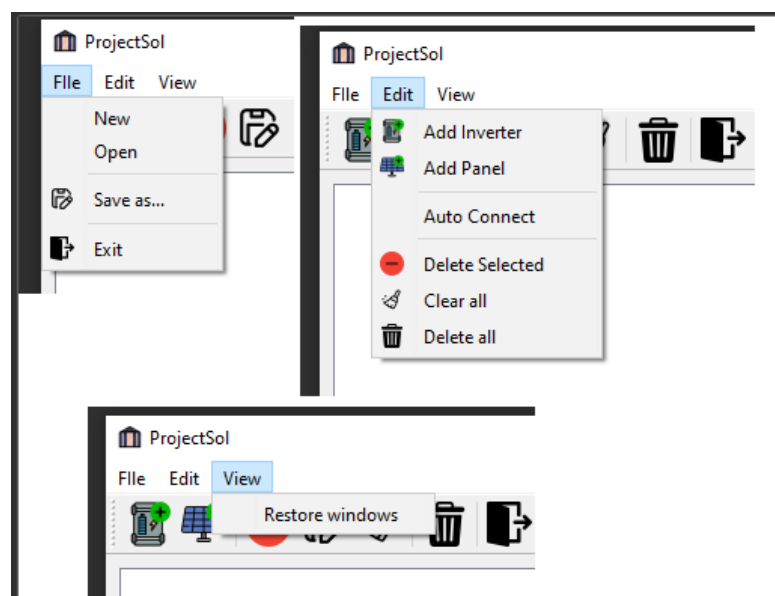


Рисунок 4.3 – Можливості меню головного вікна

Додавання сонячних панелей та інверторів відбувається за прикладом, зображеним на рисунку 4.4.

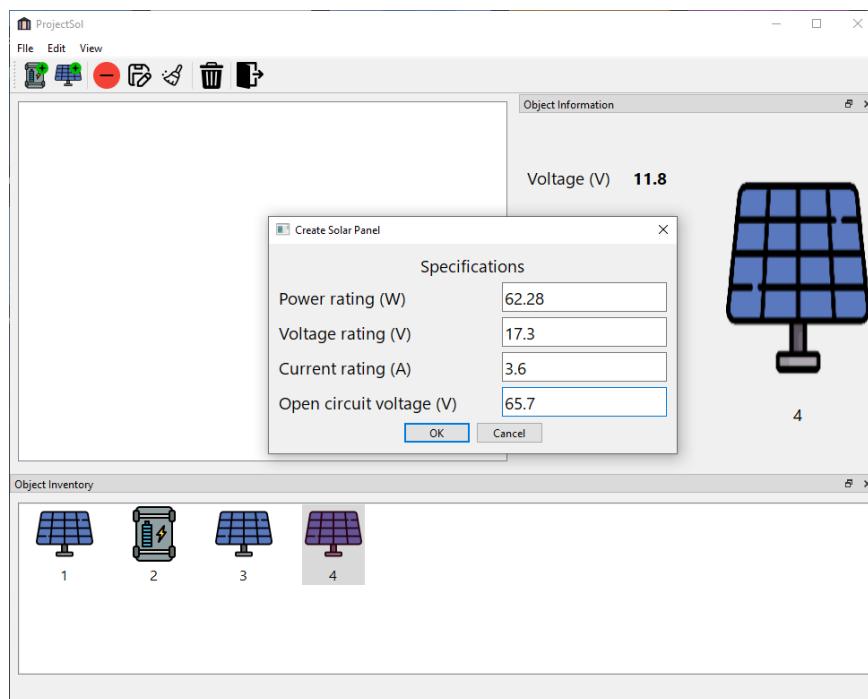


Рисунок 4.4 – Введення даних для додання нового елемента

Додамо ще кілька елементів та скористаємося функцією меню Auto Connect під категорією меню Edit, щоб побудувати схему комутації сонячних панелей. Алгоритм збудує та відобразить граф з'єднань, вершини якого відображають елементи системи. Вузли підписані за відповідним номером зі списку всіх елементів. За схемою цього графа можна збудувати таку сонячну електростанцію, очікувані характеристики якої виведено у інформаційному віджеті. Приклади схем, які будує застосунок – на рисунках 4.5 та 4.6.

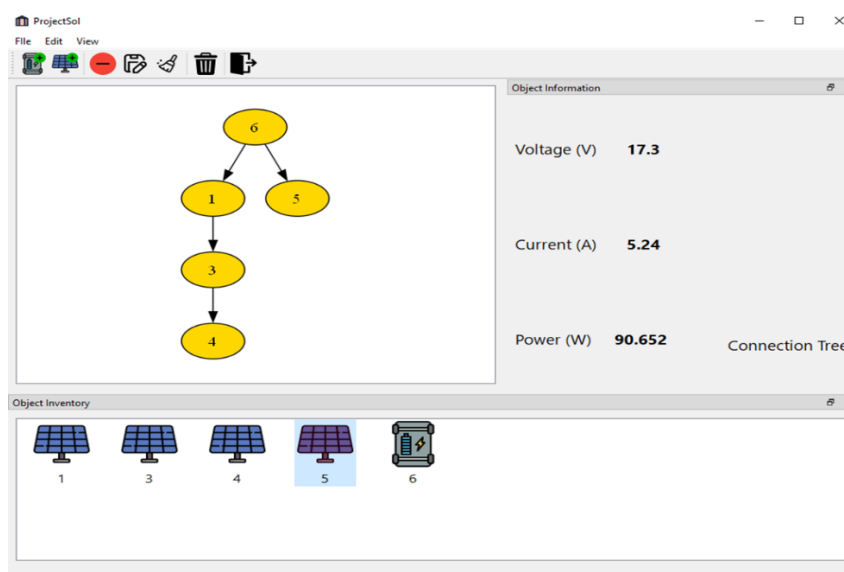


Рисунок 4.5 – Простий приклад побудованого дерева комутації

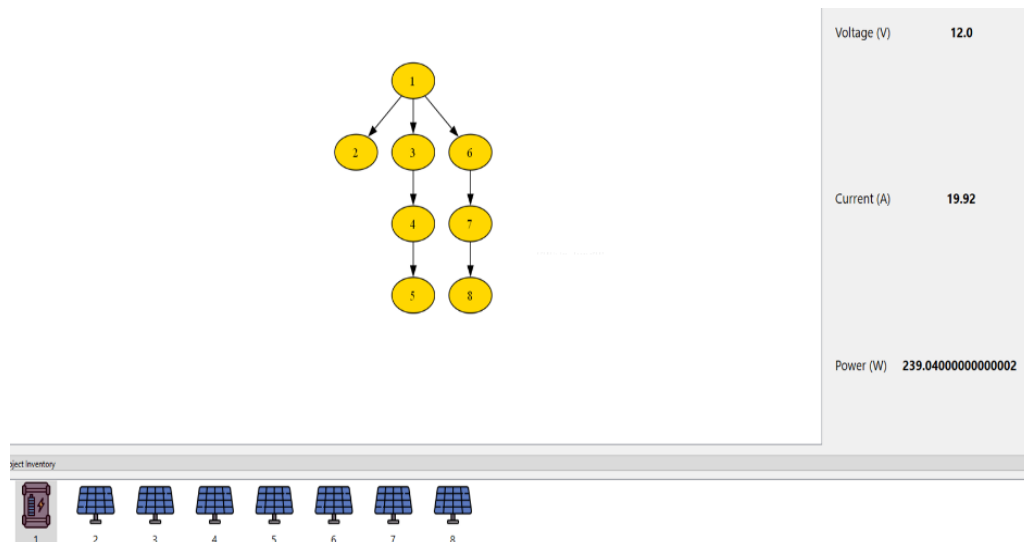


Рисунок 4.6 – Більш складний приклад дерева комутації

На рисунку 4.5 коренем дерева є електричний інвертор, якому було надано ідентифікаційний номер 6. До нього паралельно з'єднуються дві гілки сонячних панелей: 5 та 1,3,4. Те, що панелі під номерами 1, 3, 4 об'єднані в одну гілку означає, що між ними тип з'єднання послідовний, тобто «плюс» панелі 4 з'єднується із «мінусом» панелі 3, і так далі. В результаті із цього ряду будуть вільними «плюс» панелі 1 та «мінус» панелі 4 або навпаки, «мінус» панелі 1 та «плюс» панелі 4. Їх необхідно об'єднати із такими ж виходами панелі 5 та разом підключити до інвертора. Автоматичний алгоритм оцінив, що загальна вихідна потужність саме такого рішення комутації буде найкращою із можливих при заданих панелях або достатньо близькою до найкращої.

В цьому підрозділі було описано процес типової взаємодії користувача із розробленою програмною системою, наведено графічні приклади використання можливостей програми.

Висновки до розділу 4

У четвертому розділі було наведено, як виконувалося тестування застосунку в межах даної роботи, описані методи та висновки, а також наведено приклади типової взаємодії користувача із системою, її функціоналом та відображенням.

ВИСНОВКИ

У результаті виконання дипломної роботи було створено програмний застосунок для моделювання сонячних електростанцій із різнорідних сонячних панелей. Основна мета проєкту полягала в тому, щоб надати користувачеві інструмент, який дозволяє зручно проєктувати оптимальні конфігурації сонячних панелей, перевіряти їх на відповідність технічним обмеженням та візуалізувати отримані характеристики. У розробленому застосунку користувач має змогу вводити чи імпортувати параметри панелей та електричного інвертора, автоматично формувати з'єднання та зображувати їх у вигляді графа, а система в реальному часі розраховує напругу, струм, потужність і визначає, чи відповідає така конфігурація допустимим умовам для інвертора.

Робота застосунку базується на змішаному евристичному алгоритмі, який поступово будує можливі комбінації панелей, частково повертаючись назад і оцінюючи власні дії, що дозволяє швидко знайти оптимальне з'єднання за заданими критеріями. Під час тестування система показала стабільну роботу з типовими наборами даних і середніми за розміром конфігураціями.

На поточному етапі система працює стабільно й швидко для невеликих і середніх (домашніх або мікроіндустріальних) проєктів. При значному збільшенні кількості елементів, часова складність алгоритму автоматичної оцінки буде очікувано збільшена.

При виконанні роботи не було враховано змінні зовнішні та погодні умови, такі як температура повітря, хмарність, розміщення панелей, інсоляція, і використовує лише базові технічні характеристики обладнання, що вказуються виробником.

Проєкт має потенціал для подальшого розвитку. Можна розширити функціонал, оптимізувати алгоритм для роботи з більшими обсягами даних, додати можливість імпорту та експорту технічних характеристик з відкритих баз даних, реалізувати генерацію звітів у PDF або Excel, а також можливість обміну проєктами

між користувачами. Це зробить застосунок ще зручнішим і потужнішим. У перспективі, за потреби, застосунок може бути адаптований для роботи у веб-середовищі або на мобільних пристроях (зокрема, планшетах), що зробить його доступним для ширшого кола користувачів або способів використання, зокрема проєктування прямо в польових умовах.

Незважаючи на це, програмний продукт уже зараз може бути корисним для новачків та більш професійних проєктувальників подібних систем, а також будь-кого, хто цікавиться або досліджує галузь відновлювальної енергетики.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бакалаврська кваліфікаційна робота. Організація виконання і захисту, вимоги до структури, змісту та оформлення. Навчальний посібник. [Електронний ресурс]. – Режим доступу: <https://ela.kpi.ua/handle/123456789/57422>
2. Переддипломна практика: Організація, підготовка, проведення. Навчальний посібник. [Електронний ресурс]. – Режим доступу: <https://ela.kpi.ua/handle/123456789/57414>
3. Photovoltaics and electricity – U. S. Energy Information Administration. [Електронний ресурс]. – Режим доступу: <https://www.eia.gov/energyexplained/solar/photovoltaics-and-electricity.php>
4. Types of solar panels: monocrystalline, polycrystalline, and thin-film. [Електронний ресурс]. – Режим доступу: <https://www.solarreviews.com/blog/pros-and-cons-of-monocrystalline-vs-polycrystalline-solar-panels>
5. Solar Electric System Design, Operation and Installation. Washington State University. [Електронний ресурс]. – Режим доступу: <https://www.energy.wsu.edu/documents/solarpvforbuildersoct2009.pdf>
6. PySide6.QtWidgets Documentation. [Електронний ресурс]. – Режим доступу: <https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/index.html>

ДОДАТОК А Лістинг розробленої системи

```
---main.py---
# This Python file uses the following encoding: utf-8
from PySide6.QtWidgets import QApplication
from ui.widgets.MainWindow import MainWindow
from ui.widgets.DataWidget import DataWidget
from ui.widgets.InventoryWidget import InventoryWidget

if __name__ == "__main__":
    app = QApplication([])
    mainWindow = MainWindow(app)
    dataWidget = DataWidget()
    inventoryWidget = InventoryWidget()
    mainWindow.set_dockwidgets(dataWidget, inventoryWidget)
    mainWindow.show()
app.exec()

---AutoTreeBuilder.py---
from dataclasses import dataclass
from typing import List
from data.DataTreeNodes import InverterNode, SolarPanelNode

@dataclass
class SeriesString:
    panels: List[SolarPanelNode]

    def total_mpp_voltage(self) -> float:
        return sum(p.mpp_voltage for p in self.panels)

    def total_voc(self) -> float:
        return sum(p.oc_voltage for p in self.panels)

    def limiting_current(self) -> float:
        return min(p.mpp_amps for p in self.panels)

    def effective_power(self) -> float:
        return self.total_mpp_voltage() * self.limiting_current()

@dataclass
class ParallelArray:
    strings: List[SeriesString]

    def total_current(self) -> float:
        return sum(s.limiting_current() for s in self.strings)

    def total_power(self) -> float:
        return sum(s.effective_power() for s in self.strings)

class AutoTreeBuilder:
    def __init__(self, inverter: InverterNode, voltage_tol=5.0,
current_tol=1.0):
        self.inverter = inverter
        self.arrays: List[ParallelArray] = []
        self.unassigned: List[SolarPanelNode] = []
        self.voltage_tol = voltage_tol
        self.current_tol = current_tol
```

```

        self.total_voltage = 0.0
        self.total_current = 0.0
        self.total_power = 0.0

    def grow_tree(self, panels: List[SolarPanelNode]):
        for panel in panels:
            if not self._assign(panel):
                self.unassigned.append(panel)
                self._backtrack(panel)

    def _assign(self, panel: SolarPanelNode) -> bool:
        # Try to add to any existing string
        for array in self.arrays:
            for string in array.strings:
                if self._can_add(panel, string):
                    string.panels.append(panel)
                    if self._check_system_limits():
                        return True
                    string.panels.pop()

            # Try starting a new string in this array
            new_str = SeriesString([panel])
            if self._check_string_limits(new_str) and
self._matches_mpp_voltage(array, new_str):
                array.strings.append(new_str)
                if self._check_system_limits():
                    return True
                array.strings.pop()

            # Try creating a brand-new array
            new_arr = ParallelArray([SeriesString([panel])])
            if not self._check_string_limits(new_arr.strings[0]):
                return False
            self.arrays.append(new_arr)
            if self._check_system_limits():
                return True
            self.arrays.pop()
            return False

    def _can_add(self, panel: SolarPanelNode, string: SeriesString) ->
bool:
        # current match + Voc + MPP-voltage both within
inverter.voltage_max
        return (
            abs(panel.mpp_amps - string.limiting_current()) <=
self.current_tol
            and string.total_voc() + panel.oc_voltage <=
self.inverter.voltage_max
            and string.total_mpp_voltage() + panel.mpp_voltage <=
self.inverter.voltage_max
        )

    def _check_string_limits(self, string: SeriesString) -> bool:
        return (
            string.total_voc() <= self.inverter.voltage_max
            and string.total_mpp_voltage() <= self.inverter.voltage_max
        )

    def _matches_mpp_voltage(self, array: ParallelArray, string:
SeriesString) -> bool:

```

```

ones
        # ensure new string's MPP voltage is within tolerance of existing
        if not array.strings:
            return True
        avg_v = sum(s.total_mpp_voltage() for s in array.strings) /
len(array.strings)
        return abs(avg_v - string.total_mpp_voltage()) <=
self.voltage_tol

def _check_system_limits(self) -> bool:
    all_strings = [s for arr in self.arrays for s in arr.strings]
    total_power = sum(s.effective_power() for s in all_strings)
    total_current = sum(s.limiting_current() for s in all_strings)
    # global inverter constraints
    return (
        total_power <= self.inverter.max_power
        and total_current <= self.inverter.amps_max
    )

def _backtrack(self, failed_panel: SolarPanelNode):
    # rebuild from scratch in descending MPP power order
    all_panels = [p for arr in self.arrays for s in arr.strings for
p in s.panels]
    all_panels += self.unassigned + [failed_panel]
    self.arrays.clear()
    self.unassigned.clear()
    self.grow_tree(sorted(all_panels, key=lambda p: p.max_power,
reverse=True))

# word summary (for testing)
def summary(self):
    print(f"Built {len(self.arrays)} parallel array(s).")
    tempV = []
    for i, arr in enumerate(self.arrays, 1):
        print(f"Array {i}:")
        for j, s in enumerate(arr.strings, 1):
            ids = [p.ui_id for p in s.panels]
            print(
                f"String {j}: Panels {ids} → "
                f"Voc={s.total_voc():.1f}V, "
                f"MPP_V={s.total_mpp_voltage():.1f}V, "
                f"I={s.limiting_current():.2f}A, "
                f"P≈{s.effective_power():.1f}W"
            )
            self.total_current += s.limiting_current()
            tempV.append(s.total_mpp_voltage())
        self.total_voltage = min(tempV)
        self.total_power = self.total_voltage * self.total_current
        print(f"Total stats: {self.total_power} W, {self.total_voltage}
V, {self.total_current} A")
    if self.unassigned:
        print("Unassigned panels:", [p.ui_id for p in
self.unassigned])

# build bigtree
def construct_tree(self):
    for array_item in self.arrays:
        for string_item in array_item.strings:
            self.inverter >> string_item.panels[0]
            for item, nextitem in zip(string_item.panels,
string_item.panels[1:]):

```

```
item >> nextitem
```

```
---DataTreeNodes.py---
```

```
from dataclasses import dataclass
from bigtree import Node
from data.SolarDataclasses import SolarPanel, Inverter

class SolarPanelNode(Node, SolarPanel):
    def __init__(self, ui_id: int, max_power: float, mpp_voltage: float,
mpp_amps: float, oc_voltage: float):
        super().__init__(str(ui_id))
        self.ui_id = ui_id # unique id to link UI to data
representation. id assignment is handled by factory.
        self.max_power = max_power
        self.mpp_voltage = mpp_voltage
        self.mpp_amps = mpp_amps
        self.oc_voltage = oc_voltage

class InverterNode(Node, Inverter):
    def __init__(self, ui_id: int, max_power: float,
voltage_min: float, voltage_max: float, amps_max:
float, autonomous: bool):
        super().__init__(str(ui_id))
        self.ui_id = ui_id # unique id to link UI to data
representation. id assignment is handled by factory.
        self.max_power = max_power
        self.voltage_min = voltage_min
        self.voltage_max = voltage_max
        self.amps_max = amps_max
        self.autonomous = autonomous

class NodeFactory:
    """
    An object that globally controls unique ID assignment for new
nodes/objects and creates new nodes

    Attributes:
        global_current_id - keeps track of incremental id assignment
    Returns:
        SolarPanelNode or InverterNode via get_new_node()
    """
    def __init__(self):
        self.global_current_id = 0

    def get_new_node(self, kind: str, **kwargs):
        if kind == "panel":
            self.global_current_id += 1
            return SolarPanelNode(self.global_current_id,
                                   kwargs["max_power"],
                                   kwargs["mpp_voltage"],
                                   kwargs["mpp_amps"],
                                   kwargs["oc_voltage"])
        elif kind == "inverter":
            self.global_current_id += 1
            return InverterNode(self.global_current_id,
                                 kwargs["max_power"],
                                 kwargs["voltage_min"],
                                 kwargs["voltage_max"],
```

```

        kwargs["amps_max"],
        kwargs["autonomous"])
    else:
        raise

@dataclass()
class AutoTree:
    """
    Auto tree generated by the heuristic algorithm out of provided
    elements.
    """
    root: InverterNode

---SolarDataClasses.py---
# This Python file uses the following encoding: utf-8
from dataclasses import dataclass

@dataclass
class Inverter:
    """
    A class representing the physical side of an electric inverter

    Attributes:
        max_power: float
            Maximum power rating of inverter.
        voltage_min: float
            Minimum required voltage to start working.
        voltage_max: float
            Maximum supported incoming voltage of solar power
generation.
        amps_max: float
            Maximum supported current of solar power generation.
        autonomous: bool
            Whether the inverter is able to draw additional power from
the external power grid
            in case of low power generation (determines if voltage_min
parameter is soft or hard minimum).
    """
    max_power: float
    voltage_min: float
    voltage_max: float
    amps_max: float
    autonomous: bool

@dataclass
class SolarPanel:
    """
    A class representing the physical side of a single solar panel

    Attributes:
        max_power: float
            Maximum power rating of the solar panel.
        mpp_voltage: float
            Maximum Power Point (working) voltage.
        mpp_amps: float
            Maximum Power Point (working) current.
        oc_voltage: float

```

Open Circuit voltage of a panel with no load. For connection, this is used instead of mpp_voltage.

```
"""
max_power: float
mpp_voltage: float
mpp_amps: float
oc_voltage: float
```

---CreateInverterDialog.py---

```
from PySide6.QtWidgets import QDialog, QDialogButtonBox, QMessageBox
from ui.ui_CreateInverterDialog import Ui_CreateInverterDialog
from PySide6.QtCore import Signal
```

```
class CreateInverterDialog(QDialog, Ui_CreateInverterDialog):
    submit_form = Signal(dict) # new signal to submit dialog form data

    def __init__(self):
        super().__init__()
        self.setupUi(self)
        # override default connection
        self.buttonBox.accepted.disconnect()
        self.buttonBox.rejected.disconnect()
        # set custom connection
```

```
self.buttonBox.button(QDialogButtonBox.StandardButton.Ok).clicked.connect(self.accept_and_emit)
```

```
self.buttonBox.button(QDialogButtonBox.StandardButton.Cancel).clicked.connect(self.rejected_close)
```

```
    def accept_and_emit(self):
        try:
            power = float(self.power_value.text().strip())
            min_voltage = float(self.min_voltage_value.text().strip())
            max_voltage = float(self.max_voltage_value.text().strip())
            max_current = float(self.max_current_value.text().strip())
            auto = self.autonomous.isChecked()
            form_data = {
                "kind": "inverter",
                "max_power": power,
                "voltage_min": min_voltage,
                "voltage_max": max_voltage,
                "amps_max": max_current,
                "autonomous": auto
            }
            self.submit_form.emit(form_data)
            self.accept()
        except:
            QMessageBox.critical(self, "Invalid Input", "All fields must be filled correctly with numerical data.")
            return
```

```
    def rejected_close(self):
        self.reject()
```

---CreatePanelDialog.py---

```
from PySide6.QtWidgets import QDialog, QMessageBox, QDialogButtonBox
from ui.ui_CreatePanelDialog import Ui_CreatePanelDialog
from PySide6.QtCore import Signal
```

```

class CreatePanelDialog(QDialog, Ui_CreatePanelDialog):
    submit_form = Signal(dict) # new signal to submit dialog form data

    def __init__(self):
        super().__init__()
        self.setupUi(self)
        # override default connection
        self.buttonBox.accepted.disconnect()
        self.buttonBox.rejected.disconnect()
        # set custom connection

self.buttonBox.button(QDialogButtonBox.StandardButton.Ok).clicked.connect(se
lf.accept_and_emit)

self.buttonBox.button(QDialogButtonBox.StandardButton.Cancel).clicked.conne
ct(self.rejected_close)

    def accept_and_emit(self):
        try:
            power = float(self.power_value.text().strip())
            voltage = float(self.voltage_value.text().strip())
            current = float(self.current_value.text().strip())
            ocv = float(self.ocv_voltage_value.text().strip())
            form_data = {
                "kind": "panel",
                "max_power": power,
                "mpp_voltage": voltage,
                "mpp_amps": current,
                "oc_voltage": ocv
            }
            self.submit_form.emit(form_data)
            self.accept()
        except:
            QMessageBox.critical(self, "Invalid Input", "All fields must
be filled correctly with numerical data.")
            return

    def rejected_close(self):
        self.reject()

---DataWidget.py---
from PySide6.QtCore import QSize, Qt
from PySide6.QtGui import QPixmap, QIcon
from PySide6.QtWidgets import QWidget
from ui.ui_DataWidget import Ui_DataWidget

class DataWidget(QWidget, Ui_DataWidget):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.realObjectInFocus = False # true = shows info about object,
false = shows info about entire system
        self.focusObjectId = -1

    def updateData(self, datadict: dict):
        if datadict["kind"] == "panel":
            self.focusObjectId = datadict["id"]
            self.realObjectInFocus = True

```

```

self.object_picture_label.setPixmap(QPixmap(":/res/solar_panel.png"))

.scaled(self.object_name_label.size(),

aspectMode=Qt.AspectRatioMode.KeepAspectRatioByExpanding))
    self.voltage_value_label.setText(str(datadict["voltage"]))
    self.current_value_label.setText(str(datadict["current"]))
    self.power_value_label.setText(str(datadict["power"]))
    self.object_name_label.setText(str(datadict["id"]))
elif datadict["kind"] == "inverter":
    self.focusObjectId = datadict["id"]
    self.realObjectInFocus = True

self.object_picture_label.setPixmap(QPixmap(":/res/electric_invertor.png"))

.scaled(self.object_name_label.size(),

aspectMode=Qt.AspectRatioMode.KeepAspectRatioByExpanding))
    self.voltage_value_label.setText(str(datadict["voltage"]))
    self.current_value_label.setText(str(datadict["current"]))
    self.power_value_label.setText(str(datadict["power"]))
    self.object_name_label.setText(str(datadict["id"]))
elif datadict["kind"] == 0:
    self.voltage_value_label.setText("0")
    self.power_value_label.setText("0")
    self.current_value_label.setText("0")
    self.focusObjectId = -1
    self.realObjectInFocus = False
    self.object_picture_label.clear()
    self.object_name_label.clear()
elif datadict["kind"] == "tree":
    self.realObjectInFocus = False
    self.object_picture_label.clear()
    self.voltage_value_label.setText(str(datadict["voltage"]))
    self.current_value_label.setText(str(datadict["current"]))
    self.power_value_label.setText(str(datadict["power"]))
    self.object_name_label.setText("Connection Tree")
else:
    raise

---InventoryWidget.py---
from PySide6.QtCore import Signal, Qt
from PySide6.QtGui import QIcon
from PySide6.QtWidgets import QWidget, QListWidgetItem, QMessageBox
from ui.ui_InventoryWidget import Ui_InventoryWidget
from data.DataTreeNodes import NodeFactory, SolarPanelNode,
InverterNode, AutoTree
from data.AutoTreeBuilder import AutoTreeBuilder
from bigtree import print_tree, tree_to_dot

class InventoryWidget(QWidget, Ui_InventoryWidget):
    submit_item_data = Signal(dict) # signal to submit clicked item
data
    tree_updated = Signal() # signal emitted to signify that the tree
was updated

    def __init__(self):
        super().__init__()
        self.setupUi(self)

```

```

        self.item_factory = NodeFactory()
        self.dataInventoryList = []
        self.listWidget.itemClicked.connect(self.get_item_info)
        self.auto_tree = None
        self.auto_tree_bulder = None

    # add new item to list
    def new_item(self, signal_data):
        newNode = self.item_factory.get_new_node(**signal_data)
        self.dataInventoryList.append(newNode)
        self.add_to_inventory(newNode)

    # visualize item from list to inventory
    def add_to_inventory(self, item: SolarPanelNode | InverterNode):
        newItem = QListWidgetItem()
        newItem.setText(str(item.ui_id))
        if type(item) == SolarPanelNode:
            newItem.setIcon(QIcon(":res/solar_panel.png"))
        elif type(item) == InverterNode:
            newItem.setIcon(QIcon(":res/electric_invertor.png"))
        self.listWidget.addItem(newItem)

    def get_item_info(self, item: QListWidgetItem):
        uid = int(item.text().strip()) # get id from name str
        node_obj = next((x for x in self.dataInventoryList if x.ui_id
== uid), None) # find object by id
        if node_obj is not None:
            if type(node_obj) == SolarPanelNode:
                voltage = node_obj.mpp_voltage
                power = node_obj.max_power
                current = node_obj.mpp_amps
                data_package = {
                    "id": uid,
                    "kind": "panel",
                    "voltage": voltage,
                    "power": power,
                    "current": current
                }
                self.submit_item_data.emit(data_package)
            elif type(node_obj) == InverterNode:
                voltage = node_obj.voltage_max
                power = node_obj.max_power
                current = node_obj.amps_max
                data_package = {
                    "id": uid,
                    "kind": "inverter",
                    "voltage": voltage,
                    "power": power,
                    "current": current
                }
                self.submit_item_data.emit(data_package)
            else:
                raise
        else:
            raise

    def full_item_remove(self, ui_id: int):
        # remove from list
        object_in_list = next((x for x in self.dataInventoryList if
x.ui_id == ui_id), None) # find object by id

```

```

        if self.auto_tree is not None and object_in_list.ui_id ==
self.auto_tree.root.ui_id: # if deleted root Inverter
            self.auto_tree = None
            self.tree_updated.emit()
            self.dataInventoryList.remove(object_in_list)
            # remove from UI
            listWidget_item = self.listWidget.findItems(str(ui_id),
Qt.MatchFlag.MatchExactly)[0]
            self.listWidget.takeItem(self.listWidget.currentRow())
            self.listWidget.clearSelection()
            self.tree_updated.emit()
            self.submit_item_data.emit({"kind": 0})

def remove_all_items(self):
    # clear list
    self.dataInventoryList = []
    # clear UI
    self.listWidget.clear()
    # reset indexing
    self.item_factory.global_current_id = 0
    # set data to base
    self.auto_tree = None
    self.tree_updated.emit()
    self.submit_item_data.emit({"kind": 0})

def auto_build_tree(self):
    suitable_roots = []
    for item in self.dataInventoryList: # get all invertor nodes
        if type(item) == InverterNode:
            suitable_roots.append(item)
    suitable_roots.sort(key=lambda x: x.max_power, reverse=True)
    if len(suitable_roots) == 0: # no inverter
        QMessageBox.warning(self, "No Inverter", "You have no
inverter in the system!")
        self.auto_tree = None
        self.tree_updated.emit()
        return
    suitable_root: InverterNode = suitable_roots[0] # get the
highest power invertor
    panels_list = []
    for panelcan in self.dataInventoryList: # get all panel nodes
        if type(panelcan) == SolarPanelNode:
            panels_list.append(panelcan)
    builder = AutoTreeBuilder(suitable_root)
    try:
        builder.grow_tree(panels_list)
        builder.summary()
        builder.construct_tree()
        self.auto_tree = AutoTree(suitable_root)
        self.auto_tree_bulder = builder
        print_tree(self.auto_tree.root)
        graph = tree_to_dot(self.auto_tree.root,
node_colour="gold")
        graph.write_png("ui/res/tree.png")
        self.tree_updated.emit() # emit tree update
    except RecursionError:
        QMessageBox.critical(self, "Bad Layout", "Could not
automatically generate with these items.")
        self.auto_tree = None
        self.tree_updated.emit()
        return

```

```

def get_sys_stats(self):
    if self.auto_tree_bulder is not None:
        data_package = {
            "id": "tree",
            "kind": "tree",
            "voltage": self.auto_tree_bulder.total_voltage,
            "power": self.auto_tree_bulder.total_power,
            "current": self.auto_tree_bulder.total_current
        }
        self.submit_item_data.emit(data_package)
    else:
        self.submit_item_data.emit({"kind": 0})

---MainWindow.py---
from PySide6.QtCore import Qt
from PySide6.QtGui import QPixmap
from PySide6.QtWidgets import QMainWindow, QApplication, QDockWidget,
QGraphicsPixmapItem, QGraphicsScene
from ui.ui_MainWindow import Ui_MainWindow
from ui.widgets.CreatePanelDialog import CreatePanelDialog
from ui.widgets.CreateInverterDialog import CreateInverterDialog
from ui.widgets.DataWidget import DataWidget
from ui.widgets.InventoryWidget import InventoryWidget
from PySide6.QtGui import QMouseEvent

class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self, app: QApplication):
        super().__init__()
        self.dockwidget1 = None
        self.dockwidget2 = None
        self.setupUi(self)
        self.app = app
        self.lastFocusedObject = self

        self.actionExit.triggered.connect(self.quit)

self.actionRestore_windows.triggered.connect(self.view_restore)

self.actionAdd_Panel.triggered.connect(self.open_create_panel_dialog)

self.actionAdd_Inverter.triggered.connect(self.open_create_inverter_dialog)
self.actionDelete_Selected.triggered.connect(self.remove_item)

    def mousePressEvent(self, event: QMouseEvent, /) -> None:
        if self.dockwidget1 is not None and
self.dockwidget1.widget().realObjectInFocus is True:
            self.dockwidget2.widget().get_sys_stats()

    def set_dockwidgets(self, data: DataWidget, inventory:
InventoryWidget):
        self.dockwidget1 = QDockWidget()
        self.dockwidget1.setWidget(data)
        self.dockwidget1.setWindowTitle("Object Information")
        self.dockwidget2 = QDockWidget()
        self.dockwidget2.setWidget(inventory)
        self.dockwidget2.setWindowTitle("Object Inventory")

```

```

        self.addDockWidget(Qt.DockWidgetArea.RightDockWidgetArea,
self.dockwidget1)
        self.addDockWidget(Qt.DockWidgetArea.BottomDockWidgetArea,
self.dockwidget2)
        inventory.submit_item_data.connect(data.updateData)

self.actionDelete_all.triggered.connect(inventory.remove_all_items)

self.actionAuto_Connect.triggered.connect(inventory.auto_build_tree)
        inventory.tree_updated.connect(self.updateTreeImage)

    def updateTreeImage(self):
        if self.dockwidget2 is not None:
            gfxScene = QGraphicsScene(self.graphicsView)
            self.graphicsView.setScene(gfxScene)
            if self.dockwidget2.widget().auto_tree is not None:
                treepixmap = QPixmap("ui/res/tree.png")
                if treepixmap.isNull():
                    return
                gfxPixmap = QGraphicsPixmapItem(treepixmap)
                gfxScene.addItem(gfxPixmap)
                self.graphicsView.setScene(gfxScene)
            else:
                gfxScene.clear()
            # todo:

    def quit(self):
        self.app.quit()

    def view_restore(self):
        if self.dockwidget1 is not None:
            self.dockwidget1.show()
        if self.dockwidget2 is not None:
            self.dockwidget2.show()

    def open_create_panel_dialog(self):
        self.panel_dialog = CreatePanelDialog()
        if self.dockwidget2 is not None:

self.panel_dialog.submit_form.connect(self.dockwidget2.widget().new_item)
        self.panel_dialog.exec()

    def open_create_inverter_dialog(self):
        self.inverter_dialog = CreateInverterDialog()
        if self.dockwidget2 is not None:

self.inverter_dialog.submit_form.connect(self.dockwidget2.widget().new_item)
        self.inverter_dialog.exec()

    def remove_item(self):
        if self.dockwidget1 is not None and
self.dockwidget1.widget().focusObjectId != -1:
            if self.dockwidget2 is not None:

self.dockwidget2.widget().full_item_remove(self.dockwidget1.widget().focusObjectId
)

```




Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

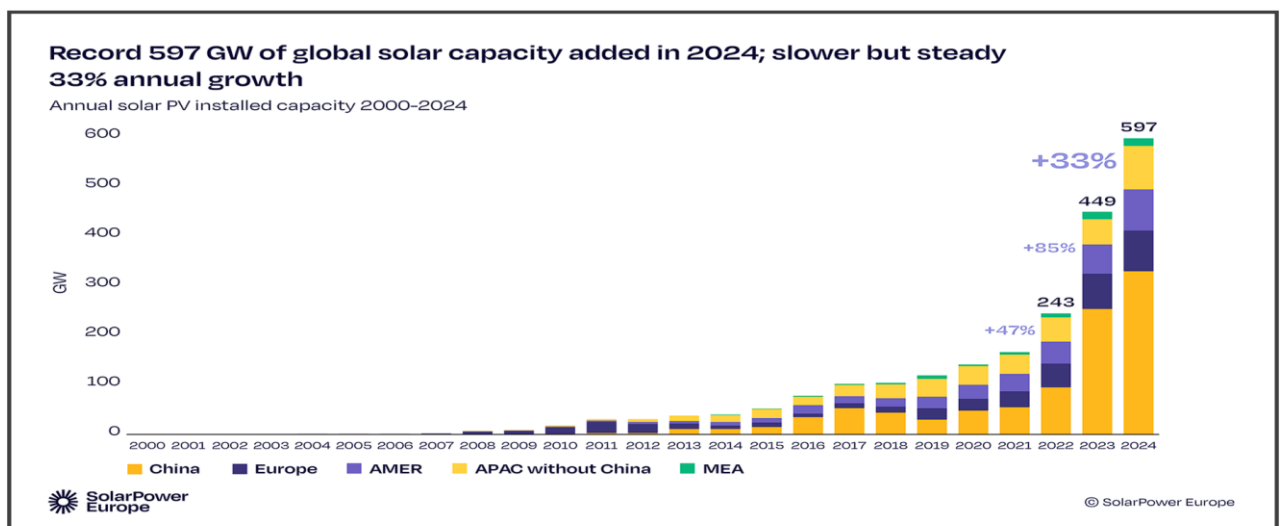
ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ОПТИМАЛЬНОЇ КОМУТАЦІЇ РІЗНОРІДНИХ СОНЯЧНИХ ПАНЕЛЕЙ В СОНЯЧНУ ЕЛЕКТРОСТАНЦІЮ

виконав: студент групи ТВ-11 Андрійчук Антон Юрійович

керівник: Барабаш Олег Володимирович, доктор технічних наук, професор

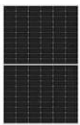
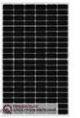
Київ - 2025

Актуальність теми



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Актуальність теми

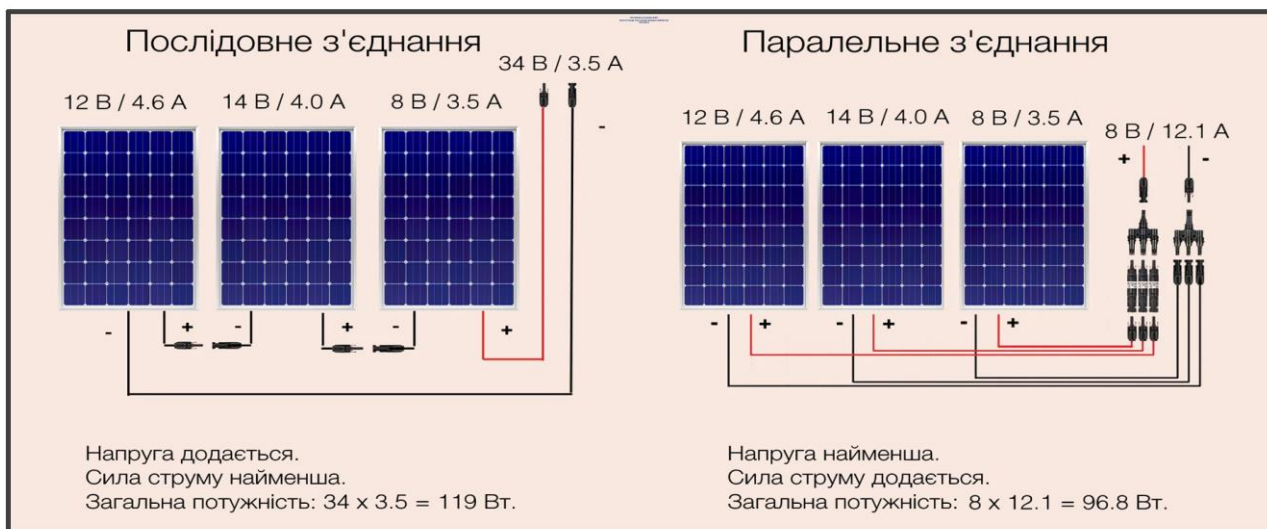
 Сонячна панель Longi Solar LR7-72HGD-600M, 600 Вт, Bifacial 3 943 ₴ В наявності Тільки оптом Купити	 Сонячна панель AXIOMA energy AX-80M 2 407 ₴ В наявності Купити	 Сонячна панель AXIOMA Energy AX-150M 4 150 ₴ В наявності Купити	 Сонячна батарея Tina Solar TSM-DE015H-410M, 999 Вт, 410W, Half Cell, Mono 4 731 ₴ ГД, замовлення Тільки оптом Купити	 Сонячна панель Longi Solar LR4-72H7H-425M 2 905 ₴ В наявності Тільки оптом Купити	 Сонячна панель Risen RSM120-B-950M, 950 Вт, Mono Tier1 9 670 ₴ ГД, замовлення Тільки оптом Купити
 Сонячна панель Tina Solar TSM-433NE-G96C-27, 433 Вт (Bifacial) 2 905 ₴ В наявності Тільки оптом Купити	 Сонячна панель Longi Solar LR5-72H1H-560M 4 150 ₴ Немає в наявності Тільки оптом Якщо потрібно, повідомте про це!	 Сонячна панель AXIOMA Energy AX-200M 4 980 ₴ Немає в наявності Якщо потрібно, повідомте про це!	 Фотомодуль Suntech Half-cell STP 285-204Wp 3 154 ₴ Немає в наявності Оптом і в роздріб Якщо потрібно, повідомте про це!	 Сонячна панель Ufesa Solar UL-260P-60 3 154 ₴ Немає в наявності Оптом і в роздріб Якщо потрібно, повідомте про це!	 Сонячна панель Amerisolar AS-6P-330 330W: 540 4 358 ₴ Немає в наявності Оптом і в роздріб Якщо потрібно, повідомте про це!



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

3

Актуальність теми



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

4

Актуальність теми

Цільовою аудиторією проекту є:

Ентузіасти сфери відновлювальної енергетики для особистих потреб

- Для них важливою вимогою є можливість створити найбільш ефективну систему із доступних можливостей

Бізнес, що впроваджує сонячні електростанції

- Вони можуть використовувати застосунок як інструмент при плануванні та обговоренні проекту, а також візуалізації для клієнтів.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

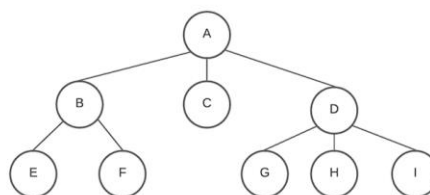
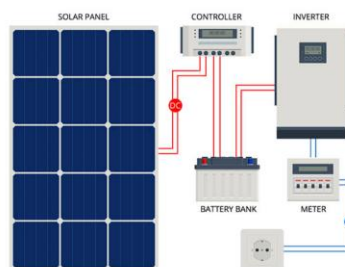
5

Постановка задачі

Мета роботи полягає у розробці програмного застосунку оптимальної комутації різномірних сонячних панелей в сонячну електростанцію.

Програмний застосунок повинен дозволяти:

- зручно вводити дані про наявні сонячні панелі,
- формувати оптимальні схеми підключення з урахуванням обмежень,
- наочно відображати отриману конфігурацію у вигляді дерева або графу.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

6

Постановка задачі

Часткові завдання, що передбачаються для реалізації проєкту:

- Вибір структури даних.
- Розробка алгоритму формування з'єднань.
- Визначення форм подання результату користувачеві.
- Вжити заходів щодо уникнення виключних ситуацій під час виконання програми.
- Розробка користувацького інтерфейсу.
- Тестування та валідація програмного застосунку.

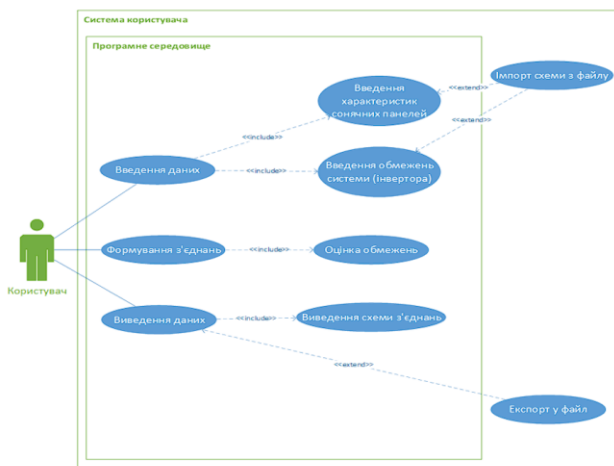


Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

7

Проектування системи

Use-case діаграма системи:



Вимоги до системи:

- Зручність введення
- Наочність виведення
- Оптимальний за критерієм максимуму ефективності сонячної станції (за принципом «good-enough») алгоритм формування з'єднань



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

8

Реалізація системи: вибір інструментів

Для розробки було обрано наступні інструменти:

- Мова програмування Python
- Фреймворк графічних інтерфейсів Qt для мови Python (представлено модулем Pyside6)
- Модуль bigtree для мови Python (побудова дерев/графів та їх візуального відображення)

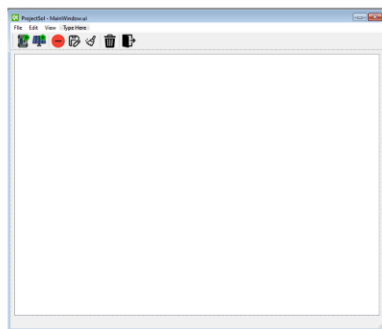


Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

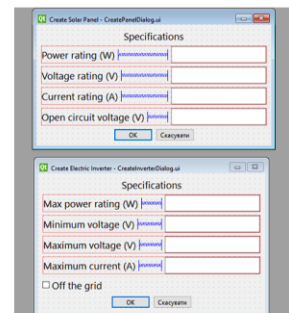
9

Проектування компонентів графічного інтерфейсу

Основу до графічного інтерфейсу у фреймворку Qt було створено за допомогою інструменту Qt Designer. Інструмент дозволяє, подібно до Windows Forms у .NET, створювати дизайн віджетів (складових частин графічного інтерфейсу). На основі цих дизайнів створюються класи віджетів в програмній реалізації.



Головне вікно



Деякі діалогові вікна

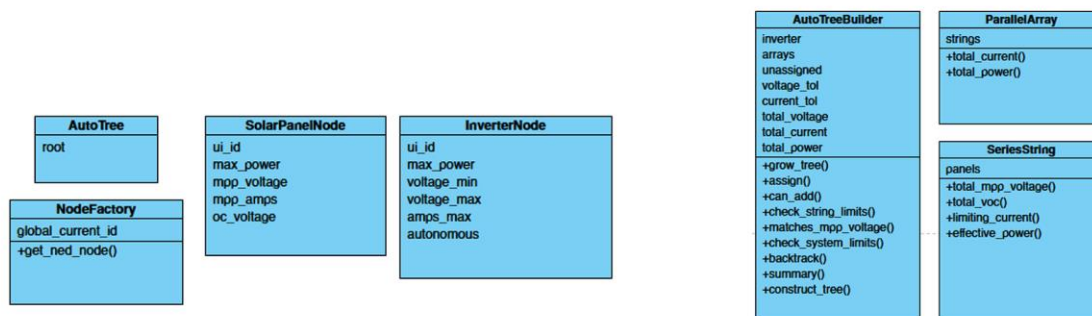


Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

10

Реалізація системи: математична складова

Для математичної складової було розроблено структуру даних, що містить класи, що обслуговують сонячні панелі, електричний інвертор та процес їх створення при виконанні застосунку. Також було розроблено класи, що обслуговують побудову і виведення дерева з'єднань. Нижче наведено деякі діаграми до вищеназваних класів.



Реалізація алгоритму

Для автоматичної побудови дерева з'єднань було розроблено змішаний жадібно-евристичний алгоритм. Принцип роботи полягає у поступовій побудові дерева за жадібним принципом із наступним поверненням на початок і оцінкою.

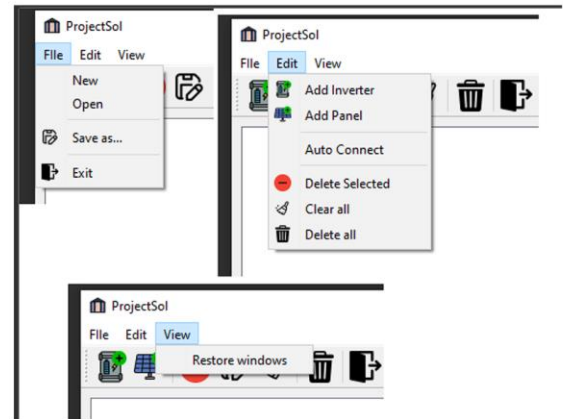
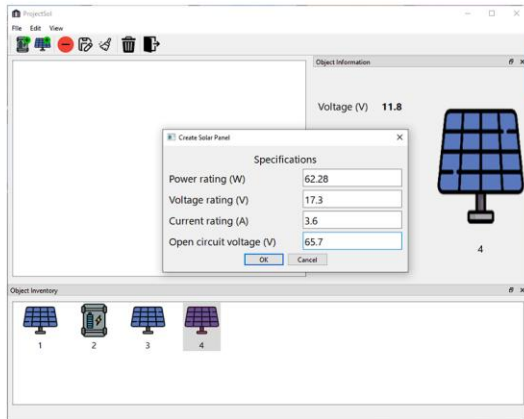
Після оцінки алгоритм впроваджує зміни для покращення ефективності до певного моменту, щоб система була достатньо оптимальною, не займаючи багато часу для побудови.

```
def grow_tree(self, panels: List[SolarPanelNode]):
    for panel in panels:
        if not self._assign(panel):
            self.unassigned.append(panel)
            self._backtrack(panel)
```



Реалізація системи: приклади роботи

Скріншоти роботи програми наведено нижче.

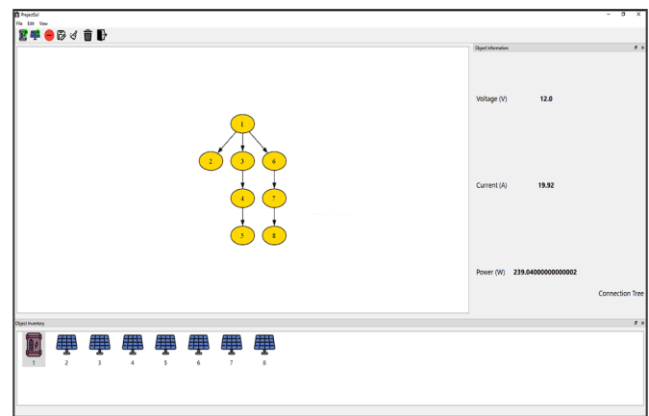
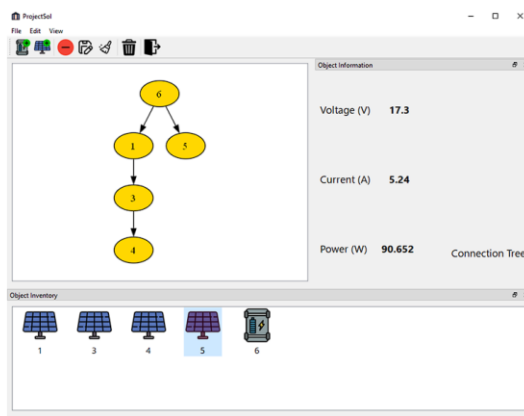


Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

13

Реалізація системи: приклади роботи

Приклади побудованих схем з'єднання панелей.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

14

Тестування системи

Під час розробки складові частини системи (особливо обчислювальної складової) модульно тестувалися (unit-тести) на введення та цілісність даних. Елементи графічного інтерфейсу тестувалися системним шляхом. Розроблена система проявила стійкість та відповідність вимогам. Нижче наведено приклад unit-тесту одного із модулів.

```
class MyTestCase(unittest.TestCase):
    def test_assign(self):
        p = 120.3333
        v = 12.35
        a = 5.66
        ocv = 15.659
        testobj1 = SolarPanelNode(1, p, v, a, ocv)
        self.assertEqual(testobj1.max_power, p)
        self.assertEqual(testobj1.mpp_voltage, v)
        self.assertEqual(testobj1.mpp_amps, a)
        self.assertEqual(testobj1.oc_voltage, ocv)

def test_node_connections(self):
    p = 156.3333
    v = 55.35
    a = 34.66
    ocv = 80.659
    testobj = SolarPanelNode(75, p, v, a, ocv)
    self.assertIsNone(testobj.parent)
    self.assertEqual(testobj.children, ())
```



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

15

Впровадження та супровід



ІНТЕГРАЦІЯ
ІЗ ХМАРОЮ

ІНТЕГРАЦІЯ З
БАЗАМИ ДАНИХ

QUALITY OF LIFE



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

16

Висновки

Всі попередньо поставлені задачі було виконано в процесі розробки, а саме:

- Вибрано і розроблено структуру даних,
- Розроблено алгоритм формування з'єднань,
- Розроблено графічний інтерфейс,
- Враховано обмеження системи
- Впроваджені заходи стосовно уникнення виключних ситуацій.

Проект було протестовано та налагоджено до відповідності поставленим вимогам.

Результатом є кросплатформний програмний застосунок оптимальної комутації сонячних панелей в сонячну електростанцію. Застосунок готовий до впровадження та супроводу.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

17



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"