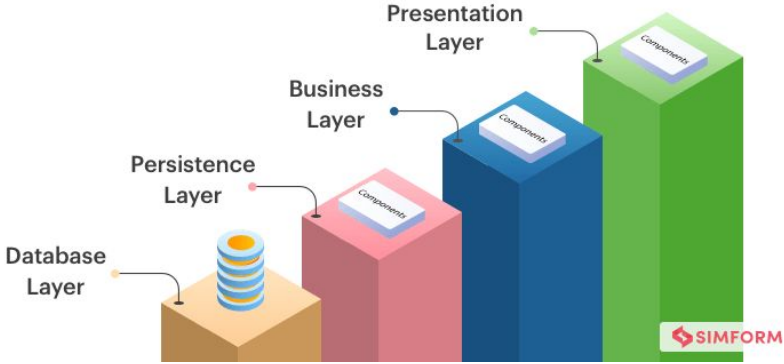
Архітектурні шаблони проектування. MVC, MVP та MVVM

Архітектурні шаблони програмного проектування

Це підходи до організації коду, які допомагають:

1. **Розділяти відповідальність** (логіка, інтерфейс, дані)
2. **Покращувати підтримку та масштабування**
3. **Спрощувати тестування**
4. **Уникати хаосу в коді**



Для чого використовуються?

- **MVC, MVP, MVVM** — це шаблони для розробки **UI** (користувацьких інтерфейсів).
- Вони допомагають відокремити бізнес-логіку від відображення даних.
- Використовуються у **веб-додатках, мобільних та десктопних програмах**.

Основна ідея - поділ коду на частини, щоб зміни в одній не ламали інші.

Трохи історії

MVC: Smalltalk (1970-ті)

- **Створено** Трюґве Ріанес-Кнудсеном та Адельєю Голдберг (Xerox PARC).
- **Мета**: Упорядкувати код у **графічних редакторах** для комп’ютера Xerox Alto.
- **Еволюція**:
  - 1980-1990 — адаптація у **Desktop-додатках** (Java Swing).
  - 2000-ні — популярність у **веб-фреймворках** (Ruby on Rails, Django).

MVP: Відповідь на проблеми MVC (1990-ті)

- **Поява**: У відповідь на **тісний зв’язок View і Controller** у MVC.
- **Авторство**: Ідею популяризували Microsoft у **Windows Forms** та **Win32 API**.
- **Головна відмінність**:
  - View взаємодіє **тільки з Presenter**, а не з Model.

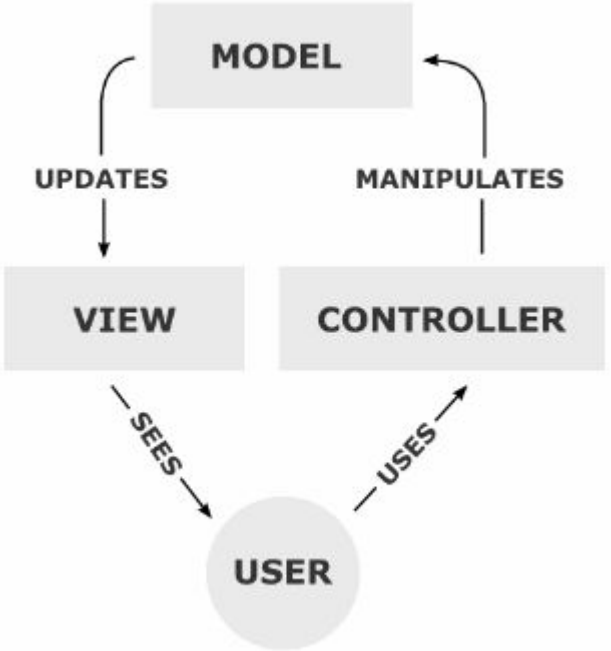
MVVM: Ера сучасних UI (2005+)

- **Створено** Джоном Госсманом для **WPF (Windows Presentation Foundation)**.
- **Ключова іновація**: **Data Binding** — автоматична синхронізація View і ViewModel.
- **Розширення**:
  - 2010-ні — адаптація у **JavaScript-фреймворках** Angular
  - 2020-ні — стандарт для **Android (Jetpack Compose)** та **iOS (SwiftUI)**.

MVC (Model-View-Controller)

- це **архітектурний шаблон**, який розділяє програму на **3 основні компоненти**:

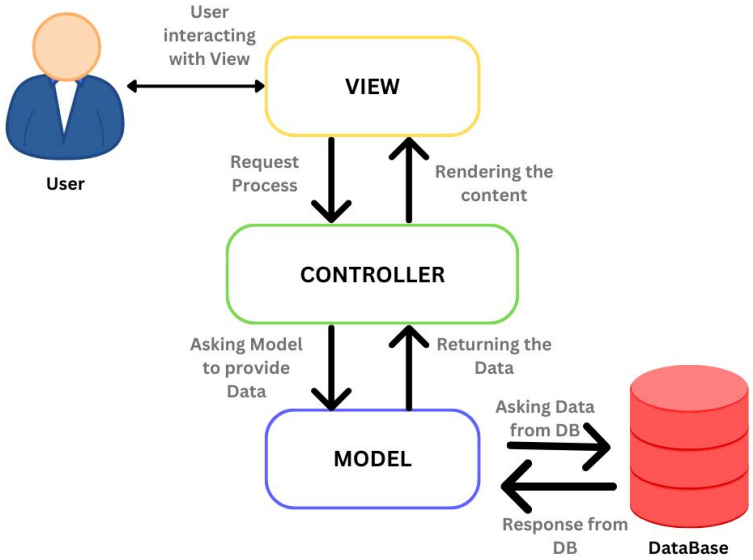
- **Model (Модель)** — відповідає за дані та бізнес-логіку.
- **View (Представлення)** — відображає інформацію для користувача.
- **Controller (Контролер)** — обробляє введення користувача та керує потоком даних.



Як працює MVC?

Послідовність дій:

1. **Користувач** взаємодіє з **View** (натискає кнопку, вводить текст).
2. **View** передає подію **Controller**.
3. **Controller** вирішує, що робити:
   - Звертається до **Model** (наприклад, додає дані в базу).
   - Оновлює **View** (показує нові дані).
4. **Model** зберігає/змінює дані та повідомляє **View** про зміни.



Схематично: Користувач → View → Controller → Model → View → Користувач

Особливості MVC

Переваги:

1. **Поділ відповідальностей** (легше знайти помилку).
2. **Підходить для вебу** (Django, ASP.NET MVC).
3. **Можна використовувати різні View** (наприклад, для мобільного додатку).

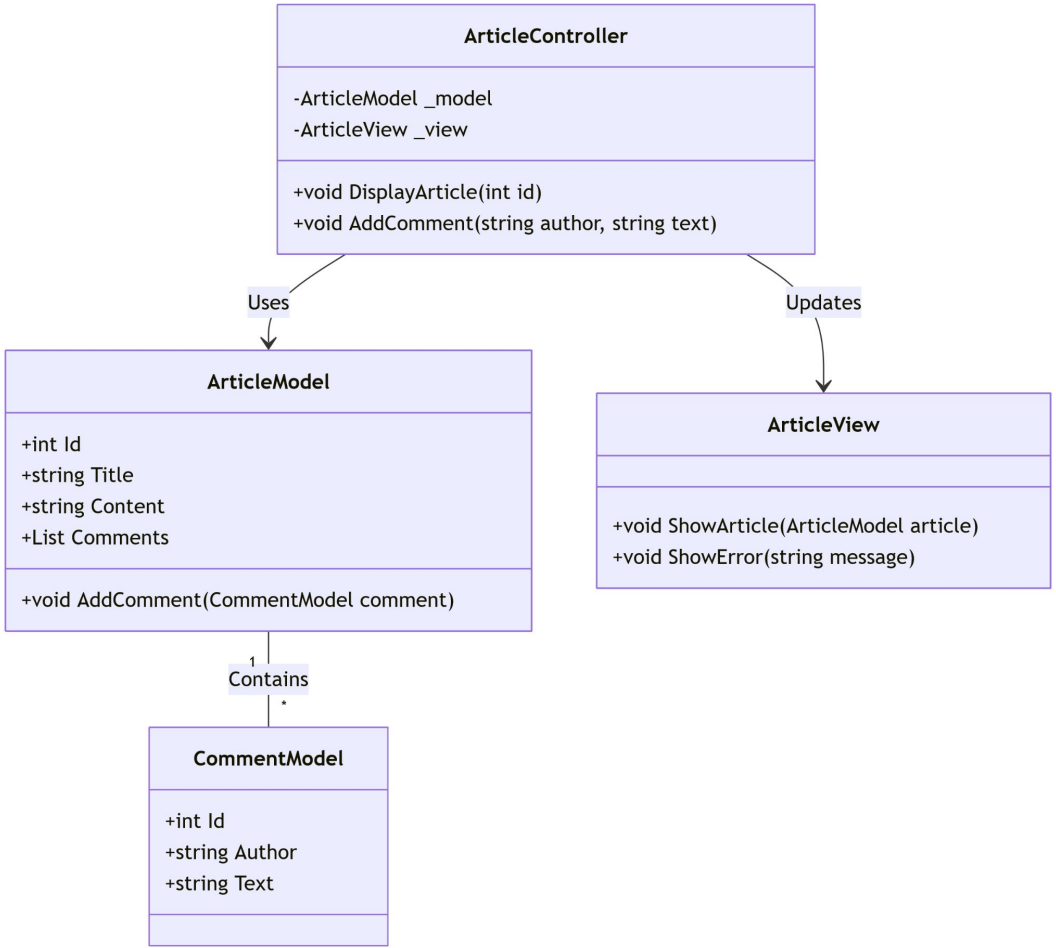
Недоліки

1. **Контролер може стати "роздутим"** (якщо в ньому забагато логіки).
2. **View і Controller іноді тісно пов’язані** (ускладнює тестування).
3. **Не ідеальний для складних UI**

Де використовується MVC?

- **Веб-фреймворки**: Ruby on Rails, Laravel, Spring MVC.
- **Десктопні додатки**: Java Swing, .NET WinForms (з модифікаціями).
- **Ігри**: Unity.

Приклад: веб-додаток для блогу (статті + коментарі)



```
// Модель статті
3 references
public class ArticleModel
{
    1 reference
    public int Id { get; set; }
    2 references
    public string Title { get; set; }
    2 references
    public string Content { get; set; }
    3 references
    public List<CommentModel> Comments { get; } = new List<CommentModel>();

    1 reference
    public void AddComment(CommentModel comment)
    {
        Comments.Add(comment);
    }
}

// Модель коментаря
4 references
public class CommentModel
{
    1 reference
    public int Id { get; set; }
    1 reference
    public string Author { get; set; }
    1 reference
    public string Text { get; set; }
}
```

```
public class ArticleView
{
    // Відображення статті з коментарями
    2 references
    public void ShowArticle(ArticleModel article)
    {
        Console.WriteLine($"\n--- {article.Title} ---");
        Console.WriteLine(article.Content);
        Console.WriteLine("\nКоментарі:");

        foreach (var comment in article.Comments)
        {
            Console.WriteLine($"- {comment.Author}: {comment.Text}");
        }
    }

    // Відображення помилки
    1 reference
    public void ShowError(string message)
    {
        Console.WriteLine($"Помилка: {message}");
    }

    // Отримання введення користувача
    0 references
    public (string author, string text) GetCommentInput()
    {
        Console.Write("Ваше ім'я: ");
        string author = Console.ReadLine();
        Console.Write("Текст коментаря: ");
        string text = Console.ReadLine();
        return (author, text);
    }
}
```

```
public class ArticleController
{
    8 references
    private readonly ArticleModel _model;
    4 references
    private readonly ArticleView _view;

    0 references
    public ArticleController(ArticleModel model, ArticleView view)
    {
        _model = model;
        _view = view;
    }

    // Завантаження та відображення статті
    0 references
    public void DisplayArticle(int id)
    {
        try
        {
            // На практиці тут буде запит до бази даних
            _model.Id = id;
            _model.Title = "Що таке MVC?";
            _model.Content = "MVC - це архітектурний шаблон...";

            _view.ShowArticle(_model);
        }
        catch (Exception ex)
        {
            _view.ShowError(ex.Message);
        }
    }

    // Додавання коментаря
    0 references
    public void AddComment(string author, string text)
    {
        var comment = new CommentModel
        {
            Id = _model.Comments.Count + 1,
            Author = author,
            Text = text
        };
        _model.AddComment(comment);
        _view.ShowArticle(_model); // Оновлюємо відображення
    }
}
```

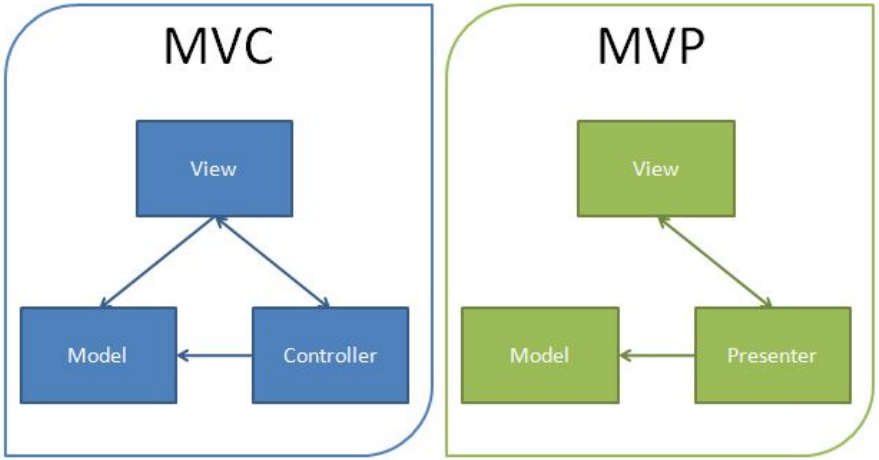
Висновки

1. MVC ідеально підходить для **структурованих додатків** з чіткою логікою
2. Кожен компонент має **одну відповідальність**
3. Патерн легко **адаптується** під різні інтерфейси (консоль, веб, мобільні додатки)

MVP (Model-Viev-Presenter)

MVP — це архітектурний шаблон, що **розширює ідеї MVC** для більш ефективної роботи з користувацькими інтерфейсами. Він складається з трьох компонентів:

- **Model** (Модель) — відповідає за дані та бізнес-логіку.
- **View** (Представлення) — відображає інтерфейс і *делегує* обробку подій Presenter'у.
- **Presenter** (Посередник) — керує взаємодією між View та Model.

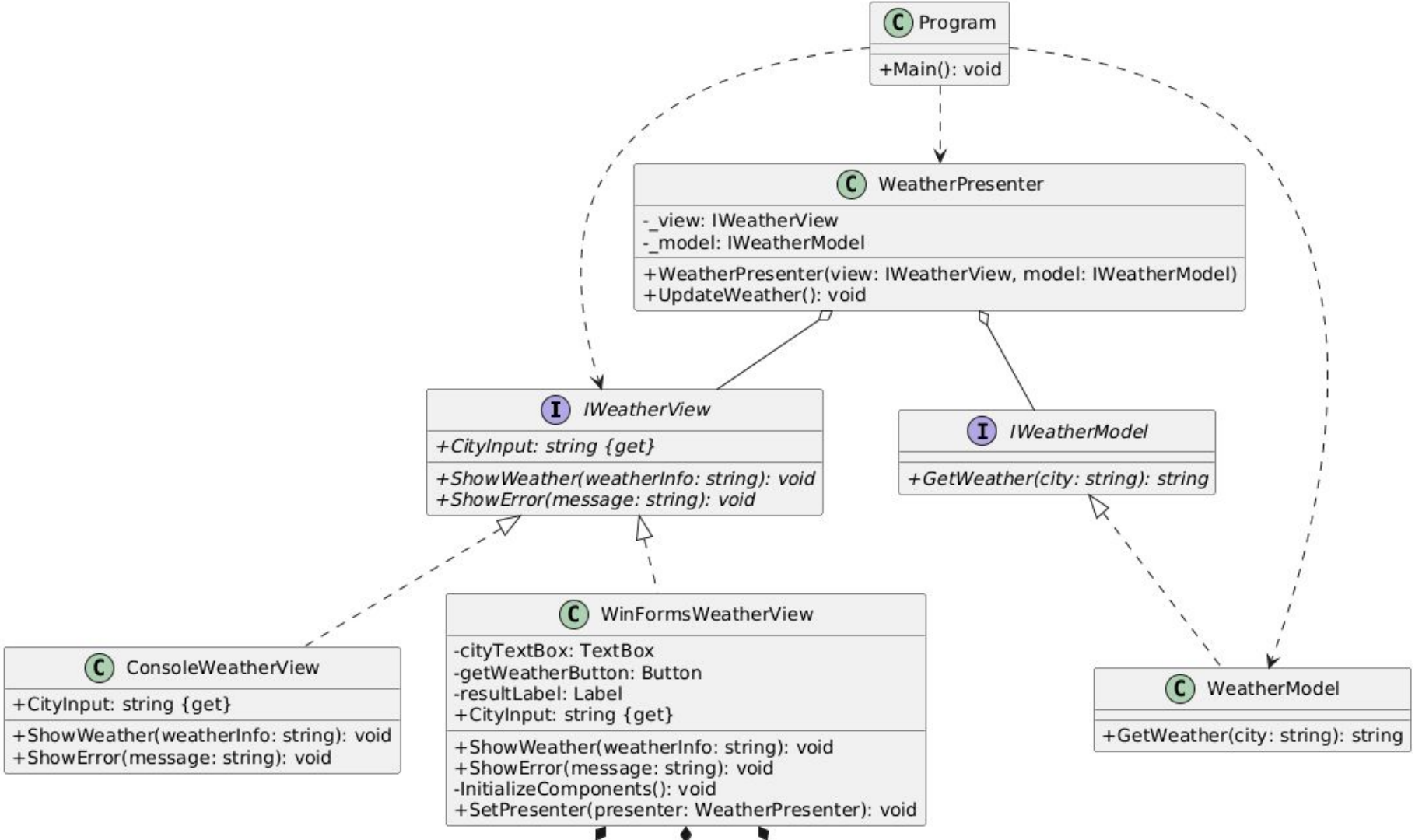


Ключова відмінність від MVC:

- View **не знає** про Model (на відміну від MVC, де View може звертатися до Model).
- Presenter виконує роль "посередника", що робить код **більш тестованим**.

Приклад: додаток для перегляду погоди, де:

- Model отримує дані про погоду
- View відображає інтерфейс і передає події
- Presenter керує логікою


Program
+Main(): void
WeatherPresenter
-_view: IWeatherView
-_model: IWeatherModel
+WeatherPresenter(view: IWeatherView, model: IWeatherModel)
+UpdateWeather(): void
IWeatherView
+CityInput: string {get}
+ShowWeather(weatherInfo: string): void
+ShowError(message: string): void
IWeatherModel
+GetWeather(city: string): string
ConsoleWeatherView
+CityInput: string {get}
+ShowWeather(weatherInfo: string): void
+ShowError(message: string): void
WinFormsWeatherView
-cityTextBox: TextBox
-getWeatherButton: Button
-resultLabel: Label
+CityInput: string {get}
+ShowWeather(weatherInfo: string): void
+ShowError(message: string): void
-InitializeComponents(): void
+SetPresenter(presenter: WeatherPresenter): void
WeatherModel
+GetWeather(city: string): string

Ключові особливості цього прикладу:

1. **Чіткий поділ відповідальностей**:
   - Model знає тільки про дані
   - View знає тільки про відображення
   - Presenter керує логікою
2. **View абсолютно пасивне**:
   - Не викликає методи Model напряму
   - Всі дії делегуються Presenter'у
3. **Легка заміна View**:
   - Можна створити WebView, MobileView тощо
   - Presenter залишається незмінним
4. **Просте тестування**:
   - Можна легко замокати View для тестів Presenter'a
5. **Підтримка різних інтерфейсів**:
   - Консольний і графічний інтерфейси використовують один Presenter

Ключові особливості цього прикладу:

1. **Чіткий поділ відповідальностей**:
   - Model знає тільки про дані
   - View знає тільки про відображення
   - Presenter керує логікою
2. **View абсолютно пасивне**:
   - Не викликає методи Model напряму
   - Всі дії делегуються Presenter'у
3. **Легка заміна View**:
   - Можна створити WebView, MobileView тощо
   - Presenter залишається незмінним
4. **Просте тестування**:
   - Можна легко замокати View для тестів Presenter'a
5. **Підтримка різних інтерфейсів**:
   - Консольний і графічний інтерфейси використовують один Presenter

MVVM (Model-View-ViewModel)

MVVM — це сучасний шаблон для **реактивних UI**, де ViewModel виступає посередником з підтримкою **прив’язки даних (Data Binding)**. Компоненти:

- **Model (Модель)** — дані та бізнес-логіка (як у MVP).
- **View (Представлення)** — відображає UI *через прив’язку до властивостей ViewModel*.
- **ViewModel** — перетворює дані Model у формат, зручний для View, і надає **команди** для обробки дій.

Ключова відмінність від MVP:

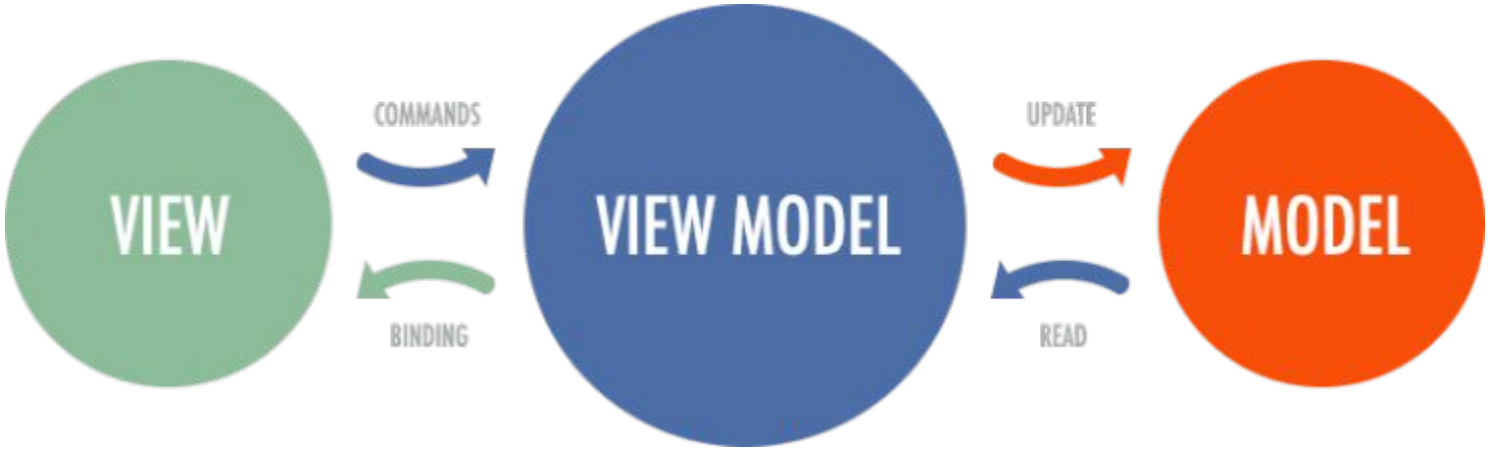
- View **не має посилань** на ViewModel (взаємодія через прив’язку даних).
- ViewModel **не знає** про View (на відміну від Presenter, який викликає методи View).
- Оновлення UI **автоматичне** (через INotifyPropertyChanged).

Як працює?

1. Користувач клікає кнопку → View викликає **Command** з ViewModel.
2. ViewModel оновлює Model → змінюються її властивості.
3. View **автоматом** оновлюється через Data Binding.

Приклад використання:

- WPF/UWP, Angular/React, Android (Jetpack Compose).



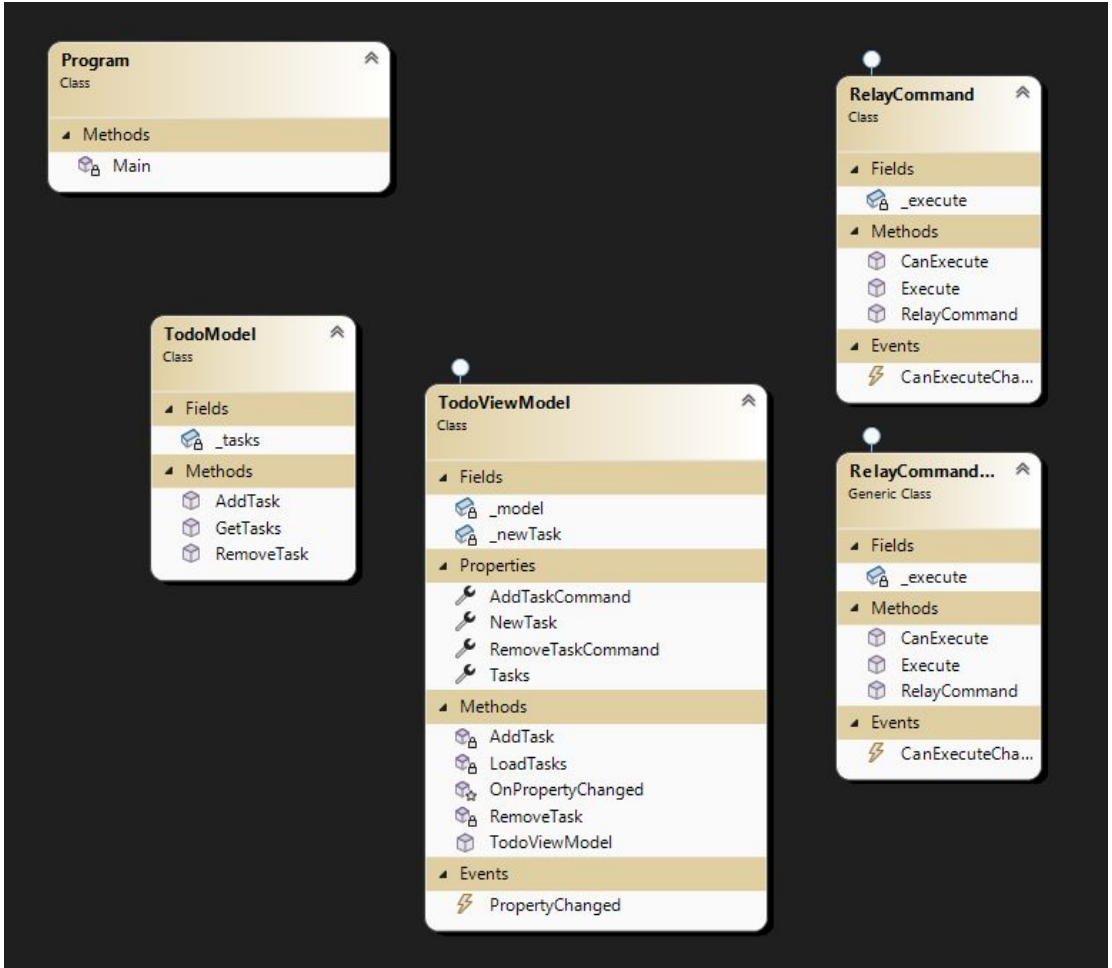
Порівняння MVP та MVVM

Критерій	MVP	MVVM
Зв’язок з View	Presenter викликає методи View	View прив’язується до ViewModel
Оновлення UI	Вручну (Presenter → View)	Автоматичне (Data Binding)
Тестування	Легше, ніж у MVC	Найлегше (ViewModel ізольована)
Підхід до логіки	Імперативний (ручне керування)	Реактивний (зміни поширюються)

Приклад для розуміння:

- У **MVP** Presenter каже View: "Покажи температуру 25°C".
- У **MVVM** ViewModel містить властивість Temperature = 25, а View **сама** відображає зміни через Binding.

Приклад: TO-DO list



Ключові особливості цього прикладу MVVM:

1. **ViewModel**:
   - Містить логіку представлення
   - Реалізує INotifyPropertyChanged для сповіщення про зміни
   - Надає команду GetWeatherCommand для взаємодії з View
2. **View**:
   - Тільки відображає дані та передає введення користувача
   - Підписується на події зміни властивостей ViewModel
   - Не містить жодної бізнес-логіки
3. **Model**:
   - Як і раніше відповідає за отримання даних
4. **Data Binding**:
   - Імітується через ручну підписку на PropertyChanged
   - У реальному WPF додатку це буде автоматично через XAML
5. **Команди**:
   - Використовується проста реалізація ActionCommand
   - У WPF буде ICommand з підтримкою CanExecute

Цей приклад демонструє основні принципи MVVM:

- View не знає про Model
- ViewModel не знає про View
- Всі зміни передаються через механізм сповіщень
- Команди використовуються для обробки дій користувача

Цей приклад демонструє основні принципи MVVM:

- View не знає про Model
- ViewModel не знає про View
- Всі зміни передаються через механізм сповіщень
- Команди використовуються для обробки дій користувача

Висновок

Архітектурні патерни **MVC**, **MVP** та **MVVM** - це способи організації коду в застосунках, які допомагають розділити логіку, інтерфейс і обробку подій. Вони полегшують підтримку, тестування та масштабування програм. Слід запам’ятати, що чим складніший шаблон (MVC — найпростіший, MVVM — найскладніший), тим більше шарів абстракції він додає. І хоча це може полегшити тестування, у невеликих проектах така складність може бути надмірною та ускладнити розуміння коду.