Міністерство освіти і науки України
Тернопільський національний технічний університет
імені Івана ПУЛЮЯ

кафедра програмної інженерії

Навчальний посібник

Сучасні технології ООП-проектування та автоматичного генерування програмного коду

Тернопіль 2018

Петрик М., Мудрик І., Петрик О., Ю. Стоянов. Сучасні технології ООП-проектування та автоматичного генерування програмного коду: навчальний посібник, Тернопіль: ТНТУ імені Івана Пулюя, 2018. 48 с.

Посібник містить практичні рекомендації для допомоги студентам у вивченні дисципліни “Архітектура та проектування програмного забезпечення”, зокрема щодо підходів та технологій об’єктно-орієнтованого проектування та набуття практичних навіків щодо застосування сучасних інструментальних засобів автоматичного генерування програмного коду, включаючи класи і операції. Розроблено на основі використання раціонального уніфікованого підходу до проектування програмного забезпечення, включаючи фази аналізу вимог предметної області (на прикладі проєктування елементів інформаційних систем керування тепличним господарством та виробництва автобілів), моделювання та проектування, побудови та тестування програмного коду та налаштування системи в цілому. Посібник орієнтований для студентів спеціальності 121 інженерія програмного забезпечення для набуття практичних навиків виконання різних класів завдань для різних фаз і робочих процесів розробки програмного забезпечення.

Укладачі: М. Петрик, І. Мудрик, О. Петрик, Ю. Стоянов
Відповідальний за випуск: М. Петрик
Рецензент: С. Лупенко

Розглянуто на засіданні кафедри програмної інженерії, протокол №1 від 14.08.2018р.

Схвалено на засіданні методичної ради факультету комп’ютерно-інформаційних систем і програмної інженерії Тернопільського національного технічного університету імені Івана Пулюя, протокол №1 від 3 вересня 2018 р

ЗМІСТ

Розділ 1. Створення коду на C++ з допомогою RSA

Вступні зауваження

На основі діаграми класів IBM Rational Software Architect дозволяє створювати код класу на обраною мовою. Для того щоб скористатися даною можливістю, необхідно переконатися, що обраний мову програмування встановлений за допомогою Add-Ins менеджера. Раніше вивчався отриманий код класу після зміни установок діаграми класів. Тут подивимося, як необхідно змінити специфікації, для того щоб отримати певний код, тобто будемо спочатку виходити з коду класу, і розглянемо необхідні кроки для його отримання.

Потрібно розуміти, що дії, які необхідно зробити для створення коду на одній мові програмування, швидше за все не підійдуть для роботи з іншими мовами. Тому спочатку розберемо більш універсальний варіант створення коду на C++, незалежного від використовуваного компілятора, а створення коду на Visual C++ розберемо в наступному розділі.

Еталонний код класу

Повернемося до нашої теплиці і згадаємо, що датчиків в теплиці може бути кілька, і тому необхідно точне визначення місця розташування датчика. Використовуємо для цього його номер. На самому справі для визначення місця розташування датчика можуть використовуватися координати, які допоможуть нам вивести свідчення конкретного датчика на дисплей, але поки використовуємо тільки номер.

Для установки поточного місця розташування будемо використовувати тип Location, а для температури - тип Temperature.

Припустимо, що ми хочемо отримати наступний код на C++

```
//Температура за Цельсієм
typedef float Temperature;
//Число, що однозначно визначає положення датчика
typedef unsigned int Location;
class TemperatureSensor
{
public:
TemperatureSensor(Location);
~TemperatureSensor();
void calibrate(Temperature actualTemperature);
Temperature currentTemperature() const;
};
```

«Для об’єктів, що створюються в програмі, або, як їх ще називають — Реалізацій класів, зручно використовувати псевдоніми простих типів, наприклад, Temperature або Location замість unsigned int. Таким чином, можна описати одержувані абстракції мовою предметної області. У конструктор передається місце розташування датчика, мається можливість калібрування та отримання виміряної температури.

Асоціація класу з мовою C++

У IBM Rational Software Architect всі типи за замовчуванням визначаються як класи. Тому створимо два нових класи: Location і Temperature. Для кожного з них проробимо наступне: Виберемо клас, потім Menu: Tools => C++ => Code Generation. Тут необхідно вибрати клас, для якого призначається мову програмування, і натиснути Assign (призначити).

Перегляд коду класу

Після генерації коду в контекстному меню стане доступний додатковий пункт C++, в якому можна переглянути заголовний файл (Header). h і файл тіла класу (Body). cpp. Тепер можна переглядати файли тіла класу і заголовний файл після кожного внесеного зміни, щоб перевірити, як ці зміни відіб'ються на одержуваному коді.

Установка типу об'єкта

Тепер у знову з'явився меню вибираємо Open Specification => C++ => Implementation Type => Override, потім у графі Value заповнюємо unsigned int. Має вийти так, як показано на рис.

Порада: Якщо на будь-якому властивості натиснути праву кнопку миші, то відкриється опис цієї властивості з вбудованою довідки. Короткий опис властивостей наведено в кінці розділу. Виконайте те ж саме з класом Temperature, тільки не забудьте встановити тип float і запустіть генерацію коду RClick =:> C++ => Code Generation. Перегляньте заголовок і ви побачите, що вийшло саме те, що замовляли. Тепер можна скористатися отриманими типами для класу датчика температури.

Додавання нових операцій

Вибираємо TemperatureSensor => RClick => New Operation і вводимо ім'я calib-rate (actualTemperature: Temperature): void. Зазначимо, що в відміну від семантики мови C++, тут спочатку вказується мінлива, а потім, після двокрапки, її тип, аналогічно і повертається значення вказується після операції через двокрапку. Зліва від операції з'явився значок, якщо його вибрати, то відкривається набір значків, які відображають доступність операції, відповідно: public, protected, private і implementation. В останньому випадку, якщо елемент визначений у пакеті, він буде видно тільки для об'єктів, визначених у цьому пакеті.

Аналогічно додамо конструктор і операцію отримання температури

Порада: Якщо заповнити поле TemperatureSensor => RClick => Open Specification => Documen-tation, то ви отримаєте автоматично створювані коментарі у вихідному тексті, які висвічуються в вікні Documentation, коли виділяється документований клас.

Установка залежності класів

У нашому випадку для зв'язку класів використовується компіляційного залежність, яка має на увазі, що в клас датчика температури будуть включені визначення необхідних типів. Для того щоб показати, що в класі датчика температури повинні використовуватися типи Location і Temperature, скористаємося зв'язком Dependency, для чого виберемо її значок з рядка інструментів. Клацнемо на ньому, потім клацнемо по класу TemperatureSensor і, не відпускаючи кнопку миші, тягнемо лінію до класу Location. Аналогічно з класом Temperature. При генерації вихідного тексту в цьому випадку IBM Rational Software Architect автоматично включить файли location.h і temperature.h в заголовний файл TemperatureSensor.

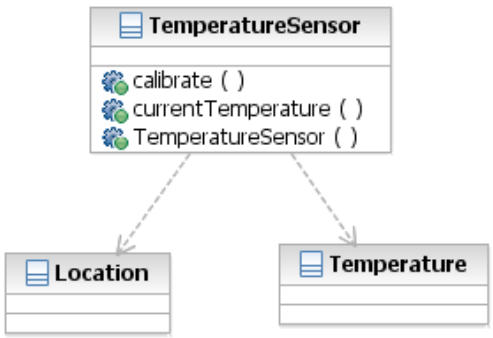

Рис. 1.1 Залежності класу Temperature Sensor

Порада: Для того щоб показати / сховати параметри і типи повернення операцій, використовуй-ті RClick => Options => Show Operation Signature.

Тепер для створеного класу можна запустити генерацію вихідного коду і порівняти його з тим, що нам необхідно. Доведення отриманого коду Якщо подивитися на отриманий файл заголовка, то можна побачити, що вийшло не зовсім те, що потрібно. Я наведу отриманий файл без деякої службової інформації, яку включає генератор в код класу. Вся службова інформація вставляється як коментарі і не впливає на подальшу генерацію виконуваного файлу. Ви можете відрізнити створені генератором рядки по символах «# #», Що вставляються в коментарі.

Include "Location.h" tfinelude "Temperature.h" //Датчик температури, вимірює температуру в теплиці в градусах Цельсія .

```
class TemperatureSensor
{
public:
//## Constructors (generated) TemperatureSensor();^
TemperatureSensor(const TemperatureSensor & right); //## Constructors (Specified)
TemperatureSensor(Location Location); //## Destructor (generated)
~TemperatureSensor(); //## Other Operations (specified)
void calibrate (Temperature actualTemperature);
Temperature currentTemperature ();
}
```

Тут видно, що в код включено заголовні файли визначення даних та коментар (це було заповнене поле RClick => Open Specificati-on => Documentation). Однак конструкторів у класі цілих три: два автоматично створених і один той, який визначили ми. І не вистачає службового слова const в операції currentTemperature.

Виправимо цю невідповідність. Спочатку видалимо автоматично створювані конструктори класу: Wind: Class Diagram => TemperatureSensor => RClick => OpenSpecification => C++ => Generate Default Constructor = DoNotDeclare і там же Generate Copy Constructor = DoNotDeclare.

Поставимо тип const: Wind: Browser + Logical View + TemperatureSensor => currentTemperature => C++ => Operation Is Const = True.

Налаштування властивостей C++

Поки ми проробили всі ці дії без докладних пояснень, з метою показати можливості IBM Rational Software Architect зі створення коду програми на C++. Для того щоб ви могли користуватися можливостями C++, далі опишемо призначення цих властивостей, список яких доступний у вкладці C++ специфікацій класу.

CodeName встановлює ім’я класу в створюваному коді. Дане властивість необхідно встановлювати тільки в тому випадку, якщо ім’я класу повинно бути відмінно від імені заданого в моделі IBM Rational Software Architect.

ImplementationType дозволяє використовувати прості типи замість визначення класу, встановлюваного IBM Rational Software Architect за замовчуванням. При завданні цього параметра створюється директива typedef.

ClassKey використовується для завдання типу класу, такого як class, struct, або union. Якщо тип не вказаний, то створюється клас.

GenerateEmptyRegion - властивість вказує, як буде створюватися порожній розділ protected: None - порожній розділ не будуть створено; Preserved -Порожній розділ буде створений, якщо буде встановлено властивість «Preserve = yes»; Unpreserved - порожній розділ буде створений, якщо буде встановлено властивість «preserve = no»; All - завжди буде створюватися.

PutBodiesInSpec якщо встановлено як True, то в заголовний файл потрапить і опис тіла класу. Використовується для компіляторів, яким необхідно визначення шаблону класу в кожному компилируемая файлі. GenerateDefaultConstructor дозволяє встановити, чи необхідно створювати конструктор для класу за замовчуванням. Може приймати наступні значення: DeclareAndDefine - створюється визначення для конструктора і скелет конструктора в тілі класу; Declare Only - створюється тільки визначення; DoNotDeclare - не створюється ні визначення, ні скелета.конструктора.

DefaultConstructorVisibility встановлює розділ, в якому буде визначений конструктор за замовчуванням: public, protected, private, implementation.

InlineDefaultConstructor встановлює, чи буде конструктор по Типово створюватися як inline підстановка.

ExplicitDefaultConstructor встановлює конструктор за замовчуванням як explicit (явно заданий).

GenerateCopyConstructor встановлює, чи буде створена копія конструктора.

CopyConstructorVisibility встановлює розділ, в якому буде створено копію конструктора.

InlineCopyConstructor встановлює, чи буде копія конструктора створюватися як inline підстановка.

ExplicitCopyConstructor встановлює, що копія конструктора буде створена explicit (явно задана).

GenerateDestructor встановлює, чи буде створюватися деструктор для класу.

Destructor Visibility встановлює розділ, де буде створюватися деструктор.

DestructorKind встановлює вид створюваного деструктора: Common звичайний, Virtual - віртуальний, Abstract - абстрактний.

InlineDestructor встановлює, чи буде деструктор створюватися як inline підстановка.

GenerateAssignmentOperation встановлює, чи буде створюватися функція перевизначення оператора привласнення (=).

Assignment Visibility визначає розділ, де буде створюватися функція оператора привласнення.

The AssignmentKind визначає вид функції оператора присвоєння: Common - звичайна, Virtual віртуальна, Abstract абстрактна, Friend - дружня.

InlineAssignmentOperation визначає, чи буде оператор присвоєння створюватися як inline підстановка.

GenerateEqualityOperations визначає, чи будуть перевизначатися оператори порівняння на рівність (== і! =).

Equality Visibility визначає розділ, в який будуть поміщені оператори порівняння на рівність.

EqualityKind визначає вид функцій операторів порівняння на рівність: Common - звичайна, Virtual - віртуальна, Abstract - абстрактна, Friend - дружня.

InlineEqualityOperations визначають, чи будуть функції операторів порівняння на рівність створюватися як inline.

GenerateRelationalOperations визначає, чи будуть перевизначатися оператори порівняння (<, <=,>,> =).

Relational Visibility визначає розділ, в який будуть поміщені оператори порівняння.

RelationalKind визначає вид функцій операторів порівняння: Common звичайна, Virtual - віртуальна, Abstract - абстрактна, Friend - дружня.

InlineRelationalOperations визначає, чи будуть функції операторів порівняння створюватися як inline підстановка.

GenerateStorageMgmtOperations визначає, чи будуть перевизначатися оператори new і delete в класі.

StorageMgmtVisibility визначає розділ, в який будуть поміщені оператори new і delete.

InlineStorageMgmtOperations визначає, чи будуть оператори new і delete визначені як inline підстановка.

GenerateSubscriptOperation визначає, чи буде перевизначений оператор []. Subscript Visibility визначає розділ, в який буде поміщений оператор [].

SubscriptKind визначає вид функцій оператора []: Common - звичайна, Virtual - віртуальна, Abstract - абстрактна.

SubscriptResultType визначає тип повертається вирази для оператора [].

InlineSubscriptOperation визначає, чи буде оператор [] визначено як inline підстановка.

GenerateDereferenceOperation визначає, чи буде перевизначений оператор *. Dereference Visibility визначає розділ, в який буде поміщений оператор *.

DereferenceKind визначає вид функцій оператора *: Common - звичайна, Virtual-віртуальна, Abstract - абстрактна.

DereferenceResultType визначає тип повертається вирази для оператора *.

InlineDereferenceOperation визначає, чи буде оператор * визначений як inline підстановка.

GenerateIndirectionOperation визначає, чи буде перевизначений оператор. Indirection Visibility визначає розділ, в який буде поміщений оператор. IndirectionKind визначає вид функцій оператора: Common - звичайна, Virtual-віртуальна, Abstract - абстрактна.

IndirectionResultType визначає тип повертається вирази для оператора.

InlineIndirectionOperation визначає, чи буде оператор визначений як inline підстановка.

GenerateStreamOperations визначає, чи будуть перевизначені оператори потоків (<< і >>).

Stream Visibility визначає розділ, в який будуть поміщені оператори потоків.

InlineStreamOperations визначає, чи будуть оператори потоків визначені як inline підстановка.

Необхідні умови та загальна інформація про процес

Тут розглянуто як програміст в C + + може використовувати структури даних C + + при розробці додатків навіть у тому випадку, якщо він створює моделі за допомогою уніфікованої мови моделювання (UML). Щоб отримати користь з читання цієї статті, достатньо мати базові знання UML і програмних продуктів IBM ® Rational ® Software Architect і IBM ® Rational ® Systems Developer, однак автори припускають, що ви вмієте виконувати перетворення і застосовувати профілі та стереотипи. Процес можна розділити на наступні етапи:

1. Починаємо з створення простої UML-моделі в IBM Rational Systems Developer;
2. Застосовуємо до цієї моделі профіль C + +, щоб можна було використовувати при моделюванні структури даних, специфічні для C + +;
3. Імпортуємо в модель бібліотеку типів C + +, щоб можна було використовувати примітивні типи C + +;
4. Генеруємо код і побіжно переглядаємо його;
5. Потім додаємо тіло в метод в якому-небудь класі;

6. Додаємо в модель кілька незначних деталей, щоб поспостерігати за тим, як при повторному застосуванні перетворення UML - C + + зберігаються зроблені вами зміни в коді і моделі.

Створення простої UML-моделі

Профіль UML - це механізм, який розширює UML стандартизованим способом, що дозволяє моделювати в UML певні специфічні для даної предметної області функції, не обтяжуючи UML додатковими відомостями про наявні та можливі в майбутньому мовах і технологіях. Автори UML не могли передбачити всіх областей, для моделювання яких використовуватиметься UML, тому для таких випадків вони передбачили механізм розширення, що дозволяє UML НЕ розростатися до занадто великих розмірів.

Застосування профілю C + +

Для моделювання, наприклад, таких специфічних для C + + елементів, як структури, об'єднання, визначаються типи *typedef* і т. п., необхідно застосувати до UML-моделі профіль C + +. Профіль C + + поставляється разом з інструментом перетворення UML-C + +. Щоб застосувати профіль, виконайте такі кроки:

1. Переконайтеся, що знаходитесь у поданні **Modeling**;
2. Виберіть модель, а потім перейдіть до подання властивості;
3. Виберіть категорію **Profiles** зі списку в лівій частині **представления Properties**;
4. Натисніть кнопку **Add Profiles** виберіть профіль **C++ Transformation** в списку **Deployed Profile** в діалоговому вікні **Select Profile**. (Див. Рис 1.2 і 1.3.)

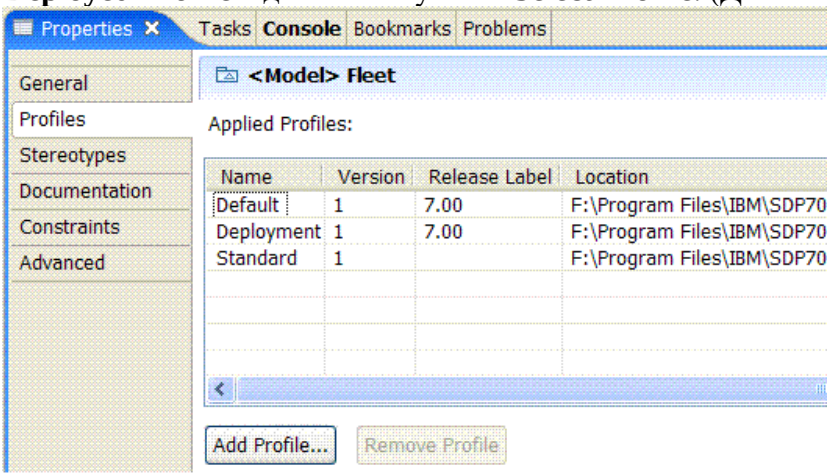

Рис1.2. Застосування профілю C + + (CPP)

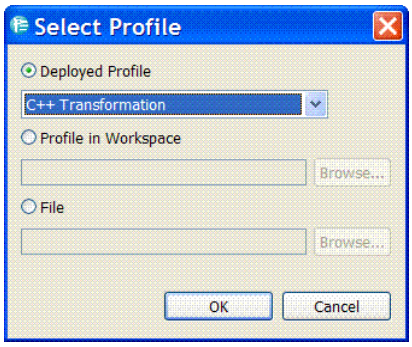

Рис1.3 Вибір профілю C + +

Імпорт бібліотеки типів C + +

UML надає дуже обмежений набір вбудованих типів. Це, як правило, такі типи, як Boolean, Integer, String і UnlimitedNatural. Більшість мов програмування, в тому числі C + +, пропонують набагато більш багатий набір примітивів. При створенні моделей для C + + вам часто будуть потрібні вбудовані примітивні типи мови C + + при присвоєнні типу атрибутам, параметрах, типами повернення операції і так далі. Щоб імпортувати бібліотеку моделювання C + +, яка поставляється разом з інструментом перетворення C + +, виконайте такі дії:

1. Натисніть правою кнопкою миші на UML-моделі в панелі оглядача проектів **Project Explorer**;
2. Виберіть з контекстного меню команду Import Model Library, як показано на Рис.1.4;
3. **У вікні Import Model Library** виберіть опцію **Deployed Library**, а потім виберіть зі списку пункт C + + типи, як показано на Рис.1.5.

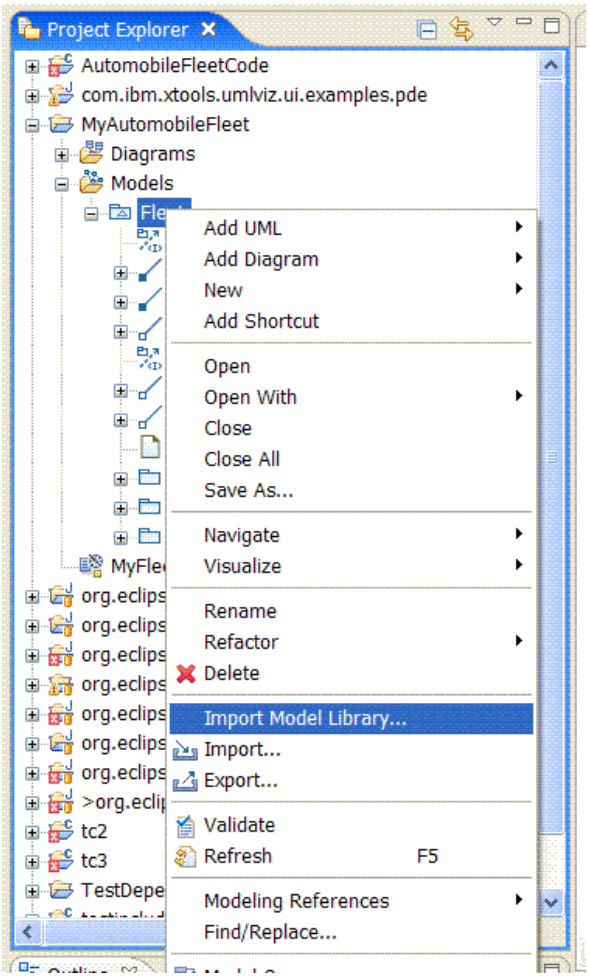

Рис 1.4. Імпорт бібліотеки типів C + +

Тепер можна переходити до моделювання специфічних для C ++ елементів, які не мають еквівалентів в UML. Приклад.

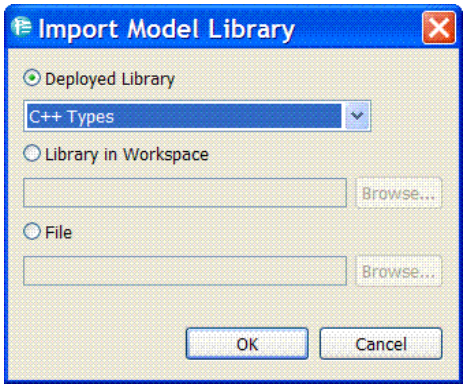

Рис.1.5 Вибір імпортованої бібліотеки типів C + +

Створення простої моделі заводу, що виробляє автомобілі

Приступимо до створення простої UML-моделі, показаної на Рис13.6. Існують різні способи створення цієї моделі, але вам пропонується слідувати показаної на малюнку схемою. Наприклад:

- Ця проста модель містить клас **Vehicle** (Транспортний засіб), базовий для класів **Bus** (Автобус) і **Car** (Автомобіль);
- Класи **Bus** (Автобус), **Car** (Автомобіль) і **Vehicle** (Транспортний засіб) створюються в UML-пакеті з ім'ям **Vehicles** (Транспортні засоби), це на схемі не показано;
- Аналогічно, класи **Engine** (Двигун) і **Wheels** (Колеса) створюються в UML-пакеті з ім'ям **Parts** (Компоненти), що також не показано на схемі;
- Клас **Car** (Автомобіль) пов'язаний асоціацією "містить" з класом **Engine** (Двигун);
- На цьому уявному заводі класи **Car** (Автомобіль) і **Engine** (Двигун) є нероздільними, оскільки жоден автомобіль не може обійтися без двигуна;
- Класи **Car** (Автомобіль) і **Bus** (Автобус) пов'язані асоціацією "збираються в одне ціле" з класом **Wheels** (Колеса);
- У цьому прикладі колеса можуть існувати без автомобіля, а автомобіль може існувати без коліс - принаймні, поки не буде закінчена збірка.
-

Створення елементів C + + в моделі

1. Потім створюємо UML-пакет з ім'ям **Stategy** на рівні моделі;
2. У пакеті **Strategy** визначимо маршрути і вихідні точки для автобусів;
   A. Створіть клас з ім'ям Route, який представлятиме маршрут, і клас Address, який представлятиме певну адресу, відповідній точці відправлення автобусного маршруту;
   B. Адреса повинна бути елементом C + + "структура", а не звичайним класом.

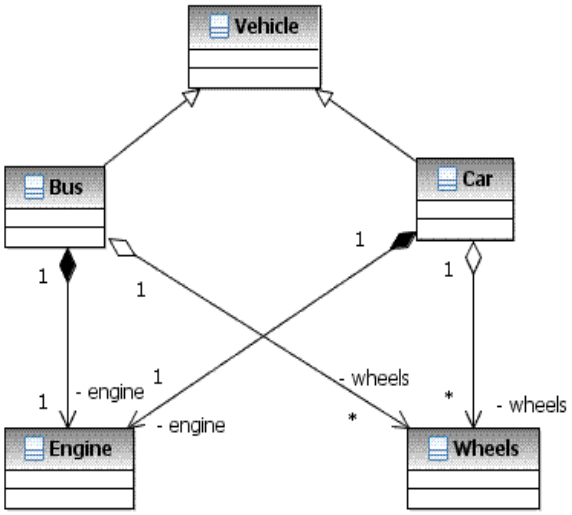

Рис.1.6. Приклад - проста UML-модель заводу з виробництва автомобілів

3. Уточнюємо UML-модель, вказавши, що маршрут призначається для кожного автобуса. Це можна зробити, додавши асоціацію "додається" від класу автобуса в пакеті **Vehicle** до класу **Route**, як показано на рис. 1.6;
4. Одним з основних властивостей маршруту є startingPoint . Значить, потрібно додати атрибут з ім'ям і типом Початкова точка Адреса класу маршруту.

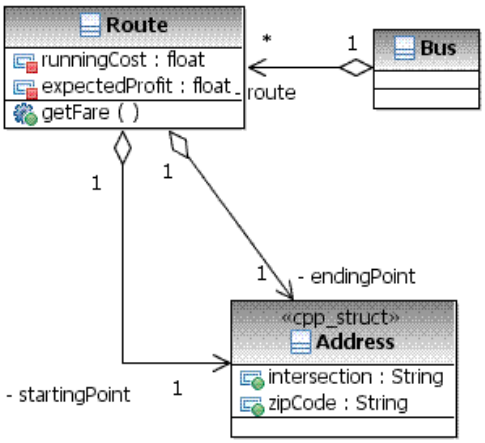

Рис.1.7 Вміст нового UML-пакета Стратегії

Зверніть увагу, що на Рис.1.7 Адреса являє собою клас << cpp_struct>>. Для створення стереотипного елемента з ім'ям і типом Address << cpp_struct>> виконайте такі кроки:

1. Створіть UML-клас і надайте йому ім'я Address;
2. Застосуйте до щойно створеного класу стереотип <<cpp_struct>>;
   A. Щоб застосувати стереотип до якого-небудь елементу UML, необхідно переключитися на подання Properties для цього елемента;
   B. Виберіть пункт **Stereotypes** зі списку категорій в лівій частині вікна подання, а потім натисніть кнопку **Apply Stereotypes**, як показано на Рис.1.8.

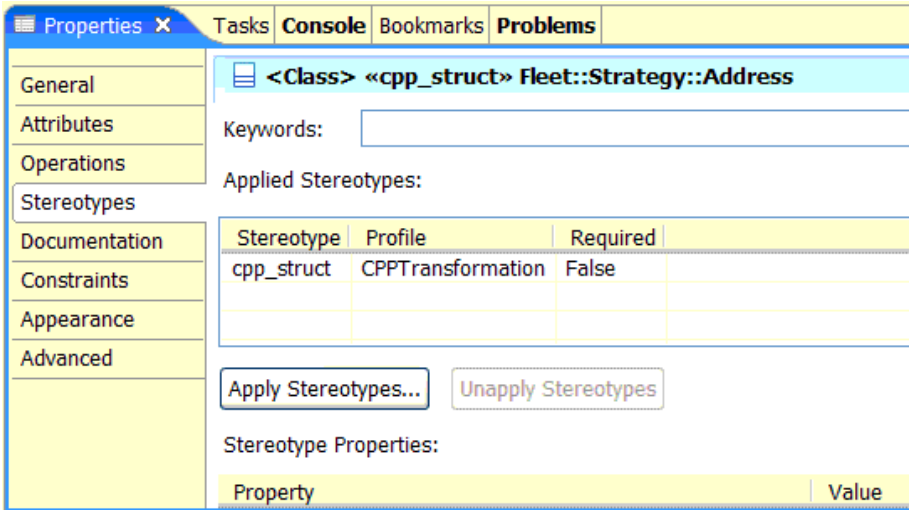

Рис.1.8. Застосування стереотипу

3. Виберіть зі списку стереотипів, застосовних для цього елемента, стереотип **cpp_struct**. Крім того, перегляньте і інші елементи списку, просто для довідки, на майбутнє.

Дуже скоро ви побачите, що при перетворенні цієї моделі в код UML-клас Address зі стереотипом <<cpp_struct >> буде згенерований як елемент структура, а не як клас.

Налаштування перетворення та генерація коду

Перш ніж ви зможете згенерувати код C + + зі своєї моделі, необхідно визначити параметри для перетворення UML - C + + шляхом створення конфігураційного файлу перетворення.

Створення конфігураційного файлу перетворення

Ось один із способів, який можна використовувати для виконання цього завдання:

1. Виберіть у меню команди File> New> Other, а потім виберіть елемент **transformation configuration** в папці **Transformation**;
2. У вікні, майстри створення конфігурацій перетворення **New Transform Configuration** wizard вкажіть ім'я для нового конфігураційного файлу. Для цієї вправи вкажіть ім'я tc1;
3. Виберіть **тип перетворення** і **проект**, в якому буде збережений конфігураційний файл;
   A. Для вибору типу перетворення розгорніть список **IBM Rational Transformations**, а потім виберіть зі списку **UML в C + +**;
   B. Щоб зберегти конфігураційний файл для цієї вправи, використовуйте вже існуючий проект UML-моделі.
4. Потім перейдіть до наступній сторінці майстра, натиснувши кнопку Next;
5. На вкладці **Source and Target**:
   A. Виберіть UML-модель в якості джерела;
   B. Створіть новий керований проект C + + в якості призначення.
6. Переконайтеся, що **джерело** і **призначення** обрані, після чого натисніть кнопку **Finish**.
   Примітка: Якщо потрібно, конфігураційний файл можна згодом змінити.

Генерація коду

Це не повинно викликати складнощів

1. Ви вже зробили саму важку роботу, тому можете приступити до створення коду, для цього просто натисніть правою кнопкою миші на tc1.tc і виберіть команди Transform> UML в C + + з контекстного меню. Після цього в проекті, який був обраний в якості призначення при створенні конфігураційного файлу перетворення, буде згенеровано потрібний код;
2. Перегляньте код, який був згенерований для класу Route. Він містить атрибути startingPoint і endingPoint, а також метод getFare () (див. код лістингу 1).

Лістинг 1. Route.h - код, згенерований для класу маршруту

```
#ifndef ROUTE_H
#define ROUTE_H
//Початок секці для файла Route.h
//TODO: Добавити визначення, які мають бути збережені
//Завершення секції  для файла Route.h

struct Address;

//@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
class Route
{

   private:
  //@uml.annotationsderived_abstraction="platform:/resource/UML-1/
DWArticle-1.emx#_Hk7qMACuEdy9t-_gdCbefQ"
      //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
      float runningCost;

            //@uml.annotationsderived_abstraction="platform:/resource/UML-1/
DWArticle-1.emx#_hKd58ACuEdy9t-_gdCbefQ"
      //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
      float expectedProfit;

            //@uml.annotationsderived_abstraction="platform:/resource/UML-1/
DWArticle-1.emx#_ieiw8ACuEdy9t-_gdCbefQ"
      //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
      Address * startingPoint;


//@uml.annotationsderived_abstraction="platform:/resource/UML-1/
DWArticle-1.emx#_jkEOkACuEdy9t-_gdCbefQ"
        //@generated              "UML              to              C++
(com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
        Address * endingPoint;
    public:


//@uml.annotationsderived_abstraction="platform:/resource/UML-1/
DWArticle-1.emx#_kawmAACuEdy9t-_gdCbefQ"
```

```
        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"

        float getFare()const ;

};  //Завершення опису  класу Route

#endif
```

Тіло методу getFare за замовчуванням було згенеровано відповідно з лістингом 2.

Лістинг 2. Route.cpp

```
#include "Route.h"
//початок секції для класуRoute.cpp
//TODO: Необхідно додати визначення, які мають бути збережені
//Завершення секції для файла Route.cpp


float Route::getFare() const
{
    //TODO Автоматично згенерований метод заглушок
    return 0;
}
```

Зміна коду

Ви повинні були помітити, що тіло за замовчуванням навряд чи використовується часто, тому що тариф завжди дорівнює 0 (нулю). Навряд чи це зробить підприємство вигідним! Тому нам доведеться замінити тіло методу getFare кодом, представленим в лістингу 3.

Лістинг 3. Route.cpp

```
float Route::getFare() const
{
    // Розрахунок тарифу
    return runningCost * expectedProfit;
}
```

Розробка моделі та повторне застосування перетворення

Далі, додамо ще один метод з ім'ям print_disclaimer () у клас Route UML-моделі.

1. Вкажіть тип повернення для методу - **String**;
2. Зверніть увагу на те, що ми змінили також файл **Route.cpp**, додавши тіло для методу **getFare**;
3. Виконайте перетворення, скориставшись тією ж конфігурацією перетворення (tc1.tc> **Transformation**> UML в C + +);
4. Налаштування за замовчуванням, які ви побачите в діалоговому вікні; вони попереджають про те, що будуть оновлені файли призначення.

Після цього вивчите оновлений код (Лістинг 4).

Лістинг 4. Оновлений файл Route.cpp

```
#include "Route.h"
//Початок секції для файла Route.cpp
//TODO: Слід добавти визначення, які мають бути збережені
//Завершення секції для файла Route.cpp


float Route::getFare() const
{
    // Розрахунок тарифу
    return runningCost * expectedProfit;
}

//@generated "UML to C++
(com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
const char * Route::print_disclaimer()
{
     //TODO Автоматично сгенерерований метод заглушок
    return 0;
}
```

Тег @generated

У коментарях до методу ви напевно звернули увагу на **тег << @generated >>.** Тіло методу зберігається автоматично. Однак якщо ви видалите метод з моделі і знову виконайте перетворення C + +, то метод буде видалено разом з тілом коду. Це гарантує, що ваш коду не буде засмічуватися всілякими методами, які ви коли-небудь розробили в моделі і які інакше довелося б видаляти з нього вручну.

Якщо ви не хочете, щоб це сталося, і, навпаки, хочете зберегти даний метод в коді, то вам доведеться видалити **тег << @generated >>** з коментаря до методу. Таким чином, ви збережете контроль над цим методом, і він залишиться в коді після повторного виконання перетворення.

Зверніть увагу, що змінене тіло для методу getFare () збережено. У той же час новий метод print_disclaimer, доданий в модель, викликав генерацію нового коду у файлах Route.h і Route.cpp файлів. Так ви можете продовжувати реалізувати свої методи в коді і, одночасно, вносити поетапні зміни в модель. Завдяки цьому у вас з'являється можливість вести керовану моделями ітеративну розробку.

Додавання користувача пропозицій include в секцію збереженого коду

У лістингу 5 написано код для методу print_disclaimer. Зверніть увагу на використання оператора cout. Перетворенню C + + про це нічого не відомо. Однак, щоб повторне виконання пройшло коректно, важливо, щоб код можна було піддати синтаксичному розбору для збереження структури коду і тіла методів при повторному виконанні перетворення. Щоб зробити код компільованим, необхідно додати у файл з розширенням CPP пропозиції # include <iostream> і using namespace std. Ці пропозиції обов'язково будуть збережені при повторних виконаннях перетворення UML - C + +. Для цього необхідно вставити зазначені пропозиції між коментарями:

Все, що додасте в цю секцію, буде збережено дослівно і не буде інтерпретуватися перетворенням C + +.

Лістинг 5. Route.cpp

```
#include "Route.h"
//Початок секції для файла Route.cpp
#include <iostream>
using namespace std;
//Завершення секції для файла Route.cpp


float Route::getFare() const
{
    // Розрахунок тарифу
    return runningCost * expectedProfit;
}

//@generated "UML to C++
(com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
const char * Route::print_disclaimer()
{
    const char *s = "No Refund once the ticket is purchased\n";
    cout << s;
    return s;
}
```

Розділ 2. Створення робочого додатка на VC++ та шаблона додатку

Стратегічні і тактичні рішення

До цього моменту вся попередня робота була більше направлена на вивчення IBM Rational Software Architect і підготовку до створення додатку, розробку спільної структури класів і їх взаємодій. І ось, нарешті, настав момент для створення робочого додатка. При цьому потрібно бути готовим, що стратегічні рішення, прийняті нами на попередньому етапі, можуть суперечити тактичним рішенням, які необхідно прийняти для розробки додатка. Стратегічними є такі рішення, від яких залежить робота програми. Це алгоритми роботи, формати збереження даних, структура класів і таке інше. Зазвичай такі рішення приймаються при проектуванні задачі постановником і відображаються в специфікаціях на програму.

Для того, щоб реалізувати в програмі стратегічні рішення, потребується прийняття більшої кількості тактичних рішень. Зазвичай тактичні рішення приймаються самим програмістом та цілком і повністю залежать від досвіду роботи.

До тактичних рішень можна віднести рішення конкретної реалізації змінних, функцій та класів, їх імена, області видимості, взаємодія в межах позначених специфікаціями алгоритму. З цього випливає, що чим докладніші специфікації програми, тим менше тактичних рішень необхідно приймати програмісту. І, відповідно, програміст може менше впливати на створюваний код.

Які ж рішення проектувальники залишають на відкуп програмістам? Це може бути вибір між використанням посилання або значення змінних, використання тої чи іншої бібліотечної функції або класу для реалізації спільного алгоритму, використання тих або інших конструкцій мови, наприклад, використання конструкції goto. Тактичні рішення незамітні зовні програми, але здатні зробити серйозний вплив на швидкодію, об'єм потрібної пам'яті, легкість подальшого супроводу і час розробки програми. По прийнятих тактичних рішеннях можна в значній мірі судити про кваліфікації програміста.

Чому ж при постановці задач виділяється на порядок менше уваги тактичним рішенням, чим стратегічним? Адже проектувальник за визначенням досвідченіший програміста і може підказати те чи інше рішення в конкретній ситуації. Тільки по причині великої трудомісткості опису таких рішень.

Легше зразу написати програму, ніж описувати такі рішення на папері. Але з появою IBM Rational Software Architect ситуація докорінно змінилась. Тепер вже при постановці задачі можна задати посередництвом діаграми імена використовуваних змінних і класів, що значно звужує область прийняття рішень кінцевим програмістом і підвищує якість програмного коду. На якість програмного коду також впливає угода про використання імен, основи якої розглянемо в наступному параграфі.

Угода про використання імен

Корпорація Microsoft розробила для програмістів спеціальну угоду про використання імен в програмах. Згідно цій угоді для функцій використовуються імена, які побудовані з дієслів і іменників, при чому перші букви цих слів — великі. Для імен змінних Microsoft пропонує більш складну систему, яка передбачає позначення іменованих типів даних. Для цього використовується невеликий префікс із рядкових букв, власне ім'я починається з великої букви. Далі наведена таблиця префіксів, запропонованих Microsoft.

Таблиця 2. Префікси змінних

Префікс	
b	Булевий(байт)
c	Символ(байт)
S	Рядок(char чи CString)
dw	Довге без знакове ціле(DWORD)
f	16 бітний прапорець(бітова карта)
fn	Функція
h	Дескриптор(handle)
I	Довге ціле(long)
i	Дані типу Int
Ip	Довгий вказівник(long pointer)
n	Ціле(16 біт)
p	Вказівник(pointer)
pt	Точка(два 32 бітних цілих)
w	Ціле без знаку(WORD, 16 біт)
sz	Вказівник на стрічку, яка закінчується на 0(string zero)
Ipsz	Довгий вказівник на sz(long pointer string zero)

rgb	Довге ціле, яке містить кольорову комбінацію RGB

Для програм, написаних з використанням бібліотеки MFC, можна додати, що ім'я класу починається на C, а ім'я атрибута класу на m_. При чому при створенні класів за допомогою VC++ автоматично створюються заголовочні файли і файли тіла класу без букви C на початку імені.

Використання угоди про імена в програмах не являється обов'язковим, але при використанні угод програма не просто стає легкою для читання, але і значно полегшується її подальший супровід. Звичайно, вказувати чи ні тип змінної в її назві — це особиста справа кожного, але звичка користуватися такою угодою є ознакою гарних манер програмної інженерії.

З самого початку в програмі контролю клімату тепличного господарства тмких угод не дотримувалось. На це є свої причини. Перша і найголовніша з яких — це те, що при написанні програми Г. Буч (тут використані його головними ідеї при проектуванні класів тепличного господарства) не дотримується таких угод. Класи імен і змінних залишені такими же, то від дотримання угод потрібно було трішки відступити. Однак ті класи, які я створював з тактичних помислів, я постарався називати, притримуючись вказаних угод.

Структура додатка

Так як ми створюємо гідропонну систему на мові VC++ з використанням бібліотеки MFC, то для того щоб в подальшому добре орієнтуватися в створеному коді програми, коротко розглянемо структуру додатка MFC.

Майстер створення додатків VC++ (AppWizard) може створювати декілька типів додатків, кожен з яких використовується для своїх цілей. Перечислимо ці типи:

Single document (додаток працює тільки з одним документом);

Multiple document (додаток працює з декількома документами);

Dialog based (додаток створений на основі вікна діалогу).

Для нашого випадку найбільш зручна структура додатка, працюючого з одним документом.

Поняття "документ" для тепличного господарства

Поняття "документ" широко розповсюдилось у практиці програмування з подачі компанії Microsoft. Документом тепер прийнято вважати будь-яку сукупність даних, а не тільки тексту. Так, звукозапис чи відео фрагмент, вміщені в комп'ютер, вважаються документом. Для створення програм відображення і редагування документів компанія Microsoft створила шаблон роботи з документами, де вже передбачені основні необхідні функції, такі як відображення у вікні, збереження чи друк.

Дані про стан теплиці також можна показати у вигляді документа, хоч це не є обов'язковим. Просто це більш швидший спосіб для створення робочого додатка, ніж створення його з нуля на базі вікна діалогу.

Фактично, якщо прийняти за основу шаблон документа, то можна за короткий час створити графічну систему управління процесами в теплиці, яка буде відображати на екрані

стан пристроїв в графічному вигляді і підпорядковуватиметься оператору, який одним натисканням мишки може змінити стан пристроїв або вибрати іншу програму вирощування. Так створюються автоматизовані системи управління технологічними процесами (АСУТП).

Тут не будемо приділяти великої уваги зовнішньому відображенню, так як це вже не залежить від структури класів, які створюються за допомогою IBM Rational Software Architect, а залежить лише від зовнішнього прояву їх взаємодії, тому наша система буде просто відображати стан пристроїв у текстовому виді, фактично, виводити протокол роботи на дисплей.

Класи, які створені майстером додатків

При створенні шаблона додатка майстер створення додатків створить код наступних класів:

- головний клас додатка CGreenhouseApp;
- клас документа CGreenhouseDoc;
- клас перегляду CGreenhouseView;
- клас для вікна “Про програму” CAboutDlg;
- клас основного вікна програми CMainFrame;

Всі додатки VC++ MFC є об’єктами. Таким чином, додаток – головний клас, який включає в себе всі необхідні для роботи класи: документів, вікон перегляду. Цей об’єкт є глобальним і створюється при виклику додатка.

Зазвичай Майстер називає його theApp. Дотримуючись угоди про імена, Майстер має створити головний клас додатка з іменем проекту, до якого додають на початку букву C – class (клас), а в кінці App – application (додаток).

І отримаємо CGreenhouseApp, який унаслідується із бібліотечного класу CWinApp.

Майстер зразу має перевизначити функцію класу CWinApp::InitInstance, код якої виконується в першу чергу при загрузці додатка, і ініціалізує всі необхідні ресурси для роботи додатка. Непоганою ідеєю розділення коду створення вікна і візуалізації даних від власне даних. В такому випадку можна змінювати внутрішнє представлення(графічне або текстове), не змінюючи змісту даних.

Майстер створення додатку саме так і поступає. Він створює клас документу (в нашому випадку CGreenhouseDoc), в якому має проходити вся обробка даних, і наслідує його із бібліотечного класу CDocument, і клас перегляду (CGreenhouseView), який відображає дані на екран комп’ютера. Дані відображаються у вікні (CMainFrame), клас якого унаслідується із бібліотечного класу CFrameWnd. Описана ієрархія показана на рис.2.1.

Для нашого додатку спеціально взятий клас CEditView, а не клас CView, для того, щоб простіше було відображати стан приладів простої текстової стрічки, для чого клас CEditView підходить максимально, так як він призначений для роботи саме з тестовими документами.

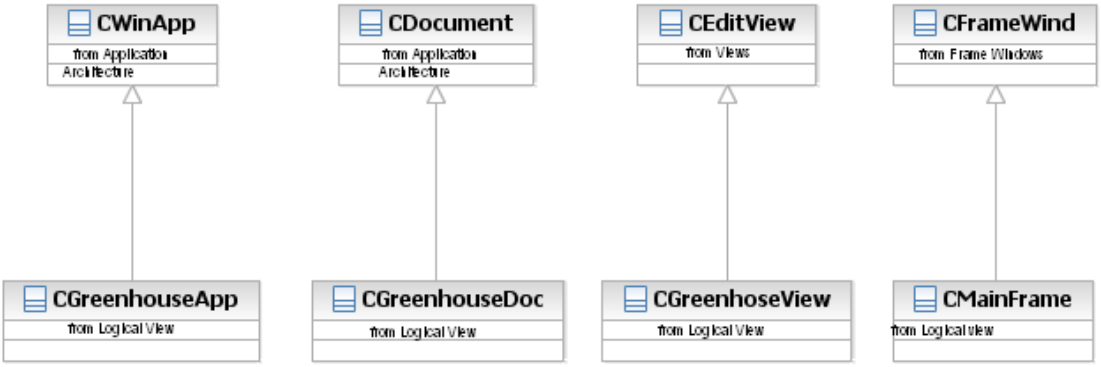

Рис. 2.1. Ієрархія класів додатків

Клас вікна “Про програму” я не розглядаю докладно, так як він зовсім не потрібний в додатку.

Операція OneNewDocument

Важливою операцією в тільки створеному класі CGreenhouseDoc буде операція OneNewDocument. Майстером вона не створюється, але нам потрібно її пізніше перевизначити. В даній операції будуть ініціалізовані всі змінні, які відносяться до документу.

Виникає питаня, чому ці змінні ініціалізуються не в конструкторі класу, так як це робиться для більшості класів? В даному випадку ініціалізація саме в цій операції витікає від призначення класу. Так як новий документ можна створити, коли клас вже ініціалізований конструктором, і отже, необхідно десь провести ініціалізацію змінних нового документу. І саме підходяще місце для цього – операція OneNewDocument.

Так як ми будемо відображати стан приладів з допомогою класу CEditView, то найбільш зручним буде включити обробку події таймера саме в цей клас. При створенні обробника таймера з допомогою інструмента Model Assistant вона отримає назву OnTimer. Як це зробити, розглянемо пізніше. Таким чином, на основі стандартних класів документу, які представляють MFC, ми отримаємо додаток, в якому нам потрібно буде тільки добавити функціональність, а за відображення документу на екрані буде відповідати бібліотека.

Тепер, коли зрозуміла структура додатку, потрібно створити його код. Раджу створити новий проект VC-H, для того, щоб пройти шлях створення проекту ще раз виключити наслідки, які могли залишитися від попередніх експериментів.

Створення шаблону додатка

Для того, щоб асоціювати проект IBM Rational Software Architect з проектом Visual C++, якщо Ви цього ще не зробили, потрібно вибрати вікно Menu: Tools => Visual C++ => Component Assignment Tool. В ньому мишкою перетягуємо необхідні класи в стрічку VC++ на питання, чи хочете ви створити компонент VC++, для того, щоб призначити вибрані класи в новий компонент, потрібно відповісти “Так”.

Всі класи, для яких потрібно створити вихідний код, потрібно поміщати в програмні компоненти. Для простоти роботи створимо тільки один компонент, який включений в проект greenhouse.

У вікні “Select a VC+ project file” , яке з’явилося, нажати кнопку Add, після чого заповнити вікно створення нового проекту так, як показано на . Таким чином, ми скористаємося майстером створення MFC додатка.

Створимо одновіконний додаток обробки документу з допомогою майстера, для чого натиснемо кнопку “Single document. Подальші кроки майстера показувати не варто, так як в них можна залишити всі запропоновані параметри без змін. Пройдіть всі кроки майстера , почергово натискаючи кнопку “Next”. Однак на шостому кроці, перед тим, як нажати кнопку “Finish” змініть Base class (базовий клас) на CEditView.

Після завершення всіх процесів програма повернеться у вікно вибору проекту, де потрібно вибрати тільки що створений проект.

Призначення класів в проект

Нажміть ОК і проект Visual C++ готовий. Тепер можна призначити класи в проект, назва якого з’явилася під написом VC++ у вікні Component Assignment Tool. Для цього переміщено мишкою необхідні класи в проект. Для того, щоб отримати перший варіант робочого додатку, поки включіть в проект тільки класи пристроїв. Потроху ми будемо розширяти проект, але поки для швидкості роботи будемо працювати тільки з класами пристроїв.

Імпорт бібліотеки MFC

Для того, щоб можна було в подальшому працювати з класами MFC, потрібно імпортувати цю бібліотеку класів в модель. Виберіть Menu Tools => Visual C++ => Quick Import MFC 6.0. Після цього з’явиться повідомлення про загрузку класів в модель.

Загрузка створених класів у модель

Тепер у модель IBM Rational Software Architect можна загрузити класи, створені у Visual C++ . Це класи, про які ми вже згадували:

- головний клас додатка CGreenhouseApp;
- клас документу CGreenhouseDoc;
- клас перегляду CGreenhouseView;
- клас вікна “про програму” CAboutDlg;
- клас головного вікна програми CMainFrame.

Для того, щоб вони з’явились в проекті, необхідно обновити проект по готовому коду за допомогою наступної послідовності дій: Menu: Tools=>Visual C++ => Update Model From Code.

Зараз це робити необов’язково, тому що в код необхідно внести деякі зміни для нормальної компіляції, проте, для того щоб подивитись як будуть відображені класи в проекті IBM Rational Software Architect, ви можете обновити модель із коду. Після оновлення IBM Rational Software Architect автоматично створить діаграму, яка відображає структуру класів, які унаслідуються з класів бібліотеки.

Головне в цьому — не видаляйте класи, IBM Rational Software Architect не знайде в отриманому коді. Їх там ще немає, і для того щоб вони з’явились, необхідно оновити код по моделі: Menu: Tools=>Visual C++ => Update Code. Саме це я раджу зробити зразу, для того щоб отримати додаток, який компілюється без помилок.

В подальшому в будь-який момент можна оновити модель по зміненому коду або оновити код по змінах в моделі, для того щоб синхронізувати проекти. Після того як пройде оновлення, ми отримаємо одинакові класи в моделі IBM Rational Software Architect і в проекті VC++.

Перший запуск додатка

Що ж вийшло після генерації початкового коду ?

Запустити Visual Studio і перейдіть в проект greenhouse. Нажміть F7 для створення EXE файлу. Якщо ви очікували , що компіляція пройде без помилок з першого разу, то я вас розчарую, в отриманий за допомогою IBM Rational Software Architect код необхідно внести невеликі зміни для створення файла, який виконується. Радує те, що зміни, внесені один раз, запам'ятовуються, і при оновленні моделі немає необхідності вносити їх повторно.

При першій компіляції може виникнути багато помилок.

Загляньте в протокол. Найпоширенішою помилкою вважається:

"Fatal error C1010: unexpected end of file while looking for precompi led header directive". Це означає, що потрібно підключити файл "stadfx.h" у всі файли, де виникла така помилка.

Відкрийте ці файли і додайте рядок #include "stdafx.h" перед іншими операторами #include. Після завершення цієї роботи знову запустіть генерацію.

Помилок стало набагато менше, тепер можна зайнятися помилками типу syntax error : missing ; before identifier …

Це не IBM Rational Software Architect створив код з помилками – це просто невизначені типи, які ми використали в моделі, але для яких IBM Rational Software Architect не створює код.

На жаль, генератор коду Visual C++ в IBM Rational Software Architect не підтримує ключове слово typedef і enum, тому потрібно визначити даний тип безпосередньо у вихідному коді, наприклад, створимо файл, який включається з визначенням типів.

Створимо безпосередньо в проекті VC++ файл defs.h Project => Add To Project => .

```
#pragma //для того, щоб файл включився один раз
//стан пристрою включений або вимкнений enum DeviceState{Off, On}
//тип температури краще зробити простим типом, а не класом typedef float Tenperature
//денний або нічний час
enum Lights {day, night};
//тип для опису pH
typedef float pH;
//типи для опису поточного часу, години і часу
typedef unsigned int Day;
typedef unsigned int Hour;
typedef unsigned int Minute;
```

Можна обійтися і без перевизначень. Так як для компілятора слова typedef означає тільки синонім стандартних типів. Але для написання програм визначення типів конкретної предметної області зручне для подальшого супроводження програми, а також при розробці зручно мати справу не з беззнаковим float, а з конкретним Temperature.

Після того, як файл визначень створений, його можна включить в модулі, де використовуються ці визначення. Так як ми створили визначення і для типу температури, то клас Temperature нам більше не потрібний, і його можна видалити з моделі.

Порада: Файл визначень зручно включати в файл stdafx.h.

Тепер в деяких класах є операції, які повертають значення. Поки для них створені тільки шаблони, тому поки тексту немає. В такі операції поставимо return з будь-яким значенням потрібного типу.

Що ми отримаємо при першому запуску програми? Ми отримаємо додаток Windows, який поки що не виконує необхідних нам функцій. Але це вже повноцінний додаток, який включає шаблони розроблюваних нами класів, які залишилися довести до робочого стану.

Розділ 3. Додавання функціональності в клас перегляду

Створення пункту меню Work (робота)

Майстер програми створив для нас повноцінний текстовий редактор, на основі якого і буде здійснюватися подальша робота. Функції редактора нам не потрібні, я їх використовував для більш простого виведення тексту на екран. Але, що найголовніше, у проекті вже є заготовки класів для роботи нашої теплиці. Їх тільки необхідно наповнити змістом.

Якщо порівняти отримане у нашому випадку вікно програми з представленим на малюнку, то помітите, що в моєму варіанті присутній пункт меню Work (робота). Воно створене для того щоб дати команду програмі почати обробляти план вирощування, тобто при запуску програми вона чекає від оператора команди, коли необхідно почати відстеження стану середовища.

Такий тактичний хід цілком логічний. Адже якби ми створювали програму для роботи реальної теплиці, то залишили б можливість створення і редагування планів вирощування різних рослин і конфігурування пристроїв теплиці, наприклад зазначення їх розташування і кількості. І тільки після цього можна було б запустити план вирощування.

Цей пункт меню був створений таким чином: з проекту Visual Studio я додав новий пункт у ресурс Menu і присвоїв йому ID = ID_WORK_START-PLAN.

Асоціація операції з пунктом меню

Для того щоб цей пункт меню почав працювати, його необхідно асоціювати з операцією класу, для чого є два шляхи: створити операцію за допомогою Visual Studio Class Wizard, створити обробку даного пункту в IBM Rational Software Architect. Другий варіант найбільш цікавий для нас. Для того щоб ним скористатися, перейдемо в IBM Rational Software Architect і проведемо оновлення моделі з коду (Menu: Tools => Visual C + + => Update Model From Code).

Тепер можна створити його обробку в класі CGreenHouseView. Для цього необхідно виділити клас, наприклад, у вікні Browse або LogicalView і викликати для нього Model Assistant з контекстного меню. Тут якраз і проявляється глибина можливостей інтеграції IBM Rational Software Architect і VC + +. Після перемикання у вкладку MFC ми отримаємо доступ до всіх атрибутів та операцій поточного класу і всіх батьківських класів, для чого немає

необхідності вивчати ієрархію класів MFC. Для того щоб створити обробник для команди ID_WORK_STARTPLAN, виділимо пункт Command Handlers (обробниккоманд) і з контекстного меню виберемо пункт New Command Handler (новий обробник), після чого активізується вікно.

Тепер у рядку Command Handlers утворився новий пункт OnWorkStart-plan, який включає в себе інформацію про новий обробнику, таким же чином сюди можна додавати будь-які нові обробники меню, причому їх можна створити спочатку тут, а потім вже внести до ресурсів меню або інших ресурсів. Тепер можна додати й інші операції, які нам знадобляться для роботи.

Додавання операцій у Model Assistant

Для виведення поточного стану пристроїв і показників датчиків необхідно створити операцію, яка б виводила цю інформацію в головне вікно програми. Назвемо її Message. Для того щоб додати операцію, необхідно на рядку Operation (операції) з контекстного меню вибрати пункт New Operation (нова операція) і заповнити параметри операції. Не забудьте, що при заповненні імен і типів змінних спочатку необхідно вказати ім'я, а потім, після двокрапки, її тип. У будь-якому випадку завжди можна перейменувати змінну, якщо щось було заповнено неправильно за допомогою RClick => Rename.

Відстеження повідомлень таймера

Також нам знадобиться відстежувати повідомлення таймера. Раніше, до того як стала остаточно відома структура програми, ми планували вбудувати таймер безпосередньо в клас EnvironmentalControiler, хоча і не відмітили можливості визначення таймера в якомусь зовнішньому класі. Тепер можна точно визначити, куди включити даний клас. Включимо ініціалізацію та обробку таймера в клас CGeenhouseView.

Для створення обробника таймера необхідно у вкладці MFC вибрати пункт Windows Messages (повідомлення Windows) і поставити галочку на пункті OnTimer.

Таким же чином програмний інженер може створити обробники для будь-яких повідомлень Windows, досить встановити позначку напротів необхідних повідомлень, і IBM Rational Software Architect включить їх у створюваний код.

Для нормальної роботи таймера необхідно додати в клас новий атрибут m_TimerID з типом UINT. Ця змінна буде зберігати ідентифікатор таймера. Якщо даний ідентифікатор дорівнює нулю, то таймер не активізовано, а якщо не нуль, то таймер працює. Цю інформацію ми використовуємо, коли будемо заповнювати обробник OnWorkStartplan. У разі, коли додаток буде завершено до зупинки таймера, щоб, вивільнити цей ресурс Windows (адже кількість ресурсів обмежена), необхідно при закритті вікна зупинити таймер. Для цього поставте відмітку навпроти операції Destroy Window шляхом Model Assistant => MFC => CWnd => Destroy Windows = ON.

І звичайно ж, для ініціалізації змінної таймера необхідно встановити позначку для генерації конструктора класу. Редагування шаблону класу CGeenhouseView. Таким чином, шаблон для створення класу CGeenhouseView повністю створений. Далі необхідно додати в нього зміст для безпосередньої роботи. Це можна буде зробити після оновлення коду за моделлю. Так як цей процес проходить не миттєво, то можна оновити тільки код класу CGeenhouseView. Після оновлення коду можна заповнити отриманий шаблон наступним чином.

У конструкторі обнуляємо змінну таймера.

```
CGreenhouseView::CGreenhouseView()
{
m_TimerID=0;
}
```

Операцію Message заповнюємо таким чином.

```
void CGreenhouseView::Message(LPCTSTR IpszMessage)
{
CString strTemp = IpszMessage;
strTemp += _T("\r\n");
int len=GetWindowTextLength();
GetEditCtrl().SetSel(len,len);
GetEditCtr I().Rep
IaceSeI(strTemp);
}
```

Дана операція буде просто виводити в головне вікно програми передану їй рядок, додаючи символи перекладу рядка. Включимо ініціалізацію таймера в операції OnWorkStartplan. Зауважте, що зупинити виконання плану можна, повторно вибравши той же пункт меню. У даному випадку таймер встановлений на оновлення раз на секунду (другий параметр бібліотечної функції SetTimer дозволяє задавати інтервал у мілісекундах).

```
void CGreenhouseView::OnWorkStartplan()
{
if (m_TimerlD!=:0){
Ki IITimer(m_TimerlD);
m_TimerlD = 0;
Message("Stop Plan"); }
else {
m_TimerID=SetTimer(1,1000,NULL);
Message("Start Plan");
}
}
```

Заодно перевіримо, як працює операція Message.

Після перевірки змініть установку таймера на SetTimer (1,1, NULL), для того щоб час у нашій теплиці йшов швидше, ніж насправді. Це дозволить простіше налагодити програму. Змінимо операцію DestroyWindow

```
<bool CGreenhouseView::DestroyWindow()
{
if (m_TimerlD!=0){
KiIITimer(m_TimerlD); m_TimerlD =0;
}
return CEditView::DestroyWindow();
}
```

І для обробника таймера включимо повідомлення про його роботу

```
void CGreenhouseView::OnTimer(UINT nIDEvent)
{
Message(«Timer»);
CEditView::OnTimer(nlDEvent);
}
```

Внесіть зазначені зміни і створіть виконуваний файл. Якщо виникли помилки, виправте їх і запустіть програму. Виберіть пункт меню Work, при цьому у вікно програми почнуть виводитися повідомлення про те, що таймер працює. Зупиніть роботу таймера, вибравши ще раз пункт Work, після цього екран програми повинен виглядати

Розділ 3. Додавання функціональності в клас документа

Створення структури Condition

Наступним кандидатом для зміни буде клас CGeenhouseDoc. У ньому будуть зберігатися основні дані по пристроях і станом датчиків. Для того щоб цей клас міг впливати на середовище, доповнимо структуру Condition додатковими параметрами. тепер ця структура повинна буде моделювати «реальний» стан середовища, з якого датчики будуть прочитувати свідчення. Також в цій структурі буде зберігатися «реальний» стан виконавчих пристроїв (включено / вимкнено) і поточний час вирощування - день, годину, хвилина. Перейдіть в діаграму класів і створіть структуру Condition так.

Створення моделі середовища

Слід зауважити, що під час роботи теплиці датчики повинні зчитувати реальну інформацію, яка змінюється в залежності від зміни навколишнього середовища. У нашій програмі немає датчиків, з яких можна зчитувати реальну інформацію, тому необхідно створити клас, який буде вносити деяке обурення в параметри середовища. Цей клас буде моделювати підвищення або зниження температури і зміна інших параметрів.

Створимо клас CEnvironment з операціями Change і Update. Перша буде служити для внесення випадкових збурень у середовище, а друга буде служити для занесення поточного стану пристроїв і показань датчиків в структуру Condition після їх зміни за допомогою контролера середовища. Для того щоб відображати зміни середовища на екрані, створимо клас CLog, який займатиметься саме цією справою, і включимо його в клас документа. Створення зв'язків класу документа.

Також створимо клас таймера, який буде відслідковувати поточний стан часу вирощування. З'єднаємо за допомогою зв'язку Unidirectional Association класи, які ми хочемо помістити в клас документа, і отримаємо мал. 16.1.

Зауважте, що всі класи, крім GrowingPlan, включаються як агрегати, а для структури Condition, включеної в план вирощування, ми передбачили не агрегований зв'язок спеціально, для того щоб можна було динамічно створювати масив станів на кілька вказаних при створенні масиву днів. У класі CGreenhouseDoc всі операції вже створені при генерації вихідного тексту Майстром створення додатку VC + +, і немає необхідності тут їх міняти.

Додавання функціональності

Створимо вихідний текст отриманих класів і додамо в нього необхідну функціональність. Для цього оновимо код за моделлю і перейдемо до проекту greenhouse середовища Visual Studio.

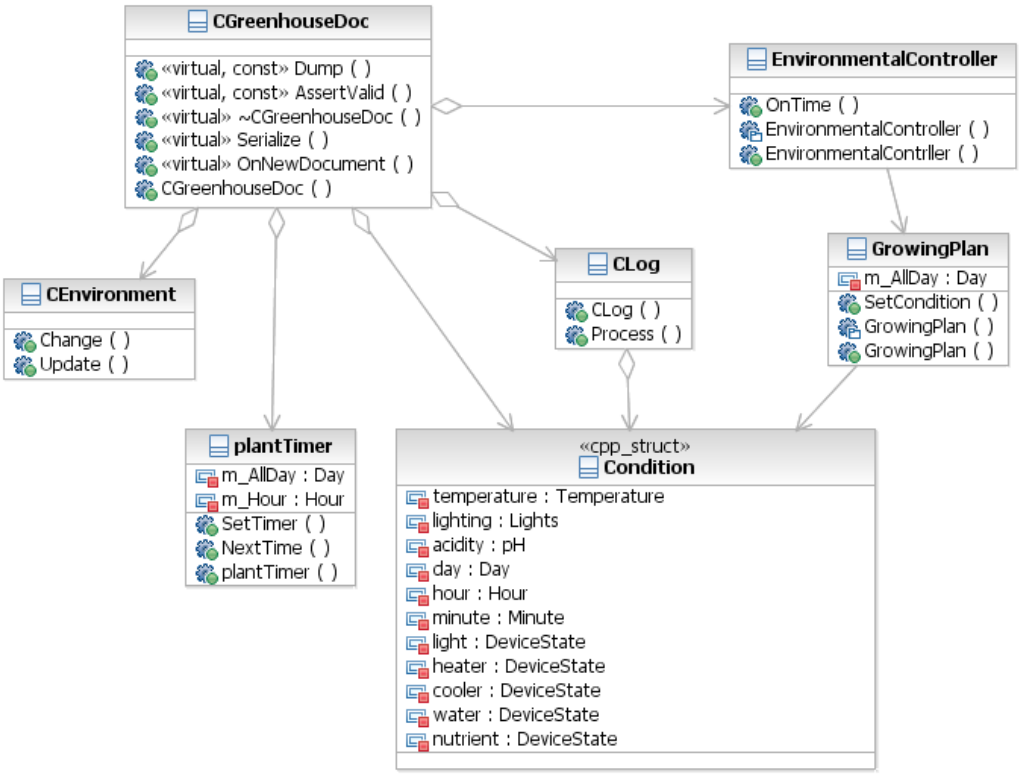

Рис. 3.1. Зв'язки класу CGreenhouseDoc

Структура класу Condition повинна виглядати приблизно так.

```
//##Modelld=3A1A18280104
struct Condition
public:
//##Modelld=3A81B6CA008C
Day day;
//##Modelld=3A81B6CA0020!
Hour hour;
//##Modelld=3A81B6C9035C
Minute minute;
//##Mode-lld=3A1A185C0370|
Temperature temperature; //##Modelld=3A1A18710096 Lights
Iighting;
//##Modelld=3A1A189103AC pH acidity;
//tfttModelld=3A81B6C9028A DeviceState Iight;
//##Modelld=3A81B6C?021E\
DeviceState heater;
//fttfModelld=3A81B6C901EO
DeviceState cooler;
//##Modelld-3A81B6C90174
DeviceState water;
//##Modelld-3A81B6C90140 \
 DeviceState nutrient;
};
```

Додамо в програму ініціалізацію плану вирощування. Для цього перетворимо клас CGrowingPlan, як показано на наступному лістингу. Сам клас виглядає після генерації таким чином.

```
class GrowingPlan
{
private:
//##Modelld=3A81B6D001CC
Day m_AII Day; public:
//##Modelld=3A81B6DOOOFO
Condition * m_Condition;
//##Modelld=3A81B6D1006E
GrowingPlan(Day day);
//##Modelld=3A81B6D1000A
~GrowingPlan();
//##Modelld=3A81B6D001CF
SetCondition(Day,Condition * condition);
 }
```

Змінна m_AllDay показує нам, на скільки днів розрахований план вирощування. Додамо вміст у тіло класу, і отримаємо наступне.

```
//##Modelld=3A81B6D1006E
GrowingPlan::GrowingPlan(Day day)
{
m_AIIDay=day;
m_Condition=new Condition[day];
}
//##Modelld=3A81B6D1000A GrowingPlan::"GrowingPlan()
{ delete[] m_Condition;
}
//##Modelld=3A81B6D001CF
GrowingPlan::SetCondition(Day day, Condition * condition)
{
for (unsigned int i=day; i< m_AIIDay; i++){
m_Condition[i].acidity=condition->acidity;
m_Condition[i].temperature=condition->temperature; }
 }
```

При створенні класу в конструктор передається кількість днів плану, для якого динамічно створюється необхідний масив структури Condition.

Для заповнення структури використовуємо операцію SetCondition. Ця операція призначена для заповнення значеннями еталонної структури плану вирощування. У структурі нас цікавлять тільки еталон стану температури і pH. Звичайно, для того щоб зменшити витрату пам'яті, можна було б створити зменшений варіант тільки з двома полями, але для простоти роботи будемо використовувати туж структуру Condition, яка використовується у всій програмі.

Необхідно пам'ятати, що першим індексом в масиві буде 0 це необхідно враховувати при перевірці в класі на закінчення часу вирощування. Клас таймера буде виглядати наступним чином.

```
//##Modelld=3A2011330320
class plantTimer
 {
protected:
//##Modelld=3A81B6CB0046
Day m_AII Day; Hour m_Hour; public:
//##Modelld=3A81B6CBOOB4
plantTimer();
//##Modelld=3A201B8A03AC\
SetTimer(Day day,Hour hour);
//##Modelld=3A81B6CB0049
BOOL NextTime(Condition *);
);
```

При установці таймера допомогою операції SetTimer в неї передається кількість днів вирощування. Ми прийняли допущення, що час вирощування рослин буде вимірюватися саме у добі. Також передамо годину початку запуску плану, для того щоб можна було б закінчити роботу в той же час, але через певну кількість діб.

І тіло класу перетворимо наступним чином.

```
plantTimer::plantTimer()
{
m_AIIDay=0; m_Hour=0;
//##ModeMd=3A201B8A03AC
plantTimer::SetTimer(Day day, Hour hour)
{
m_AIIDay=day-1; m_Hour=hour;
}
bool plantTimer::NextTime(Condition * Condition)
{
Condition->minute++; if (Condition->minute>59){ Condition-
>hour++;
Condition->minute=0;
}
if (Condition->hour>24){
Condition->day++; Condition->hour-0; - }
(Condition->day>m_AII Day &&
Condition->hour>-m_Hour)
return FALSE; return TRUE;
}
```

Створений клас таймера розрахований на роботу з інтервалом в одну хвилину, а запускаємо ми його і інтервалом в 1/1000 секунди, отже, час у теплиці біжить приблизно

60000 разів швидше, ніж насправді (насправді не настільки швидше - все залежить від мінімального інтервалу між генерацією подій апаратним таймера конкретного комп'ютера).

Тепер можна встановити кількість днів у плані і звертатися до таймерfм, коли необхідно вказати, що настав час оновлення. Зауважимо, що для поновлення таймера використовується зовнішній виклик операції NextTime, так як даний клас не має власного обробника черги повідомлень. Звичайно, можна створити повністю самостійний таймер, який буде взаємодіяти з системою, але в даному випадку це тільки ускладнить приклад, не даючи додаткових можливостей для навчання.

Клас CLog дозволяє вести протокол стану середовища та виконавчих пристроїв. У операцію Process передається покажчик на структуру Condition і покажчик на клас CGreenhouseView, для того щоб викликати операцію Message даного класу.

```
class Clog
{
private:
public:
//##Modelld=3A81B6C50244
void Process(Condition * condition,CGreenhouseView * pView);
//##Modelld=3A81B6C501D6
Condition m_Cond i t ion;
//##Modelld=3A81B6C50212
CLog(); };
```

Тіло класу після перетворення виглядає наступним чином. У конструкторі встановлюється початковий стан пристроїв і стан середовища.

```
CLog::CLog() 228
 {
m_Condition.acidity=0; m_Condition.Iighting=night;
m_Condition.temperature=0; m_Condition.heater=0ff;
m_Condition.cooler=0ff; m_Condition.water=0ff;
m_Condition.nutrient=0ff;
}
```

При запуску процесу, перед тим як вивести протокол роботи на екран, ми зробимо перевірку на наявність змін в параметрах середовища. Це необхідно, для того щоб не захаращувати екран інформацією, яка вже виведена на попередньому кроці. Єдина зміна, яку нам у цьому випадку немає необхідності виводити на екран - це зміна у свідченнях минулих діб, годин і хвилин.

```
//##Modelld=3A81B6C50244
void CLog::Process(Condition * condition,
CGreenhouseView * pView)
{ if (m_Condition.acidity==condition->acidity &&
m_Gondition.Iighting==condition->Iighting &&
m_Condition.temperature-
=condition->temperature && m_Condition.heater==condition-
>heater &&
```

```
m_Condition.cooler==condition->cooler &&
m_Condition.water==condition->water && m_Condition.nutrient-
=condition->nutrient&&
m_Condition.Iight"condition->light) return;
CString Buffer,b1;
Buffer.Format("Day:%02i %02i:%02i %sT:%02.2f
pH:%02.2f L:%s H:%s C:%s W:%s N:%s", condition->day,
conditioп->hour
condition->minute, (condition->lighting==day)?"day": "night",
condition->temperature, condition->acidity,
(condition->Iight==0n)?"ON" : "OFF",
(condition->heater==On)?"ON":"OFF",
(condition->cooler==On)?"ON":"OFF",
(condition->water==On)?"ON":"OFF",
(condition->nutrient==On)?"ON":"OFF");
pView->Message(Buffer);
m_Condition.acidity=condition->acidity;
m_Condition.Iighting=condition->lighting;
m_Condition.temperature=condition->tempenature;
m_Condition.heater=condition->heater;
m_Condition.cooler=condition-
>cooler; m_Condition.water^condition->water;
m_Condition.nutrient=condition->nutrient;
m_Condition.Iight=condition->Iight; }';
```

Тепер добавимо функціональність до класу CEnvironment.

```
//##Modelld=3A7BB6E801F4
class CEnvironment
 {
public:
//##Modelld=3A81B6D302BC
Update(Condition * condition, EnvironmentaIController .*
controller);
//##Modelld=3A81B6D300C8
Change(Condition * condition, EnvironmentalController *
controller); };
```

Щогодини цей клас буде вносити обурення в навколишнє середовище, змінюючи випадковим чином температуру і стан pH, а також при включених виконавчих пристроях буде вносити зміни в стан навколишнього середовища.

```
CEnvironment::Change(Condition * condition,
EnvironmentalController * controller)
{
if (condition->minute==0){ condition->temperature=condition-
>temperature +3- (int)(rand()/(RAND_MAX/6)); condition-
>acidity=condition->acidity +2-
```

```
(int)(rand()/(RAND_MAX/4)); }
if (controller->m_Heater.get_CurrentState()==On){
condition->temperature+=Temperature(0.1);
}
if (controller->m_Cooler.get_CurrentState()==On){
condition->temperature-=Temperature(0.1);
}
if (controller->m_WaterTank.get_CurrentState()==On){
condition->acidity-=pH(0.2);
}
if (controller->m_NutrientTank.get_CurrentState()==0n){
condition->acidity+=pH(0.2);
}
condition->acidity=round(condition~>acidity,1);
condition->temperature=round(condition->temperature,1);
Update(condition,controller);
}
//##Modelld=3A81B6D302BC
CEnvironment::Update(Condition *
condition,EnvironmentalController * controller)
{
condition->heater=controller->m_Heater.get_CurrentState();
condition->cooler=controller->m_Cooler.get_CurrentState();
condition->water=controller->m_WaterTank.get_CurrentState();
condition->nutrient=controller-
>m_NutrientTank.get_CurrentState();
condition->Iighting=(condition->hour>=9 && condition-
>hour<=20)?day:night;
conditionalight=controller->m_Light.get_CurrentState();
controller->m_TemperatureSensor.set_Value(condition-
>temperature);
controller->m_pHSensor.set_Value(condition->acidity);
}
```

Тепер, коли готове все, крім виконавчих пристроїв, можна подивитися, як працюватиме наше тепличне господарство для цього додамо в клас CGreenhouseDoc наступний код для ініціалізації початкового стану структури Condition і завдання плану вирощування.

```
//##Modelld=3A1AD7A301F4 [
bool  CGreenhouseDoc::OnNewDocument()
{
if (! CDocument: : OnNewDocumentO)
return FALSE;
m_CurrentCondition.day=0; m_CurrentCondition.hour=9;
```

```
m_CurrentCondition.minute=0;
m_CurrentCondition.temperature=25;
m_CurrentCondition.Iighting=day;
m_CurrentCondition.acidity=7;
m_CurrentCondition.Iight=0ff; m_CurrentCondition.heater=0ff;
m_CurrentCondition.cooIer=0ff; m_CurrentCondition water=0ff;
m_CurrentCondition. nutrient^Off;
m_EnvironmentalController.m_GrowingPlan->SetCondition(0,
&m_CurrentCondition);
m_plantTimer.SetTimer(1,m_CurrentCondition. hour); return
TRUE;
} // Добавим обробку повідомлененя таймера в CGreenhouseView.
void CGreenhouseView::OnTimer(UINT nIDEvent)
}
GetDocument()->m_Environment.Change(&GetDocument()-
>m_CurrentCondition,
&GetDocument()->m_EnvironmentalController);
GetDocument()->m_Environment.Update(&GetDocument()-
>m_CurrentCondition,
&GetDocument()->m_EnvironmentalController);
GetDocument()->m_Log.Process(&GetDocument()-
>m_CurrentCondition,this);
if (!GetDocument()->m_pIantTimer.NextTime(
&GetDocument()->m_CurrentCondition)){
if (m_TimerlD!=0){
KillTimer(m_TimerID); m_TimerlD = 0;
}
GetDocument()->m_Log.Process(
&GetDocument()->m_CurrentCondition,this); Message("End
plan");
}
CEditView: : OnTimer(nIDEvent); }
```

Після запуску отриманої програми повинно вийти наступне. Всі пристрої вимкнені, тому температура і pH змінюються випадковим чином. У нашій теплиці тепер не вистачає тільки виконавчих пристроїв, які компенсуватимуть зміни температури і pH.

Позначення:

Day - скільки днів пройшло з запуску плану вирощування,

показує поточний час. У моєму випадку посадки були вироблені в дев'ятій ранку, і врожай повинні будемо знімати також у дев'ятій ранку, але вже через вказану кількість днів;

Показує поточний час доби day / night;

T - поточна температура в теплиці;

pH - поточний показник кислотності;

L - відображає стан освітлювачів;

H - відбиває стан нагрівачів;
C - відображає стан вентиляторів;
W - відбиває стан крана для подачі води. ON - вода подається;
N - відбиває стан заслонки для подачі добрив. ON - добрива подаються.

Розділ 4. Створення коду виконавчих пристроїв системи

Завершенням розробки системи буде створення виконавчих пристроїв. Ці пристрої не відрізняються складною поведінкою, вони можуть знаходитися в двох станах: увімкнено та вимкнено. На прикладі виконавчих пристроїв ми розглянемо створення ієрархії класів додатку. Досі ніякої ієрархії не було створено, що не зовсім правильно. Ми розглянемо, як перетворити створені класи в класи з певною ієрархією.

Список виконавчих пристроїв

До виконавчих пристроїв нашої теплиці можна віднести:
Cooler (вентилятор);
Heater (нагрівач);
Light (лампочка);
NutrientTank (сховище добрив);
WaterTank (сховище води).

Їх усіх об'єднує те, що вони виробляють деякі дії, що виконують команди. При цьому всі вони, крім лампочки, впливають на параметри середовища, такі як температура і pH.

В результаті попередніх дій на діаграмі класів вийшов приблизно наступний стан класів цих пристроїв (рис. 4.1).

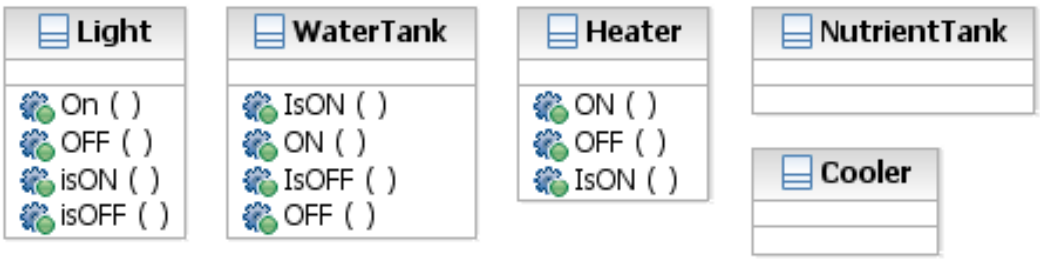

Рис. 4.1 Стан класів виконавчих пристроїв

Якщо у вашій моделі виконавчі пристрої не присутні на діаграмі класів, то додайте їх за допомогою Menu: Query => Add Classes.

Якщо подивитися на код, отриманий для пристрою, наприклад, Light (лампочка), то ми побачимо, що IBM Rational Software Architect створив заголовок класу і передбачив шаблони для занесення туди обробників операцій. Одержаний наступний заголовний файл.

```
# if defined (_MSC_VER) && (_MSC_VER> = 1000) # pragma once
#endif
#ifndef _INC_LIGHT_3A17E8EF021C_INCLUDED
```

```
#define _INC_LIGHT_3A17E8EF021C_INCLUDED
//##Modelld=3A17E8EF021C
class Light
{
public:
//##Modelld=3A2605C30384
ON();
//##Modelld=3A2605CC02E4
OFF();
//##Modelld=3A2605D3014A
lsON();
//##Modelld=3A2605DA0244
IsOFF();
};
#endif /* INC_LIGHT_3A17Ъ8EF021C_I NCLUDED і файл тіла
класу*/
#include "Light.h" //##Modelld=3A2605C30384 Light::ON()
{
}
//##Modelld=3A2605CC02E4
Light::OFF()
{
}
//##Modelld-3A2605D3014A
Light::lsON()
{
}
//##Modelld-3A2605DA0244
Light::lsOFF()
{
}
```

Тепер одного погляду на отриману модель досить, для того щоб зрозуміти, що спочатку класи не були спроектовані правильно (ось де необхідний об'єктно-орієнтований аналіз). Необхідний головний клас, з якого класи цих пристроїв будуть успадковані, тоді методи класу ON (), IsON (), IsOFF (), OFF () будуть належати головному класу, а для решти класів необхідно пере визначення методів ON (), OFF () для звернення безпосередньо до необхідного пристрою. У IBM Rational Software Architect легко виправити таку помилку.

Створення батьківського класу пристроїв

Створимо клас CPlantDevice зі специфікаціями.

Тепер необхідно створити операції, використовувані у всіх пристроях. Для цього перейдемо у вкладку Operations і за допомогою контекстного меню додамо операції таким чином. Всі операції повертають значення типу BOOL.

Якщо IsON () повертає TRUE, це означає, що пристрій включено.
Якщо IsOFF() повертає TRUE, це означає, що пристрій вимкнено.

Якщо ON () повертає TRUE, це означає, що пристрій включився нормально.
Якщо OFF () повертає TRUE, це означає, що пристрій вимкнувся нормально.

Непогано було б додати ще одну операцію get_CurrentState (отримати поточний стан), яка повинна повертати стан пристрою: On - включено або Off - вимкнено. Ми додамо її дещо пізніше. Для того щоб почати працювати з станами пристроїв, у нас вже є тип даних DeviceState, що перераховується.

Для зберігання поточного стану створимо змінну m_CurrentState з поки ще невизначеним типом DeviceState. Для цього перейдіть у вкладку Attributes і додайте за допомогою контекстного меню новий атрибут. Заповніть специфікації атрибуту.

Установка успадкування

Тепер необхідно видалити операції з виконавчих пристроїв і провести стрілки Generalization до новоствореного класу CPlantDevice таким чином, щоб вийшов стан, показане на рис. 4.2.

Тепер у нас всі пристрої успадковуються з одного класу CPlantDevice. Створимо вихідний код пристроїв і перевіримо, що вийшло.

Код класу Light став зовсім простим:

```
# include "CPlantDevice.h"
// # # Modelld = 3A6746750352
class Light: pubIic CPlantDevice
{
};
```

Зате код класу CPlantDevice отримав все необхідне для роботи.

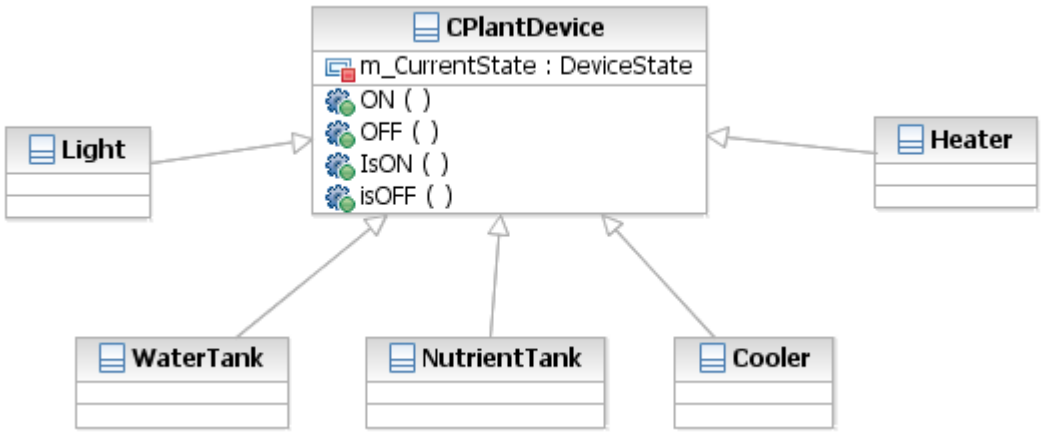

Рис. 4.2 Успадкування класів пристроїв

```
CPlantDevice.h: class CPIantDevice
{
public:
// Включення пристрою
```

```
// # # Modelld = 3A65601C0208
bool ON ();
// Вимкнення пристрою
// # # Modelld = 3A656044015E
bool OFF ();
// Перевірка чи виключений пристрій
// # # Modelld = 3A6560530122
bool lsOFF ();
// Перевірка чи включений пристрій
// # # Modelld-3A6560640136
bool lsON (); private:
// Змінна для зберігання стану пристрою.
// # # Modelld = 3A65607C0212
DeviceState m_CurrentState;

CPIantDevice. cpp:
# include "CPlantDevice.h" // # # Modelld-3A65601C0208 BOOL
CPIantDevice :: ON ()
{
// TODO: Add your specialized code here.
// NOTE: Requires a correct return value to compile.
}
// # # Modelld = 3A656044015E
bool CPIantDevice :: OFF ()
{
// TODO: Add your specialized code here.
// NOTE: Requires a correct return value to compile.
}
//##Modelld=3A6560530122 BOOL CPIantDevice::IsOFF()
{
// TODO: Add your specialized code here.
// NOTE: Requires a correct return value to compile.
} //##Modelld=3A0560640136 BOOL CPIantDevice::lsON()
{
// TODO: Add your specialized code here.
// NOTE: Requires a correct return value to compile.
}
```

Додавання значення для повернення

Тут можна помітити, що компіляція отриманого коду призведе до помилок. Хоча IBM Rational Software Architect і дозволяє з легкістю тасувати класи та їх зв'язки, але все-таки необхідно провести наповнення методів класів змістом, для того щоб отримати повноцінний додаток. Для цього необхідно внести деяку правку в код.

Додамо в методи класу IsON () і IsOFF () значення повернення безпосередньо в коді класу з оболонки Visual Studio.

```
bool CPIantDevice :: IsOFF ()
{
return m_CurrentState-^ Off;
}
// # # Modelld = 3A6560640136 BOOL CPlantDevice:: lsON ()
{
return m_CurrentState == On;
>
```

Включимо заголовний файл stdafx.h перед включенням заголовного файлу класу.

```
# include "stdafx.h"
# include "CPIantDevice.h"
```

Якщо отримано при генерації повідомлення: «fatal error C1010: unexpected endof file while looking for precompiled header directive », це означає, що ви просто забули включити файл stdafx.h у ваш файл з розширенням. cpp.

Використання Model Assistant для зміни классу

Тепер необхідно провести наступні дії:

Змінній m_CurrentState необхідно присвоїти початкове значення. Зазвичай це відбувається в конструкторі класу, який у нас ще не створений.

Для операцій ON () і OFF () можна позначити тип функцій virtual, щоб показати, що процес включення і виключення кожного пристрою повинен бути спеціальним.

Для цього переходимо у вікно Model Assistant за допомогою відповідного пункту контекстного меню, доступного після натискання правої кнопки миші на класі CPlantDevice. Тепер встановимо позначку навпроти конструктора класу

У пункті Attributes (атрибути) можна побачити, що IBM Rational Software Architect вже подбав про операції, які встановлюватимуть значення змінної m_CurrentState і отримувати це значення. Поки нам необхідна операція get_CurrentState. Встановимо позначку навпроти цієї операції і заповнимо необхідні установки так.

Необхідно зауважити, що за замовчуванням IBM Rational Software Architect створює тип значення, що повертається (Return Type) як посилання. У нашому випадку в посиланні немає необхідності.

Не забудьте встановити для операцій ON () і OFF () поле Operation Kind (Тип операції) у Virtual, і можна закривати це вікно.

Після генерації ми отримаємо наступний код класу.

```
CPlantDevice. h:
class CPlantDevice
{
public:
// # # Modelld = 3A73FAC302DO CPlantDeviceO;
```

```
     // # # Modelld = 3A73FAC3037A DeviceStateget_CurrentState ();
// Включення пристрою // # # Modelld = 3A65601C0208
virtual BOOL ON ();
// Вимкнення пристрою // # # Modelld = 3A656044015E
virtual bool OFF ();
// Перевірка чи виключено пристрій // # # Modelld = 3A6560530122
bool IsOFF ();
     // Перевірка чи включено пристрій
     // # # Modelld = 3A6560640136
bool  lsON (); private:
     // Змінна для зберігання стану пристрою.
// # # Modelld = 3A65607C0212 DeviceState m_CurrentState;
};
```

Файл CPlantDevice.cpp буде виглядати наступним чином:

```
# include "stdafx.h" # include "CPIantDevice.h".
// # # Modelld = 3A65601C0208 BOOL CPIantDevice :: ON ()
     {
     // TODO: Add your specialized code here.
     // NOTE: Requires a correct return value to compile.
     return TRUE;
     }
     // # # Modelld-3A656044015E
     bool CPlantDevice :: OFF ()
     {
     // TODO: Add your specialized code here.
     // NOTE: Requires a correct return value to compile.
     return TRUE;
     }
     // # # Modelld = 3A6560530l22 BOOL CPlantDevice :: IsOFF ()
     {
     return m_CurrentState == Off;
     }
     // # # tModelld = 3A6560640136
     bool CPIantDevice :: lsON ()
     {
     return m_Curre.ntState == On;
     }
     // # # Modelld = 3A73FAC3'02DO CPIantDevice :: CPIantDevice ()
     {
     //ToDo: Add your specialized code here and/or cal I the base class}
     // # # Modelld = 3A73FAC3037A
     DeviceState CPIantDevice :: get_CurrentState ()
     {
     return m_CurrentState;
     }
```

Додавання початкових станів пристроїв

Тепер у вихідний код конструктора додаємо початкове стан пристроїв. Всі пристрої на початку роботи у нас будуть вимкнені. А в операції ON() і OFF() додаємо присвоєння нового стану пристрою в змінну m_CurrentState. Отриманий код буде виглядати наступним чином:

```
# include "stdafx.h" # include "CPIantDevice.h"
// # # Modelld = 3A65601C0208
bool CPIantDevice :: ON ()
{
m_CurrentState=On; return TRUE;
}
//##Modelld=3A656044015E
bool CPIantDevice::OFF()
{ m_CurrentState=Off;
return TRUE;
}

//##Modelld=3A6560530122
bool  CPIantDevice::IsOFF()
{  return m_CurrentState==Off; }
//##ModelИ-3A6560640136
bool CPIantDevice::lsON()
{  return m_CurrentState==On; }
//##Modelld=3A73FAC302DO
CPlantDevice::CPlantDevice()
{  m_CurrentState=Off; }
//##Modelld=3A73FAC3037A
DeviceState CPlantDevice::get_CurrentState()
{  return m_CurrentState; }
```

І головний клас пристроїв готовий.

Створення операцій ON / OFF для дочірніх класів

Тепер для класів пристроїв, успадкованих з класу CPlantDevice, можна створити операції ON () і OFF (), які повинні бути перевизначені. Для цього, знову ж таки, необхідно активізувати вікно Model Assistant на необхідному класі. Візьмемо для прикладу клас Light.

Тут можна побачити, що з'явилася папка Overrides (перевизначення), в якій можна вибрати ті операції, для яких необхідно перевизначення. Поставимо позначки на ON () і OFF () і оновимо код, для того щоб подивитися, що вийшло.

```
Файл Light.h:
# include "CPIantDevice.h"
// # # Model'ld = 3A6746750352
class Light: public CPIantDevice
```

```
{
public:
// Включення пристрою
// # # ModelId = 3A7401CA0280
 virtual bool ON();
// Вимкнення пристрою
// # # ModelId = 3A7401CA0398
virtual bool OFF();
}
Light.cpp
# include "stdafx.h" # include "Light.h"
// # # ModelId = 3A7401CA0280
 bool Light:: ON )
  {return CPIantDevice :: ON ();}
// # # ModelId = 3A7401CA0398
bool Light ::OFF()
{ return CPIantDevice :: OFF ();
}
```

Зауважимо, що в операціях, перевизначених з головного класу, у вихідний код були включені навіть коментарі, які були введені в головному класі

Порада: Вставляйте коментарі скрізь, де це можливо.

Після цих дій діаграма класів повинна буде прийняти приблизно такий вигляд (рис. 4.3).

При створенні справжньої, а не навчальної програми, для кожного пристрою в операціях ON (), OFF () необхідно описати взаємодію з фізичним пристроєм за допомогою послідовного або іншого порту, але в нашому випадку пристрої віртуальні і не існують насправді, тому ці операції не заповнюються.

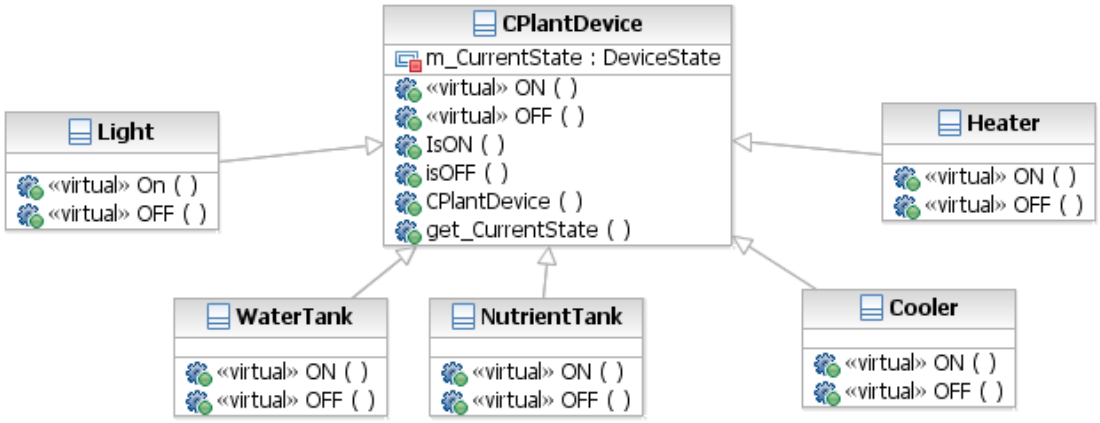

Рис. 4.3. Ієрархія класів виконавчих пристроїв

Створення адреси розташування пристроїв

Згадаймо, що якщо приймати кожен пристрій як фізична, то воно має характеризуватися своїм місцем розташування. Для задання місця розташування у нас вже

створено клас Location, який використовується для задання, розташування датчика температури.

Перша думка, яка приходить в голову - це включити в клас CPlant-Device змінну, що зберігає точку розташування пристрою. У даному випадку це буде поганим рішенням.

Порада: Не реалізовувати перше рішення, що прийшло. Воно буває не оптимально.

Кращим рішенням буде створення класу CDevices, з якого будуть успадковуватися класи виконавчих пристроїв і сенсорів. Цей клас і відповідатиме за обробку розташування пристроїв. При створенні програм без застосування IBM Rational Software Architect ви, напевно, відмовилися б змінювати ієрархію вже готових класів, згадавши про достатньою трудомісткості такої операції, але з Rational Sofware Architect – це сущі дрібниці.

Створимо новий клас CDevices. У цьому класі створимо новий атрибут m_LastID з типом int і вкажемо, що атрибут повинен бути Static.

Для чого потрібен цей атрибут? Припустимо, що кожен пристрій в системі має унікальний номер, який його ідентифікує. Створимо атрибут m_ID з типом int, який буде зберігати ідентифікатор. Для того щоб при створенні пристрою кожного разу не задавати цей номер з великою ймовірністю помилитися, створимо статичний елемент, який буде зберігати останній ID.

Для вказівки розташування ми вже створили клас Location. Необхідно перенести зв'язок Unidirectionlal Association з класу TemperatureSensor у новостворений клас CDevices. Внаслідок ми будемо наслідувати клас сенсорів з класу CDevices і можливість вказівки місце розташування буде і у класу датчиків.

Порада: Після перенесення зв'язку з одного класу в інший на діаграмі класів видалити зайві зв'язку у вікні RClick ==> OpenSpecification => Relofion. Зазвичай для роботи з змінними класу створюються операції.

Скористаємося для цього вікном ModelAssistant і створимо для класу Location операції get_value і set_value.

Тепер залишається вказати наслідування раніше створеного классу CPlant-Device з класу CDevices за допомогою стрілки Generalization. У вас має вийти так, як на рис. 4.4.

Що ми повинні отримати тепер? У нашій гідропонній системі є деяка кількість виконавчих пристроїв, кожне з яких має свій ідентифікатор, унікальний у системі, і своє місце розташування, яке можна встановлювати і зчитувати. Кожне пристрій має два стани включено / вимкнено. Можна перевірити стан пристрою, включити його або вимкнути.

Подивимося на вихідний код головного класу CDevices, який вийде після генерації: RClick => Update Code. Однак не забудьте призначити нові класи в проект VC + + за допомогою Tools => Visual C + + => Component Assignment Tool. В іншому випадку можете отримати приблизно таке повідомлення при генерації коду:

«Warning: Class: CDevices; Skipped, supplier not assigned to VC + + component ».

Заголовочний файл класу CDevices виглядає так:

```
class CDevices
{
protected:
// # # Modelld = 3A581C8B01A4
Location m_location;
```

```
 public:
// # # Modelld-3A752F8F012C
CDevices (); private:
// # # Modelld = 3A77C8EE0046
int m_lD;
// # # Modelld = 3A752F3F0064
static int m_LastlD;};
```

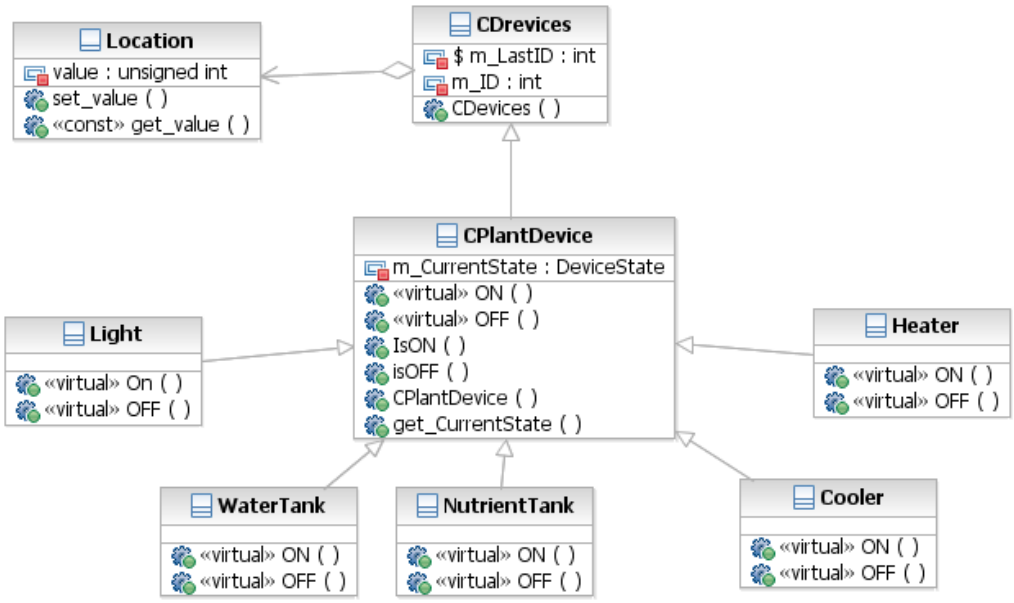

Рис. 4.5. Ієрархія класів пристроїв після додавання CDevices

Агрегування пристроїв

Домовимося, що всі пристрої тепличного господарства входять як агрегати в клас CEnvironmentalControiler (рис. 4.4).

Таким чином, код класу CEnvironmentalController буде наступним.

```
class EnvironmentaIController
{
public:
// CGreenhouseView * pView;
public:
// # # Modelld-3A81B6C602A8
Heater m_Heater;
// # # Modelld = 3A81B6C6023C
Cooler m_Cooler;
// # # Modelld-3A81B6C601FE
Light m_Light;
// # # Modelld-3A81B6C601CC
```

```
pHSensor m_pHSensor;
// # # Modelld = 3A81B6C60190
TemperatureSensor m_TemperatureSensor;
// # # Modelld = 3A81B6C60124
NutrientTank m_NutrientTank;
// # # Modelld = 3A81B6C600F2
WaterTank m_WaterTank;
// tfftModelld = 3A81B6C600B4
GrowingPlan * m_GrowingPlan;
public:
// # # Modelld = 3A81B6C701F4 EnvironmentalControler ();
// # # Modelld = 3A81B6C70154 ~ Environmental Controller ();
// # # Modelld = 3A201BD4032A OnTimer (Condition * condition);
};
```

А код тіла класу повинен бути таким:

```
Environmenta I Controller :: OnTimer (Condition * condition)
{
if (m_TemperatureSensor.currentTemperature ()>
m_GrowingPlan ~> m_Condition-> temperature) {if (m_Heater.
IsON ())
m_Heater.OFF (); if (m_Cooler. IsOFF ())
m_Cooler.ON ();} if ((m_TemperatureSensor.currentTemperature
()) <
(m_GrowingPlan-> m_Condition-> temperature)) {if (m_Cooler.
IsON ())
m_Cooler.OFF (); if (m_Heater. IsOFF ())
m_Heater.ON ();
  }:
if ((m_GrowingPlan-> m_Condition-> temperature) - =
(m_TemperatureSensor.currentTemperature())) {if (m__Cooler.
IsON())
m_Cooler.OFF();
if (m_Heater.IsON())
m_Heater.OFF ();
}
if (m_pHSensor.get_Value () <
m_GrowingPIan-> m_Condition->acidity){if (m_WaterTank.IsON())
m_WaterTank.OFF (); if (m_NutrientTank. IsOFF())
m_NutrientTank.ON();
}
if (m_pHSensor.get_Value ()>
m_Growi ngPIan-> m_Condition-> acidity) {if (m_WaterTank.
IsOFF (0)
m_WaterTank.ON(); if (m_NutrientTank. IsON())
m_NutrientTank.OFF();
```

```
}
if (m_pHSensor.currentpH()=
m_GrowingPIan->m_Condition->acidity){ if (m_WaterTank.lsON())
m_WaterTank.OFF(); if (m_NutrientTank. IsON())
m_NutrientTank.OFF();
}
if ((condition->light ing)==day){ if (m_Light. IsOFF()) )
m_Light.ON();
} else {if (m_Light. IsON()) m_Light.OFF();
}
m_Heater. NextTime(); m_Cooler.NextTime();
m_WaterTank,NextTime(); m_NutrientTank.NextTime();
}
//##Modelld=3A81B6G701F4
EnvironmentaI Controller::EnvironmentalController()
 { m_GrowingPlan = new GrowingPlan(10);
m_TemperatureSensor.calibrate(); m_pHSensor.calibrate(); }
//##Modelld=3A81B6C70154
EnvironmentalController:: ~Environmental.Controller()
{
delete m_GrowingPlan;
}
```

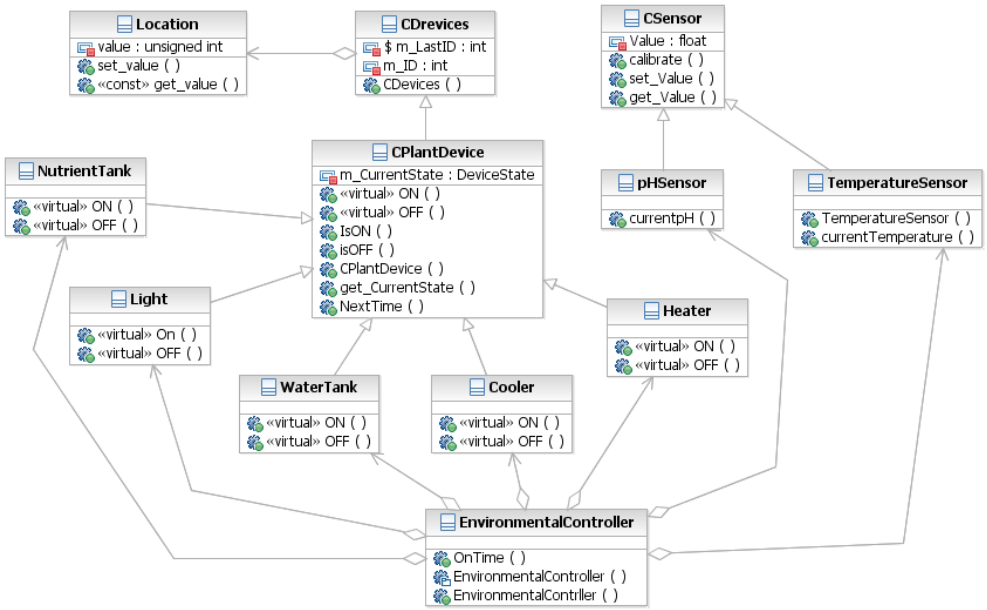

Рис. 4.4. Агрегування класів пристроїв

Коригування відхилень показників

Для того щоб контролер почав коригувати відхилення показників середовища, необхідно змінити код обробника таймера для класу CGreen-houseView, додавши в нього виклик операції m_EnvironmentalController.OnTimer().

```
void CGreenhouseView :: OnTimer (UINT nIDEvent)
```

```
{
GetDocument () -> m__Environment. Change (
& GetDocument () -> m_CurrentCondition,
& GetDocument () -> m_EnvironmentaIControIler);
GetDocument()-> m_EnvironmentaIContrdIler.OnTimer (
& GetDocument () -> m_CurrentCondit ion);
GetDocument () -> m_Environment.Update (
& GetDocument () -> m_CurrentCondition,
& GetDocument () -> m_EnvironmentalControIler);
GetDocument () -> m_Log.Process (
& GetDocument () -> m_CurrentCondition, this);
if (! GetDocument () -> m_pIantTimer.NextTime (
& GetDocument () -> m_CurrentCondition)) {if, (m_TimeMD! = 0)
                { KiIITimer (m_TimerlD); m_TimerlD = 0; }
GetDocument () -> m_Log.Process (
& GetDocument () -> m_CurrentCondition, this);
 Message ("End plan");
}
CEditView :: OnTimer (nIDEvent);
}
```

Після запуску програми можна побачити, як всі відхилення в параметрах середовища тут же починають компенсуватися включенням відповідних виконавчих пристроїв.

За малюнком видно, що виконання плану почалося в 9:00 ранку. При цьому температура і показник pH вже були вище норми. Для зниження температури був включений вентилятор, а для зниження рівня pH відкритий кран для вступу води. При цьому також було включено освітлення теплиці. Поступово температура і рівень pH прийшли в норму, і в 9:10 виконавчі пристрої були вимкнені. У 10:00 рівень pH був у нормі, проте, температура знову піднялася на 1 градус, і знову знадобилося включення вентилятора. Таким чином, тепличне господарство прекрасно працює. Пристрої виконують свої функції, що й було потрібно від програмної системи.

Висновки

Тепер, коли ви отримали уявлення про такому чудовому інструменті, як IBM Rational Software Architect, виникає питання, що робити далі. Звичайно ж, продовжувати вивчення самостійно. Не можна забувати, що ми тільки ознайомилися з основними можливостями, яких уже достатньо для ефективної розробки та супроводу програм. Це тільки верхівка айсберга, який називається IBM Rational Software Architect. Поза нашої уваги залишилися такі інструменти, як Model Integrator для порівняння моделей і з'ясування їх відмінностей, використання макросів для оптимізації роботи. Version Control для супроводу версій моделей, можливості, надаються програмою для роботи групи програмістів над одним проектом і інші, приховані в цьому досить складному і об'ємному пакеті. Міць IBM Rational Software Architect в повній мірі відкривається при спільному використанні програми з іншими продуктами компанії IBM Rational software. Такими як Rational RequisitePro для управління вимог. Rational Test, для створення сценаріїв тестування, Rational SoDa для створення документації, Rational Clear Case для управління версіями. Тож, продовжуємо застосовувати отримані знання на практиці та вивчення продуктів компанії IBM Rational Software.

Література

1.Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер с англ. М.: «Издательство Бином», СПб.: «Невский диалект», 1999 г. 560 с.
2.Страуструп Б. Язык программирования C++: Пер. с англ. М.Радио и связь. 1991 352 с.
3.Шилдт. Г. MFC: основы программирования: Пер. с англ. - К.: BHV, 1997 560 с.
4.Шилдт Г. Теория и практика C++: пер. с англ. СПб.: BHV Санкт-Петербург, 1999. 416 с.
5.Шилдт. Г. MFC: основы программирования: Пер. с англ. – К.: BHV, 1997. - 560 с.
6.Шилдт. Г; Программирование на C и C++ для Windows 95. - К: BHV, 1996. 400 с..
7.Вендров А.М. CASE-технологии. Современные методы и средства проектирования информационных систем. М.: Финансы и статистика, 1998. 176 с.
8.Липаев В. В. Системное проектирование сложных программных средств для информационных систем. М.: СИНТЕГ, 1999. 224 с.
9.Кратчен Ф. Введениев Rational Unified Process. 2 изд.: Пер. с англ. М.:Вильямс, 2002. 240 с.
10.Ларман К. Применение UML и шаблонов проектирования. Пер с англ. М.: Вильямс, 2001
11.Фраулер М., Скотт К. UML в кратком изложении. Применение стандартного .языка объектного моделирования: Пер. с англ. - М.: Мир, 1999.

Навчальний посібник

Сучасні технології ООП-проектування та автоматичного генерування програмного коду

Упорядники:

М.. Петрик,
І. Мудрик
О.. Петрик
Ю. Стоянов

Відповідальний за випуск М. Петрик

Видавництво ТНТУ ім. Івана Пулюя

46000, Тернопіль, вул. Руська 56

2018

49