Лабораторная работа № 12
Создание кода на C++

Вступительные замечания

На основе диаграммы классов Rational Rose позволяет создавать код класса на выбранном языке. Для того чтобы воспользоваться данной возможностью, необходимо убедиться, что выбранный язык программирования установлен при помощи Add-Ins менеджера.

Ранее мы изучали полученный код класса после изменения установок диаграммы классов. Мы посмотрим, как необходимо изменить спецификации, для того чтобы получить определенный код, то есть будем изначально исходить из кода класса, и рассмотрим необходимые шаги для его получения.

Нужно понимать, что действия, которые необходимо произвести для создания кода на одном языке программирования, скорее всего не подойдут для работы с другими языками. Поэтому сначала разберем более универсальный вариант создания кода на C++, независимого от используемого компилятора.

Эталонный код класса

Вернемся к нашей теплице и вспомним, что датчиков в теплице может быть несколько, и поэтому необходимо точное определение местоположения датчика. Используем для этого его номер. На самом деле для определения местоположения датчика могут использоваться координаты, которые помогут нам вывести показания конкретного датчика на дисплей, но пока используем только номер.

Для установки текущего местоположения будем использовать тип Location, а для температуры — тип Temperature.

Допустим, что мы хотим получить следующий код на C++:

```
// температура по Цельсию
typedef float  Temperature;
// число, однозначно определяющее положение датчика
typedef unsigned  Location;
class  TemperatureSensor {
public:
TemperatureSensor(Location);
_ TemperatureSensor():
void calibrate (Temperature actual Temperature);
Temperature currentTemperature() const;
};
```

Для объектов, создаваемых в программе, или как их еще называют — реализаций классов, удобно использовать псевдонимы простых типов, например, Temperature или Location вместо unsigned int. Таким образом, мы можем описать получаемые абстракции на языке предметной области.

В конструктор передается местоположение датчика, имеется возможность калибровки и получения измеренной температуры.

Ассоциация класса с языком C++

В Rational Rose все типы по умолчанию определяются как классы. Поэтому создадим два новых класса: Location и Temperature. для каждого из них проделаем следующее:

Выберем класс, затем Menu: Tools=>C++=>Code Generation. Появится диалоговое окно, показанное на рис. 90.

Здесь необходимо выбрать класс, для которого назначается язык программирования, и нажать Assing (назначить).

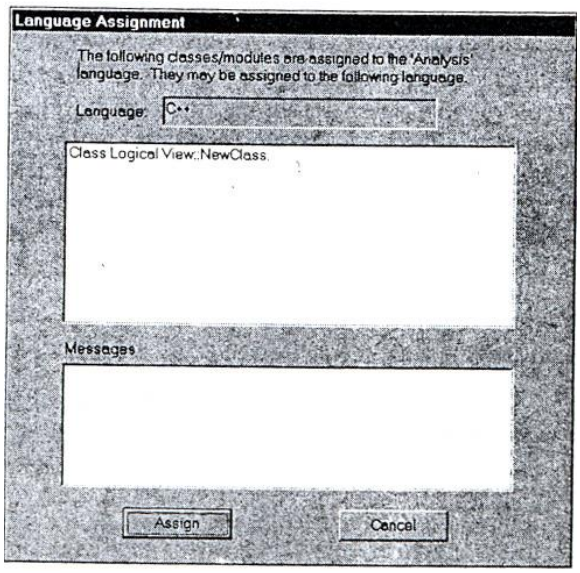

Рис.90. Назначение классу языка программирования C++

Просмотр кода класса

После генерации кода в контекстном меню станет доступен дополнительный пункт — C++, в котором можно просмотреть заголовочный файл (Header).h и файл тела класса (Body) .cpp, как показано на рис. 91.

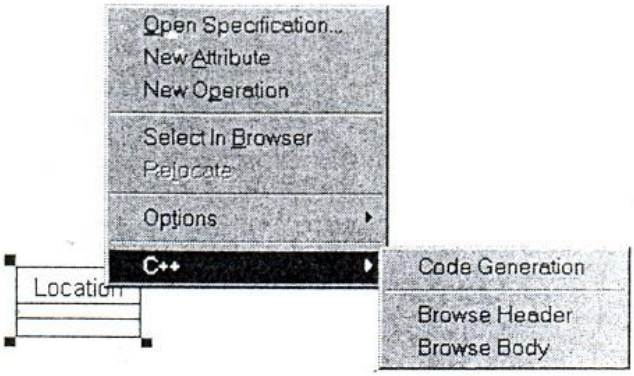

Рис. 91. Доступное меню C++

Теперь можно просматривать файлы тела класса и заголовочный файл после каждого внесенного изменения, чтобы проверить, как эти изменения отразятся на получаемом коде.

Установка типа объекта

Теперь во вновь появившемся меню выбираем Open Specification=> C++=>Implementation Type=>Override затем в графе Value заполняем unsigned int. Должно получиться так, как показано на рис. 92.

Проделайте то же самое с классом Temperature, только не забудьте установить тип float и запустите генерацию кода RClick=C++=>Code Generation. Просмотрите заголовок, и вы увидите, что получилось именно то, что заказывали.

Теперь мы можем воспользоваться полученными типами для класса датчика температуры.

Добавление новых операций

Выбираем TemperatureSensor=>RClick=>New Operation и вводим имя calibrate (actualTemperature : Temperature): void. Отметим, что в отличие от семантики языка C++, здесь сначала указывается переменная, а затем, после двоеточия, ее тип, аналогично и

возвращаемое значение указывается после операции через двоеточие. Слева от операции появился значок, если его выбрать, то открывается набор значков, которые отражают доступность операции, соответственно: public, protected, private и implementation. В последнем случае, если элемент определен в контейнере, он будет виден только для объектов, определенных в этом контейнере.

Аналогично добавим конструктор и операцию получения температуры.

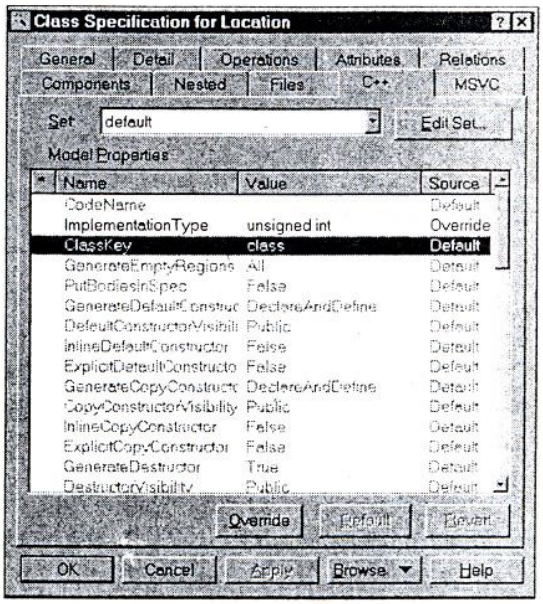

Рис.92. Заполнение свойства Implementation Type

Установка зависимости классов

В нашем случае для связи классов используется компиляционная зависимость, которая подразумевает, что в класс датчика температуры будут включены определения необходимых типов.

Для того чтобы показать, что в классе датчика температуры должны использоваться типы Location и Temperature, воспользуемся связью Dependency, для чего выберем ее значок из строки инструментов.

Щелкаем на нем, затем щелкнем по классу TemperatureSensor и, не отпуская кнопку мыши, тянем линию до класса Location. Аналогично с классом Temperature. При генерации исходного текста в этом случае Rational Rose автоматически включит файлы location.h и temperature.h в заголовочный файл TemperatureSensor.

У вас должно получиться изображение, аналогичное рис. 93.

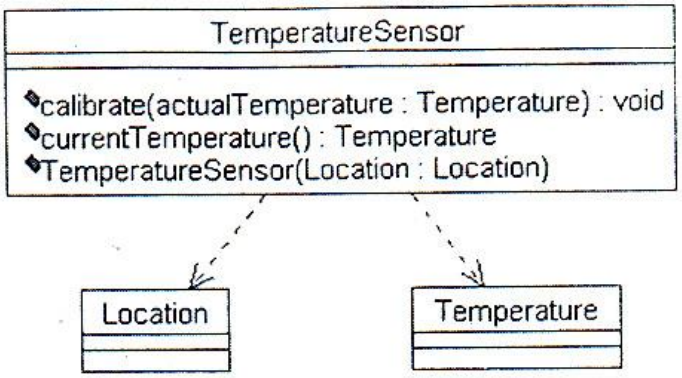

Рис.93. Зависимости класса TemperatureSensor

Теперь для созданного класса можно запустить генерацию исходного кода и сравнить его с тем, что нам необходимо.

Доводка полученного кода

Если посмотреть на полученный файл заголовка, то можно увидеть, что получилось не совсем то, что нужно. Приведем полученный файл без некоторой служебной информации, которую включает генератор в код класса. Вся служебная информация вставляется как комментарии и не влияет на последующую генерацию исполняемого файла.

```
#include “Location.h”
#include “Temperature.h”
// Датчик температуры, измеряет температуру е теплице в Цельсиях
class TemperatureSensor {
public:
        //##Constructors(generated)
        TemperatureSensor();
        TemperatureSensor(const TemperatureSensor&right);
        //##Constructors(specified)
        TemperatureSensor(Location Location);
        //##Destructor(generated)
        _ TemperatureSensor():
        //## Other Operations(specified)
        void calibrate (Temperature actual Temperature);
        Temperature currentTemperature();
}
```

Здесь мы видим, что в код включились заголовочные файлы определения данных и комментарий (это было заполнено поле RClick=>Open Specification=>Documentation). Однако конструкторов в классе целых три: два автоматически созданных и один, который определили мы. И не хватает служебного слова const в операции currentTemperature.

Исправим это несоответствие. Сначала удалим автоматически создаваемые конструкторы класса: Wind:Class Diagram=>TemperatureSensor=>RClick=>Open Specification=>C++=> Generate Default Constructor=> DoNotDeclare и там же Generate Copy Constructor = =DoNotDeclare. Поставим тип const: Wind:Browser+Logical View+ TemperatureSensor => currentTemperature=>C++=>Operation Is Const = True.

Проверьте полученный исходный код и убедитесь, что все сработало как нужно.

Настройка свойств C++

Пока мы проделали все эти действия без подробных объяснений, для того чтобы показать возможности Rational Rose по созданию кода приложения на C++. для того чтобы вы могли пользоваться возможностями C++, далее опишем назначение этих свойств, список которых доступен во вкладке C++ спецификаций класса.

- CodeName устанавливает имя класса в создаваемом коде. данное свойство необходимо устанавливать только в том случае, если имя класса должно быть отлично от имени заданного в модели Rational Rose. Данное свойство необходимо использовать для создания работоспособного кода C++, если для классов в модели используются русские имена.
- ImplementationType позволяет использовать простые типы вместо определения класса, устанавливаемого Rational Rose по умолчанию. При задании этого параметра создается директива typedef.
- ClassKey используется для задания типа класса, такого как class, struct или union. Если тип не указан, то создается класс.
- GenerateEmptyRegion — свойство указывает, как будет создаваться пустой раздел protected: None — пустой раздел не будут создан; Preserved — пустой раздел будет создан, если будет установлено свойство preserve-yes; Unpreserved — пустой раздел

будет создан, если будет установлено свойство preserve-no; All — всегда будет создаваться.

- PutBodiesInSpec если установлено как True, то в заголовочный файл попадет и описание тела класса. Используется для компиляторов, которым необходимо определение шаблона класса в каждом компилируемом файле.
- GenerateDefaultConstructor позволяет установить, необходимо ли создавать конструктор для класса по умолчанию. Может принимать следующие значения: DeclareAndDefine — создается определение для конструктора и скелет конструктора в теле класса; Declare Only — создается только определение; DoNotDeclare — не создается ни определения, ни скелета конструктора.
- DefaultConstructorVisibility устанавливает раздел, в котором будет определен конструктор по умолчанию: public, protected, private, implementation.
- InlineDefaultConstructor устанавливает, будет ли конструктор по умолчанию создаваться как inline подстановка.
- ExplicitDefaultConstructor устанавливает конструктор по умолчанию как explicit (явно заданный).
- GeneralCopyConstructor устанавливает, будет ли создана копия конструктора.
- CopyConstructorVisibility устанавливает раздел, в котором будет создана копия конструктора.
- InlineCopyConstructor устанавливает, будет ли копия конструктора создаваться как inline подстановка.
- ExplicitCopyConstructor устанавливает, что копия конструктора будет создана explicit (явно задана).
- GenerateDestructor устанавливает, будет ли создаваться деструктор для класса.
- DestructorVisibility устанавливает раздел, где будет создаваться деструктор.
- DestructorKind устанавливает вид создаваемого деструктора: Common — обычный, Virtual — виртуальный, Abstract — абстрактный.
- InlineDestructor устанавливает, будет ли деструктор создаваться как inline подстановка.
- GenerateAssigmentOperation устанавливает, будет ли создаваться функция переопределения оператора присваивания (=).
- AssigmentVisibility определяет раздел, где будет создаваться функция оператора присваивания.
- The AssigmentKind определяет вид функции оператора присваивания: Common — обычная, Virtual — виртуальная, Abstract — абстрактная, Friend — дружественная.
- InlineAssigmentOperation определяет, будет ли оператор присваивания создаваться как inline.
- GenerateEqualityOperations определяет, будут ли переопределяться операторы сравнения на равенство (= = и !=).
- EqualityVisibility определяет раздел, в который будут помещены операторы сравнения на равенство.
- EqualityKind определяет вид функций операторов сравнения на равенство: Common — обычная, Virtual — виртуальная, Abstract — абстрактная, Friend — дружественная.
- InlineEqualityOperations определяют, будут ли функции операторов сравнения на равенство создаваться как inline.
- GenerateRelationalOperations определяет, будут ли переопределяться операторы сравнения (<, <=, >,> =).
- RelationalVisibility определяет раздел, в который будут помещены операторы сравнения.

- RelationalKind определяет вид функций операторов сравнения: Common — обычная, Virtual — виртуальная, Abstract — абстрактная, Friend — дружественная.
- InlineRelationalOperations определяет, будут ли функции операторов сравнения создаваться как inline подстановка.
- GenerateStorageMgmtOperations определяет, будут ли переопределяться операторы new и delete в классе.
- StorageMgmtVisibility определяет раздел, в который будут помещены операторы new и delete.
- InlineStorageMgmtOperations определяет, будут ли операторы new и delete определены как inline подстановка.
- GenerateSubscriptOperations определяет, будет ли переопределен оператор [].
- SubscriptVisibility определяет раздел, в который будет помещен оператор [].
- SubscriptKind определяет вид функций оператора []: Common — обычная, Virtual — виртуальная, Abstract — абстрактная.
- SubscriptResultType определяет тип возвращаемого выражения оператора [].
- InlineSubscriptOperation определяет будет ли оператор [] определен как inline подстановка.
- GenerateDereferenceOperation определяет будет ли переопределен оператор *.
- DereferenceVisibility определяет раздел в который будет помещен оператор *.
- DereferenceKind определяет вид функций оператора *: Common — обычная, Virtual — виртуальная, Abstract — абстрактная.
- DereferenceResultType определяет тип возвращаемого выражения для оператора *.
- InlineDereferenceOperation определяет будет ли оператор * определен как inline подстановка.
- GenerateIndirectionOperation определяет будет ли переопределен оператор —>.
- IndirectionVisibility определяет раздел в который будет помещен оператор —>.
- IndirectionKind определяет вид функций оператора →: Common — обычная, Virtual — виртуальная, Abstract — абстрактная.
- IndirectionResultType определяет тип возвращаемого выражения для оператора ->
- InlineIndirectionOperation определяет будет ли оператор -> определен как inline подстановка.
- GenerateStreamOperations определяет будут ли переопределены операторы потоков (<< и >>).
- StreamVisibilityопределяет раздел, в который будут помещены операторы потоков.
- InlineStreamOperations определяет, будут ли операторы потоков определены как inline подстановка.