ТЕМА 9

План лекции:

1. Диаграмма Component в Rational XDE.
2. Диаграмма Class в Rational XDE.

9.1 Диаграмма Component в Rational XDE

Количество инструментов диаграммы компонентов в *Rational XDE* по сравнению с *Rational Rose* сокращено. Здесь используется только два основных значка для обозначения компонентов: *Component и Component Realization*. Поскольку *.NET* – полностью объектно-ориентированная среда разработки, при создании программ C# не используется ни разделение файлов, ни отдельное обозначение подпрограмм, как это было в случае построения диаграмм компонентов в Rational *Rose*.

Диаграмма компонентов предназначена для отображения зависимостей между компонентами системы и находящимися в них классами. Для построения диаграммы нужно выбрать из контекстного меню модели *Add Diagram=> Component.* На рис. 9.1 приведен Toolbox диаграммы Component.

Рассмотрим основные элементы диаграммы компонентов.

Значок *Component* позволяет создать на диаграмме отображение компонентов. Компонент – это элемент реализации с четко определенным интерфейсом. Компонентами могут быть любые библиотечные или программные файлы, содержащие реализации классов системы.

Значок *Dependency* позволяет создать на диаграмме отображение использования одного компонента другим или их зависимость друг от друга.

Инструмент *Interface* отображает на диаграмме интерфейс. Это список операций, посредством которых компоненты взаимодействуют между собой.

Элемент *Realization* показывает на диаграмме реализацию интерфейса компонентом.

Значок *Association* отражает на диаграмме ассоциации элементов.

Инструмент *Reside* создает на диаграмме отображение принадлежности класса к компоненту.

Значок *Component Instance* предназначен для создания на диаграмме отображения экземпляра компонента.

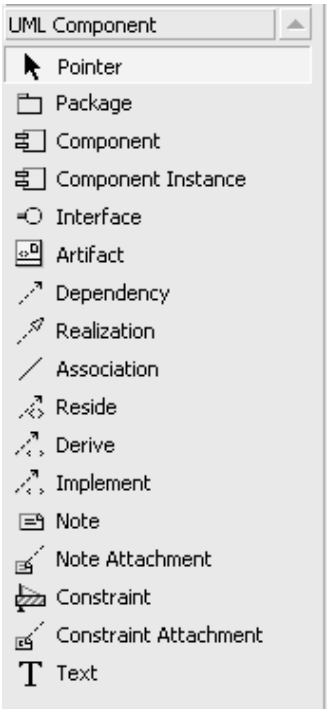

Рисунок 9.1 – Toolbox диаграммы Component

9.2 Проектирование классов приложения с помощь диаграммы Class

Изучив все диаграммы, предназначенные для построения модели системы, можно переходить к разработке диаграммы классов, которая считается основной в создании приложения ASP.NET. Диаграмма *Class* выполняет целый ряд функций: используется для создания иерархии классов; на ее основе строятся модели данных и проектируется структура Web-приложений; на этапе анализа и проектирования используется для создания диаграмм реализации прецедентов; с ее помощью создается модель предметной области, которая используется на этапе анализа. Кроме того, широко применяются стереотипы классов, позволяющие адаптировать стандартную UML-диаграмму для конкретных целей, расширяя ее возможности. Рассмотрим основные этапы разработки системы с помощью диаграммы классов.

Для создания модели используются три стереотипа классов, которые определяют их назначение: граничный класс (*boundary*), сущность (*entity*), управление (*control*).

Стереотип *граничный класс* показывает, что класс предназначен для взаимодействия с внешними актерами и стоит на границе системы, поэтому и называется граничным. Такой класс, получая сообщение от внешнего актера, транслирует их внутрь системы, генерируя и передавая соответствующие сообщения другим классам.

Классы *сущности* используются для моделирования классов, которые отвечают за хранение определенной информации. Эти классы реализуют

возможности по получению, изменению и сохранению информации в базе данных. Классы *сущности* обычно не отражаются ни на одной диаграмме прецедентов, но требуются для выполнения внутреннего хранения данных.

Классы *управления* используются для координации работы других классов приложения. Поведение этих классов обычно реализует один или несколько прецедентов, показанных на диаграммах моделирования. Классы *управления* реализуют поведение системы при помощи потоков управления. Они являются промежуточными звеньями между *граничными* классами и классами *сущностями*.

На этапе проектирования диаграмма классов используется для проектирования подсистем и иерархии классов. Одна или несколько диаграмм классов описывают классы верхнего уровня. При включении на диаграмму пакетов в модель добавляются диаграммы классов, описывающие содержимое пакетов. Пакеты обычно используют для группирования классов по подсистемам. Кроме классов, в подсистемы могут включаться реализации вариантов использования, интерфейсы и другие подсистемы. Разделение на подсистемы значительно упрощает параллельную разработку, конфигурирование и инсталляцию конечного продукта. Создание подсистем позволяет проще устанавливать различные категории доступа к информации для пользователей, а также отделить алгоритмы для организации связи с внешними продуктами.

Для разрабатываемого виртуального книжного магазина можно выделить следующие подсистемы: *Управление регистрацией, Управление каталогами, Управление заказами, Управление сервисными функциями.*

При разработке приложения .NET интерфейс обозначает полноценные объекты, которые проектируются наравне с классами системы. В .NET интерфейс – это элемент, включаемый в классы, которые должны его реализовывать. Такой подход позволяет разрабатывать интерфейсы отдельно, а затем включать их в нужные классы. На структуру интерфейса накладываются ограничения. В его состав входят только абстрактные члены, в нем могут быть определены события, методы, свойства, но он не может содержать конструкторов, деструкторов и констант.

Следующий этап работы с диаграммой *Class* – это описание логического представления системы. Обычно диаграмма классов создается для всех классов системы в отличие от других диаграмм, которые строятся для отдельных объектов со сложным поведением и взаимодействием. Классы на диаграмме могут представлять любые C#-классы: простые, параметризованные или метаклассы.

Рассмотрим работу с диаграммой *Class* при проектировании подсистем виртуального магазина. На рис. 9.2 приведен *Toolbox* диаграммы классов, где доступны все необходимые элементы.

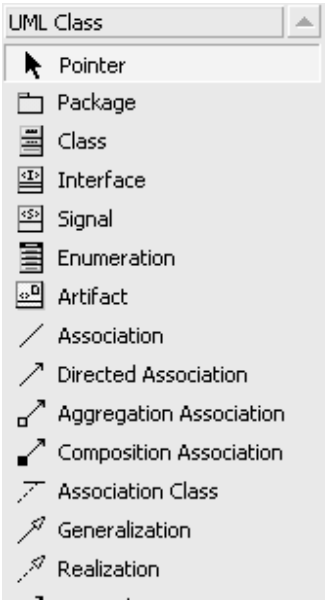

Рисунок 9.2 – Toolbox диаграммы Class

Рассмотрим назначение и возможности инструментов диаграммы *Class*.

Для создания классов используется значок *Class*. Важным достоинством работы в *Rational XDE* является возможность одновременного просмотра диаграммы в графическом виде и получаемого по этой диаграмме исходного кода (рис. 9.3). Для этого необходимо воспользоваться пунктом меню *Window=>New Horizontal Tab.*

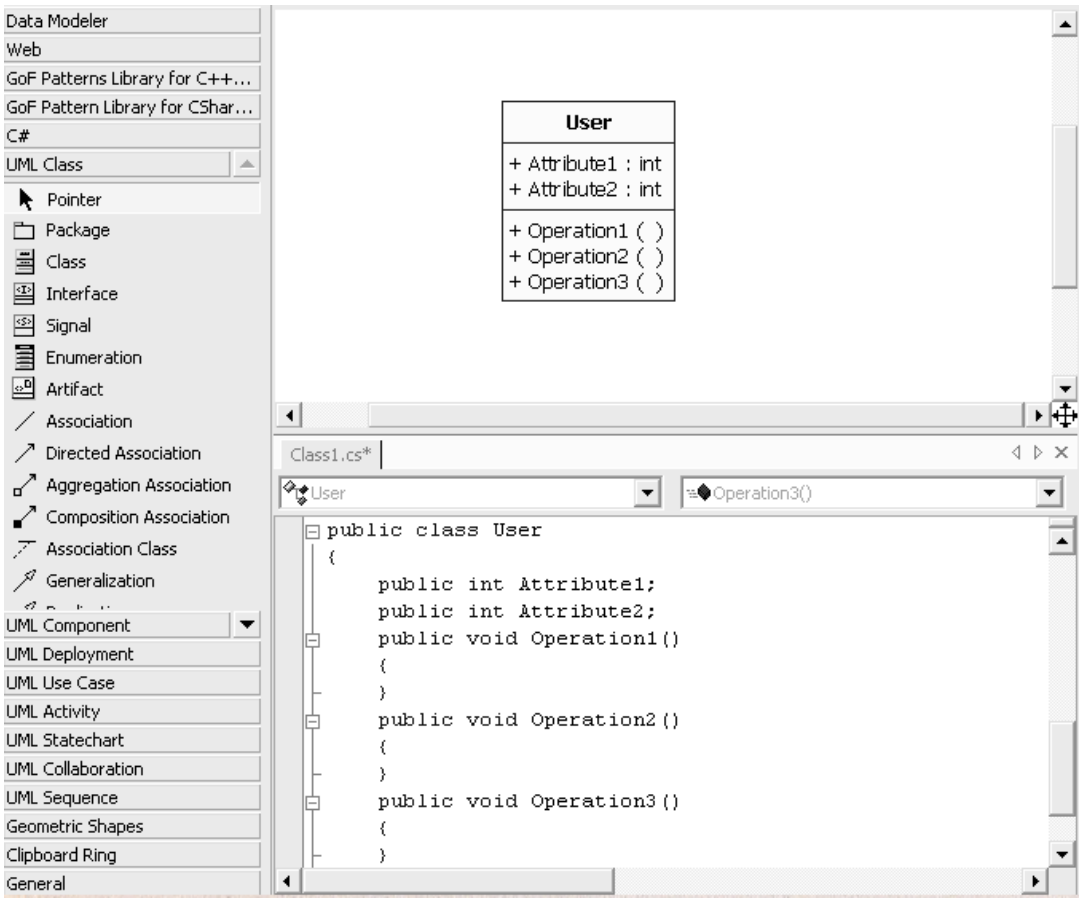

Рисунок 9.3 – Работа с классом в среде Rational XDE

Добавлять атрибуты и операции в класс можно двумя способами: во-первых, посредством контекстного меню *Add UML => Attribute (Operation)*, а во-вторых, с помощью пункта контекстного меню *Add C-Sharp*. Вторым способом можно добавлять все элементы, которые необходимы для создания приложения, а не только свойства и операции. Добавим классу *Account* свойство *Name*, для чего выберем *Add C-Sharp=>Property*. Соответствующее этому окно показано на рис. 9.4. Видно, что в окне можно ввести ряд данных, относящихся к свойству класса. Автоматически Rational XDE добавляет операции доступа к свойству, создавая необходимый для этого исходный код. Поле *Associated Field* позволяет создать скрытый атрибут класса, к которому осуществляется доступ посредством операций *Set* и *Get*. Также в этом окне можно настроить параметры для создания свойства.

В C# каждый класс может содержать в себе определения других элементов, а именно: классов, интерфейсов, структур и перечислений. При помощи пункта контекстного меню *Add C-Sharp* возможно добавление этих элементов и делегатов, не выходя из диаграммы. Делегаты в C# представляют собой контейнеры, где хранятся описания метода класса. Они позволяют вызывать статические методы класса под именем делегата, скрывая от клиента информацию о том, что он пользуется методом класса.

Окно *Collection*, которое активизируется при вызове пункта контекстного меню *Collection Editor*, предоставляет доступ к настройке элементов класса, таких, как атрибуты, операции, связи и другие элементы, которые невозможно отредактировать через окно свойств, поскольку они имеют списочную структуру. Просматривать свойства списком удобно, если на диаграмме элемент не показан или его свойства скрыты.

Перейдем к рассмотрению остальных инструментов диаграммы классов. Значок *Interface* позволяет создать элемент интерфейса. Он представляет собой абстрактный контейнер абстрактных операций, которые должны быть реализованы в классе. Значок *Signal* позволяет создать элемент сигнала, который отражает асинхронное сообщение от одного класса к другому, т.е. без ожидания ответа. Значок *Enumeration* создает элемент перечисление. Это тип данных, состоящий из набора констант базового типа. Перечисления используются для создания набора именованных страниц.

Классы не могут существовать обособленно. Основная их задача – взаимодействовать друг с другом при помощи обмена сообщениями. При построении модели классы связываются друг с другом связями различных видов. Согласно спецификации UML таких связей довольно много, однако в отличие от C++ в C# многие виды связей дают одинаковые результаты при генерации исходного кода.

Значок *Association* показывает ассоциации классов. Поскольку направление не указано, Rational XDE не может определить, какой класс первичен в установленной ассоциации, поэтому считает оба класса равноправными. Этот вид связи не используется для проектирования классов, по которым планируется получить исходный код, т.к. он никак на него не влияет.

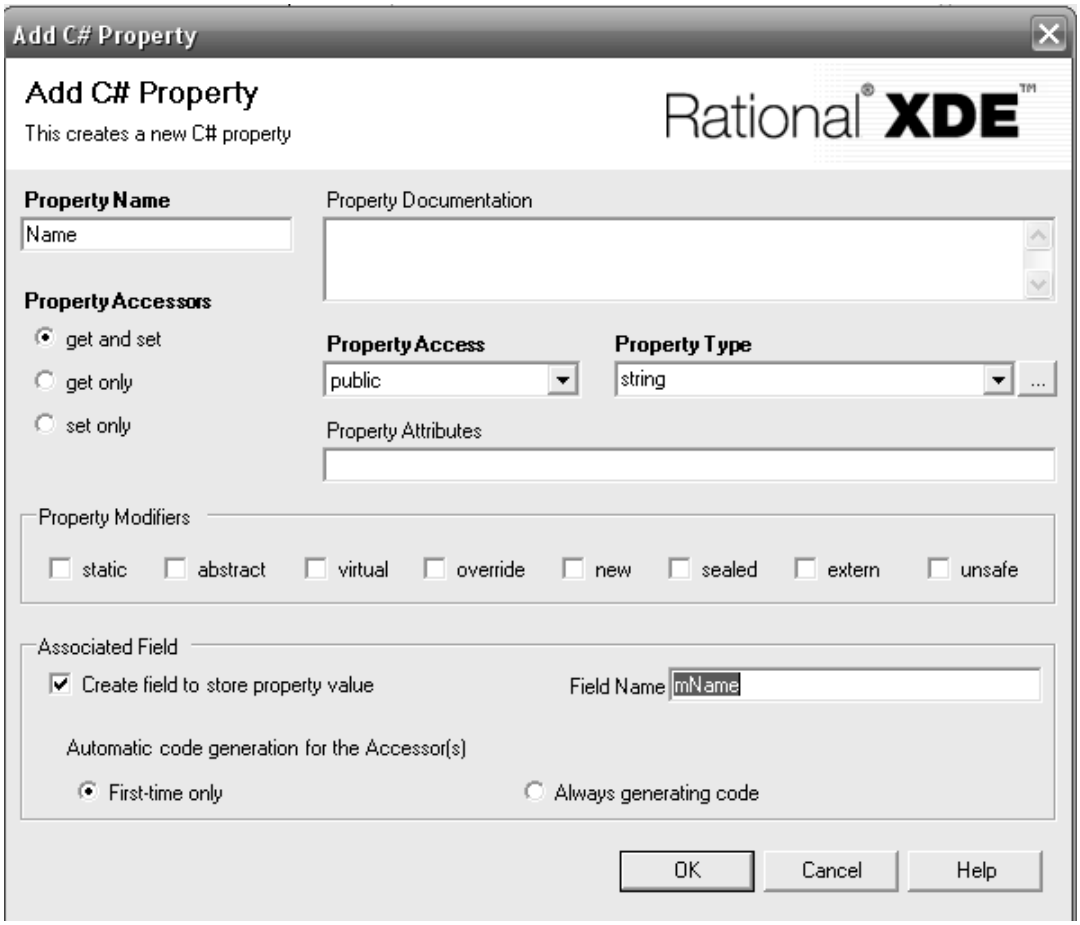

Рисунок 9.4 – Окно Rational XDE C# Property

Значок *Directed Association* позволяет создать направленную ассоциацию между классами. Этот тип связи показывает, что объект одного класса включается в другой класс для получения доступа к его методами. Rational XDE определяет название переменной для объекта класса, и при генерации исходного кода эта переменная будет включена в него перед определением методов и атрибутов класса.

Значок *Aggregation association* позволяет отразить на диаграмме агрегацию элементов. Этот тип связей показывает, что один элемент входит в другой как часть. Агрегация используется при моделировании как дополнительное средство, показывающее, что элемент состоит из отдельных частей.

Значок *Composition* позволяет отразить состав объекта и элементы, включенные в композицию.

Значок *Association Class* используется для отображения класса, ассоциированного с двумя другими. Фактически данный тип связи отражает то, что некоторый класс со своими атрибутами включается как элемент в два других, при этом никак не отражаясь на создаваемом исходном коде.

Значок *Realization* отражает на диаграмме отношение между классом и интерфейсом, который этим классом реализуется.

Значок *Dependency* показывает на диаграмме такое отношение зависимости, когда один класс использует объекты другого. Для классов C# этот тип связи не имеет широкого применения и на создаваемый код не влияет.

Значок *Generalization* указывает, что один класс является родительским по отношению к связанному, при этом будет создан код наследования класса.

Значок *Bind* создает особый тип зависимости между классами и используется для работы с шаблонами классов. Эта связь показывает замещение параметров, определенных в шаблоне.

Значок *Usage* отражает тип зависимости, показывающей, что один из элементов модели требует наличия другого, связанного с ним такой связью.

Значок *Friend Permission* строит тип зависимости, показывающий, что один класс предоставляет доступ к своему содержимому другому классу.

Значок *Abstraction* определяет, что элементы модели представляют одно понятие, но на разных уровнях абстракции.

Значок *Instantiate* позволяет показать, что элемент модели отражает особый случай другого элемента.