

ТЕМА 8

План лекции:


1. Диаграмма Component в Rational Rose.
2. Диаграмма Class в Rational Rose.
3. Связи на диаграмме Class.

8.1 Построение диаграммы Component в Rational Rose

Диаграмма компонентов позволяет создать физическое отражение текущей модели. *Component* показывает организацию и взаимосвязи программных компонентов, представленных в файлах различных типов, а ее связи отражают зависимости одного компонента от другого. В текущей модели может быть создано несколько диаграмм компонентов для отражения пакетов, компонентов верхнего уровня или описания содержимого каждого пакета компонентов.

Для систем, состоящих из большого количества классов, целесообразно строить диаграмму компонентов, когда определены все связи классов и структура наследования. Но поскольку на всем протяжении проектирования системы, вплоть до выхода готового программного продукта в диаграммы будут вноситься изменения, оправдано создание *Component* для нескольких классов, чтобы получить практику работы с данным типом диаграмм.

В *Rational Rose* заложена возможность работы с программными библиотеками, их можно как создавать, так и пользоваться уже готовыми. Необходимо только указать, какие классы, в каких компонентах будут находиться. Для того чтобы обеспечить минимальные трудозатраты на разработку и сопровождение, тесно связанные между собой классы собираются в библиотеки.

Диаграмму компонентов можно построить двумя способами: с помощью меню *Browse=> Component diagram* или воспользовавшись значком *Component diagram*  на панели инструментов. После чего будет активизировано диалоговое окно выбора диаграммы, посредством которого создается, удаляется, переименовывается диаграмма.

Рассмотрим построения *Component diagram* на примере системы обслуживания банкоматов архитектуры клиент-сервер (рис. 8.1), на диаграмме показаны компоненты клиента. Система реализуется на языке программирования *Visual C++*. У каждого класса имеется свой собственный заголовочный файл и файл с расширением. *cpp*, поэтому каждый класс преобразуется в свои собственные компоненты на диаграмме, представляющие тело и заголовок класса. Темные компоненты соответствуют файлам тела класса на *Visual C++*, прозрачные компоненты – заголовочным файлам классов языка *Visual C++*.

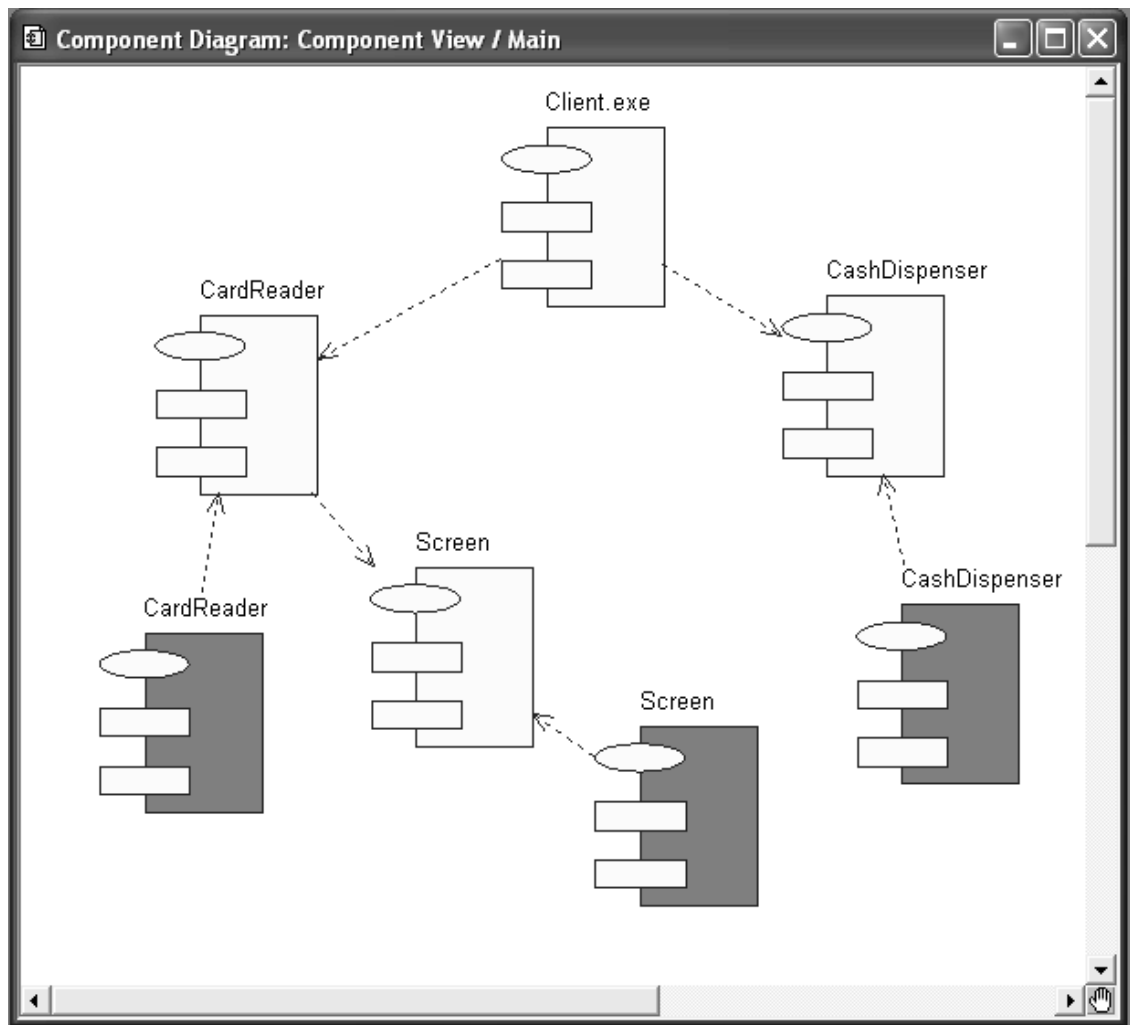











Рисунок 8.1 – Диаграмма компонентов в Rational Rose


Для работы исполняемого файла *Client.exe* необходимы заголовочные файлы *CardReader* и *CashDispenser*, для того чтобы класс *CardReader* мог быть скомпилирован, класс *Screen* должен уже существовать. В свою очередь, заголовочные классы *CardReader*, *CashDispenser* и *Screen* используются для компиляции соответствующих им файлов на языке *Visual C++*. После компиляции всех классов может быть создан исполняемый файл *Client.exe*.

Для построения приведенной диаграммы компонентов были использованы некоторые из ее инструментов. Значок *Package specification*  позволяет отобразить определение пакета, а значок *Package body*  выполняет описание тела пакета. Обычно эти инструменты связаны между собой. Здесь *Package specification* – заголовочный файл с расширением *.h*, а *Package body* – файл с расширением *.cpp*. С помощью значка dependency  устанавливаются связи между компонентами. Этот тип связи показывает, что классы, содержащиеся в компоненте-клиенте, наследуются, содержат элементы, используют или каким-либо другим образом зависят от классов, которые экспортируются из компонента-сервера.

Строка инструментов диаграммы компонентов содержит еще несколько элементов, позволяющих отражать программную реализацию системы. Значок  - *Component* (компонент) на диаграмме представляет собой модуль программного обеспечения, такой как исходный код, двоичный файл, выполняемый файл, динамически подключаемые библиотеки. Компоненты могут использоваться для показа взаимосвязи модулей на этапе компиляции или выполнения программы, а также показывать, какие классы используются для создания определенных компонентов. Значок  - *Package* (пакет) позволяет отобразить пакет, который объединяет группу компонентов в модели. Значок  - *Main program* (главная программа) позволяет добавить в модель компонент, обозначающий главную программу. Значок  - *Subprogram body* (тело подпрограммы) позволяет добавить в модель компонент, обозначающий тело подпрограммы. Значки  и  - *Task specification / body* (определение / тело задачи) позволяют отобразить независимые потоки в многопоточковой системе.

8.2 Построение диаграммы Class

Диаграмма классов является основной для создания кода приложения. С ее помощью строится внутренняя структура системы. Обычно данная диаграмма строится для всех классов, становясь логической моделью системы. Кроме того, *Rational Rose* позволяет на основе *Class diagram* создавать исходный код приложения на любом языке программирования, который поддерживается генератором кода *Rational Rose*. С ее помощью возможно изменение свойств любого класса или его связей, при этом диаграммы или спецификации, связанные с изменяемым классом, будут автоматически обновлены.

Главная диаграмма классов (*Main*) уже присутствует во вновь созданной пустой модели, но возможно создание дополнительных диаграмм посредством контекстного меню *Logical View* в окне *Browse* или при помощи кнопки *Class diagram* . Создание нового класса и помещение его на диаграмму выполняется с помощью соответствующего значка на строке инструментов или из меню *Tools => Create=> Class*. После добавления класса в диаграмму становится доступно его контекстное меню. Содержание меню изменяется при ассоциации класса с разными языками программирования.

Рассмотрим пункты меню для класса и свойства класса, не ассоциированного с каким-либо языком программирования (рис. 8.2, а, б).

Назначение отдельных пунктов меню:

- *Open Specifications* – открытие диалогового окна заполнения спецификаций.
- *Sub Diagrams* – создание для текущего класса диаграммы активности и состояний или переход на поддиаграммы класса.

- *New Attribute* – добавление нового атрибута классу.
- *New Operation* – добавление новой операции классу.
- *Select in Browser* – выделение класса в окне *Browser*.
- *Relocate* – перемещение класса в новый пакет или на новое местоположение.
- *Options* – вызов подменю настройки значка класса.
- *Format* – вызов подменю настройки шрифта, цвета, заливки диаграммы.

Rational Rose позволяет устанавливать значительное количество свойств класса, которые влияют на генерацию его кода. Спецификация класса имеет несколько вкладок, и первой из них активизируется вкладка *General* (рис. 8.2, б).

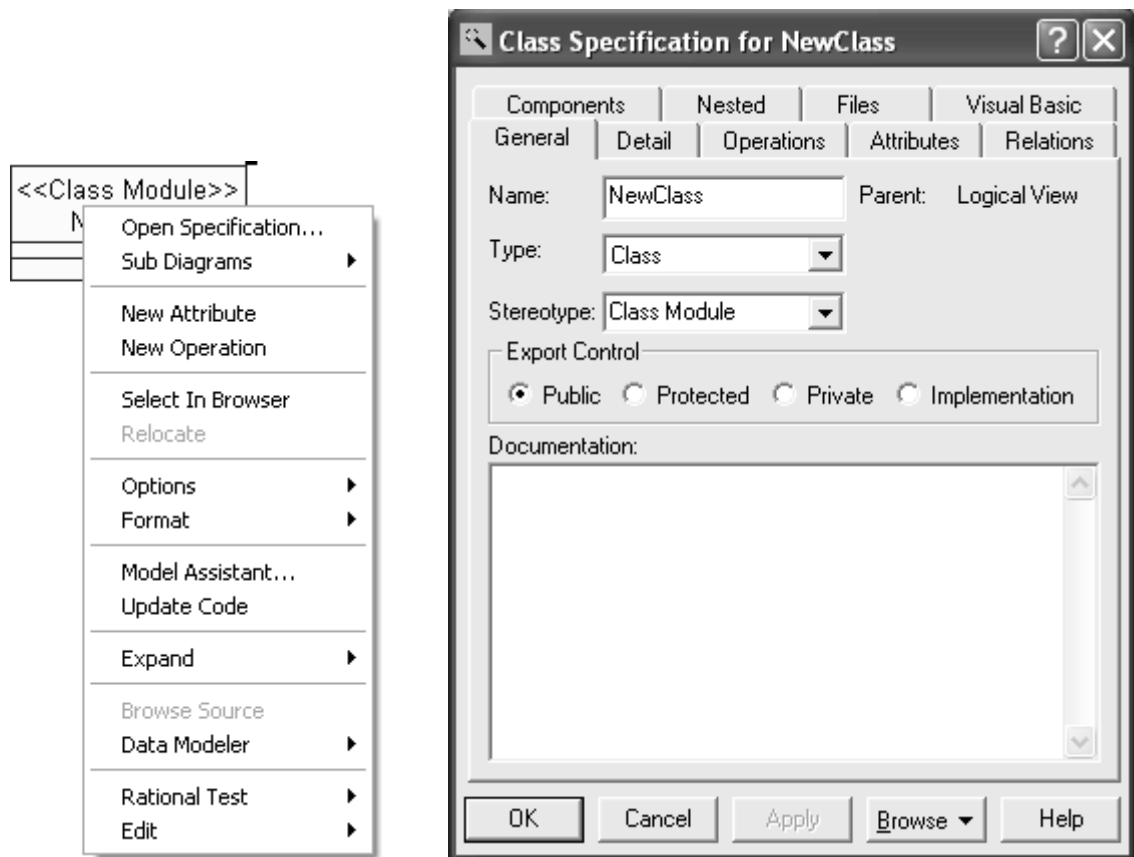


Рисунок 8.2: а) Контекстное меню класса; б) Окно спецификации класса

В этом окне задаются главные свойства класса: имя, тип, стереотип и доступ к нему, когда класс находится в пакете, а также документация к классу. Вкладка *Detail* позволяет указать дополнительные характеристики класса (рис. 8.3):

- *Multiplicity* – ожидаемое количество объектов, которые будут созданы на основе данного класса.
- *Space* – количество оперативной памяти, необходимой для создания объекта, учитывая накладные расходы на его создание плюс размер всех объектов, входящих в данный.

- *Persistence* – признак, указывающий время жизни объекта.
- Concurrency – поведение элемента в многопоточковой среде.
- *Abstract adornment* обозначает, что класс является абстрактным, т.е. базовым, который должен быть наследован подклассами.
- *Formal Arguments* заполняется только для параметризованных классов и утилит классов.

Вкладка *Components* отражает компоненты, с которыми ассоциирован класс (рис 8.4). На вкладке рядом с иконками помечаются компоненты, которые должны включаться в модель, и могут быть показаны остальные компоненты модели. Вкладка *Attributes* позволяет добавлять, удалять, редактировать атрибуты класса (рис. 8.5). На ней представлен список атрибутов класса, который можно редактировать при помощи контекстного меню.

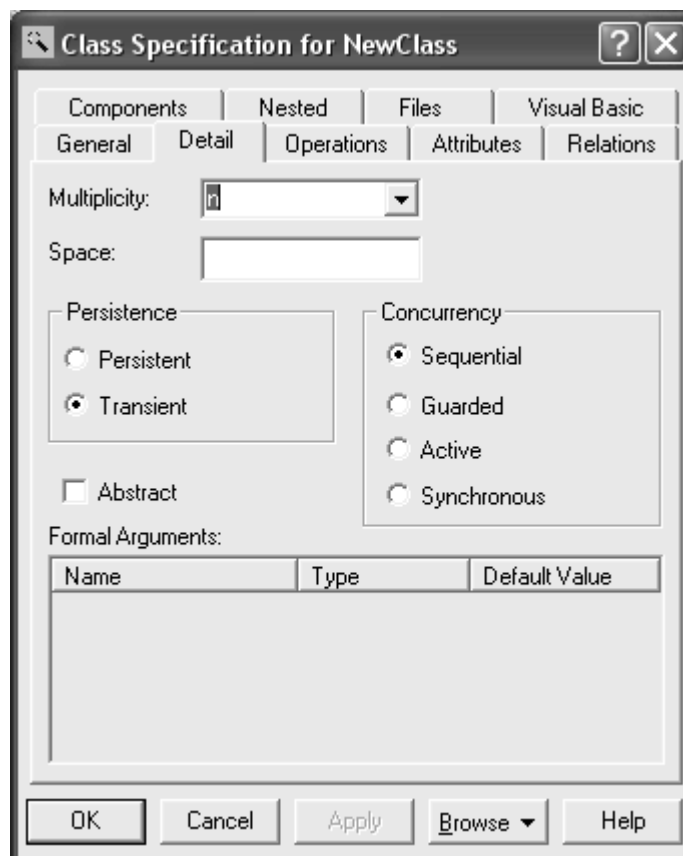


Рис. 8.3 Окно вкладки Detail

Флажок *Show inherited* позволяет скрыть или показать доступные атрибуты родительских классов. Для того чтобы добавить атрибут, необходимо из контекстного меню выбрать пункт *Insert*. В диалоговом окне спецификаций атрибутов можно изменить его название, тип и стереотип, задать начальное значение и тип доступа к атрибуту. Вкладка *Detail* спецификаций атрибутов класса позволяет задать тип хранения атрибута в классе:

- *By Value* – по значению.
- *By Reference* – по ссылке.

- *Unspecified* – не указано.

Кроме того, можно указать, что атрибут является Static (статическим) или Derived (производным).

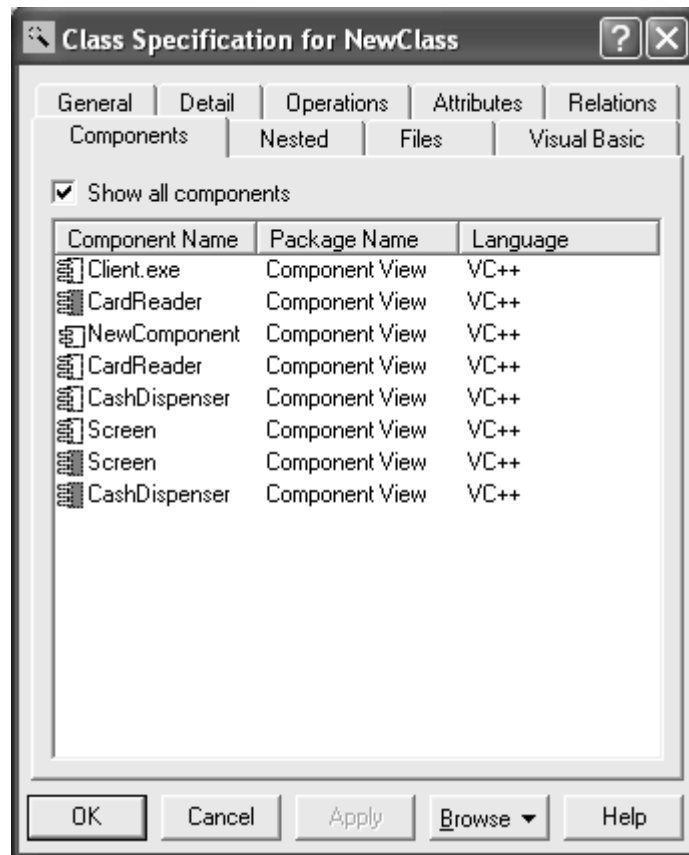


Рис. 8.4 Окно вкладки Components

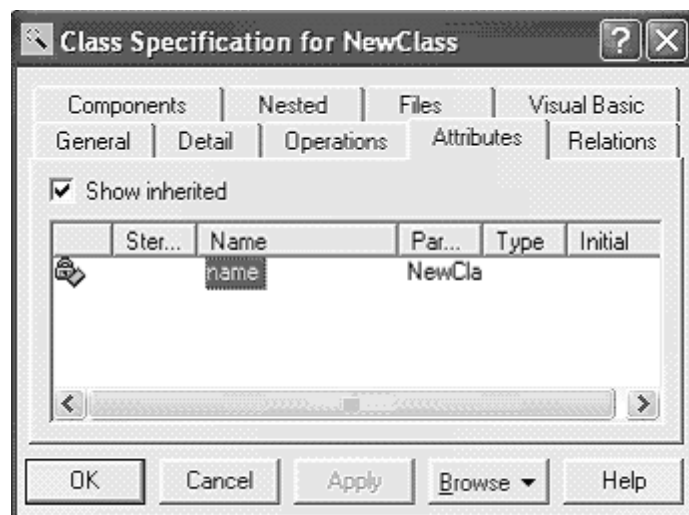


Рис. 8.5 Окно вкладки Attributes

Вкладка *Operations* предназначена для добавления, удаления, редактирования операции класса (рис. 8.6), на ней представлен список операций класса, которые можно редактировать при помощи контекстного

меню. Для того чтобы добавить операцию, необходимо из контекстного меню выбрать пункт *Insert*. С операциями связано окно их спецификаций.

Вкладка *Relations* позволяет добавлять, удалять, редактировать связи класса. На ней представляется список связей класса, которые можно редактировать при помощи контекстного меню.

Вкладка *Visual Basic*, появившаяся после ассоциации класса с языком Visual Basic, предназначена для изменения свойств, связанных с данным классом. Ее поля не предназначены для редактирования.

На вкладке *COM* устанавливаются свойства для классов, которые связаны с созданием *COM* объектов в модели. Если такие объекты импортируются в модель, в них также появляется подобная вкладка.

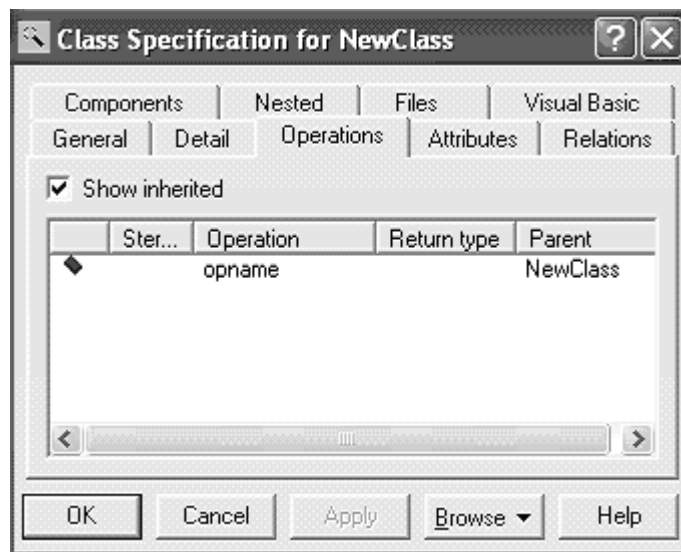


Рис. 8.6 Окно вкладки Operations

8.3 Связи на диаграмме Class

В большинстве случаев классы взаимодействуют друг с другом, что отображается при помощи различного вида связей, влияющих на получаемый при генерации код. В диаграмме классов различают следующие виды связей:

- *Unidirectional association* (однонаправленная ассоциация).
- *Dependency* (зависимость).
- *Association class* (ассоциированный класс).
- *Generalization* (наследование).
- *Realization* (реализация).

Unidirectional association – это один из важных и сложных типов связи. Она показывает, что один класс включается в другой как атрибут по ссылке или по значению. На рис 8.7 приведен пример связи *Unidirectional association*.

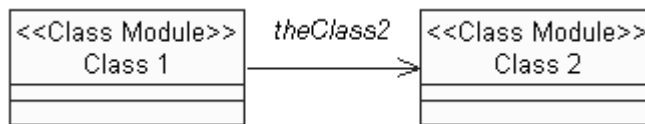


Рисунок 8.7 – Однонаправленная ассоциация

Создаваемый код класса зависит от установленных спецификаций связи. При активизации окна спецификаций открывается ее вкладка *General*, где содержится следующая информация о связи:

- *Name* – имя связи.
- *Parent* – имя пакета, которому принадлежит связь.
- *Stereotype* – стереотип.
- *Role A/Role B* – имя роли, с которой один класс ассоциируется с другим.
- *Element A/Element B* – имя класса, который ассоциирован с данной ролью.

На вкладке *Detail* указываются дополнительные свойства связи, такие как:

- *Name Direction* – имя связанного класса.
- *Constraints* – выражение семантического условия, которое должно быть выполнено, в то время как система находится в устойчивом состоянии.

Вкладка *Role General* отражает настройки переменной, которая будет включена в класс. Поскольку направление связи на рис. 8.8 от *Class1* к *Class2*, то *Role A* – это *Class2*, а *Role B* – это *Class1*. Рассмотрим вкладку *Role A General*. Она имеет следующие поля:

- *Role* – имя переменной для класса.
- *Element* – имя класса, для которого создается переменная.
- *Export Control* – доступ к элементу; имеется четыре переключателя: *Public*, *Protected*, *Private*, *Implementation*, которые указывают, в какой секции была создана переменная.

Вкладка *Role Detail* детализирует установки для связи и имеет поля:

- *Role* – имя переменной класса.
- *Element* – имя класса, для которого создается переменная.
- *Constraints* – выражение семантического условия, которое должно быть выполнено, в то время как система находится в устойчивом состоянии.
- *Multiplicity* – ожидаемое количество объектов данного класса, которые задаются числом, отображаемым рядом со стрелкой связи или буквой «n», указывающей, что количество не лимитировано.

- *Navigable* – направление, в котором действует связь. На какой класс будет направлена стрелка связи, тот и будет включаться в другой. Для того чтобы изменить направление связи, достаточно снять флажок с вкладки *Role A Detail* и установить его во вкладке *Role B Detail*. В случае, когда сняты флажки на обеих вкладках, ни один элемент не будет включен в другой, на диаграмме этому будет соответствовать просто линия.

- *Aggregate* – один класс содержит другой. Для того чтобы показать, что класс *Class2* входит в класс *Class1*, необходимо установить этот флажок во вкладке *Role B Detail*. При этом стрелка связи на диаграмме приобретает ромб с обратной стороны стрелки (рис. 8.8).

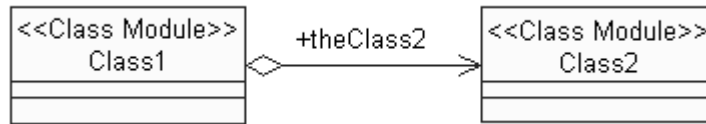


Рисунок 8.8 – Агрегирование класса

- *Static* – общность реквизита для всех объектов данного класса. При этом после инициализации к нему можно обращаться, даже если еще не было создано ни одного объекта класса. *Static* применяется, чтобы переменные такого типа не тиражировались при создании нового объекта класса.

- *Friend* – указанный класс является дружественным, то есть имеет доступ к защищенным методам и атрибутам.

- *Key/qualifier* – атрибут, который идентифицирует уникальным образом единичный объект, что не влияет на генерацию кода.

Тип связи *Dependency* позволяет показать, что один класс использует объекты другого. Это может осуществляться при передаче параметров или вызове операций класса. В таком случае генератор кода *Rational Rose* включает заголовочный файл в класс, который использует операторы или объекты другого класса. Графическое изображение этого вида связи показано на рис. 8.9.



Рисунок 8.9 – Связь Dependency

Тип связи *Association class* используется для отображения свойства ассоциации. Свойства сохраняются в классе и соединяются связью *Association* (рис. 8.10). Этот тип не имеет своих спецификаций. Ассоциация предназначена для задания дополнительных атрибутов у связи. Она обозначает, что некоторый класс со своими атрибутами включается как элемент в два других, хотя при генерации кода это не отображается.

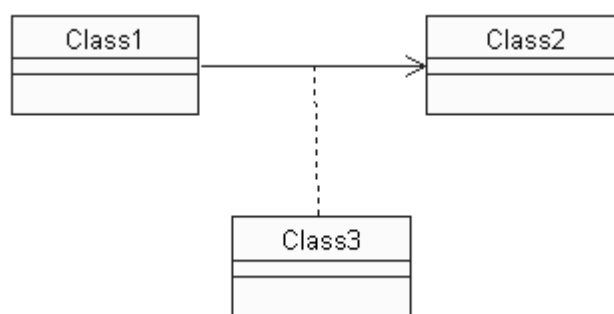


Рис. 8.10 – Связь Association class

Тип связи *Generalization* позволяет указать, что один класс является родительским по отношению к другому, при этом будет создан код наследования класса. Пример такой связи показан на рис. 8.10.

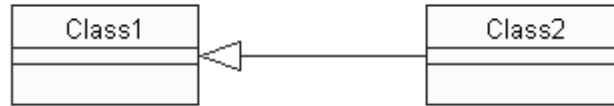


Рисунок 8.11 – Связь Generalization

Тип связи *Realization* позволяет показать, что один класс является реализацией, т.е. создан на основе шаблона другого. В Rational Rose для обозначения класса шаблона используется понятие параметризованный класс. Графическое изображение этого типа связи показано на рис. 8.12.



Рисунок 8.12 – Связь Realization

На практике чаще других используются два вида связей: *Unidirectional association* для агрегирования включения ссылок на классы и *Generalization* для создания иерархии наследования.

Ниже на рис. 8.13 приводится пример фрагмента диаграммы классов.

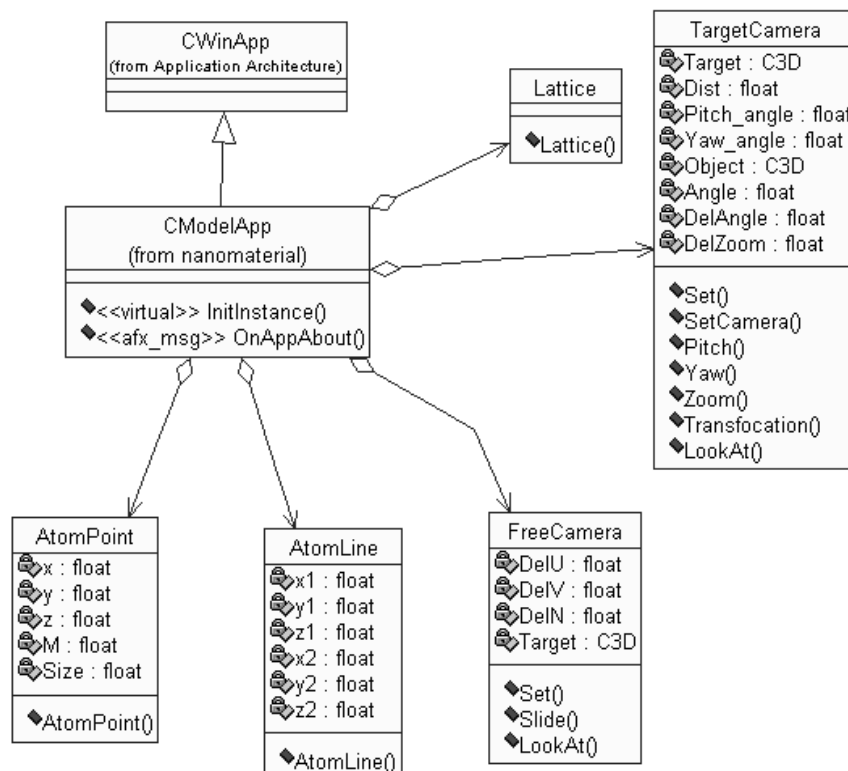


Рисунок 8.13 – Фрагмент диаграммы классов

На ней отражается внутренняя структура системы, описывается наследование и взаимное положение классов относительно друг друга путем определения между ними связей различных типов. В случае программирования на языке *Microsoft Visual C++* разработчик получает дополнительные возможности – доступ ко всей иерархии классов библиотеки MFC. Тогда с помощью мастера создания приложений автоматически строится шаблон приложения, в котором классы наследуются из библиотеки MFC. На рис. 8.13 показано, что главный класс приложения *CModelApp* наследуется из библиотечного класса *CWinApp*. Связи *CModelApp* с остальными классами показывают, что они содержатся в главном классе приложения.

На диаграмме вместе с атрибутами и операциями представлены следующие классы: *AtomPoint*; *AtomLine*; *Lattice*; *TargetCamera* и *FreeCamera*.