

УДК 004.43

Трофимов С.А.

CASE-технологии: практическая работа в Rational Rose. Изд. 2-е. – М.: Бином-Пресс, 2002 г. - 288 с.: ил.

Эта книга знакомит читателя с таким популярным CASE-средством как Rational Rose. В ней показывается, как при помощи пакета Rational Rose на основе UML-диаграмм создается программная система от замысла до создания исполняемого кода. Повторяя описанные в книге действия, читатель сам пройдет этот путь, целью которого будет создание полноценного приложения.

На каждом шаге в создании проекта подробно описываются возможности этого сложного инструмента для проектирования и создания программного кода системы. Книга открывает пути использования пакета Rational Rose не только для аналитика, но и для программиста, показывая процесс описания поведения объектов и иерархии классов для конкретного приложения.

На примере создания системы управления тепличным хозяйством демонстрируются возможности взаимодействия Microsoft Visual C++ и Rational Rose при создании и реинженеринге программного кода, представлены примеры генерации программного кода по готовым UML диаграммам на языках C++ и Visual Basic.

Подробно описываются возможности создания Web приложений и проектирование баз данных при помощи Rational Rose.

© Трофимов С.А., 2002,

© Издательство БИНОМ, 2002.

Оглавление

ОГЛАВЛЕНИЕ	3
ПРЕДИСЛОВИЕ.....	9
ВСТУПЛЕНИЕ	10
Цели написания этой книги	11
Для кого написана эта книга	12
Структура книги	13
Использование терминов	15
ЧАСТЬ 1. RATIONAL ROSE ОДНИМ ВЗГЛЯДОМ	18
ГЛАВА 1. ПЕРВОЕ ЗНАКОМСТВО	18
Что такое RATIONAL ROSE?	18
Язык UML	18
RATIONAL ROSE - лидер среди CASE-средств	18
Что может и чего не может сделать RATIONAL ROSE	19
Преимущества от применения RATIONAL ROSE	20
Главное окно RATIONAL ROSE	21
Какие диаграммы дает в наше распоряжение RATIONAL ROSE?	24
USE CASE DIAGRAM (диаграммы сценариев)	24
DEPLOYMENT DIAGRAM (диаграммы топологии)	25
STATE MACHINE DIAGRAM (диаграммы состояний)	25
INTERACTION DIAGRAM (диаграммы взаимодействия)	26
Вопросы для повторения	30
ГЛАВА 2. ЗНАКОМСТВО С МЕНЮ RATIONAL ROSE	31
Структура меню	31
NEW (новый)	32
OPEN (открыть)	33
SAVE (сохранить)	34
SAVE AS (сохранить как)	34
CLEAR LOG (очистить протокол)	35
LOAD MODEL WORKSPACE (загрузить рабочую область модели)	35
SAVE MODEL WORKSPACE (сохранить рабочую область модели)	35
SAVE MODEL WORKSPACE AS (сохранить рабочую область модели как)	35
UNITS (единицы)	35
IMPORT (импорт)	36
EXPORT MODEL (экспорт)	36
PRINT (печать)	37
PAGE SETUP (настройка страницы)	38
EDIT PATH MAP (редактирование карты путей)	38
EXIT (выход)	39

EDIT (РЕДАКТИРОВАНИЕ)	40
FORMAT (ФОРМАТ)	48
BROWSE (ПРОСМОТР)	52
REPORT (ОТЧЕТ)	54
QUERY (ЗАПРОС)	57
TOOLS (ИНСТРУМЕНТЫ)	61
ADD-INS (ПОДКЛЮЧАЕМЫЕ МОДУЛИ)	61
WINDOWS (ОКНА)	62
HELP (ПОМОЩЬ)	62
Вопросы для повторения	62
ЧАСТЬ 2 ПРОЕКТИРОВАНИЕ ГИДРОПОННОЙ СИСТЕМЫ	63
ГЛАВА 3. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ПАРАДИГМА	63
ОБЩАЯ КОНЦЕПЦИЯ	63
Стили программирования	63
ОТЛИЧИТЕЛЬНЫЕ ОСОБЕННОСТИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА	63
КОНЦЕПТУАЛЬНАЯ БАЗА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО СТИЛЯ	64
КЛАССЫ И ОБЪЕКТЫ	64
СВОЙСТВА КЛАССОВ	65
ИНКАПСУЛЯЦИЯ	65
НАСЛЕДОВАНИЕ	65
ПОЛИМОРФИЗМ	66
АТТРИБУТЫ И МЕТОДЫ КЛАССОВ	66
Вопросы для повторения	66
ГЛАВА 4. ОПРЕДЕЛЕНИЯ ТРЕБОВАНИЙ К СИСТЕМЕ ПРИ ПОМОЩИ USE CASE	67
СПЕЦИФИКА СОЗДАНИЯ ПРОГРАММНЫХ СИСТЕМ	67
ОПИСАНИЕ ЗАДАЧИ	67
НАЗНАЧЕНИЕ	68
ПОДГОТОВКА К РАБОТЕ С USE CASE	68
СОЗДАНИЕ НОВЫХ ЭЛЕМЕНТОВ	69
СТРОКА ИНСТРУМЕНТОВ	70
SELECTION TOOL (ИНСТРУМЕНТ ВЫБОРА)	70
TEXT BOX (ТЕКСТ)	70
NOTE (ЗАМЕЧАНИЕ)	71
NOTE ANCHOR (ЯКОРЬ ДЛЯ ЗАМЕЧАНИЯ)	71
PACKAGE (ПАКЕТ)	72
USE CASE (СЦЕНАРИИ ПОВЕДЕНИЯ)	72
АКТОР (АКТЕР)	72
UNIDIRECTIONAL ASSOCIATION (ОДНОНАПРАВЛЕННАЯ СВЯЗЬ)	73
СОЗДАНИЕ ДИАГРАММЫ USE CASE ДЛЯ ГИДРОПОННОЙ СИСТЕМЫ	73
ИСПОЛЬЗОВАНИЕ ДИАГРАММЫ USE CASE ДЛЯ МОДЕЛИРОВАНИЯ ПРОИЗВОДСТВА	75
BUSINESS ACTOR (БИЗНЕС-АКТОР)	76
BUSINESS WORKER (БИЗНЕС-РАБОТНИК)	76
BUSINESS ENTITY (БИЗНЕС-СУЩНОСТЬ)	76

ORGANIZATION UNIT (Организационное подразделение)	76
BUSINESS USE CASE (Бизнес-процесс).....	76
Вопросы для повторения.....	76
ГЛАВА 5. ИСПОЛЬЗОВАНИЕ DEPLOYMENT ДИАГРАММЫ ДЛЯ АНАЛИЗА УСТРОЙСТВ.....	77
Назначение	77
Устройства тепличного хозяйства	81
Вопросы для повторения.....	81
ГЛАВА 6. СОЗДАНИЕ МОДЕЛИ ПОВЕДЕНИЯ СИСТЕМЫ ПРИ ПОМОЩИ ДИАГРАММЫ STATECHART	83
Назначение диаграммы.....	83
Создание заготовок классов.....	83
Создание STATECHART диаграммы.....	84
Инструменты диаграммы STATECHART	85
Первые шаги в создании диаграммы	86
Состояние тестирования датчиков	87
Вывод диаграммы из цикла	89
Добавление замечания.....	90
Настройка среды	90
Скрытие вложенных состояний	92
STATES HISTORY (история состояний)	93
Вопросы для повторения.....	94
ГЛАВА 7. СОЗДАНИЕ МОДЕЛИ ПОВЕДЕНИЯ ПРИ ПОМОЩИ ACTIVITY DIAGRAM	95
Назначение диаграммы активности.....	95
Отличия между ACTIVITY и STATECHART	95
Создание диаграммы активности	95
Строка инструментов.....	96
Начало создания диаграммы.....	97
Редактирование спецификаций SWIMLANES	98
Настройка спецификаций значка активности.....	100
Создание значка анализа времени	103
Добавление значка получения времени	103
Добавление решения	104
Синхронизация процессов	105
Добавление опроса датчиков	106
Создание вложенной диаграммы.....	109
Вопросы для повторения.....	111
ГЛАВА 8. ОПИСАНИЕ ВЗАИМОДЕЙСТВИЯ ПРИ ПОМОЩИ.....	112
SEQUENCE DIAGRAM.....	112
Назначение диаграммы.....	112
Создание диаграммы	112
Строка инструментов диаграммы	113
Настройка времени жизни объекта.....	114

Создание класса из окна настройки параметров	115
Создание сообщений	116
Свойства сообщений	118
Добавление класса PLANTDayHour	119
Окончательный вариант диаграммы	120
Вопросы для повторения	121
ГЛАВА 9. ОПИСАНИЕ ВЗАИМОДЕЙСТВИЯ С ПОМОЩЬЮ COLLABORATION DIAGRAM	122
Назначение диаграммы	122
Создание диаграммы	122
Строка инструментов	124
Создание объекта	125
Контекстное меню объекта	125
Создание связи между объектами	126
Автоматический перенос данных в диаграмму SEQUENCE	127
Вопросы для повторения	131
ГЛАВА 10. ДИАГРАММА КОМПОНЕНТОВ	132
Назначение диаграммы	132
Замечания по созданию диаграммы компонентов	132
Создание диаграммы компонентов	133
Строка инструментов	133
Вопросы для повторения	140
ЧАСТЬ 3 РАБОТА С ДИАГРАММОЙ КЛАССОВ	141
ГЛАВА 11. ВОЗМОЖНОСТИ ДИАГРАММЫ КЛАССОВ	141
Назначение диаграммы	141
Создание диаграммы	142
Строка инструментов	142
Контекстное меню класса	146
Спецификации класса	149
Вкладка SOM	158
Вопросы для повторения	159
ГЛАВА 12. СВЯЗИ	160
Назначение и виды связей	160
Вопросы для повторения	168
ГЛАВА 13. СОЗДАНИЕ КОДА НА C++	169
Вступительные замечания	169
Эталонный код класса	169
Ассоциация класса с языком C++	170
Просмотр кода класса	170
Установка типа объекта	171
Добавление новых операций	172
Установка зависимости классов	173
Настройка свойств C++	174

Вопросы для повторения.....	177
ГЛАВА 14. СОЗДАНИЕ КОДА КЛАССА НА MICROSOFT VISUAL C++	178
ВСТУПИТЕЛЬНЫЕ ЗАМЕЧАНИЯ	178
СТРУКТУРА СОЗДАВАЕМОГО КОДА КЛАССА	178
АССОЦИАЦИЯ КЛАССА С ЯЗЫКОМ VC++	179
МЕНЮ ИНСТРУМЕНТОВ VISUAL C++	181
SOM.....	190
QUICK IMPORT MFC 6.0.....	190
PROPERTIES	190
СОЗДАНИЕ КОДА КЛАССА.....	191
Вопросы для повторения.....	191
ГЛАВА 15. СОЗДАНИЕ КОДА НА VISUAL BASIC	192
ВСТУПИТЕЛЬНЫЕ ЗАМЕЧАНИЯ	192
КЛАССЫ В VISUAL BASIC.....	192
ИЕРАРХИЯ КЛАССОВ.....	193
АССОЦИАЦИЯ КЛАССА С ЯЗЫКОМ VISUAL BASIC.....	193
МЕНЮ ИНСТРУМЕНТОВ VISUAL BASIC	195
PROPERTIES (СВОЙСТВА)	195
MODEL ASSISTANT	197
СОЗДАНИЕ КОДА КЛАССА.....	199
OPTION EXPLICIT	201
Вопросы для повторения.....	202
ЧАСТЬ 4. СОЗДАНИЕ РАБОТАЮЩЕГО ПРИЛОЖЕНИЯ НА VC++	203
ГЛАВА 16. СОЗДАНИЕ ШАБЛОНА ПРИЛОЖЕНИЯ.....	203
СТРАТЕГИЧЕСКИЕ И ТАКТИЧЕСКИЕ РЕШЕНИЯ.....	203
СОГЛАШЕНИЕ ОБ ИСПОЛЬЗОВАНИИ ИМЕН	204
СТРУКТУРА ПРИЛОЖЕНИЯ 	205
ПОНЯТИЕ «ДОКУМЕНТ» ДЛЯ ТЕПЛИЧНОГО ХОЗЯЙСТВА	205
ОПЕРАЦИЯ ONNEWDOCUMENT	207
СОЗДАНИЕ ШАБЛОНА ПРИЛОЖЕНИЯ	208
НАЗНАЧЕНИЕ КЛАССОВ В ПРОЕКТ	211
ИМПОРТ БИБЛИОТЕКИ MFC	211
ЗАГРУЗКА СОЗДАННЫХ КЛАССОВ В МОДЕЛЬ.....	212
ПЕРВЫЙ ЗАПУСК ПРИЛОЖЕНИЯ.....	212
Вопросы для повторения.....	215
ГЛАВА 17. ДОБАВЛЕНИЕ ФУНКЦИОНАЛЬНОСТИ В КЛАСС ПРОСМОТРА	216
СОЗДАНИЕ ПУНКТА МЕНЮ WORK (РАБОТА)	216
АССОЦИАЦИЯ ОПЕРАЦИИ С ПУНКТОМ МЕНЮ	216
ДОБАВЛЕНИЕ ОПЕРАЦИЙ В MODEL ASSISTANT	218
ОТСЛЕЖИВАНИЕ СООБЩЕНИЙ ТАЙМЕРА	219
Вопросы для повторения.....	222

ГЛАВА 18. ДОБАВЛЕНИЕ ФУНКЦИОНАЛЬНОСТИ В КЛАСС ДОКУМЕНТА.....	223
Создание структуры CONDITION	223
Создание модели среды	223
Добавление функциональности.....	224
Вопросы для повторения.....	231
ГЛАВА 19. СОЗДАНИЕ ИСПОЛНИТЕЛЬНЫХ УСТРОЙСТВ.....	232
Вступительные замечания	232
Список исполнительных устройств	232
Создание родительского класса устройств.....	233
Установка наследования	236
Добавление значения для возврата	238
Использование MODEL ASSISTANT для изменения класса	238
Добавление начальных состояний устройств	240
Создание операций ON/OFF для дочерних классов	241
Создание адреса расположения устройств	243
Агрегирование устройств	246
Корректирование отклонений показателей	250
Вопросы для повторения.....	251
ЧАСТЬ 5. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ	252
ГЛАВА 20. РАЗРАБОТКА WEB ПРИЛОЖЕНИЙ	252
Вступление	252
Назначение WEB MODELER	252
Меню	253
Связи.....	257
Шаги создания Web приложения	257
Спецификации CLIENT PAGE	258
Вопросы для повторения.....	262
ГЛАВА 21. МОДЕЛИРОВАНИЕ ДАННЫХ.....	263
Вступление	263
Меню DATA MODELER.....	263
Порядок создания новой структуры данных	265
Создание структуры данных	265
Создание таблиц.....	266
Создание связей	269
Перенос структуры в базу	270
Диаграмма данных	271
Создание простой структуры данных.....	272
Вопросы для повторения.....	273
ЗАКЛЮЧЕНИЕ	275
ЛИТЕРАТУРА.....	276
ПРИЛОЖЕНИЯ.....	277

Предисловие ко второму изданию

С момента выхода первого издания книги прошло больше года. За это время многое изменилось. Вышла новая версия программы, в которой значительно расширены возможности интегрирования с Web и работы с базами данных. Добавились новые встраиваемые модули (Add-ins). Развиваются технологии создания программ, но процесс моделирования систем при помощи UML диаграмм не теряет свою актуальность, наоборот, все большее количество больших и малых компаний приходят к тому, что для создания программных систем нужно использовать современные технологии.

Компания Rational Software продолжает выпуск линейки программных продуктов, направленных на поддержку технологического процесса разработки программных систем от замысла до завершения жизненного цикла программы. Можно вспомнить такие программы как Rational Clear Case - для управления конфигурациями, Rational Clear Quest - для управления запросами на изменение программ, Rational RequisitePro - для управления требованиями к программному обеспечению и другие, однако центральной частью и связующим звеном во всей работе является Rational Rose, как основной инструмент создания моделей.

Чем больше времени я уделяю работе с Rational Rose, тем больше я влюбляюсь в этот инструмент, который позволяет экономить время и помогает качественно делать свою работу.

Во второе издание книги я включил описание новых возможностей версии 2002, а также некоторые моменты, которые не вошли в первое, таким образом, книга стала больше на несколько глав. Были расширены и стали более легкими для понимания основные разделы. Упорядочено употребление некоторых терминов. Так в первом издании термин «package» переводился как «контейнер», что по моему мнению достаточно точный перевод, однако сейчас я употребляю слово «пакет», поскольку такой перевод стал стандартом де-факто.

Вступление

В карьере любого серьезного программиста рано или поздно наступает момент, когда нужно сделать качественный скачок в своем образовании. Обычно это происходит при смене версии языковых средств или инструментов, с которыми за многие годы работы просто сроднился, зная все «подводные камни» и способы их преодоления. Однако наступает момент, когда нужно сказать себе «пора».

У меня, как и у многих, таких моментов было несколько. Сначала был вопрос выбора и последующего изучения выбранного языка программирования. Начинал, как и все, с Basic. После создания нескольких программ на нем, обратил внимание на Pascal. Но этот период прошел довольно быстро, и я, получив в свое распоряжение персоналку, погрузился в изучение ее возможностей при помощи Assembler.

Первой книгой по C++, которая попала мне в руки в далеком 1991 году, была, ныне многократно переизданная книга основателя C++ Б. Страуструпа, «Язык программирования C++» [2]. И я, покоренный красотой языка и его возможностями, просто погрузился в этот совершенно новый мир - мир, состоящий из объектов, которые используются для создания программ.

Сменялись версии языка, добавлялись новые объекты, появилась библиотека MFC и различные мастера (Wizard), помогающие создать приложение за несколько минут. Но созданные этими мастерами приложения необходимо наполнять реальными классами, устанавливать их взаимосвязи, а это требует «ручного» программирования и, как следствие, трудоемкой поддержки и сопровождения. Даже если вы используете собственную библиотеку классов, внесение изменений в код самого класса чревато нарушением его работоспособности, особенно после нескольких месяцев или лет работы, когда многие ограничения и связи, которые учитывались при создании, забыты. Так где же решение? А решение, как это часто бывает, лежит на поверхности. Нужно уменьшить «ручное» программирование, чреватое ошибками и вносимыми каждым разработчиком личными предпочтениями в программу.

Как это сделать, спросите вы. Ведь создание программ - это творческий процесс, полностью зависящий от программиста. Это достаточно распространенное заблуждение. Конечно, как и в любой работе, здесь есть элементы творчества, однако, это процесс, который можно и нужно ставить на индустриальные рельсы. Разработка программ напоминает создание, например, самолета, где влияние на проект конкретного инженера минимально. Необходим инструмент, который позволит уменьшить долю ручного труда при программировании и предоставит большие возможности для творчества, освобождая программиста от выполнения рутинных операций и от ошибок при принятии решений.

И такой инструмент есть - это Rational Rose Enterprise Edition. Эту

программу можно использовать отдельно, а можно вместе с другими продуктами Rational Software, которые позволяют создавать сложные программные системы быстрее, качественнее и легче. Недаром Rational Rose включается во все конфигурации Rational Suite - инструмента для аналитиков, программистов и тестировщиков. Переход от программирования к созданию моделей, а затем генерации кода по моделям - вот следующий качественный скачок, который я предлагаю вам сделать.

Rational Rose позволяет создавать модели будущей системы, удобные для понимания алгоритмов работы, взаимосвязей между объектами, по которым в дальнейшем создается программный каркас будущей программной системы.

Сейчас уже не для кого не секрет, что моделирование - одно из средств, позволяющих значительно сократить время разработки, уложиться в бюджет и создать систему с нужным качеством. Модель будущей системы помогает уже на стадии проектирования, без вкладывания больших средств в «пилотный» проект получить представление о поведении-системы и избежать дорогостоящих ошибок в дальнейшем, когда в написание программного кода вложены значительные силы. И можно с уверенностью сказать, что как инструмент моделирования, Rational Rose не знает себе равных и история создания продукта это показывает.

Можно создавать UML модели при помощи других программных продуктов, например редактора Visio, однако при помощи Rational Rose это сделать проще и удобнее. Созданные в едином стиле UML диаграммы понятны любому программисту, который придет к вам на смену, когда вы, изучив концепцию объектно-ориентированного проектирования, уйдете в аналитики или руководители проекта. И что самое ценное, эти диаграммы позволяют прямо из проекта Rational Rose создать исходный текст программы на C++, Visual Basic, Java или другом языке, поддерживаемом Rational Rose. Дополнительные модули (add-ins) позволяют создавать исходный текст на языках программирования, поддержка которых не включена в базовую поставку. Например, для работы с популярным в России Borland C++ Builder создан дополнительный модуль. Так что программисты-поклонники средств разработки от компании Borland не будут чувствовать себя обделенными.

При помощи входящего в пакет анализатора или подключаемого модуля легко преобразовать имеющийся у вас программный проект C++ в проект Rational Rose, чтобы в дальнейшем получить все преимущества от использования UML моделей.

Итак, скажите себе «пора», устанавливайте на своем домашнем или рабочем компьютере этот пакет и приступайте к изучению. Время, которое вы потратите на освоение этого замечательного инструмента программиста и проектировщика - ничто, по сравнению с теми преимуществами, которые вы получите.

Цели написания этой книги

Эта книга написана для того, чтобы помочь изучить Rational Rose -

достаточно сложный программный продукт. В ней на примере станции управления тепличным хозяйством показано, как происходит объектно-ориентированная разработка при помощи CASE-инструмента Rational Rose.

Эта книга не дает ответов на то, как нужно проектировать объектно-ориентированные системы, этот процесс подробно рассмотрен, например, в [1, 7, 8] и массе других изданий, здесь лишь перечисляются общие подходы.

Эта книга не является учебником по программированию на C++, как, например [2, 4, 6], однако, даже не зная ничего о синтаксисе языка, можно использовать эту книгу для объектно-ориентированного проектирования, ведь Rational Rose поддерживает также CORBA, Java, Visual Basic, Oracle 8, и исходный текст на любом из этих языков может быть создан на основе готового проекта Rational Rose.

Также эта книга не является полноценным учебником по языку UML, хотя даже не зная его, можно легко ориентироваться в диаграммах, представленных в книге, и уже после прочтения, обратившись к другим изданиям, например [9, 10], вы заметите, что назначение и основные пути применения UML диаграмм вам уже понятны и можно спокойно углубляться дальше, непосредственно в процесс проектирования.

Для создания приложения мы будем использовать Visual C++ с применением библиотеки Microsoft MFC, потому что C++ использует для примеров Г. Буч в книге [1], и этот язык создан для разработки объектно-ориентированных систем, а библиотека MFC позволит быстро создать шаблон приложения. Подробнее о применении библиотеки MFC см. в [3, 5].

Также, для сравнения возможностей создания кода, мы рассмотрим создание кода на языке Visual Basic и C++ без применения библиотеки Microsoft MFC.

Эта книга даст ответы на вопросы как сделать то или иное действие при помощи данного инструмента и полностью отражает весь путь объектно-ориентированного создания программной системы от замысла до генерации исходного кода. Весь этот процесс показан на примере гидропонной системы, некоторые части которой даются в качестве примера в [1]. Поэтому для более глубокого изучения процесса объектно-ориентированного проектирования вы можете использовать работу Г. Буча, и данная книга будет в этом хорошим подспорьем.

Для кого написана эта книга

Эта книга написана для всех, кто хочет получить представление о современных методах создания объектно-ориентированных систем, для тех, кто хочет применить эти методы на практике, решая конкретные задачи.

Книга будет полезна как студенту, имеющему базовые понятия о программировании, так и серьезному программисту с многолетним опытом, желающему освоить передовые технологии программной инженерии.

Для примеров исходного кода, получаемого при помощи Rational

Rose, в основном используется Microsoft VC++, поэтому читатель, работающий с этим языком, может на практике убедиться в предоставляемых пакетом возможностях по генерации и реинженерингу исходного кода.

Пользователи, не работавшие ранее с Rational Rose, узнают о его возможностях и на практике, повторяя описанные действия, могут пройти через все этапы разработки простой системы, для того чтобы дальше попробовать себя в чем-то более сложном.

Все примеры в книге даны для версии программы 2002 Enterprise Edition, возможности которой подробно обсуждаются. Если в вашем распоряжении находится версия 2000 или 2001, то многие возможности (хотя и не все), описанные в книге, доступны и в этих версиях, однако могут возникать различия в меню и дополнительных поставляемых компонентах. Характерно то, что компания Rational Software, начиная с версии 2001, перестала акцентировать внимание пользователя на ее номере, который раньше можно было видеть на всех заставках. Это показывает, что этот программный продукт вошел в период стабильности и каких-то резких перемен в нем не намечается, а основное расширение функциональности будет производиться при помощи дополнительных модулей (Add-Ins).

Однако эффективная работа и создание модели учебной программной системы возможно во всех упомянутых версиях, так как основные функции, которые необходимы для этого, совпадают. И если учесть, что каждая последующая версия, требует больших вычислительных ресурсов, то работа не с самым последним релизом продукта может быть более эффективной по времени, несмотря на некоторые ограничения в возможностях.

Структура книги

Книга состоит из пяти частей, каждая из которых посвящена определенной теме. И все они постепенно подводят читателя к созданию законченного приложения при помощи программного пакета Rational Rose Enterprise Edition. После каждой главы даются контрольные вопросы, которые позволяют оценить глубину усвоения материала.

Часть I. Rational Rose одним взглядом

Здесь описывается назначение и основные возможности пакета. Читатель знакомится с теми преимуществами, которые получает программист от использования данного инструмента. Здесь же будет описана структура меню и рабочего стола программы и общий порядок работы на Rational Rose.

Часть II. Проектирование гидропонной системы

Здесь мы по шагам создадим рабочий проект гидропонной системы и при помощи диаграмм Rational Rose досконально разберем алгоритмы работы и взаимодействия объектов с одновременным изучением

возможностей используемых диаграмм. Будут рассмотрены все доступные диаграммы и работа с ними в применении к созданию гидропонной системы.

Часть III. Диаграмма классов

Для создания работающего приложения необходимо, кроме создания проекта, создать основу программ – классы. В этой части книги подробно рассматриваются возможности диаграммы классов для создания иерархии классов приложения и генерации программного кода. Рассматриваются возможности генерации кода на языках C++, Visual C++ и Visual Basic.

Часть IV. Создание работающего приложения

Эта часть посвящена созданию кода работающего приложения. Мало создать проект приложения, нужно воплотить этот проект в работающие алгоритмы, а это не всегда так просто, как кажется. После прочтения этой части и повторения описанных в ней действий у вас должно получиться готовое приложение Visual C++ на основе описанных в предыдущих главах классов и алгоритмов работы.

Часть V. Дополнительные возможности

В этой части рассматриваются дополнительные возможности пакета, которые мало использовались или не использовались вообще для создания гидропонной системы, однако они важны для более глубокого изучения программы. Это работа с подключаемыми модулями Web Modeler и Data Modeler.

Условные обозначения

Для того чтобы не писать несколько раз одно и то же, например: перейдите в окно «Window», нажмите правую кнопку мыши и из появившегося меню выберите пункт меню «Item», будем придерживаться следующих обозначений:

Menu: - меню

Wind: - окно

RClick: - нажатие правой кнопки мыши. Будем считать, что во всех случаях используется левая кнопка мыши и только когда необходимо уточнить, что нужна правая, будем применять это обозначение. Меню, вызываемое по правой кнопке мыши, будем называть контекстным.

=>выбрать

= присвоить значение

+ раскрыть содержимое, будем использовать для перехода к объектам нижнего уровня, если слева от объекта есть знак +.

Если нужно выбрать пункт из панели инструментов, то добавим название панели: Standard - главная, ToolBox - рабочая

Таким образом получаем такую запись:

Wind:Wmdow=>RClick=>Item

Или, если нужно написать: войдите в верхнее меню, выберите пункт Tools, и в нем пункт Options, в появившемся окне выбрать закладку Diagram, на которой установить Show visibility в положение ON, напомним кратко: Menu:Tools=>Options=>Diagram=>Show visibility=ON.

В течение всего изложения будут даваться советы, выделенные так:

Совет

Эти советы могут облегчить действия пользователя или показывают как использовать дополнительные, скрытые возможности программы, которые сделают выполнение задач более простым и комфортным.

Также в тексте будут присутствовать замечания и напоминания.

Эти замечания содержат дополнительную информацию, комментарии, ссылки на другие главы и разделы книги или напоминание о том, что вам необходимо сделать.

Так как книга описывает создание реальной программы на Visual C++, то в тексте будут присутствовать листинги программ, которые мы выделим так:

```
/* текст программы */  
#include "stdafx.h" // листинг программ
```

В листингах, согласно синтаксису языка C++, будут присутствовать комментарии, которые отделяются двойной обратной чертой или косой чертой со звездочкой.

Использование терминов

Rational Rose - англоязычная программа и пока не локализована для России, по крайней мере, мне такая локализация не попадалась. Но для русских программистов, для которых английский язык стал уже почти родным, это, как мне кажется, не очень большой недостаток.

Однако выбор, терминов, которые наиболее точно соответствуют английскому слову, может быть довольно сложной задачей. Вспомните, как звучали в первых локализациях языка Basic команды Goto как Перейти_на..., Gosub как Перейти_в_подпрограмму. Для англоязычной среды, которой я считаю компьютер, часто такие переводы звучат не просто необычно, а приводят к непониманию между пользователями да и между программистами тоже.

С плохо переведенными на русский язык терминами работать хуже, чем напрямую с английским словом. Поэтому при переводе англоязычных терминов, чтобы уменьшить разночтения для терминов, которые могут быть переведены неоднозначно и еще не получили широкого распространения в среде российских программистов и пользователей программ, буду оставлять английское написание, указывая наиболее приемлемый, с моей точки зрения, перевод в скобках. Русский перевод обязательно буду указывать при первом упоминании такого термина, а в дальнейшем, по возможности, буду пользоваться английским термином.

Конечно, за время существования программ и компьютеров многие слова просто переключались из английского языка в русский. В этом случае я постараюсь при указании перевода подобрать

общеупотребительное слово русского языка, которое наиболее соответствует английскому термину.

Что нужно для работы с книгой

Самое первое - это, конечно, сам пакет Rational Rose. Бесполезно читать книгу без возможности сразу применить на практике полученные знания, хотя для оценки возможностей программы иллюстраций, приведенных в книге, вполне достаточно.

Для ознакомления с программой в сети Internet по адресу www.rational.com вы можете получить версию с ограничением срока работы размером около 249 Мб. Неплохая подборка статей по Rational Rose есть на русскоязычном сервере www.interface.ru, где можно записаться на семинары и курсы по продуктам Rational Software.

Как указывается в сопроводительной документации, для установки программы необходима минимальная конфигурация программно-аппаратных средств, представленная в таблице 1.

Таблица 1. Требования к аппаратному обеспечению

Параметр	Требования	
	Минимальные	Рекомендуется
ОС	Windows 98 SE, ME, NT4 SP6, 2000, XP	
Аппаратное обеспечение	Pentium 300	Pentium 600 MHz
Оперативная память	128Мб	256 Мб (в зависимости от размера модели больше)
Дисковое пространство	500 Мб (230 для запуска Rose, 270 для установки)	
Web Browsers	Internet Explorer 4.01 или выше	
БД Rose Data Modeler	DB2 Universal Database 5.x, 6.x & 7.x, DB2 OS390 5.x & 6.x, SQL Server 6.x & 7.x, Oracle 7.x & 8.x, Sybase System 12, SQL Server 2000	
Монитор	SVGA (256 цветов или больше) разрешение 1024 x 768 pixels	
Архиватор	WinZip или эквивалент	

Эти требования, определенные в документации, не означают, что программа вовсе не будет работать на более слабых машинах. Работа просто будет менее комфортной.

Для того чтобы работать с примерами на C++, необходим любой компилятор C++.

Для того чтобы работать с примерами на Visual C++, необходим компилятор Microsoft Visual C++6.0. Причем последний вариант предпочтительней, так как Rational Rose позволяет проектировать программные системы с использованием библиотеки классов Microsoft (MFC 6.0), диаграммы которых включаются в поставку Rational Rose.

Для того чтобы работать с примерами на Visual Basic, необходима установка среды разработки Visual Basic.

Чтобы не думать каждый раз о том, какой язык установлен, а какой нет, то, если позволяет возможность, рекомендуется установить Microsoft Visual Studio целиком.

Все исходные тексты C++, используемые в книге, можно получить в сети Internet по адресу http://www.binompress.ru/www/books/rational_rose2.htm на сайте издательства «Бином», или <http://progcpp.narod.ru/rational.htm> на личной страничке автора, где также можно оставить свои пожелания и замечания.

Вот коротко и все, что необходимо знать до того, как приступить к изучению программы.

Часть 1. Rational Rose одним взглядом

Глава 1. Первое знакомство

Что такое Rational Rose?

На вопрос «а что же такое Rational Rose?» я бы кратко ответил: программный пакет для визуального объектно-ориентированного моделирования систем на основе классов и их взаимодействия, а если еще более упрощенно, это визуальный редактор, позволяющий моделировать программные системы любой сложности на основе графических диаграмм языка UML (Unified Modeling Language).

Язык UML

Язык UML кардинально отличается от таких языков программирования как, например, Visual C++ или Visual Basic. Он предназначен для описания моделей, причем для работы с этим языком используется специальные редакторы диаграмм, такие как Rational Rose. Этот язык слишком сложен, чтобы оперировать им вручную.

К счастью, пользователь пакета Rational Rose полностью огражден от ручного ведения кода, и Rational Rose сам создает и сохраняет по визуальным диаграммам все, что необходимо.

UML не зависит от объектно-ориентированных языков программирования и может поддерживать любой из них. Этот язык также не зависит от используемой методологии разработки проекта, и созданные на UML диаграммы выразительны и понятны для всех разработчиков, вовлеченных в проект, причем, что немаловажно, не только в момент разработки, но и много месяцев спустя.

UML является открытым и обладает средствами расширения базового ядра. На UML можно содержательно описывать классы, объекты и компоненты в различных предметных областях, часто сильно отличающихся друг от друга. Однако пакет Rational Rose поддерживает не только UML, но и другие нотации создания диаграмм, такие как OMT или Booch.

Замечание

Подробнее о нотациях, поддерживаемых Rational Rose, мы поговорим в главе 2.

Rational Rose - лидер среди CASE-средств

Мощный толчок CASE-средства получили в пору внедрения объектно-ориентированной технологии разработки программного обеспечения. Старые технологии разработки программ «сверху вниз» уже не могли справиться с все усложняющимися, труднообозримыми программными комплексами.

Сегодня Rational Rose лидирует среди других CASE-средств, и не случайно. То, что этот пакет позволяет создавать сложные программные системы от замысла до создания исходного кода, привлекает не только проектировщиков систем, но и программистов-разработчиков. За рубежом по причине сильной конкуренции между фирмами-разработчиками программ ни один, даже небольшой программный проект, не обходится без применения CASE-средств. Уже более 50 тысяч больших и маленьких компаний по всему миру используют Rational Rose для разработки программных систем. Это такие известные компании как NASA, Boeing, Lockheed Martin, Honey-well, NBC, Reuters, AT&T и другие.

Что может и чего не может сделать Rational Rose

Инструменты компании Rational Software как нельзя лучше подходят для объектно-ориентированной разработки программ от замысла до реализации в коде.

Rational Rose в соединении с другими программными пакетами приобретает свою неповторимую мощь. В сочетании со средствами документирования (Rational SoDA) он может давать полное представление о проекте.

Полностью интегрируясь с Microsoft Visual Studio, этот пакет дает возможность получать исходный код взаимодействующих классов и строить визуальные модели по уже написанному исходному коду.

Возможность интеграции со средствами управления требованиями (Requisite Pro), со средствами тестирования (SQA Suite, Performance Studio), со средствами конфигурационного управления (ClearCase, PVCS) поднимает процесс ведения программного проекта на совершенно новый уровень.

Открытая архитектура Rational Rose позволяет включать в него поддержку языков программирования, которые не предусмотрены стандартной поставкой, например, языка Assembler, для чего достаточно написать лишь собственный модуль.

Главное отличие Rational Rose от других CASE-средств в том, что он полезен не только проектировщику систем, но и разработчику программного кода.

Для проектировщика преимущества использования CASE-средств не вызывают сомнений, но для разработчика кода использование Rational Rose дает такие же преимущества, как если бы вы сменили велосипед на самолет.

На первый взгляд управление кажется сложным, но постепенно к нему привыкаешь. Имея такой инструмент, достижение целей становится проще и быстрее, а вся работа может быть окинута одним взглядом. Если в проект приходит новый человек, он, ознакомившись с диаграммами Rational Rose, без труда войдет в курс дела.

С Rational Rose вам не нужно будет создавать заново те части проекта, которые вел уволившийся сотрудник, ведь разобраться в графических диаграммах намного проще, чем продираться сквозь дебри плохо документированного исходного кода, когда проще и быстрее

переписать код заново, чем разбираться в «тяжелом наследии» предшественника.

Однако нужно понимать, что Rational Rose не всемогущ. Диаграммы не появляются сами по себе. Это результат кропотливого проектирования системы, который лишь отражается на диаграммах, то есть проектировать, все-таки, придется, но с этим пакетом данный процесс будет идти значительно проще и нагляднее.

К процессу проектирования можно и нужно привлекать заказчиков системы, чтобы они еще до написания исходного кода могли видеть процессы, происходящие в системе, поняли ее возможности и ограничения и могли дать свои замечания и пожелания, которые «имелись в виду», но по каким-то причинам не были отражены в проектном задании.

Замечание

Сейчас существует пакет, опять же от компании Rational Software, который позволяет создавать полностью работающее приложение, - это Rational Real Time.

И самое главное, Rational Rose не создаст для вас готовый исходный код. Пакет сможет создать основу для системы, заготовки классов вместе с их взаимодействием, а наполнять методы содержанием должен все-таки программист.

Но, исправив что-либо даже в структуре классов, программист всегда может получить визуальное отражение этих изменений в Rational Rose. Так что переходить к ведению проекта при помощи этого замечательного инструмента никогда не поздно, лишь бы был исходный код, и конечно, сам Rational Rose!

Преимущества от применения Rational Rose

Преимущества от применения Rational Rose значительны.

Это:

- сокращение цикла разработки приложения «заказчик - программист - заказчик». Теперь заказчику нет необходимости ждать первой альфа-версии, чтобы убедиться, что все делается совсем не так, как он ожидал;

- увеличение продуктивности работы программистов. Меньше ручного кодирования - меньше ошибок, меньше ошибок - меньше отладки, меньше отладки - больше продуктивность;

- улучшение потребительских качеств создаваемых программ за счет ориентации на пользователей и бизнес;

- способность вести большие проекты и группы проектов;

- возможность повторного использования уже созданного ПО за счет упора на разбор их архитектуры и компонентов;

- язык UML служит универсальным «мостиком» между разработчиками из разных отделов.

Главное окно Rational Rose

Итак, вы завершили установку пакета и ввели лицензионную информацию, как указано в сопроводительном буклете.

После запуска открывается главное окно программы, показанное на рис. 1.1.

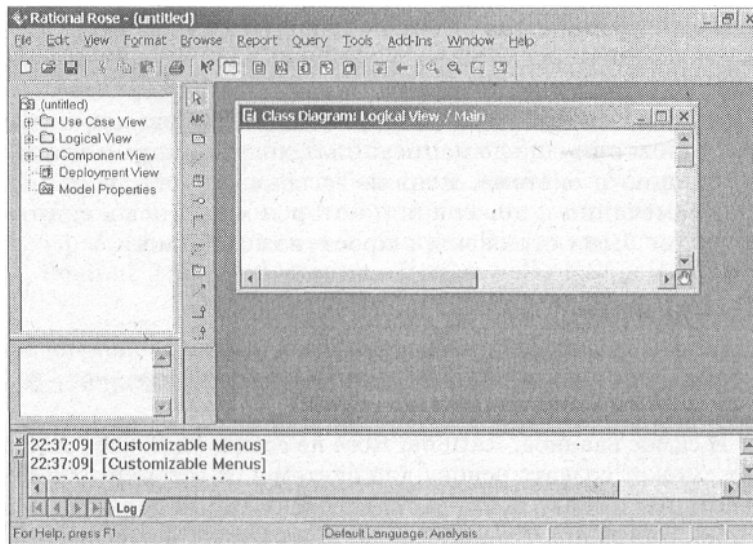


Рис. 1.1 Главное окно программы

Если вы ни разу не видели программу Rational Rose, но знакомы с пакетом Visual Studio, то вы увидите, что у них есть нечто общее. Среда Rational Rose - типичное окно в стиле Microsoft MFC со швартуемыми панелями.

В верхней части экрана, как и у большинства редакторов в стиле Windows, находится меню и строка инструментов (Tool Bar), которую, впрочем, как и в других программах, можно перетащить мышкой в любое удобное место, как показано на рис. 1.2.

Если у вас при запуске программы набор инструментов отличается от показанного на рисунке, не волнуйтесь, их можно настроить, скрывая менее используемые или активизируя те, которые в текущий момент нужны.

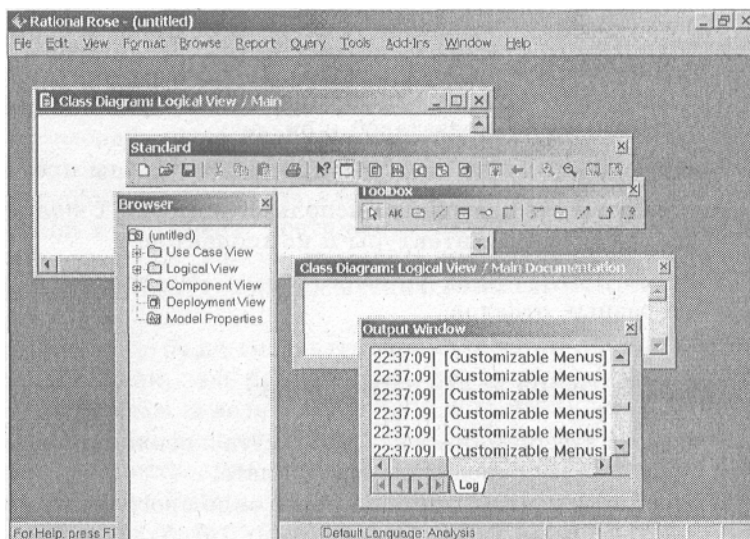


Рис. 1.2 Возможности настройки главного окна

Все окна и строки инструментов можно расположить в зависимости от ваших предпочтений, и такое положение при запуске не является обязательным.

Совет

Не изменяйте месторасположение элементов, по крайней мере, на первых порах работы с пакетом, по той простой причине, что затем можно долго искать отбуксированные в неизвестном направлении или скрытые панели инструментов. Если вы все-таки «потеряли» какие-то панели, загляните в пункт меню View.

Слева находится окно Browser для быстрого доступа к диаграммам. Это окно позволяет легко перемещаться по дереву диаграмм, буксировать диаграммы мышкой и изменять структуру модели по своему усмотрению.

Замечание

Как и во многих пакетах, здесь есть возможность буксировки диаграмм не только в пределах окна Browser, но и непосредственно на Рабочий стол Rational Rose.

Под окном Browser находится окно Documentation (документация). В этом окне появляется описание, которое введено разработчиком для выделенного в текущий момент элемента.

Замечание

Документирование элементов непосредственно в момент разработки является неплохой возможностью, для того чтобы не забыть назначение создаваемых объектов.

В правой части экрана находятся те диаграммы, которые открыты в текущий момент, обычно это поле называется рабочим столом Rational Rose.

При создании новой модели на рабочем столе открывается Class Diagram (диаграмма классов), в нашем случае эта диаграмма пуста,

однако вы можете воспользоваться мастером создания моделей, для того чтобы не начинать с чистого листа.

Замечание

Подробнее мастер создания моделей описан в главе 2, в разделе, посвященном меню File.

Присмотревшись, вы можете заметить маленькую «ладошку» в правом нижнем углу окна диаграммы классов. Как вы в последствии сможете убедиться, этот значок присутствует во всех окнах создаваемых диаграмм. Если на него нажать и удерживать, то появится окно Overview, которое в уменьшенном масштабе покажет содержимое текущего окна. При этом все перемещения мыши будут приводить к перемещению по диаграмме. Как только вы отпустите мышь, то окно Overview будет закрыто, и диаграмма будет отображена в том месте, в которое вы переместились с помощью курсора мыши (рис. 1.3). Окно Overview позволяет легко перемещаться по диаграмме, не меняя ее масштаба, без труда, одним щелчком мыши перебрасывать окно в любую часть диаграммы.

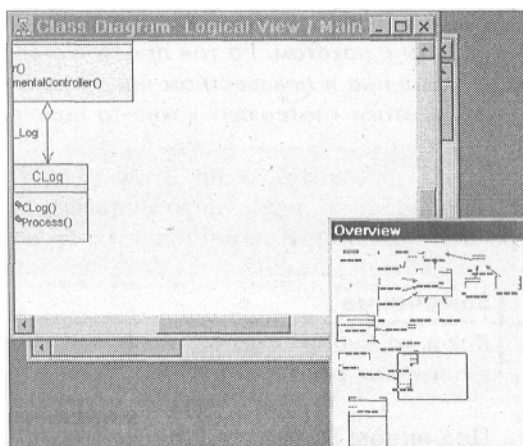


Рис. 1.3 Окно Overview

Между окном Browser и окном Diagram находится строка инструментов, которая изменяется в зависимости от выбранной диаграммы.

Замечание

В этом пакете огромную роль играет контекстное меню, которое активизируется при помощи правой кнопки мыши. Каждый объект в Rational Rose имеет свое контекстное меню, через которое можно изменить свойства объекта. Также посредством этого меню можно получить доступ к дополнительным возможностям, связанным с конкретным объектом, так что не ленитесь пользоваться правой кнопкой мыши в течение работы, это может оказаться удобнее, чем перебирать пункты верхнего меню.

Внизу рабочего стола находится обычно свернутое окно Log (протокол). В нем Rational Rose постоянно фиксирует все действия,

произведенные над диаграммами. Программа помещает туда также и сообщения об ошибках, которые произошли в течение работы. Все сообщения помещаются в окно независимо от того, свернуто ли оно или вовсе закрыто.

Какие диаграммы дает в наше распоряжение Rational Rose?

Вы уже поняли, что UML - это набор диаграмм, которые позволяют создавать модели сложных программных систем. Перед тем как начать вплотную изучать работу с Rational Rose, необходимо кратко ознакомиться с тем, какие диаграммы есть в нашем распоряжении и для чего каждая из них будет использоваться.

В распоряжение проектировщика системы Rational Rose предоставляет следующие типы диаграмм, последовательное создание которых позволяет получить полное представление о всей проектируемой системе и об отдельных ее компонентах:

- Use case diagram (диаграммы сценариев);
- Deployment diagram (диаграммы топологии);
- Statechart diagram (диаграммы состояний);
- Activity diagram (диаграммы активности);
- Interaction diagram (диаграммы взаимодействия);
- Sequence diagram (диаграммы последовательностей действий);
- Collaboration diagram (диаграммы сотрудничества);
- Class diagram (диаграммы классов);
- Component diagram (диаграммы компонент).

Use case diagram (диаграммы сценариев)

Этот вид диаграмм позволяет создать список операций, которые выполняет система. Часто этот вид диаграмм называют диаграммой функций, потому что на основе набора таких диаграмм создается список требований к системе и определяется множество выполняемых системой функций.

Каждая такая диаграмма или, как ее обычно называют, каждый Use case - это описание сценария поведения, которому следуют действующие лица (Actors). Пример такой диаграммы показан на Рис. 1.4.

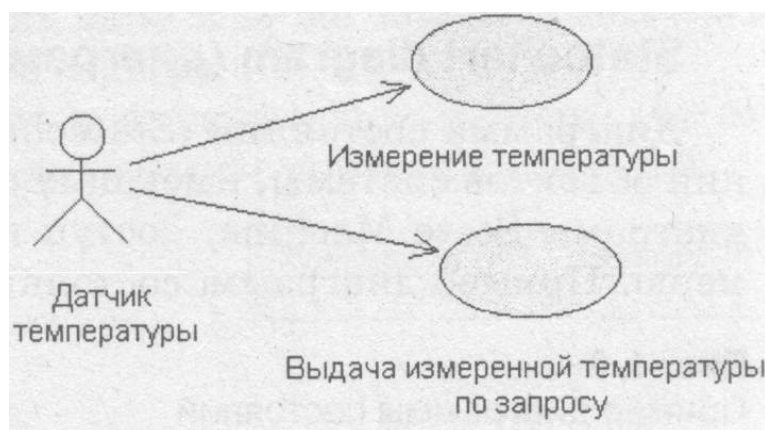


Рис.1.4 Пример диаграммы Use case

Данный тип диаграмм используется при описании бизнес процессов автоматизируемой предметной области, определений требований к будущей программной системе. Отражает объекты как системы, так и предметной области и задачи, ими выполняемые.

Deployment diagram (диаграммы топологии)

Этот вид диаграмм предназначен для анализа аппаратной части системы, то есть «железа», а не программ. В прямом переводе с английского Deployment означает «развертывание», но термин «топология» точнее отражает сущность этого типа диаграмм.

Для каждой модели создается только одна такая диаграмма, отображающая процессоры (Processor), устройства (Device) и их соединения (рис. 1.5).

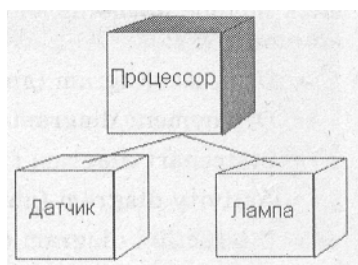


Рис. 1.5 Пример диаграммы Deployment

Обычно этот тип диаграмм используется в самом начале проектирования системы для анализа аппаратных средств, на которых она будет эксплуатироваться.

State Machine diagram (диаграммы состояний)

Каждый объект системы, обладающий определенным поведением, может находиться в определенных состояниях, переходить из состояния в состояние, совершая определенные действия в процессе реализации сценария поведения объекта. Поведение большинства объектов реальных систем можно представить с точки зрения теории конечных автоматов, то есть поведение объекта отражается в его состояниях, и данный тип диаграмм позволяет отразить это графически. Для этого используется два вида диаграмм: Statechart diagram (диаграмма состояний) и Activity diagram (диаграмма активности)

Statechart diagram (диаграмма состояний)

Диаграмма состояний (Statechart) предназначена для отображения состояний объектов системы, имеющих сложную модель поведения. Это одна из двух диаграмм State Machine, доступ к которой осуществляется из одного пункта меню. Пример диаграммы состояний показан на рис. 1.6.

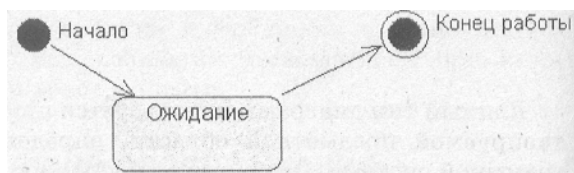


Рис. 1.6 Пример диаграммы состояний

Activity diagram (диаграммы активности)

Это дальнейшее развитие диаграммы состояний. Фактически данный тип диаграмм может использоваться и для отражения состояний моделируемого объекта, однако, основное назначение Activity diagram в том, чтобы отражать бизнес-процессы объекта. Этот тип диаграмм позволяет показать не только последовательность процессов, но и ветвление и даже синхронизацию процессов. Пример диаграммы активности показан на рис. 1.7.

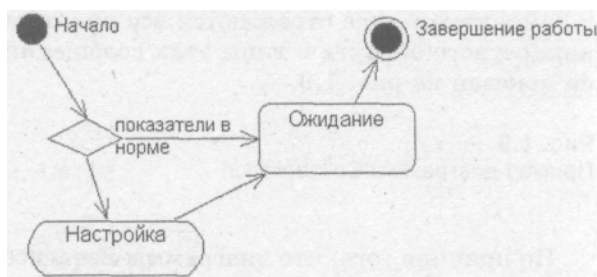


Рис. 1.7 Пример диаграммы активности

Замечание

Этот тип диаграмм позволяет проектировать алгоритмы поведения объектов любой сложности, в том числе может использоваться для составления блок-схем.

Interaction diagram (диаграммы взаимодействия)

Этот тип диаграмм включает в себя диаграммы Sequence diagram (диаграммы последовательностей действий) и Collaboration diagram (диаграммы сотрудничества). Эти диаграммы позволяют с разных точек зрения рассмотреть взаимодействие объектов в создаваемой системе.

Sequence diagram (диаграммы последовательностей действий)

Взаимодействие объектов в системе происходит посредством приема и передачи сообщений объектами-клиентами и обработки этих сообщений объектами-серверами. При этом в разных ситуациях одни и те же объекты могут выступать и в качестве клиентов, и в качестве серверов.

Данный тип диаграмм позволяет отразить последовательность передачи сообщений между объектами. Пример такой диаграммы показан на рис. 1.8.



Рис. 1.8

Пример Sequence диаграммы

Этот тип диаграммы не акцентирует внимание на конкретном взаимодействии, главный акцент уделяется последовательности приема/передачи сообщений. Для того чтобы окинуть взглядом все взаимосвязи объектов, служит Collaboration diagram.

Collaboration diagram (диаграммы сотрудничества)

Этот тип диаграмм позволяет описать взаимодействия объектов, абстрагируясь от последовательности передачи сообщений. На этом типе диаграмм в компактном виде отражаются все принимаемые и передаваемые сообщения конкретного объекта и типы этих сообщений. Пример диаграммы Collaboration показан на рис. 1.9.



Рис. 1.9 Пример диаграммы Collaboration

По причине того, что диаграммы Sequence и Collaboration являются разными взглядами на одни и те же процессы, Rational Rose позволяет создавать из Sequence диаграммы диаграмму Collaboration и наоборот, а также производит автоматическую синхронизацию этих диаграмм.

Class diagram (диаграммы классов)

Этот тип диаграмм позволяет создавать логическое представление системы, на основе которого создается исходный код описанных классов.

Значки диаграммы позволяют отображать сложную иерархию систем, взаимосвязи классов (Classes) и интерфейсов (Interfaces). Данный тип диаграмм противоположен по содержанию диаграмме Collaboration, на котором отображаются объекты системы. Rational Rose позволяет создавать классы при помощи данного типа диаграмм в различных нотациях. В нотации, предложенной Г. Бу-чем, которая так и называется Booch, классы изображаются в виде чего-то нечеткого, похожего на облако. Таким образом Г. Буч пытается показать, что класс - это лишь шаблон, по которому в дальнейшем будет создан конкретный объект. Пример изображения классов в этой нотации показан на рис. 1.10.

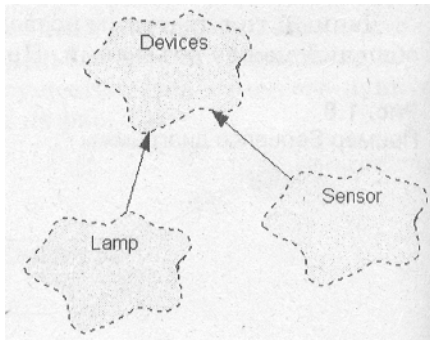


Рис. 1.10. Пример изображения классов в Booch нотации

Нотация ОМТ, на мой взгляд, более строга. Пример изображения классов в нотации ОМТ показан на рис. 1.11.

И конечно же, Rational Rose позволяет создавать диаграмму классов в унифицированной нотации, пример изображения классов в которой показан на рис. 1.12.

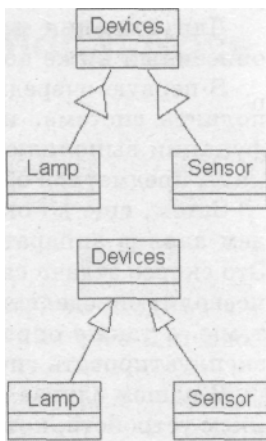


Рис. 1.11, Рис. 1.12

Пример изображения классов в нотации ОМТ

Пример изображения классов в нотации Unified

Component diagram (диаграммы компонентов)

Этот тип диаграмм предназначен для распределения классов и объектов по компонентам при физическом проектировании системы. Часто данный тип диаграмм называют диаграммами модулей.

При проектировании больших систем может оказаться, что система должна быть разложена на несколько сотен или даже тысяч компонентов, и этот тип диаграмм позволяет не потеряться в обилии модулей и их связей. Пример диаграммы компонентов показан на рис. 1.13.

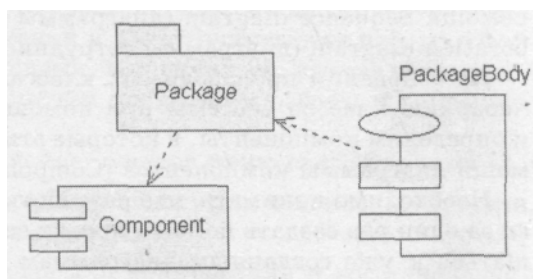


Рис. 1.13 Пример диаграммы компонентов

Общий порядок работы

Для создания систем различных предметных областей общий порядок работы может несколько отличаться от приведенного, поэтому при разработке другой системы необходимо внести в него соответствующие изменения.

Замечание

Здесь мы не будем глубоко вникать в процесс проектирования систем, так как цели книги больше практические. В книге Г. Буча подробно разбирается этот процесс, и дается пример создания пяти программных систем из разных областей.

Для создания программной системы нашего класса будем использовать описанный ниже порядок работы.

В первую очередь произведем анализ списка операций, которые будет выполнять система, и определим список объектов системы, которые данные функции выполняют. Таким образом, определим требования к системе и границы предметной области. Для этого будем использовать диаграммы Use case.

Затем, еще до окончательной детализации сценариев поведения, произведем анализ аппаратной части системы при помощи Deployment диаграммы. Это скорее задача системного анализа, чем практическая. В общем случае она позволит определиться в таких вопросах как технологичность и стоимость системы, а также определить набор аппаратных средств, на которых предстоит эксплуатировать систему.

В нашем случае анализ аппаратной части системы ограничится перечислением устройств, которые будут работать под управлением нашего программного обеспечения и ничего более.

Возможно именно аппаратные средства внесут свои коррективы, ограничения или дополнительные требования к создаваемому программному обеспечению.

Затем определим список классов, которые должны присутствовать в системе, пока без конкретной детализации и подробного описания действий. Для этого будем использовать диаграмму классов (Class diagram).

После заведения в системе необходимых классов определим поведение конкретных классов при помощи диаграмм Statechart diagram (диаграмма состояний) и Activity diagram (диаграммы активности).

Дальнейшая детализация взаимодействия классов будет

производиться при помощи Sequence diagram (диаграммы последовательностей действий), Collaboration diagram (диаграммы сотрудничества).

На основании производимых классами действий создадим окончательную иерархию классов системы при помощи диаграммы классов (Class diagram) и определим компоненты, в которые эти классы необходимо включить при помощи диаграммы компонентов (Component diagram).

Необходимо понимать, что разработка - это итерационный процесс. Нельзя за один раз создать полный проект системы. Придется многократно возвращаться к уже созданным диаграммам и вносить в них изменения. Если вы хоть раз писали программу, то, я полагаю, этот процесс вам знаком.

Вопросы для повторения

1. Что представляет собой программа Rational Rose?
2. Что такое язык UML?
3. Почему Rational Rose является лидером среди CASE-средств?
4. Что может и чего не может делать Rational Rose?
5. Какие преимущества дает применение Rational Rose при разработке программных систем?
6. Какие UML диаграммы доступны в Rational Rose?
7. Каково назначение каждого типа диаграмм?

Глава 2. Знакомство с меню Rational Rose

Структура меню

Для того чтобы лучше ориентироваться в Rational Rose, необходимо подробно узнать о структуре меню и возможностях применения того или иного пункта меню.

Замечание

Как и во многих программах для Windows, в Rational Rose есть возможность настроить верхнее меню на свой вкус. Есть еще одна интересная особенность, которая вытекает из возможности подключения внешних модулей. Все меню в программе описываются на специальном языке и хранятся в простых текстовых файлах с расширением .mnu. Главное меню называется Rose.mnu, так что есть возможность не просто настроить меню по своему усмотрению, а создавать динамически изменяемое меню в зависимости от подключенных внешних модулей и запущенных скриптов.

Для работы будем использовать меню, которое предоставляется по умолчанию, без каких либо изменений. Структура меню похожа на структуру меню большинства других Windows приложений и имеет пункты для работы с файлами, форматирования, управления окнами, и конечно же, вызова встроенной справки, где можно подробно ознакомиться с порядком работы и описанием программы.

Кратко перечислим состав и назначение пунктов меню слева направо:

File (файл) предназначен для сохранения, загрузки, обновления проекта, печати диаграмм и дополнительных настроек;

Edit (редактирование) предназначен для копирования и восстановления данных в буфер обмена Windows, а также для редактирования свойств и стилей объектов;

View (вид) предназначен для настройки представления окон меню и строк инструментов;

Format (форматирование) предназначен для настройки формата текущего значка, цветовой гаммы, линий и т.д.;

Browse (просмотр) предназначен для навигации между диаграммами и спецификациями диаграмм, представленных в модели;

Report (отчет) предназначен для получения различного вида справок и отчетов;

Query (запрос) предоставляет возможности контролировать, какие элементы модели будут показаны на текущей диаграмме;

Tools (инструменты) предоставляет доступ к различным дополнительным инструментам и подключаемым модулям;

Add-Ins (добавить) предоставляет доступ к менеджеру подключаемых модулей;

Window (окно) позволяет управлять окнами на рабочем столе;

Help (помощь) позволяет получать справочную информацию.

File (файл)

Пункт File позволяет производить работу с файлами диаграммы, сохранять, создавать, печатать диаграммы (рис. 2.1). Также в данном меню имеются дополнительные возможности для работы, такие как импорт или экспорт частей модели.

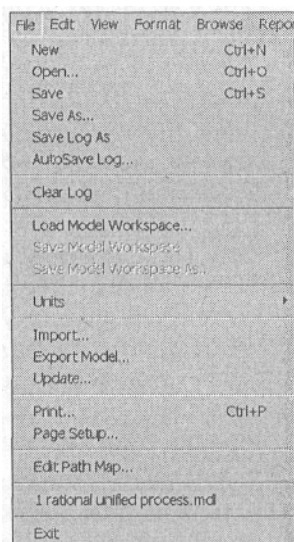


Рис. 2.1 Меню File

Кратко рассмотрим пункты этого меню.

New (новый)

Используйте данную команду для создания новой модели. Вы можете воспользоваться горячими клавишами Ctrl+N для вызова этого пункта.

При создании новая модель имеет заголовок untitled (без заголовка) до тех пор, пока ей не будет присвоено имя при помощи команды Save (сохранить) или Save As (сохранить как).

При создании новой модели все открытые на данный момент диаграммы будут закрыты и, если имеются несохраненные данные, то будет задан вопрос на сохранение таких данных.

По умолчанию новая модель создается с одной только диаграммой - диаграммой классов, которая в начале работы пуста.

При выборе данного пункта может быть активизирован мастер создания модели (рис. 2.2).

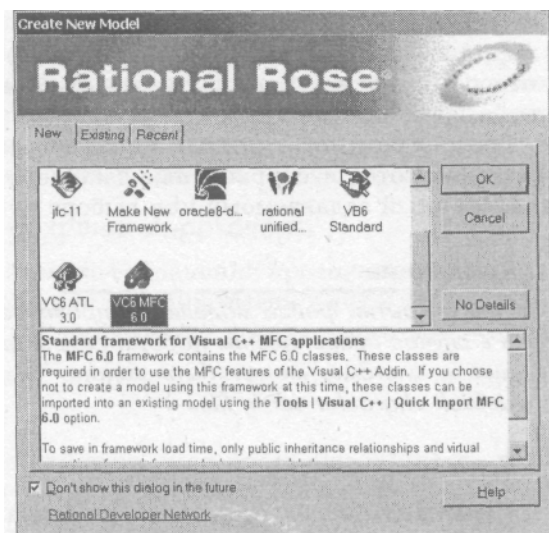


Рис. 2.2 Создание новой модели

Совет

Для того чтобы это окно не появлялось в дальнейшем, установите флажок **Don't show this dialog in the future**. Если вы передумали и хотите снова работать с мастером, установите флажок **Menu:Add-Ins=>Framework Wizard=ON**.

Мастер позволяет создать модель на основе шаблона, учитывающего цели и архитектуру будущей системы, а также загрузить уже созданные модели. При помощи вкладки **Existing** (существующие) можно выбрать из списка уже созданных проектов. Вкладка **Recent** (недавно используемые) позволяет открыть проект из списка последних используемых проектов. Также имена последних использовавшихся проектов автоматически включаются в меню **File**. Кнопка **Detail** (детализировать) позволяет включить просмотр описания шаблона создаваемого проекта (на рисунке включено) или скрыть его.

И конечно же, компания Rational Software не упустила возможности включить ссылку на информационный ресурс в сети Internet (ссылка на **Rational Developer Network** в нижней части экрана).

Создание новой модели на основе шаблона будет хорошим решением для экономии времени и сил. Если вы создаете программную систему с использованием конкретного языка программирования, такого как **Visual C++** или **Visual Basic**, создаете модель данных **Oracle** или создаете несколько однотипных моделей, то используйте готовые шаблоны. Шаблоны уже включают классы библиотек, таких как **MFC** и определенные типы данных. Вы также всегда можете создать свой шаблон, для чего при создании новой модели нужно выбрать значок «**Make New Framework**».

Открыть (открыть)

Пункт **Open** позволяет открыть уже созданную модель. Горячие клавиши **Ctrl+O**.

При выборе данного пункта активизируется системное окно выбора файла, которое позволяет задать имя необходимого файла или найти этот

файл на диске.

Все диаграммы, открытые в момент вызова данного пункта, будут закрыты, и если имеются несохраненные данные, то будет задан вопрос на сохранение, аналогичный задаваемому при выборе пункта New.

Замечание

При открытии файла модели, он проверяется на установленный набор символов и в случае несовпадения набора символов, сохраненных в модели, с установленным в системе, может быть выдано следующее предупреждение «Non system default character set in file».

Save (сохранить)

Пункт Save используется для сохранения текущей модели. Горячие клавиши - Ctrl+S. В случае если модели еще не было присвоено имя, то активизируется окно Save As (сохранить как), которое позволяет задать новое имя модели.

Если у файла текущей модели был установлен атрибут read-only (только чтение), то будет активизировано окно сообщения об ошибке.

При сохранении в модели будет установлен используемый в системе по умолчанию набор символов.

При сохранении модели возможно создание файла резервной копии последнего состояния. Для этого установите флажок Menu:Tools=>Options==Create backup file=ON.

Save As (сохранить как)

Пункт Save as используется для сохранения текущей модели в новый файл. При этом активизируется диалог сохранения файла, который позволяет задать путь и имя файла.

Этот пункт позволяет выбрать формат файла для сохранения, например, сохранить в формате предыдущих версий Rational Rose.

Если вы постоянно работаете с форматами файлов предыдущих версий, например версии 4.0, то вместо того чтобы каждый раз задавать этот формат при сохранении, установите в файле rose.ini в секции «Rational Rose» параметр Default-PetalVersion=40.

Save Log As (сохранить Save

Save Log As (сохранить протокол как)

Позволяет сохранить информацию в файл из окна Log (протокол). Работа с протоколом добавилась в последних версиях программы. Раньше информация из протокола просто терялась, поскольку ей не уделялось большого внимания, сейчас же с протоколом можно работать вполне комфортно. Информация может быть сохранена, восстановлена или полностью очищена. Этот пункт меню позволяет сохранить протокол под новым именем. Эта команда доступна также из контекстного меню окна протокола.

Auto Save Log (автоматически сохранять протокол)

Активизирует функцию автоматического сохранения протокола. При выборе этого пункта меню активизируется окно задания имени файла для автоматического сохранения. Команда также доступна из контекстного меню окна протокола.

Clear Log (очистить протокол)

Пункт меню позволяет полностью очистить окно протокола. Очистка происходит без каких-либо запросов и предупреждений. Таким образом перед тем, как воспользоваться этой функцией необходимо точно решить, что протокол уже не нужен, поскольку восстановление протокола (если он не был заранее сохранен) невозможно.

Load Model Workspace (загрузить рабочую область модели)

Пункт позволяет загрузить сохраненную ранее рабочую область. По умолчанию расширение рабочей области .wsp. Главное отличие рабочей области от модели в том, что в ней сохраняются не все диаграммы, а только часть их, определенная пользователем. Таким образом, можно получить мгновенный снимок (snapshot) активных диаграмм в текущей модели. Также рабочие области могут использоваться при командной работе над большим проектом.

Save Model Workspace (сохранить рабочую область модели)

Пункт позволяет сохранить рабочую область модели, все активные в данный момент диаграммы в отдельный файл с расширением .wsp для последующей загрузки при помощи предыдущего пункта.

Save Model Workspace As (сохранить рабочую область модели как)

Пункт позволяет создать новый файл рабочей области под новым именем. При вызове этого пункта активизируется окно задания имени файла, под которым будет сохранена рабочая область.

Units (единицы)

Пункт меню Units позволяет получить доступ к подменю управления контролируемыми единицами.

Rational Rose хранит модель в одном или нескольких файлах. Эти файлы и называются controlled units (контролируемые единицы).

Controlled units могут хранить не только информацию о модели, но и конфигурационные установки для команды разработчиков, которые работают вместе и используют программу контроля версий (Version control).

Так, controlled units могут создаваться по частям различными разработчиками, и Rational Rose контролирует их совместное использование таким образом, что возможна параллельная работа над проектом.

Если контроль версий не установлен, то данное меню будет недоступно.

Import (импорт)

Пункт Import используется для импорта файлов в модель. В Rational Rose возможен импорт нескольких типов файлов:

- petal (лепесток) - расширение .ptl;
- model (модель) - расширение .mdl;
- category (категория) - расширение .cat;
- subsystem (подсистема) - расширение .sub.

Этот пункт активизирует диалог выбора Petal файла, который может содержать классы, логические пакеты, компоненты или полную модель. Для импорта требуется, чтобы была активна диаграмма классов или диаграмма компонентов.

Элементы импортируются в текущую диаграмму или в текущий пакет в активной диаграмме, при этом если импортируемые элементы уже существуют на диаграмме, то будет активизирован запрос на перезапись этих элементов.

Export model (экспорт)

Пункт Export model активизирует диалог записи Petal файла с возможностью указания имени и местоположения экспортируемого файла.

Экспортируются только выделенные элементы текущей модели, при этом в названии этого пункта появляется название выделенного элемента или количество элементов, если их выделено больше одного.

Посредством данного пункта могут быть экспортированы следующие элементы:

- вся модель;
- классы;
- логические пакеты;
- компоненты.

Если будут выделены элементы, не входящие в данный список, то экспорт будет невозможен.

В случае экспорта логических пакетов или компонентов все диаграммы, которые содержатся в данных пакетах и компонентах, будут также экспортированы.

В случае экспорта отдельных классов с ними будут также экспортированы диаграммы состояний этих классов.

В случае экспорта всей модели экспортируются все диаграммы, находящиеся в данной модели.

Используйте экспорт если необходимо:

- перенести отдельный элемент в другую диаграмму;
- перенести модель или отдельные ее элементы на другую компьютерную платформу;
- перенести модель или отдельный ее элемент в новую версию программы.

Print (печать)

Пункт Print активизирует окно настроек печати диаграммы (рис. 2,3). Этот пункт также доступен по горячим клавишам Ctrl+P. Возможность напечатать диаграммы модели, которые чаще всего не помещаются на один лист на принтере формата A4, реализована в Rational Rose очень хорошо. Диаграммы могут быть напечатаны в различном масштабе, только выделенный или все в текущей модели. Для того чтобы пользователь мог ориентироваться в том, как на листе будут расположены диаграммы есть возможность включить просмотр линий разделения страниц. А сами страницы могут быть напечатаны с перекрытием для склеивания или без такового, если планируется склейка встык. Таким образом настройка печати диаграмм становится не совсем тривиальным занятием.

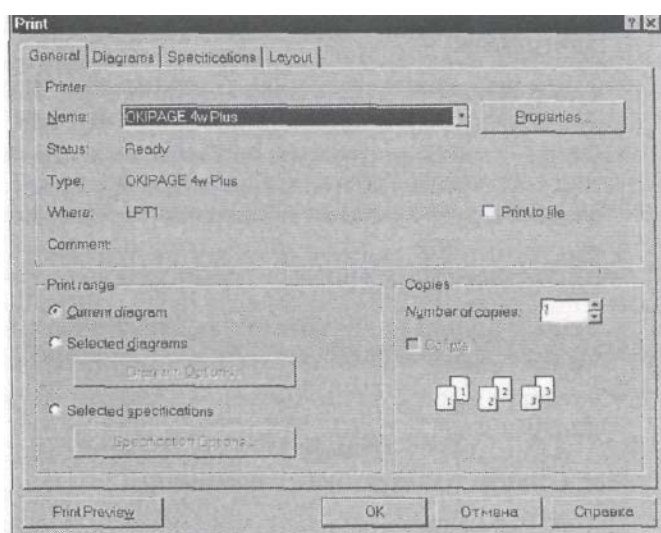


Рис. 2.3 Настройки печати модели

Диалоговое окно печати включает четыре вкладки. Вкладка General позволяет установить принтер для печати и настроить диапазон выводимых на печать диаграмм и спецификаций. Для доступа к вкладкам Specifications и Diagrams необходимо переключить соответствующую радио кнопку на вкладке General. При этом можно ограничить печать только выбранными диаграммами или спецификациями.

Вкладка Layout, показанная на рис. 2.4, позволяет задавать размещение диаграмм при печати.

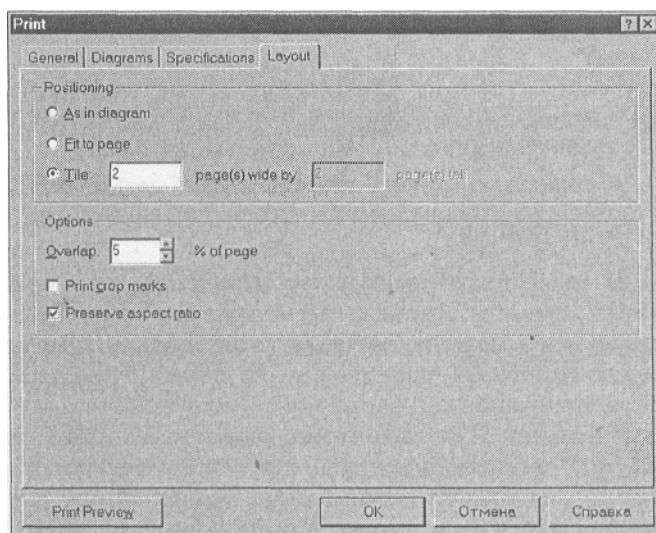


Рис. 2.4. Вкладка Layout настройки печати

На этой вкладке присутствуют следующие поля:

As in diagram означает печать диаграмм в том размере, в котором создана диаграмма.

Fit to page означает печать диаграмм на одном листе с уменьшением масштаба диаграммы таким образом, чтобы диаграмма уместилась на лист.

Tile - печать диаграммы на нескольких листах. При этом можно указать как количество колонок (pages wide), так и количество рядов (page tall), от чего будет зависеть масштаб изображения.

Также можно указать процент перекрытия изображения на листах для склейки (Overlap).

Print crop marks указывает что будут напечатаны линии по которым диаграмма должна быть склеена после печати.

Preserve aspect ratio позволяет сохранить пропорции первоначальной диаграммы.

Page setup (настройка страницы)

Пункт Page setup позволяет управлять настройками страниц при печати на принтер. При его выборе активизируется окно, в котором можно установить ориентацию страницы, поля и размер бумаги, выбрать и настроить принтер.

Edit Path Map (редактирование карты путей)

Пункт Edit Path Map позволяет изменять карту виртуальных путей в программе (рис. 2.5).

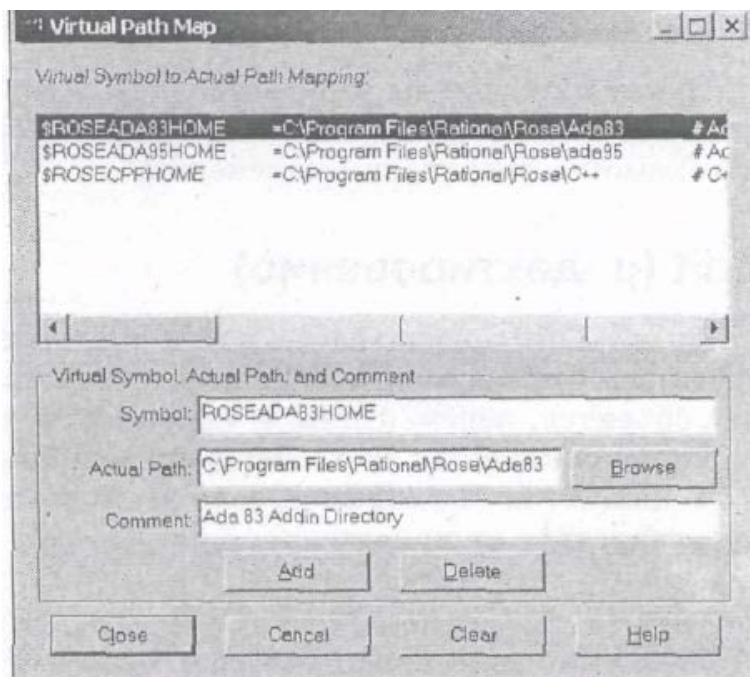


Рис. 2.5 Окно Virtual Path Map

Так как Rational Rose хранит пути к различным файлам одной модели, то этот режим позволяет переносить модель между компьютерами с различной структурой каталогов. Для организации такой поддержки в программе используется специальный механизм, называемый Virtual path maps (карты виртуальных путей). При работе с файлами модели вместо абсолютных путей используются виртуальные.

Этот механизм работает следующим образом. Когда сохраняется файл модели, Rational Rose пытается заменить каждый абсолютный путь на виртуальный. При следующем открытии файла каждый виртуальный путь трансформируется обратно в абсолютный.

Например, если пользователь определил следующий виртуальный путь:

\$MYPATH=Z:\ordersystem Затем сохранил пакет как: Z:\ordersystem\user_services.cat Тогда модель получает следующую ссылку на пакет: \$MYPATH\user_services.cat

Если другой пользователь определил \$MYPATH как: \$MYPATH=X:\ordersystem

Затем он открывает ту же модель на диске «X», в этом случае Rational Rose преобразует виртуальный путь в абсолютный следующим образом:

X:\ordersystem\user_services.cat

Подробнее об алгоритме трансформации виртуальных и абсолютных путей смотрите во встроенной справке в статье «Path Map Algorithm».

Exit (выход)

Пункт Exit, как вы уже догадались, завершает работу программы. В случае наличия изменений с момента последнего сохранения будет задан вопрос о необходимости сохранения изменений.

Edit (редактирование)

Пункт Edit предназначен для копирования и восстановления данных посредством буфера обмена Windows, а также для редактирования свойств и стилей объектов, перемещения и удаления объектов (рис. 2.6).

Горячие клавиши этого меню аналогичны клавишам в других приложениях Windows, таким образом, если вы привыкли, например, к Visual Studio, то переучиваться не придется.

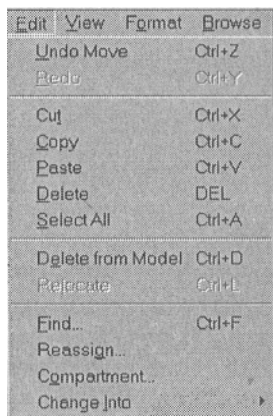


Рис. 2.6 Меню Edit

Undo (отмена)

Пункт Undo позволяет отменить последнее действие, такое как копирование, удаление, перемещение объектов диаграммы. Горячие клавиши - Ctrl+Z. К сожалению, Rational Rose не хранит историю изменений, как некоторые приложения Windows, поэтому и отменить можно только последнее произведенное действие.

Redo (вернуть)

Пункт Redo позволяет вернуть последнюю отмененную команду, причем команда, которую можно вернуть, отображается в меню рядом с командой Redo. Горячие клавиши - Ctrl+Y.

Cut (вырезать)

Пункт Cut удаляет выделенный элемент из диаграммы и помещает его в буфер обмена Windows.

Если пользователь пытается вырезать такие элементы, которые потом не могут быть вставлены в диаграмму, например, элемент связи (relationship) без выделения объектов, то будет выдано предупреждение.

Если вырезается элемент, то вырезаются также и все его связи на диаграмме. Горячие клавиши - Ctrl+X.

Замечание

Данная команда отражается только на графическом представлении диаграммы, но не затрагивает саму модель, то есть не приводит к

удалению элементов модели и удаленные таким образом элементы доступны в окне Browse.

Сору(копировать)

Аналогичен пункту Cut, но без удаления выделенного элемента диаграммы. Горячие клавиши - Ctrl+C.

Rational Rose поддерживает OLE, поэтому есть возможность вставить скопированный значок или целую диаграмму в другие приложения Windows. Таким образом, для того чтобы вставить диаграмму, например в MS Word достаточно выделить ее в окне Browse и скопировать в буфер обмена. После этого она будет доступна для вставки как графический объект.

Paste (вставить)

Пункт Paste позволяет вставить элемент из буфера в текущую диаграмму. Добавляемый элемент помещается в центр текущего окна диаграммы. Для того чтобы установить место, куда будет вставлен элемент, необходимо заранее переместить текущее окно в нужное место диаграммы.

Совет

Для того чтобы быстро переместиться по диаграмме, используйте окно Overview, как описано в главе 1, в разделе «Главное окно Rational Rose».

Если вы пытаетесь вставить элемент из другой модели, имя которого уже присутствует в текущей, то, без каких-либо предупреждений, на диаграмме будет создано представление существующего элемента.

Совет

Для того чтобы вставить в текущую модель элемент другой модели, воспользуйтесь Menu:Edit=>Import, предварительно экспортировав нужные элементы.

Горячие клавиши - Ctrl+N.

Delete (удалить)

Пункт Delete предназначен для удаления выделенного элемента из диаграммы. Этот пункт не удаляет элементы из модели. Однако если элемент не имеет наименования, то он будет удален не только из текущей диаграммы, но и из модели.

Если Delete используется для удаления спецификаций, то выделенный элемент удаляется из модели и из всех диаграмм, в которых эти спецификации присутствуют, и такая команда не может быть отменена.

При удалении элементов диаграммы также удаляются и все связи данных элементов. Горячая клавиша - Delete.

Select All (выделить все)

Данная команда позволяет выделить сразу все элементы текущей диаграммы.

Для выделения нескольких отдельных элементов используйте левую кнопку мыши, одновременно удерживая клавишу Ctrl. Для того чтобы снять выделение, укажите мышкой в любое место диаграммы, которое не попало в выделение.

Эту команду удобно использовать для генерации отчета, изменения шрифта, цвета или просто перемещения сразу для всех элементов диаграммы. Горячие клавиши - Ctrl+A.

Delete from Model (удалить из модели)

Пункт Delete from Model, в отличие от команды Delete (удалить), всегда удаляет выделенные элементы из модели. При этом изображение элемента удаляется из всех диаграмм, в которых он присутствовал. Этот пункт не доступен для спецификаций. Горячие клавиши - Ctrl+D.

Relocate (переместить)

В процессе анализа и разработки приложения логическая и физическая архитектура проекта может меняться от одной итерации к другой. Так, возможно перемещение классов из одного логического пакета в другой или перемещение классов из одного компонента в другой.

Пункт Relocate поддерживает этот процесс и позволяет переместить элементы: классы (class), компоненты (component), пакеты (package) или ассоциации (association) в новые логические пакеты или компоненты. Горячие клавиши - Ctrl+L.

Find (поиск)

Пункт Find активизирует окно поиска по модели (рис. 2.7).

Если в момент активизации режима был выделен один из объектов на диаграмме, то его имя подставляется в строку поиска. Для поиска можно использовать символ * для задания любых символов. Так, для поиска всех элементов, начинающихся на A, необходимо задать A*. Строка поиска ограничивается 250 символами. Горячие клавиши - Ctrl+F.

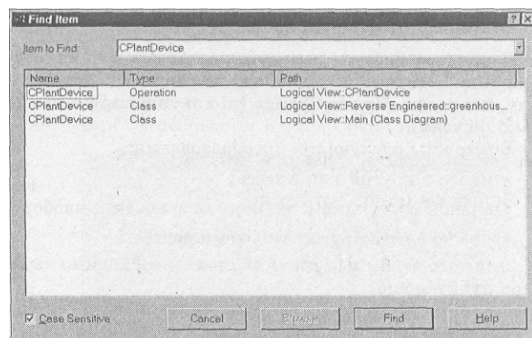


Рис. 2.7 Окно поиска по модели

Reassign (переназначение)

Каждый значок на диаграмме является представлением элемента модели, и пункт Reassign позволяет переназначить выделенный элемент на элемент из другой диаграммы или назначить один и тот же элемент в разные диаграммы.

Для переназначения необходимо выделить элемент и выбрать этот пункт меню, после чего будет активизировано окно переназначения класса, представлено на рис. 2.8.

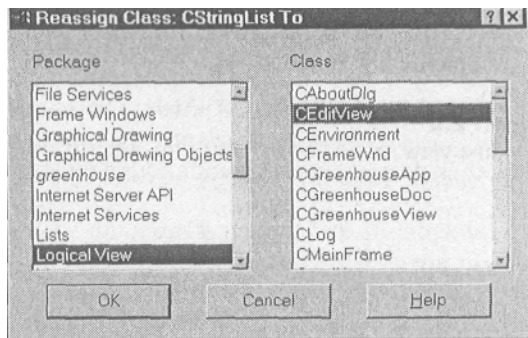


Рис. 2.8 Переназначение класса

После выбора выделенный элемент будет заменен на выбранный, а остальные элементы останутся в своем оригинальном виде. Этот пункт доступен, если выбран только один элемент на диаграмме.

Compartment (содержание)

Активизирует окно выбора для скрытия/показа списка атрибутов и операций.

Замечание

Подробнее работа с данным пунктом будет рассмотрена в главе 11, в разделе, посвященном меню Options.

Change Into (преобразовать в)

В процессе работы над моделью может возникнуть необходимость изменения типа элемента модели. Так, вам может понадобиться изменить тип класса или тип связи. Пункт Change Into активизирует подменю со списком возможных значений.

Возможны следующие преобразования:

класс в другой тип класса;

значок состояния в значок активности и наоборот;

компонент в другой тип компонента.

Для того чтобы изменить элемент, необходимо выделить элемент и выбрать этот пункт меню.

Замечание

Другим способом изменения типа элемента будет выбор необходимого значка в строке инструментов диаграммы, затем выбор

мышкой элемента на диаграмме, который необходимо изменить, удерживая клавишу Alt.

View (вид)

Меню предназначено для настройки представления окон меню и строк инструментов (рис. 2.9). Состав меню изменяется в зависимости от текущего активного окна. Например, при активном окне протокола в этом меню остаются всего три первых пункта. Поэтому рассмотрим состав меню при активном окне диаграммы классов.

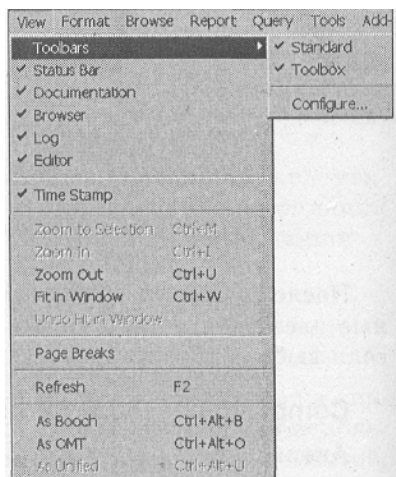


Рис. 2.9 Меню View

Toolbars (инструменты)

Посредством пункта Toolbars производится включение/скрытие линеек инструментов Standard - главная линейка инструментов, ToolBox - линейка инструментов окна диаграмм. При помощи пункта Configure возможна настройка состава линеек инструментов посредством окна Options (рис. 2.10).

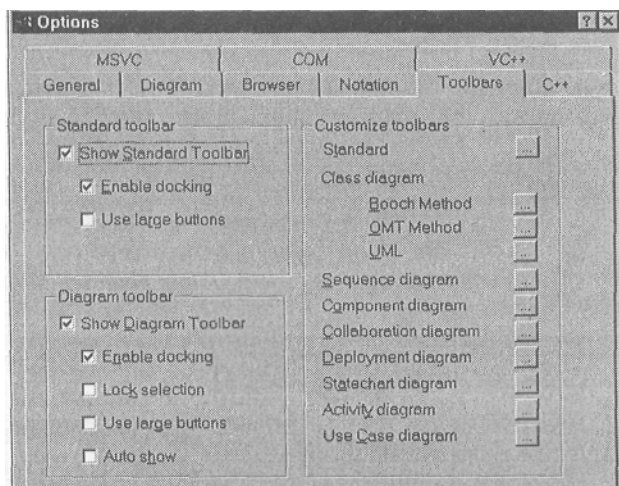


Рис. 2.10 Окно настройки состава панели инструментов

Это окно позволяет настроить стандартную панель инструментов при помощи группы элементов Standard Toolbar.


Show Standard Toolbar позволяет включить или выключить стандартную линейку инструментов.

Enable docking позволяет при перемещении строки инструментов по экрану «прикрепляться» к сторонам экрана, сливаясь с главным окном.

Use large buttons устанавливает кнопки на строке инструментов нормального или большого размера.

Для панели инструментов диаграммы перечисленные настройки аналогичны.

Lock selection определяет поведение кнопки на панели инструментов после создания элемента. Если данный пункт не выбран, то после создания элемента происходит переключение на кнопку Selection

Tool. 

В противном случае переключения не происходит.

Если необходимо создать несколько элементов одного типа, то можно установить флажок Lock selection и создать столько элементов, сколько необходимо, без постоянного выбора типа элемента.

Блок Customize toolbars позволяет изменять состав линеек инструментов.

Status Bar (строка состояния)

Пункт Status Bar позволяет включать и выключать строку состояния, расположенную внизу рабочего стола Rational Rose.

Строка состояния показывает информацию о процессе загрузки модели или информацию о том, что модель открыта в режиме read-only (только чтение).

Также там показывается путь на диске к файлам controlled unit или краткая подсказка.

Documentation (документация)

Пункт Documentation позволяет показать или, скрыть окно документации для выбранного элемента диаграммы. Окно документации показывает содержание поля Documentation в спецификациях объекта. Эта возможность позволяет быстро получать доступ к документации без активизации окна Specification объекта. Если размер окна не позволяет вместить всю необходимую информацию, то автоматически добавляются полосы прокрутки.

Замечание

Окно документации показывает информацию только одного объекта, и может быть активизировано только одно такое окно. При выборе нескольких объектов в нем высвечивается только Multiple selection (выбрано несколько объектов), но не документация.

Browser (окно просмотра)

Активизирует или скрывает окно просмотра Browser, предназначенное для быстрой навигации по объектам модели.

Нажатие на знак « + » раскрывает следующий уровень диаграмм, «-» означает, что дальше уровней нет.

Двойное нажатие в окне Browser левой кнопки мыши на имени или иконке диаграммы позволяет открыть эту диаграмму на рабочем столе Rational Rose.

При двойном нажатии левой кнопки мыши на любом другом элементе раскрывается окно спецификаций для данного элемента.

Log (протокол)

Активизирует или скрывает окно протокола, отображающего протокол работы. Работу с этим окном мы рассмотрели при обсуждении меню File.

Editor (редактор)

Активизирует или скрывает окно редактора. Редактор позволяет отображать исходный код классов, ассоциированный с языком ANSI C++. В окне могут быть отображены следующие типы файлов: .cpp, .h, .Java, .idl, .xml, .dtd. В отличие от других окон, редактор не изменяет своего содержимого в зависимости от выбранного значка класса. И в случае, когда открыто несколько файлов в редакторе, это окно получает вкладки по одной на каждый файл (рис. 2.11)

Совет

Для подключения внешнего редактора используйте Menu:Tools=>ANSI C++=>Preferences=>Custom Editor

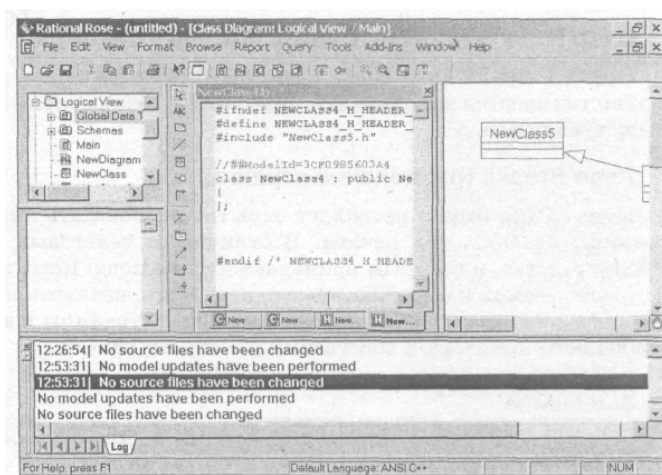


Рис. 2.11 Окно Editor

Time Stamp (время)

Позволяет задать или скрыть отображение времени создания записи протокола. При установленном маркере все записи, которые возникают в

окне Log (протокол) предваряются часом, минутой и секундой создания записи. Если снять маркер, то записи будут создаваться без указания такой информации. На уже созданные записи этот параметр влияния не оказывает (рис. 2.11).

Zoom to Selection (изменить масштаб до выделенного)

Пункт Zoom to Selection позволяет изменить масштаб диаграммы таким образом, чтобы все выделенные элементы полностью уместились в окно и общий центр всех выделенных элементов при этом перемещается в центр окна. Очень удобная функция, когда необходимо редактировать несколько связанных друг с другом элементов диаграммы. Горячие клавиши - Ctrl+M.

Zoom In (увеличить)

Пункт Zoom In позволяет «приблизить» диаграмму. Этот режим действует как лупа, при помощи которой можно увеличить диаграмму. Горячие клавиши - Ctrl+I.

Zoom Out (уменьшить)

Пункт Zoom Out позволяет «отдалить» диаграмму. Режим противоположен предыдущему. Горячие клавиши - Ctrl+U.

Fit in Window (подогнать под размер окна)

Пункт Fit in Window позволяет изменить масштаб диаграммы таким образом, чтобы в окно умещались все элементы диаграммы. Горячие клавиши -Ctrl+W.

Undo Fit in Window (отменить подгонку размера)

Пункт Undo Fit in Window позволяет отменить изменение масштаба, когда в окно умещаются все элементы диаграммы. Режим отменяет выполнение предыдущего пункта.

Page Breaks (разделение страниц)

Пункт Page Breaks позволяет скрыть или показать линии, показывающие границы страниц при печати. В отличие от текстовых редакторов, многие CASE-средства, к которым принадлежит и Rational Rose, предоставляют пользователю возможность работать с одним «бесконечным» листом, который при печати программа «разрезает» на отдельные страницы в зависимости от установленного принтера и собственных настроек печати.

Замечание

Подробно настройки печати обсуждаются в разделе, посвященном меню File, пункт Print.

В случае, когда диаграмма занимает больше одного листа принтера,

удобно видеть, где проходит граница печати, для того чтобы соответствующим образом располагать элементы диаграммы и избежать их попадания на стык листов.

Refresh (обновить)

Пункт Refresh позволяет обновить изображение во всех открытых окнах.

Если в течение сеанса работы вы видите, что после перемещения элементов остались какие-то непрорисованные участки диаграммы, просто нажмите F2, и диаграмма будет обновлена.

As Booch, As OMT, As Unified

Эти пункты предназначены для переключения между различными нотациями представления диаграмм.

Rational Rose позволяет создавать диаграммы в различных нотациях, которые представляют собой определенный набор правил и соглашений. Причем, что приятно, переключение можно осуществить в любой момент, а не только при создании новой диаграммы.

Основное отличие этих нотаций в визуальном изображении классов, объектов и их связей. Таким образом, проект, созданный в одной нотации, можно мгновенно преобразовать в другую, однако, при использовании некоторых возможностей, доступных в одной нотации и недоступных в другой, эти возможности будут преобразованы в доступные, что, впрочем, при обратном преобразовании восстановится без потерь.

Полностью таблица отличий этих нотаций дана в сопроводительной документации. Здесь и далее будем использовать унифицированную нотацию, которая более компактна. Поэтому, чтобы получить аналогичное показанному в примерах, отображение, не забудьте выбрать пункт As Unified.

Format (формат)

Пункт Format позволяет изменять параметры отображения объектов, такие как шрифт, заливку, формат линий и т.д. (рис. 2.12).

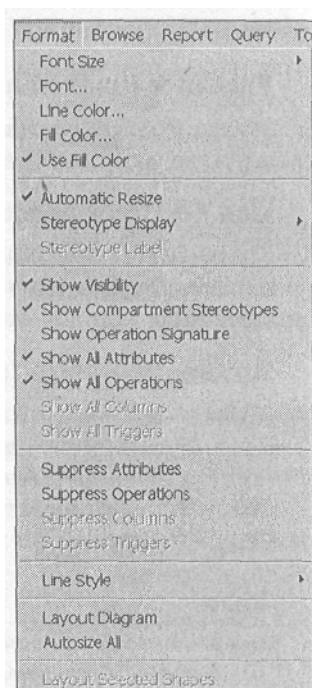


Рис. 2.12 Меню Format

Font Size (размер шрифта)

Пункт Font Size позволяет изменять размер шрифта надписей для выбранного элемента диаграммы. Размер изменяется в зависимости от выбранного шрифта и может быть установлен от 8 до 18.

Замечание

Размер шрифта также можно установить в пункте Font.

Font (шрифт)

Пункт Font позволяет изменять характеристики шрифта для выбранного элемента диаграммы. Этот пункт активизирует окно выбора шрифта в котором можно задать имя (Helvetica, Times, и т.д.), стиль шрифта, тип выделения, размер, специальные эффекты, например, подчеркивание и цвет.

Line Color (цвет линии)

Пункт Line Color позволяет изменять цвет линий, обрамляющих элемент диаграммы, а также любых линий на диаграмме Rational Rose. При выборе этого пункта активизируется окно выбора цвета с возможностью как выбора цвета из предлагаемой палитры, так и определения своего собственного.

Fill Color (цвет заливки)

Пункт Fill Color позволяет изменять цвет заливки диаграммы. При этом активизируется окно выбора цвета заливки, аналогичное окну

выбора цвета линий.

Use Fill Color (использовать заливку)

Пункт Use Fill Color позволяет включать и выключать заливку диаграммы цветом для выбранного элемента диаграммы без изменения цвета заливки. При включенной заливке на данном пункте меню появляется отметка.

Automatic Resize (автоматическое изменение размера)

Пункт Automatic Resize позволяет устанавливать автоматическое изменение размера элемента диаграммы при вводе надписей, если надписи превышают текущий размер элемента.

Эта установка влияет только на выделенный элемент и не может быть применена к пакетам. По умолчанию настройка включена для всех элементов.

Совет

Для того чтобы изменить эту настройку для вновь создаваемых элементов, необходимо проделать следующее: Menu:Tools=>Options=>Diagram=>Automatic resizing=Off или установить в файле rose.ini значение параметра AutoResizeIcons=No.

Stereotype Display(показ стереотипов)

Пункт Stereotype Display позволяет показывать или скрывать стереотип для объекта, а также устанавливать тип показа стереотипа.

Замечание

Подробнее о стереотипах мы поговорим в главе 11.

Stereotype Label (метка стереотипа)

Пункт Stereotype Label позволяет отображать метки установленных стерео типов, которые обычно показываются между знаками <<>>.

Show Visibility (показать область видимости)

Пункт Show Visibility позволяет включать или выключать изображение области видимости атрибутов и операторов на элементе класса диаграммы.

Замечание

Подробнее об области видимости см. в главе 9.

Show Compartment Stereotypes (показать внутренние стереотипы)

Пункт Show Compartment Stereotypes позволяет включать или

выключать отображение стереотипов для операций или атрибутов, если они определены для этих операций или атрибутов. Этот пункт действует на все выделенные на диаграмме классы.

Show Operation Signature (показать сигнатуру операций)

Пункт Show Operation Signature позволяет включать или выключать отображение сигнатуры операций. В сигнатуру операций входят параметры и возвращаемое значение.

Show All Attributes (показать все атрибуты)

Пункт Show All Attributes позволяет включать или выключать отображение атрибутов класса и удобен в случае, когда необходимо скрыть или показать сразу все атрибуты для выбранного класса.

Show All Operations (показать все операции)

Пункт Show All Operations аналогичен предыдущему, но действует по отношению к операциям классов.

Show All Columns (показать все колонки)

Пункт Show All Attributes позволяет включать или выключать отображение колонок при использовании модели базы данных.

Замечание

Моделирование базы данных при помощи Rational Rose мы рассмотрим в главе 21 книги.

Show All Triggers (показать все триггеры)

Пункт Show All Triggers позволяет включать или выключать отображение триггеров (хранимых процедур) при использовании объекта данных.

Suppress Attributes (подавить вывод атрибутов)

Пункт Suppress Attributes позволяет скрыть атрибуты класса таким образом, что не будет показана даже разделительная линия между атрибутами и операциями.

Замечание

При установленном на данном пункте маркере команда Show All Attributes не действует.

Suppress Operations (подавить вывод операций)

Пункт Suppress Operations аналогичен предыдущему, но действует по отношению к операциям классов.

Suppress Columns (подавить отображение колонок)

Пункт Suppress Columns скрывает показ всех колонок при использовании модели базы данных.

Suppress Triggers (подавить отображение триггеров)

Пункт Suppress Triggers скрывает показ всех триггеров при использовании модели базы данных.

Line Style (стиль линий)

Пункт Line Style позволяет изменять стиль выделенной линии следующим образом:

Rectilinear (прямолинейный) - линия будет ломаной, в которой отрезки располагаются только вертикально и горизонтально;

Oblique (наклонный) - наклонная линия, соединяющая две диаграммы;

Toggle (переключатель) - промежуточное положение между двумя предыдущими. Ломаная линия с наклонными отрезками. Горячие клавиши - Ctrl+Shift+L.

Layout Diagram (расположение диаграммы)

Пункт Layout Diagram изменяет расположение элементов оптимальным, по мнению Rational Rose, образом. Элементы располагаются в несколько рядов начиная с самого верхнего по иерархии, например, с главного класса.

Autosize All (настроить размер всех)

Пункт Autosize All позволяет увеличить или уменьшить каждый элемент в текущей диаграмме таким образом, чтобы имя, атрибуты, операции или другие элементы диаграммы полностью уместились в каждом элементе.

Этот пункт удобно использовать в случае, если на диаграмме есть элементы, для которых не установлена опция auto resize (автоматическая настройка размера), и необходимо настроить размер сразу всех таких элементов.

Layout Selected Shapes (расположить выделенные элементы)

Пункт позволяет расположить оптимальным образом не все элементы диаграммы, а только выделенные, что достаточно удобно если нужно перегруппировать, например, элементы наследуемые из одного класса.

Browse (просмотр)

Пункт Browse позволяет активизировать, создавать, переименовывать или удалять диаграммы, переключаться между

диаграммами и окнами спецификаций (рис. 2.13).

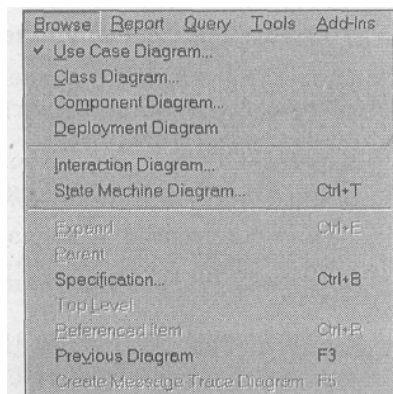


Рис. 2.13 Меню Browse

Блок активизации диаграмм

Основной блок меню Browse позволяет переключаться между диаграммами посредством выбора пункта с названием необходимой диаграммы. При этом если выбранной диаграммы в модели еще не существует, то будет предложено ее создать. Блок состоит из следующих пунктов:

Use Case Diagram активизирует диаграмму сценариев;

Class Diagram активизирует диаграмму классов;

Component Diagram активизирует диаграмму компонентов;

Deployment Diagram активизирует диаграмму топологии;

Interaction Diagram активизирует диаграмму взаимодействия;

State Machine Diagram активизирует диаграмму состояний. Горячие клавиши - Ctrl+T.

Expand (расширить)

Пункт Expand позволяет показать главную диаграмму для каждого выделенного логического пакета или компонента. Горячие клавиши - Ctrl+E.

Parent (родительский)

Пункт Parent позволяет показать компоненты или логические пакеты, которые содержат выделенный компонент или логический пакеты, то есть являются родительскими по отношению к выделенному элементу.

Specification (спецификации)

Пункт Specification позволяет открыть окно описания компонента (аналогично такому же пункту меню в контекстном меню). Окна спецификаций можно открывать для каждого элемента, то есть в каждый момент может быть открыто несколько окон спецификаций для разных элементов. Если для выбранного элемента окно спецификаций уже открыто, то произойдет переключение в это окно.

Top Level (верхний уровень)

Пункт Top Level позволяет показать диаграмму самого верхнего уровня. Это может быть верхняя диаграмма классов, верхняя диаграмма компонентов или диаграмма топологии.

Referenced Item (ссылающиеся элементы)

Пункт Referenced Item позволяет показать диаграммы или компоненты, имеющие связи с выделенным. При помощи данного пункта вы можете просмотреть:

- диаграммы классов, реализацией которых является выбранный объект;

- диаграммы, в которых определен выбранный класс, use case или пакет;

- спецификации операций для сообщений, которые связаны с этими операциями.

Если выбранный элемент является объектом, то при помощи данной команды будет найден его родительский класс и активизирована диаграмма, в которой данный класс появляется. Если найденный класс включен в несколько диаграмм, то будет активизирована диаграмма, в которой этот класс создан.

Если выбранный элемент является классом, созданным в другом логическом пакете, то этот пункт активизирует диаграмму логического пакета, в котором класс появляется или в котором этот класс создан. Горячие клавиши -Ctrl+R.

Previous Diagram (предыдущая диаграмма)

Пункт Previous Diagram позволяет быстро перейти на предыдущую диаграмму. Этот пункт применяется для последовательного переключения между двумя диаграммами. При этом та диаграмма, которая была текущей, запоминается как «предыдущая», и происходит активизация диаграммы, которая была активизирована перед текущей. Таким образом, переключение происходит только между двумя последними диаграммами. Горячая клавиша - F3.

Create Message Trace Diagram (создать диаграмму трассировки сообщений)

Пункт Create Message Trace Diagram позволяет быстро создать диаграмму Sequence по уже готовой Collaboration диаграмме. Если такая диаграмма уже создана, то данный пункт меняет свое название на Go to Sequence Diagram (перейти на диаграмму Sequence), а при нахождении в диаграмме Sequence, меняет свое название на Go to Collaboration Diagram (перейти на диаграмму Collaboration). Горячая клавиша - F5.

Report (отчет)

Данное меню позволяет создавать различные отчеты и проверять

правильность диаграммы (рис. 2.14).

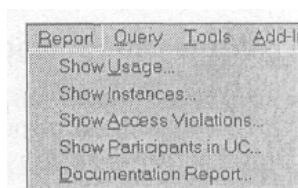


Рис. 2.14 Меню Report

Show Usage (показать использование)

Пункт Show Usage позволяет активизировать диаграммы или компоненты, в которых используется выбранный элемент. При выборе этого пункта активизируется диалоговое окно, показанное на рис. 2.15.

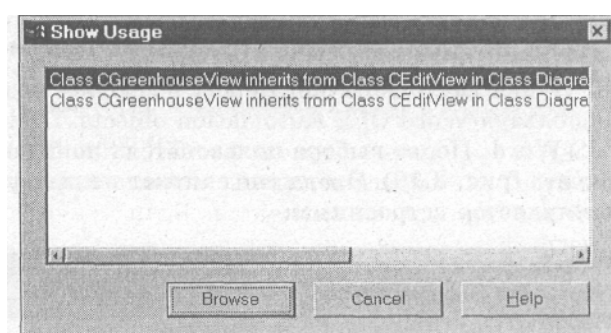


Рис. 2.15 Окно Show Usage

При помощи этого окна пользователю предоставляется возможность выбрать нужную диаграмму из списка, в котором представлены диаграммы, использующие текущий элемент. После выбора или нажатия кнопки Browse выбранная диаграмма становится текущей.

Show Instances (показать реализацию)

Пункт Show Instances позволяет получить список всех Collaboration диаграмм, в которых используются экземпляры данного класса. При выборе этого пункта активизируется диалоговое окно, аналогичное окну, описанному в предыдущем пункте, но только с диаграммами Collaboration.

Show Access Violation (показать нарушения доступа)

Пункт Show Access Violation позволяет показать список всех нарушений доступа между пакетами на диаграммах классов. Ошибки доступа случаются, когда класс из одного пакета связан с классом из другого пакета, однако, связи между пакетами не установлены. Также нарушение доступа может произойти, когда пакет, который экспортирует элементы для создания экземпляров классов, связан с классом, находящимся в другом пакете. Все ссылки на реализации классов в различных пакетах интерпретируются как нарушения.

Show Participants in UC (показать участников Use Case)

Пункт Show Participants in UC позволяет показать всех участников на диаграмме Use Case. Участниками в данном типе диаграмм могут быть классы и операции, включенные в любые диаграммы, используемые в Use Case.

Show Unresolved Objects (показать несвязанные объекты)

Пункт Show Unresolved Objects позволяет показать список всех объектов, для которых не определены классы. Этот пункт доступен только в Collaboration или Interaction диаграммах.

Show Unresolved Messages (показать несвязанные сообщения)

Пункт Show Unresolved Messages показывает несвязанные сообщения для выбранного элемента на диаграммах Collaboration или Sequence. Несвязанными называются сообщения, которые не прикреплены ни к одной операции. Это может произойти при удалении операции без удаления сообщения.

Documentation Report (создание документации)

Пункт Documentation Report позволяет создавать словарь данных модели, используя Word OLE Automation objects. При этом отчет создается в формате MS Word. После выбора пользователь попадает в окно настройки параметров отчета (рис. 2.16). Этот пункт может не присутствовать в меню, поскольку он не является встроенным.

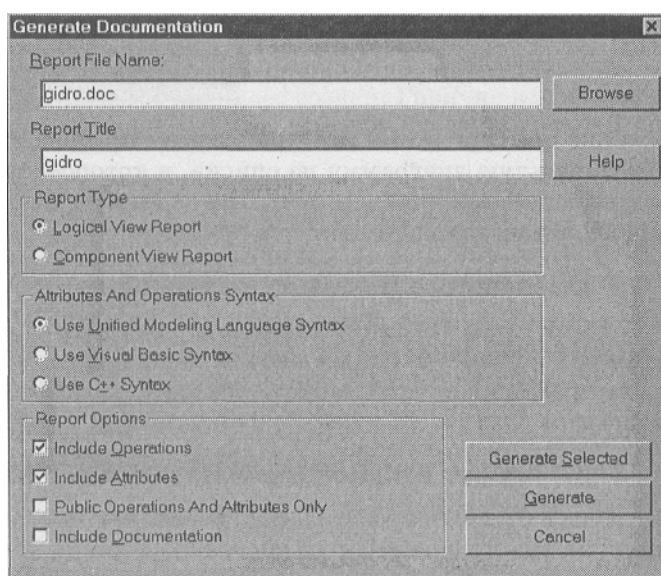


Рис. 2.16 Настройка Documentation Report

Здесь пользователю предлагается настроить Report File Name (имя файла отчета), Report Title (заголовок отчета).

Report Type предоставляет пользователю возможность выбора типа.

Это может быть Logical View (логическое представление), при этом выводятся данные по диаграммам, а также Component View (представление компонентов), в этом случае выводятся данные о компонентах.

Пункт Attributes And Operations Syntax (синтаксис атрибутов и операций) позволяет задавать язык, в котором будут представлены данные отчета. Здесь дается выбор из трех вариантов:

Use Unified Modeling Language Syntax - использовать синтаксис языка UML;

Use Visual Basic Syntax - использовать синтаксис языка Visual Basic;

Use C++ Syntax - использовать синтаксис языка C++.

Report Options (установки отчета) позволяют задать дополнительные ограничения отчета.

Include Operations - включить в отчет информацию об операциях;

Include Attributes - включить в отчет информацию об атрибутах;

Public Operations And Attributes Only - включить операции и атрибуты, которые обозначены как public (общие);

Include Documentation - включить в отчет документацию.

Query (запрос)

Меню Query позволяет производить различные манипуляции с объектами диаграмм: скрывать, добавлять, фильтровать (рис. 2.17).

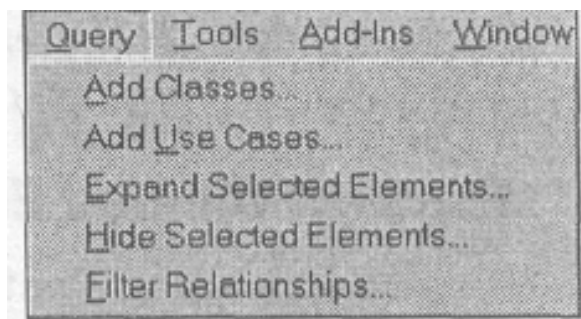


Рис. 2.17 Меню Query

Add Classes (добавить классы)

Пункт Add Classes активизирует диалоговое окно добавления классов, представленное на рис. 2.18, при помощи которого пользователь получает возможность копировать классы из одной диаграммы в другую.

Совет

Для управления показом конкретных связей на диаграмме используйте команду Filter Relationships.

Если добавляются классы с установленными связями, то эти связи будут показаны на диаграмме, как только классы будут перенесены.

Совет

Для добавления на диаграмму классов, имеющих связи с выбранным, используйте команду Expand Selected Elements.

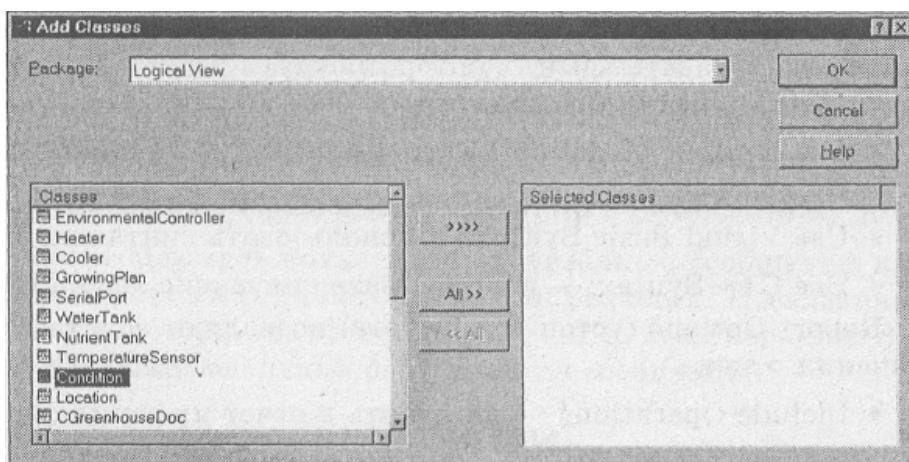


Рис. 2.18 Окно Add Classes

Используйте этот пункт для добавления классов, имена которых известны.

Часто необходимо перенести класс, находящийся в другой диаграмме, в текущую. Для этого Rational Rose предоставляет возможность буксировки необходимого класса мышкой из окна Browse в конкретную диаграмму, однако, это не всегда удобно и, более того, не всегда возможно. Таким образом, использование пункта Add Classes позволяет легко добавлять классы в диаграмму без утомительного перемещения по окнам.

Add Use Cases (добавить Use Cases)

Пункт Add Use Cases позволяет добавлять элементы Use Case из других диаграмм в текущую и по своим возможностям аналогичен предыдущему пункту.

Expand Selected Elements (показать выделенные элементы)

Пункт Expand Selected Elements активизирует диалоговое окно, показанное на рис. 2.19. Это окно позволяет управлять отображением элементов на диаграмме, основываясь на связях этих элементов. Все установки, сделанные в окне, запоминаются и при следующей активизации выставляются в последнее состояние.

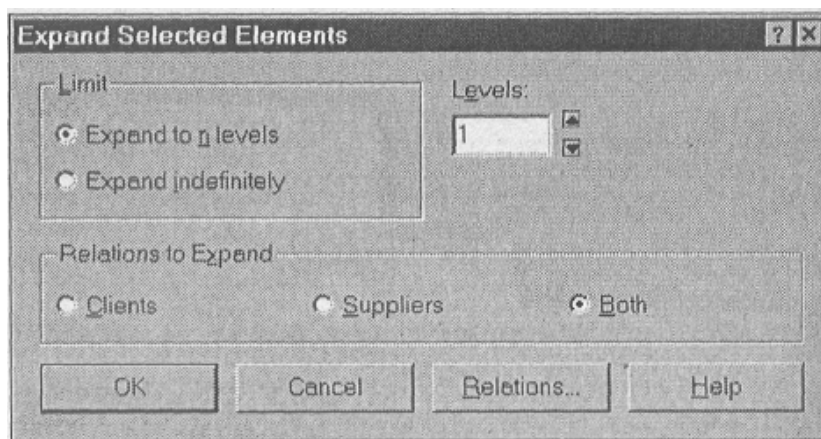


Рис. 2.19 Окно Expand Selected Elements

Замечание

Если установлен флажок Expand indefinitely (без учета уровней), то номер уровня (Level) нельзя изменить.

Используйте пункт Expand Selected Elements для показа дополнительных элементов на диаграмме, которые связаны с текущим элементом, например, для показа всех классов, которые наследуются из текущего, классов, из которых наследуется текущий, или классов использующих выбранный.

Hide Selected Elements (скрыть выделенные элементы)

Пункт Hide Selected Elements активизирует диалоговое окно, показанное на рис. 2.20.

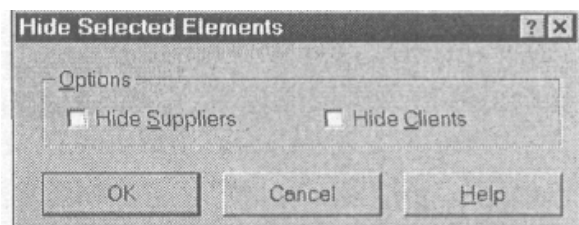


Рис. 2.20 Окно Hide Selected Elements

При помощи этого окна пользователь может скрыть элементы, представленные на диаграмме: классы, use case, элементы диаграммы состояний или активности. При этом элементы только скрываются, но не удаляются из модели.

По умолчанию данная команда скрывает только выбранные элементы, однако, при установке переключателей Hide Suppliers или Hide Clients будут также скрыты элементы, для которых выделенный является клиентом, или те, которые для выделенного являются клиентами соответственно.

Можно скрыть и вновь показать значки на диаграмме, однако, есть несколько ограничений, которые, возможно, будут исправлены в следующей версии Rational Rose.

На мой взгляд это досадные недоработки, которые мешают

пользоваться этими режимами на полную мощность. При скрывании, а затем показе теряется местоположение значка на диаграмме. Вы должны быть готовы, что все вновь активизированные значки будут находиться в «творческом беспорядке», и вам придется расставлять их заново. Наверняка это связано с тем, что при скрывании теряются координаты расположения.

Но что еще более неприятно, при скрывании теряется информация о Swim-lane, и при активизации значки показываются в текущей колонке Swimlane.

Замечание

Работа с элементами Swimlane и поддиаграммами рассматривается в главе 7.

Последнее меня больше всего удручает, поэтому я не стал пользоваться этим режимом постоянно. И на мой взгляд, лучший вариант - не скрывание и показ значков, а создание Sub диаграмм, т.е. диаграмм нижнего уровня, которые фактически скрыты постоянно, но в них всегда можно перейти при помощи контекстного меню.

Filter Relationships (фильтр связей)

Пункт Filter Relationships активизирует диалоговое окно установки фильтра связей, представленное на рис. 2.21. Это окно позволяет скрыть указанные связи, загромождающие диаграмму и мешающие ее восприятию.

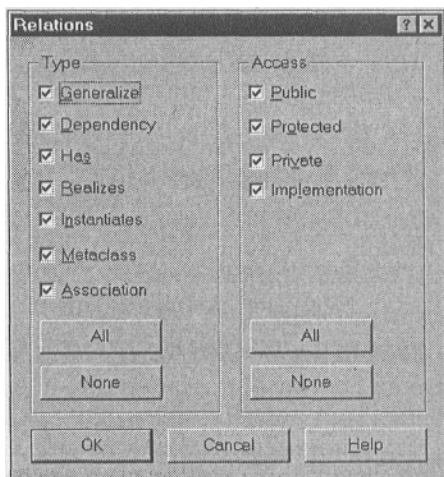


Рис. 2.21 Окно фильтра связей

Замечание

Установленный фильтр оказывают влияние только на уже установленные связи. При добавлении в диаграмму новых связей, установленный до их добавления фильтр на них не действует.

Все установки окна запоминаются и при последующем вызове показываются в виде последней редакции.

Замечание

При выборе этого пункта для Statechart или Activity диалоговое окно не высвечивается.

Пункт Filter Relationships позволяет скрыть и снова отобразить конкретные типы связей для того чтобы получить возможность охватить взглядом связи и взаимодействие отдельных элементов.

Tools (инструменты)

Меню Tools предоставляет доступ к различным инструментам и настройкам. Оно достаточно обширно и зависит от установленных Add Ins модулей, поэтому будем изучать отдельные инструменты непосредственно в момент работы с каждым из них. В Части 5 книги будут рассмотрены некоторые дополнительные модули, которые не были использованы для нашей задачи, но важны для общего понимания принципов работы в Rational Rose.

Add-Ins (подключаемые модули)

Меню Add-Ins состоит из одного пункта Add-In Manager, который позволяет устанавливать или удалять дополнительные инструменты, модули и элементы Rational Rose для поддержки различных языков программирования.

После установки можно проверить, поддержка каких языков программирования сейчас активна, и установить или удалить поддержку необходимого языка, как показано на рис. 2.22.

Также Add-In Manager позволяет установить дополнительные модули, которые позволяют организовать работу с другими программными продуктами как компании Rational, так и сторонних производителей. С версией Rational Rose 2002 поставляются более двадцати подключаемых модулей, рассмотрение возможностей которых достойно отдельной книги. Некоторые возможности Add-In мы рассмотрим в части 5 книги.

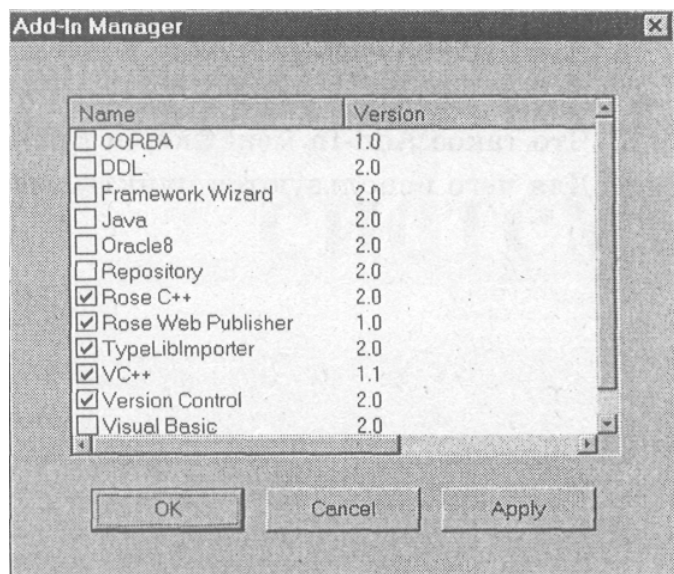


Рис. 2.22 Окно менеджера Add-Ins

При установке дополнительных модулей нужно помнить, что некоторые Add-Ins нельзя установить, если на компьютере нет соответствующих программ. Например, нельзя установить Add-Ins VC+, если на компьютере не установлен Visual C++ 6.0.

Замечание

В случае, если данный пункт меню недоступен, необходимо установить менеджер Add-Ins с установочного компакт диска.

Windows (окна)

Меню Windows позволяет контролировать размещение окон диаграмм и спецификаций на рабочем столе Rational Rose. Используйте данный пункт для автоматической расстановки окон, которые были скрыты или свернуты.

Help (помощь)

Меню Help позволяет активизировать окно встроенной документации Rational Rose. В случае если окно помощи уже активизировано, но скрыто под другими окнами, то при помощи этого меню оно будет выведено на передний план.

Вопросы для повторения

1. Какова структура меню программы?
2. Как создать новую модель и сохранить ее?
3. Как изменить формат диаграммы, шрифт, цвет?
4. Какие доступны возможности изменения вида диаграмм?
5. Как настроить рабочий стол Rational Rose?
6. Как переключаться между диаграммами?
7. Какие доступны отчеты и для чего они используются?
8. Что такое Add-In менеджер и для чего он используется?
9. Для чего используются пункты меню Tools, Windows, Help?

Часть 2

Проектирование гидропонной системы

Глава 3. Объектно-ориентированная парадигма

Общая концепция

Перед тем как вплотную приступить к проектированию системы при помощи Rational Rose, необходимо определиться с терминами и общей концепцией, или, как часто говорят, - парадигмой.

Тот читатель, который достаточно хорошо разбирается в объектно-ориентированном проектировании и программировании, может пропустить этот раздел, так как ничего нового относительно классиков здесь сказано не будет. Однако я бы не советовал этого делать по той простой причине, что здесь будут обсуждаться термины, которыми мы будем пользоваться на всем протяжении работы с Rational Rose.

Здесь мы кратко рассмотрим вопросы проектирования систем, так как создание программных продуктов на объектно-ориентированном языке, таком как, например, C++, требует объектно-ориентированного подхода как к проектированию, так и к программированию.

Стили программирования

При создании программ можно пользоваться различными стилями программирования. Их выделяют всего пять:

- процедурно-ориентированный направленный на представление программы как множества поочередно вызываемых процедур;

- объектно-ориентированный - направленный на представление программы как набора взаимодействующих объектов;

- логико-ориентированный направленный на выполнение целей, выраженных в терминах исчисления предикатов;

- ориентированный на правила – выполнение правил «если - то»;

- ориентированный на ограничения.

Как утверждает Г. Буч, невозможно признать какой-либо стиль программирования наилучшим во всех областях практического применения, однако, объектно-ориентированный стиль наиболее приемлем для широчайшего круга задач.

Отличительные особенности объектно-ориентированного подхода

Главное отличие процедурно-ориентированного программирования от объектно-ориентированного в том, что при использовании первого программист создает процедуры, которые вызывают друг друга для выполнения поставленных задач и обработки данных.

При использовании объектно-ориентированного стиля программист создает программные объекты и наделяет их определенным поведением,

реакцией на изменения внешних условий. Такие объекты взаимодействуют между собой, выполняют определенные задачи, принимают, обрабатывают и передают данные.

С объектно-ориентированным программированием тесно связано объектно-ориентированное проектирование. Если программирование направлено на правильное и эффективное использование конкретных языков, то проектирование направлено на правильное и эффективное структурирование сложных систем. При этом объектно-ориентированное проектирование подразумевает уже на этапе замысла системы анализ ее как набора взаимодействующих объектов.

Концептуальная база объектно-ориентированного стиля

Каждый стиль программирования имеет свою концептуальную базу. Для объектно-ориентированного стиля концептуальная база - объектная модель, создание которой требует особого объектно-ориентированного мышления.

Объектная модель имеет четыре главных свойства. Это [1]:

абстрагирование – выделение существенных характеристик объекта, отличающих его от других видов объектов;

инкапсуляция – скрывание внутренней реализации объекта за предоставляемым этим объектом интерфейсом;

модульность – способность системы быть разложенной на внутренние сильно или слабо связанные между собой модули; и иерархия – упорядочивание абстракций по уровням;

иерархия – упорядочивание абстракций и расположение их по уровням.

Эти свойства являются главными, и при отсутствии любого из них модель не будет объектно-ориентированной. Также существуют и три дополнительных свойства, которые полезны в объектной модели, но без которых можно обойтись. Это:

- типизация – создание объектов на основе шаблонов определенного типа;

- параллелизм - способность системы обрабатывать несколько сообщений

или задач параллельно;

- сохраняемость – способность хранить не только данные, но и объекты

в промежутке между отдельными запусками системы.

Классы и объекты

Для того чтобы создать объектно-ориентированную программу, необходимо создать некоторый набор объектов с определенным поведением, определить их взаимосвязи. В свою очередь, для создания объектов необходимо создать их описание, называемое классом в терминах C++.

Класс – это шаблон, на основе которого создаются объекты. Нельзя путать класс и объект. Класс – это лишь матрица, на основе которой

создаются объекты.

Класс определенного типа может быть только один, а объектов в программе может быть сколько угодно (точнее, на сколько хватит ресурсов системы). После создания класса проверить правильность его описания можно только после создания объектов на его основе. Когда объекты начинают работать и взаимодействовать, только тогда можно оценить точность поведения объекта, описанного посредством класса.

Если взять пример телефонного аппарата, то его электрическая схема сродни классу, а сам аппарат сродни объекту. На основе одной электронной схемы можно сделать сколько угодно аппаратов и все они будут работать и выглядеть одинаково, конечно, если на заводе не допущен брак. В этом отличие программирования от реального производства. При создании программы за точностью изготовления объекта следит языковой компилятор, а программист должен сосредоточиться лишь на правильном описании класса.

Свойства классов

Важнейшими свойствами классов считаются: инкапсуляция, наследование и полиморфизм.

Инкапсуляция

Инкапсуляция аналогична свойствам объектно-ориентированной модели и подразумевает скрытие ненужных деталей реализации класса. В самом классе могут храниться данные и методы их обработки, которые доступны только посредством предоставляемого классом интерфейса и защищены от нежелательного воздействия извне. Для использования класса нет необходимости знать его внутреннее устройство, как для того, чтобы воспользоваться телефоном, нет необходимости знать его электронную схему. Нужно лишь знать, как набрать номер или ответить на звонок, то есть знать описание внешнего интерфейса.

Наследование

Наследование одно из самых привлекательных свойств классов. Это свойство позволяет создавать на основе одного или нескольких родительских классов дочерние классы (подклассы) со свойствами родительского и дополнительными возможностями.

Возможность наследования свойств позволяет программисту значительно сократить объем ручного кодирования и позволяет изменить поведение всех дочерних объектов, изменив поведение родительского класса.

Если в реальном производстве в схеме уже выпущенных телефонных аппаратов будет найдена ошибка, то вся партия будет отправлена на свалку. В программировании же, исправив найденную ошибку в родительском классе, мы после запуска программы немедленно заставляем правильно работать и все дочерние классы.

Полиморфизм

Полиморфизм – возможность объектов, создаваемых на основе классов, изменять свою реакцию на одни и те же воздействия при различных внешних условиях.

Атрибуты и методы классов

Классы имеют атрибуты и методы. В приложении к исходному коду программ атрибуты, часто называемые свойствами, – это переменные, описанные в теле класса, которые могут быть как скрыты от внешнего воздействия (в этом случае изменение свойств производится посредством доступных извне методов), так и доступны для изменения. Методы – это функции, определенные в теле класса, которые могут быть доступны или скрыты от внешних программ.

В терминах Rational Rose методы называют операциями, поэтому в дальнейшем будем пользоваться обоими терминами. Свойства и методы могут иметь различные типы доступа и реализации, что в Rational Rose отражается специальными обозначениями, которые мы разберем непосредственно при создании свойств и методов.

Вопросы для повторения

1. Какие стили программирования используются для создания программ?
2. Какие отличительные особенности объектно-ориентированного подхода?
3. Какова концепция объектно-ориентированного стиля?
4. Какие отличия между классами и объектами?
5. Какие вы знаете важнейшие свойства классов?
6. Что такое атрибуты класса?
7. Что такое методы класса?

Глава 4. Определения требований к системе при помощи Use Case

Специфика создания программных систем

Методике создания сложных программных комплексов посвящены многие тома научных исследований. Я не буду вдаваться в тонкости этого процесса, укажу только, что для создания системы необходимо погрузиться в предметную область, то есть необходимо четко представить работу всего того механизма, с которым будет иметь дело программная система. Не секрет, что программисты всеми правдами и неправдами стараются отойти от морально тяжелого процесса общения с заказчиками системы. Однако мечта руководителя программного проекта – получить от заказчика все необходимые спецификации и описания алгоритмов работы, чтобы потом передать их программисту, пока так и остается мечтой.

Заказчики часто сами с трудом представляют работу будущей системы и не могут ответить на вопросы: какие необходимо автоматизировать функции, какие алгоритмы будут задействованы? Не секрет, что пользователи идеализируют будущую систему, считая, что в ней воплотится все, о чем они мечтают (часто не сообщая об этих мечтах разработчику) и даже немножко больше.

Задача проектировщика системы - «вытащить» из заказчика всю необходимую информацию еще на этапе проектирования, до того, как в создание кода вложены значительные ресурсы.

И в этом не последнюю роль может сыграть Rational Rose. Достаточно создавать диаграммы с описанием поведения объектов системы и последовательно уточнять их у заказчика и, как говорил герой одной известной сказки, «золотой ключик у нас в кармане».

В нашем случае не придется обсуждать с заказчиком детали проекта, так как все описания системы уже есть, однако мы шаг за шагом будем детализировать описание системы, как если бы вы обсуждали эти детали с будущими пользователями системы.

Замечание

В компании Rational разработано программное средство для управления требованиями – Rational RequisitePro, где есть возможность записать требование и сопоставить ему диаграмму Use Case.

Описание задачи

Представим себе, что вам необходимо разработать программное обеспечение для тепличного хозяйства, использующего гидропонику. Растения в таком хозяйстве выращиваются без грунта на специальном питательном растворе. Для нормального роста и созревания урожая необходимо соблюдение режима выращивания. Управление различными параметрами тепличного хозяйства достаточно трудоемкая задача для человека, ведь следить за ними необходимо круглосуточно. Поэтому для

соблюдения режима выращивания конкретных растений управление режимом парниковой установки осуществляется при помощи автоматических устройств.

На режим выращивания влияют различные внешние показатели, которые необходимо поддерживать в заданном диапазоне. Это могут быть температура, влажность, освещение, показатели кислотности почвы и другие факторы, которые в нашем случае не рассматриваются.

Для измерения этих показателей используются датчики, с которых информация поступает в систему. Датчики представляют собой «глаза» и «уши» системы, без них ввод информации придется осуществлять при помощи человека-оператора, тогда ни о какой автоматизации не может быть и речи. Представьте себе оператора, который раз в десять минут обходит теплицу и фиксирует показания термометров в журнале.

Для такой системы обязательно наличие исполнительных устройств, таких как нагреватели, осветители, вентиляторы, контроллеры внесения удобрений. Эти устройства - «руки» системы, при помощи которых осуществляется изменение внешних условий, таких как температура или кислотность почвы.

Изменение условий осуществляется на основе плана выращивания растений, в котором хранится информация о моментах времени и необходимых действиях в эти моменты. Так, например, для некоторого растения необходимо на 15-е сутки роста поддержание температуры 25°C, из них 14 часов с освещением, а затем понижение температуры до 18°C в остальное время суток.

Для контроля за происходящими процессами необходимо отображать текущее состояние системы с возможностью воздействия оператора и протоколировать действия в журнале.

Теперь, когда мы в общих чертах уяснили, что должна делать система, можно при помощи диаграммы Use Case (сценариев поведения) определить объекты системы и действия, которые эти объекты должны производить.

Назначение

Как мы уже обсуждали в главе 1, данный тип диаграммы предназначен для создания списка операций, которые выполняет система и часто его называют диаграммой функций, потому что на основе набора таких диаграмм строится набор функций, выполняемых системой и описываются сценарии поведения объектов, которые взаимодействуют с системой.

Подготовка к работе с Use Case

Запустите Rational Rose и создайте новую пустую модель. Для этого нажмите кнопку Cancel в окне мастера создания модели (рис. 2.2). Это будет наша рабочая модель, в которой и должны будут отражены все нюансы будущей гидропонной системы. Перейдите на диаграмму Use Case, как показано на рис. 4.1,

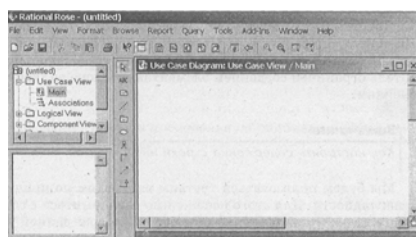


Рис. 4.1 Активизация Use Case диаграммы

Создание новых элементов

Rational Rose предоставляет несколько способов создания новых элементов в модели.

Вы можете создавать элементы, пользуясь контекстным меню, как показано на рис. 4.2.

Вы можете создать элементы при помощи Menu:Tools=>Create.

Вы можете создать элементы при помощи строки инструментов.

В первом случае элемент создается непосредственно в модели, но его значок не включается ни в одну диаграмму. После создания элемента таким образом необходимо поместить его на выбранную диаграмму.

Во втором и третьем случае вместе с созданием элемента его значок помещается на текущую диаграмму автоматически, что в нашем случае исключает один промежуточный шаг.

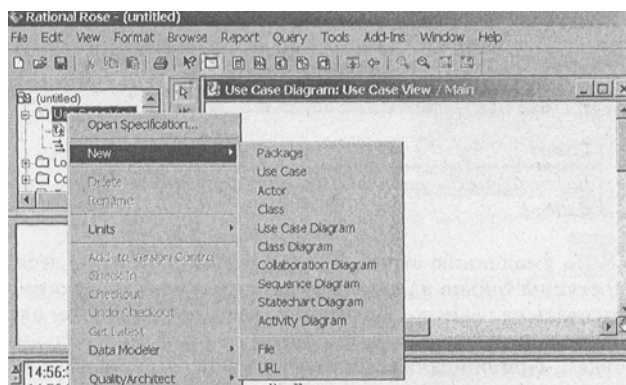


Рис. 4.2 Создание элемента при помощи контекстного меню

При создании элементов посредством меню. Tools программа предоставляет возможность создавать все элементы, которые можно включить в текущую диаграмму, тогда как при создании средствами строки инструментов пользователь ограничен созданием элементов согласно включенным в данную строку значкам.

Замечание

Как настроить содержание строки инструментов, описано в главе 2.

Мы будем пользоваться третьим вариантом по причине большей простоты и наглядности. Для этого необходимо ознакомиться с содержанием строки инструментов, установленной по умолчанию для

данной диаграммы. Для моделирования бизнес-процессов Rational Rose предоставляет дополнительные элементы Use Case, которые можно активизировать при помощи режима настройки инструментов. Но для создания гидропонной системы достаточно значков, установленных по умолчанию.

Строка инструментов

После активизации диаграммы Use Case строка инструментов диаграммы приобретает вид, представленный на рис. 4.3.



Рис. 4.3 Строка инструментов диаграммы Use Case

Совет

Для того чтобы увеличить размер значков, выберите Toolbox=>RClick=>Use Large Buttons.

По умолчанию строка инструментов состоит из десяти значков. Некоторые доступны только на данной диаграмме, но есть и такие, которые могут присутствовать на разных диаграммах, выполняя при этом одинаковые функции. Работу с такими значками мы разберем один раз и больше возвращаться к ней не будем. При описании инструментов будем пользоваться их названиями, которые «всплывают» при наведении на значок курсора мыши.

Selection Tool (инструмент выбора)



Основной инструмент, который позволяет выбирать элементы диаграммы, для того чтобы производить с ними дальнейшие действия. Если вы не создаете новый элемент, то этот инструмент активен. При создании нового элемента диаграммы необходимо выбрать нужный инструмент в строке инструментов, кнопка «залипает», а после создания необходимо опять перейти в режим Selection Tool.

Замечание

Переход в режим выбора происходит автоматически сразу после создания элемента, если не установлен флажок Menu:Tools=>Options=>Toolbars=>Lock selection.

Text Box (текст)



Данный инструмент позволяет создать произвольную надпись на диаграмме, не привязанную ни к какому элементу.

Замечание

Эта надпись не является полноценным элементом модели и не отображается в окне Browse, а используется как комментарий в конкретной диаграмме.

Для создания надписи необходимо нажать кнопку Text Box, при этом курсор примет вид вертикальной стрелки, и щелкнуть на том месте диаграммы, где необходимо создать надпись. В обозначенном квадратными точками в углах окне можно вводить надпись.

Совет

Для того чтобы изменить введенную надпись, нужно активизировать редактирование двойным щелчком мыши. Для того чтобы изменить шрифт или его размер, воспользуйтесь контекстным меню

Note (замечание)

Данный инструмент создает элемент замечания, позволяющий вписать в него принятые во время анализа решения. Заметки могут содержать простой текст, фрагменты кода или ссылки на другие документы. Обычно окно Note соединяют с другими элементами диаграммы при помощи инструмента Anchor Note, для того чтобы показать к какому элементу диаграммы относится замечание (рис. 4.5). В этом отличие от элемента Text Box, который располагается на диаграмме без присоединения к другим элементам.

Данный элемент не имеет ограничения на количество вводимых символов, и окно Note может быть растянуто, для того чтобы вместить необходимый текст. При активизации этого инструмента курсор принимает форму креста. Контекстное меню для значка Note позволяет кроме установки шрифта устанавливать цвет линий и заливки.

Замечание

Курсор принимает форму креста при создании элементов диаграммы, являющихся объектами, и форму стрелки при создании связей.

Note Anchor (якорь для замечания)



Данный инструмент позволяет соединить элемент Note с любым элементом на диаграмме, в том числе и с другим элементом Note.

Замечание

Нельзя соединить между собой два элемента Note Anchor.

Для того чтобы присоединить замечание к элементу диаграммы, необходимо выбрать инструмент Note Anchor, при этом курсор приобретает форму вертикальной стрелки, щелкнуть по значку Note и, не

отпуская кнопки мыши, «тянуть» линию до нужного значка, по достижению которого кнопку мыши отпустить.

Замечание

Аналогично происходит соединение при помощи других инструментов для установки связей.



Package (пакет)

Данный инструмент позволяет создавать пакеты, которые могут включать в себя группы элементов Use Case и в данной диаграмме может использоваться для определения более крупных сценариев поведения объектов с дальнейшей детализацией. Причем пакеты могут включать в себя другие пакеты, что позволяет создавать значительный уровень вложенности детализации.



Use Case (сценарии поведения)

Данный инструмент позволяет создавать простые формы сценариев поведения объектов системы. Это представление работы системы с точки зрения исполнителей (actors), то есть объектов, выполняющих в системе определенные функции.

Use Case могут отображать:

- образцы поведения для отдельных объектов системы;
- последовательность связанных транзакций, представляемых объектами или системой;
- получение некоторой информации объектами.

Создание Use Case необходимо для того, чтобы:

- формализовать требования к системе;
- организовать взаимодействие с будущими пользователями системы и экспертами предметной области;
- тестировать систему.

Замечание

Use case – лучший путь к тому, чтобы определить исполнителей системы и выполняемые ими задачи. За подробностями можно обратиться к книге [4].

Actor (актер)

Данный инструмент используется для создания действующих лиц в системе. На диаграмме Use Case значком actor часто обозначают пользователей системы, для того чтобы определить задачи, выполняемые пользователями и их взаимодействие.

Обычно значком Actor обозначают объект, который:

взаимодействует с системой или использует систему;
передает или принимает информацию в/из системы;
является внешним по отношению к системе.

Актор позволяют узнать:

кто пользуется системой;

кто отвечает за сопровождение системы;

внешнее аппаратное обеспечение, которое используется системой;

другие системы, которые должны взаимодействовать с данной системой.

Замечание

В последних версиях методологии разработки программного обеспечения Rational Unified Process, вместо понятия «исполнитель» рекомендуется использовать понятие «роль», поскольку каждый исполнитель может выполнять определенные роли и каждая роль может быть исполнена различными исполнителями.

Unidirectional Association (однонаправленная связь)

Данный инструмент позволяет обозначать связи между элементами. На диаграмме Use Case эти связи могут быть определены между use case и actor.

Замечание

Работу с инструментами Dependency of instantiates и Generalization мы рассмотрим в главе 12, так как в этой диаграмме они нам не понадобятся.

Создание диаграммы Use Case для гидропонной системы

После изучения основных инструментов диаграммы можно приступить к созданию сценариев поведения системы. Согласно постановке задачи существует некоторый план выращивания растения. Он должен быть введен в систему оператором. Для отражения этого процесса создадим новое действующее лицо (Actor) и присвоим ему имя «Оператор». Создадим новый значок use case и присвоим ему имя «создать план выращивания», после чего необходимо соединить эти значки ассоциативной связью, как показано на рис. 4.6.



Рис. 4.6 Диаграмма Use Case после добавления значка Оператор

Замечание

Название use case начинается с глагола и обозначает действие элемента actor.

Заметим, что план выращивания должен поступать в систему и обрабатываться. Также оператор должен иметь возможность просматривать протокол работы системы.

Создадим новый объект и назовем его Контроллер. Соединим его связью с элементом «создать план выращивания». Создадим новый значок «создать протокол», который соединим связью с Контроллером. Теперь у нас получился диаграмма, представленная на рис. 4.7.

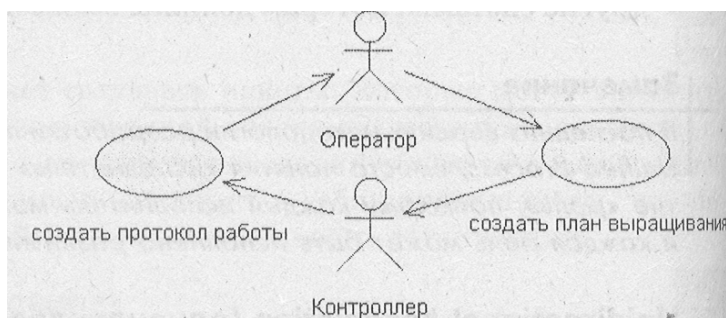


Рис. 4.7 Диаграмма Use Case после добавления значка Контроллер

Также Контроллер должен управлять исполнительными устройствами. Для отражения этого процесса создадим use case с именем «управлять устройствами» и новое действующее лицо Устройства.

Также необходимо создать новое действующее лицо Датчик и use case «измерить показатели среды». После соединения связями этих значков у вас должно получиться изображение, показанное на рис. 4.8.

Совет

Чтобы не потерять работу, сохраните проект нажатием Ctrl+S.

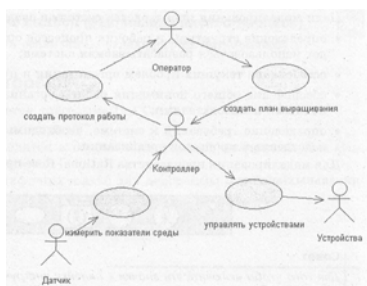


Рис. 4.8 Окончательный вид диаграммы Use Case

Таким образом, в окончательном варианте мы получили следующие требования к системе управления тепличным хозяйством:

- оператор должен иметь возможность задать план выращивания;
- оператор должен иметь возможность просматривать протокол работы системы;
- система должна получать информацию от датчиков;
- система должна иметь возможность управлять внешними устройствами на основе показателей датчиков и введенного плана

выращивания.

Как вы заметили, я не углублялся в детали поведения того или иного актера, так как написание эффективных сценариев поведения – это целая наука и здесь мы лишь попробовали свои силы при помощи инструмента Rational Rose.

Для такой небольшой программы, как система управлением тепличным хозяйством, создание диаграммы Use Case может показаться лишней работой, однако, для ознакомления с диаграммой Use Case это просто необходимо.

Использование диаграммы Use Case для моделирования производства

В последних версиях программы Rational Rose появились значки, позволяющие создавать модели производства. Конечно, до их появления можно было воспользоваться стандартными, но использование значков, специально предназначенных для моделирования производства, видится разработчикам программы более удобным.

Для нашей небольшой непроизводственной системы в создании такой модели нет необходимости, но при создании программного комплекса, который должен влиться в работу предприятия и позволить выполнять все текущие

и перспективные задачи это просто необходимо.

Цели моделирования производства состоят в следующем:

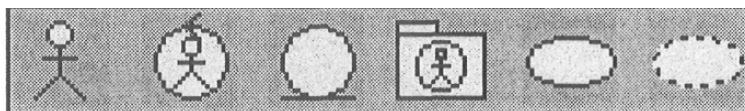
определение структуры и рабочих процессов организации, в которой будет использоваться разрабатываемая система;

осмысление текущих проблем организации и путей улучшения;

обеспечение общего понимания работы организации заказчиками и конечными пользователями;

определение требований к системе, необходимых для поддержки производственных процессов организации.

Для моделирования производства Rational Rose предоставляет шесть дополнительных значков.



Совет

Для того чтобы включить эти значки в линейку инструментов, необходимо из контекстного меню инструментов диаграммы Use Case выбрать **RClick=>Customize** или **Menu:Tools->Options=>Toolbars=>Customize Toolbars=>Use Case Diagram**

Отличительной особенностью значков, относящихся к моделированию производства, будет то, что они закрашены желтым цветом и имеют косую черту для выделения этих значков на черно-белой печати. Кратко перечислим назначение каждого из них.

Business actor (бизнес-актор)



Позволяет представить исполнителя бизнес-функций, связанного с работой системы, например, пользователя.

Business worker (бизнес-работник)



Позволяет представить работника внутри системы, который выполняет некоторые бизнес-функции, например, клерк.

Business entity (бизнес-сущность)



Позволяет представить пассивный бизнес-объект, который сам не инициирует взаимодействие, например, счет.

Organization Unit (организационное подразделение)



Позволяет представить пакет, который содержит другие бизнес-объекты

в том числе и организационные подразделения.

Business Use Case (бизнес-процесс)



Позволяет представить отдельный бизнес-процесс.

Business Use Case Realization (реализация бизнес-процесс)

Позволяет представить реализацию отдельного бизнес-процесса.

Вопросы для повторения

1. Для чего используется диаграмма Use Case?
2. Как создать новую диаграмму?
3. Какие значки находятся в строке инструментов диаграммы Use Case и каково их назначение?
4. Какие значки специфичны только для диаграммы Use Case?
5. Как при помощи диаграммы создать сценарий поведения?

Глава 5. Использование Deployment диаграммы для анализа устройств

Назначение

Диаграмма Deployment (топология) предназначена для анализа аппаратной части системы. При помощи данной диаграммы проектировщик может произвести анализ необходимой аппаратной конфигурации, на которой будут работать отдельные процессы системы, и описать их взаимодействие между собой и другими аппаратными устройствами.

Этот тип диаграмм также позволяет анализировать взаимодействие процессов, работающих на разных компьютерах сети.

Замечание

Для каждой модели такая диаграмма может быть только одна.

Этот тип диаграмм, по моему мнению, самый простой в Rational Rose, так

как в нем используются только два вида основных значков. Для

активизации диаграммы выберите значок:



При активизации Deployment диаграммы строка инструментов приобретает следующий вид (рис. 5.1).

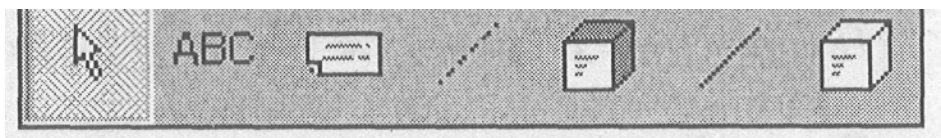


Рис. 5.1 Строка инструментов Deployment диаграммы

Кроме уже знакомых нам значков Selection Tool, Text Box, Note, Anchor Note to Item, выполняющих те же функции, что и в Use Case диаграмме, добавились значки для создания процессоров, устройств и соединений.

Processor (процессор)



Процессор - • это устройство, способное выполнять программы. Процессор обязательно должен иметь свое имя, которое, однако, никак не связано с другими диаграммами модели по причине того, что процессор обозначает не программное обеспечение, а аппаратуру.

Создадим для нашей системы новый процессор и назовем его «Компьютер». Посмотрим, какие возможности предоставляет в наше распоряжение программа из контекстного меню, активизируемого правой кнопкой мыши.

Замечание

В контекстном меню могут присутствовать дополнительные пункты, в зависимости от установленных Add-Ins.

Open Specification -- открытие окна спецификаций элемента.

Select in Browser позволяет быстро найти элемент в окне Browser.

Show Sheduling позволяет включать или выключать показ порядка вы
полнения процессов.

Show Processes позволяет включать или выключать показ процессов на
диаграмме.

Замечание

Подробнее о процессах и их планировании - в разделе, описывающем спецификации процессора.

Stereotype Display позволяет изменить показ стереотипа, но обычно для
данного типа диаграмм стереотипы не используются.

Format позволяет изменять формат элемента.

Замечание

Подробнее о стереотипах читайте в главе 11.

Спецификации процессора

При выборе пункта контекстного меню Specification активизируется диалоговое окно установки спецификаций процессора, состоящее из двух вкладок, При активизации пользователь попадает во вкладку General (главная), показанную на рис. 5.3.

На вкладке General предоставляется возможность изменить название элемента и ввести дополнительное описание для элемента в поле Documentation, Кроме стандартных кнопок сохранения и отказа, внизу окна есть кнопка Browse, которая активизирует подменю, предоставляющие дополнительные функции просмотра.

Вкладка Detail (детализация) более насыщена (рис. 5.4).

Здесь пользователю предоставляется возможность ввести дополнительные данные, характеризующие процессор.

Поле Characteristics (характеристики) предназначено для указания физических характеристик используемого аппаратного обеспечения, таких как изготовитель, модель, объем дисковой и оперативной памяти и т.д. Эта информация не будет отображаться на диаграммах, но поможет определить общие требования системы к аппаратному обеспечению.

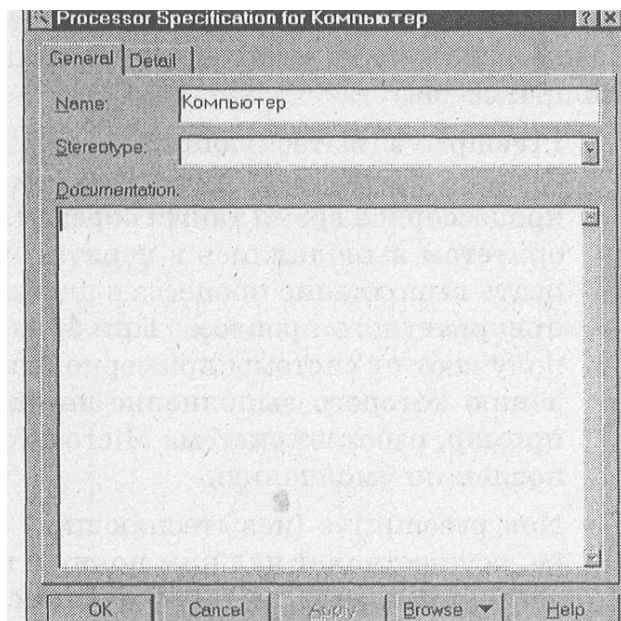


Рис. 5.3 Вкладка General спецификаций процессора

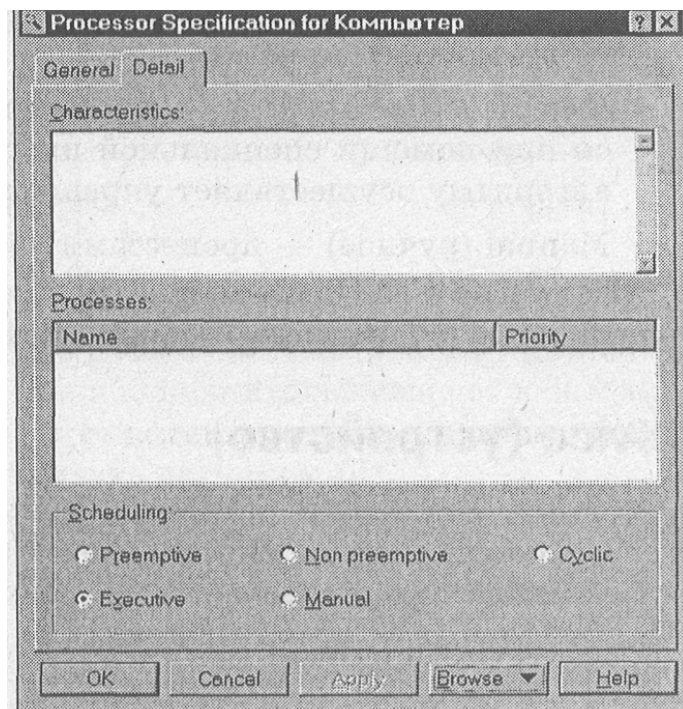


Рис. 5.4 Вкладка Detail спецификаций процессора

Поле process (процессы) позволяет создать список процессов, которые работают на данном процессоре. Каждый процесс обозначает или главную программу, или активный объект системы.

Совет

Для того чтобы добавить процесс в список, необходимо внутри списка Processes выбрать RClick=>Insert.

Для того чтобы показать порядок выполнения процессов на каждом

процессоре, мы можем воспользоваться группой переключателей Scheduling (планирование). Возможна установка следующих вариантов планирования выполнения процессов:

- Preemptive (вытесняющий) – процесс с более высоким приоритетом вытесняет процессы с более низким приоритетом. Система распределяет процессорное время таким образом, чтобы процессы с более высоким приоритетом выполнялись в первую очередь, и может в любой момент прервать выполнение процесса с низким приоритетом для выполнения более приоритетного процесса. При этом процессы с одинаковым приоритетом получают от системы примерно одинаковое процессорное время, по истечению которого выполнение передается следующему процессу. Так, например, работает система Microsoft Windows 95 и выше. Этот пункт установлен по умолчанию.

Non preemptive (невывтесняющий) процесс, запущенный на процессоре, осуществляет над ним полный контроль до тех пор, пока сам не передаст управление другому процессу. Так, например, работала система Microsoft Windows 3.11.

- Cyclic (циклический) – всем процессам выделяется равное количество процессорного времени.

Executive (диспетчер) – переключение между процессами осуществляется при помощи специальной программы-диспетчера, которая по своему алгоритму осуществляет управление процессами.

Manual (ручное) – процессами управляет оператор.

Для гидропонной системы, которая не предъявляет требований к порядку выполнения процессов, можно оставить параметры, установленные по умолчанию.

Device (устройство)



Данный инструмент позволяет создавать на диаграмме объект устройства, неспособного выполнять программы. Каждое такое устройство также относится к аппаратному обеспечению и должно иметь общее для данного вида имя, такое как «модем» или «терминал».

Замечание

Пользователь может создать диаграмму Statechart или Activity устройства из контекстного меню.

Connection (соединение)

Данный инструмент позволяет связать между собой устройства и процессоры. Connection представляет собой некоторый тип кабельного или другого соединения, например, соединение при помощи сетевых карт, последовательных или параллельных портов или даже связь «Земля - спутник». В отличие от реального соединения, на диаграмме не может

быть показано направление перемещения информации посредством соединения, и считается, что соединение всегда двунаправлено.

Для объектов Connection и Device контекстное меню и спецификации достаточно просты, поэтому нет смысла их здесь обсуждать. Вы можете разобраться в них самостоятельно.

Устройства тепличного хозяйства

Теперь вы можете создать устройства, таким образом, как это показано на рис. 5.5.

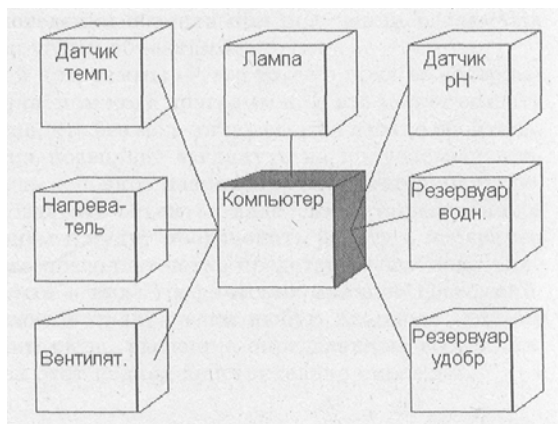


Рис. 5.5 Устройства тепличного хозяйства

Полученная диаграмма показывает, что работа системы планируется на одном компьютере, соединенном с датчиками и исполнительными устройствами.

Датчиков в системе планируется два: датчик температуры и датчик кислотности, который в дальнейшем будем называть датчиком pH.

Исполнительные устройства представлены лампой, нагревателем, вентилятором, водным резервуаром и резервуаром с удобрениями.

Лампа должна включаться при недостатке освещения, однако, датчик освещенности не предусмотрен, значит лампа должна гореть всегда, когда система переходит в режим «день», и выключаться, когда система переходит в режим «ночь».

Нагреватель и вентилятор должны обеспечивать необходимую для выращивания температуру. При снижении температуры должен включаться нагреватель, а при ее повышении - вентилятор.

Водный резервуар и резервуар для удобрений планируется использовать для изменения уровня кислотности. При повышении кислотности из водного резервуара должна поступать вода, при этом уровень кислотности будет снижаться. При понижении кислотности из резервуара поступают удобрения, и уровень кислотности повышается.

Вопросы для повторения

Каково назначение Deployment диаграммы?

Каково назначение объекта Processor?

Какие возможности контекстного меню для объекта Processor?

Какие спецификации можно настроить для объекта Processor?
Какое назначение объекта Device?
Какое назначение объекта Connection?

Глава 6. Создание модели поведения системы при помощи диаграммы Statechart

Назначение диаграммы

Диаграмма Statechart (диаграмма состояний) предназначена для описания состояний объекта и условий перехода между ними. Описание состояний позволяет точно описать модель поведения объекта при получении различных сообщений и взаимодействии с другими объектами.

Единственный недостаток этой диаграммы - это то, что пока ее содержание никак не отражается на получаемом коде программы. И это может создать у программиста ошибочное мнение, что без модели поведения можно обойтись. Но это не так. Модель поведения позволяет взглянуть на получаемый программный объект со стороны, ведь основное назначение объектно-ориентированного программирования - создавать объекты, наделенные определенным поведением, которые в дальнейшем и будут производить работу в программном коде. А данный тип диаграмм позволяет четко представить все поведение полученного программного объекта в виде графических значков состояний. Согласно теории конечных автоматов практически любую сложную машину можно разложить на простые автоматы, имеющие определенные состояния, поэтому в программных системах этот подход действительно оправдан.

Создание заготовок классов

Для того чтобы начать создавать модель поведения, необходимо добавить в модель классы, на основе которых эти объекты впоследствии будут создаваться.

Исходя из рассмотренных в главе 5 устройств, получаем следующих кандидатов для создания классов:

EnvironmentalController	контроллер	управления
исполнительными устройствами;		

TemperatureSensor - датчик температуры;

pH Sensor - датчик кислотности;

Heater - нагреватель;

Cooler - вентилятор для снижения температуры;

Light - осветитель;

WaterTank - хранилище для воды;

NutrientTarik - хранилище для удобрений.

Я назвал их так, но это дело вкуса и желания соблюдения соглашений об использовании имен. Конечно, для примера можно было бы использовать русские наименования, но поскольку программа Rational Rose не локализована для России, то английские наименования позволят не опасаться, что при генерации исходного кода на выбранном языке программирования процесс может закончиться неудачей в случае конфликта кодировок. Для добавления класса в модель необходимо

перейти в окно Browser и на строке Logical View проделать RClick=>New=>Class, как показано на рис. 6.1.

Измените название NewClass на EnvironmentalController, и первый класс готов. Аналогично добавьте остальные классы, и можно начинать создавать модель поведения.

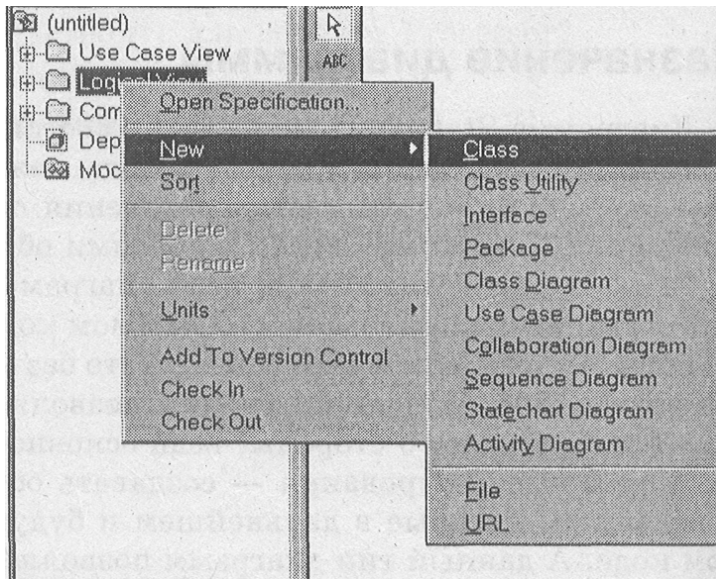


Рис. 6.1 Создание нового класса в окне Browse

Создание Statechart диаграммы

В нашей системе самым сложным поведением обладает класс Environmental-Controller, поэтому будем разбирать создание модели поведения на его примере. Для создания диаграммы есть несколько путей.

Создать посредством меню MenuBrowseState Machine Diagram.

Создать посредством строки инструментов при помощи значка



Statechart.

3. Выбрать пункт New Statechart Diagram из контекстного меню класса.

В последних двух случаях происходит активизация окна выбора диаграммы (рис. 6.2).

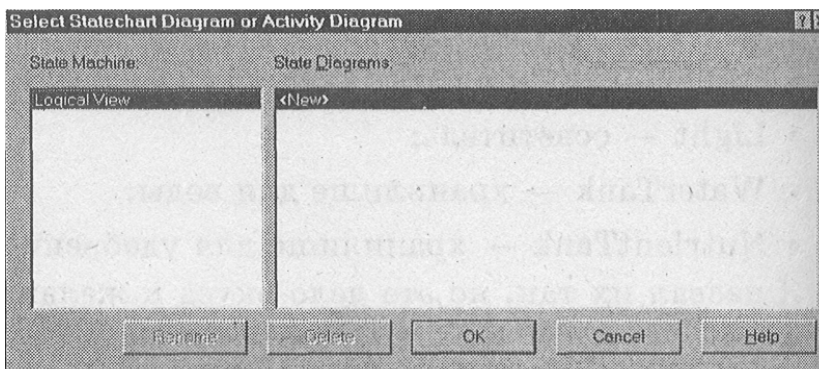


Рис. 6.2 Выбор диаграммы Statechart или Activity

При помощи этого диалогового окна пользователь может выбрать диаграмму для назначения ее текущей, создать новую, удалить или переименовать уже существующую. При создании новой диаграммы будет предложено выбрать из двух возможных (рис. 6.3). Для создания диаграммы выберем тип Statechart. Создание диаграммы Activity мы рассмотрим в следующей главе.

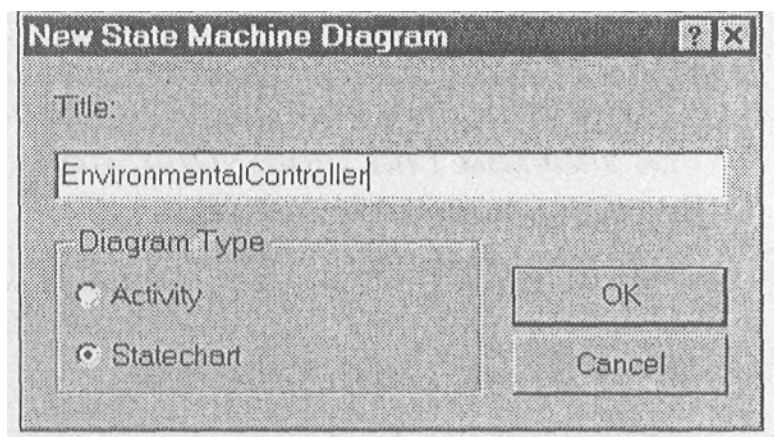


Рис. 6.3 Выбор типа диаграммы

Инструменты диаграммы Statechart

После активизации диаграммы становятся доступны следующие инструменты (рис. 6.4).

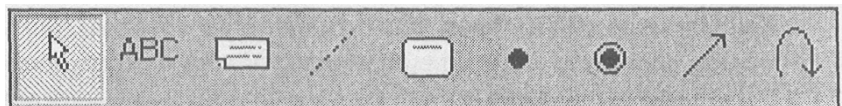


Рис. 6.4 Инструменты Statechart Diagram

Не будем подробно останавливаться на значках, которые уже знакомы нам по предыдущим диаграммам, а лишь перечислим их, для того чтобы освежить их назначение в памяти:

Selection Tool выбора объекта, с которым будет производиться дальнейшая работа;

TextBox – создание отвлеченной надписи на диаграмме;

Note – создание надписи к значку состояния State;

Anchor Note - соединение значка Note и State.

State (состояние)

Инструмент State позволяет отразить состояние или ситуацию в течение жизни объекта, которая отвечает некоторому положению объекта или ожиданию им некоторого события. Каждое состояние представляет собой совокупную историю поведения объекта. Его имя должно быть уникально внутри класса, так как состояния с одинаковыми именами считаются представлением одних и тех же состояний. В текущем значке State могут быть отражены действия по входу, выходу из состояния, действия не связанные с событиями или

реакция на события. Подробнее установку этих действий

рассмотрим далее.



Start State (начало)

Инструмент Start State позволяет создать значок начала работы. Для диаграммы Statechart он обозначает событие, которое переводит объект в первое состояние на диаграмме.

Замечание

Все диаграммы состояний начинаются со значка Start State и должны заканчиваться значком End State. При этом значок начало работы может быть только один, а значков окончания может быть сколько угодно. За этим Rational Rose следит самостоятельно.

End State (завершение)

Инструмент End State позволяет создать значок окончания работы. Направление перехода может быть установлено только в данный значок, однако, никаких ограничений на количество переходов в End State, а также на количество таких элементов на диаграмме, не налагается.



State Transition (состояние перехода)

Инструмент State Transition позволяет создать значок состояния перехода, который означает, что объект переходит из одного состояния в другое в случае наступления определенного события или по изменению определенных условий.

Пользователь может указать несколько переходов из одного состояния в другое в случае, если каждый такой переход осуществляется при наступлении разных событий или при соблюдении разных условий.



Transition To Self (переход на себя)

Инструмент Transition To Self позволяет создать значок перехода в то же состояние, из которого осуществляется переход.

Данный переход похож на State Transition, однако, он не осуществляет переход в другое состояние при наступлении некоторого события. Таким образом, при наступлении события оно обрабатывается, и после обработки объект возвращается в то состояние, в котором он находился до наступления события.

Первые шаги в создании диаграммы

Первым шагом для создания диаграммы будет создание точки начала работы. Создайте ее при помощи кнопки Start State.

Совет

Для добавления в диаграмму новых элементов можно воспользоваться Menu:Tools = Create.

Обычно первым состоянием системы после начала работы будет

ожидание наступления событий. И в нашем случае не будем делать исключений. Создадим новое состояние (State) и соединим его стрелкой State Transition с начальной точкой.

Каждое состояние или событие должно иметь свое имя, поэтому присвоим имя состоянию ожидания RClick=>OpenSpecification=>General==>Name = Idle, что в переводе означает «ожидание» (рис. 6.5).

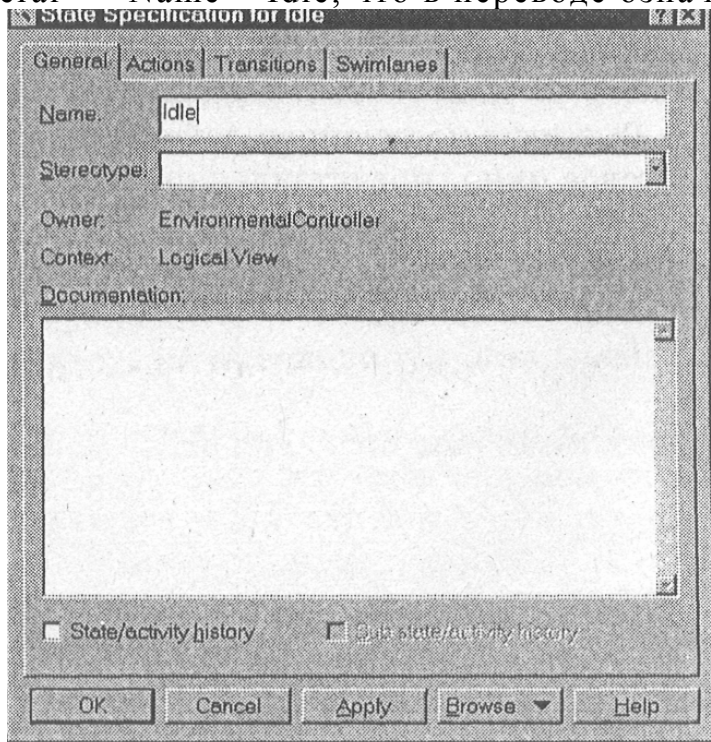


Рис. 6.5 Вкладка General для спецификаций состояния

Для того чтобы назвать событие, которое переводит систему из начального состояния в состояние ожидания, выделим стрелку события и сделаем следующее RClick =>OpenSpecification =>General =>Event=New Planting, что в переводе означает «посадка семян». Должно получиться состояние, показанное на рис. 6.6.

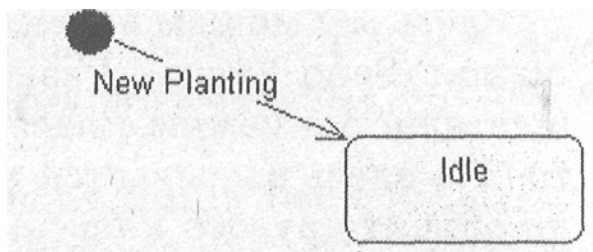


Рис. 6.6 Состояние ожидания

Состояние тестирования датчиков

Задача нашего контроллера состоит в поддержании заданных значений; параметров среды теплицы: температуры, освещенности, концентрации рН. Для того чтобы эти параметры поддерживать согласно плану, необходимо считывать текущее состояние среды с помощью датчиков.

В нашем случае не будем рассчитывать на большую интеллектуальность датчиков. Датчики не будут иметь своего процессора и будут только выдавать значения измененных параметров по запросу контроллера. Таким образом, следующим состоянием будет опрос датчиков. Добавим это состояние на диаграмму, назовем Testing Environment и соединим с состоянием Idle. Это событие назовем Timer. Имеется в виду, что у контроллера есть встроенные часы, которые через заданное время инициализируют это событие.

Теперь разберем подробнее, что необходимо сделать для тестирования параметров среды.

Активизировавшись, контроллер опрашивает датчики температуры и pH. Для того чтобы отразить, что опрос датчиков происходит в течение данного состояния, необходимо добавить новое действие (Action). Для этого во вкладке Action из контекстного меню выбираем Insert.

Выберем полученное действие двойным нажатием мыши и попадем в диалоговое окно, представленное на (рис. 6.7).



Рис. 6.7 Настройка параметров действия

Здесь мы можем переключаться между действием (Action) и посылкой сообщения (Send Event). Разница между этими пунктами в том, что действие осуществляется самим классом, для которого мы создаем диаграмму состояния, то есть здесь вызывается метод этого класса, а посылка сообщения направлена на объект другого класса, метод которого вызывается при помощи этого сообщения. Этот объект задается в строке Send Target. Также можно задать имя вызываемого метода класса в строке Send Event и аргументы в строке Send Arguments.

Можно настроить момент, в который происходит отмеченное действие. Здесь предоставляется выбор из четырех состояний:

On entry показывает, что указанное действие необходимо производить при входе в состояние до выполнения остальных действий.

On exit показывает, что действие должно быть выполнено перед самым выходом из указанного состояния.

Do - действия производимые в течение состояния до выхода. Если таких действий набирается несколько, то почти наверняка их можно

выделить в отдельную диаграмму состояния.

On Event – действия в ответ на определенные события, обрабатываемые в текущем состоянии. При выборе этого пункта открывается возможность заполнить события (Event), параметры (Arguments) и условия, когда это действие обрабатывается (Condition), то есть может случиться так, что событие произошло, а условие его обработки не наступило. Этот пункт удобно использовать при обработке события, которое не приводит к переходу в другие состояния и отражается значком Transition to, Self.

Нужно заметить, что при изменении этих параметров изменяется и надпись на значке состояния.

События отражаются при помощи символа ^ перед ним. Действия: при входе: entry:, при выходе: exit:, в течение работы: do:, действия по сообщению: on:. Условие обработки показывается выражением в квадратных скобках. Данная нотация довольно удобна и позволяет, не активизируя окно свойств, одним взглядом оценить сделанное.

Вывод диаграммы из цикла

После того как протестировано состояние среды, необходимо привести показатели к норме посредством включения соответствующих исполнительных устройств. Для отражения этого действия создадим состояние Adjusting Envi-ronment, которое соединим с состоянием Testing Environment.

После настройки среды система переходит в ожидание следующего момента времени. Отразим это при помощи стрелки, соединяющей состояния Adjusting Environment и Idle.

Однако у нас получилась система с бесконечным циклом. У нее нет выхода. Но мы знаем, система должна закончить работу в тот момент, когда урожай созрел. Ключом созревания будем считать истечение времени выращивания растения по плану выращивания.

Для отражения этого введем состояние Analysing Time, которое включим между состоянием Idle и Testing Environment. Перетащим конец стрелки Timer на Analysing Time и соединим его новой стрелкой Testing Environment. Добавим End State, соединим это состояние с Analysing Time, заполним условие для необходимого действия RClick=>Open specification:=>Detail (рис. 6.9), говорящее о том, что переход осуществляется только тогда, когда истекло время роста растения. Это окно имеет практически те же заполняемые поля, что и рассмотренное ранее, поэтому не будем на нем подробно останавливаться. Единственное, что хотелось бы отметить, это то, что стрелку можно перенести прямо отсюда, изменив позиции From (откуда) и To (куда).

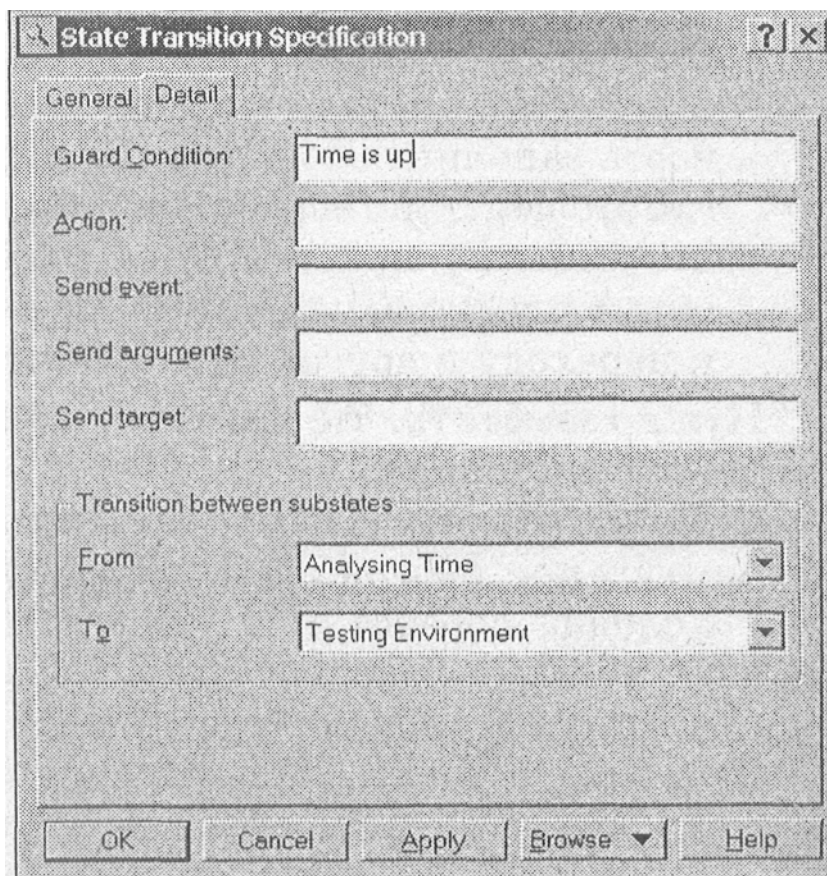


Рис. 6.9 Заполнение условия выполнения перехода

Совет

Для того чтобы быстрее добраться до спецификаций, можно использовать двойное нажатие мыши на диаграмме или стрелке.

Добавление замечания

Для добавления последнего, еще не использованного нами значка, введем в диаграмму комментарий, который соединим при помощи значка Anchor Note to Item с состоянием Analysing Time, и должно получиться примерно следующее (рис. 6.10).

Настройка среды

Распишем подробнее, какие состояния должна пройти система для настройки среды в соответствии с планом выращивания, при этом сделаем следующие допущения:

для увеличения температуры в теплице необходимо включить нагреватель;

для уменьшения температуры необходимо включить вентилятор;

для переключения в режим «День» необходимо включить освещение;

для переключения в режим «Ночь» необходимо выключить освещение;

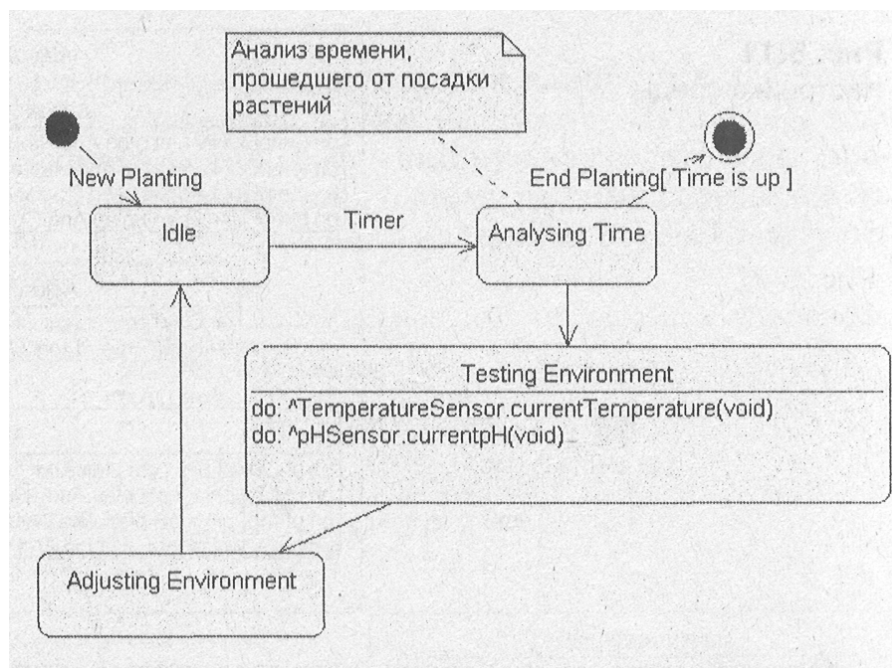


Рис. 6.10 Диаграмма состояния после добавления замечания

для уменьшения рИ необходимо добавить в раствор воды;
для увеличения рН необходимо добавить в раствор удобрений.

Причем возможно одновременное включение нагревателя, лампочек и водяной заслонки, но должно быть невозможно одновременное включение нагревателя и вентилятора, а также одновременное поступления воды и удобрений.

Произведя анализ необходимых действий, нетрудно заметить, что перечисленный набор можно разделить на три независимые части, это:

- настройка температуры;
- настройка освещения;
- настройка уровня рН.

Так как мы приняли допущение, что процессор у нас только один, то эти состояния будут пройдены последовательно. Но если для каждой части используется свой процессор, то можно было бы создать автономные классы для управления этими состояниями.

Rational Rose предоставляет возможность создания вложенных диаграмм состояния, что удобно для большей детализации каждого состояния. Воспользуемся этим в нашем случае.

Добавим три новых состояния Adjusting t (настройка температуры), Adjusting Light (настройка освещения) и Adjusting pH (настройка рН). Для этого добавим новые состояния прямо в значок Adjusting Environment. Перед добавлением значок будет выделен рамкой, а после добавления будет автоматически раздвинут, чтобы в него уместился добавленный элемент. Изменим направления входящей и выходящей стрелок так, чтобы входящая указывала на Adjusting pH, а выходящая исходила из Adjusting t, и соединим эти состояния последовательно. Теперь распишем, что происходит в этих состояниях (рис. 6.11).

Настройка температуры происходит только при ее изменении, что показывают функции tempDown и tempUp. Если произошло одно из этих событий, то температура сравнивается с той, которая должна быть по

плану и если она меньше, то включается нагреватель, а если больше - вентилятор.

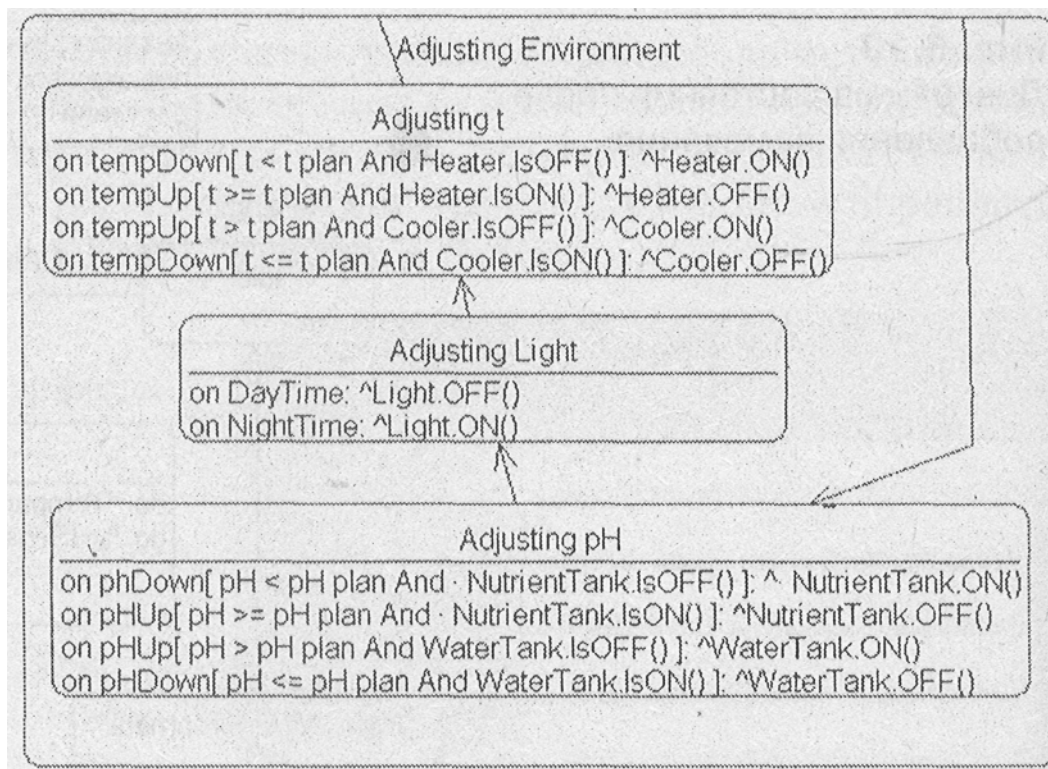


Рис. 6.11 Настройка среды

Однако если температура достигла нормального уровня, то вентилятор и нагреватель отключаются. Аналогично происходит и с уровнем pH.

Для освещения лампочка просто включается при наступлении времени освещения и выключается при наступлении ночного времени суток.

Совет

Для того чтобы создать ломаную стрелку, нужно «потянуть» мышкой за ее середину.

Скрытие вложенных состояний

Теперь диаграмма стала достаточно трудна для обозрения. Представьте, что у вас таких состояний десятков. Общая картина теряется при большом количестве объемных состояний. Rational Rose позволяет скрыть ненужные в данный момент вложенные состояния. Для этого нужно выделить состояние, а затем выбрать Menu: View=>Show Nested Elements. При этом в правом нижнем углу состояния появится многоточие, а входящие и исходящие стрелки приобретут вид, показанный на рис. 6.12 в состоянии Adjusting Environment.

Совет

Часто удобнее пользоваться не вложенными диаграммами, а

создавать поддиаграммы (Sub diagram) из контекстного меню состояния.

States History (история состояний)

Единственное, что осталось не рассмотренным на этом типе диаграмм, это настройка States History на вкладке General спецификаций.

Включение этой настройки позволяет показать, что в следующий раз, когда система попадает в указанное состояние, она должна не начинать с начала состояний, а сразу перейти на последнее состояние, из которого вышла, то есть при первом входе в некоторое состояние производятся единичные действия, которые при следующем входе проделывать уже не нужно. Например, при первом входе в режим протоколирования сообщений нужно создать файл протокола, который при последующих обращениях к этому режиму пересоздавать не нужно. На рис. 6.12 видно, что установленная настройка States History отражается буквой H в кружке.

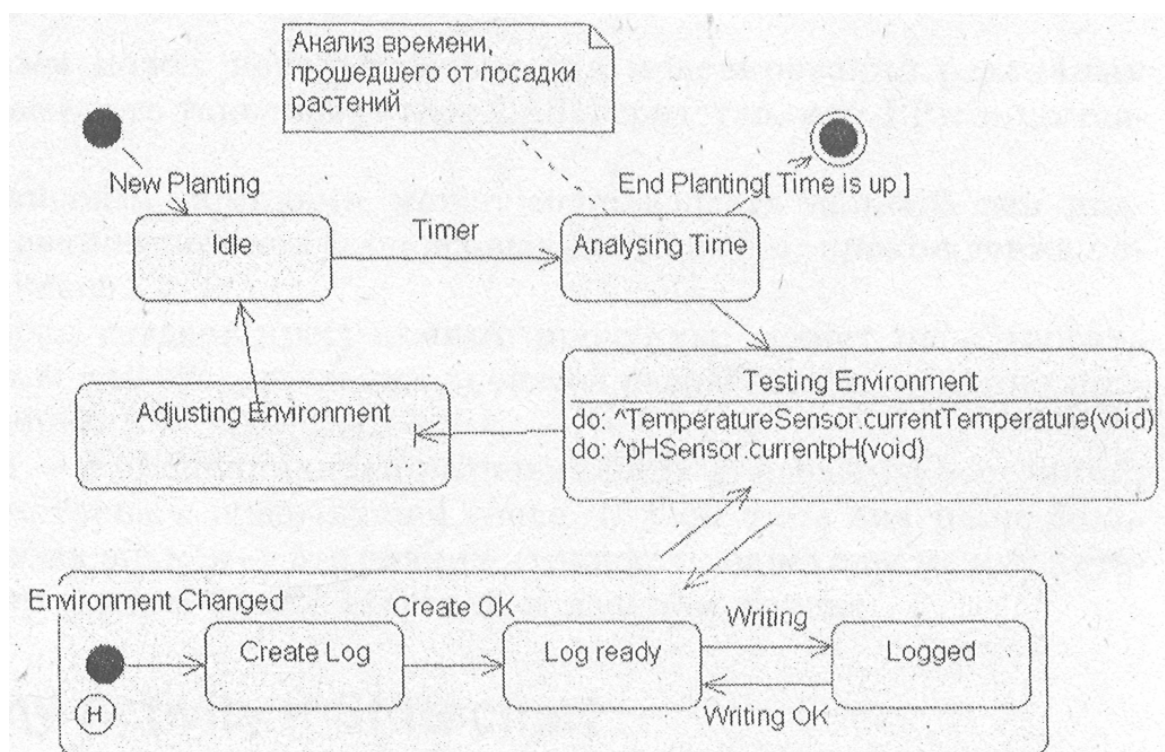


Рис. 6.12 Окончательный вариант диаграммы состояний контроллера среды

Протоколирование начинается при изменениях в условиях (Environment Changed). После этого создается Log файл (Create Log), и система переходит в состояние ожидания (Log ready). При необходимости записать изменения они записываются (Logged), и система снова переходит в состояние ожидания. При этом в следующий раз необходимо начинать с состояния Log ready, а не с создания протокола.

Вопросы для повторения

1. Для чего предназначена диаграмма Statechart?
2. Какие инструменты доступны в диаграмме?
3. Какие бывают переходы между состояниями?
4. Какие спецификации можно задать для переходов между состояниями?
5. Что такое история состояний?

Глава 7. Создание модели поведения при помощи Activity Diagram

Назначение диаграммы активности

Этот тип диаграмм может использоваться для моделирования различных типов действий и заменять такое известное CASE-средство, как BPwin компании PLATINIUM.

Например, финансовая компания может использовать данный тип диаграмм для моделирования потоков финансовых документов, прохождения оплаты счетов или заказов.

Компания, которая создает программные продукты, может использовать данный тип диаграмм для отслеживания процесса разработки и создания программного обеспечения.

Activity diagram - это специальная разновидность диаграммы состояний, которая была рассмотрена в предыдущей главе. В этом типе диаграмм большинство используемых знаков – это знаки активности, переходы между которыми вызваны завершением одних действий и началом других.

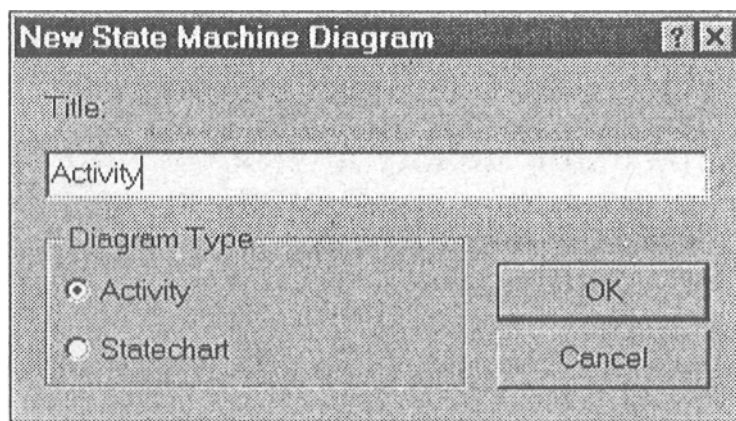
Отличия между Activity и Statechart

Главное отличие между Activity и Statechart diagram в том, что в первом случае основное действия, а во втором статичное состояние. При этом Activity diagram больше подходит для моделирования последовательности действий, а Statechart для моделирования дискретных состояний объекта.

Если посмотреть на созданную ранее Statechart, то можно заметить, что многие использованные нами значки описывают не просто состояние, а состояние некоторого действия. Поэтому диаграмма активности для нашего случая подходит даже лучше. Но в учебных целях Statechart была рассмотрена нами раньше.

Создание диаграммы активности

Выберем пункт Browse=>State Machine Diagram и создадим новую диаграмму Activity, как показано на рис. 7.1.

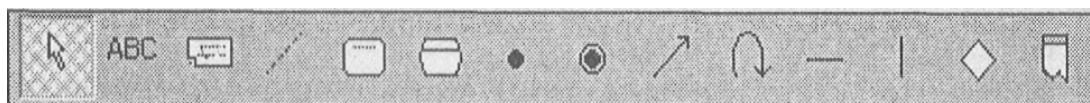


Замечание

Для создания диаграммы вы можете воспользоваться любым способом, аналогичным описанному для создания Sfatecharf диаграммы в главе 6.

Строка инструментов

После того как диаграмма будет активизирована, строка инструментов будет иметь следующий вид (рис. 7.2)



Инструменты, рассмотренные ранее

На элементах TextBox, Note и Anchor to Note останавливаться не будем по причине того, что эти элементы не имеют ничего нового по отношению к предыдущим диаграммам и достаточно просты для понимания.

Значок State аналогичен такому же на диаграмме Statechart и предназначен для обозначения ситуации в течение жизни объекта, когда объект ожидает некоторое событие или находится в некотором состоянии, как, собственно, и переводится термин State.

Из набора значков состояний можно составить представление о всем жизненном цикле объекта.

Start State и End State – начало и окончание работы объекта соответственно.



Значок Activity обозначает выполняемые задачи или выполнение определенных действий в течение жизни объекта. Этот значок, в общем-то, похож на предыдущий, но если значок State обычно обозначает ожидание какого-либо события, то значок Activity обозначает непосредственное действие.

State Transition (переход состояния)



State Transition - переход из одного состояния в другое или по завершении выполнения определенного действия в начало другого. Этот значок также может характеризовать получение объектом некоторого сообщения с дальнейшей его обработкой. State Transition может осуществляться как между Action-Action и State-State, так и между State-Action и Action-State.

Возможна также установка нескольких переходов между двумя состояниями или действиями. Каждый такой переход уникален и показывает реакцию объекта на определенное сообщение. Таким образом, нельзя создать несколько переходов между двумя состояниями с указанием одного и того же сообщения.

Synchronizations (синхронизация)



Значок Synchronizations позволяет определить независимо выполняемые действия. При этом действия разделяются на несколько выполняемых независимо, и только по завершении всех действий объект продолжает работу. Этот значок представляет собой горизонтальную или вертикальную черту, обозначающую синхронизацию выполняемых работ.

Decision (решение)



Decision позволяет показать зависимость дальнейшей работы от внешних условий или решений. Этот значок аналогичен командам языка программирования if или case и может иметь больше двух выходов, но обычно используют выбор из двух переходов, определенных Булевым выражением.

Swimlanes (плавательные дорожки)



Swimlanes позволяет моделировать последовательность действий различных объектов и связи между ними. При помощи этого элемента можно моделировать бизнес-процессы организации, отражая на диаграмме различные подразделения и объекты, играющие важные роли в модели бизнеса.

Swimlanes позволяет показать, кто выполняет те или иные роли в процессе. Для этого необходимо переместить соответствующие значки активности или состояний в зону определенного подразделения, отделенного от остальных Swimlanes.

Начало создания диаграммы

Теперь посмотрим, как изменится диаграмма, если создавать не

диаграмму состояния, диаграмму активности для работы объекта `environmentalController`.

Создадим начало диаграммы, поместив на диаграмму значок `Start State` и состояние `Idle`. Соединим их связью `State Transition` и назовем ее `New Planting`.

Чтобы не вводить еще раз эти значки, можно выделить их на уже созданной диаграмме `Statechart` и перенести во вновь созданную.

Совет

Для выделения нескольких элементов диаграммы необходимо, удерживая клавишу

`Ctrl`, отметить мышкой поочередно каждый выделяемый элемент.

Теперь вспоминаем, что перед тем, как перейти в состояние ожидания, контроллер должен установить значение таймера, продолжительность выращивания, время ожидания между опросами датчиков. В этом случае можно сразу разделить функции, выполняемые таймером и контроллером, для чего создадим отдельные разделители `Swimlanes` для этих объектов.

Редактирование спецификаций `Swimlanes`

Добавим новые разделители `Swimlanes`- для таймера и контроллера. Эти разделители будут названы программой `Rational Rose` автоматически `NewSwimlane` и `HNewSwimlane1`, что, конечно же, необходимо исправить. Для этого активизируем окно свойств `Swimlane`. Для доступа к окну свойств необходимо выбрать заголовок линии и далее `RClick->Open Specification`. Откроется окно, показанное на рис. 7.3.

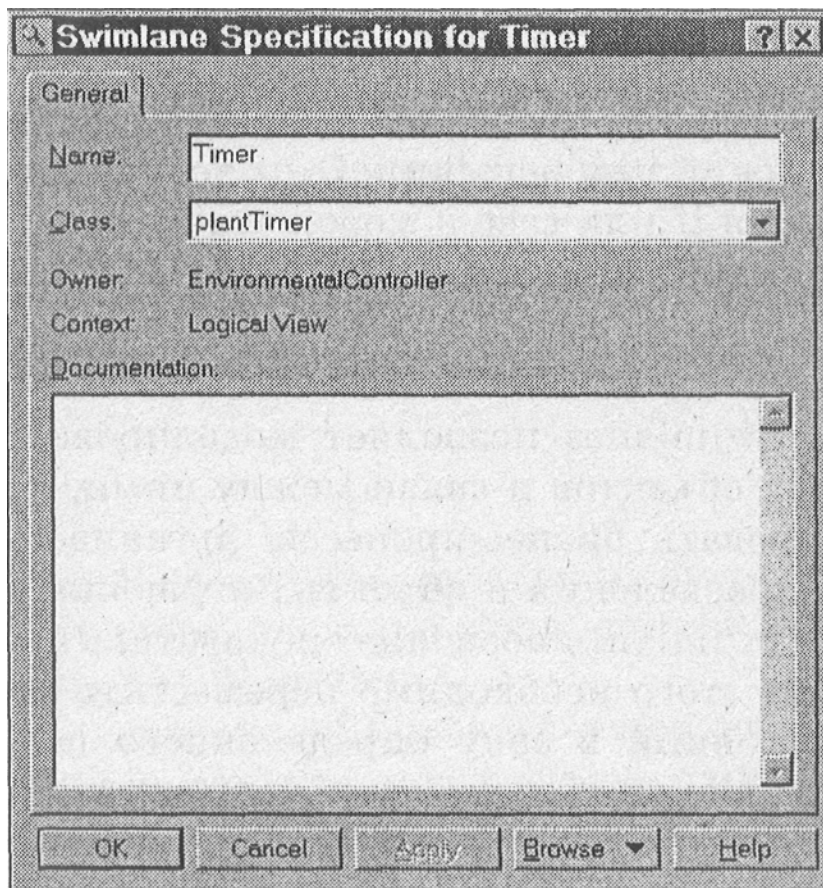


Рис. 7.3 Редактирование свойств Swimlanes

Здесь можно изменить название и имя класса, из которого получен объект. В нашем случае это `plantTimer`. И конечно же, можно документировать полученную Swimlane.

Введем информацию, как она представлена на рисунке. Аналогично изменим информацию для следующей линии, которая представляет Controller, полученный из класса `EnvironmetalController`.

Теперь необходимо «перетащить» введенные значки в колонку Controller для того, чтобы показать, что основным действующим лицом в нашем случае является именно он.

Если вы проделывали все действия, то должно получиться то, что показано на рис. 7.4.

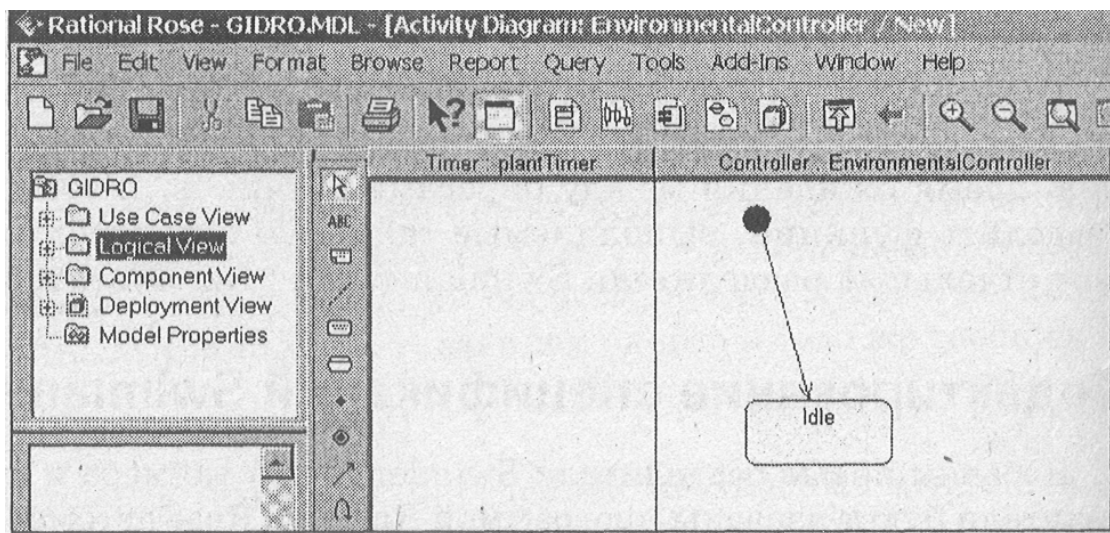


Рис. 7.4 Добавленные в диаграмму активности Swimlanes

Однако для установки таймера необходимо произвести действие установки. Отразим этот момент добавлением нового значка активности.

Настройка спецификаций значка активности

Для значка активности доступно следующее окно изменения свойств, показанное на рис. 7.5.

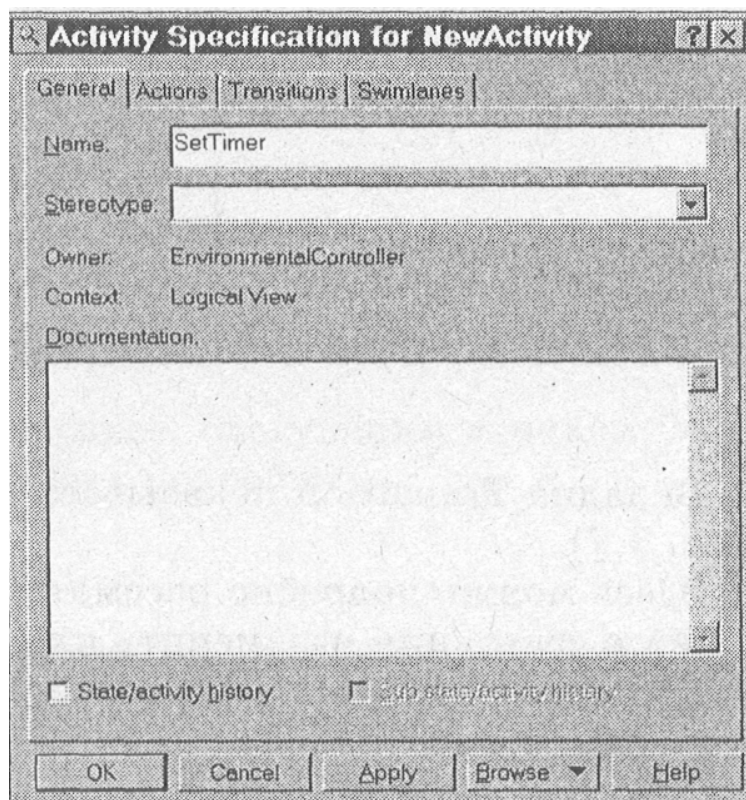


Рис. 7.5 Свойства значка активности

Изменим наименование на SetTimer. Откроем вкладку Actions и добавим новое действие RClick=>Insert. Перейдем в окно свойств действия и получим окно, показанное на рис. 7.6.

Коротко повторим рассмотренные в главе 6 варианты настройки момента наступления указанного действия. Возможные варианты:

- On Entry - действие происходит при входе в данный тип активности;

On Exit – действие происходит при выходе из данного типа;

D6 – действие происходит между действиями входа и выхода;

- On Event – действие происходит в ответ на определенное событие. Если выбрано действие в ответ на определенное событие, то открываются поля ввода аргументов этого события.

В поле ввода типа можно указать, какой тип активности предполагается использовать. Это может быть Action (действие) или Send Event (посылка сообщения).

- Action может использоваться для того, чтобы показать, что данный тип активности не взаимодействует с другим объектом, и действие происходит внутри объекта.

Send Event показывает, что происходит взаимодействие с другими объектами посредством посылки сообщений. Здесь можно ввести имя сообщения, его аргументы и адресат сообщения, как сделано в нашем случае.



Рис. 7.6 Настройка спецификаций

Вкладка Transition показывает все входы и выходы из значка активности (рис. 7.7).

Здесь можно подробно рассмотреть все входящие и выходящие события из данного состояния и изменить их спецификации прямо из этого окна.

Вкладка Swimlanes показывает, с какими Swimlanes значок Activity или State взаимодействует.

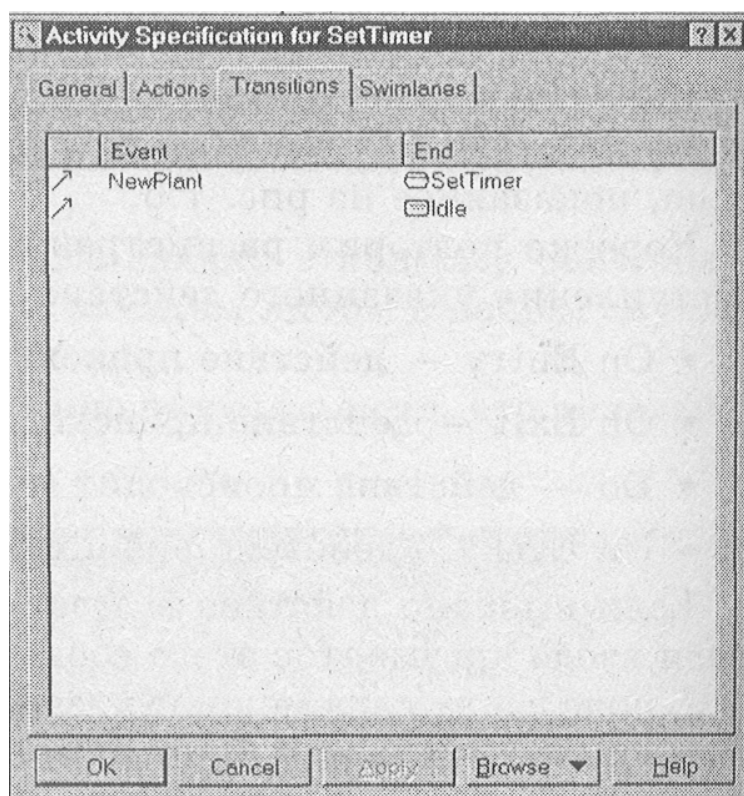


Рис. 7.7 Вкладка Transition

Если все было заполнено правильно, то должно получиться состояние, показанное на рис. 7.8.

Совет

Значок активности можно преобразовать в значок состояния при помощи `RClick=> Change Into State`.

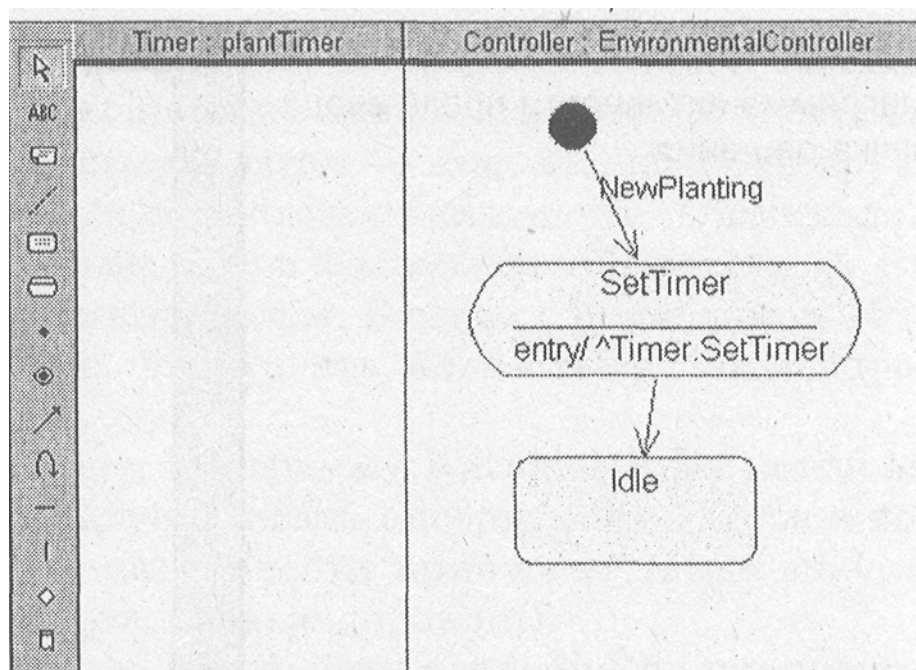


Рис. 7.8 Диаграмма после заполнения значка активности

Создание значка анализа времени

На диаграмме Statechart, созданной нами ранее, следующим значком был анализ времени, который здесь можно расписать подробнее.

Совет

Переключаться между двумя последними диаграммами удобно при

помощи значка Browse Previous Diagram.



Добавление значка получения времени

Из состояния ожидания контроллер должен быть выведен таймером, после чего происходит проверка текущего времени выращивания на предмет его окончания. Допустим, что таймер может выдать время, которое прошло от момента начала посадок, и контроллер должен получить это время для анализа, не пора ли заканчивать работу.

Для отображения этого процесса создадим новый значок Activity и назовем его GetCurrentTime, который соединен со значком Idle событием Timer. Добавим значок GetCurrentTime для отражения обращения к таймеру за текущим временем. Создадим значок ReturnTime для объекта Timer и введем значок решения для контроллера рис. 7.9.

Для того чтобы отразить, что работа контроллера должна быть закончена, добавим значок EndState и соединим его стрелкой с условием, что переход должен осуществляться только в случае, когда закончилось время выращивания растений. На рис. 7.10 показано, как необходимо заполнить свойства для StateTransition в этом случае.

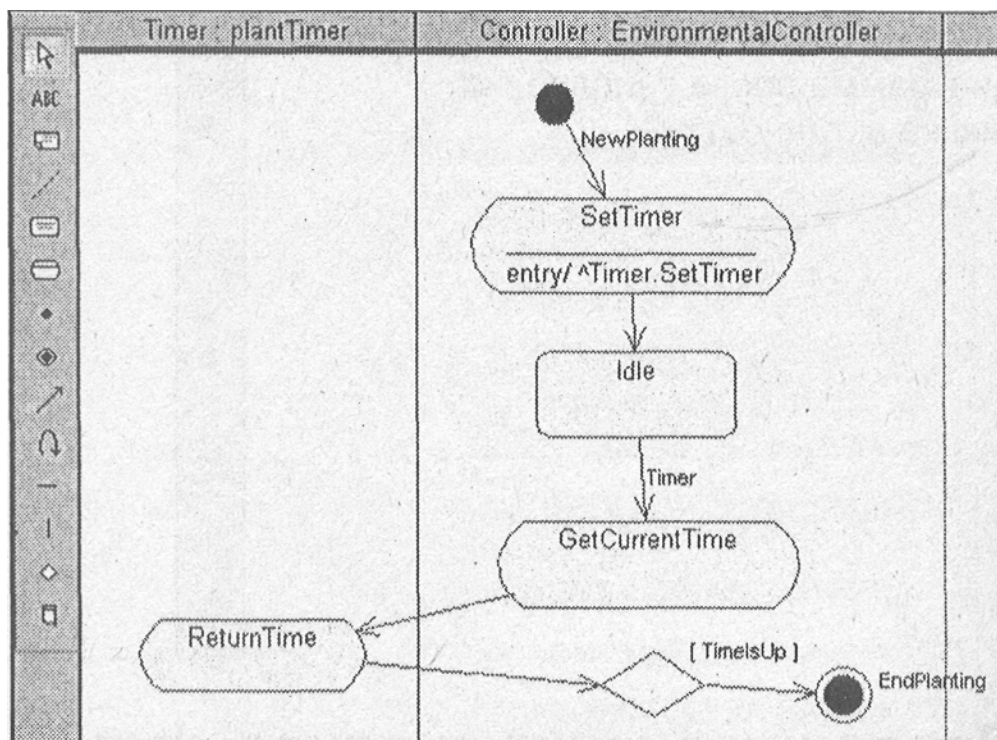


Рис. 7.9 Диаграмма активности после ввода значка решения

State Transition Specification

General | Detail

Guard Condition:

Action:

Send event:

Send arguments:

Send target:

Transition between substates

From:

To:

OK Cancel Apply Browse Help

Рис. 7.10 Установка свойств для StateTransition

Добавление решения

Значок решения стал, по моему мнению, достаточно неплохим добавлением относительно ранних версий Rational Rose. Теперь есть возможность показать ветвление по определенным условиям, и хотя

данный тип диаграмм никак не отражается на получаемом программном коде (по крайней мере пока), но он позволяет точно описать последовательность действий при выполнении определенных задач, для которых показ ветвления может оказаться просто необходимым.

Если вы программист, то попробуйте вспомнить, когда вы последний раз рисовали блок-схемы алгоритмов. Сейчас это происходит все реже и реже, именно по причине ограничения времени на выполнение проектов и доступности вычислительных машин.

В то время, когда доступ к ЭВМ осуществлялся по предварительной записи, никто не мог позволить себе такую роскошь, как изобретать алгоритм, сидя за клавиатурой терминала. Сначала блок схема, затем - код, написанный вручную на листах перфорированной бумаги, и только затем клавиатура терминала.

Сейчас многое изменилось, программы стали создаваться быстрее, но их поддержка становится все более трудоемким делом. В связи с появлением объектов, обменивающихся сообщениями, нарисовать блок-схему таких программ стало значительно труднее.

Но теперь вместо блок-схем мы имеем программу Rational Rose, которая позволяет значительно проще и нагляднее описать алгоритм любой сложности, и немалую часть этой простоты и наглядности приносят такие значки как Decision (решение) и Swimlane (плавательные дорожки).

В нашем случае значок решения будет обозначать проверку на окончание плана выращивания растения. Для большего упрощения задачи будем использовать специализированный контроллер, который запрограммирован на выращивание определенного типа растений. Если вспомнить, что за хранение плана выращивания отвечает специализированный класс `GrowingPlan`, то необходимо обращение контроллера к этому классу для проверки, завершилось ли время выращивания нашего растения.

Синхронизация процессов

Для того чтобы обозначить действия, совершаемые объектом `GrowingPlan` (план выращивания), добавим в диаграмму новую линию `Swimlane` и назовем ее `GrowingPlan`, после чего добавим в поле `GrowingPlan` новое действие `ReturnAllTime`. Теперь для отражения того, что для дальнейшей работы необходимо получить оба времени, и текущее, и полное, введем линию горизонтальной синхронизации.

Конечно, в данном случае можно было бы обойтись и без синхронизации, если создать последовательность действий, отражающую последовательное получение указанных данных. В данном случае мы использовали синхронизацию для показа примера ее применения.

После добавления синхронизации диаграмма получит следующий вид (рис. 7.11).

Линия синхронизации обычно включается в том случае, когда имеются независимые процессоры, которые выполняют задачи параллельно. Если бы `GrowingPlan` был свой процессор, который позволял бы выполнять задачи параллельно, то синхронизация была бы

жизненно необходима. Но в нашем случае она только показывает, что дальнейший переход не может быть осуществлен без обработки данных как таймером, так и планом выращивания, только после того, как оба этих объекта возвратят запрашиваемую информацию, контроллер может произвести ее анализ и принять решение о дальнейших действиях.

Обычно линии синхронизации используются при проектировании процессов клиент-сервер, когда клиенты выдают запросы сразу нескольким серверам, ожидают от них ответа, или при проектировании бизнес-процессов, происходящих в нескольких структурных подразделениях одного офиса, где, например, различные документы обрабатываются в подразделениях, а затем, по завершении создания всех документов, происходит их группировка в один отчет.

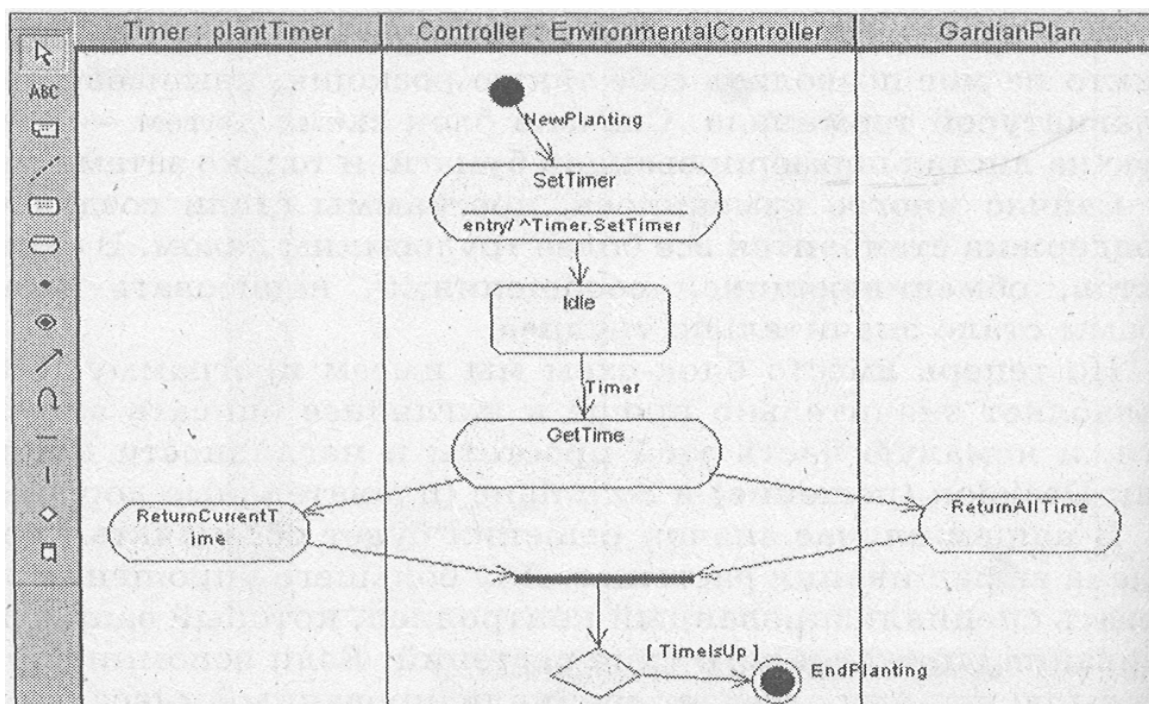


Рис. 7.11 Диаграмма после добавления значка синхронизации

Добавление опроса датчиков

Далее для получения картины работы контроллера введем значок активности `TestingEnvironment`, который, как и в Statechart диаграммах показывает опрос датчиков теплицы.

Алгоритмы опроса датчиков могут быть различными. Выбор алгоритма зависит от проектировщика системы, его опыта и предпочтений. Здесь задача состоит в том, что необходимо в первую очередь создать работоспособную систему и на ее примере научиться пользоваться инструментом. Разберем различные варианты создания датчиков:

Создать активные датчики, которые будут сами посылать сообщения контроллеру при изменении параметров, а контроллер будет активизировать исполнительные устройства.

Опрашивать датчики по одному с немедленной реакцией на отклонения,

Опросить сначала все датчики, а затем, сравнив с необходимыми значениями параметров, заложенными в плане выращивания, активизировать исполнительные устройства для приведения этих параметров в должное состояние.

Первый вариант подразумевает создание нескольких контроллеров определенных параметров среды, которые не связаны друг с другом. С точки зрения объектно-ориентированного проектирования я бы счел этот вариант предпочтительным. Он позволит на основе класса контроллера создавать контроллеры различных параметров и, главное, позволит в дальнейшем расширять систему, добавляя новые контроллеры дополнительных параметров среды практически без затрагивания уже работающих контроллеров.

Второй вариант тоже может быть реализован, но он не далеко уходит от последнего, который, опять же, для простоты создания нашей учебной системы мы и примем к реализации.

По принятому варианту необходим опрос датчиков температуры и рН, затем сравнение с необходимыми значениями и настройка среды путем включения или выключения исполнительных устройств и занесение в протокол данных при изменении параметров.

Окончательный вариант диаграммы

Для отражения этих действий добавим на диаграмму значки действий `TestingEnvironment`, `Writing`, `AdjustingEnvironment`. Мы не будем расшифровывать содержание этих действий, чтобы не загромождать диаграмму, но добавим значки решений, которые направляют действия при изменении (`IsChanged`) среды и при необходимости настройки (`IsAdjust`) среды. При этом должна получиться диаграмма, представленная на рис. 7.12.

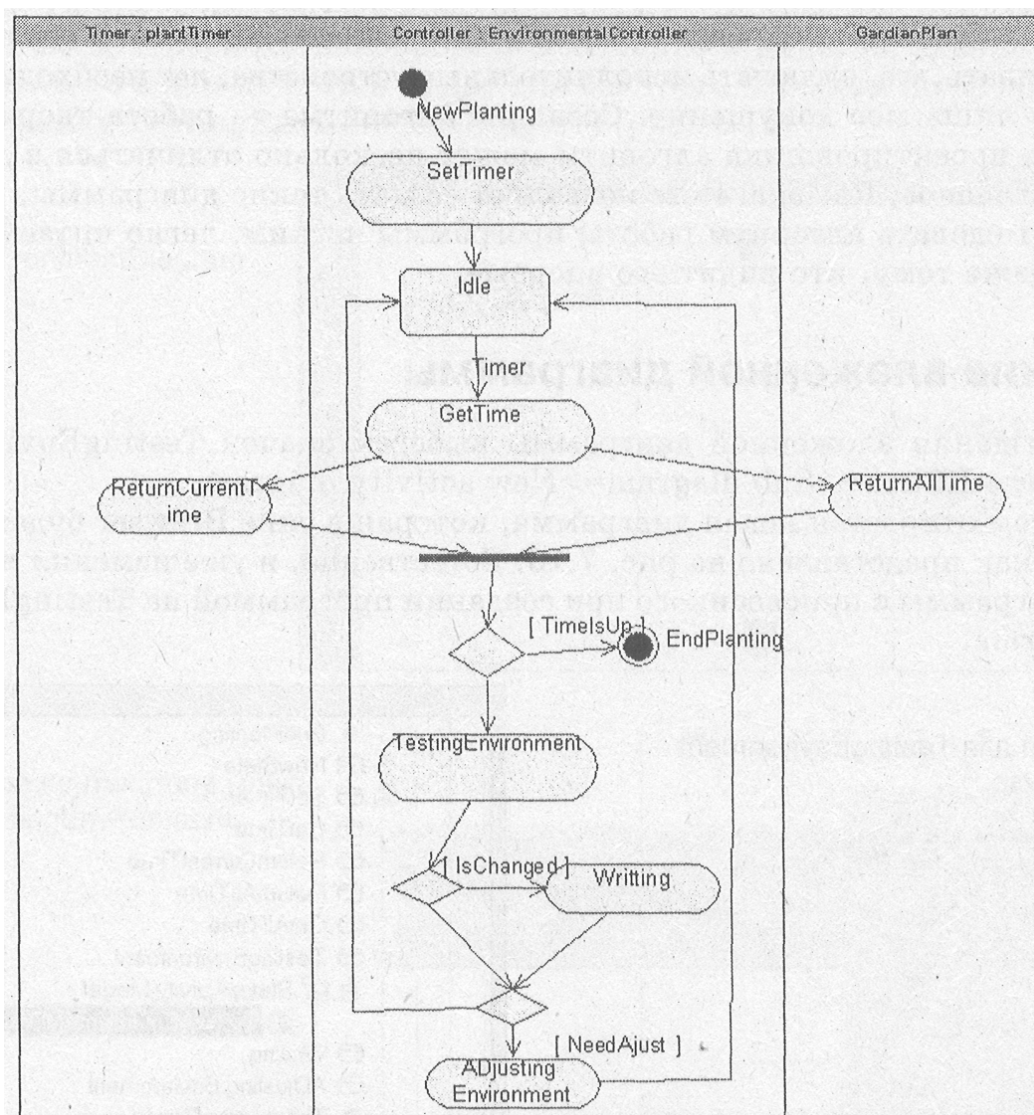


Рис. 7.12. Диаграмма после добавления всех действий

В окончательном варианте на диаграмме получен полный алгоритм работы контроллера от тестирования параметров среды до их настройки с протоколированием.

В отличие от Statechart Diagram, здесь мы не стали подробно расписывать действия TestingEnvironment, Writing, AdjustingEnvironment, которые далее распишем при помощи вложенных диаграмм. Вложенные диаграммы можно создать для каждого значка состояния или активности при помощи нескольких щелчков мыши.

Эта возможность позволяет не загромождать саму диаграмму, а создавать сколько угодно вложенных, что не только скрывает ненужные детали, но и позволяет легко разделить проект для работы команды программистов и также легко собрать проект обратно.

Если вы посмотрите на полученную диаграмму, то заметите, что по алгоритму запись в протокол происходит до проверки на необходимость настройки среды. Это сделано потому, что возможно изменение среды без ее настройки. Например, согласно плану выращивания необходимо скорректировать показатели, а показатели уже изменились в нужную сторону благодаря изменению внешних условий. В этом случае

необходимо записать данные об изменениях, но настраивать, т.е. включать исполнительные устройства, нет необходимости.

Но это лишь мое допущение. Создание алгоритма - работа творческая и у каждого проектировщика алгоритм может несколько отличаться в деталях. Но самое главное, Rational Rose позволяет создать такие диаграммы, которые позволяют сделать алгоритм работы программы четким, легко читаемым, понятным даже тому, кто видит его впервые.

Создание вложенной диаграммы

Для создания вложенной диаграммы выберем значок TestingEnvironment и сделаем RClick=>Sub diagram=>New activity diagram.

При этом откроется новая диаграмма, которая в окне Browser будет выглядеть так, как представлено на рис. 7.13. Естественно, я уже изменил название новой диаграммы с присвоенного при создании программой на TestingEnvironmentDiagram.

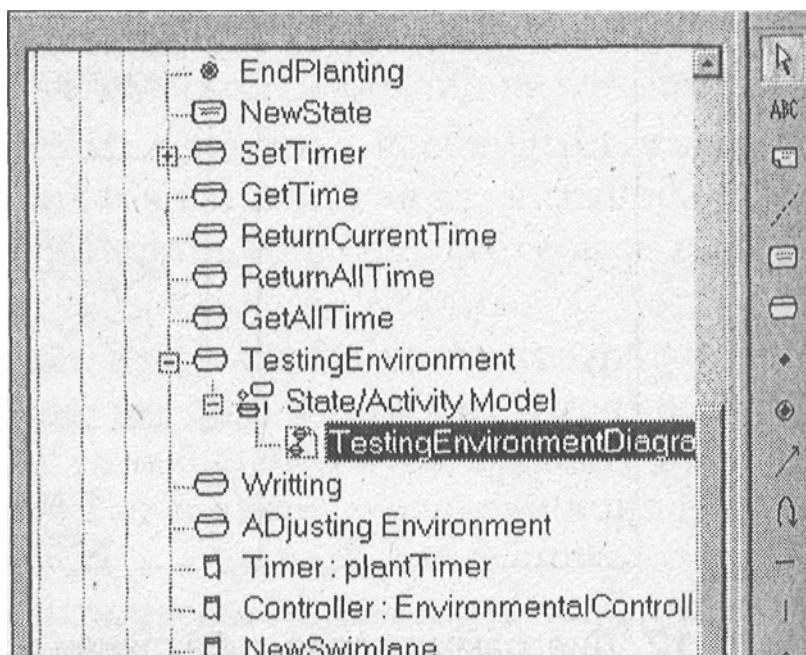


Рис. 7.13 SubDiagram для TestingEnvironment окне Browse

Пользователи, знакомые с таким CASE-средством проектирования бизнес-процессов как BPwin заметят, что здесь нет чрезвычайно необходимой возможности – переноса стрелок с диаграммы верхнего уровня во вновь созданную. Таким образом, разработчик сам должен следить за тем, чтобы эти диаграммы не входили в противоречие между собой по входным и выходным переходам состояний и действий.

Но в этом есть также и некоторые плюсы. Такой подход позволяет не уделять много внимания синтаксически правильной стыковке диаграмм различных уровней, а сосредоточиться на логике процесса. Но все же, воспитанный на строгости языка C++, я предпочел бы дополнительную проверку, которая позволит избежать трудно

обнаруживаемых впоследствии логических ошибок, но с возможностью включения и выключения такой проверки, когда мне это необходимо.

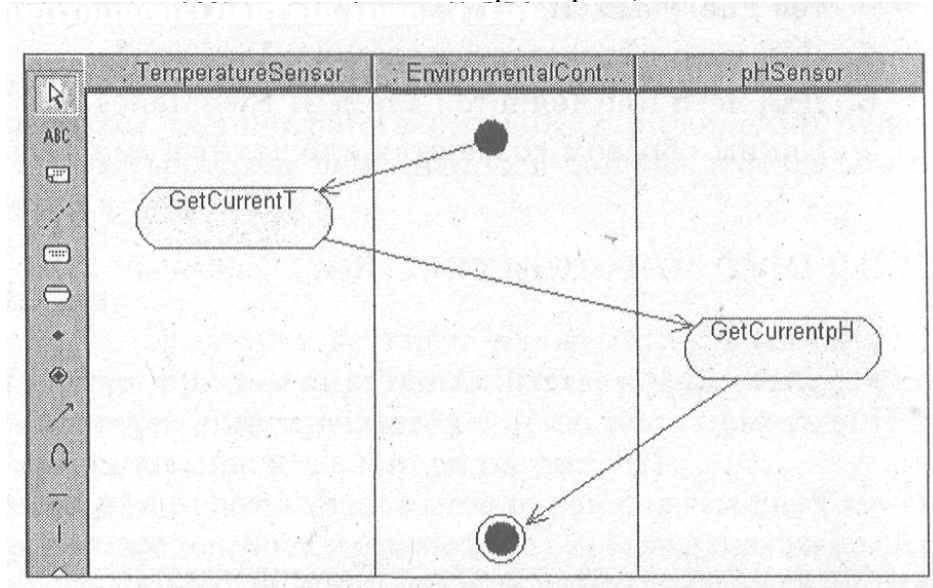


Рис. 7.14 Диаграмма TestingEnvironmentDiagram

Добавим во вновь созданную диаграмму значки начала и конца действия, значки действия получения значения температуры и получения значения pH и, конечно же, используем для помещения этих значков соответствующие Swimlane. Таким образом, должна получиться диаграмма, показанная на рис. 7.14. Если теперь активизировать контекстное меню на значке Testing-Environment, то в нем появится пункт для быстрого перехода на вновь созданную диаграмму TestingEnvironmentDiagram (рис. 7.15).

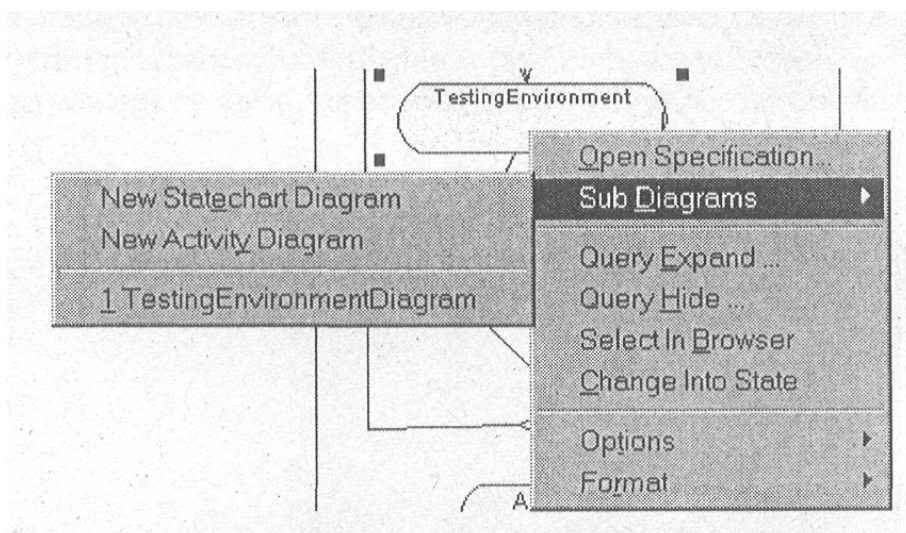


Рис. 7.15 Измененное контекстное меню значка TestingEnvironment

Мы получили вложенную диаграмму, которая связана с основной. Конечно, в нашем случае вложенная диаграмма проста, однако, в других системах такие диаграммы могут быть сложны настолько, что потребуется создание еще более глубокой вложенности. А ее создание в Rational Rose не составит никакого труда. Таким образом, можно

детализировать алгоритм программы, начиная с выполнения крупных задач и постепенно разбивая их на более мелкие, вплоть до выполнения отдельных операторов выбранного языка.

Теперь в качестве тренировки вы можете самостоятельно создать вложенные диаграммы для значков Writing и AdjustingEnvironment. Диаграммы создаются аналогично рассмотренной, а алгоритм работы мы разобрали в предыдущей главе.

Вопросы для повторения

1. Для чего используется диаграмма активности?
2. Какие отличия между диаграммами Activity и Statechart?
3. Каков состав строки инструментов в диаграмме активности?
4. Какие существуют возможности редактирования спецификаций элементов диаграммы?
5. Для чего применяется элемент Decision?
6. Для чего применяется элемент Swimlanes?
7. Каким образом создаются вложенные диаграммы?

Глава 8. Описание взаимодействия при помощи Sequence diagram

Назначение диаграммы

Продолжением создания гидропонной системы будет диаграмма взаимодействия между объектами.

Кроме сценария поведения каждого объекта системы необходимо точно представлять взаимодействие этих объектов между собой, определение клиентов и серверов и порядка обмена сообщений между ними.

Обмен сообщениями происходит в определенной последовательности, и Sequence diagram позволяют получить отражение этого обмена во времени.

В течение работы сложной системы объекты, являющиеся клиентами, посылают друг другу различные сообщения, а объекты, являющиеся серверами, обрабатывают их. В простейшем случае можно рассматривать сообщение как вызов метода какого-либо класса, в более сложных случаях сервер имеет обработчик очереди сообщений, и сообщения им обрабатываются асинхронно, т.е. сервер накапливает несколько сообщений в очереди, если не может обработать их сразу.

На основе приема-передачи сообщений основана многозадачность Windows, а в нашем случае для простоты демонстрации создания приложения будем считать, что сообщения обрабатываются немедленно в той последовательности, в которой они выдаются клиентами.



Создание диаграммы

Создадим Interaction Diagram при помощи Menu:Browse=>Interaction Diagram=>New или значка в строке инструментов.

При этом активизируется окно, представленное на рис. 8.1.

Присвоим полю Title значение Process. Здесь можно выбрать из двух типов диаграмм. Эти диаграммы показывают взаимодействие объектов с разных сторон. Collaboration показывает взаимодействие объектов независимо от времени. Sequence, в свою очередь, представляет собой разворот взаимодействия во времени и отражает последовательность выдачи сообщений клиентами. Создадим Sequence диаграмму.

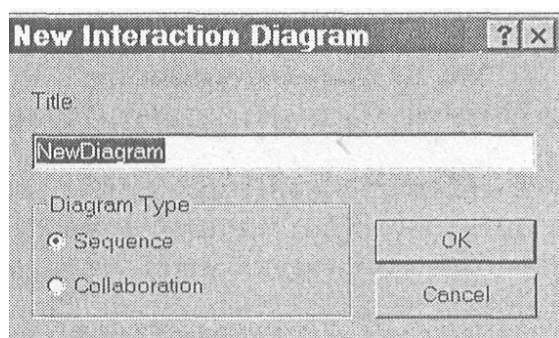


Рис. 8.1 Создание Interaction Diagram

Строка инструментов диаграммы

При переходе в эту диаграмму панель инструментов сменилась на новую, в которой теперь доступны значки Text Box, Note, Anchor to Item, Object, Message, Message to Self. Первые три были рассмотрены для предыдущих диаграмм и здесь выполняют те же функции, поэтому не будем на них останавливаться.

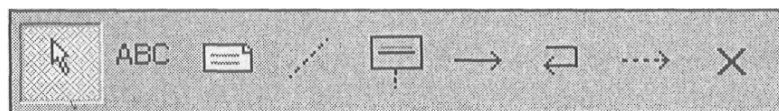


Рис. 8.2 Строка инструментов Sequence diagram

Object (объект)



Значок Object позволяет включить новый объект в диаграмму. Каждый объект является реализацией класса, поэтому в нем можно указать класс, на основе которого он создан.

Message (сообщение)



Значок Message позволяет создать сообщение, передаваемое от одного объекта к другому. Так как все взаимодействие в объектно-ориентированных системах осуществляется при помощи сообщений между объектами, то классы должны позволять отправку или прием сообщений.

Message to Self (сообщение самому себе)



Данный значок позволяет показать, что отправитель сообщения является одновременно и его получателем.

При обмене сообщениями одни классы являются клиентами и принимают сообщения, а другие - серверами и отправляют сообщения. В том случае, когда объект отправляет сообщение самому себе, он одновременно является и сервером и клиентом, что и отражает данный

значок.

Return Message (возврат сообщения)



Данный значок позволяет показать, что происходит возврат управления из вызванной подпрограммы на сервере клиенту.

Destruction Marker (маркер уничтожения)



Данный значок позволяет показать, что происходит уничтожение программного объекта.

Настройка времени жизни объекта

Добавим в диаграмму два новых объекта. Первый назовем Timer, второй -Controller. Для задания названий необходимо сначала выделить значок, а затем просто указать мышкой в середину значка и ввести название.

По двойному нажатию мышки на значке или через RClick=>Open Specification активизируется диалоговое окно параметров объекта. Для объекта Controller оно показано на рис. 8.3.

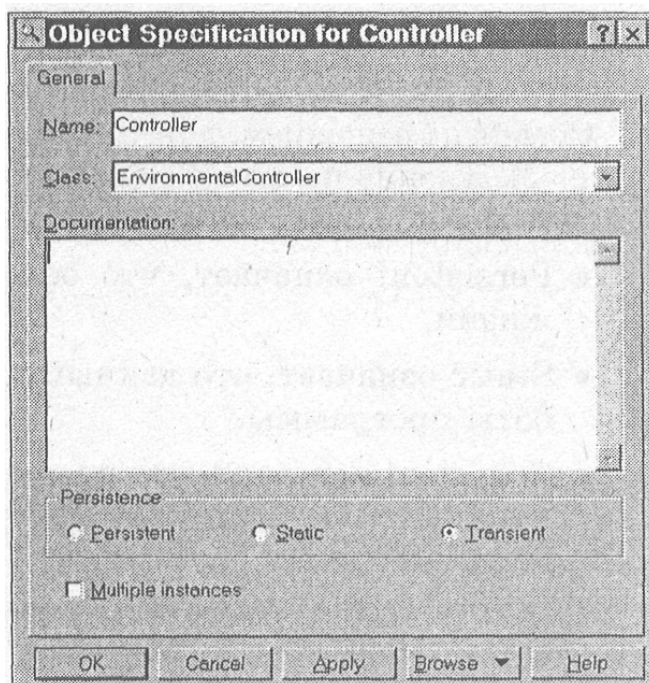


Рис. 8.3 Настройка параметров объекта

Здесь также можно ввести название объекта и класс, реализацией которого является данный объект. Для контроллера мы уже создали класс Environ-mentalController, поэтому его и выберем. Для объекта Timer класс еще не создан, что является нашим упущением, поэтому пока

оставим поле Class пустым. В дальнейшем наилучшим решением для нашей диаграммы будет добавление класса Timer в диаграмму классов.

Для реализации объектов в окне параметров доступна опция Persistence, отражающая время жизни объекта. Время жизни объекта – это время от его создания до уничтожения. Обычно объекты существуют в пределах их области видимости и автоматически уничтожаются системой, когда выходят за эту область. Это происходит в тех случаях, когда, например, в подпрограмме определяется объект класса, который автоматически уничтожается при завершении подпрограммы.

В языке C++ можно создать и удалить объект, не дожидаясь, когда это сделает система, при помощи команд new и delete. Обычно к такому варианту прибегают еще и для уменьшения исполняемого файла. Известно, что при создании программы на Visual C++ такое определение:

```
char s[10000];
```

приведет к увеличению исполняемого файла на 10000 байт, так как для этой переменной будет зарезервировано место в области данных программы. Однако создание массива при помощи операторов:

```
char * s;
```

```
s = new char[10000];
```

```
// работа с массивом
```

практически не приведет к увеличению размера исполняемого файла, хотя подразумевает, что программист освободит выделенную память до завершения программы при помощи оператора:

```
delete s[];
```

иначе она будет недоступна для системы в целом.

Обычным решением, для того чтобы не забыть освободить выделенную память, будет выделение памяти в конструкторе класса и освобождение ее в деструкторе. Таким образом, можно настроить следующие параметры жизни объекта:

Persistent означает, что область видимости объекта превышает время жизни.

Static означает, что данный элемент существует на всем протяжении работы программы.

Transient означает, что время жизни объекта и область видимости совпадают. Этот вариант присваивается по умолчанию и нам он подходит для обоих случаев.

Для того чтобы показать, что эти свойства описывают состояние всех объектов данного класса, можно указать Multiple Instance, но для нашего случая это не обязательно.

Совет

Для непосредственного указания момента уничтожения объекта используйте значок Destruction Marker.

Создание класса из окна настройки параметров

Теперь вспомним, что для нашего объекта Timer не указан класс, на основе которого этот объект создается. Зайдем в окно свойств объекта,

перейдем в поле Class и нажмем клавишу «стрелка вверх». При этом сразу открывается окно свойств класса (рис. 8.4). Пока не будем вникать в эти свойства, они будут рассмотрены подробно позднее в части III.

Здесь нам нужно ввести только имя plantTimer (это будет класс, отвечающий за расчет текущего времени роста растений) и нажать ОК.

Теперь, по своему усмотрению, можно скрыть или показать наименование класса на значке объекта, как показано на рис. 8.5.

Создание сообщений

Перейдем непосредственно к сообщениям. Выбираем стрелку Object message и соединяем пунктирные линии контроллера и таймера. Затем еще раз, но от таймера к контроллеру ниже первого сообщения. В данном случае мы подразумеваем, что самым первым сообщением будет указание таймеру времени, через которое происходит активизация и передача адреса обратного вызова таймером. Вторым сообщением будет сообщение от таймера, что указанный промежуток времени истек.

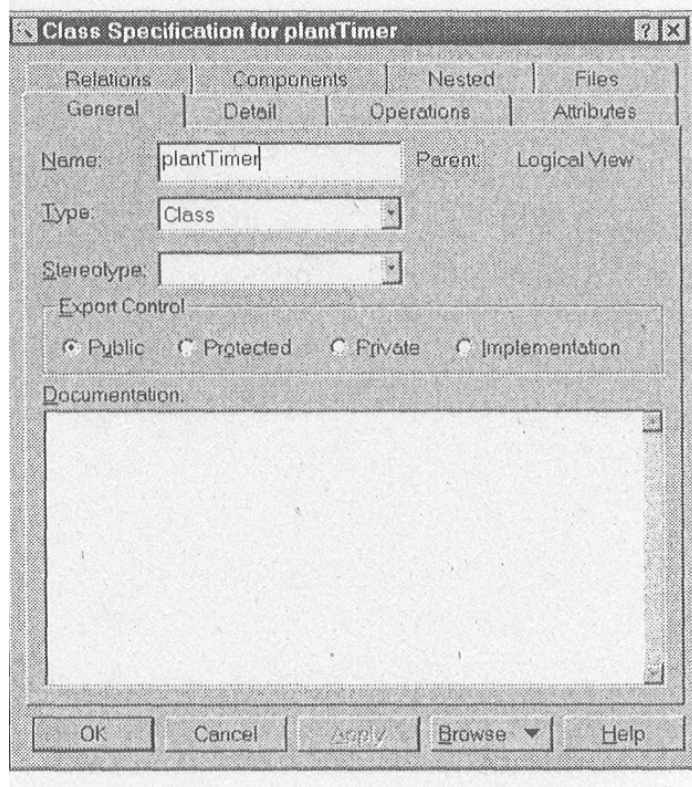


Рис. 8.4 Свойства класса

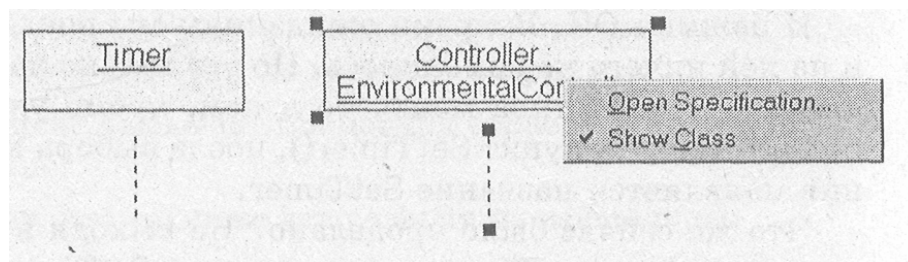


Рис. 8.5 Показ и скрывание названия класса

В РС совместимых компьютерах системные часы дают возможность

устанавливать минимальное время таймера 1/60 секунды. Но для нашей системы такая точность совершенно не нужна. Растения растут медленно, а температура или pH падает или увеличивается на ощутимую величину, по крайней мере за несколько минут. Поэтому для нашего таймера приемлемой величиной будет активизация раз в минуту. Однако позволяя устанавливать промежуток времени произвольно, мы, во-первых, придаем нашей системе большую гибкость, и во-вторых, что не маловажно, оставляем себе лазейку для отладки системы. Можем установить заведомо большее или меньшее время для ускоренного тестирования процесса.

Выделим первое сообщение и вызовем контекстное меню (RClick). Здесь мы увидим уже знакомый пункт Open Specification и пункт New operation.

При помощи последнего можно добавить новую операцию (метод) к классу, из которого создан объект. Выберем этот пункт и введем название операции SetTimer (рис. 8.6).

Замечание

Подробно редактирование спецификаций операции будет рассмотрено в главе 11.

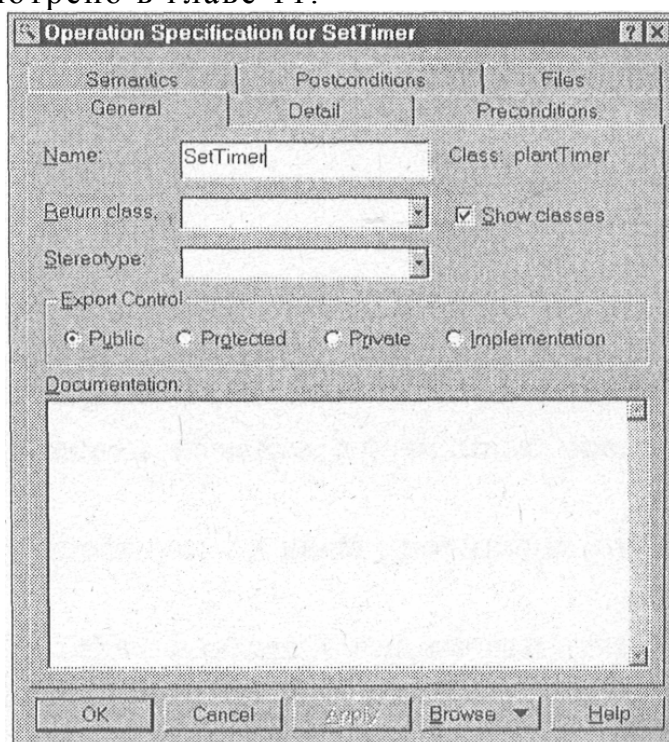


Рис. 8.6 Редактирование названия операции

И нажмем ОК. Вопреки ожиданиям мы попали обратно в нашу диаграмму и на ней ничего не изменилось. Но это только на первый взгляд. Если еще раз вызвать контекстное меню, то в нем, кроме уже описанных пунктов, будет присутствовать пункт SetTimerQ, после выбора которого над стрелкой сообщения появляется название SetTimer.

Что же сейчас было сделано? Не выходя в диаграмму классов, мы добавили новый класс Timer и метод к нему SetTimer. В этом и проявляется мощь Rational Rose, что диаграммы связаны. Классы и

методы к ним добавляются в том месте, где это наиболее логично и понятно. Причем, если внести изменения в описание класса на любой диаграмме, то эти изменения тут же появятся на остальных диаграммах модели.

Свойства сообщений

Теперь разберемся со свойствами только что введенного сообщения. При вызове SetTimer()=>RClick=>Detail мы попадаем в диалоговое окно, показанное на рис. 8.7.

Здесь мы видим две группы радио-кнопок (кнопок с зависимой фиксацией):

1. Synchronization определяет порядок обмена сообщениями и может быть выбрана из следующих вариантов:

Simple - простая посылка сообщения;

Synchronous – операция происходит только в том случае, когда клиент посылает сообщение, а сервер может принять сообщение клиента;

Balking – операция происходит только в том случае, когда сервер готов немедленно принять сообщение, если сервер не готов к приему, клиент не выдает сообщение;

Timeout - клиент отказывается от выдачи сообщения, если сервер в течение определенного времени не может его принять;

Procedure Call - клиент вызывает процедуру сервера и полностью передает ему управление;

Return -- определяет, что происходит возврат из процедуры;

Asynchronous - клиент выдает сообщение, и, не ожидая ответа сервера, продолжает выполнение своего программного кода;

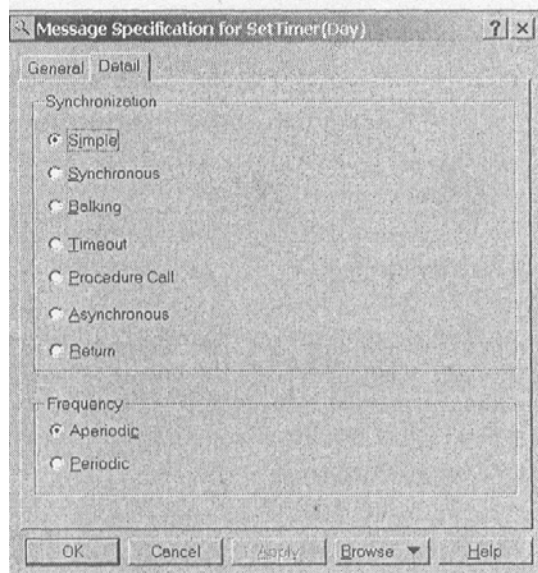


Рис. 8.7 Установка свойств сообщения

Для каждого вида операций стрелка сообщения изменяется в соответствии с рис. 8.8.

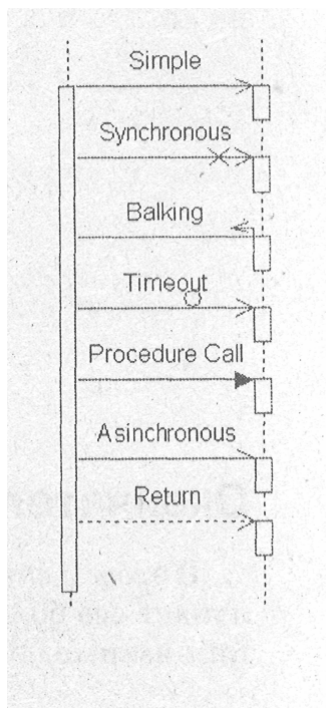


Рис. 8.8 Различные виды сообщений

2. Frequency • - определяет частоту обмена сообщениями:

Periodic – сообщения поступают от клиента с заданной периодичностью;

Aperiodic - сообщения поступают от клиента нерегулярно.

Совет

Для того чтобы удалить неправильно введенное сообщение с диаграммы, нужно нажать Ctrl+D.

Для операции OnTimer() подходит установка Asynchronous и Periodic, потому что классу таймера все равно, что будет делать с его сообщением другой класс, причем в нашем случае Timer обслуживает только класс контроллера, а в реальной системе возможно понадобится вызывать прерывание для нескольких объектов.

Добавление класса plantDayHour

Если посмотреть на созданную ранее диаграмму состояний (рис. 6.12), то после получения сообщения от таймера происходит анализ времени, прошедшего от начала посадки растений. Этот анализ неплохо было бы переложить на плечи объекта специального класса, например, класса plantDayHour. Этот класс будет отвечать за отсчет текущего дня и часа. Добавим объект m_plantDayHour и создадим для него новый класс plantDayHour. Соединим новыми сообщениями UpdateTime() и GetCurrentTime() объект контроллера и m_plantDayHour и получим рис. 8.9.

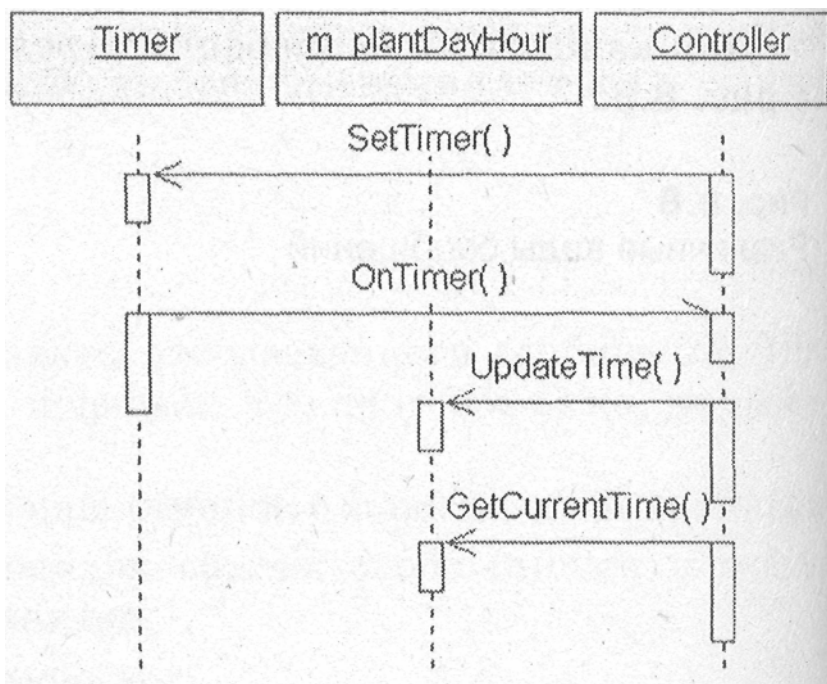


Рис. 8.9 Добавленный объект m_plantDayHour

Окончательный вариант диаграммы

В этом и заключается процесс визуального проектирования, что система получает все большую детализацию в процессе добавления диаграмм и детализации взаимодействия между ними.

Теперь, когда все необходимые возможности изучены, можно добавить сообщения между датчиками и исполнительными устройствами. Начало этого процесса показано на рис. 8.10.

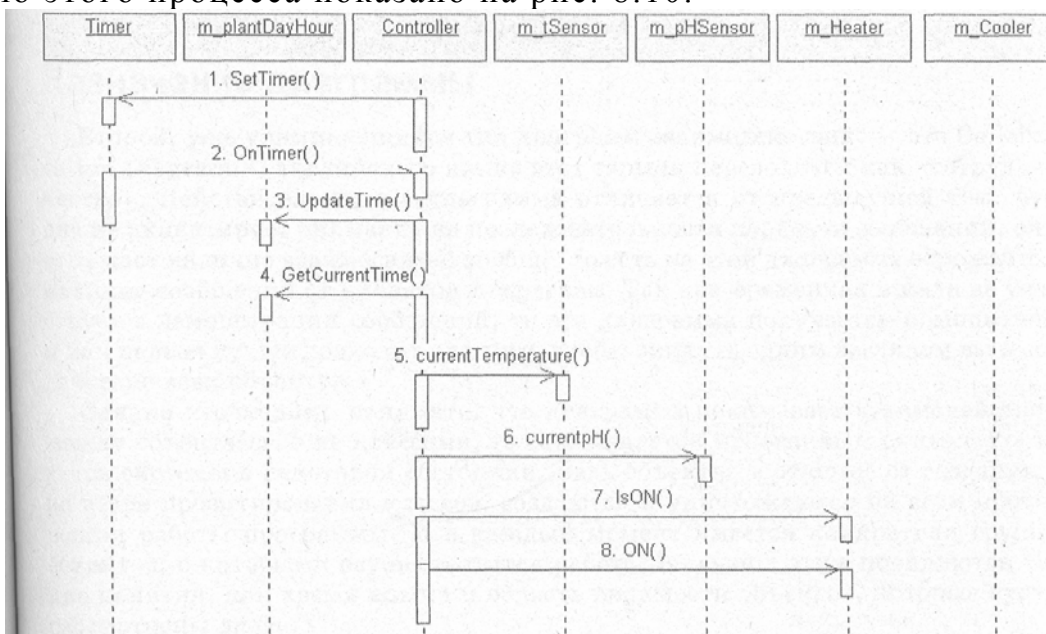


Рис. 8.10. Добавление в диаграмму датчиков и исполнительных устройств

На этом рисунке добавлены датчики, нагреватель и вентилятор. Добавить WaterTank, NutrientTank и Light вы можете самостоятельно.

Совет

Для того чтобы пронумеровать сообщения по порядку, необходимо сделать Menu: Tools=>Options=>Diagram:=>Sequence Numbering=ON.

Вопросы для повторения

1. Для чего предназначена диаграмма Sequence?
2. Какие виды сообщений позволяет отразить диаграмма?
3. Как настроить отображение времени жизни объекта.
4. Как создать классы, не выходя из диаграммы?
5. Какой порядок обмена сообщениями может быть задан?
6. Какая может быть задана частота обмена сообщениями?

Глава 9. Описание взаимодействия с помощью Collaboration Diagram

Назначение диаграммы


Второй, уже упоминавшийся тип диаграмм взаимодействия, – это Collaboration Diagram. С английского языка этот термин переводится как «сотрудничество». Действительно, эта диаграмма отличается от предыдущей тем, что она не акцентирует внимание на последовательности передачи сообщений, она отражает наличие взаимосвязей вообще, то есть на этой диаграмме отражается наличие сообщений от клиентов к серверам. Так как временная шкала не участвует в демонстрации сообщений, то эта диаграмма получается компактней и как нельзя лучше подходит для того, чтобы окинуть одним взглядом взаимодействие всех объектов.

Однако необходимо понимать, что диаграмма показывает взаимодействие между объектами, а не классами, то есть является мгновенным снимком объектов системы в некотором состоянии. Ведь объекты, в отличие от созданных на этапе проектирования классов, создаются и уничтожаются на всем протяжении работы программы. И в каждый момент имеется конкретная группа объектов, с которыми осуществляется работа. В связи с этим появляются такие понятия, как время жизни и область видимости объектов, которые будут рассмотрены далее.

Создание диаграммы

Разработчики Rational Rose заложили удобную возможность создания на основе диаграммы Sequence диаграммы Collaboration и наоборот.

А так как диаграмма Sequence у нас уже есть, то создадим на ее основе Collaboration. Для этого, находясь в диаграмме, сделаем следующее: Menu: Browse=>Create Collaboration diagram.

Вы также можете создать диаграмму, нажав значок Interaction diagram. 

При этом для создания необходимого типа диаграммы в диалоговом окне (рис. 9.1) выберите тип диаграммы Collaboration.

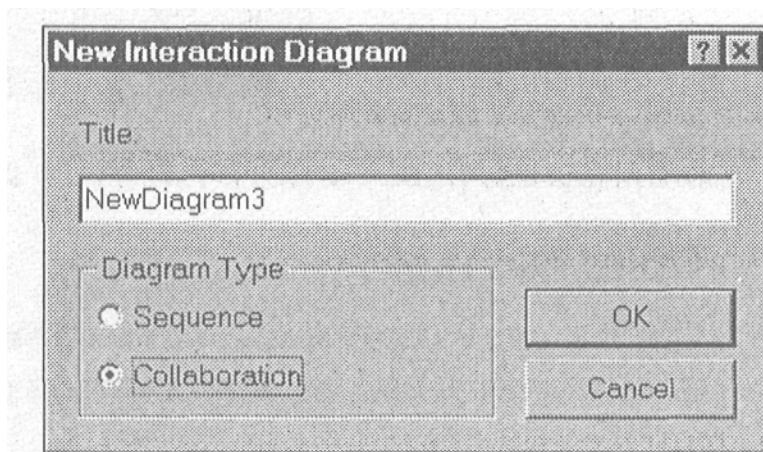


Рис. 9.1 Окно выбора создания Collaboration диаграммы

Однако в этом случае будет создана пустая диаграмма, и уже созданные объекты не будут туда перенесены. Поэтому для нас лучше подходит первый вариант.

Будет создана диаграмма, представляющая, на первый взгляд, нагромождение значков. Но после того как вы ее «растяните» мышкой, диаграмма приобретет вполне читаемый вид (рис. 9.2).

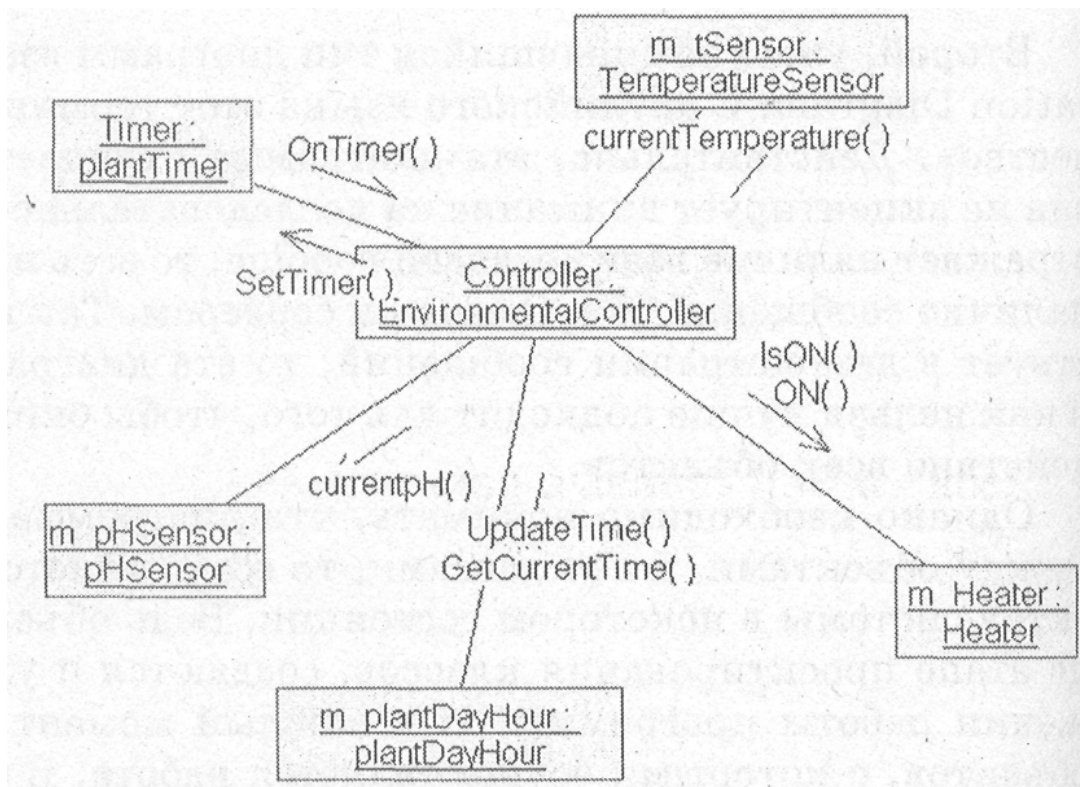


Рис. 9.2 Автоматически созданная Collaboration диаграмма

На этой диаграмме объект Controller – центральный, и все сообщения поступают к нему или исходят от него, что совершенно правильно отражает наше представление о системе. Взаимодействия между контроллером и остальными устройствами изображаются линиями с добавленными стрелками, аналогичными тем, которые изображаются на

Sequence Diagram. Но не трудно заметить, что все сообщения одного направления собираются вместе и даются как подпись к одной стрелке, таким образом, получаем полную картину взаимодействия.

Строка инструментов

При активизации диаграммы строка инструментов приобретает следующий вид (рис. 9.3).

Как и в предыдущих диаграммах, мы не будем рассматривать общие для остальных диаграмм инструменты, а рассмотрим только новые.

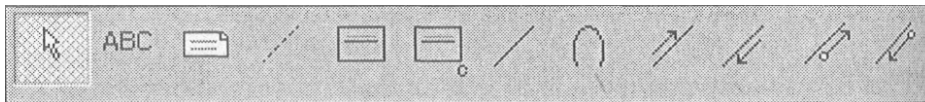


Рис. 9.3 Строка инструментов Collaboration diagram

Object (объект)



Object позволяет создавать объекты, которые имеют состояния, поведение и индивидуальны. Каждый объект на диаграмме показывает реализацию некоторого класса.

Class Instance (реализацией класса)



Class Instance позволяет добавлять абстрактные реализации класса в диаграмму. В чем разница между объектом и абстрактной реализацией класса?

При добавлении значка на диаграмму внешне эти значки не отличаются. Отличие в их внутреннем содержании. Объект подразумевает настройку его времени жизни и других свойств, присущих только конкретному объекту. Абстрактная реализация класса не позволяет изменять эти свойства и предназначена только для показа взаимодействия.

Object Link (связь объекта)



Взаимодействия объектов отражается посредством показа их связей. Существование связей между двумя классами символизирует взаимодействие между их реализациями (объектами, созданными на основе этих классов). При этом один объект может посылать сообщение другому объекту.

Link To Self (связь с самим собой)



Так как объекты могут посылать сообщения самим себе, то данный значок показывает, что объект имеет обратную связь с самим собой.

Link Message (передача сообщения)



Link Message позволяет отразить связь, которая подразумевает обязательную передачу сообщения.

Reverse Link Message (обратная передача сообщения)



Reverse Link Message позволяет отразить связь, которая подразумевает обязательную передачу сообщения аналогично предыдущему пункту, но в обратном направлении.

Data Flow (поток данных)



Data Flow позволяет отразить связь показывающую, что происходит передача данных от одного объекта другому.

Reverse Data Flow (обратный поток данных)



Как и предыдущий значок, позволяет отразить связь, показывающую, что происходит передача данных от одного объекта к другому, но в обратном направлении.

Создание объекта

Добавим в диаграмму новый объект m_WaterTank.

Совет

Создать объект нужного класса можно при помощи «буксировки» мышкой этого класса из окна Browse в окно текущей диаграммы, конечно, после этого ему необходимо присвоить имя.

Контекстное меню объекта

После добавления можно посмотреть, что нам предлагает контекстное меню (рис. 9.4).

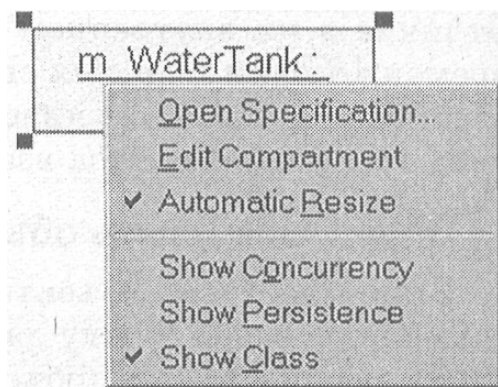


Рис. 9.4 Контекстное меню объекта

Коротко перечислим возможности, предоставляемые посредством данного меню.

Open Specification -- редактирование спецификаций объекта;

Edit Compartment активизирует диалоговое окно показа дополнительной информации об объекте. Содержание такой информации зависит от типа объекта;

Замечание

Подробнее работа с данным пунктом будет рассмотрена в главе 2, в разделе, посвященном меню Options.

Automatic Resize позволяет устанавливать автоматическую настройку размера объекта по длине содержащегося в нем текста;

Show Concurrency позволяет включить показ на данном значке типа согласования при создании много потоковой программы. Данный тип определен в классе;

Show Persistence позволяет показать на диаграмме время жизни объекта;

Show Class позволяет показать на диаграмме имя класса.

Создание связи между объектами

Соединим новый объект с объектом Controller линией Object Link. Для того чтобы показать, что от контроллера будут поступать сообщения, нажмем Link Message. После того как курсор приобретет вид креста, укажем на линию. Рядом с ней появится стрелка сообщения, для которой можно ввести свойства, аналогичные свойствам сообщения на Sequence Diagram, при помощи контекстного меню.

Для того чтобы показать, что объект m_WaterTank возвращает данные по запросам контроллера IsON (включен ли) и IsOFF (выключен ли), создадим указатель Reverse Data Flow. Для этого нажмем на указанный значок, и после того, как курсор примет форму креста, укажем на стрелку сообщения, но не на линию Link Message. После заполнения свойств сообщения должно получиться примерно следующее (рис. 9.5).

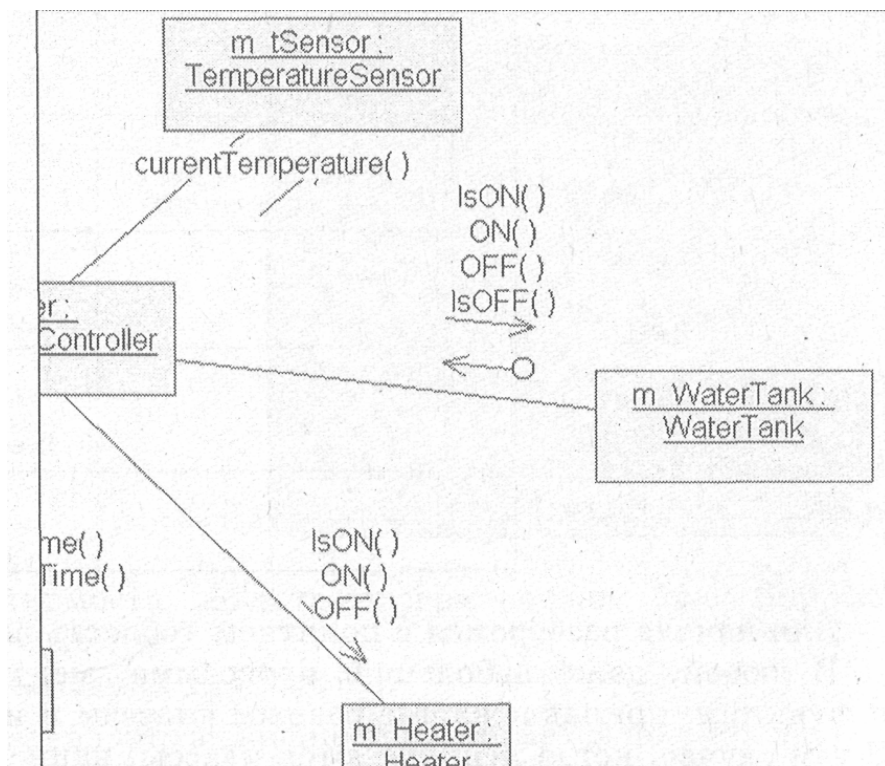


Рис. 9.5 Указание передачи данных

Указание передачи данных является справочной информацией и показывает нам, что как минимум одно из сообщений должно возвращать значение, обрабатываемое контроллером.

Совет

Редактируйте название посредством окна Browse, а не в диаграмме.

Автоматический перенос данных в диаграмму Sequence

Если вы переключитесь в диаграмму Sequence, то увидите рис. 9.6. Это Rational Rose автоматически перенес введенные нами изменения в диаграмму Sequence, избавив нас тем самым от монотонного труда по переносу данных из одной диаграммы в другую. Однако необходимо расставить сообщения в нужном порядке, о котором программа, естественно, не догадывается и имеет в виду, что в каком порядке сообщения введены, в таком порядке они и выполняются.

Настройка области видимости объектов

У каждой связи Link Message есть соответствующие свойства, которые позволяют настроить область видимости для связанных объектов.

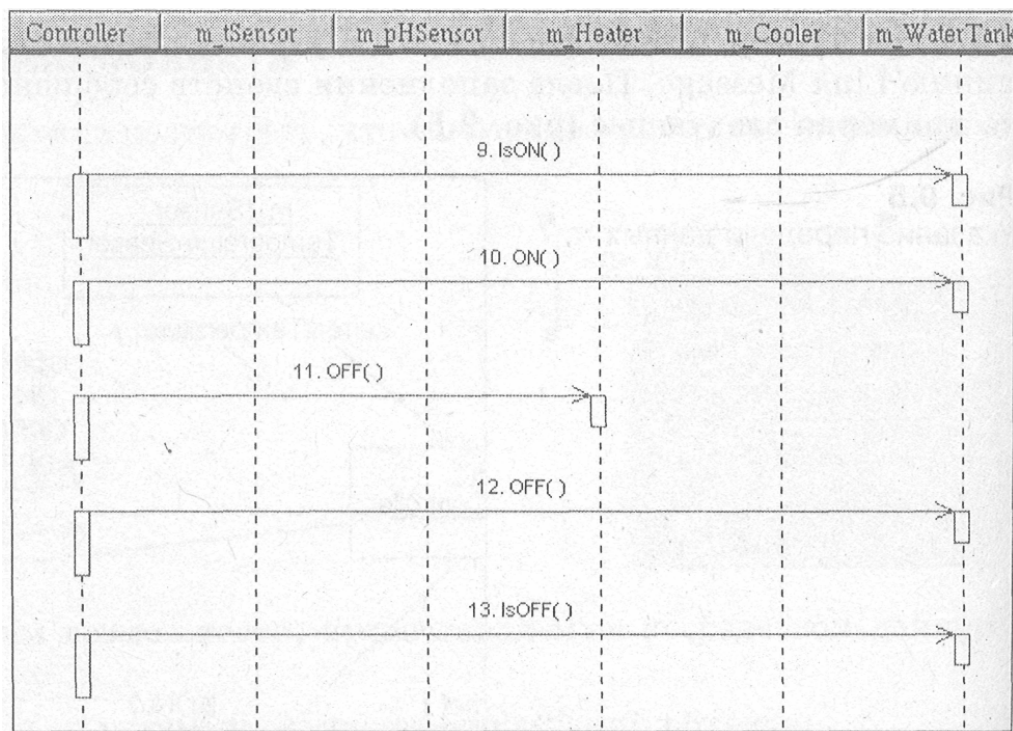


Рис. 9.6 Автоматическое заполнение Sequence Diagram

Для начала разберемся с понятием «область видимости».

В любой, даже небольшой программе, могут существовать переменные и функции, принадлежащие разным классам и имеющие одинаковые имена. В том случае, когда используются классы, написанные разными людьми, которые возможно никогда не видели друг друга, например, при использовании библиотек классов, трудно добиться уникальности имен. Читатель, знакомый с языком программирования СУБД FoxPro, знает, как трудно найти ошибку при изменении незначительной переменной в подпрограмме. Для тех, кто не знаком с этим языком поясню, что в этом языке область видимости простирается от определения переменной и далее. Таким образом все определенные переменные в головной программе по умолчанию видны в подпрограммах.

Это удобно для того, чтобы не передавать параметры. Но представьте, что в некотором цикле по переменной *i* вызывается функция, которая написана другим программистом и тоже использует свой цикл по переменной *i*. В этом случае работа программы будет нарушена с непредсказуемыми последствиями.

К счастью, язык C значительно строже относится к области видимости и переменные, определенные в подпрограмме или даже выделенные блоком в самой программе, будут, скрыты от изменения в других частях программы. Однако необходимо заботиться о передаче этих переменных как параметров.

При создании класса в C++ область видимости переменных имеет нечто промежуточное между описанными выше двумя вариантами. Переменные, определенные как члены класса, видны всеми его методами так, как если бы они были определены глобально, и, в то же время, они могут быть скрыты от внешнего случайного или даже намеренного

изменения, если они определены в разделе `private` или `protected`. Подробнее об области видимости в C++ можно прочитать, например, в [2].

Для того чтобы задать область видимости, перейдем в `RClick=>Open Specification` и попадем в диалоговое окно, представленное на рис. 9.7.

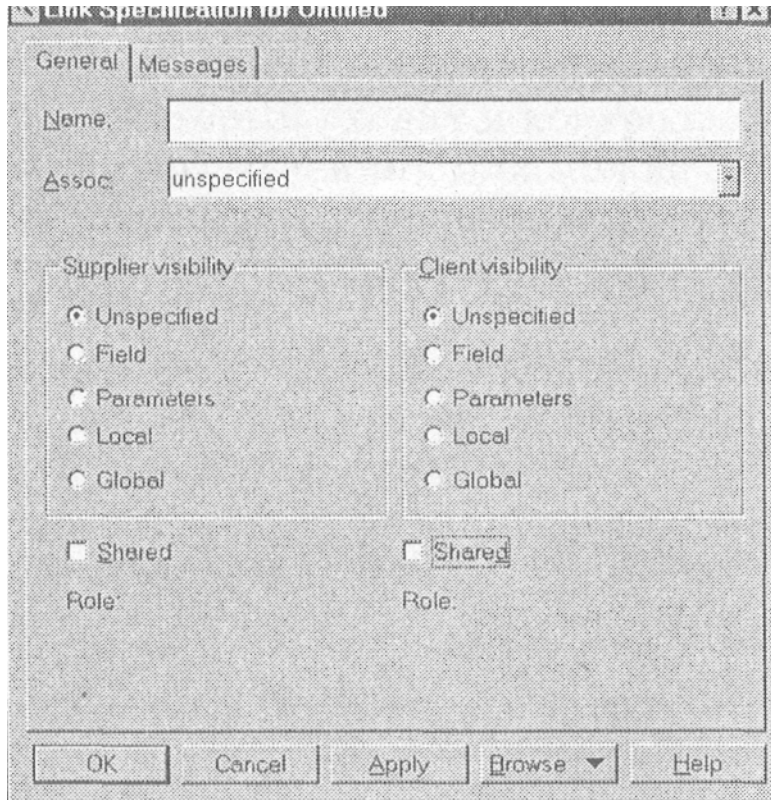


Рис. 9.7 Задание области видимости для Link Message

Для задания области видимости объекта-сервера служит блок `Supplier visibility`, клиента – `Client visibility`.

В этих блоках доступны значения для выбора:

`Unspecified` – не определено, это значение присваивается по умолчанию;

`Field` - объект включен в другой объект;

`Parameter` – объект передается параметром в другой объект;

`Local` – объект локально определен в границах другого объекта;

`Global` -- объект глобален по отношению к другому объекту.

При изменении области видимости на концах соединяющей линии появляется квадратик с указанной областью видимости. Здесь же можно установить флажок, показывающий, что объект используется совместно (`Shared`).

Область видимости можно изменить из контекстного меню, причем, изменяется та сторона `Link` линии, к которой был ближе курсор мыши при активизации контекстного меню.

Изменение свойств сообщений

Закладка Messages (рис. 9.8) позволяет изменять свойства сообщений для выбранной связи. Причем можно добавлять методы прямо из этого окна как для клиента, так и для сервера. На рисунке показано окно списка сообщений с активизированным контекстным меню.

Однако нужно помнить, что добавление методов в этом окне, не добавит их в соответствующие классы. Это, с одной стороны, не совсем удобно, так как нужно добавлять методы непосредственно в классе, но становится просто необходимо, если методы наследуются из базовых классов, и добавлять их в дочерние вы не собираетесь.

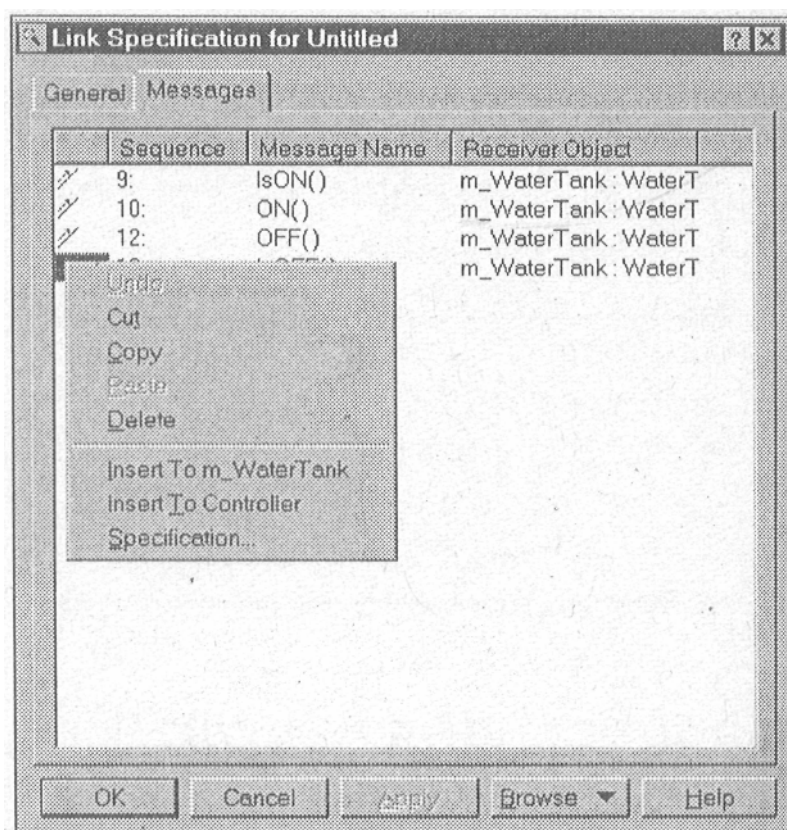


Рис. 9.8 Доступ к списку сообщений

Теперь можно добавить объект Light (лампочка) и указать область видимости для остальных Link линий. Должно получиться следующее (рис. 9.9).

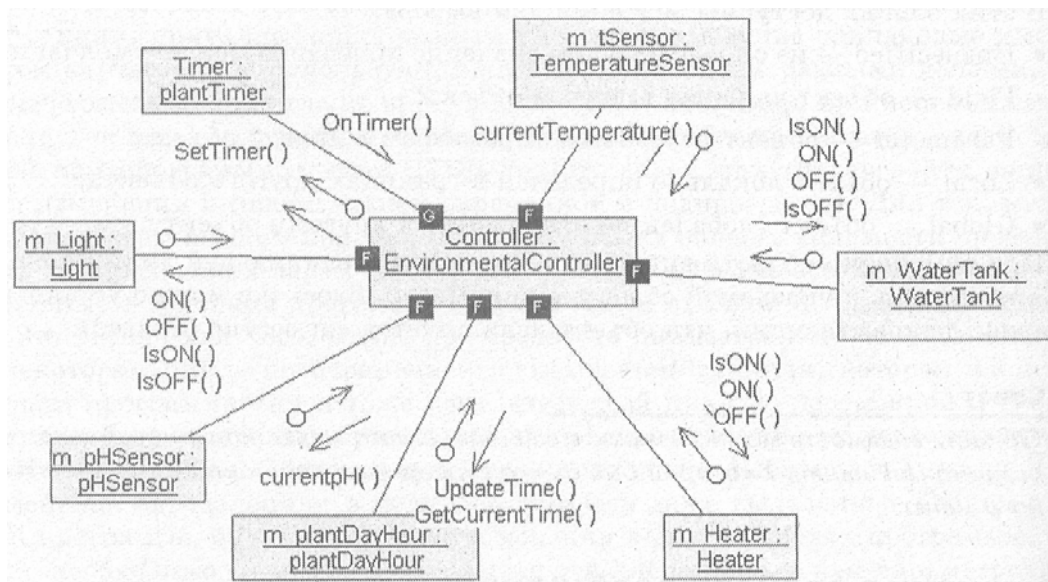


Рис. 9.9. Итоговая Collaboration диаграмма

Еще раз об области видимости

Если внимательно присмотреться, то для всех объектов, кроме таймера, я поставил область видимости Field, а на объекте Timer стоит область видимости Global. Это не случайно.

Область видимости конкретных объектов определяется при проектировании структуры программы, и в данном случае я все объекты сделал полями в объекте контроллера, а таймер будет внешним по отношению к контроллеру, поэтому я указал его как глобальный. Под глобальным объектом я подразумеваю объект, созданный в объекте-контейнере, в который входит и контроллер. Таким объектом может быть Application (объект приложения), или можно создать объект GreenHouse (теплица), который описывает физическое наличие в нашей теплице различных устройств, при этом все исполнительные и измерительные устройства входили бы в этот объект.

В последнем случае можно было бы использовать массивы устройств, для того чтобы пользователь системы мог указывать количество конкретных управляющих устройств (в том числе и контроллеров), находящихся в теплице. Но так как задача больше учебная, чем практическая, то используем классы минимальной сложности и наибольшей иллюстративности.

Вопросы для повторения

1. Какое назначение диаграммы Collaboration?
2. Как перенести данные между диаграммами Sequence и Collaboration?
3. Какие инструменты доступны в диаграмме?
4. Какие команды доступны через контекстное меню объектов?
5. Какие есть возможности настройки области видимости объектов?
6. Какие есть возможности настройки свойств сообщений?

Глава 10. Диаграмма компонентов

Назначение диаграммы

Component diagram (диаграмма компонентов) позволяет создать физическое отражение текущей модели. Диаграмма компонентов показывает организацию и взаимосвязи программных компонентов, представленных в исходном коде, двоичных или выполняемых файлах. Связи в данном типе диаграммы представляют зависимости одного компонента от другого и имеют специальное отображение через значок «зависимости».

Также данный тип диаграммы позволяет получить представление о поведении компонентов по предоставляемому ими интерфейсу. Интерфейс показывает взаимодействие компонентов, и хотя значки интерфейса принадлежат логическому представлению системы, они могут присутствовать и на диаграмме компонентов.

В текущей модели может быть создано несколько диаграмм компонентов для отражения пакетов, компонентов верхнего уровня или описания содержимого каждого пакета компонентов. В последнем случае диаграмма компонентов принадлежит тому пакету, для которого отражено содержимое.

Замечания по созданию диаграммы компонентов

Сейчас будет несколько преждевременно создавать полноценную диаграмму компонентов системы управления тепличным хозяйством так как еще не определены все связи классов и структура наследования.

Однако для небольшой системы, каковой является тепличное хозяйство, состоящей из одного выполняемого файла, разработка такой диаграммы, по моему мнению, вообще не является целесообразной. Но для того чтобы ознакомиться с возможностями диаграммы компонентов и не отвлекаться на нее в дальнейшем я предлагаю завершить предварительное проектирование системы именно этой диаграммой.

Только предварительное, потому что жизнь сложнее всяких правил. Существуют стратегические и тактические решения, о которых мы поговорим в главе 16, и которые иногда противоречат друг другу.

На всем протяжении проектирования системы, вплоть до выхода готового программного продукта, в диаграммы будут вноситься изменения. Мы не будем полностью отражать этот итерационный процесс, поэтому некоторые принятые во время проектирования системы решения могут не совпадать с полученным в конце кодом. Это нормально. Во время разработки необходимо просто отразить эти изменения на созданных ранее диаграммах, что вы можете сделать самостоятельно. А пока создадим диаграмму компонентов лишь для нескольких классов, чтобы получить практику работы с данным типом диаграмм.

В Rational Rose заложена прекрасная возможность работы с программными библиотеками. Причем можно как разрабатывать библиотеки, так и пользоваться уже готовыми. Для этого необходимо лишь указать, какие классы в каких компонентах будут находиться.

При разработке программного проекта разделение классов по компонентам является серьезной задачей. Для того чтобы обеспечить минимальные трудозатраты на разработку и сопровождение, тесно связанные между собой классы собираются в библиотеки DLL, OCX и т.п. Этим обычно занимаются системные аналитики, которые проектируют структуру программного обеспечения, Rational Rose предоставляет все возможности для такого проектирования.

Создание диаграммы компонентов



Для создания диаграммы используйте меню Browse=>Component diagram или воспользоваться значком Component diagram на панели инструментов.

При этом будет активизировано диалоговое окно выбора диаграммы, посредством которого пользователь может создавать, удалять, переименовывать диаграммы.

Строка инструментов

При активизации диаграммы строка инструментов приобретает следующий вид (рис. 10.1).



Рис. 1.0.1 Строка инструментов Component диаграммы

Здесь также присутствуют стандартные инструменты, уже рассмотренные ранее, это Selection Tool, Note, Text Box, Anchor Note to Item. Они выполняют те же функции, что и в других диаграммах, поэтому не будем на них останавливаться и разберем другие инструменты диаграммы.

Component (компонент)



Значок Component позволяет показать создание компонента. Компонент представляет собой модуль программного обеспечения, такой как исходный код, двоичный файл, выполняемый файл, динамически подключаемые библиотеки и т.д.

Взаимодействие элементов представляется на диаграмме одним или не-сколькими значками интерфейса.

Компоненты также могут использоваться для показа взаимосвязи модулей на этапе компиляции или выполнения программы, а также

показывать, какие классы используются для создания определенных компонентов.

В связи с тем что система может состоять из модулей различного типа, пользователь может использовать стереотипы для определения этих различий, причем изменение стереотипа часто ведет к изменению графического отображения компонента на диаграмме.

Замечание

В отличие от других диаграмм, при создании элементов диаграммы компонентов в модель всегда добавляется один тип элементов - Component, который отличается только стереотипом. Например, при создании элемента Subprogram body (тело подпрограммы) в модель добавляется компонент, который имеет стереотип Subprogram body, и который в любой момент можно изменить, что повлечет изменение графического отображения компонента.

Обычно имя компонента совпадает с именем файла, представляющего компонент, и для каждого компонента должен быть назначен язык программирования.

Package (пакет)



Значок Package позволяет отобразить пакет, который объединяет группу компонентов в модели.

Совет

Для того чтобы перенести созданный компонент в пакет, в окне Browse перетяните нужный компонент мышкой на значок пакета.

Dependency (зависимость)



Пользователь может установить связи данного типа между компонентами, которые изображаются как пунктирная стрелка от одного компонента к другому. Этот тип связей показывает, что классы, содержащиеся в компоненте-клиенте, наследуются, содержат элементы, используют или каким-либо другим образом зависят от классов, которые экспортируются из компонента-сервера.

Пользователь может отобразить связь компонента и интерфейса другого компонента и это означает, что компонент-клиент использует операции другого компонента (рис. 10.2). Если в этом случае связь будет направлена в другую сторону, то это означает, что компонент-клиент предоставляет операции компоненту-серверу.

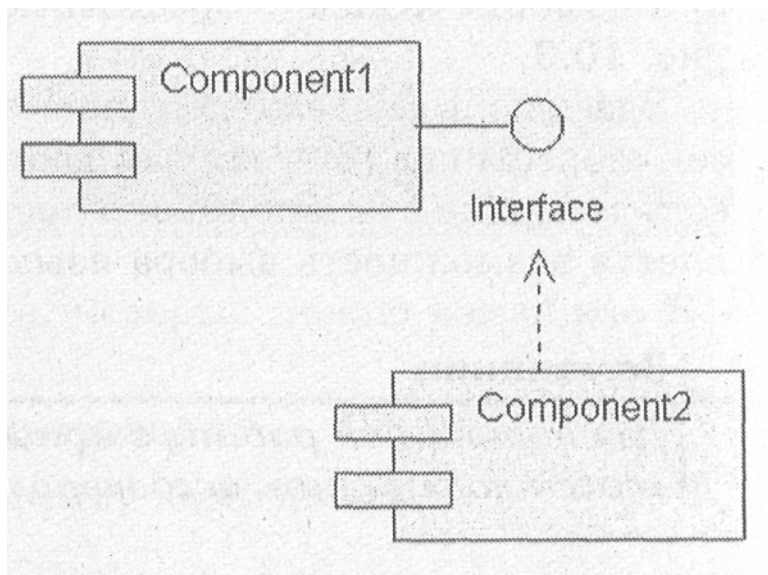


Рис. 10.2 Пример использования операций компонента

Main program (главная программа)



Значок Main program позволяет добавить в модель компонент, обозначающий главную программу. Для задач, которые создаются в объектно-ориентированной среде Visual Studio, главная программа скрыта библиотекой MFC, а приложение начинает работать с создания главного класса приложения theApp, который наследуется из библиотечного класса CWinApp, поэтому данный значок используется редко.

Subprogram body (тело подпрограммы)



Значок Subprogram body позволяет добавить в модель компонент, обозначающий тело подпрограммы и используется также для не объектно-ориентированных компонентов.

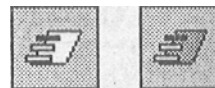
Package specification/body (определение/тело пакета)



Значки Package specification/body позволяют отобразить определения пакета (Package specification) и описание пакета (Package body), которые обычно связаны между собой.

Для языка C++ Package specification - это заголовочный файл с расширением .h, а Package body - это файл с расширением .cpp.

Task specification/body (определение/тело задачи)



Значки Task specification/body позволяют отобразить независимые потоки в многопоточковой системе. Если такие задачи компилируются независимо от обычных модулей, то появляется возможность разместить классы и их определения в таких задачах.

Свойства компонента

Для описанного ранее класса EnvironmentalController создадим значок Package Specification. Этим мы покажем, что определение данного класса будет находиться в файле EnvironmentalController.h

Вкладка General (главная)

После создания элемента перейдите во вкладку General его спецификаций при помощи RClick=>Specification=>General и заполните ее, как показано на рис. 10.3.

Здесь пользователю доступно изменение имени компонента (Name), дан список стереотипов (Stereotype), доступных для выбора, как и в других элементах, есть возможность заполнения документации (Documentation). Также предоставляется возможность выбора языка программирования для компонента.

Замечание

Для нормальной работы с исходным текстом компонента все включенные в него классы должны быть ассоциированы с одним языком программирования.

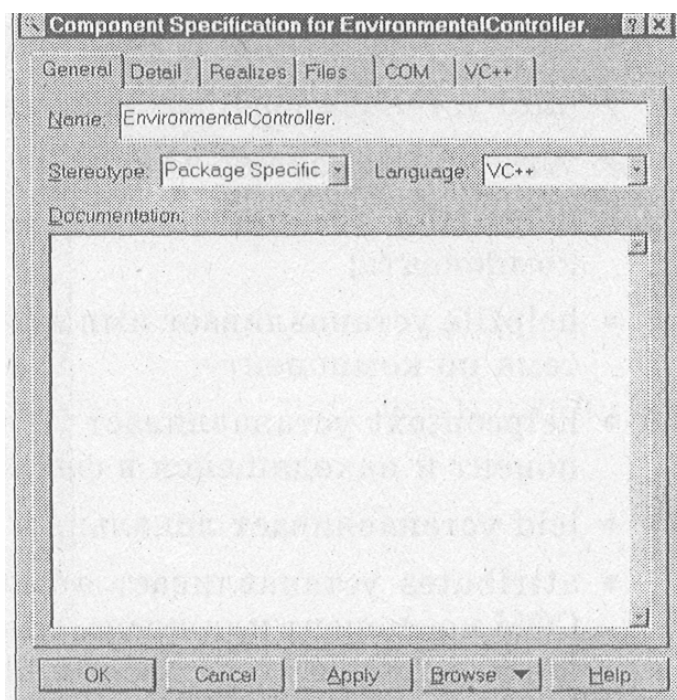


Рис. 10.3 Вкладка General спецификаций компонента

В связи с тем что мы создаем систему управления тепличным хозяйством на Visual C++, необходимо установить этот язык, изменив предлагаемое по умолчанию значение Analysis, на значение VC++. После чего при следующей активизации данного окна появятся вкладки COM и VC++.

Вкладка COM

Вкладка COM предназначена для установки свойств COM-объектов (рис. 10.4).

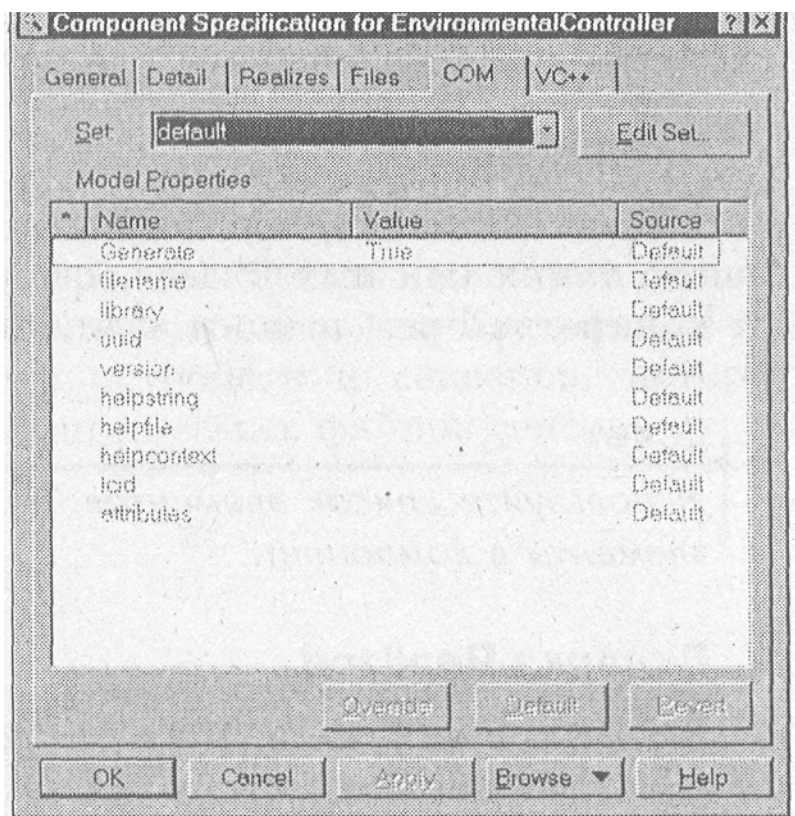


Рис. 10.4 Вкладка COM спецификаций компонента

На ней отражается список свойств объекта, которые можно изменить. Коротко перечислим эти свойства:

- Generate устанавливает необходимость генерации;
- filename устанавливает полный путь к файлу компонента;
- library устанавливает имя библиотеки, например, stdole;
- uuid устанавливает идентификатор для COM объекта;
- version устанавливает версию компонента;
- helpstring устанавливает строку, которая используется для описания компонента;
- helpfile устанавливает имя файла, в котором содержится справочная информация по компоненту;
- helpcontext устанавливает ID темы справки, описывающей данный

ком

понент и находящейся в файле, установленном как helpfile;

lcid устанавливает локальный идентификатор;

attributes устанавливает атрибуты, которые в дальнейшем определяют

COM компонент или класс, например, control, hidden, restricted, licensed, appobject, nonextensible или oleautomation.

Совет

Для того чтобы использовать классы компонентов ActiveX или других COM компонентов, загрузите их в модель при помощи Menu:Tool==>COM=>Import Type Library.

Вкладка VC++

Вкладка VC-H- предназначена для отображения свойств объекта, ассоциированного с языком VC++, однако, согласно встроенной документации, эти свойства не предназначены для установки пользователем.

Вкладка Detail

Вкладка Detail (детализация) показывает описание определений для компонента, таких как имя класса, переменные и другие конструкции, зависящие от конкретной реализации языка программирования.

Совет

Используйте список элементов, для того чтобы отразить физически включаемые элементы в компонент.

Вкладка Realized

Вкладка Realized спецификаций компонента позволяет показать включаемые в компонент классы, а также включить или исключить такие классы из компонента, при этом классы, включенные в компонент, отмечены значком (рис. 10.5).

Флажок Show All Classes (показать, все классы) позволяет показать только включенные в данный компонент классы или все классы, которые имеются в модели, например, для дальнейшего включения их в компонент.

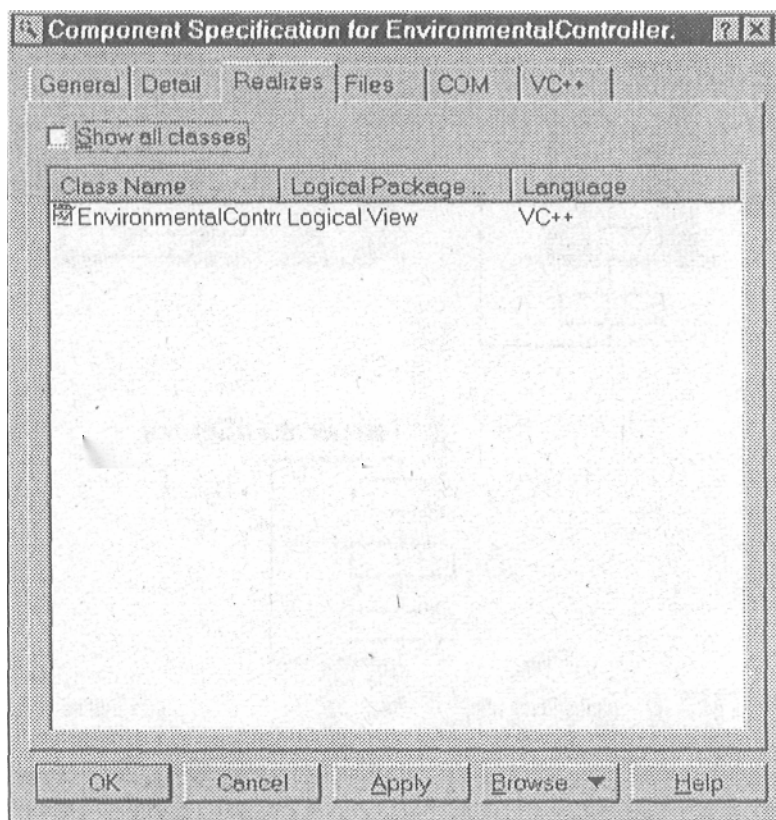


Рис. 10.5 Вкладка Realized спецификаций компонента

Вкладка Files

На вкладке Files представлен список файлов и URL, которые присоединены или добавлены в компонент.

Данная возможность используется для управления связями с дополнительными документами, которые указывают сведения по генерации системы или компонента. Из данного окна возможно открытие и просмотр связанных документов путем двойного нажатия мышкой на строке документа или из контекстного меню.

Создание диаграммы

Теперь создадим отображение остальных компонентов, связанных с контроллером. Создадим значки сенсоров и устройств. Соединим значки связями Dependency таким образом, чтобы показать, что для заголовочного файла контроллера требуются заголовочные файлы устройств и сенсоров, которые, в свою очередь, используются для компиляции самих файлов сенсоров и устройств так, как это показано на рис. 10.6.

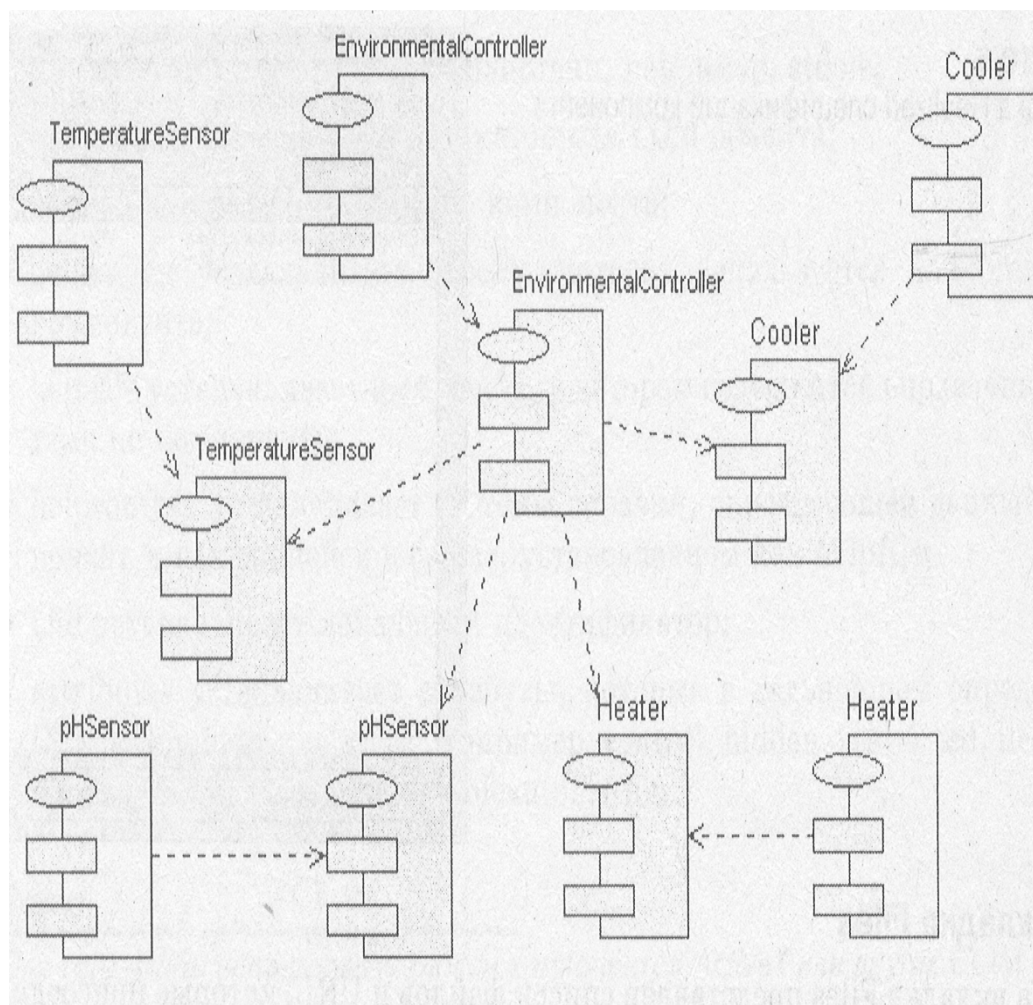


Рис. 10.6. Диаграмма компонентов

Вопросы для повторения

1. Каково назначение диаграммы компонентов?
2. Какие инструменты предоставляет диаграмма?
3. Какие свойства можно установить у компонентов?
4. Что такое стереотип и для чего он применяется?
5. Какие свойства можно настроить во вкладке COM?

Часть 3 Работа с диаграммой классов

Глава 11. Возможности диаграммы классов

Назначение диаграммы

Class diagram (диаграмма классов) основная диаграмма для создания кода приложения. При помощи диаграммы классов создается внутренняя структура системы, описывается наследование и взаимное положение классов друг относительно друга. Здесь описывается логическое представление системы. Именно логическое, так как классы - это лишь заготовки, на основе которых затем будут определены физические объекты.

Таким образом, диаграмма классов описывает общее представление системы и является противоположной Collaboration diagram, в которой представлены объекты системы. Однако такое разделение не является строгим правилом, и возможно смешанное представление классов и объектов.

Диаграмма классов используется не только для создания логического представления системы, Rational Rose позволяет на основе диаграммы классов создавать исходный код приложения. А так как описание классов создается на языке UML, то по диаграммам, созданным в едином стиле, возможна генерация исходного кода на любом языке программирования, который поддерживается генератором кода Rational Rose.

Замечание

В главах 13, 14, 15 будут рассмотрены возможности генерации исходного кода на языках программирования C++, Visual C++ и Visual Basic.

Обычно диаграмма классов создается для всех классов системы, в отличие от диаграммы объектов, которую проектировщики создают для отдельных объектов со сложным поведением и взаимодействием.

Диаграмма классов содержит значки, представляющие классы, интерфейсы и их связи. Классы могут представлять любые C++ классы: простые, параметризованные или метаклассы. Интерфейсы - это некоторый набор действий или операций, который обслуживает взаимодействие реализаций классов.

Возможно создание одной или нескольких диаграмм классов, которые описывают классы верхнего уровня в текущей модели. Также возможно создание одной или более диаграмм классов, которые описывают классы, содержащиеся в пакетах. Так, диаграмма классов сама по себе является пакетом для классов модели, но можно выделить дополнительные пакеты для логической группировки классов.

Посредством диаграммы классов возможно изменение в любой

момент свойств любого класса или его связей, и при этом диаграммы или спецификации, связанные с изменяемым классом, будут автоматически обновлены.

Диаграмма классов может быть использована как при анализе готовой системы, так и при разработке новой.

Создание диаграммы

Главная диаграмма классов (Main) уже присутствует во вновь созданной пустой модели, но возможно создание дополнительных диаграмм при помощи уже знакомых способов посредством контекстного меню Logical View в окне



Browse, при помощи пункта Browse в главном меню или при помощи кнопки Class diagram.

Строка инструментов

При активизации диаграммы строка инструментов приобретает следующий вид (рис. 11.1).

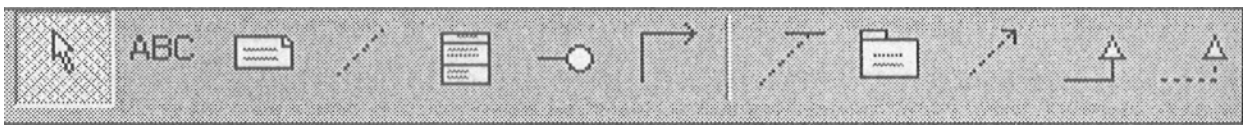


Рис. 11.1 Строка инструментов для диаграммы классов

Так же как и для предыдущих диаграмм не будем останавливаться на уже рассмотренных нами инструментах Selection Tool, Text Box, Note, Anchor Note to Item, которые выполняют стандартные функции.

Class (класс)



Данный инструмент позволяет создать новый класс в диаграмме и модели. Понятие класса в Rational Rose аналогично понятию классов в C++. Класс -это установки структуры и шаблона поведения для некоторого множества реальных объектов, которые в дальнейшем будут определены в программе на основе данного шаблона. Класс - это некоторая абстракция реального мира. Когда эта абстракция принимает конкретное воплощение, она называется объектом. Для детализации модели поведения классов создаются диаграммы состояний и действий, рассмотренные ранее.

Класс в UML нотации изображается как прямоугольник, разделенный на три части (рис. 11.2). В верхней части записывается название класса, в середине - атрибуты, в нижней части - операции.

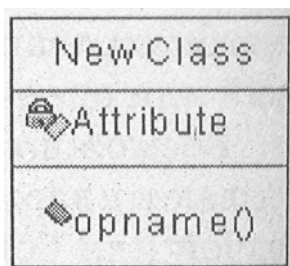


Рис. 11.2 Изображение класса

Замечание

Атрибуты и операции могут быть скрыты или вновь показаны, однако, нельзя забывать, что при скрытии этих элементов на диаграмме не отражается информация о наличии скрытых элементов.

Interface (интерфейс)



Значок Interface позволяет создать объект Interface, который указывает на видимые извне операции класса или компонента. Обычно интерфейс создается только для некоторых строго определенных классов или компонентов и предназначен скорее для логического отображения системы, но может присутствовать как на диаграмме классов, так и на диаграмме компонентов.

В диаграмме классов Interface обычно отображается как значок класса со стереотипом «interface».

Unidirectional Association (однонаправленная связь)



Значок Unidirectional Association позволяет создать однонаправленную связь класса с классом или класса с интерфейсом. Это общий и самый слабый вид связи.

Замечание

Подробно об этой и других видах связей читайте в главе 12.

Association Class (ассоциация класса)



Значок Association Class позволяет связать классы ассоциативной связью. Это свойство сохраняется в классе и для того, чтобы его установить, необходимо создать класс и связать класс реляцией с другим при помощи этого значка.

Package (пакет)



Значок Package позволяет создать элемент Package, который используется для группировки элементов. Может быть использован для

физической или логической группировки. В нашем случае пакет удобнее всего использовать для физической группировки кода. Но для небольшой системы, которой является тепличное хозяйство, мы не будем использовать пакеты.

Dependency of instantiates (зависимость реализации)



Значок Dependency of instantiates позволяет создать связь Dependency of instantiates, при этом генератор кода C++ Rational Rose создает код класса, включающий определения зависимого класса путем генерации директивы `#include`. Установка этого типа связей показывает, что класс использует другой класс как параметр в одном из методов.

Generalization (обобщение)



Значок Generalization позволяет создать связь Generalization, для которой Rational Rose создает код наследования, то есть создается подкласс для соединенного этой связью класса, наследуемого из родительского класса.

Realize (выполнять)



Значок Realize позволяет создать связь Realize между классом и интерфейсом или между компонентом и интерфейсом. Этот тип связи используется для того, чтобы показать, что класс выполняет операции, предоставляемые интерфейсом.

Помещение класса на диаграмму

Для создания нового класса и помещения его на диаграмму классов можно воспользоваться соответствующим значком из строки инструментов или меню `Menu:Tools==>Create=>Class`. Мы уже создали некоторые классы системы, но не поместили их на диаграмму. Для того чтобы поместить уже созданный класс, например, `EnvironmentalController` на диаграмму классов, есть несколько путей:

перетащить нужный класс мышкой из окна `Browse`;

воспользоваться `Menu:Query=>Add Classes` и в диалоговом окне

(рис.

11.3)

выбрать необходимые классы для включения в диаграмму.

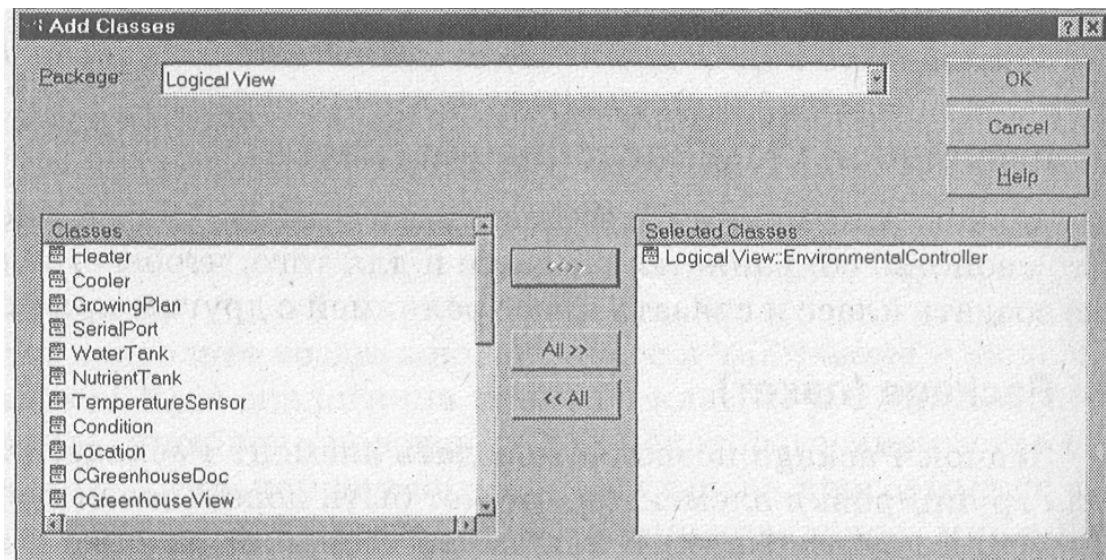


Рис. 11.3 Добавление созданного класса в диаграмму

Первый способ быстрый, но не для всех диаграмм подходит. Но если необходимо включить в диаграмму несколько уже имеющихся классов, то второй способ определенно лучше.

Понятие «стереотип» для класса

Мы уже встречались с понятием стереотипа в других диаграммах, но для диаграммы классов необходимо более подробное рассмотрение данного понятия.

Стереотип позволяет указывать дополнительные особенности для разрабатываемой модели, которые не поддерживаются языком UML. Понятие стереотипа позволяет легче добавлять информацию в новые элементы модели путем выбора стереотипа для этих элементов из уже заданных и представляет собой дополнительную классификацию элементов.

Некоторые стереотипы уже определены в Rational Rose, но всегда можно добавить новые стереотипы пользователя, описание которых сохраняется в файле стереотипов DefaultStereotypes.ini.

В качестве примера использования стереотипов можно привести следующий. Вы можете использовать для классов, которые предназначены для хранения данных, стереотип «Storage», для классов, которые предназначены для показа данных - стереотип «View», для классов, которые предоставляют пользователю возможность контроля за выполнением программы - «Controller».

Замечание

На стереотипах основаны возможности применения диаграммы классов для проектирования Web-приложений, о которых будем говорить в главе 20.

Стереотип может быть показан для класса или скрыт при помощи пункта Options контекстного меню класса.

На рис. 11.4 показан пример различных вариантов визуализации стереотипов. Слева направо: None, Label, Decoration, Icon.

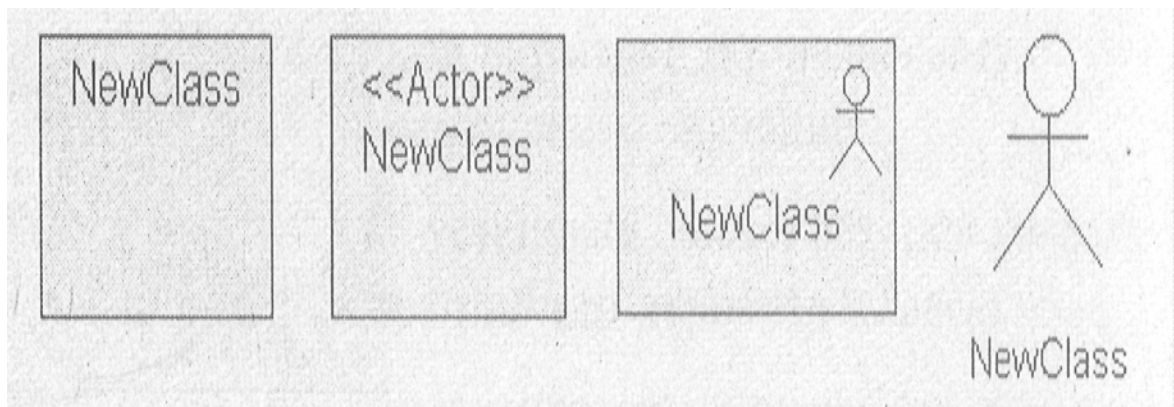


Рис. 11.4 Демонстрация различных вариантов визуализации стереотипов

Для демонстрации в предыдущем примере использован встроенный стерео-тип Actor.

Вот список некоторых стереотипов, доступных в Rational Rose:

- Actor (исполнитель);
- boundary (граница);
- business actor (бизнес-исполнитель);
- business entity (бизнес-сущность);
- business worker (работник);
- control (управление);
- entity (сущность);
- Interface (интерфейс).

Необходимость в стереотипах особенно ощущается при использовании диаграмм Use Case, разработка которых гарантирует, что система будет представлять собой именно то, что задумал пользователь, но и в диаграмме классов стереотипы могут использоваться для дополнительной детализации описания классов.

Контекстное меню класса

После добавления класса в диаграмму становится доступно контекстное меню класса. Содержание меню может изменяться при ассоциации класса с разными языками программирования. Пункты меню, относящиеся к языку программирования VC++ мы рассмотрим позднее, а сейчас ознакомимся с возможностями меню для класса, не ассоциированного с каким-либо языком программирования (рис. 11.5).

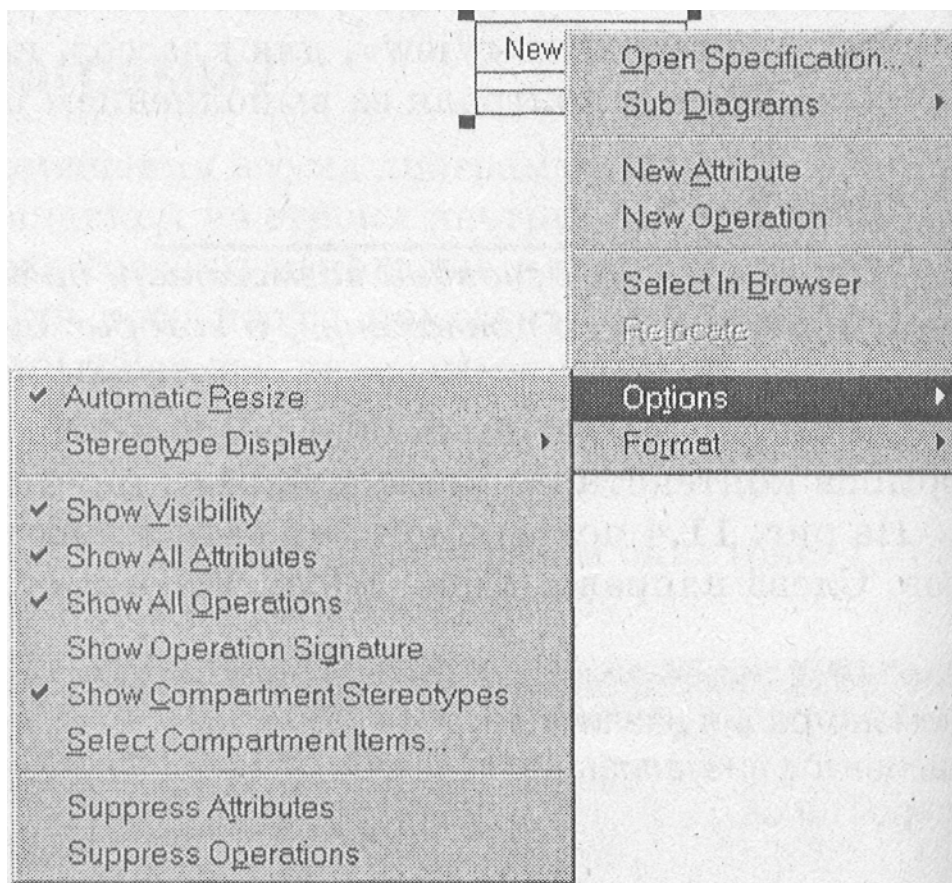


Рис. 11.5 Контекстное меню класса

Перечислим назначение отдельных пунктов:

Open Specifications - открытие диалогового окна заполнения спецификаций;

Sub Diagrams позволяет создавать к текущему классу диаграммы активности и состояний или перейти на поддиаграммы класса;

New Attribute позволяет добавлять новый атрибут класса;

New Operation позволяет добавлять новую операцию для класса;

Select in Browser позволяет выделить класс в окне Browser;

Relocate позволяет переместить класс в новый пакет или на новое место положения;

Options - вызов подменю настройки значка класса;

Format - вызов подменю настройки шрифта, цвета, заливки диаграммы.

Меню Options(свойства)

Меню Options позволяют управлять отображением класса в диаграмме классов и состоит из следующих пунктов:

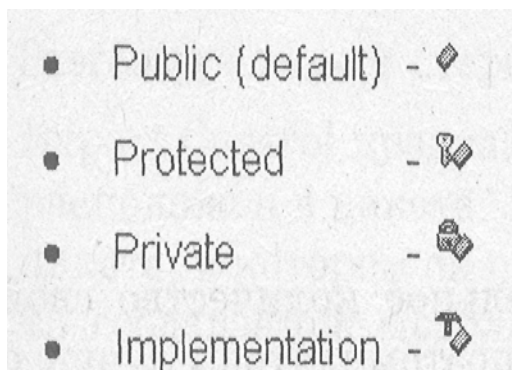
Automatic Resize - автоматическая настройка размера значка, для того чтобы вместить весь введенный текст названия, атрибута или операции. Данная функция удобна для начального заполнения названий атрибутов

и операций и включена по умолчанию. В дальнейшем, когда данный

класс уже связан с другими и занимает свое место на диаграмме классов, ее можно выключить;

Stereotype Display позволяет показать или скрыть стереотип для данного класса;

Show Visibility позволяет показать тип доступа для операторов и атрибутов, таких как public, protected, private, implementation. Причем показаны эти типы доступа будут при помощи графических значков;



Show All Attributes показывает или скрывает атрибуты класса;

Show All Operations показывает или скрывает все операции класса;

Show Operation Signature показывает или скрывает так называемую сигнатуру операции, т.е. параметры и возвращаемое значение;

Show Compartment Stereotypes эта установка позволяет показывать или скрывать имя стереотипа для операции или атрибута класса;

Select Compartment items позволяет активизировать окно выбора пунктов

операций или атрибутов для показа, в том случае если нужно скрыть не все атрибуты или операции, а только некоторые. Для этого необходимо активизировать окно Select Compartment items и выбрать необходимые для показа атрибуты и операции (рис. 11.6);

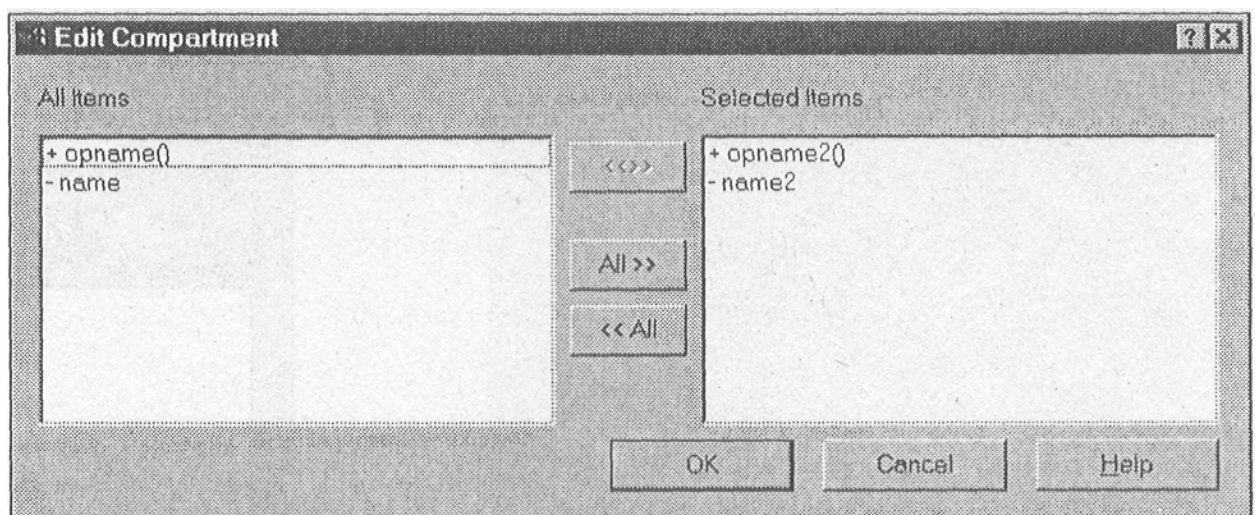


Рис. 11.6 Установка необходимых для показа атрибутов и операций

При этом из левой части окна, где присутствуют все реквизиты, необходимо переместить в правую только те, которые необходимы для показа, после чего нажать ОК.

Замечание

Для того чтобы внесенные изменения вступили в силу, необходимо снять галочку с пунктов меню Show All Attributes и Show All Operations, и только после этого все, кроме выбранных пунктов, будут скрыты.

Suppress Attributes позволяет скрыть все атрибуты, даже если они были

выбраны при помощи окна Select Compartment items. Этот пункт интересен тем, что при его выборе не только скрываются атрибуты, но и закрывается пункт меню Attributes, что не позволяет ввести новые;

Suppress Operations позволяет скрыть все операции аналогично атрибуту

там в предыдущем пункте.

Спецификации класса

Rational Rose позволяет устанавливать значительное количество свойств класса, которые влияют на генерацию кода класса, поэтому, чтобы лучше ориентироваться в дальнейших действиях, разберем вкладки окна спецификаций. Возможно, контроллер параметров среды вам уже порядком надоел, поэтому для разбора возможностей спецификаций возьмем класс датчика температуры TemperatureSensor.

Вкладка General (главная)

При выборе из контекстного меню датчика температуры пункта Open Specification открывается диалоговое окно, показанное на рис. 11.7. Спецификации класса имеют несколько вкладок, и первой активизируется вкладка General (главная).

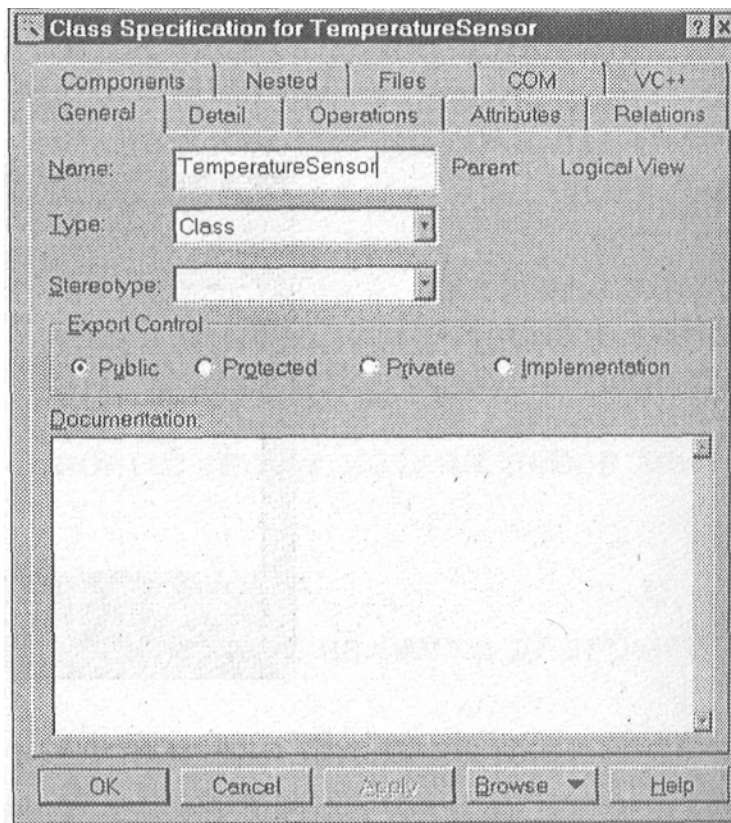


Рис. 11.7 Вкладка General спецификаций датчика температуры

Это окно позволяет задать главные свойства класса, такие как его имя, тип, определить стереотип класса и доступ к нему, когда класс находится в пакете. Так же как и во всех других диаграммах, здесь можно задать документацию к классу.

Перечислим поля, которые находятся на этой вкладке:

Name предназначено для задания имени класса;

Type предназначено для задания типа класса. В нашем случае это «класс», но может быть выбрано значение «метакласс», «параметризованный класс» и т.д.;

Stereotype задает стереотип класса;

Export Control предназначен для определения доступа к классу, когда он расположен в пакете. При этом Public определяет, что элемент виден вне пакета, в котором он определен и его можно импортировать в другие части создаваемой модели; Protected элемент доступен только для вложенных классов, классов с типом friends и собственно внутри класса; Private обозначает защищенный элемент класса; Implementation элемент виден только в том пакете, в котором определен.

Вкладка Detail (детализация)

Вкладка Detail позволяет указывать дополнительные установки класса, такие как ожидаемое количество создаваемых объектов класса, ожидаемый расход оперативной памяти и т.д. (рис. 11.8).

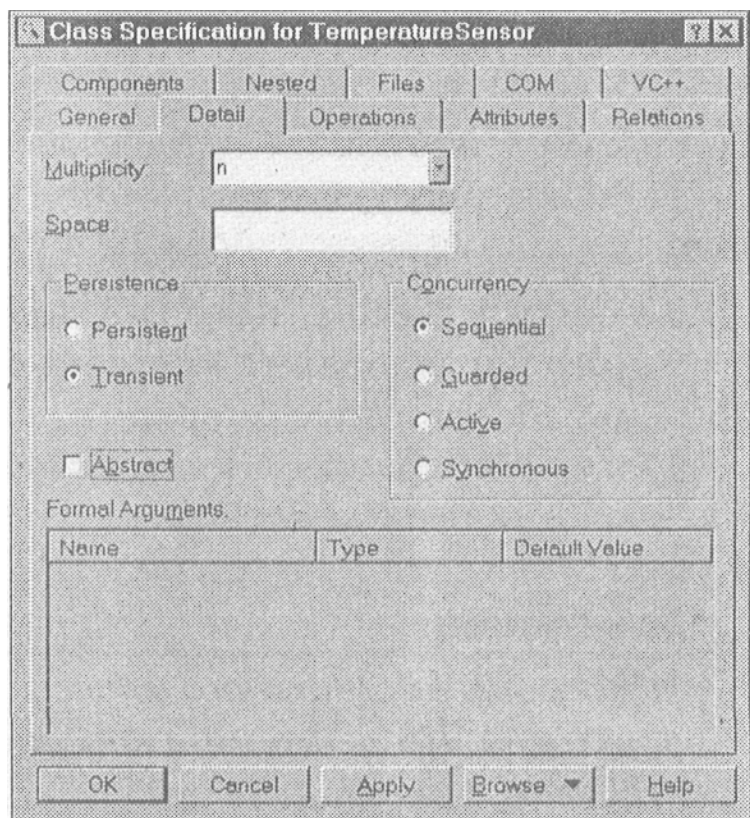


Рис 11.8 Вкладка Detail спецификаций датчика температуры

Перечислим поля, которые находятся на этой вкладке:

Multiplicity (множественность) позволяет задать ожидаемое количество объектов, которые будут созданы на основе данного класса. Обычно данный параметр удобно задавать для связанных классов;

Space показывает количество оперативной памяти, необходимой для создания объекта данного класса. Поле может быть задано напрямую или формулой, описывающей требования по памяти, и значение должно учитывать накладные расходы на создание объекта плюс размер всех объектов, входящих в данный;

Persistence определяет время жизни объекта класса. Если установлен флажок **persistent**, то объект должен быть доступен в течение всей работы программы или для доступа других потоков или процессов;

Замечание

Установка **persistence** для класса и для объекта должна быть одинаковой. Данная установка не действует для утилит классов, параметризованных утилит классов и реализаций утилит классов.

Concurrency обозначает поведение элемента в многопоточковой среде. Установка такого поля в операции не должна противоречить установке в самом классе. Данная установка может принимать следующие варианты:

1. **Sequential** (по умолчанию) - работа класса обеспечивается только для одного потока. Только один поток может быть запущен при помощи методов класса в один момент времени;
2. **Guarded** - класс обеспечивает работу с несколькими потоками.

Такой класс обеспечивает взаимодействие между потоками клиентов для достижения непротиворечивой работы потоков, является арбитром потоков, предоставляя работу конкретному потоку в конкретный момент времени;

3. Active - класс, является классом отдельного потока;

4. Synchronous класс обеспечивает работу нескольких потоков, синхронизируя их.

Abstract adornment обозначает, что класс является абстрактным, т.е. базовым классом, который должен быть наследован подклассами и без такого наследования не имеет смысла. Также в классе могут быть определены абстрактные операции, которые представляют собой шаблон для создания операций в классах, наследуемых из данного, и сами не выполняют никаких действий.

На основе абстрактного класса нельзя создавать объекты, но можно создавать подклассы. На диаграмме классов название абстрактного класса представляется курсивом.

Formal Arguments заполняется только для параметризованных классов и утилит классов. Для обычных классов данное поле недоступно.

Вкладка Components (компоненты)

Вкладка Components отражает компоненты, с которыми ассоциирован класс (рис. 11.9).

Persistence определяет время жизни объекта класса. Если установлен флажок persistent, то объект должен быть доступен в течение всей работы программы или для доступа других потоков или процессов;

Замечание

Установка persistence для класса и для объекта должна быть одинаковой. Данная установка не действует для утилит классов, параметризованных утилит классов и реализаций утилит классов.

Concurrency обозначает поведение элемента в многопоточной среде. Установка такого поля в операции не должна противоречить установке в самом классе. Данная установка может принимать следующие варианты:

1. Sequential (по умолчанию) - работа класса обеспечивается только для одного потока. Только один поток может быть запущен при помощи методов класса в один момент времени;

2. Guarded - класс обеспечивает работу с несколькими потоками. Такой класс обеспечивает взаимодействие между потоками клиентов для достижения непротиворечивой работы потоков, является арбитром потоков, предоставляя работу конкретному потоку в конкретный момент времени;

3. Active - класс, является классом отдельного потока;

4. Synchronous класс обеспечивает работу нескольких потоков, синхронизируя их.

Abstract adornment обозначает, что класс является абстрактным, т.е. базовым классом, который должен быть наследован подклассами и без такого наследования не имеет смысла. Также в классе могут быть

определены абстрактные операции, которые представляют собой шаблон для создания операций в классах, наследуемых из данного, и сами не выполняют никаких действий.

На основе абстрактного класса нельзя создавать объекты, но можно создавать подклассы. На диаграмме классов название абстрактного класса представляется курсивом.

Formal Arguments заполняется только для параметризованных классов и утилит классов. Для обычных классов данное поле недоступно.

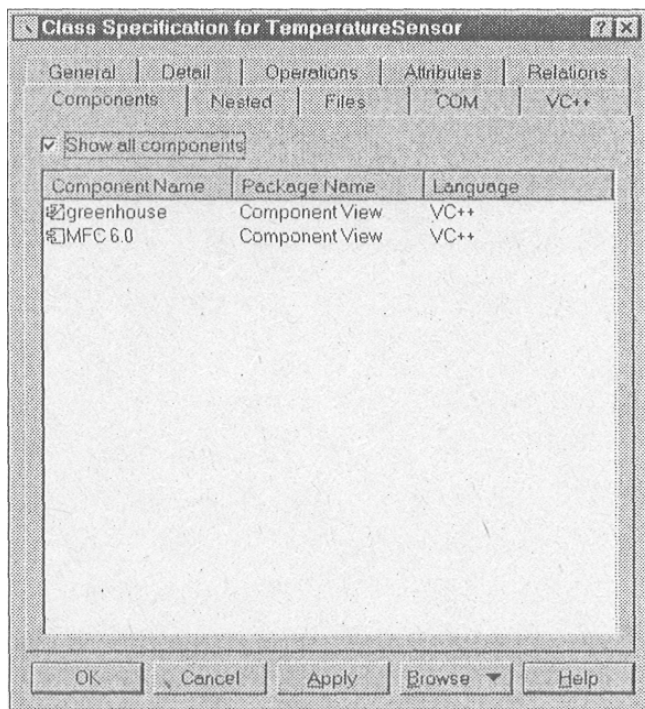


Рис. 11.9 Вкладка Components спецификаций датчика температуры

Замечание

Вкладка Components похожа на вкладку Realized в спецификациях компонента. Здесь возможна установка связи текущего класса и компонента, а в Realize установка связи текущего компонента и класса.

На вкладке помечены красным маркером компоненты, которые включены в текущую модель и могут быть показаны остальные компоненты модели. Если у вас не показан компонент MFC 6.0, то или необходимо включить флажок Show All Components, или еще не была импортирована библиотека MFC.

Совет

Для быстрого импорта библиотеки MFC необходимо проделать Menu:Tools=> Visual C++=>Quick Import MFC 6.0.

Из этой вкладки по двойному нажатию мыши на компоненте можно получить доступ к его спецификациям.

Вкладки Attributes (атрибуты)

Данная вкладка позволяет добавлять, удалять, редактировать атрибуты класса (рис. 11.10).

На данной вкладке представлен список атрибутов класса, который можно редактировать при помощи контекстного меню. Флажок Show inherited позволяет скрыть или показать доступные атрибуты родительских классов.

Для того чтобы добавить атрибут, необходимо из контекстного меню выбрать пункт Insert. По двойному нажатию мыши на атрибуте или из контекстного меню Rational Rose предоставляет доступ к диалоговому окну спецификаций атрибутов (рис. 11.11).

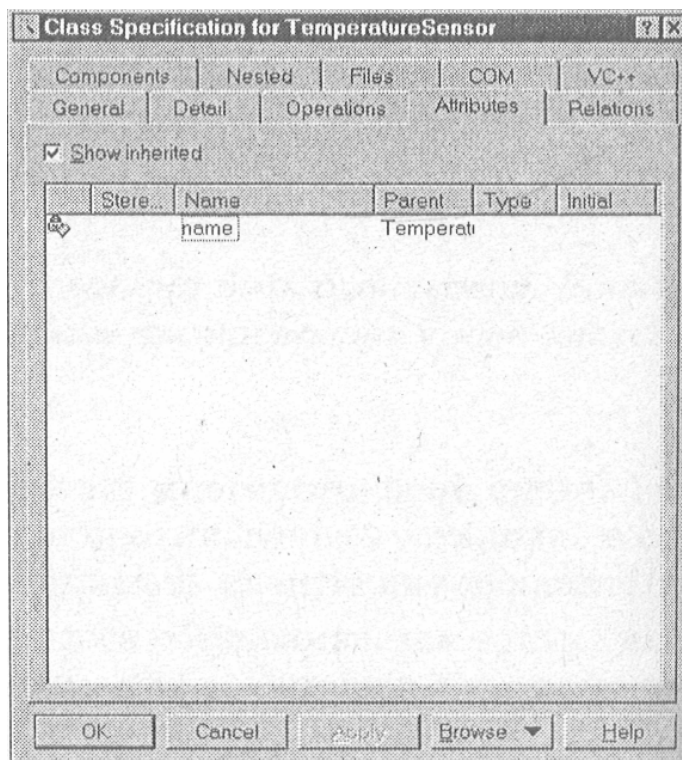


Рис. 11.10 Вкладка Attributes спецификаций класса

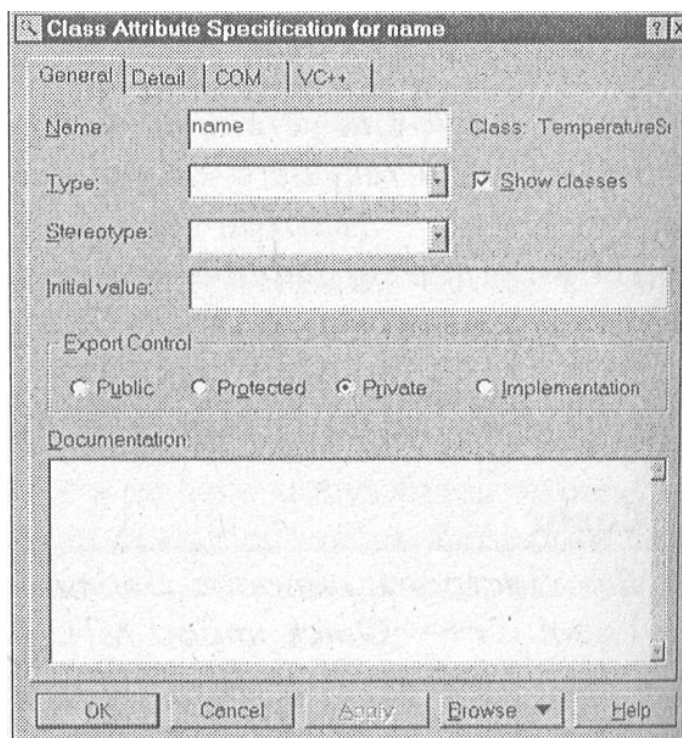


Рис. 1.1.11 Вкладка General спецификаций атрибутов класса

Здесь пользователь может изменить название атрибута (Name), его тип (Type) и стереотип (Stereotype), задать начальное значение (Initial value) и тип доступа к атрибуту (Export Control).

Дополнительная вкладка Detail спецификаций атрибутов класса (рис. 11.12) позволяет задать тип хранения атрибута в классе:

By Value - по значению;

By Reference - по ссылке;

Unspecified - не указано.

Также пользователь может указать, что атрибут является Static (статическим) или Derived (производным).

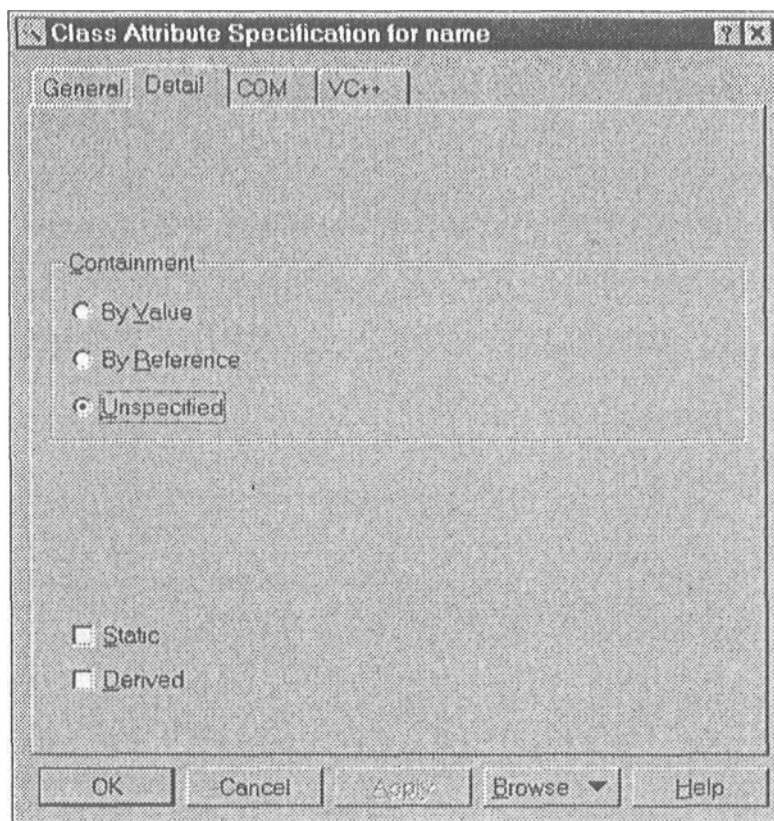


Рис. 11.12 Вкладка Detail спецификаций атрибутов класса

Вкладка Operations (операции)

Вкладка Operations позволяет добавлять, удалять, редактировать операции класса (рис. 11.13).

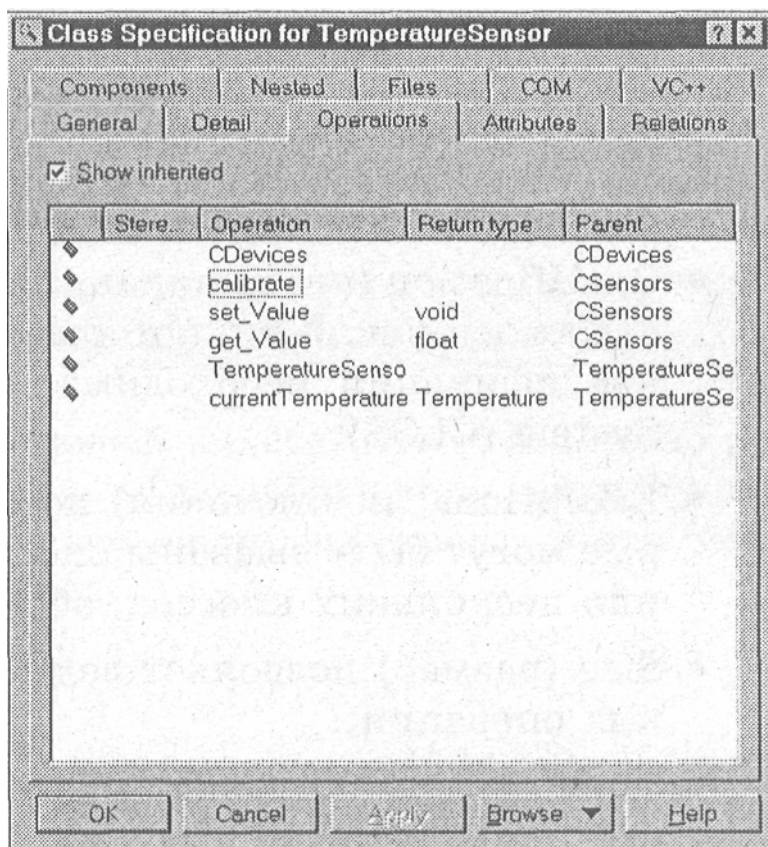


Рис. 11.13 Вкладка Operations спецификаций класса

На этой вкладке представлен список операций класса, который можно редактировать при помощи контекстного меню. Для того чтобы добавить операцию, необходимо из контекстного меню выбрать пункт Insert. По двойному нажатию мыши на операции или из контекстного меню Rational Rose предоставляет доступ к диалоговому окну спецификаций операции. Вкладка General спецификаций операции аналогична вкладке General атрибутов, поэтому не будем на ней останавливаться.

Вкладка Detail спецификаций операций позволяет устанавливать дополнительные свойства операции на этой вкладке:

Arguments (аргументы) позволяет устанавливать список аргументов для операции с их типами и значениями по умолчанию;

Protocol (протокол) позволяет задавать список установок, который предоставляется клиенту для вызова;

Qualification (квалификация) позволяет идентифицировать зависящие от языка возможности, которые позволяют квалифицировать метод. Данная квалификация необходима, если вы используете Common Lisp Object System (CLOS);

Exceptions (исключения) позволяет задавать список исключений, которые могут быть вызваны операцией. Здесь необходимо ввести имя одного или нескольких классов, обрабатывающих исключительные состояния;

Size (размер) позволяет задать размер памяти, требуемой для выполнения операции;

Time (время) позволяет задать время выполнения операции;

Concurrency (конкуренция) отражает для многопоточковой программы тип выполнения операции:

Sequential (default) - только один поток должен выполняться в одно и то же время;

Guarded - возможно выполнение нескольких потоков, управляемых некоторым классом;

Synchronous - операции могут идти параллельно.

Пользователь может устанавливать конкуренцию выполнения для класса в спецификациях класса. Это поле недоступно для редактирования утилит классов, параметризованных классов и реализаций утилит классов.

Вкладки Preconditions, Postconditions, Semantics позволяют задавать дополнительные описания процессов подготовки и завершения операции, а так-же описание алгоритма операции. Кроме текстового описания здесь же можно задать имя Interaction диаграммы, которая описывает указанные действия.

Вкладка Relations (связи)

Вкладка Relations позволяет добавлять, удалять, редактировать связи класса (рис. 11.15).

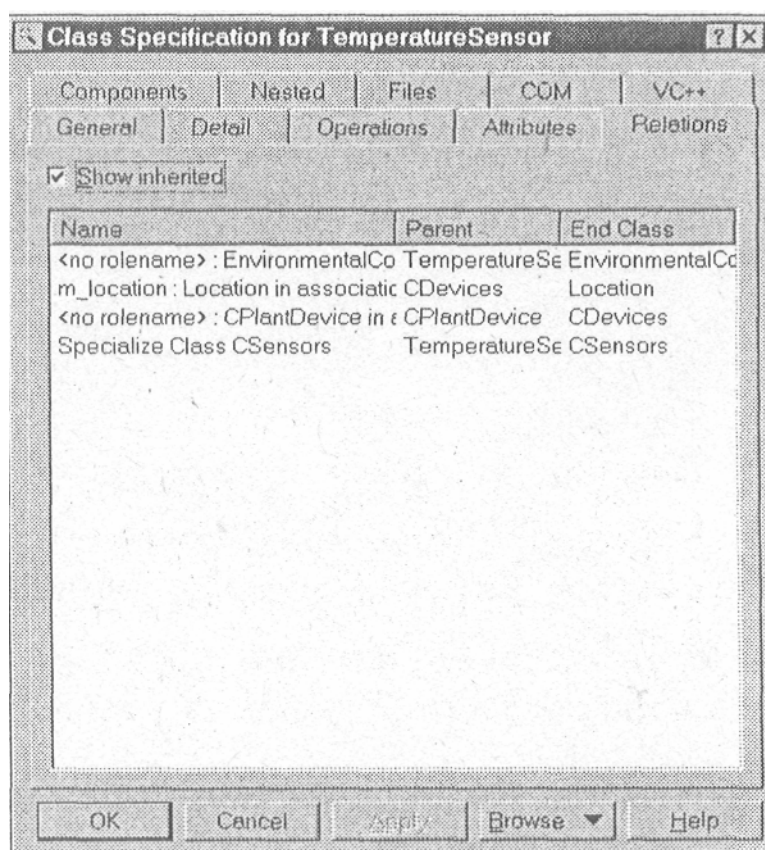


Рис. 11.15 Вкладка Relations спецификаций класса

На данной вкладке представлен список связей класса, который можно ре-дактировать при помощи контекстного меню. Для добавления связи лучше всего воспользоваться соответствующим инструментом из строки инструментов, а для удаления – контекстным меню.

Замечание

Спецификации связей будут рассмотрены позднее для связи Unidirectional association.

Замечание

После удаления связей с диаграммы полностью удалить их можно только посредством вкладки Relations.

Вкладка VC++

Вкладка VC++, которая появилась после ассоциации класса с языком [Visual C++, предназначена для изменения свойств, связанных с данным клас-

сом. Поля данной вкладки не предназначены для редактирования, поэтому не будем их рассматривать.

Вкладка COM

Вкладка COM позволяет устанавливать свойства для классов, которые предназначены для создания COM объектов в модели. В случае, если такие объекты импортируются в модель, в них также появляется такая вкладка (рис. 11.16).

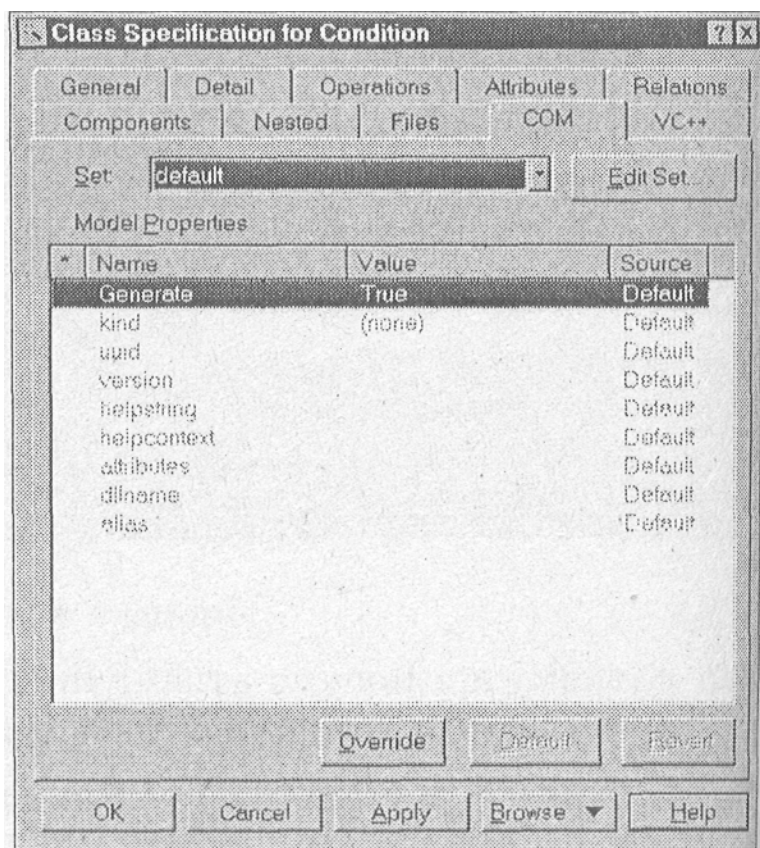


Рис. 11.16 Вкладка COM спецификаций класса

Generate - свойство, определяющее необходимость генерации исходного кода класса;

Kind - свойство, которое идентифицирует тип объекта, который может быть следующим: coclass, enum, record, module, interface, dispinterface, alias, union, max;

Uuid - свойство, которое задает строку идентификатора для класса или библиотеки, которая идентифицирует COM объект в системе, например, «11611EBF-070D-11D1-8001-00A0C922E84A».

Version - версия COM объекта;

Helpstring, helpcontext предназначены для задания строки и идентификатора файла помощи для объекта;

Attributes задает такие атрибуты для объекта как control, hidden, restricted, licensed, appobject, nonextensible или oleautomation;

Dllname задает имя файла DLL, в котором находится объект;

Alias задает тип, для которого создается псевдоним.

Вопросы для повторения

1. Каково назначение диаграммы классов?
2. Какими способами можно создать диаграмму?
3. Какие инструменты доступны для диаграммы?
4. Какие команды предоставляет контекстное меню класса?
5. Как настроить свойства атрибутов класса?
6. Как настроить свойства методов класса?

Глава 12. Связи

Назначение и виды связей

Классы редко бывают изолированы, чаще всего они взаимодействуют друг с другом. Эти взаимодействия показываются при помощи различного вида связей. Типы связей влияют на получаемый при генерации на основе диаграмм исходный код, поэтому необходимо рассмотреть связи и спецификации подробно.

В диаграмме классов используются следующие виды связей:

unidirectional association (однонаправленная ассоциация);

dependency (зависимость);

association class (ассоциированный класс);

generalization (наследование);

realization (реализация).

Рассмотрим каждый тип связи в соответствующем разделе.

Unidirectional Association

Одна из важных и сложных типов связи, которая используется в диаграмме классов. Данная связь показывает, что один класс включается в другой как атрибут по ссылке или по значению (рис. 12.1).

Используйте задание атрибутов, когда хотите включить в класс переменную простого типа и используйте Unidirectional Association, когда хотите включить в класс переменную со сложным типом, например, класс или структуру.

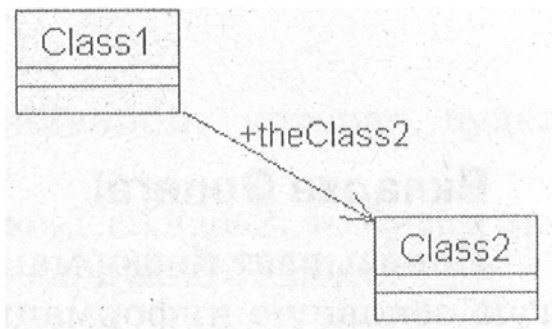


Рис. 12.1. Пример связи Unidirectional Association

Приведенный далее листинг показывает код C++, который был создан для I показанной на рисунке связи.

```
class Class2; class Class1
{
public:
    Class2* theClass2;
};
```

Замечание

Подробно создание кода класса на C++ и Visual C++ будет

рассмотрено в главах 13, 14.

Rational Rose создает код класса в зависимости от установленных спецификаций связи, поэтому рассмотрим влияние различных установок спецификаций на получаемый код. При нажатии правой кнопки мыши на связи активизируется контекстное меню, которое предоставляет быстрый доступ к установкам связи. Однако спецификации связи позволяют проделать то же самое при помощи вкладок диалоговых окон.

Активизируйте окно спецификаций посредством контекстного меню или двойного нажатия мыши на стрелке ассоциации, при этом открывается вкладка General спецификаций (рис. 12.2).

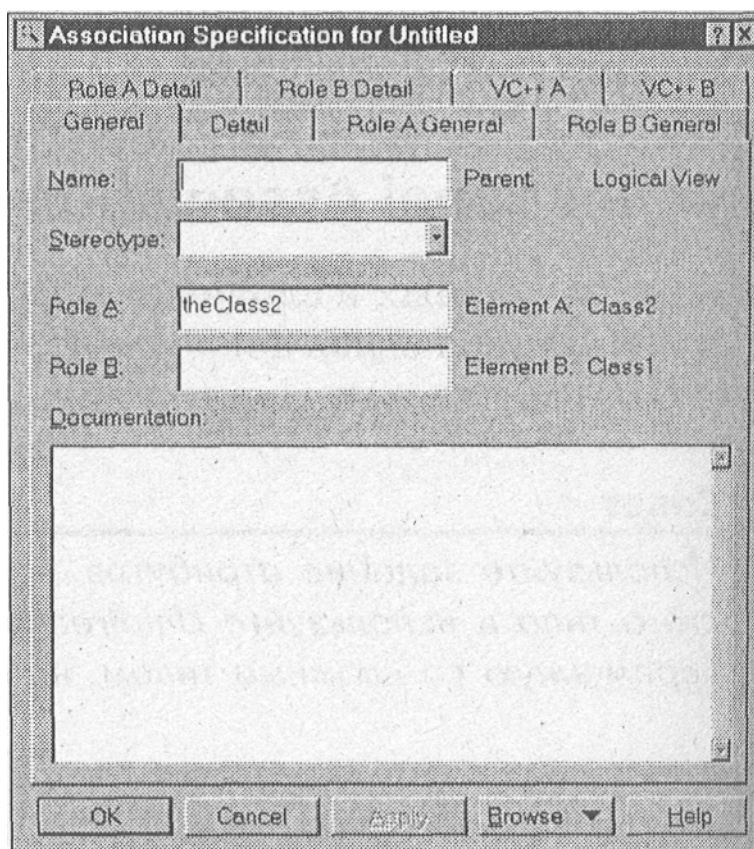


Рис. 12.2. Вкладка General спецификаций Unidirectional Association

Вкладка General

Показывает информацию об имени, стереотипе, родительском классе и другую основную информацию о связи.

Name (имя) задает имя связи. Для каждой связи может быть, хотя и не обязательно, задано имя, которое одним словом или целой фразой указывает цель или семантику связи;

Parent (родительский) указывает имя пакета, которому принадлежит связь;

Stereotype (стереотип) указывает стереотип;

Role A/Role B указывает имя роли с которой один класс ассоциируется с другим;

Element A/Element B указывает имя класса, который ассоциирован с данной ролью.

Вкладка Detail

Вкладка Detail указывает дополнительные свойства связи (рис. 12.3).

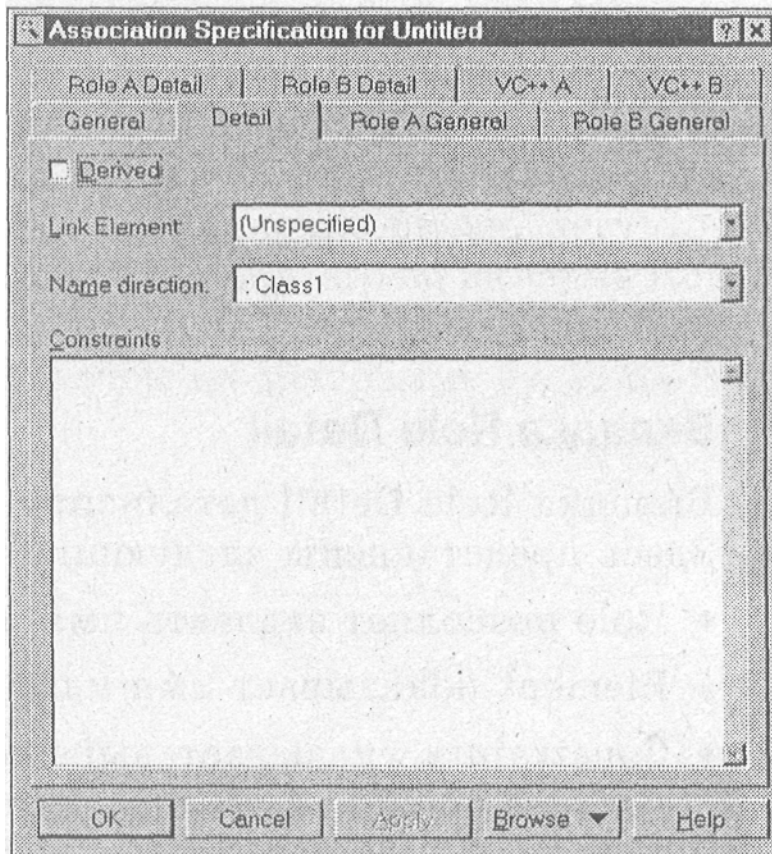


Рис. 12.3 Вкладка Detail спецификаций Unidirectional Association

Link Element указывает ассоциированный класс, если связь соединена с ним ассоциацией, как показано на рис. 12.8;

NameDirection указывает имя связанного класса;

Constraints указывает выражение некоторого семантического условия, которое должно быть выполнено, в то время как система находится в устойчивом состоянии.

Вкладка Role General

Вкладка Role General отражает настройки переменной, которая будет включена в класс (рис. 12.4).

Так как направление связи в нашем примере от Class 1 к Class2, то будем заполнять Вкладку Role A General. Эта вкладка имеет следующие поля:

Role позволяет задавать имя переменной для класса;

Element показывает имя класса, для которого создается переменная;

Export Control определяет доступ к данному элементу. В нашем случае установлен переключатель Public, поэтому переменная была создана в секции Public.

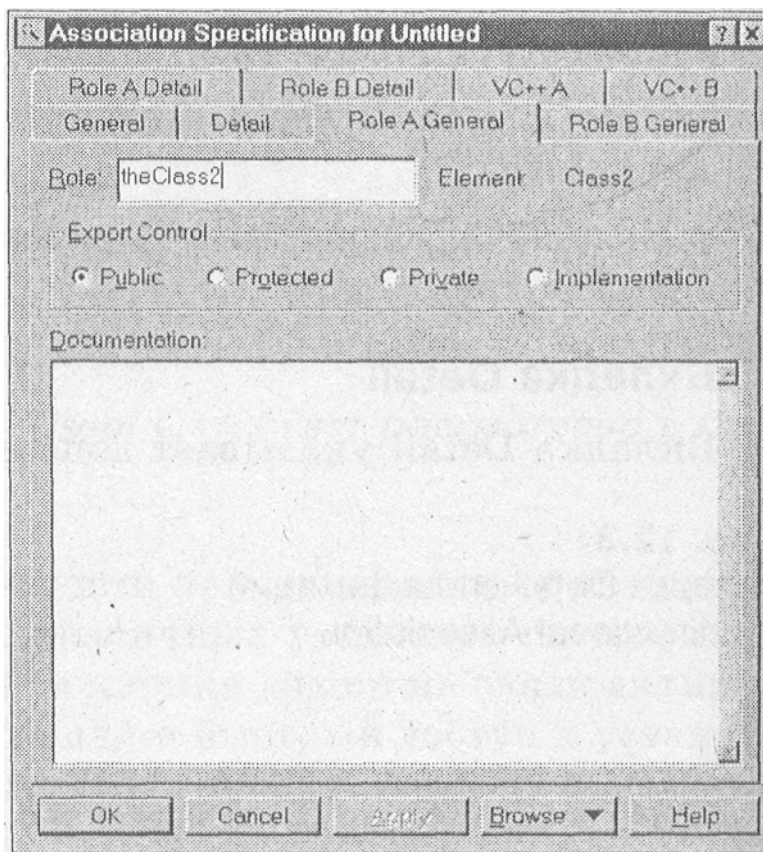


Рис. 12.4 Вкладка Role. A General спецификаций Unidirectional Association

Вкладка Role Detail

Вкладка Role Detail детализирует установки для связи (рис. 12.5). Здесь представлены следующие поля:

Role позволяет задавать имя переменной класса;

Element показывает имя класса, для которого создается переменная;

Constraints указывает выражение некоторого семантического условия, которое должно быть выполнено, в то время как система находится в устойчивом состоянии. При задании ограничения оно будет показано на диаграмме в фигурных скобках. Причем на формируемом коде данное ограничение не будет отражаться.

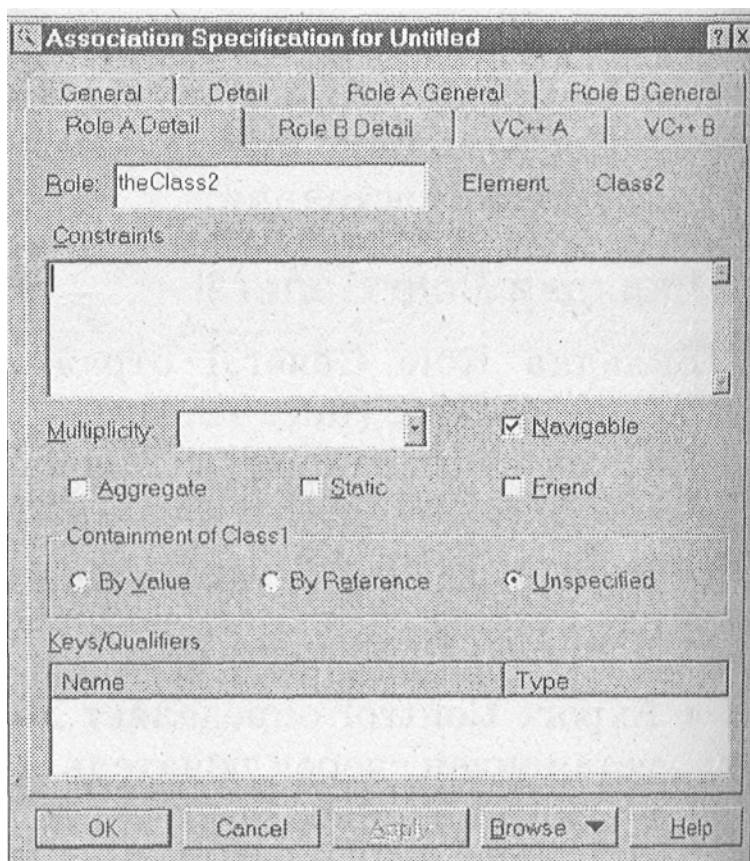


Рис. 12.5 Вкладка Role A Detail спецификаций Unidirectional Association

Замечание

Поле Constraints во вкладке Detail применяется к связи в целом, в то время как ограничение, указанное в данной вкладке, применяется непосредственно к данной роли.

Multiplicity показывает, сколько ожидается создать объектов данного класса. Может быть задано числом или буквой «n». Значение «n» указывает, что количество не лимитировано. Можно задать различные варианты ожидаемого количества или диапазон. Причем, указанное число отображается рядом со стрелкой связи;

Navigable показывает направление, в котором действует связь. При установке этого флажка связь приобретает вид стрелки, указывающей направление связи. Это поле напрямую влияет на создаваемый код класса, так как на какой класс будет направлена стрелка связи, тот и будет включен в другой. Для того чтобы изменить направление связи, достаточно снять флажок с вкладки Role A Detail и установить его во вкладке Role B Detail. При этом в случае, когда сняты флажки на обеих вкладках ни один элемент не будет включен в другой. При этом на диаграмме будет показана просто линия (рис. 12.6);

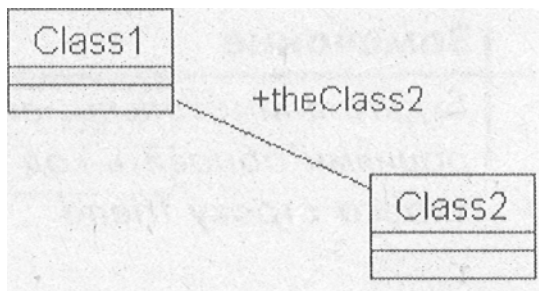


Рис. 12.6 Пример ненаправленной связи

Aggregate показывает, что один класс не просто использует, а содержит другой. Однако будьте внимательны, для того чтобы показать, что класс Class2 входит в класс Class 1, необходимо установить этот флажок во вкладке Role B Detail. При этом стрелка связи на диаграмме приобретает ромб с обратной стороны стрелки (рис. 12.7).

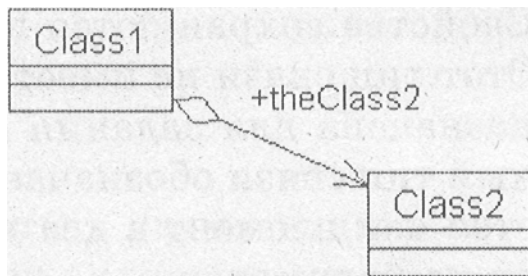


Рис. 12.7 Пример агрегирования класса

Агрегирование означает физическое включение связанного класса в другой класс. При этом генератор кода C++ Rational Rose создает код, приведенный на листинге.

```

class Class2;
class Class"
public:
Class2 theClass2;
}

```

Замечание

Агрегирование может быть включено только для одной вкладки - или для Role A, или для Ro/e, но не одновременно.

Static обозначает, что данный реквизит - общий для всех объектов данного класса. Причем после инициализации к нему можно обращаться, даже если еще не было создано ни одного объекта класса. Static применяется для того, чтобы переменные такого типа не тиражировались при создании нового объекта класса. Например, если необходимо точно знать, сколько объектов класса создано, то можно поручить самому классу следить за этим числом, создав в нем переменную типа static int iCnt, видимую во всех объектах класса;

Friend определяет, что указанный класс является дружественным классом, то есть имеет доступ к защищенным методам и атрибутам класса.

Замечание

Будьте внимательны, если флажок установить, а потом снять, а между этими операциями обновить код по модели, то затем необходимо вручную удалить из кода класса строку `friend`.

Key /qualifier - атрибут, который идентифицирует уникальным образом единичный объект. На генерацию кода влияния не оказывает.

Association Class (ассоциированный класс)

Используйте данный тип связи для отображения свойства ассоциации. Свойства сохраняются в классе и соединяются связью Association (рис. 12.8). Этот тип связи не имеет своих спецификаций. В общем виде ассоциация предназначена для задания дополнительных атрибутов у связи. Фактически данный тип связи обозначает, что некоторый класс со своими атрибутами включается как элемент в два других, хотя, как ни странно, при генерации кода это не отображается.

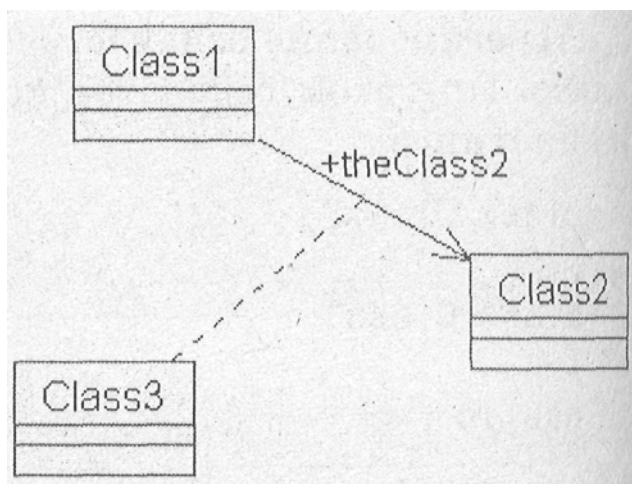


Рис. 12.8 Пример использования Association Class

Dependency of instances (зависимость)

Этот тип связи позволяет показать, что один класс использует объекты другого. Использование может осуществляться при передаче параметров или вызове операций класса. В этом случае генератор кода C++ Rational Rose включает заголовочный файл в класс, который использует операторы или объекты другого класса. Графически этот вид связи отражается пунктирной стрелкой (рис. 12.9).

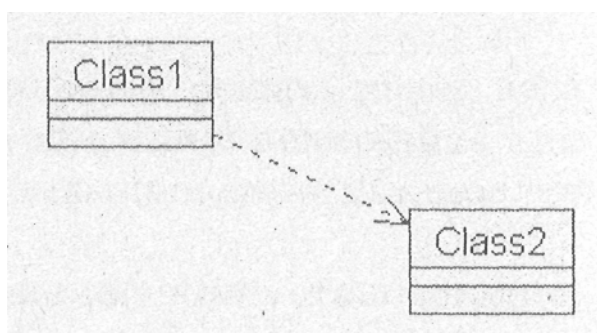


Рис. 12.9 Пример связи Dependency of instanties Generalization

Данный тип связи позволяет указать, что один класс является родительским по отношению к другому, при этом будет создан код наследования класса. Пример такой связи показан на рис. 12.10.

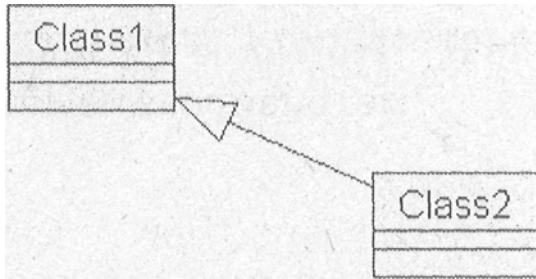


Рис. 12.10 Пример связи Generalization

Далее показан листинг, полученный при генерации кода для показанного примера.

```
#include "Class1.h"
class Class2 : public Class1 {
};
```

Далее для работы мы, в основном, будем пользоваться двумя видами связей - это Unidirectional Association для агрегирования включения ссылок на классы и Generalization для создания иерархии наследования.

Realization

Данный тип связи позволяет показать, что один класс является реализацией, т.е. создан на основе шаблона другого. В С++ широко используется понятие шаблон (template), означающее класс, на основе которого создается целое семейство классов, поведение которых определяется поведением шаблона, однако тип данных, с которым будет работать созданный на его основе класс, определяется параметром шаблона. В Rational Rose для обозначения класса шаблона используется понятие Parameterized Class (параметризованный класс). Вы можете создать его при помощи специального значка, который необходимо включить в строку инструментов. Или в окне спецификаций класса изменить его тип/Подробнее о шаблонах читайте в [2, 6].

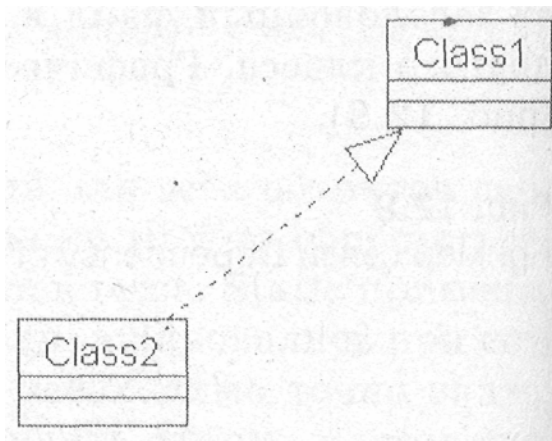


Рис. 12.11 Пример связи Realization

Вопросы для повторения

1. Какие виды связей доступны в диаграмме классов?
2. Какие спецификации доступны для связи Unidirectional Association?
3. Для чего используются связи Association Class?
4. Каково назначение связей Dependency и Generalization?
5. Что такое Parameterized Class и как обычный класс изменить на параметризованный?

Глава 13. 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. В этой главе мы немного изменим тактику. 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, а создание кода на Visual C++ разберем в следующей главе.

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 int Location;
class TemperatureSensor {
public:
    TemperatureSensor (Location);
    ~TemperatureSensor ();
    void          calibrate          (Temperature          actualTemperature);
    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 сделаем следующее:

Выберем класс, затем Menu:Tools=>C++=>Code Generation. Появится диалоговое окно, показанное на рис. 13.1.

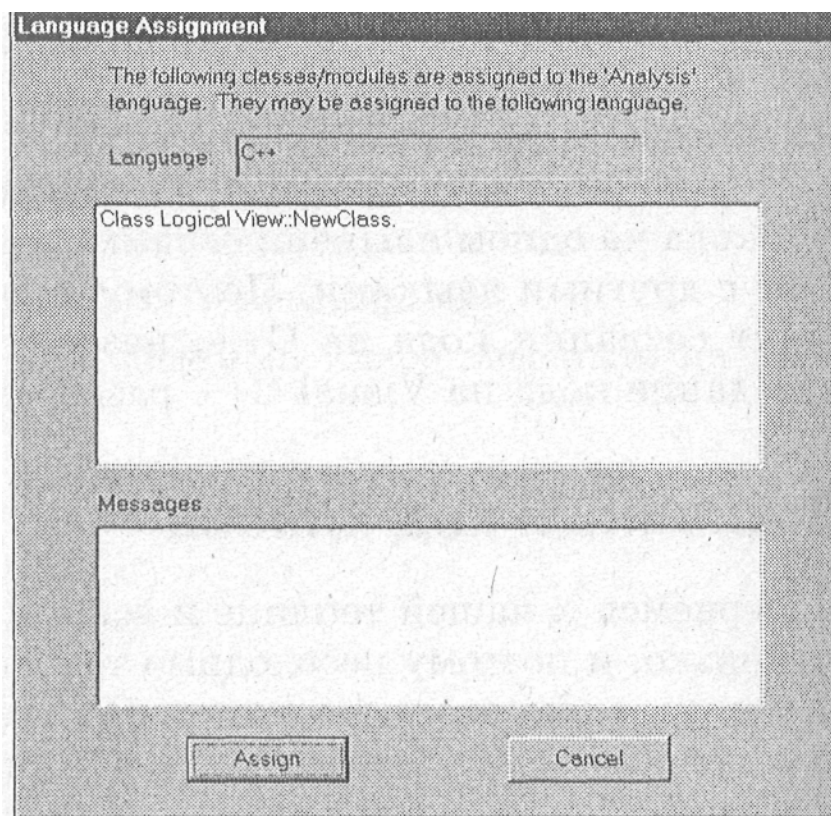


Рис. 13.1 Назначение классу языка программирования C++

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

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. 13.2.

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

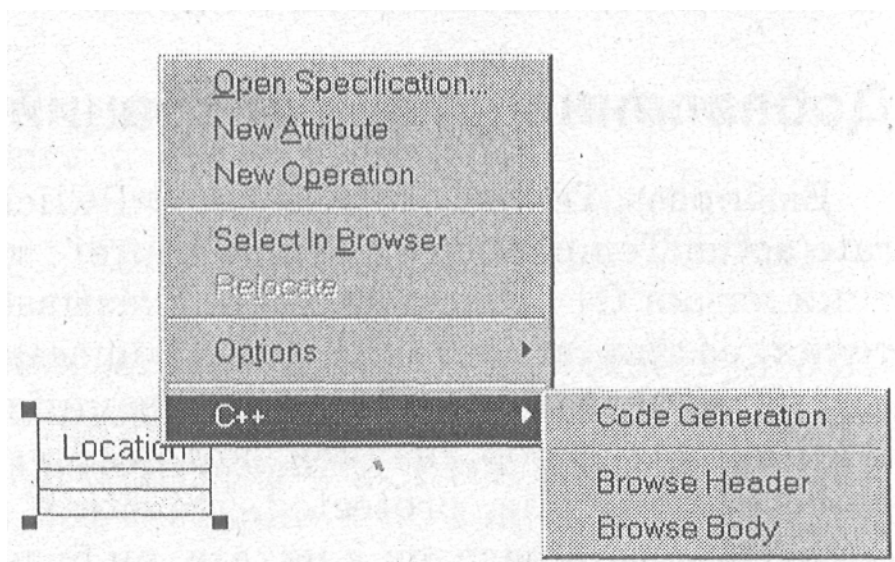


Рис. 13.2 Доступное меню C++

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. 13.3.

Совет

Если на любом свойстве нажать правую кнопку мыши, то откроется описание этого свойства из встроенной справки. Краткое описание свойств приведено в конце главы.

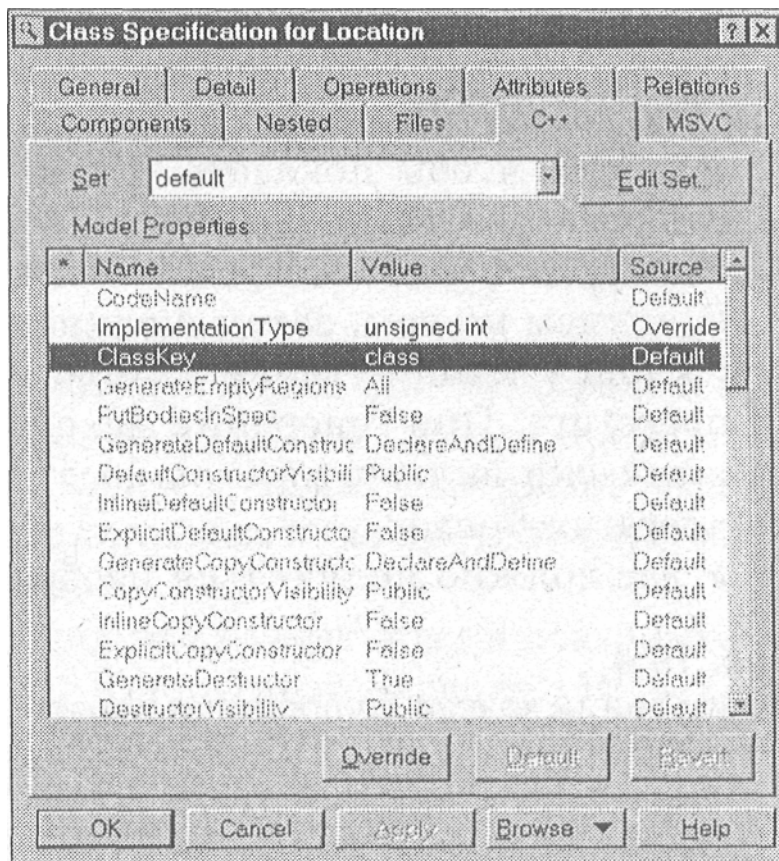


Рис. 13.3 Заполнение свойства Implementation Type

Прodelайте 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 calib-rate(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 пакете, он будет виден только для объектов, определенных в этом пакете.

Аналогично добавим конструктор и операцию получения температуры.

Совет

Если заполнить поле TemperatureSensor=>RClick=>Open Specification=>Documentation, то вы получите автоматически

создаваемые комментарии в исходном тексте, которые высвечиваются в окне Documentation, когда выделяется документированный класс.

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.

Щелкнем 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. 13.4.

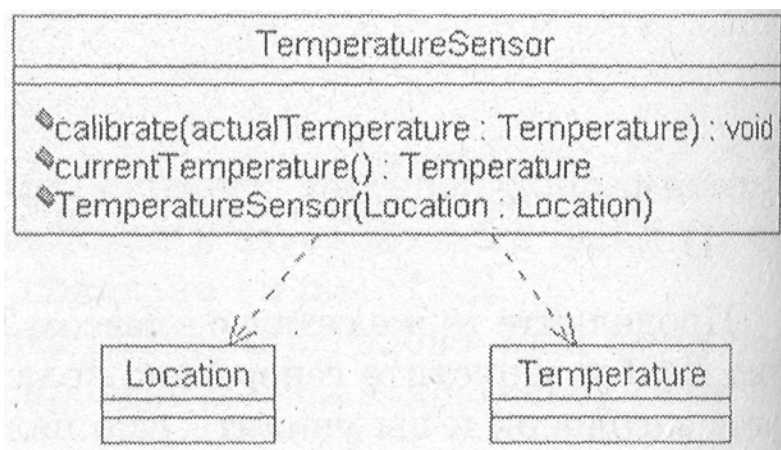


Рис. 13.4 Зависимости класса Temperature Sensor

Совет

Для того чтобы показать/скрыть параметры и типы возврата операций, используйте RClick=>Options=>Show Operation Signature.

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. Я приведу 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.

Вы можете отличить созданные генератором строки по символам «##», вставляемым в комментарии.

```
#include "Location.h"
tfinclude "Temperature.h"
```

```
// Датчик температуры, измеряет температуру в теплице в
градусах Цельсия
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 ();
}
```

Здесь мы видим, что в код включились заголовочные файлы определения данных и комментариев (это было заполнено поле RClick=>Open Specification=>Documentation). Однако конструкторов в классе целых три: два автоматически созданных и один тот, 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 => OpenSpecification => 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.

Совет

Для того чтобы быстро выделить нужный класс в окне Browser, находясь в окне Class Diagram, сделайте RClick=>SelectInBrowser.

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

Настройка свойств C++

Пока мы проделали все эти действия без подробных объяснений, с целью 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 будет 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.

Замечание

Если конструктора по умолчанию нет, то данное свойство не оказывает на код ни какого эффекта.

ExplicitDefaultConstructor устанавливает конструктор по умолчанию как explicit (явно заданный).

GenerateCopyConstructor 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.

Destructor Visibility 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.

GenerateAssignmentOperation устанавливает, будет ли создаваться функция переопределения оператора присваивания (=).

Assignment Visibility определяет раздел, где будет создаваться функция оператора присваивания.

The AssignmentKind определяет вид функции оператора присвоения:

Common - обычная, Virtual - виртуальная, Abstract - абстрактная, Friend - дружественная.

InlineAssignmentOperation определяет, будет ли оператор присвоения создаваться как inline подстановка.

GenerateEqualityOperations определяет, будут ли переопределяться операторы сравнения на равенство (== и !=).

Equality Visibility 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 (<, <=, >, >=).

Relational Visibility 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.

GenerateSubscriptOperation определяет, будет ли переопределен оператор [].

Subscript Visibility определяет раздел, в который будет помещен оператор [].

SubscriptKind определяет вид функций оператора []: Common - обычная, Virtual - виртуальная, Abstract - абстрактная.

SubscriptResultType определяет тип возвращаемого выражения для оператора [].

InlineSubscriptOperation определяет, будет ли оператор [] определен как inline подстановка.

GenerateDereferenceOperation определяет, будет ли переопределен оператор *.

Dereference Visibility 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 `->`.

`Indirection Visibility` 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 `>>`).

`Stream Visibility` определяет раздел, в который будут помещены операторы потоков.

`InlineStreamOperations` определяет, будут ли операторы потоков определены как `inline` подстановка.

Вопросы для повторения

1. Как ассоциируются классы с языком программирования C++?
2. Какие типы объектов можно установить и для чего они используются?
3. Как изменить зависимости классов?
4. Как просмотреть исходный код?
5. Какие установки свойств доступны на вкладке C++?

Глава 14. Создание кода класса на Microsoft Visual C++

Вступительные замечания

В предыдущей главе мы ознакомились с возможностями создания кода класса на языке C++, отвлеченном от конкретного компилятора. Как обычно, более универсальный подход является и более сложным, для создания необходимого кода приходится оперировать большим количеством установок.

Более простой путь для создания приложения в использовании возможности Rational Rose - создавать код класса на основе библиотеки классов фирмы Microsoft - MFC. Для этого нет необходимости вручную оперировать значительным количеством установок, так как в пакет встроен модуль Model Assistant, который позволяет изменять все необходимые установки при помощи визуальных средств.

Возможности создания кода класса

Класс в Rational Rose - это описание общей структуры (данных и связей) для дальнейшего создания объектов. Для того чтобы генератор Rational Rose имел возможность создавать на основе описанной модели программный код, для каждого класса необходимо указать язык, для которого будет создаваться код. Также необходимо определить компонент, в котором этот класс будет храниться. Если в качестве языка для создания кода указан VC++, то пользователь получает доступ ко всей иерархии классов библиотеки MFC при помощи визуальных средств Model Assistant.

При создании класса необходимо указать стереотип, который влияет на получаемый исходный код класса. Так, при изменении стереотипа на struct или union, будут созданы указанные типы данных.

Как вы уже заметили, Rational Rose поддерживает обычные для классов C++ обозначения области видимости, такие как public, private, protected. Таким образом, каждый атрибут или операция в спецификации классов будет, при создании заголовочного файла класса, определен в одну из секций public, private, или protected. Также имеется возможность не создавать программный код для определенных классов.

Структура создаваемого кода класса

Для каждого создаваемого класса Rational Rose создает следующую структуру кода:

- директивы #include, которые создаются исходя из необходимости вклю
- чения атрибутов и связей классов;
- декларация класса, имя класса, тип, наследование;
- переменные Data members, которые создаются по описанию атрибутов
- класса и его связей;

на
полнения каждой операции, заданной в описании класса;
документация для каждого создаваемого класса, переменных,
методов,
заданная в описании модели;
идентификатор ID модели, который включается в код как
комментарий
для каждого создаваемого класса, атрибута или метода, заданных в текущей модели. Например, `///ModelID=3237F8CE0053`.

Ассоциация класса с языком VC++

Для того чтобы использовать класс в программном проекте, необходимо его ассоциировать с выбранным языком в нашем случае с VC++. Для этого сделаем следующее `Menu:Tools=>Visual C++=>Component Assigned Tools`. Получаем окно, показанное на рис. 14.1.

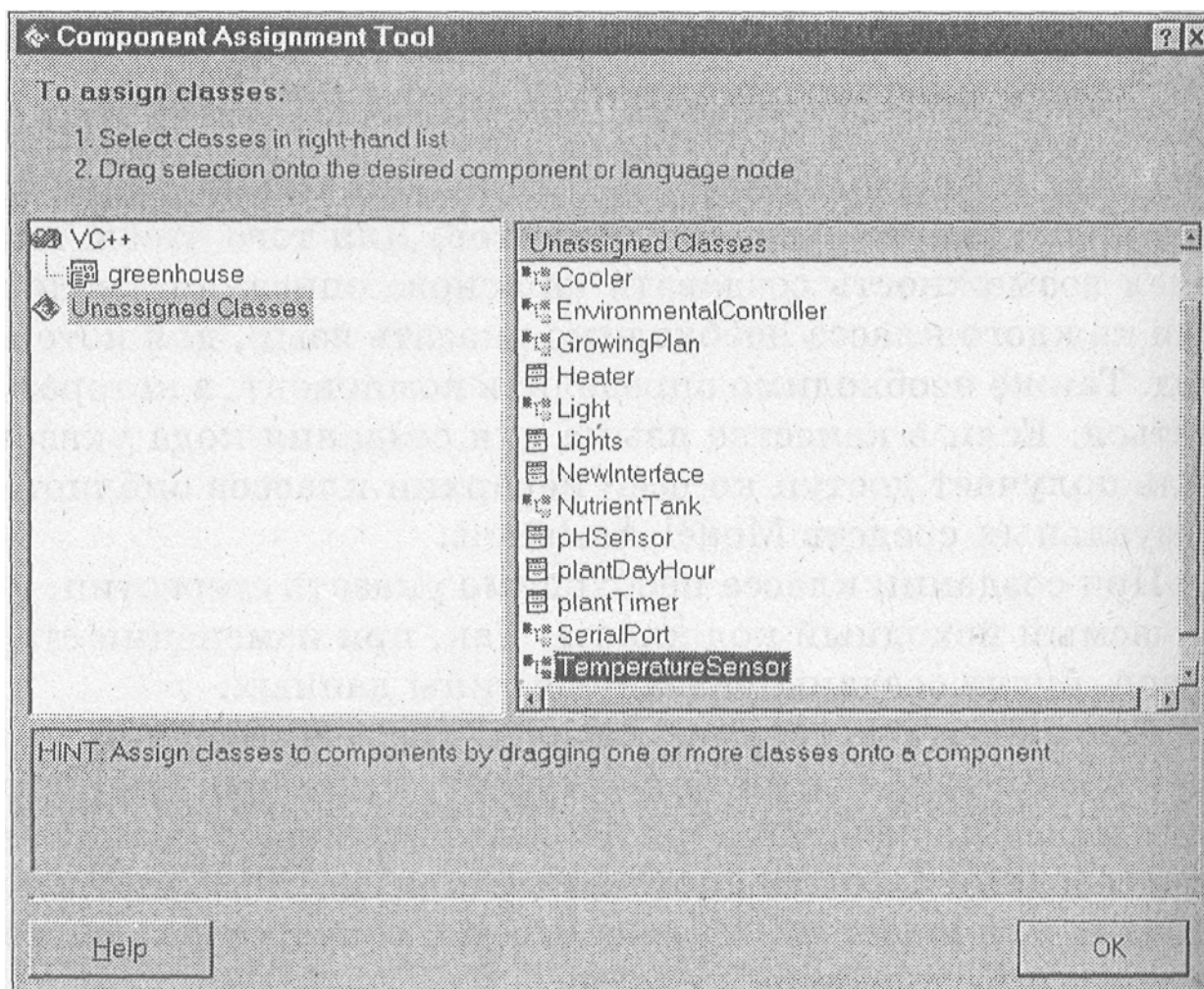


Рис. 14.1 Назначение класса в VC++ проект

В появившемся окне выбираем класс и перетаскиваем его на значок VC++. На вопрос, желаем ли мы создать VC++ компонент и

ассоциировать его с классом, отвечаем Yes и попадаем в окно выбора проекта VC++ (рис. 14.2). Здесь можно создать проект или выбрать из уже имеющихся для помещения в него нового класса. Нажмите Add и OK. У меня уже создан проект greenhouse.

Вы у себя можете создать проект с таким же именем при помощи MFC App Wizard exe (мастер создания исполняемого приложения) с типом Single document (однооконный документ).

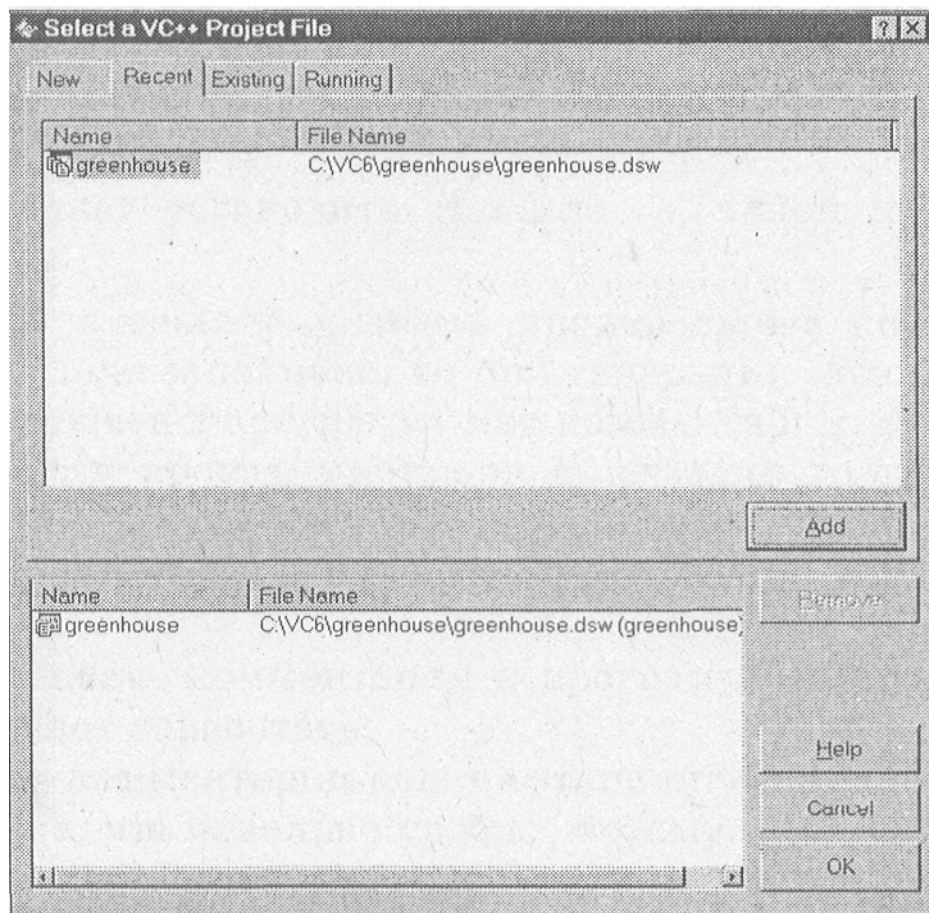


Рис. 14.2 Выбор проекта

Подробно о создании приложений Visual C++ можно прочитать, например, в книге [3]. Некоторые принципы создания приложений Visual C++ будут рассмотрены в главе 16, там же рассмотрим создание проекта при помощи мастера Visual C++.

По большому счету, класс может быть ассоциирован с любым языком, поддерживаемым генератором кода Rational Rose, и в результате ассоциации будет создан код программы на том языке, с которым ассоциирован класс, причем от конкретного языка зависят некоторые свойства класса. Поэтому неплохой идеей будет изначальная ассоциация классов с определенным языком программирования

Для того чтобы вновь создаваемые классы сразу ассоциировались с VC++ нужно проделать Menu: Tools=>Options=>Notation=>Default Language=VC++.

Меню инструментов Visual C++

После ассоциации класса с языком программирования вы можете воспользоваться пунктом главного меню Tools для Visual C++, показанном на рис. 14.3.

Model Assistant

Model Assistant позволяет обновлять и конкретизировать классы в модели, используя дополнительные ключевые слова C++ для необходимой генерации кода. Model Assistant представляет собой окно, позволяющее создавать атрибуты и операции и изменять их свойства (рис. 14.4), причем значительно проще и нагляднее, чем это происходит для C++.

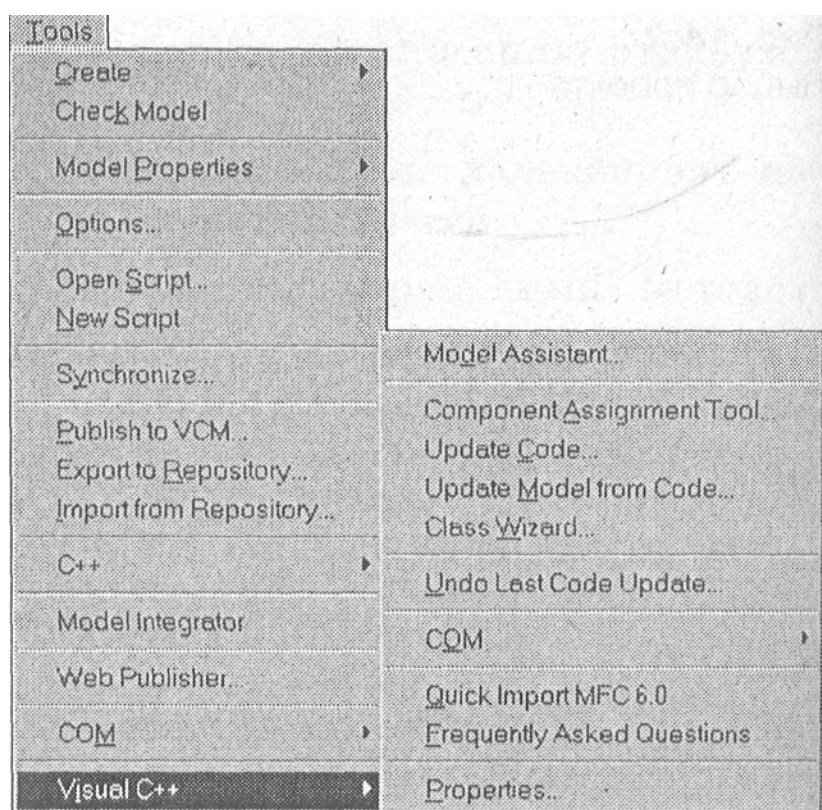


Рис. 14.3 Меню инструментов Visual C++

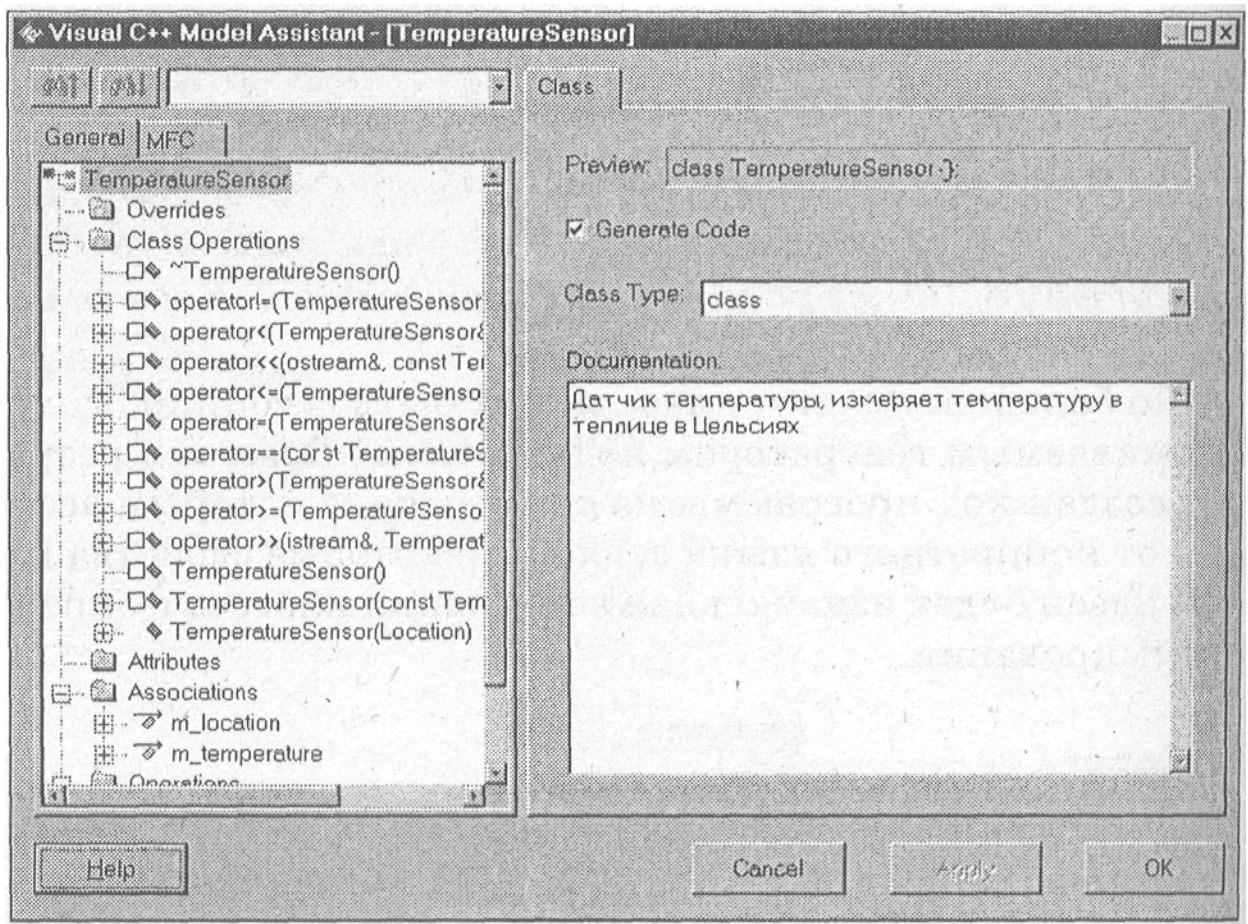


Рис. 14.4 Model Assistant для класса датчика температуры

Замечание

Те свойства, которые нельзя изменить, подсвечены серым цветом.

В этом окне вы видите следующие поля:

Preview (предварительный просмотр) показывает описание класса так, как оно определено в текущий момент;

Code Name (имя программы) показывает имя программного файла для данного класса;

Generate Code (создавать исходный текст) - ключ, определяющий необходимость создания для данного класса исходного текста на языке VC++.

Если ключ снят, то генерация кода не будет происходить, и этот класс не будет показываться в списке классов для обновления кода в окне Code Update (обновление исходного текста);

Class Type (тип класса) позволяет установить тип класса, такой как «class», «struct» или «union»;

Documentation (документация) позволяет задавать произвольные комментарии для класса. Если это поле заполнено, то при генерации исходного кода текст из него будет включен в программу как комментарии. Это

чрезвычайно удобно при просмотре программистом кода, созданного при

помощи генератора Rational Rose. По крайней мере, данная возможность позволяет создавать документацию к программе непосредственно в момент создания класса.

Я по себе знаю, как не хочется писать комментарии к программе именно в тот момент, когда она должна вот-вот заработать.

Rational Rose позволяет создавать комментарии еще на этапе проектирования, пока не написано ни строки кода, что довольно удобно, потому что именно в момент проектирования класса еще не забыты те цели и ограничения, с которыми создается проектируемый класс или метод.

Значительные возможности предоставляет Rational Rose по интерактивной установке свойств методов класса. Рассмотрим свойства операции `calibrate`, для чего активизируем строку `calibrate` в окне Model Assistant. При этом программа активизирует диалоговое окно, показанное на рис. 14.5. Это окно позволяет устанавливать и изменять атрибуты для операции, а также перегружать определенные в классе операции.

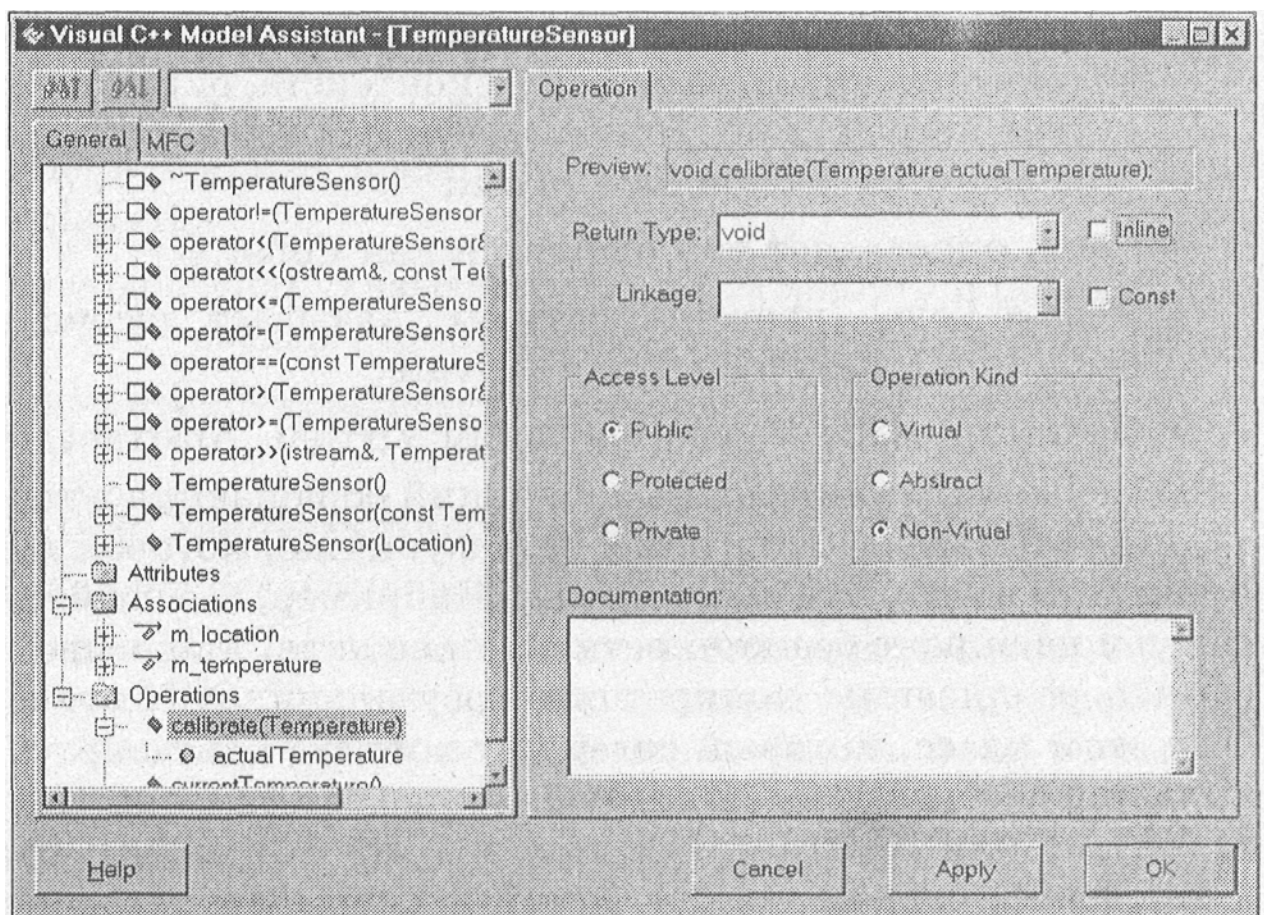


Рис. 14.5 Свойства операции `calibrate`

Совет

Для того чтобы добавить атрибут или операцию, не выходя из Model Assistant воспользуйтесь пунктом `Insert` контекстного меню.

Назначение полей в окне следующее:

Preview (предварительный просмотр) показывает описание операции таким образом, как оно было определено в текущий момент;

Code Name (наименование кода) показывает имя кода для операции. Может быть скрыто, если такое имя не задано;

Return Type (тип) позволяет выбрать из списка тип, возвращаемый операцией;

Linkage (присоединение) позволяет установить тип операции, который может быть Friend или Static;

Static обозначает, что к данной операции можно обращаться еще до создания объекта класса;

Friend определяет, что данная функция хоть и не является членом класса, но имеет доступ к его защищенным и собственным компонентам. Таким образом, определяя операцию как дружественную, мы тем самым удаляем ее из методов класса и подразумеваем, что данная функция будет описана вне класса.

Inline позволяет указать в операции ключевое слово inline, то есть операция будет создана как inline подстановка. В этом случае компилятор при создании объектного кода будет стараться подставить в текст программы код операторов ее тела. Таким образом, при многократных вызовах подставляемой функции размеры программы могут увеличиваться, однако исключаются затраты на передачу управления вызываемой функции и возвраты из нее. Как отмечает проект стандарта C++, кроме экономии времени при выполнении программы, подстановка функции позволяет проводить оптимизацию ее кода в контексте, окружающем вызов, что в ином случае невозможно. Наиболее удобны для подстановки функции, состоящие всего из нескольких строк;

Const определяет тип операции как Const.

Access Level (уровень доступа) указывает доступ к операции и может быть Public, Protected или Private;

Operation Kind - тип операции Virtual, Abstract, или Non-Virtual.

К механизму виртуальных функций обращаются в тех случаях, когда необходимо в базовый класс поместить функцию, которая должна по-разному выполняться в производных классах. Например, базовый класс может описывать фигуру на экране без конкретизации ее вида, а производные классы уже описывать реализацию конкретных треугольников, эллипсов, квадратов и т.д. При этом класс, который содержит хотя бы одну виртуальную функцию, называется абстрактным. В данном случае нет разницы между установкой пункта Virtual или Abstract. И в том, и в другом случае, будет создана функция с ключевым словом virtual, которая потребует переопределения в производных классах или как минимум создания дочерних классов для класса, имеющего виртуальную функцию.

Если щелкнуть по атрибуту класса, то открывается окно для редактирования свойств этих атрибутов, показанное на рис. 14.6, в котором можно установить основные атрибуты класса, не выходя из Model Assistant.

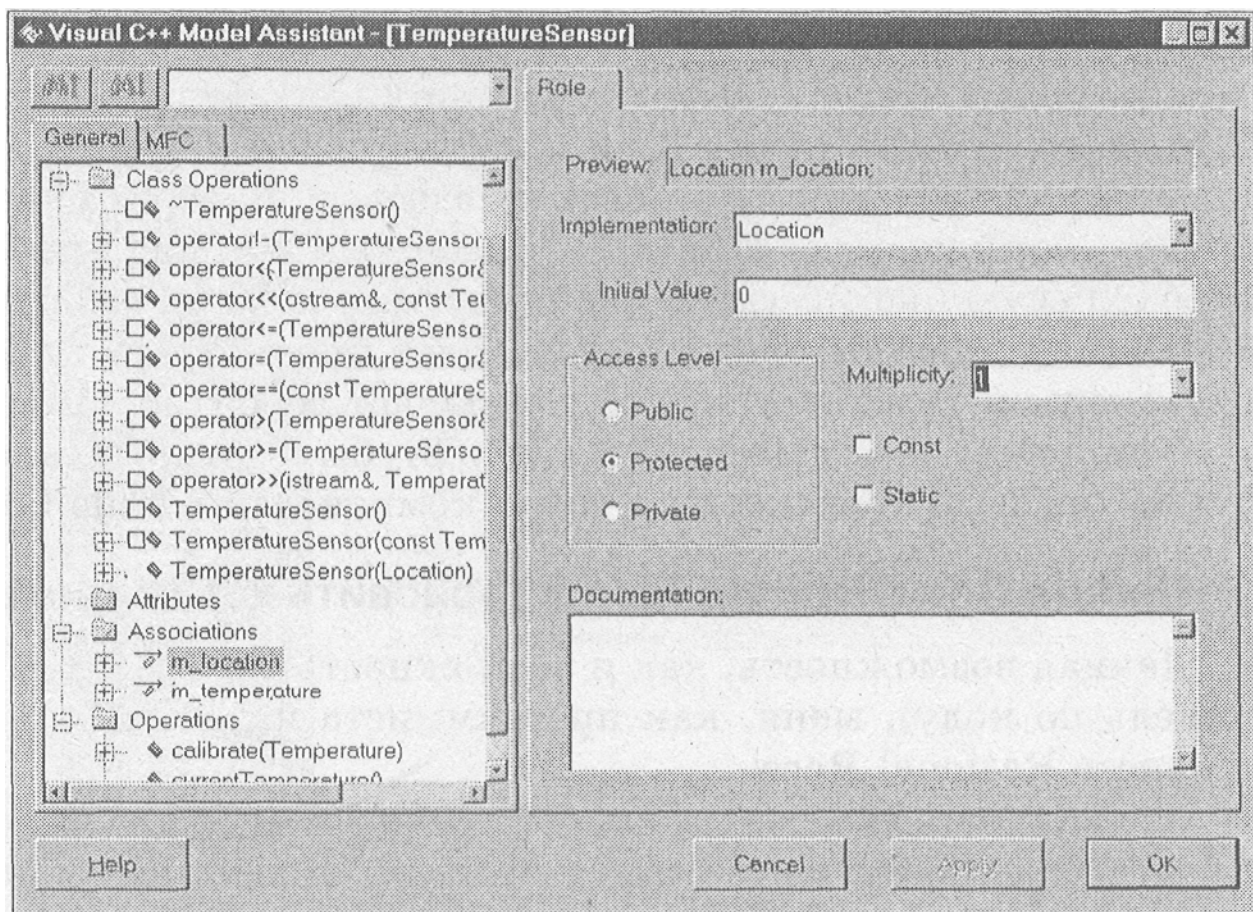


Рис. 14.6 Редактирование атрибутов класса при помощи Model Assistant

В данном окне имеются следующие поля:

Preview (предварительный просмотр) показывает описание атрибута таким образом, как оно было определено в текущий момент;

Implementation (реализация) позволяет выбрать из списка тип атрибута;

Initial Value (инициализация) позволяет устанавливать значение для инициализации атрибута;

Access Level (уровень доступа) указывает доступ к атрибуту и определяет секцию, в которой будет создан атрибут: Public, Protected или Private;

Static обозначает, что атрибут является статичным (общим для всех объектов класса);

Derived обозначает, что данный атрибут является производным;

Documentation позволяет редактировать описание атрибута.

В этом окне есть вкладка MFC, которая предназначена для классов, наследуемых из базовых классов MFC. Так как датчик температуры не наследуется ни из каких классов, то эта вкладка нам пока не понадобится и мы рассмотрим ее свойства в части IV, когда будем создавать приложение на основе классов MFC.

Замечание

Model Assistant - основное окно для работы со свойствами класса и

для создания необходимого кода, поэтому к нему также есть доступ из контекстного меню класса.

Component Assignment Tool

Активизирует диалоговое окно назначения классов в компоненты и назначения языка для класса (рис. 14.1). Это окно предоставляет возможность создания новых компонентов в модели, ассоциации компонентов с проектами на конкретных языках программирования и назначения классов в компоненты. Для того чтобы получить преимущества использования данного инструмента, необходимо создавать компоненты здесь, а не через окно Browser или в диаграмме компонентов. При этом созданные компоненты будут содержать всю необходимую информацию для генерации кода на выбранном языке программирования. Данное средство позволяет просмотреть классы, которые еще не назначены в компоненты, что уменьшает вероятность ошибки.

Component Assignment Tool может быть открыт как посредством меню Tools, так и из контекстного меню компонента в окне Code Update Tool.

Update Code/Update Model (обновить код/модель)

Данная возможность, как и возможность Update Model from code (обновить модель по коду), меня, как программиста на C++, больше всего привлекает в пакете Rational Rose.

Это возможность создать проект Visual C++ по разработанной модели и обновить модель по уже готовому проекту, созданному при помощи MFC.

Причем, что очень приятно, есть возможность не просто загрузить уже готовый код в программу Rational Rose, как это предусмотрено пунктом меню C++ Reverse Engineering, а поддерживать обмен постоянно, то есть работать именно с тем средством, которое позволяет наиболее быстро получить необходимый результат.

Например, необходимо быстро подправить что-либо в уже готовой, работающей программе, причем, как всегда бывает, это необходимо было сделать «еще вчера». Программист быстро «подкручивает» что-то в исходном коде, добавляет или изменяет методы и атрибуты и быстро сдает работающую программу. Затем просто, выбирает пункт Update Model from code и эти изменения тут же попадают на рабочий стол Rational Rose. Теперь с ними можно нормально работать и сопровождать.

Ведь не секрет, что сопровождение это кошмарный сон программиста. Часто проще написать код заново, чем разбираться в многочисленных «заплатках», сделанных в разное время и иногда даже разными людьми.

Здесь же «заплаток» нет принципиально. Вся программа составляет одну обзримую модель.

Допустим, что у нас уже создан проект greenhouse (у вас он должен уже! быть, если все сделано правильно), и мы знаем, что после того как

последний раз изменялась модель, исходный код класса датчика температуры был исправлен и его необходимо обновить.

Выберем пункт Update Model from code. Причем в контекстном меню класса, ассоциированного с VC++, также есть этот пункт, только он называется Update Model, что, впрочем, не имеет никакого значения, так как действия, выполняемые этими пунктами, одинаковы. После выбора появится окно с описанием дальнейших действий, его можно погасить, выбрав кнопку Next (далее), и, к тому же, еще предотвратить его назойливое появление в дальнейшем путем установки флажка в поле Don't show this page in the future (больше не показывать эту страницу). После этого появится окно, показанное на рис. 14.7.

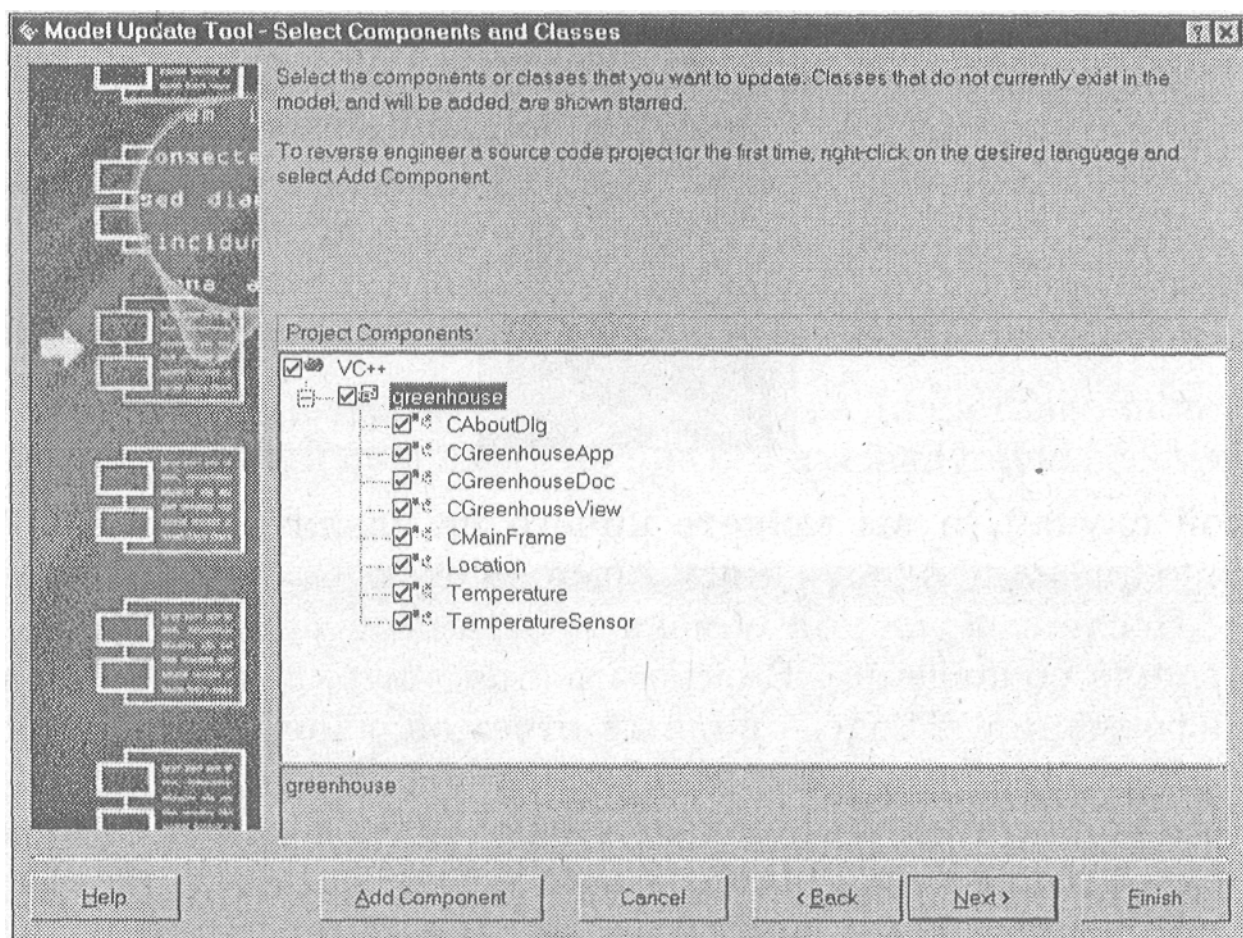


Рис. 14.7 Окно Update Model Tool

Здесь можно обновить как все классы модели, так и отдельные классы при помощи установки и снятия отметок с определенных классов.

Если классы модели еще не ассоциированы ни с одним проектом VC++-, то при помощи кнопки Add Component (добавить компонент) это можно сделать прямо из данного окна.

Некоторые классы проекта могут иметь ошибки или недостаток данных при создании их в Rational Rose с последующей генерацией кода или же при переносе данных из готового кода в модель Rational Rose. Такие классы отмечаются знаком вопроса в ярко желтом кружочке. Если выбрать такой знак вопроса, то появится сообщение Rational Rose, которое указывает на возникшую проблему или ошибку в классе, и

предлагает методы ее устранения.

Затем Rational Rose получает информацию из проекта Visual C++, для этого загружается Microsoft Visual Studio и активизируется нужный проект Visual C++. После того как обмен произошел, может быть активизировано окно удаления компонентов. Если вы проводили эксперименты с полученным кодом, а затем удалили некоторые классы, операции или атрибуты, то программа отследит, что элементы существуют в модели Rational Rose, но никак не отражены в исходном коде. Программа считает, что эти компоненты были удалены из исходного кода и предлагает их удалить и из модели (рис. 14.8).

Внимательно отнестись к этому вопросу. Возможно в проект Visual C++ были добавлены классы, например при помощи Class Wizard и они еще не отражены в модели Rational Rose. В этом случае необходимо провести обновление кода при помощи функции Update Code (обновить код). У нас именно такой случай, и вы можете ничего не удалять, если не будете устанавливать флажки на предложенных компонентах.

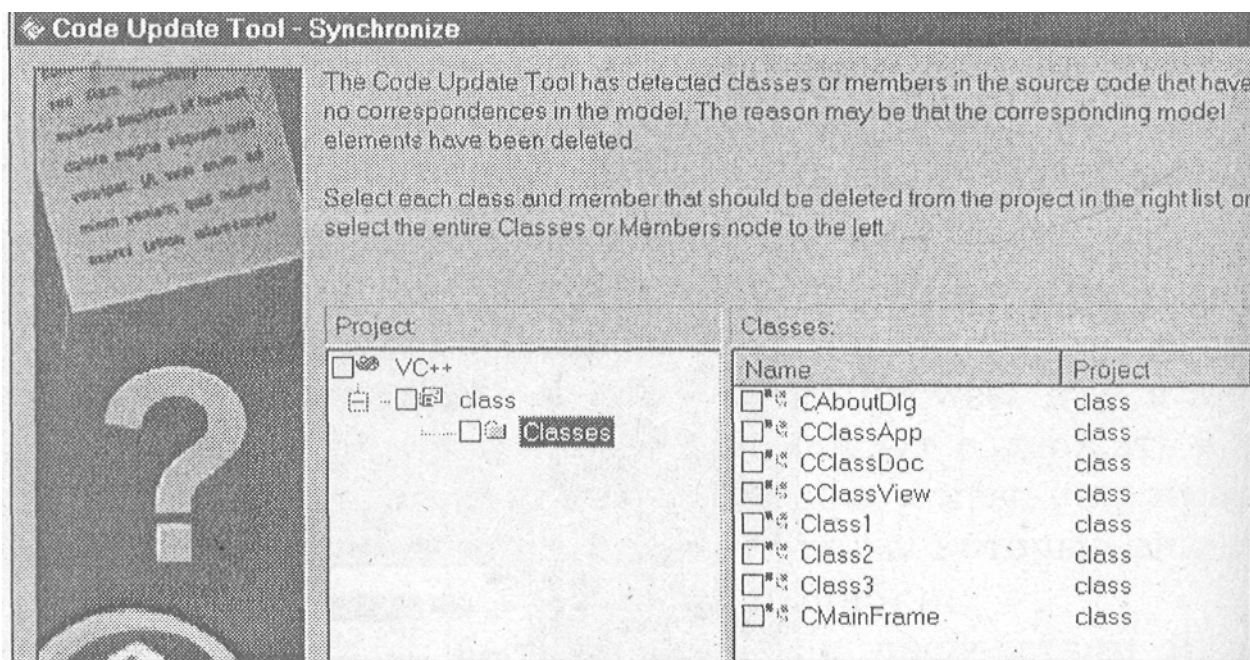


Рис. 14.8 Синхронизация кода и модели

После завершения обмена программой будет представлен отчет о том, как прошло обновление. В окне имеется две вкладки: Summary - краткая общая информация и Log - полный отчет об обновленных классах (рис. 1.4.9). Если все прошло нормально, то ошибок и предупреждений быть не должно, как показано на рисунке. Фатальная ошибка будет возникать, если на компьютере установлен пакет Visual Studio 5, а не 6, с которым работает Rational Rose 98i-2000.

При этом будет возникать ошибка обновления, и процесс завершится неудачно.

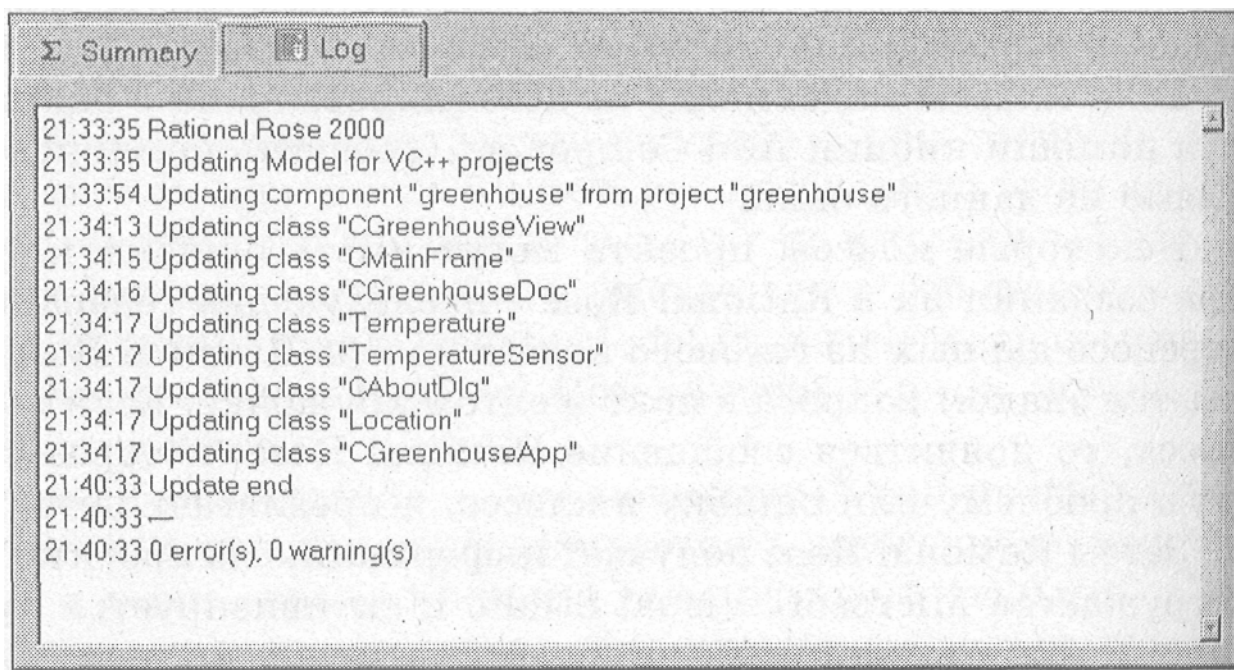


Рис. 14.9 Отчет о проведенных обновлениях компонентов

Совет

Если на вашем рабочем месте установлены обе версии Visual Studio, то для того чтобы Rational Rose мог обнаружить версию 6, перед обновлением кода просто откройте в Visual Studio пустой проект.

Процесс обновления кода по изменениям в модели происходит аналогично. Причем Rational Rose позволяет выбрать конкретные классы, которые необходимо обновить.

Class Wizard (мастер создания класса)

Class Wizard помогает создавать новые классы. Как и большинство мастеров, он посредством последовательно активизируемых окон ведет пользователя к созданию класса с необходимой информацией. Мастер предоставляет возможность устанавливать три различных варианта создания классов:

Создание нового, пустого класса.

Создание подкласса уже созданного класса.

Создание нового класса по шаблону уже существующего.

Undo Last Code Update (отмена последнего обновления)

Пункт Undo Last Code Update позволяет активизировать диалоговое окно, где вы можете заменить полученный исходный код на его предыдущую версию (рис. 14.10).

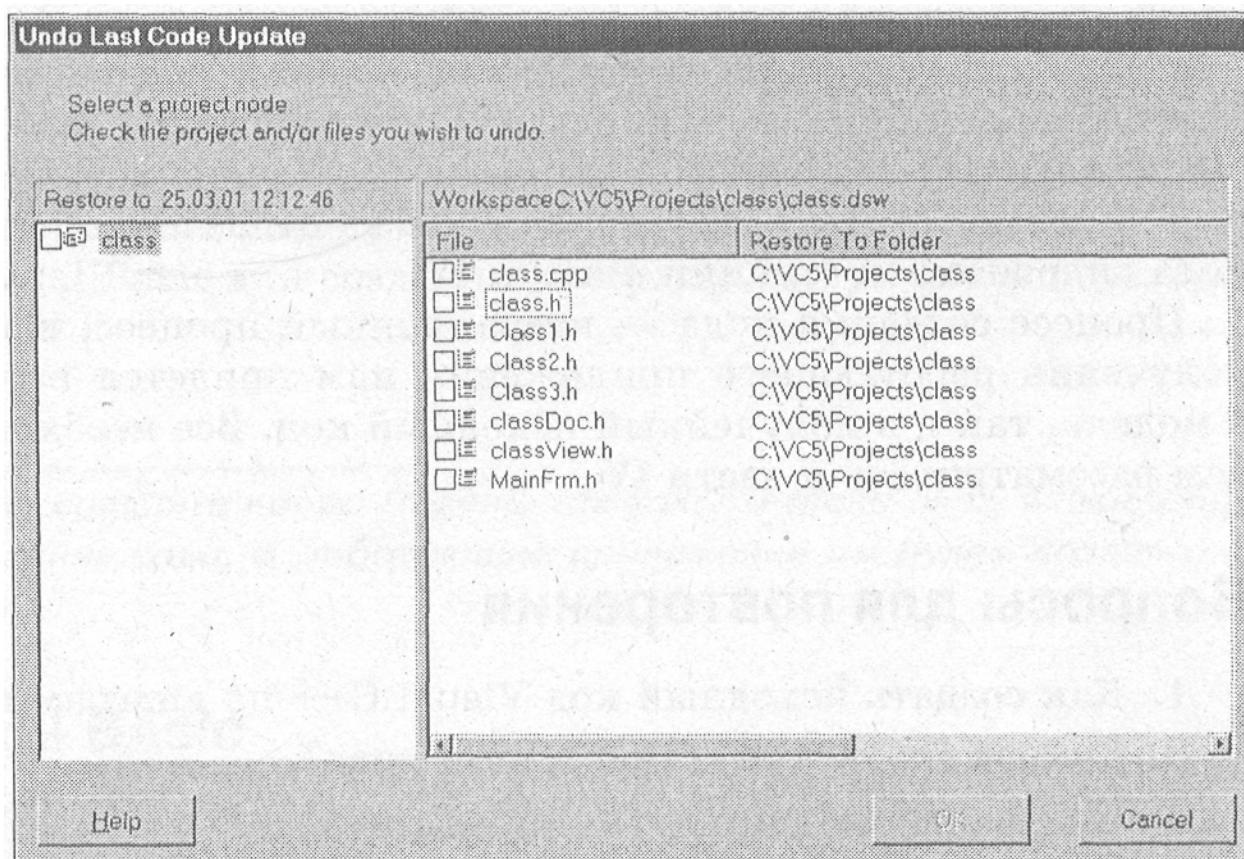


Рис. 14.10 Диалог отмены последнего обновления можете отменить только самое последнее обновление кода.

COM

Активизирует меню установок объектов COM. Мы не будем рассматривать это меню по причине того, что в нашей системе таких объектов нет.

Quick Import MFC 6.0

Позволяет импортировать классы библиотеки классов MFC в текущую модель, для того чтобы ими можно было воспользоваться при создании иерархии приложения.

Совет

Для создания модели с уже импортированной структурой классов MFC можно воспользоваться шаблоном VC6 MFC6. О в окне создания новой модели.

Properties

Позволяет устанавливать свойства генератора Visual C++, которые влияют на создаваемый код. Мы не будем на них подробно останавливаться по причине того, что для нашей задачи достаточно установок по умолчанию.

Создание кода класса

Как вы заметили, Visual Studio 6.0 взаимодействует с Rational Rose, и Rational Rose является прекрасным дополнением к Visual Studio. Таким образом, для создания кода класса достаточно воспользоваться средством Model Assistant для установки необходимых свойств генерации и провести обновление кода при помощи команды Update Code. Приемы работы со средством Model Assistant несколько отличаются от установки свойств при создании кода C++, поэтому будем рассматривать их не отвлекаясь, а на примере создания кода гидропонной станции в части IV.

Процесс создания кода - итерационный процесс, в течение которого для получения работающего приложения нам придется вносить изменения как в модель, так и в полученный исходный код. Все необходимые изменения будем рассматривать в части IV.

Вопросы для повторения

1. Как создать исходный код Visual C++ по диаграмме классов?
2. Какова структура создаваемого кода?
3. Как ассоциировать класс с языком VC++?
4. Что такое Model Assistant?
5. Какие возможности у инструмента Model Assistant для создания кода класса на выбранном языке программирования?
6. Как импортировать в модель структуру классов MFC?

Глава 15. Создание кода на Visual Basic

Вступительные замечания

В предыдущих главах мы рассмотрели создание кода классов на C++ и Visual C++, так как они максимально подходят для объектно-ориентированной разработки систем. Одним из достоинств языка UML, на котором происходит разработка диаграмм в Rational Rose, является то, что диаграммы UML не зависят от языка программирования на котором будет в дальнейшем создаваться система, лишь бы Rational Rose осуществлял поддержку генерации кода на этом языке. Поэтому наряду с C++ рассмотрим возможности создания кода на языке Visual Basic.

Вы найдете много общего с процессом создания кода на Visual C++, описанным в предыдущей главе, и после прочтения этой главы сможете использовать и тот, и другой вариант, если понадобится. Те читатели, которые не собираются использовать Visual Basic для создания программ, могут смело переходить к главе 16.

Совет

Для экспериментов создайте новую модель, так как эта глава лишь иллюстрирует возможности создания кода, а работающее приложение мы будем создавать на языке Visual C++.

Классы в Visual Basic

Класс позволяет устанавливать общее для объектов поведение, общие свойства и связи. По умолчанию класс при генерации создается как Visual Basic class module. Если в классе установлен стереотип, то он преобразуется в шаблон (template), который определяет, какой проект Visual Basic будет создан. Если создать класс со стереотипом «Form», то будет автоматически создана стандартный для форм проект с процедурами обработки события. Классы со стереотипами «Enum» (перечисление) или «Type» (тип), которые помещены в другие классы модели, создаются как определение перечисления или типа в этих классах.

Классы для создания Visual Basic кода должны быть ассоциированы с компонентами Visual Basic для указания генератору необходимых данных. При этом генератор создает следующую структуру кода:

Определение класса, полученное из имени и свойств класса в модели.

Комментарии для класса, полученные из поля Documentation.

Переменные модуля, которые создаются на основе свойств класса и связей класса.

- Шаблоны для методов, включая тело метода, для всех определенных пользователем методов и процедур.

- Отладочный код для модуля.

Visual Basic поддерживает public и private секции доступа, однако,

каждое свойство, связь или метод в модели будет помещен в наиболее подходящую, с точки зрения генератора, секцию. Методы `protected` будут преобразованы как `friend` методы Visual Basic.

Иерархия классов

Для рассмотрения возможностей создания кода я сделал один родительский класс `CSensors` (датчики), наследуемый из него класс `TemperatureSensor` (датчик температуры), в котором один атрибут `Location` (местоположение) с типом `Integer` (целое) и связанный с ним класс `Temperature` (температура), который должен быть отражен в классе датчика температуры как переменная с типом `Temperature`. Такая иерархия позволяет рассмотреть наследование и использование классов, чего для большинства случаев достаточно (рис. 15.1).

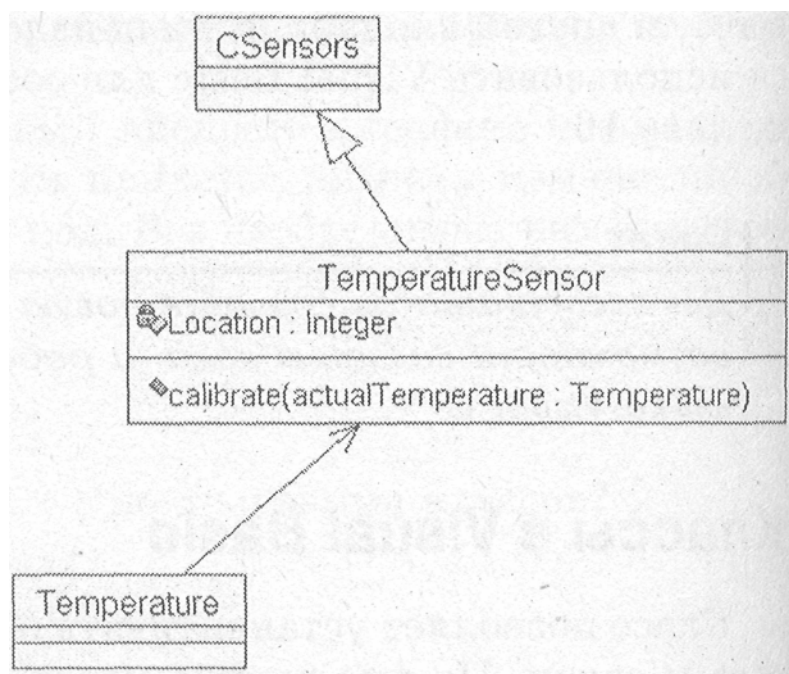


Рис. 15.1 Иерархия классов

Ассоциация класса с языком Visual Basic

Для того чтобы использовать класс в проекте Visual Basic, необходимо его ассоциировать с выбранным языком. Для этого активизируем диалоговое окно назначения классов в компоненты при помощи меню `Menu: Tools ==> Visual Basic ==> Component Assigned Tools` (рис. 15.2).

Это окно аналогично представленному в предыдущей главе для назначения в проект Visual C++ и приемы работы с ним аналогичны. Перетаскиваем необходимые классы на строку Visual Basic, после чего активизируется диалоговое окно выбора проекта Visual Basic (рис. 15.3). Здесь имеется возможность создать новый проект или выбрать из уже существующих.

Создайте проект Standard EXE. По умолчанию название проекта

будет Project1.

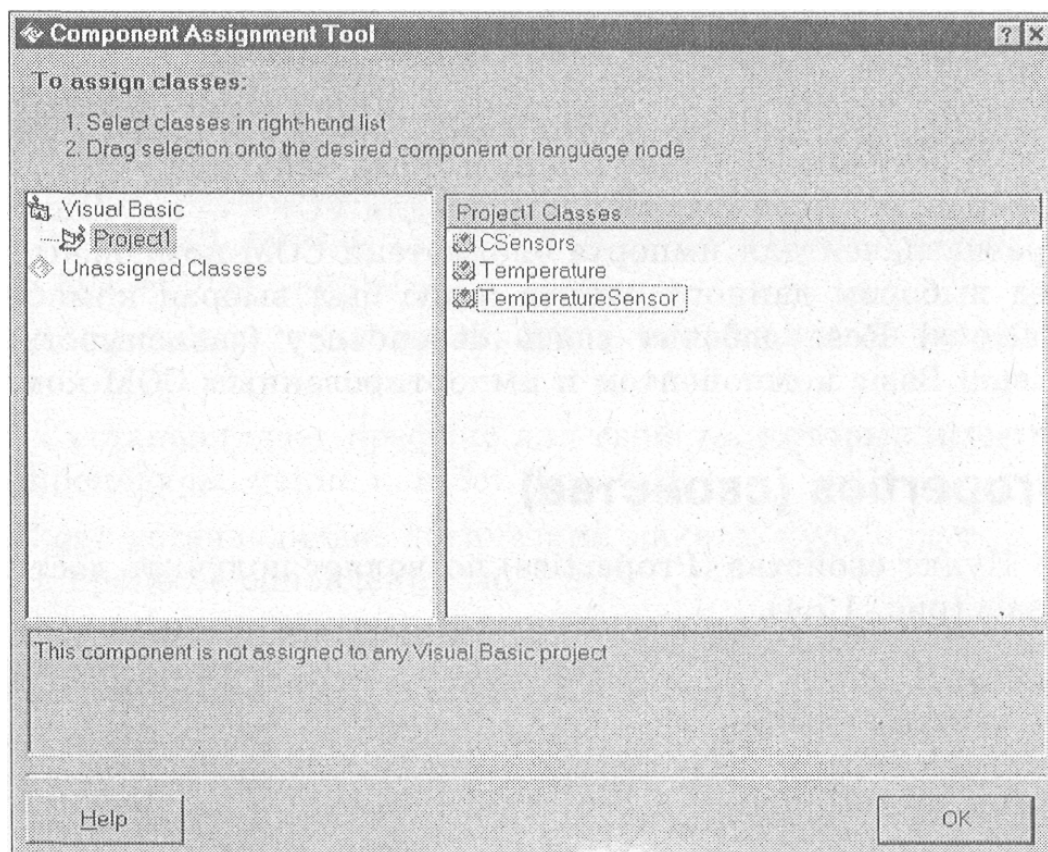


Рис. 15.2 Назначение класса в проект Visual Basic

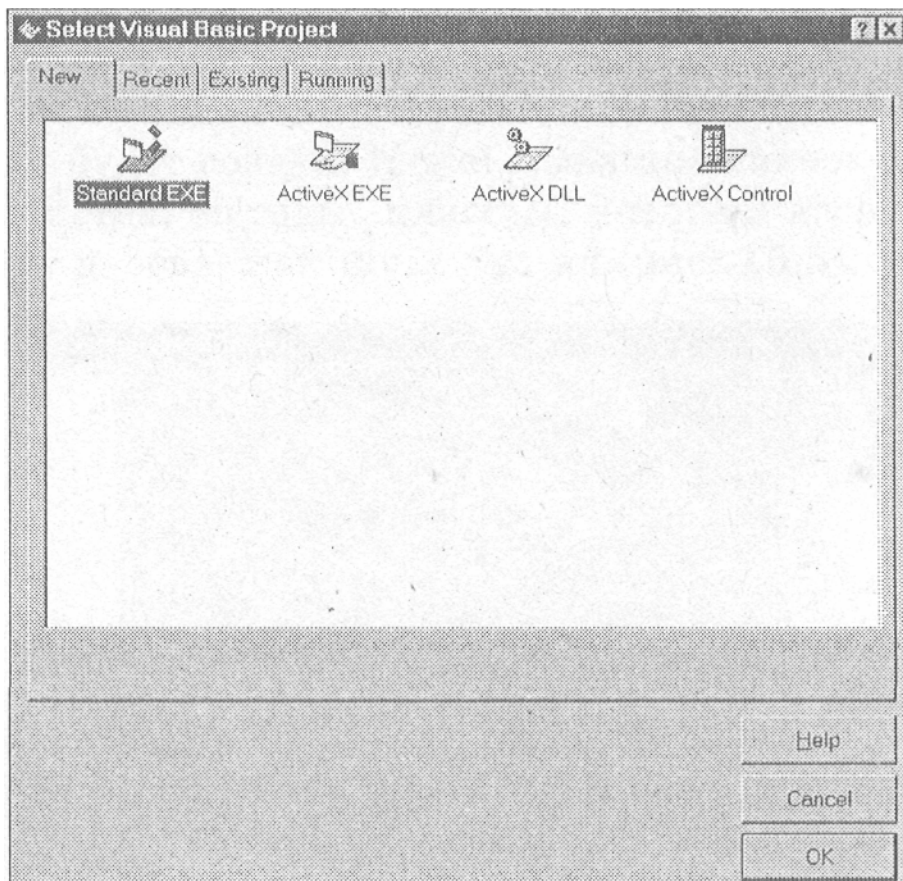


Рис. 15.3 Выбор проекта Visual Basic

Visual Basic создаст стандартный проект с формой, которой присваивается название по умолчанию Form1.

Совет

Для того чтобы вновь создаваемые классы сразу ассоциировались с Visual Basic, необходимо установить его как язык по умолчанию для создания классов Menu: Tools=>Options=>Default Language=Visual Basic.

Меню инструментов Visual Basic

Меню Visual Basic, которое доступно в меню инструментов (Tools), аналогично изученному в предыдущей главе меню для VC++. Единственный пункт, который не присутствовал в меню для VC++ - это Add Reference. Этот пункт предназначен для импорта библиотеки COM-компонентов в модель. Если перед выбором данного пункта меню был выбран компонент Visual Basic, то Rational Rose добавит связь dependency (зависимость) между выбранным Visual Basic компонентом и импортированным COM-компонентом.

Properties (свойства)

Пункт свойства (Properties) позволяет получить доступ к свойствам

Visual Basic (рис. 15.4).

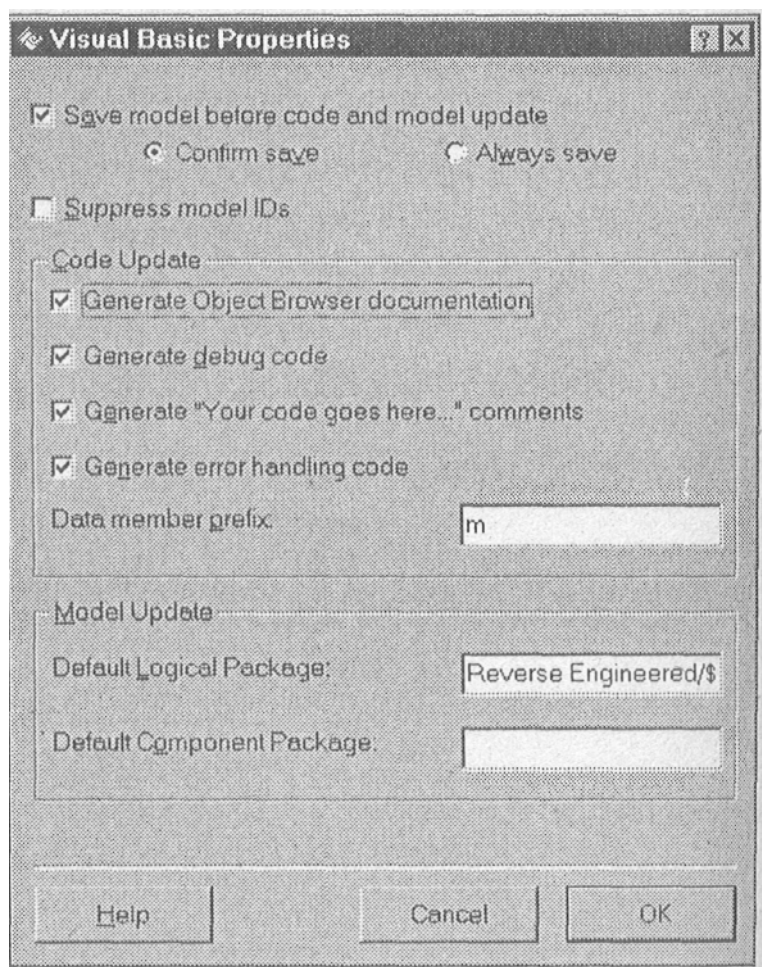


Рис. 15.4 Visual Basic Properties

Диалоговое окно Properties предоставляет возможность устанавливать параметры генерации кода Visual Basic и позволяет устанавливать следующие свойства.

Save model before code and model update позволяет устанавливать сохранение модели перед обновлением кода;

Confirm save позволяет включить запрос пользователю на подтверждение сохранения;

Always save позволяет сохранять без запросов пользователю;

Suppress model IDs устанавливает, будут ли включаться идентификаторы в создаваемый код;

Generate object Browser documentation устанавливает, будет ли включаться документация в создаваемый код;

Generate debug code устанавливает, будет ли создаваться код отладки;

Generate «Your code goes here» comments позволяет устанавливать, будут ли создаваться комментарии, которые указывают, что в данное место необходимо добавить код метода класса;

Generate error handling code устанавливает, будет ли создаваться код обработки ошибок;

Data member prefix устанавливает префикс для свойств, которые имеют ассоциированные процедуры, такие как Set, 'Let, Get;

Default logical package устанавливает логический пакет, куда будут добавляться классы в процессе обновления модели;

Default component package устанавливает пакет, куда будут добавляться компоненты в процессе обновления модели.

Model Assistant

Model Assistant позволяет при помощи графических средств устанавливать дополнительные параметры для генерации кода, просмотреть и создать новые элементы (member) Visual Basic класса.

Приятным отличием от Model Assistant Visual C++ является то, что прямо в нем можно просмотреть код, который будет создан. Model Assistant показывает каждый UML элемент (связи, свойства, методы, классы), который ассоциирован с языком Visual Basic, в виде дерева, как показано на (рис. 15.5).

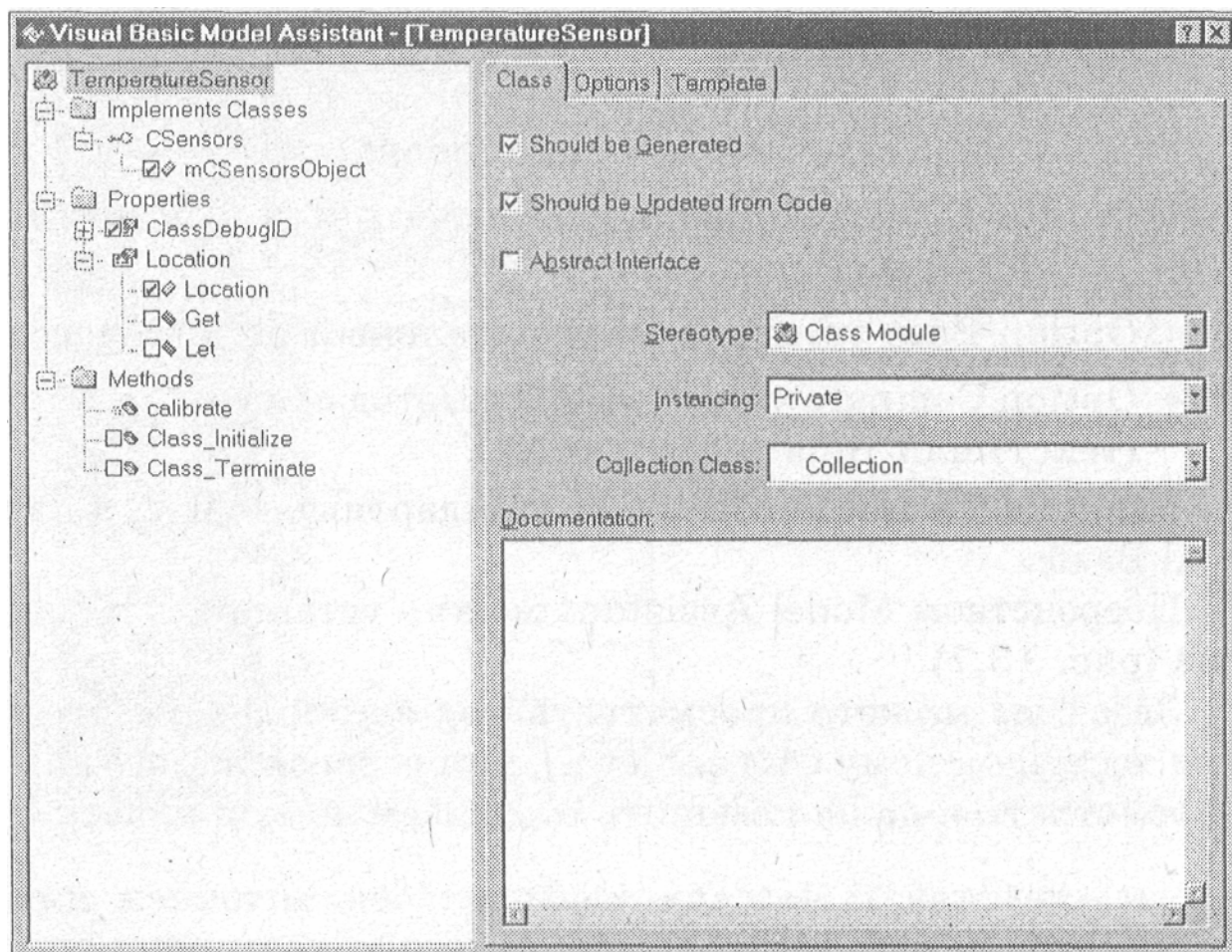


Рис. 15.5 Model Assistant для класса Visual Basic

Во вкладке Class вы видите следующие поля:

Should Be Generated устанавливает, будет ли создаваться код для данного класса;

Should Be Updated from code устанавливает возможность обновления данного класса по изменениям, внесенным непосредственно в исходный код;

Abstract Interface устанавливает, будет ли данный класс абстрактным.

Если класс абстрактный, то Visual Basic создаст пустое тело класса;

Stereotype определяет тип класса;

Instancing определяет, будет ли данный модуль класса виден вне проекта,

и определяет правила реализации класса во внешнем проекте;

Collection определяет имя collection class;

Documentation дает возможность заполнить документацию.

Вкладка Options предназначена для задания дополнительных параметров генерации (рис. 15.6).

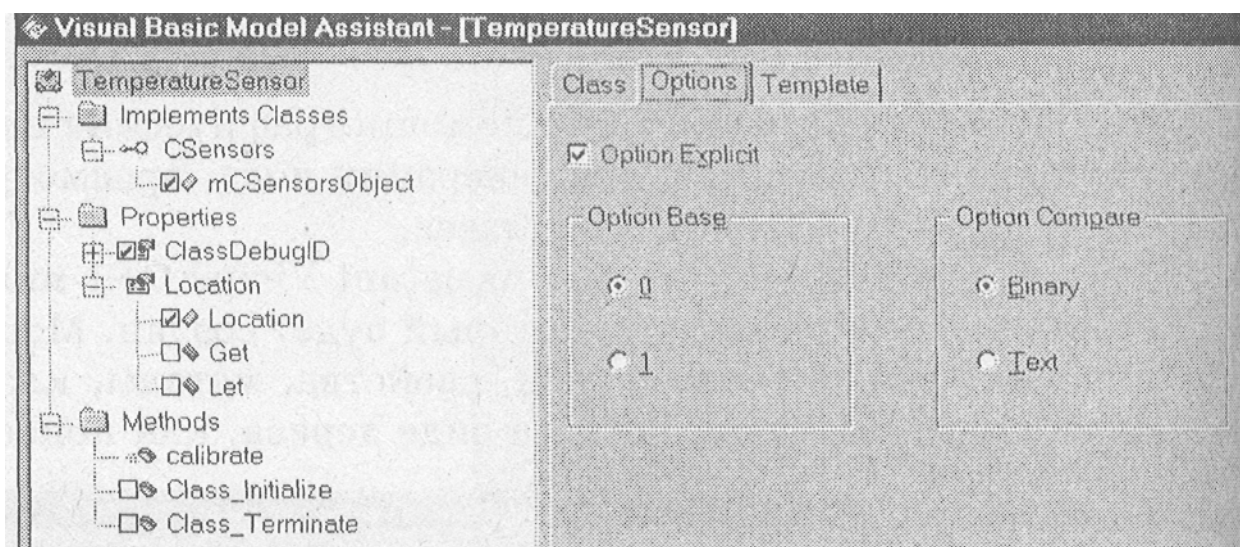


Рис. 15.6 Вкладка Options Visual Basic Model Assistant

Здесь присутствуют следующие поля:

Option Explicit позволяет включать или выключать подробное описание

установок в создаваемый код;

Option Base устанавливает начальный индекс массивов 0 или 1;

Option Compare устанавливает метод сравнения текстовых строк как Text (текст) или Binary (двоичный).

Вкладка Template содержит стандартные события для модуля класса в Visual Basic.

Посредством Model Assistant можно установить параметры свойств и методов (рис. 15.7).

Здесь вы можете просмотреть код заголовка метода (Preview), установить тип доступа к нему (Access level), тип возвращаемого

значения (Type), а также просмотреть и даже изменить создаваемый код класса.

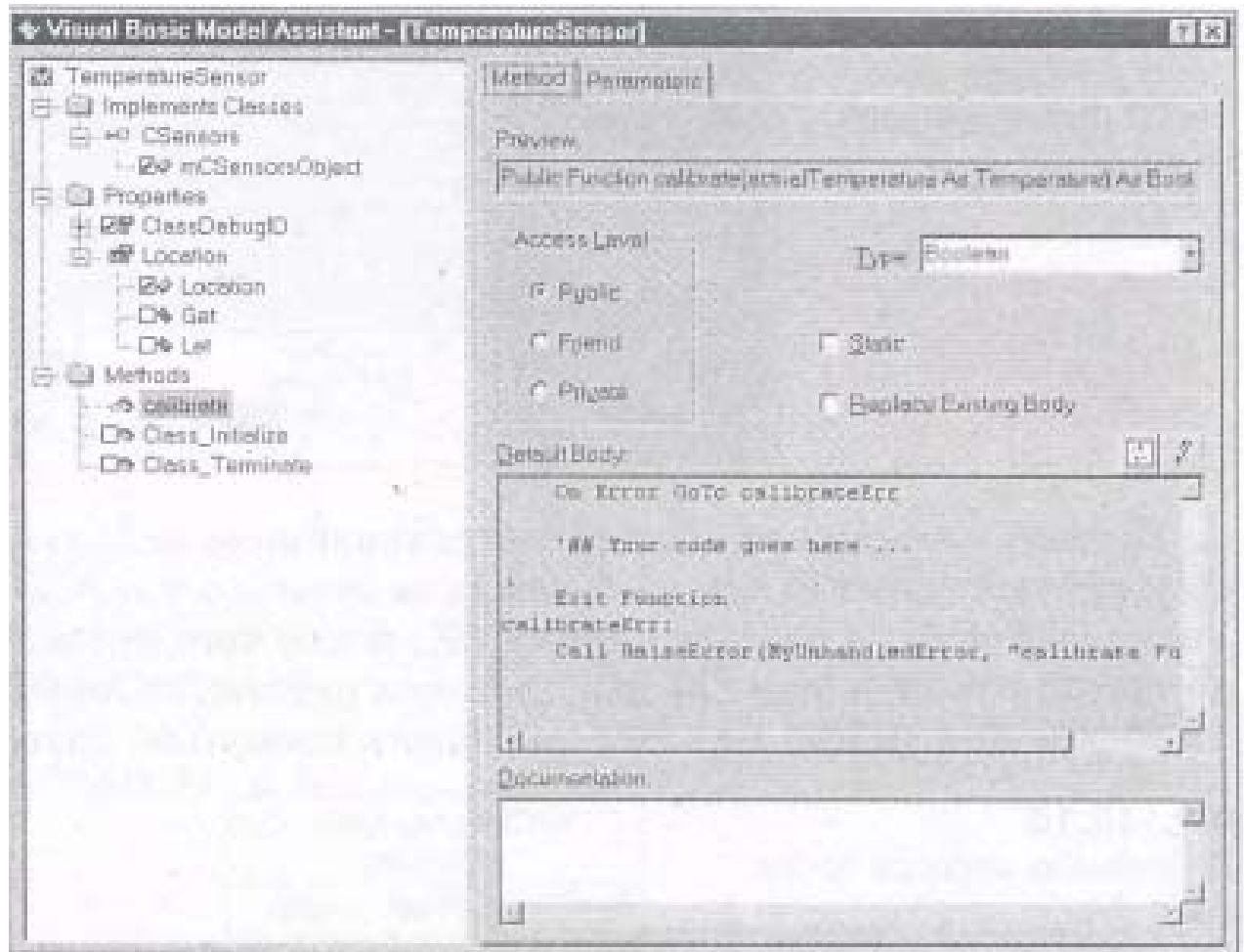


Рис. 15.7 Окно Model Assistant для метода calibrate

Создание кода класса

Теперь попробуем создать код класса с представленной иерархией и посмотрим, что получится.

Выберите из контекстного меню класса или меню Tools пункт Update Code. Будет активизировано диалоговое окно Component Update Tool (рис. 15.8).

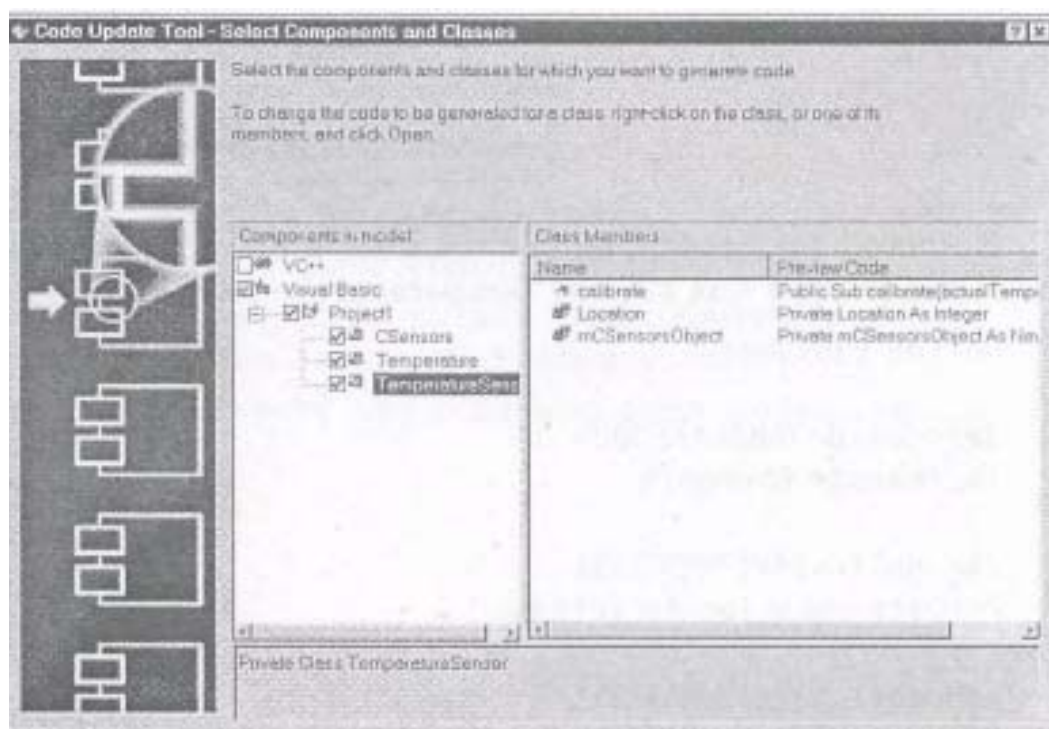


Рис. 15.8 Component Update Tool для проекта Visual Basic

Установите отметки напротив необходимых классов и нажмите кнопку Next. После завершения процесса обновления будут созданы новые классы и предложено удалить класс Form1, которого нет в модели (рис. 15.9).

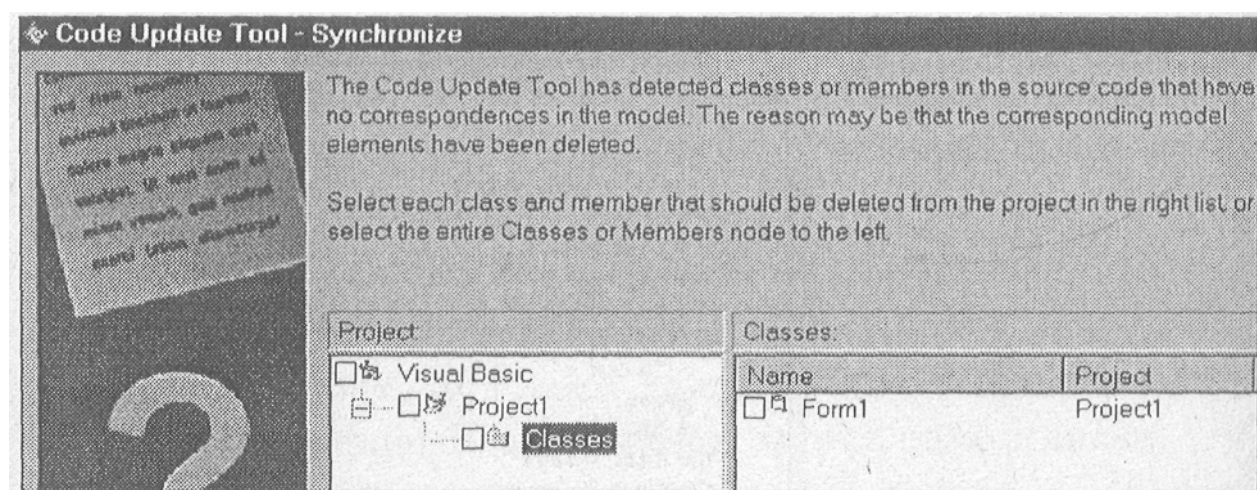


Рис. 15.9 Запрос на удаление Form1 проекта Visual Basic

Удалять его или нет, зависит от дальнейшего использования проекта. В нашем случае удалять его не будем, для этого достаточно не устанавливать отметку на Form1, а нажать кнопку ОК, после чего процесс должен благополучно завершиться, и будет выдан протокол работы, который я здесь не привожу.

Теперь диаграмма классов приобрела несколько другой вид (рис. 15.10).

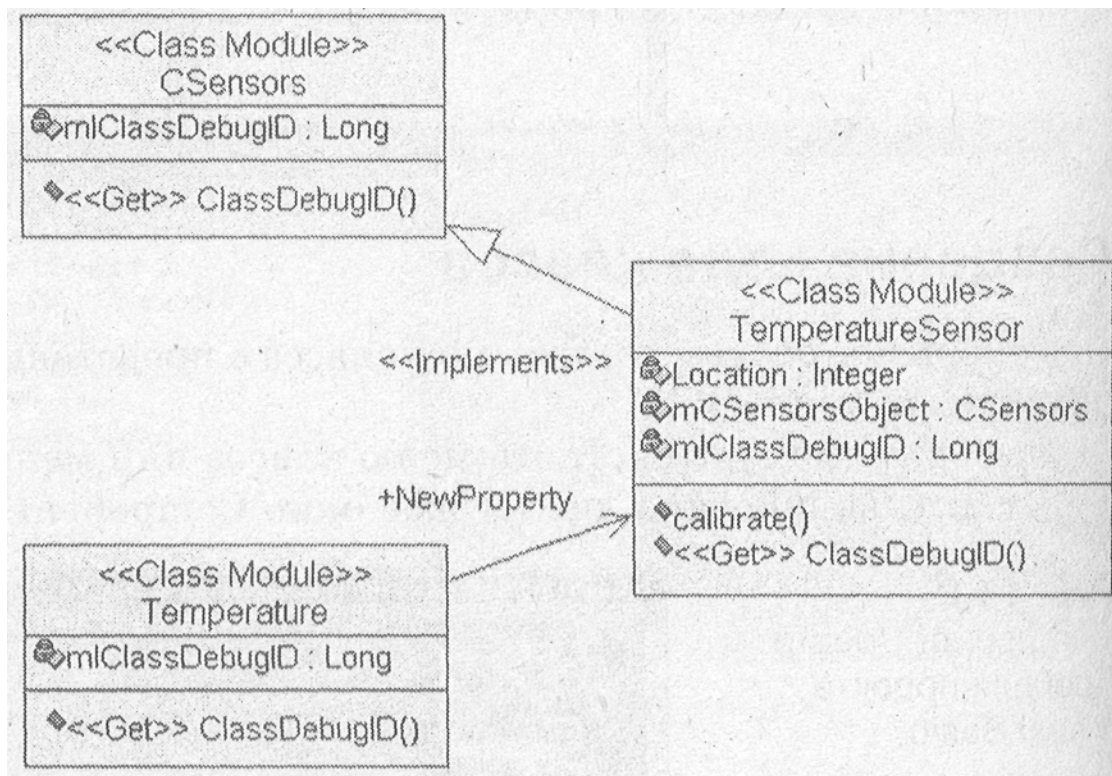


Рис. 15.10 Диаграмма классов после обновления кода проекта Visual Basic

В классы был добавлен код отладки и все необходимое для него, и был создан следующий код класса TemperatureSensor:

Option Explicit

```

##ModelId=3ABEDAA500D2 Implements CSensors
##ModelId=3ABED9CF0334 Private Location As Integer
##ModelId=3ABF82BE02DA
Private mCSensorsObject As New CSensors
##ModelId=3ABF92080118 Private mIClassDebugID As Long
##ModelId=3A8ED9D4021C
Public Sub calibrate(actualTemperature As Temperature) On Error
GoTo calibrateErr
## Your code goes here
Exit Sub calibrateErr:
Call RaiseError(MyUnhandledError, "calibrate Sub") End Sub
##ModelId=3ABF9207026C
Public Property Get ClassDebugID() As Variant On Error GoTo
ClassDebugIDErr
ClassDebugID = mIClassDebugID
Exit Property ClassDebugIDErr:
Call RaiseError(MyUnhandledError, "ClassDebugID Property")
End Property
  
```

Я не буду приводить код остальных классов, вы его можете получить сами, отмечу лишь, что, рассмотрев создание кода класса на

таких, я считаю, достаточно полярных языках как Visual C++ и Visual Basic, мы обнаружили в этом процессе много общего. Таким образом, если вам понадобится создать код класса на других языках, которые поддерживаются программой Rational Rose, вы, действуя аналогично, справитесь и с этой задачей.

Вопросы для повторения

1. Как создать код Visual Basic по диаграмме классов?
2. Как ассоциировать классы с языком Visual Basic?
3. Какие возможности имеются у инструмента Model Assistant для создания кода класса на выбранном языке программирования?
4. Какие шаги нужно предпринять для создания кода класса на языке Visual Basic?

Часть 4. Создание работающего приложения на VC++

Глава 16. Создание шаблона приложения

Стратегические и тактические решения

До сего момента вся предыдущая работа была больше направлена на изучение Rational Rose и подготовку к созданию приложения, разработку общей структуры классов и их взаимодействия. И вот, наконец, наступил момент для создания работающего приложения. При этом нужно быть готовым, что стратегические решения, принятые нами на предыдущем этапе, могут противоречить тактическим решениям, которые необходимо принять для разработки приложения.

Стратегическими решениями я называю решения, от которых зависит работа программы. Это алгоритмы работы, форматы хранения данных, структура классов и так далее. Обычно такие решения принимаются при проектировании задачи постановщиком и отражаются в спецификациях на программу.

Для того чтобы реализовать в программе стратегические решения, требуется принятие большого количества тактических решений. Обычно тактические решения принимаются самим программистом и целиком и полностью зависят от опыта работы.

К тактическим решениям можно отнести решения конкретной реализации переменных, функций и классов, их имена, области видимости, взаимодействие в пределах, обозначенных спецификациями алгоритма. Из этого вытекает, что чем подробнее спецификации программы, тем меньше тактических решений необходимо принимать программисту. И, соответственно, программист может меньше влиять на создаваемый код.

Какие же решения проектировщики оставляют на откуп программистам? Это может быть выбор между использованием ссылки или значения переменных, использования той или иной библиотечной функции или класса для реализации общего алгоритма, использование тех или иных конструкций языка, например, использование конструкции `goto`.

Тактические решения незаметны снаружи программы, но оказывают серьезное влияние на быстродействие, объем требуемой памяти, легкость дальнейшего сопровождения и время разработки программы. По принятым тактическим решениям можно в значительной степени судить о квалификации программиста.

Почему же при постановке задач уделяется на порядок меньше внимания тактическим решениям, чем стратегическим? Ведь проектировщик по определению опытнее программиста и может подсказать то или иное решение в конкретной ситуации. Только по причине большой трудоемкости описания таких решений. Проще сразу

написать программу, чем описывать такие решения на бумаге.

Но с появлением Rational Rose ситуация в корне изменилась. Теперь уже при постановке задачи можно задать посредством диаграммы имена используемых переменных и классов, области видимости, использование библиотечных классов, что значительно сужает область принятия решений конечным программистом и повышает качество программного кода.

На качество программного кода также влияет соглашение об использовании имен, основы которого рассмотрим в следующем параграфе.

Соглашение об использовании имен

Корпорация Microsoft разработала для программистов специальное соглашение об использовании имен в программах. Согласно этому соглашению для функций используются имена, построенные из глаголов и существительных, причем первые буквы этих слов - заглавные.

Для имен переменных Microsoft предлагает более сложную систему, предусматривающую обозначение именуемых типов данных. Для этого используется небольшой префикс из строчных букв, а собственно имя начинается с заглавной буквы. Далее приведена таблица префиксов, предлагаемых Microsoft.

Таблица 2. Префиксы переменных

Префикс	
b	Булевский (байт)
c	Символ (байт)
S	Строка (char или CString)
dw	Длинное беззнаковое целое (DWORD)
f	16 битный флаг (битовая карта)
fn	Функция
h	Дескриптор (handle)
I	Длинное целое (long)
i	Данные типа Int
Ip	Длинный указатель (long pointer)
n	Целое (16 бит)
p	Указатель (pointer)
pt	Точка (два 32 битных целых)

w	Целое без знака (WORD, 16 бит)
sz	Указатель на строку, заканчивающуюся 0 (string zero)
Ipsz	Длинный указатель на sz (long pointer string zero)
rgb	Длинное целое, содержащее цветовую комбинацию RGB

Для программ, написанных с применением библиотеки MFC, можно добавить, что имя класса начинается на C, а имя атрибута класса начинается на t_. Причем при создании классов при помощи VC++ автоматически создаются заголовочные файлы и файлы тела класса без буквы C в начале имени.

Использование соглашения об именах в программах не является обязательным, но при использовании соглашений программа не просто становится легко читаемой, но и значительно облегчается дальнейшее ее сопровождение. Конечно, указывать или не указывать тип переменной в ее названии - это личное дело каждого, но привычка пользоваться таким соглашением является хорошим тоном.

Как вы уже заметили, изначально в программе контроля климата тепличного хозяйства мы не придерживались таких соглашений. На то есть свои причины. Первая и самая главная причина - это то, что при написании программы Г. Буч (а именно его основными идеями мы пользовались при проектировании классов тепличного хозяйства) не придерживается таких соглашений. А так как я старался оставить имена классов и переменных такими же, то от соблюдения соглашений пришлось несколько отступить. Однако те классы, которые я создавал из тактических соображений, я постарался называть, придерживаясь указанных соглашений.

Структура приложения|

Так как мы создаем гидропонную систему на языке VC++ с использованием библиотеки MFC, то для того чтобы в дальнейшем хорошо ориентироваться в созданном коде программы, кратко рассмотрим структуру приложения MFC.

Мастер создания приложений VC++ (AppWizard) может создавать несколько типов приложений, каждое из которых применяется для своих целей. Перечислим эти типы:

Single document (приложение работает с одним документом);

Multiple document (приложение работает с несколькими документами);

Dialog based (приложение основано на окне диалога).

Для нашего случая наиболее удобна структура приложения, работающего с одним документом.

Понятие «документ» для тепличного хозяйства

Понятие «документ» широко распространилось в практике

программирования с подачи компании Microsoft. Документом теперь принято называть любую совокупность данных, а не только текста. Так, звукозапись или видео фрагмент, помещенные в компьютер, являются документом.

Для создания программ отображения и редактирования документов компания Microsoft создала шаблон работы с документами, где уже предусмотрены основные необходимые функции, такие как отображение в окне, сохранение или печать.

Данные о состоянии теплицы также можно представить в виде документа, хотя это не является обязательным. Просто это более быстрый путь для создания работающего приложения, чем путь создания приложения с нуля на базе окна диалога.

Фактически, если принять за основу шаблон документа, то можно за короткое время создать графическую систему управления процессами в теплице, которая будет отображать на экране состояние устройств в графическом виде и подчиняться оператору, который щелчком мыши может изменить состояние устройств или выбрать другую программу выращивания. Так создаются автоматизированные системы управления технологическими процессами (АСУТП).

Мы здесь не будем уделять большого внимания внешнему отображению, так как это уже не зависит от структуры классов, создаваемых при помощи Rational Rose, а зависит лишь от внешнего проявления их взаимодействия, поэтому наша система будет просто отображать состояние устройств в текстовом виде, фактически, выводить протокол работы на дисплей.

Классы, создаваемые мастером приложений

При создании шаблона приложения мастер создания приложений создаст код следующих классов:

главный класс приложения CGreenhouseApp;

класс документа CGreenhouseDoc;

класс просмотра CGreenhpuseView;

класс для окна «О программе» CAboutDlg;

класс основного окна программы CMainFrame.

Все приложения VC++ MFC являются объектами. Таким образом, приложение - это главный класс, который включает в себя все необходимые для работы классы: документов, окон просмотра. Этот объект является глобальным и создается при вызове приложения.

Обычно Мастер называет его theApp. Соблюдая соглашение об именах, Мастер создаст главный класс приложения с именем проекта, к которому прибавит в начале букву C - class (класс), а в конце App - application (приложение). И получится CGreenhouseApp, который наследуется из библиотечного класса CWinApp.

Мастер сразу переопределит функцию класса CWinApp::InitInstance(), код которой выполняется в первую очередь при загрузке приложения, и инициализирует все необходимые ресурсы для работы приложения.

Я считаю неплохой идеей разделения кода создания окна и визуализации данных от собственно данных. В этом случае можно

изменять внешнее представление (графическое или текстовое), не меняя внутреннего содержания данных.

Мастер создания приложения именно так и поступает. Создает класс документа (в нашем случае CGreenhouseDoc), в котором должна проходить вся обработка данных, и наследует его из библиотечного класса CDocument, и класс просмотра (CGreenhouseView), который отображает данные на экране компьютера. Данные отображаются в окне (класс CMainFrame), класс которого наследуется из библиотечного класса CFrameWnd. Описанная иерархия показана на рис. 16.1.

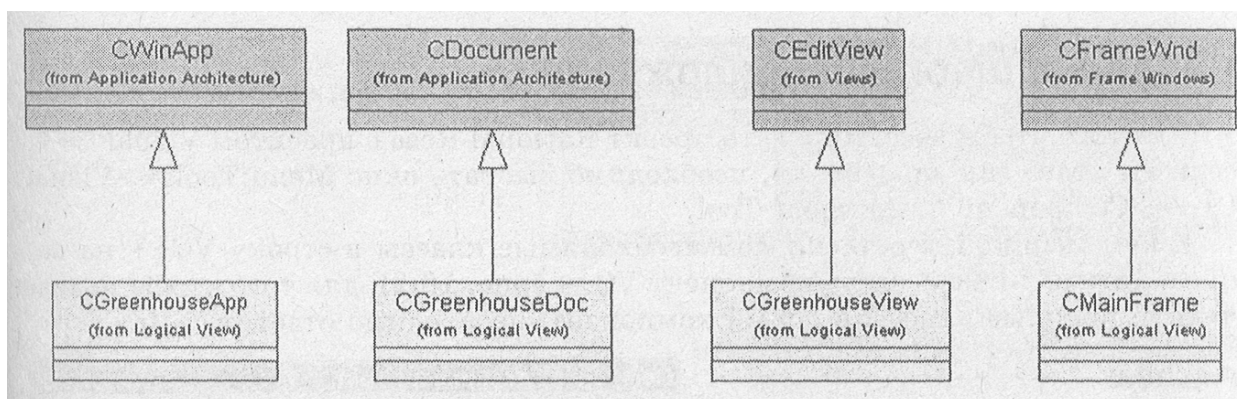


Рис. 16.1. Иерархия классов приложения

Я намеренно взял для нашего приложения класс CEditView, а не класс CView, для того чтобы проще было отображать состояние устройств простой текстовой строкой, для чего класс CEditView подходит максимально, так как он предназначен для работы именно с текстовыми документами.

Класс окна «О программе» я не рассматриваю подробно, так как он совершенно не обязателен в приложении.

Операция OnNewDocument

Важной операцией во вновь созданном классе CGreenHouseDoc будет операция OnNewDocument. Мастером она не создается, но нам необходимо будет ее позднее переопределить. В данной операции будут инициализированы все переменные, которые относятся к документу.

Вы спросите, почему эти переменные инициализируются не в конструкторе класса, как это делается для большинства классов? В данном случае инициализация именно в этой операции вытекает из назначения класса. Ведь новый документ можно создать, когда класс уже инициализирован конструктором, и следовательно, необходимо где-то провести инициализацию переменных нового документа. И самое подходящее место для этого - операция OnNewDocument.

Так как мы будем отображать состояние устройств посредством класса CEditView, то наиболее удобным будет включить обработку события таймера именно в этот класс. При создании обработчика таймера при помощи инструмента Model Assistant она получит имя OnTimer. Как это сделать, рассмотрим позднее.

Таким образом, на основе стандартных классов документа,

предоставляемых MFC, мы получим приложение, в котором нам необходимо будет только добавить функциональность, а за отображение документа на экране будет отвечать библиотека.

Теперь, когда понятна структура приложения, необходимо создать его код. Советую создать новый проект VC-N-, для того чтобы пройти путь создания проекта еще раз и исключить последствия, которые могли остаться от предыдущих экспериментов.

Создание шаблона приложения

Для того чтобы ассоциировать проект Rational Rose с проектом Visual C++, если вы этого еще не сделали, необходимо выбрать окно Menu:Tools=>Visual C++=>Component Assignment Tool.

В нем мышкой перетаскиваем необходимые классы в строку VC++ на вопрос, хотите ли вы создать компонент VC++ (рис. 16.2), для того чтобы назначить выбранные классы в новый компонент, необходимо ответить «Да».

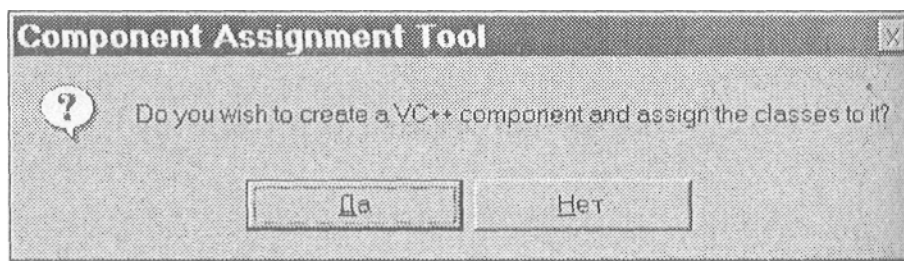


Рис. 16.2 Запрос на создание проекта

Все классы, для которых необходимо создание исходного кода, необходимо помещать в программные компоненты. Для простоты работы создадим только один компонент, который заключен в проект greenhouse.

В появившемся окне Select a VC+ project file нажать кнопку Add, после чего заполнить окно создания нового проекта так, как указано на рис. 16.3.

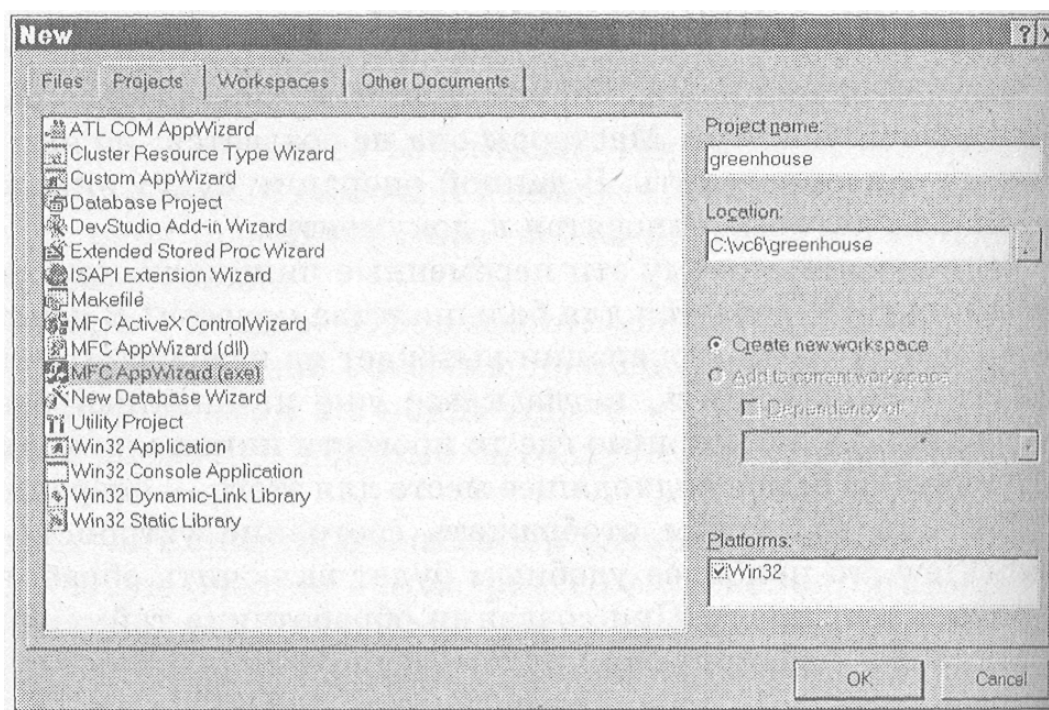


Рис. 16.3 Создание нового проекта VC++

Таким образом, мы воспользуемся мастером создания MFC приложения.

Создадим однооконное приложение обработки документа при помощи мастера, для чего установим кнопку Single document, как показано на рис. 16.4. Дальнейшие шаги мастера показывать нет необходимости, так как в них можно оставить все предлагаемые параметры без изменений. Пройдите все шаги мастера, последовательно нажимая кнопку Next.

Однако на шестом шаге, перед тем, как нажать кнопку Finish, измените Base class (базовый класс) на CEditView, как показано рис. 16.5.

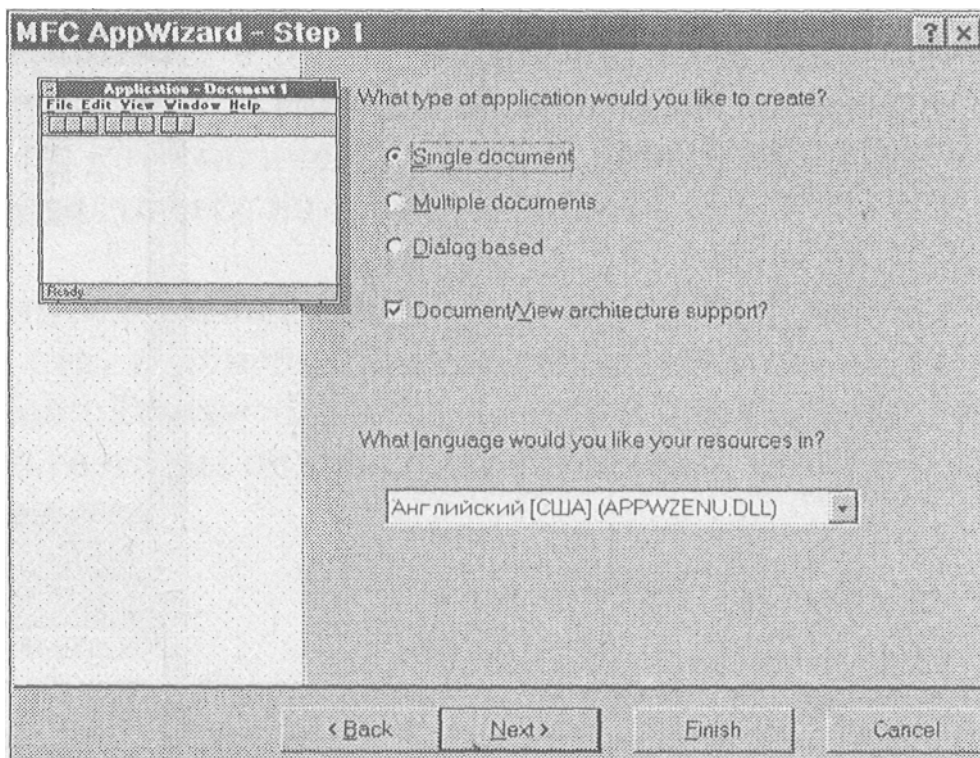


Рис. 16.4 Создание приложения для обработки документа

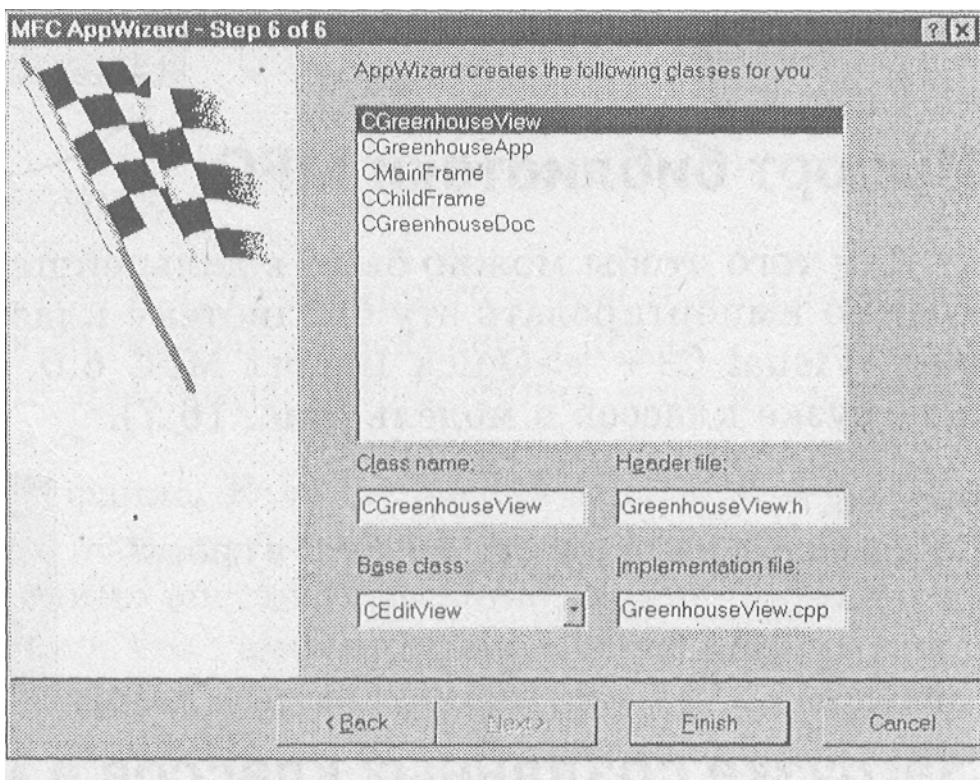


Рис. 16.5 Установка базового класса для шаблона приложения

После завершения всех процессов программа возвратится в окно выбора проекта (рис. 16.6), где необходимо выбрать только что созданный проект.

Назначение классов в проект

Нажмите ОК, и проект Visual C++ готов. Теперь можно назначить классы в проект, название которого появилось под надписью VC++ в окне Component Assignment Tool.

Для этого перетаскиваем мышкой необходимые классы в проект. Для того чтобы получить первый вариант работающего приложения, пока включите в проект только классы устройств. Постепенно мы будем расширять проект, но пока для быстрой отладки будем работать только с классами устройств.

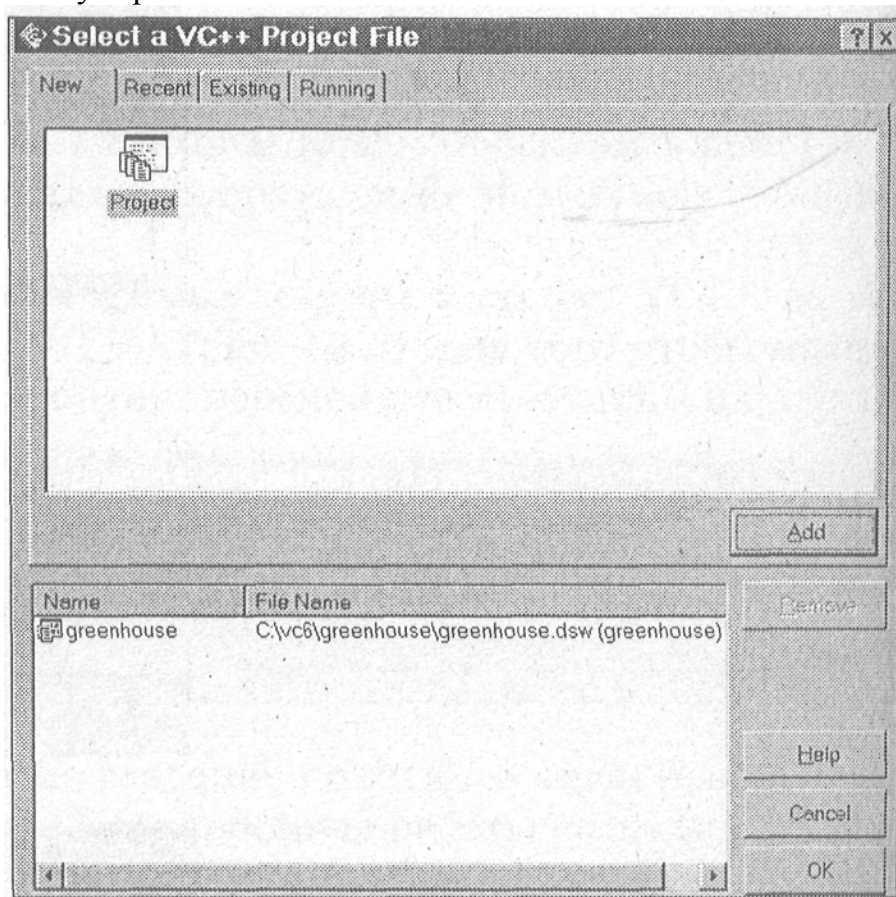


Рис. 16.6 Окно выбора проекта после создания проекта greenhouse

Импорт библиотеки MFC

Для того чтобы можно было в дальнейшем работать с классами MFC, необходимо импортировать эту библиотеку классов в модель. Выберите MenuTo-ols=>Visual C++ =>Quick Import MFC 6.0. После этого появится сообщение о загрузке классов в модель (рис. 16.7).

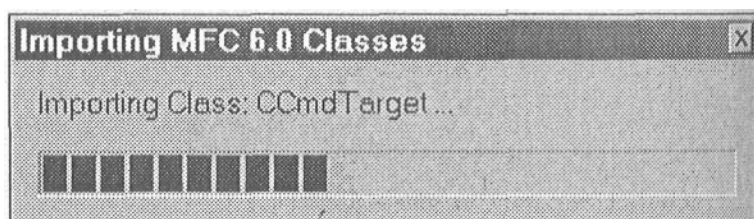


Рис. 16.7 Сообщение о загрузке классов MFC в проект

Загрузка созданных классов в модель

Теперь в модель Rational Rose можно загрузить классы, созданные в Visual C++. Это классы, о которые мы уже упоминали:

- главный класс приложения CGreenhouseApp;
- класс документа CGreenhouseDoc;
- класс просмотра CGreenhouseView;
- класс окна «о программе» CAboutDlg;
- класс основного окна программы CMainFrame.

Для того чтобы они появились в проекте, необходимо обновить проект по готовому коду при помощи следующей последовательности действий: Menu: Tools=>Visual C++=>Update Model From Code.

Сейчас это делать не обязательно, потому что в код необходимо внести некоторые изменения для нормальной компиляции, однако, для того чтобы по-

смотреть, как будут отображены классы в проекте Rational Rose, вы можете обновить модель из кода.

После обновления Rational Rose автоматически создаст диаграмму, которая отражает структуру классов, наследуемых из классов библиотеки MFC и аналогичную рис. 16.1.

Главное при этом - не удаляйте классы, которые Rational Rose не найдет в полученном коде. Их там еще нет, и для того чтобы они появились, необходимо обновить код по модели: Menu:Tools=>Visual C++=>Update Code. Вот это я советую проделать сразу, для того чтобы получить приложение, которое компилируется без ошибок.

Замечание

В дальнейшем в любой момент можно обновить модель по измененному коду или обновить код по изменениям в модели, для того чтобы синхронизировать проекты.

После того как обновление прошло, мы получим одинаковые классы в модели Rational Rose и в проекте VC++.

Первый запуск приложения

Если вы так же нетерпеливы, как и я, то вы просто сгораете от желания посмотреть, что же получилось после генерации исходного кода.

Запустите Visual Studio (если эта среда разработчика еще не запущена) и перейдите в проект greenhouse.

Нажмите F7 для создания EXE файла. Если вы рассчитывали, что компиляция пройдет без ошибок с первого раза, то я вас разочарую, В полученный при помощи Rational Rose код необходимо внести небольшие изменения для создания выполняемого файла. Радует то, что изменения, внесенные один раз, запоминаются, и при обновлении модели нет необходимости вносить их повторно.

При первой компиляции я получил 119 ошибок (у вас может получиться другое число).

Загляните в протокол. Самой распространенной ошибкой является

"fatal error C1010: unexpected end of file while looking for precompiled header directive". Это означает, что необходимо включить файл "stdafx.h" во все файлы, где возникла такая ошибка.

Откройте эти файлы и добавьте строку `#include "stdafx.h"` перед другими операторами `#include`. После завершения этой работы опять запустите генерацию.

Заметьте, что ошибок стало намного меньше, теперь можно заняться ошибками типа `syntax error : missing ; before identifier . . .`

Не волнуйтесь, это не Rational Rose создал код с ошибками - это просто не определены типы, которые мы использовали в модели, но для которых Rational Rose не создает кода.

К сожалению, генератор кода Visual C++ в Rational Rose не поддерживает ключевые слова `typedef` и `enum`, поэтому необходимо определить данный тип непосредственно в исходном коде, например, создать включаемый файл с определениями типов.

Создадим непосредственно в проекте VC++ файл `defs.h`. Project=>Add To Project=>New (рис. 16.8).

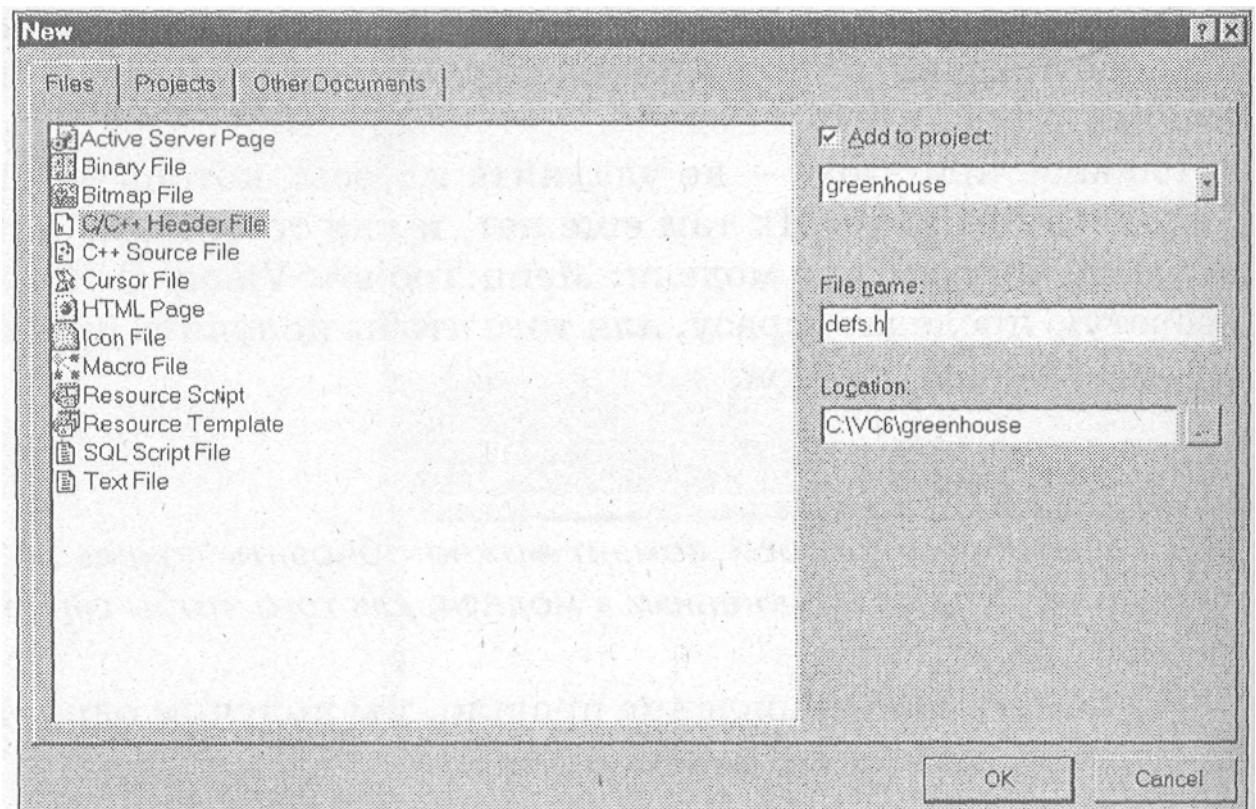


Рис. 16.8 Добавление нового заголовочного файла в проект VC++

В редакторе откроется окно для ввода, в которое введем следующие определения типов.

```
#pragma once // для того, чтобы файл включался один раз
// состояние устройства включено или выключено enum DeviceState
{Off,On};
//тип температуры лучше сделать простым типом, а не классом
typedef float Temperature;
// время дневное или ночное
```

```
enum Lights {day,night};
// тип для описания pH
typedef float pH;
// типы для описания текущего времени, часа и минуты
typedef unsigned int Day;
typedef unsigned int Hour;
typedef unsigned int Minute;
```

Конечно, можно обойтись и без переопределений. Ведь для компилятора слово `typedef` обозначает только синоним стандартных типов. Но для написания программ определение типов конкретной предметной области удобно для дальнейшего сопровождения программы, да и при разработке удобно иметь дело не с безличным `float`, а с конкретным `Temperature`.

После того как файл определений создан, его можно включить в модули, где используются эти определения. Так как мы создали определение и для типа температуры, то класс `Temperature` нам больше не нужен, и его можно удалить из модели.

Совет Файл определений удобно включать в файл `stdafx.h`.

Теперь в некоторых классах есть операции, которые возвращают значения. Пока для них созданы только шаблоны, поэтому пока текста нет, в такие операции поставим `return` с любым значением подходящего типа.

После завершения этих предварительных операций еще раз запустим компиляцию. Теперь она должна пройти без ошибок. Запускаем полученный файл (`CTRL+F5`) и видим изображение, показанное на рис. 16.9.

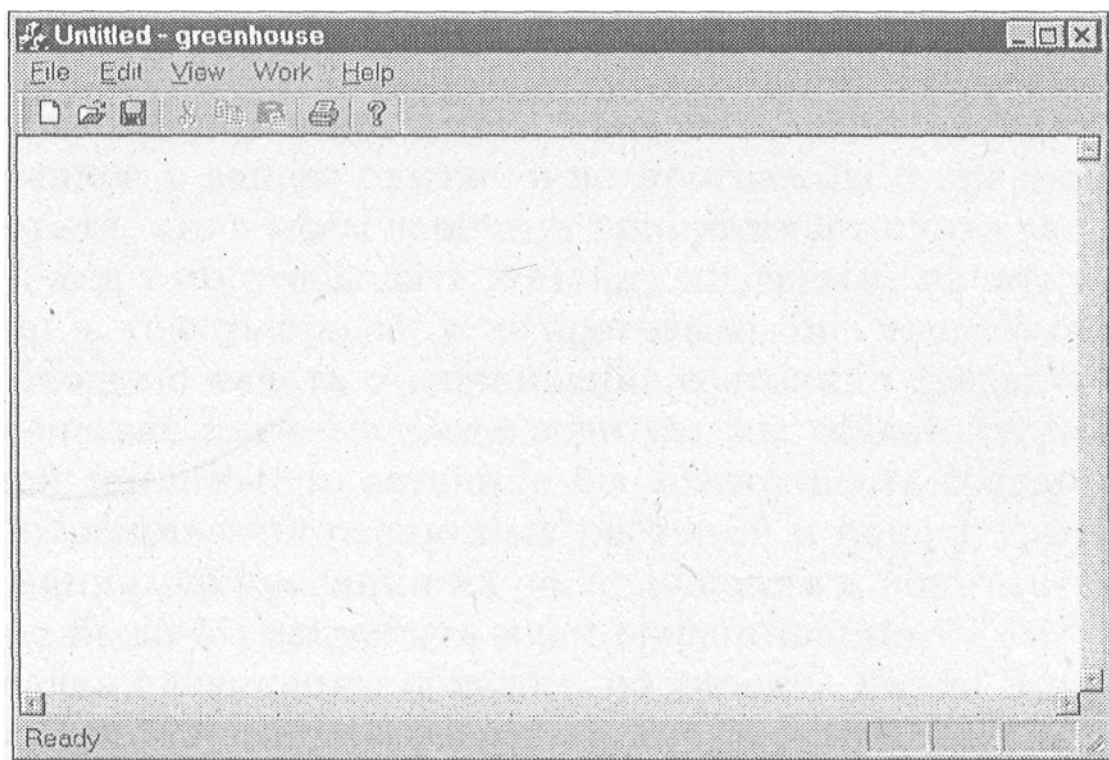


Рис. 16.9 Экран программы `GreenHouse.exe`

Что мы получили при первом запуске программы? Мы получили Windows приложение, которое пока не выполняет необходимых нам функций. Однако это уже полноценное приложение, включающее шаблоны разработанных нами классов, которые, осталось только довести до рабочего состояния.

Вопросы для повторения

1. Что такое тактические и стратегические решения, принимаемые при разработке системы?
2. В чем заключается соглашение об использовании имен компании Microsoft?
3. Какова структура MFC приложения?
4. Как создать структуру приложения при помощи мастера создания приложения Visual Studio?
5. Какие шаги нужно предпринять для обновления модели по исходному коду?
6. Как синхронизировать исходный код по созданной модели?

Глава 17. Добавление функциональности в класс просмотра

Создание пункта меню Work (работа)

Мастер приложения создал для нас полноценный текстовый редактор, на основе которого и будет осуществляться дальнейшая работа. Функции редактора нам не нужны, я их использовал для более простого вывода текста на экран. Но, что самое главное, в проекте уже есть заготовки классов для работы нашей теплицы. Их только необходимо наполнить содержанием.

Если вы сравните полученное в вашем случае окно программы с представленным на рисунке, то заметите, что в моем варианте присутствует пункт меню Work (работа). Я его создал, для того чтобы дать команду программе начать обрабатывать план выращивания, то есть при запуске программы она ждет от оператора команды, когда необходимо начать отслеживание состояния среды.

Такой тактический ход вполне логичен. Ведь если бы мы создавали программу для работы реальной теплицы, то оставили бы возможность создания и редактирования планов выращивания различных растений и конфигурирования устройств теплицы, например указания их расположения и количества. И только после этого можно было бы запустить план выращивания.

Этот пункт меню был создан следующим образом: из проекта Visual Studio я добавил новый пункт в ресурс Menu и присвоил ему ID=ID_WORK_START-PLAN, как показано на рис. 17.1.

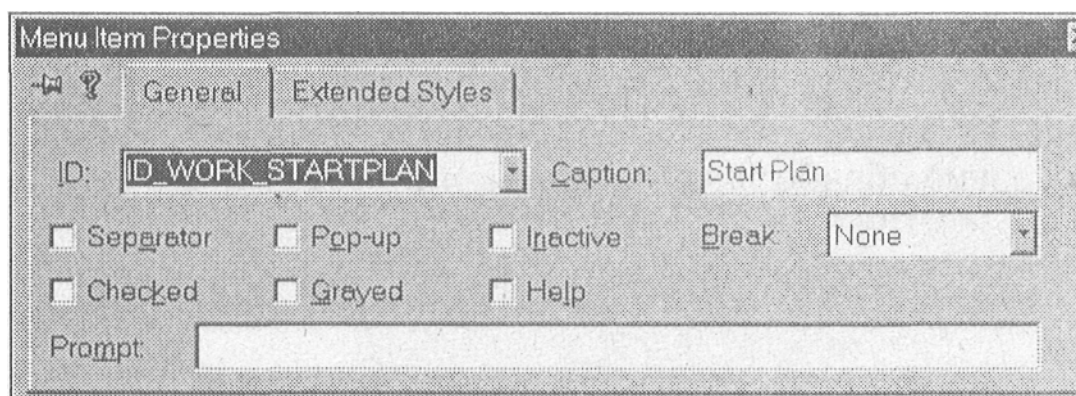


Рис. 17.1 Добавление пункта меню в ресурс IDR_MAIN_FRAME

Ассоциация операции с пунктом меню

Для того чтобы этот пункт меню начал работать, его необходимо ассоциировать с операцией класса, для чего есть два пути:

Создать операцию при помощи Visual Studio Class Wizard (рис. 17.2)

Если вы постоянно работаете с Visual C++, то этот пункт не нуждается в комментариях.

Создать обработку данного пункта в Rational Rose.

Второй вариант наиболее интересен для нас. Для того чтобы им воспользоваться, перейдем в Rational Rose и проведем обновление модели из кода (Menu:Tools=>Visual C++=>Update Model From Code).

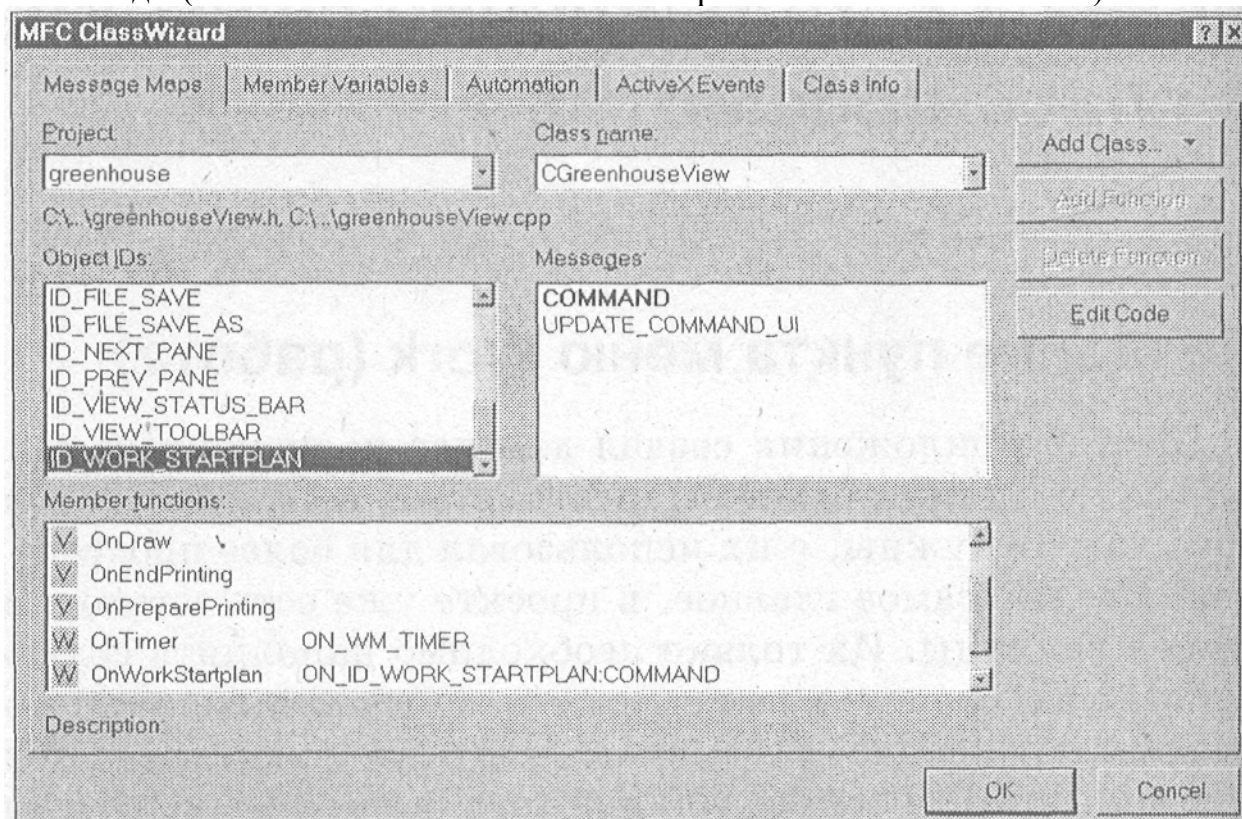


Рис. 17.2 Создание обработчика пункта меню в Class Wizard

Теперь можно создать его обработку в классе CGreenHouseView. Для этого необходимо выделить класс, например, в окне Browse или LogicalView и вызвать для него Model Assistant из контекстного меню.

Здесь как раз и проявляется глубина возможностей интеграции Rational

Rose и VC++. После переключения во вкладку MFC мы получим доступ ко всем атрибутам и операциям текущего класса и всех родительских классов, для чего нет необходимости изучать иерархию классов MFC.

Для того чтобы создать обработчик для команды ID_WORK_STARTPLAN, выделим пункт Command Handlers (обработчик команд) и из контекстного меню выберем пункт New Command Handler (новый обработчик), после чего активизируется окно, показанное на рис. 17.3.

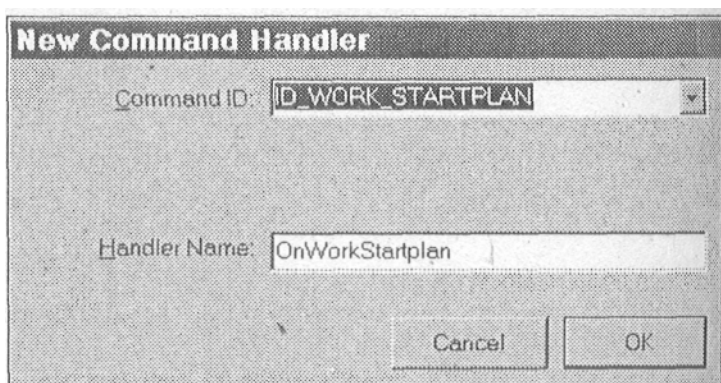


Рис. 17.3 Создание обработчика команд меню в Rational Rose

Теперь в строке Command Handlers образовался новый пункт OnWorkStart-plan, который включает в себя информацию о новом обработчике (рис. 17,4),

Таким же образом сюда можно добавлять любые новые обработчики меню, причем их можно создать сначала здесь, а потом уже внести в ресурсы меню или другие ресурсы.

Теперь можно добавить и другие операции, которые нам понадобятся для работы.

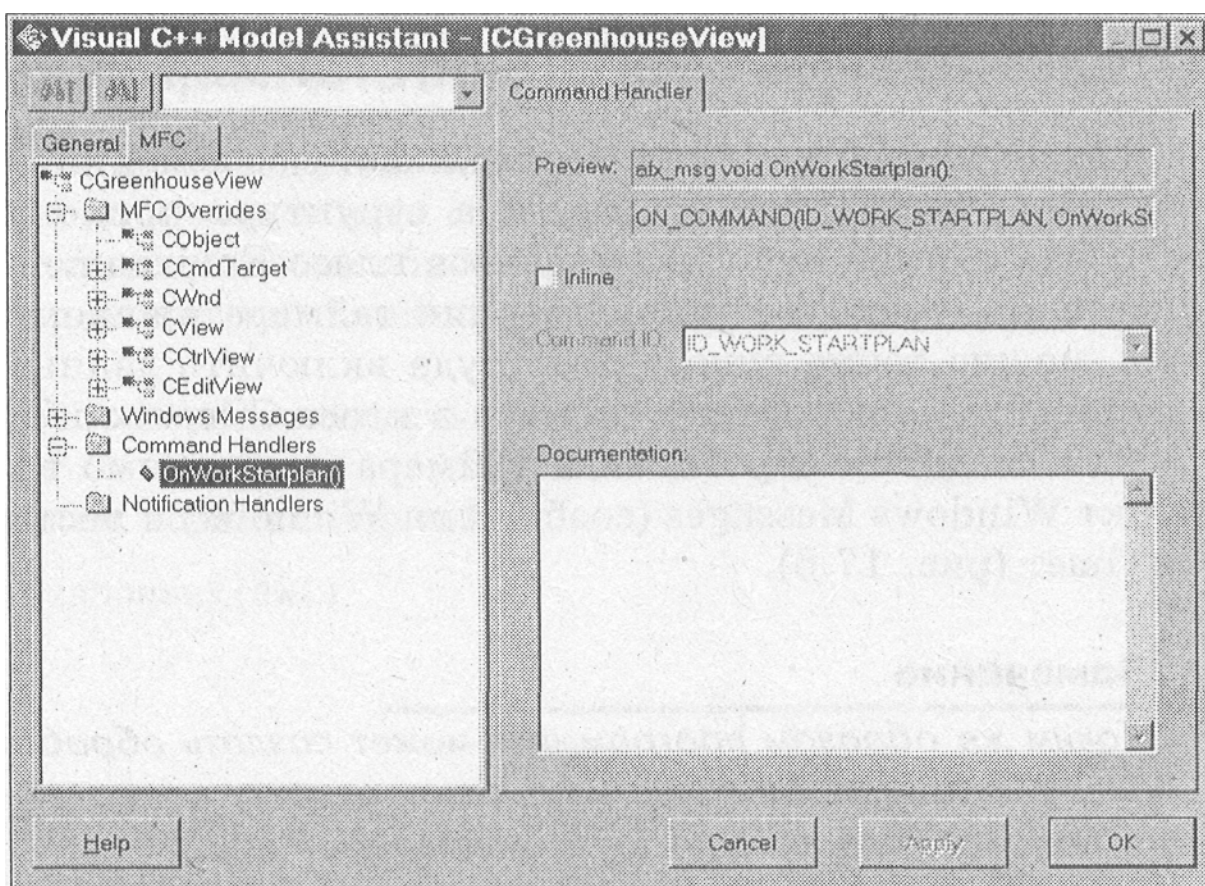


Рис. 17.4 Информация об. обработчике команд в Model Assistant

Добавление операций в Model Assistant

Для вывода текущего состояния устройств и показателей датчиков необходимо создание операции, которая бы выводила эту информацию в

главное окно программы. Назовем ее Message. Для того чтобы добавить операцию, необходимо на строке Operation (операции) из контекстного меню выбрать пункт New Operation (новая операция) и заполнить параметры операции так, как показано на рис. 17.5. Не забудьте, что при заполнении имен и типов переменных сначала необходимо указать имя, а затем, после двоеточия, ее тип. В любом случае всегда можно переименовать переменную, если что-то было заполнено неправильно при помощи RClick=>Rename.

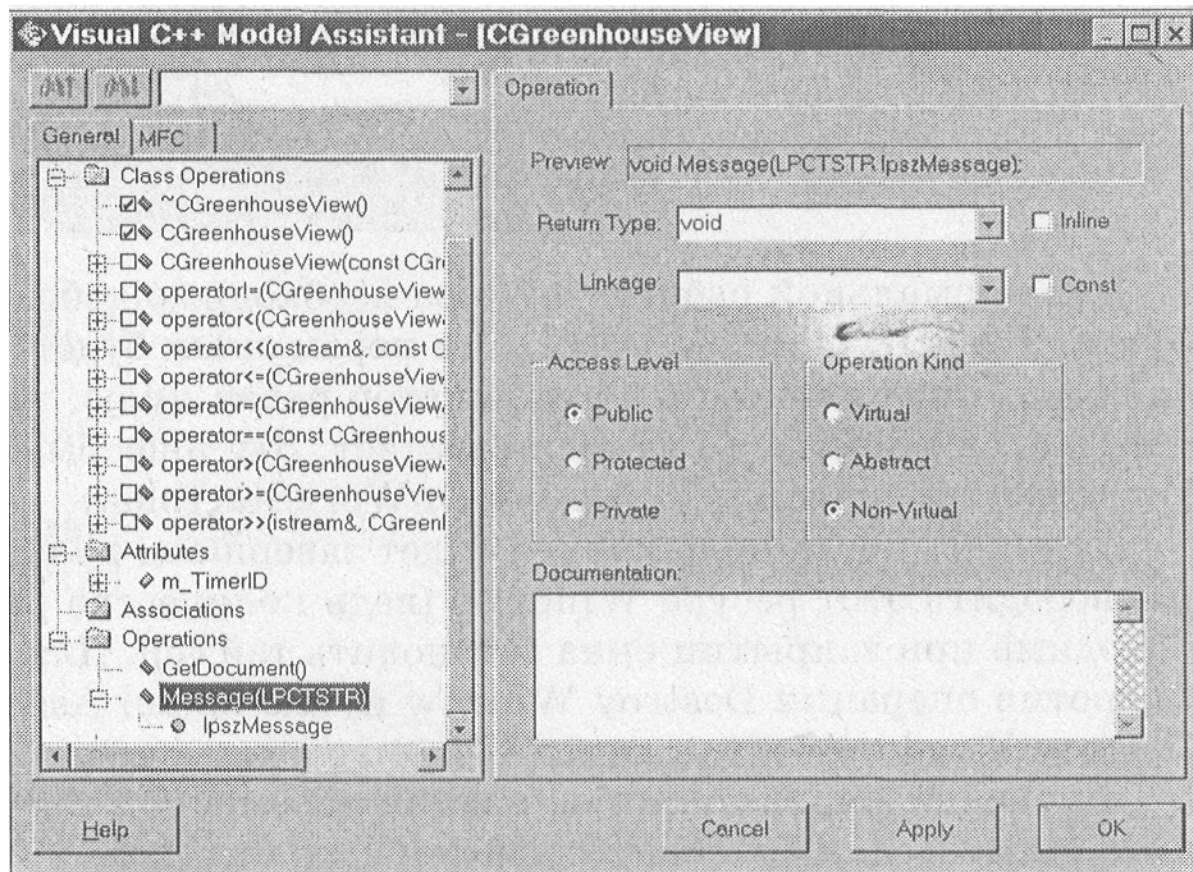


Рис. 17.5 Ввод операции Message

Отслеживание сообщений таймера

Также нам понадобится отслеживать сообщения таймера. Ранее, до того как стала окончательно известна структура приложения, мы планировали встроить таймер непосредственно в класс EnvironmentalControiler, хотя и не отметили возможности определения таймера в каком-то внешнем классе. Теперь можно точно определить, куда включить данный класс. Включим инициализацию и обработку таймера в класс CGreenhouseView.

Для создания обработчика таймера необходимо во вкладке MFC выбрать пункт Windows Messages (сообщения Windows) и поставить галочку на пункте OnTimer (рис. 17.6).

Замечание

Таким же образом программист может создать обработчики для любых сообщений Windows, достаточно установить отметку напротив

необходимых сообщений, и Rational Rose включит их в создаваемый код.

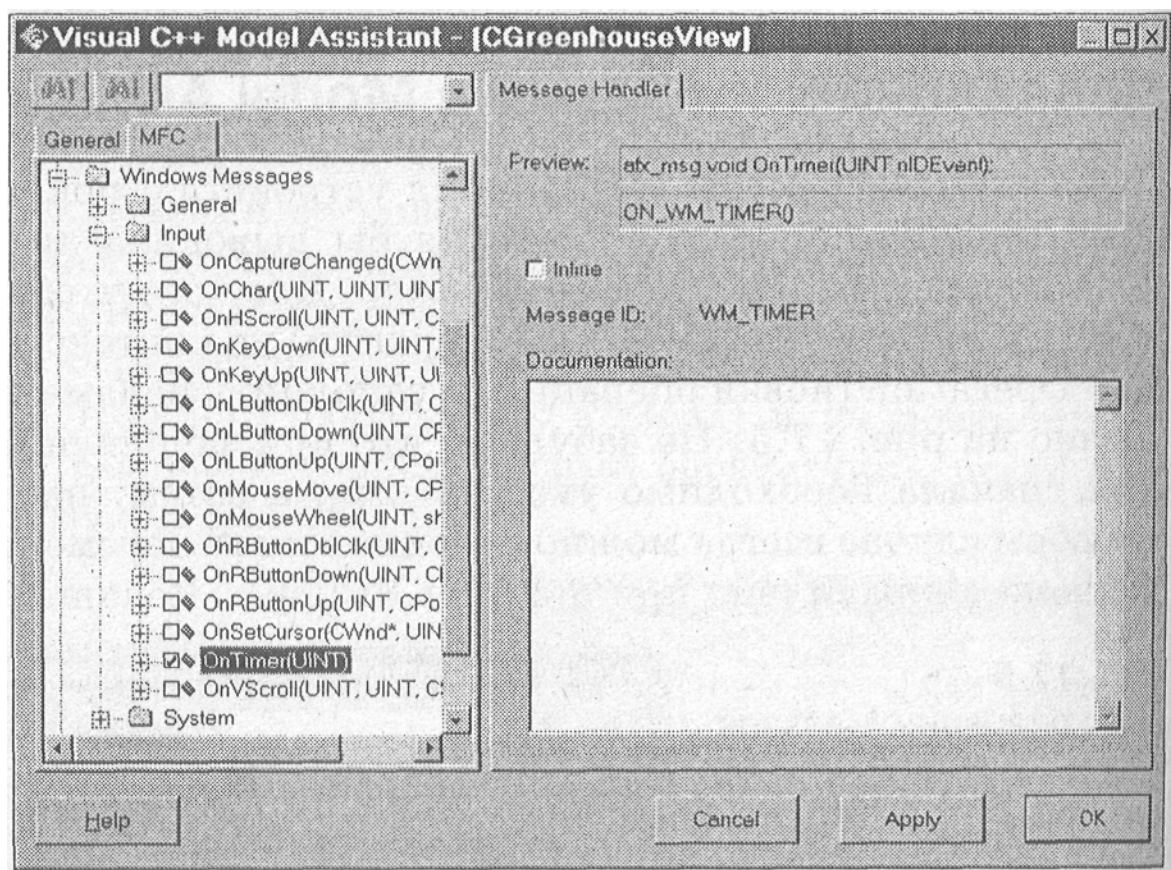


Рис. 17.6 Включение обработчика таймера

Для нормальной работы таймера необходимо добавить в класс новый атрибут `m_TimerID` с типом `UINT`. Эта переменная будет хранить идентификатор таймера. Если данный идентификатор равен нулю, то таймер не активизирован, а если не ноль, то таймер работает. Эту информацию мы используем, когда будем заполнять обработчик `OnWorkStartplan`.

В случае, когда приложение будет завершено до остановки таймера, чтобы , высвободить этот ресурс Windows (ведь количество ресурсов ограничено), необходимо при закрытии окна остановить таймер. Для этого поставьте отметку напротив операции `Destroy Window` путем `Model Assistant=>MFC=>CWnd=> Destroy Windows=ON`.

И конечно же, для инициализации переменной таймера необходимо установить отметку для генерации конструктора класса.

Редактирование шаблона класса `CGreenhouseView`

Таким образом, шаблон для создания класса `CGreenhouseView` полностью создан. Далее необходимо добавить в него содержание для непосредственной работы. Это можно будет сделать после обновления кода по модели.

Так как этот процесс проходит не мгновенно, то можно обновить только код класса `CGreenhouseView`.

После обновления кода можно заполнить полученный шаблон следующим образом.

В конструкторе обнуляем переменную таймера.

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

Операцию Message заполняем следующим образом.

```
void CGreenhouseView::Message(LPCTSTR IpszMessage)
{
```

```
    CString strTemp = IpszMessage; strTemp += _T("\r\n"); int len =
    GetWindowTextLength(); GetEditCtrl().SetSel(len,len); GetEditCtrl().Rep
    laceSel(strTemp); . }
```

Данная операция будет просто выводить в главное окно программы переданную ей строку, прибавляя символы перевода строки.

Включим инициализацию таймера в операции OnWorkStartplan. Заметьте, что остановить выполнения плана можно, повторно выбрав тот же пункт меню. В данном случае таймер установлен на обновление раз в секунду (второй параметр библиотечной функции SetTimer позволяет задавать интервал в миллисекундах).

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

Заодно проверим, как работает операция Message.

После проверки измените установку таймера на SetTimer(1,1,NULL), для того чтобы время в нашей теплице бежало быстрее, чем на самом деле. Это позволит проще отладить программу.

Изменим операцию DestroyWindow

```
BOOL CGreenhouseView::DestroyWindow()
{
    if (m_TimerID!=0){
        KillTimer(m_TimerID); m_TimerID =0;
    }
    return CEditView::DestroyWindow();
}
```

И для обработчика таймера включим сообщение о его работе

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

Внесите указанные изменения и создайте исполняемый файл. Если

возникли ошибки, исправьте их и запустите программу. Выберите пункт меню Work, при этом в окно программы начнут выводиться сообщения о том, что таймер работает. Остановите работу таймера, выбрав еще раз пункт Work, после этого экран программы должен выглядеть как на рис. 17.7.

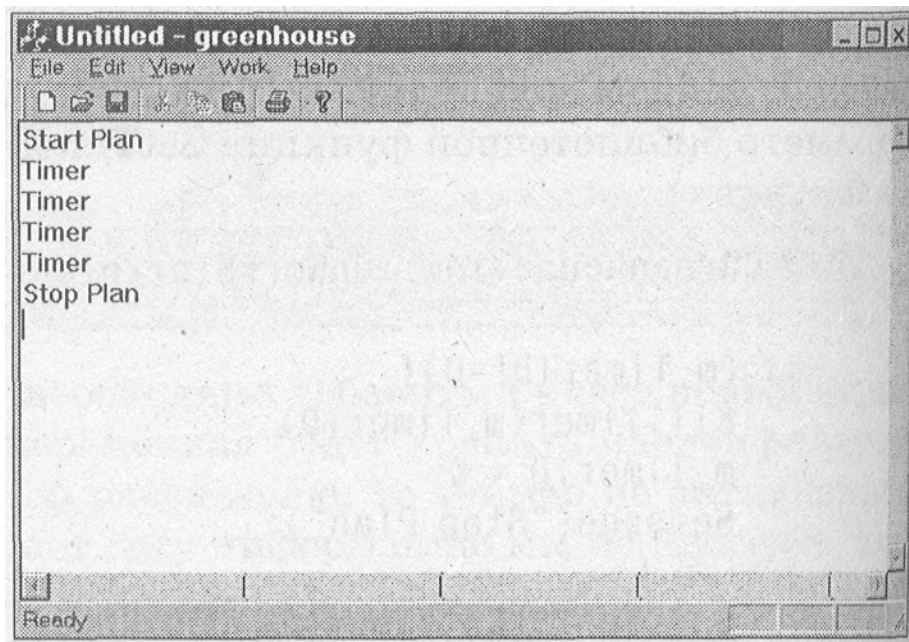


Рис. 17.7 Экран приложения после запуска и останова работы плана

Вопросы для повторения

1. Как добавить пункт меню в ресурс?
2. Как ассоциировать пункты меню с операциями классов?
3. Как создать обработчик события Windows посредством Model Assistant?

Глава 18. Добавление функциональности в класс документа

Создание структуры Condition

Следующим кандидатом для изменения будет класс CGreenhouseDoc. В нем будут храниться основные данные по устройствам и состоянию датчиков.

Для того чтобы этот класс мог воздействовать на среду, дополним структуру Condition дополнительными параметрами. Теперь эта структура должна будет моделировать «реальное» состояние среды, из которого датчики будут считывать показания. Также в этой структуре будет храниться «реальное» состояние исполнительных устройств (включено/выключено) и текущее время выращивания - день, час, минута.

Перейдите в диаграмму классов и создайте структуру Condition так, как показано на рис. 18.1.

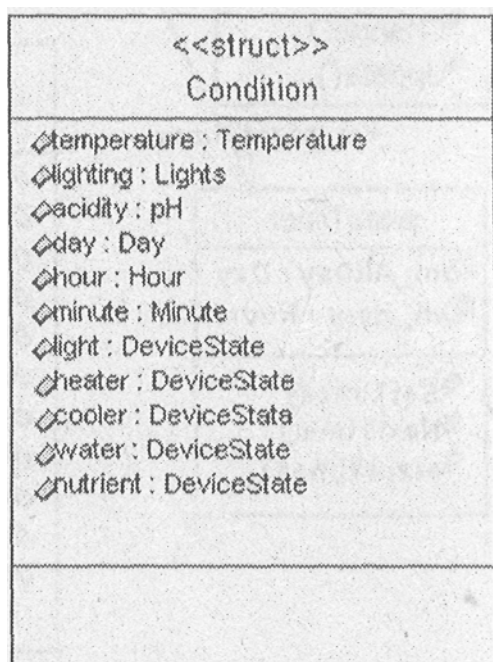


Рис. 18.1 Структура Condition

Создание модели среды

Следует заметить, что во время работы теплицы датчики должны считывать реальную информацию, которая изменяется в зависимости от изменения окружающей среды. В нашей программе нет датчиков, с которых можно считывать реальную информацию, поэтому необходимо создать класс, который будет вносить некоторое возмущение в параметры среды. Этот класс будет моделировать повышение или понижение температуры и изменение других параметров.

Создадим класс CEnvironment с операциями Change и Update.

Первая будет служить для внесения случайных возмущений в среду, а вторая будет служить для занесения текущего состояния устройств и показаний датчиков в структуру Condition после их изменения при помощи контроллера среды. Для того чтобы отражать изменения среды на экране, создадим класс CLog, который будет заниматься именно этим делом, и включим его в класс документа.

Создание связей класса документа

Также создадим класс таймера, который будет отслеживать текущее состояние времени выращивания.

Соединим при помощи связи Unidirectional Association классы, которые мы хотим поместить в класс документа, и получим рис. 18.2.

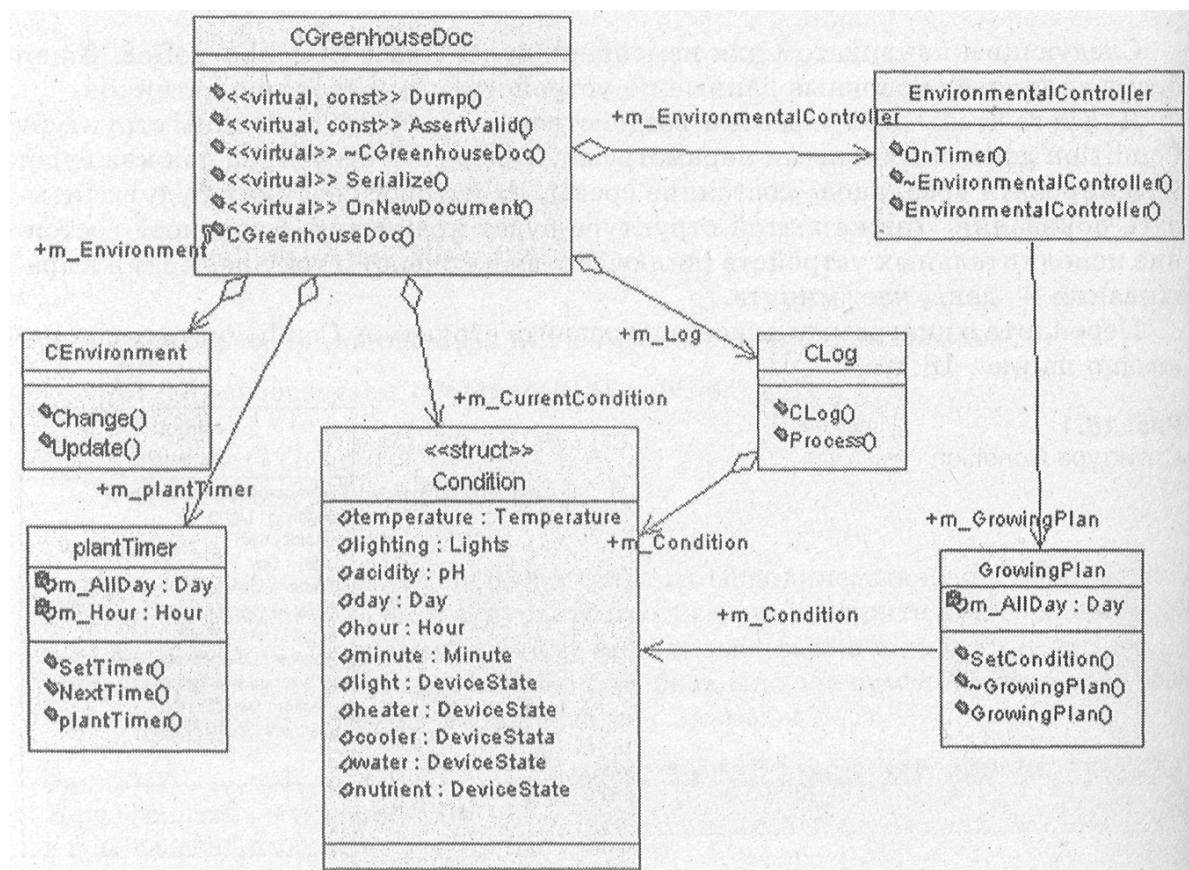


Рис. 18.2. Связи класса CGreenhouseDoc

Заметьте, что все классы, кроме GrowingPlan, включаются как агрегаты, а для структуры Condition, включенной в план выращивания, мы предусмотрели неагрегированную связь специально, для того чтобы можно было динамически создавать массив состояний на несколько указываемых при создании массива дней.

В классе CGreenhouseDoc все операции уже созданы при генерации исходного текста Мастером создания приложения VC++, и нет необходимости здесь их менять.

Добавление функциональности

Создадим исходный текст полученных классов и добавим в него необходимую функциональность. Для этого обновим код по модели и

перейдем в проект greenhouse среды Visual Studio.

Структура Condition должна выглядеть примерно так.

```
///##ModelId=3A1A18280104
struct Condition
public:
///##ModelId=3A81B6CA008C
Day day;
///##ModelId=3A81B6CA0020!
Hour hour;
///##ModelId=3A81B6C9035C
Minute minute;
///##ModelId=3A1A185C0370|
Temperature temperature; ///##ModelId=3A1A18710096 Lights Lighting;
///##ModelId=3A1A189103AC pH acidity;
//tfttModelId=3A81B6C9028A DeviceState light;
///##ModelId=3A81B6C?021E\
DeviceState heater;
//fttfModelId=3A81B6C901EO
DeviceState cooler;
///##ModelId=3A81B6C90174
DeviceState water;
///##ModelId=3A81B6C90140 \
DeviceState nutrient;
};
```

Добавим в программу инициализацию плана выращивания. Для этого преобразуем класс CGrowingPlan, как показано на следующем листинге. Сам класс выглядит после генерации таким образом.

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

Переменная m_AllDay показывает нам, на сколько дней рассчитан план выращивания. Добавим содержание в тело класса,и получим следующее.

```
///##ModelId=3A81B6D1006E GrowingPlan::GrowingPlan(Day day)
{
m_AllDay=day;
```

```

m_Condition=new Condition[day];
}
///

```

При создании класса в конструктор передается количество дней плана, для которого динамически создается необходимый массив структуры Condition.

Для заполнения структуры используем операцию SetCondition. Эта операция предназначена для заполнения значениями эталонной структуры плана выращивания. В структуре нас интересуют только эталонные состояния температуры и pH. Конечно, для того чтобы уменьшить расход памяти, можно было бы создать уменьшенный вариант только с двумя полями, но для простоты работы будем использовать ту же структуру Condition, которая используется во всей программе.

Необходимо помнить, что первым индексом в массиве будет 0. Это необходимо учитывать при проверке в классе на истечение времени выращивания. Класс таймера будет выглядеть следующим образом.

```

///

```

При установке таймера посредством операции SetTimer в нее передается количество дней выращивания. Мы приняли допущение, что время выращивания растений будет измеряться именно в сутках.

Также передадим час начала запуска плана, для того чтобы можно было бы закончить работу в то же время, но через определенное количество суток.

И тело класса преобразуем следующим образом

```

plantTimer::plantTimer()
{
m_AIIDay=0; m_Hour=0;

```

```

    ///ModeMd=3A201B8A03AC
    plantTimer::SetTimer(Day day, Hour hour)
    {
        m_AllDay=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;
        } // истекло время выращивания if (Condition->day>m_All Day &&
Condition->hour>-m_Hour)
        return FALSE; return TRUE;
    }

```

Заметьте, что созданный класс таймера рассчитан на работу с интервалом в одну минуту, а запускаем мы его и интервалом в 1/1000 секунды, следовательно, время в нашей теплице бежит примерно 60000 раз быстрее, чем на самом деле (на самом деле не настолько быстрее - все зависит от минимального интервала между генерацией событий аппаратным таймером конкретного компьютера).

Теперь можно установить количество дней в плане и обращаться к таймеру, когда необходимо указать, что наступило время обновления. Заметим, что для обновления таймера используется внешний вызов операции NextTime, так как данный класс не имеет собственного обработчика очереди сообщений. Конечно, можно создать полностью самостоятельный таймер, который будет взаимодействовать с системой, но в данном случае это только усложнит пример, не давая дополнительных возможностей для обучения.

Класс CLog позволяет вести протокол состояния среды и исполнительных устройств. В операцию Process передается указатель на структуру Condition и указатель на класс CGreenhouseView, для того чтобы вызвать операцию Message данного класса.

```

class Clog
{
private:
public:
    ///ModelId=3A81B6C50244
    void Process(Condition * condition,CGreenhouseView * pView);
    ///ModelId=3A81B6C501D6
    Condition m_Condition;
    ///ModelId=3A81B6C50212
    CLog(); };

```

Тело класса после преобразования выглядит следующим образом. В конструкторе устанавливается начальное состояние устройств и состояние среды.

```

CLog::CLog()

```



```

    {
        m_Condition.acidity=0;
        m_Condition.Lighting=night;
        m_Condition.temperature=0;
        m_Condition.heater=0ff;
        m_Condition.cooler=0ff; m_Condition.water=0ff; m_Condition.nutrient=0ff;
    }

```

При запуске процесса, перед тем как вывести протокол работы на экран, мы произведем проверку на наличие изменений в параметрах среды. Это необходимо, для того чтобы не загромождать экран информацией, которая уже выведена на предыдущем шаге. Единственное изменение, которое нам в этом случае нет необходимости выводить на экран - это изменение в показаниях прошедших суток, часов и минут.

```

    ///ModelId=3A81B6C50244
    void CLog::Process(Condition * condition,
        CGreenhouseView * pView)
    {
        if (m_Condition.acidity==condition->acidity &&
            m_Condition.Lighting==condition->Lighting && 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.Light==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,
            condition->hour
            condition->minute,
            (condition->lighting==day)?"day": "night",
            condition->temperature,
            condition->acidity,
            (condition->Light==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.Lighting=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.Light=condition->Light;
};

```

Теперь добавим функциональность к классу CEnvironment.

```

    ///ModelId=3A7BB6E801F4

```

```

class CEnvironment
{
public:

```

```

    ///ModelId=3A81B6D302BC Update(Condition * condition,
    EnvironmentalController * controller); ///ModelId=3A81B6D300C8
Change(Condition * condition,
    EnvironmentalController * controller); };
    Каждый час этот класс будет вносить возмущения в окружающую
    среду, изменяя случайным образом температуру и состояние pH, а также
    при включенных исполнительных устройствах будет вносить изменения в
    состояние окружающей среды.
    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()==On){ condition-
>acidity+=pH(0.2);
        }
        condition->acidity=round(condition->acidity,1);
        condition->temperature=round(condition->temperature,1);
        Update(condition,controller);
    }
    ///ModelId=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->lighting=
        (condition->hour>=9 && condition->hour<=20)?day:night;
        condition->light=controller->m_Light.get_CurrentState(); controller-
>m_TemperatureSensor.set_Value(
            condition->temperature); controller->m_pHSensor.set_Value(condition-
>acidity);
    }

```

Теперь, когда готово все, кроме исполнительных устройств, можно посмотреть, как будет работать наше тепличное хозяйство. Для этого добавим в класс CGreenhouseDoc следующий код для инициализации начального состояния структуры Condition и задания плана выращивания.

```

    ///##ModelId=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.lighting=day;                m_CurrentCondition.acidity=7;
m_CurrentCondition.light=0ff;                m_CurrentCondition.heater=0ff;
m_CurrentCondition.cooler=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;
    }

```

И добавим обработку сообщения 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_plantTimer.NextTime(
        &GetDocument()->m_CurrentCondition)){
            if (m_TimerID!=0){
                KillTimer(m_TimerID); m_TimerID = 0;
            }
            GetDocument()->m_Log.Process(
            &GetDocument()->m_CurrentCondition,this); Message("End plan");
        }
        CEditView: : OnTimer(nIDEvent); }

```

После запуска полученного приложения у вас должно получиться следующее (рис. 18.3). Все устройства выключены, поэтому температура и pH изменяются случайным образом. В нашей теплице теперь не хватает только исполнительных устройств, которые будут компенсировать изменения температуры и pH.

Обозначения следующие:

Day - сколько дней прошло с запуска плана выращивания, показывает текущее время. В моем случае посадки были произведены в

девять утра, и урожай должны будем снимать также в девять утра, но уже через указанное количество дней;

Показываемся текущее время суток day/night;

T - текущая температура в теплице;

pH - текущий показатель кислотности;

L - отражает состояние осветителей;

H - отражает состояние нагревателей;

C - отражает состояние вентиляторов;

W - отражает состояние крана для подачи воды. ON - вода подается;

N - отражает состояние заслонки для подачи удобрений. ON - удобрения подаются.

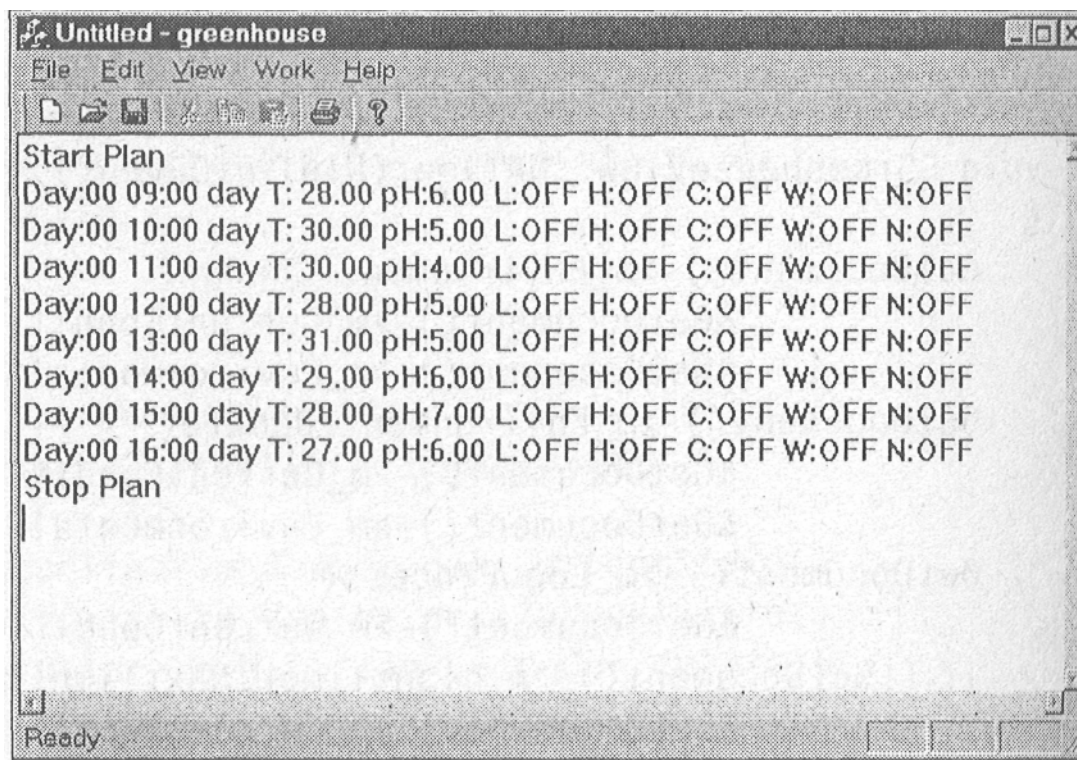


Рис. 18.3 Работа приложения без исполнительных устройств

Вопросы для повторения

1. Как создать структуру на диаграмме классов?
2. Как создать связи между классами?
3. Что необходимо добавить в шаблоны классов для получения работоспособного приложения?

Глава 19. Создание исполнительных устройств

Вступительные замечания

Завершением разработки системы будет создание исполнительных устройств. Эти устройства не отличаются сложным поведением, они могут находиться в двух состояниях: включено или выключено. На примере исполнительных устройств мы рассмотрим создание иерархии классов приложения.

До сих пор никакой иерархии не было создано, что не совсем правильно. Мы рассмотрим, как преобразовать созданные классы в классы с определенной иерархией.

Список исполнительных устройств

К исполнительным устройствам нашей теплицы можно отнести:

Cooler (вентилятор);

Heater (нагреватель);

Light (лампочка);

NutrientTank (хранилище удобрений);

WaterTank (хранилище воды).

Их всех объединяет то, что они производят некоторые действия, исполняют команды. При этом все они, кроме лампочки, воздействуют на параметры среды, такие как температура и pH.

В результате предыдущих действий на диаграмме классов получилось примерно следующее состояние классов этих устройств (рис. 19.1).

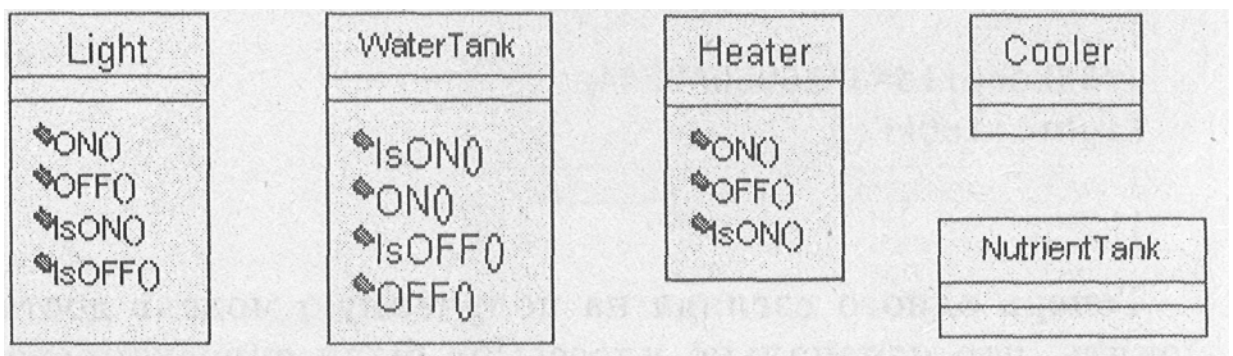


Рис. 19.1 Состояние классов исполнительных устройств

Если в вашей модели исполнительные устройства не присутствуют на диаграмме классов, то добавьте их при помощи Menu:Query=>Add Classes.

Если посмотреть на код, полученный для устройства, например, Light (лампочка), то мы увидим, что Rational Rose создал заголовок класса и предусмотрел шаблоны для занесения туда обработчиков операций. Получен следующий заголовочный файл.

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

```

#endif
#ifndef _INC_LIGHT_3A17E8EF021C_INCLUDED
#define _INC_LIGHT_3A17E8EF021C_INCLUDED
///##ModelId=3A17E8EF021C
class Light
{
public:
///##ModelId=3A2605C30384
ON();
///##ModelId=3A2605CC02E4
OFF();
///##ModelId=3A2605D3014A
IsON();
///##ModelId=3A2605DA0244
IsOFF();
};
#endif /* INC_LIGHT_3A17E8EF021C_INCLUDED */ И файл тела

```

класса:

```

#include "Light.h" ///##ModelId=3A2605C30384 Light::ON()
{
}
///##ModelId=3A2605CC02E4
Light::OFF()
{
}
///##ModelId=3A2605D3014A
Light::IsON()
{
}
///##ModelId=3A2605DA0244
Light::IsOFF()
{
}

```

Теперь одного взгляда на полученную модель достаточно, для того чтобы понять, что изначально классы не были спроектированы правильно (вот где необходим объектно-ориентированный анализ). Необходим главный класс, из которого классы этих устройств будут унаследованы, тогда методы класса ON(), IsON(), IsOFF(), OFF() будут принадлежать главному классу, а для остальных классов необходимо переопределение методов ON(), OFF() для обращения непосредственно к необходимому устройству. В Rational Rose легко исправить такую ошибку.

Создание родительского класса устройств

Создадим класс CPlantDevice со спецификациями, представленными на рис. 19.2.

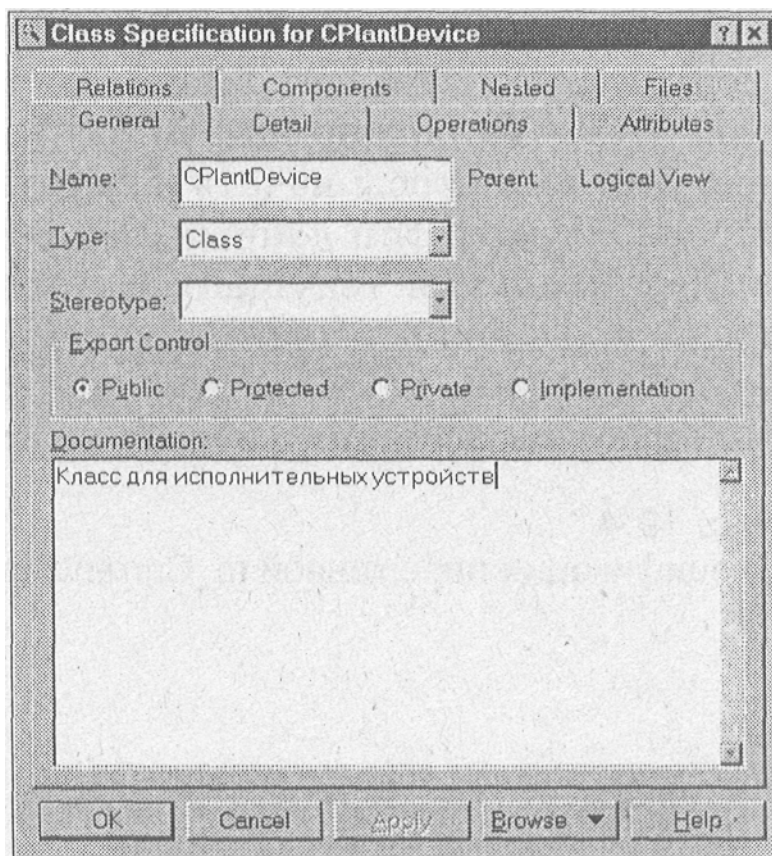


Рис. 19.2 Спецификации класса CPlantDevice

Теперь необходимо создать операции, используемые во всех устройствах. Для этого перейдем во вкладку Operations и посредством контекстного меню добавим операции таким образом, как показано на рис. 19.3. Все операции возвращают значения типа BOOL.

Если IsON() возвращает TRUE, это означает, что устройство включено.

Если IsOFFQ возвращает TRUE, это означает, что устройство выключено.

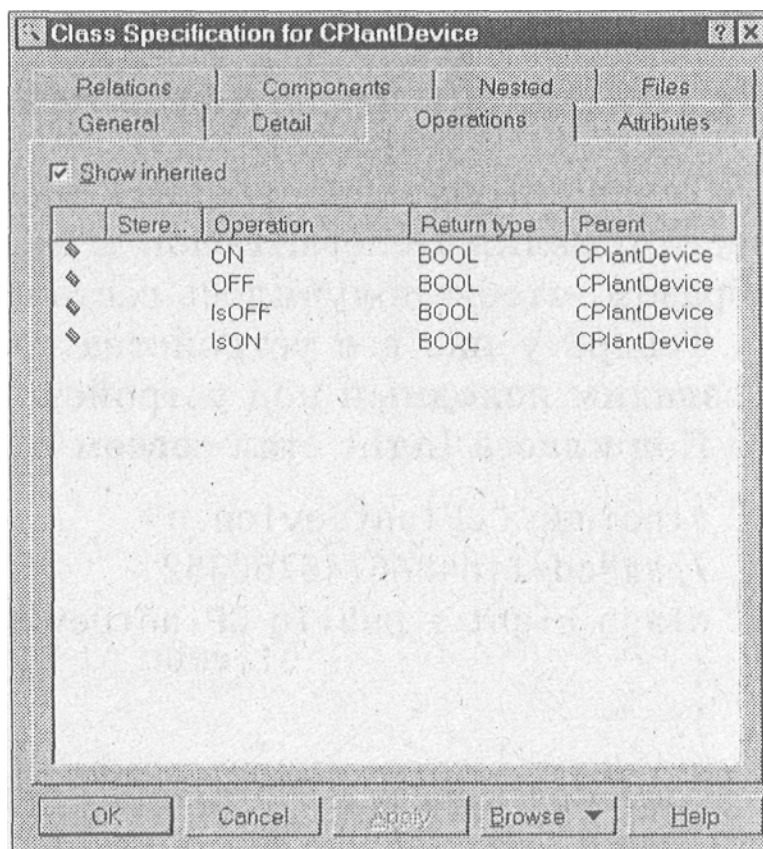


Рис. 19.3 Операции класса CPlantDevice

Если `ON()` возвращает `TRUE`, это означает, что устройство включилось нормально.

Если `OFF()` возвращает `TRUE`, это означает, что устройство выключилось нормально.

Неплохой мыслью было бы добавить еще одну операцию `get_CurrentState` (получить текущее состояние), которая должна возвращать состояние устройства: `On` - включено или `Off` - выключено. Мы добавим ее несколько позднее. Для того чтобы начать работать с состояниями устройства, у нас уже есть перечисляемый тип данных `DeviceState`.

Для хранения текущего состояния создадим переменную `m_CurrentState` с пока еще неопределенным типом `DeviceState`. Для этого перейдите во вкладку `Attributes` и добавьте при помощи контекстного меню новый атрибут. Заполните спецификации атрибута, как показано на рис. 19.4.

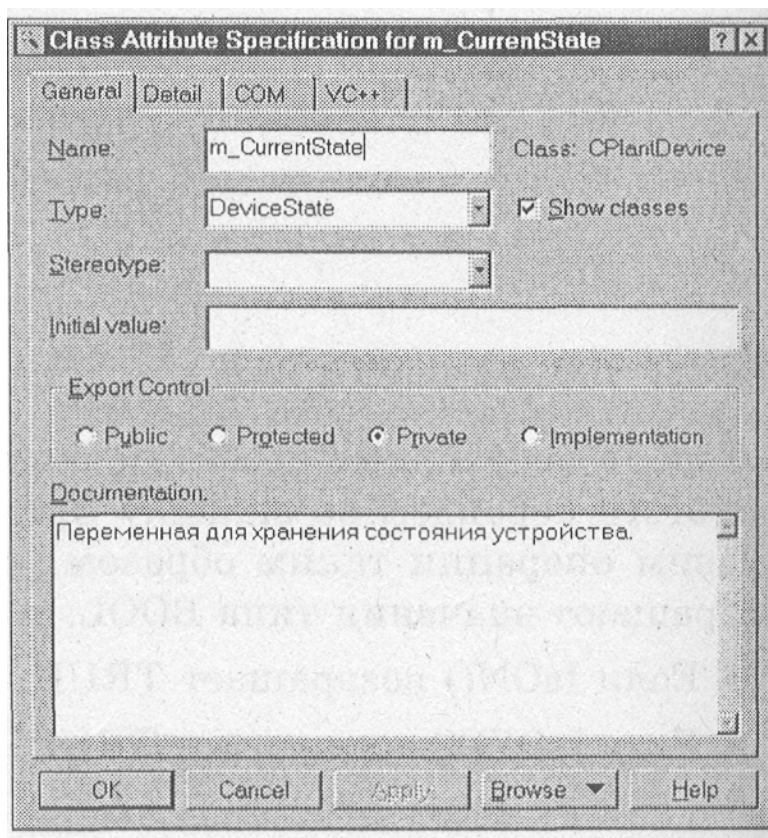


Рис. 19.4 Спецификации переменной m_CurrentState

Установка наследования

Теперь необходимо удалить операции из исполнительных устройств и провести стрелки Generalization к вновь созданному классу CPlantDevice таким образом, чтобы получилось состояние, показанное на рис. 19.5.

Теперь у нас все устройства наследуются из одного класса CPlantDevice. Создадим исходный код устройств и проверим, что получилось.

Код класса Light стал совсем простым:

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

Зато код класса CPlantDevice получил все необходимое для работы.

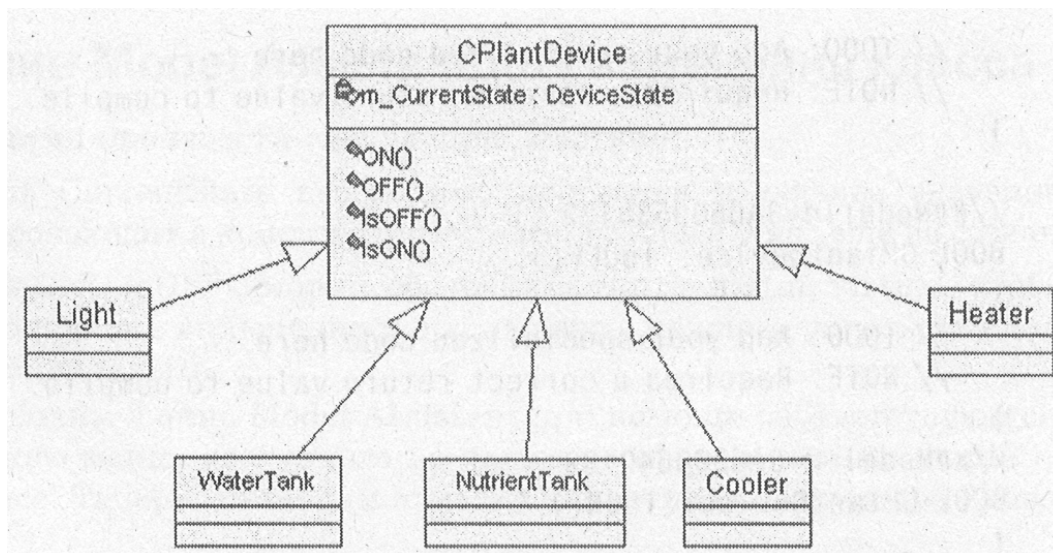


Рис. 19.5 Наследование классов устройств

CPlantDevice.h: class CPlantDevice

```

{
public:
//Включение устройства
///ModelId=3A65601C0208
BOOL ON();
//Выключение устройства
///ModelId=3A656044015E
BOOL OFF();
//Проверка выключено ли устройство
///ModelId=3A6560530122
BOOL IsOFF();
//Проверка включено ли устройство
///ModelId=3A6560640136
BOOL IsON(); private:
//Переменная для хранения состояния устройства.
///ModelId=3A65607C0212
DeviceState m_CurrentState;
};

```

CPlantDevice.cpp:

```

#include "CPlantDevice.h" ///ModelId=3A65601C0208    BOOL
CPlantDevice::ON()
{
// TODO: Add your specialized code here.
// NOTE: Requires a correct return value to compile.
}
///ModelId=3A656044015E
BOOL CPlantDevice::OFF()
{
I
//TODO: Add your specialized code here.
// NOTE: Requires a correct return value to compile.
}

```

```

}
///ModelId=3A6560530122 BOOL CPlantDevice::IsOFF()
{
// TODO: Add your specialized code here.
// NOTE: Requires a correct return value to compile.
} ///ModelId=3A0560640136 BOOL CPlantDevice::IsON()
{
// TODO: Add your specialized code here.
// NOTE: Requires a correct return value to compile.
}

```

Добавление значения для возврата

Здесь вы можете заметить, что компиляция полученного кода приведет к ошибкам. Хотя Rational Rose и позволяет с легкостью тасовать классы и их связи, но все-таки необходимо произвести наполнение методов классов содержанием, для того чтобы получить полноценное приложение. Для этого необходимо внести некоторую правку в код.

Добавим в методы класса `IsON()` и `IsOFF()` значение возврата непосредственно в коде класса из оболочки Visual Studio.

```

BOOL CPlantDevice::IsOFF()
{
return m_CurrentState-^Off;
}
///ModelId=3A6560640136 BOOL CPlantDevice::IsON()
{
return m_CurrentState==On;
}
>

```

Включим заголовочный файл `stdafx.h` перед включением заголовочного файла класса.

```

#include "stdafx.h"
#include "CPlantDevice.h"
Замечание

```

Если вы получили при генерации сообщение: «fatal error C1010: unexpected endof file while looking for precompiled header directive», это означает, что вы просто забыли включить файл `stdafx.h` в ваш файл с расширением `.cpp`.

Использование Model Assistant для изменения класса

Теперь необходимо произвести следующие действия:

переменной `m_CurrentState` необходимо присвоить начальное значение. Обычно это происходит в конструкторе класса, который у нас еще не создан.

для операций `ON()` и `OFF()` можно обозначить тип функций `virtual`, чтобы показать, что процесс включения и выключения каждого устройства должен быть специальным.

Для этого переходим в окно Model Assistant при помощи

соответствующего пункта контекстного меню, доступного по нажатию правой кнопки мыши на классе CPlantDevice. Теперь установим отметку напротив конструктора класса (рис. 19.6).

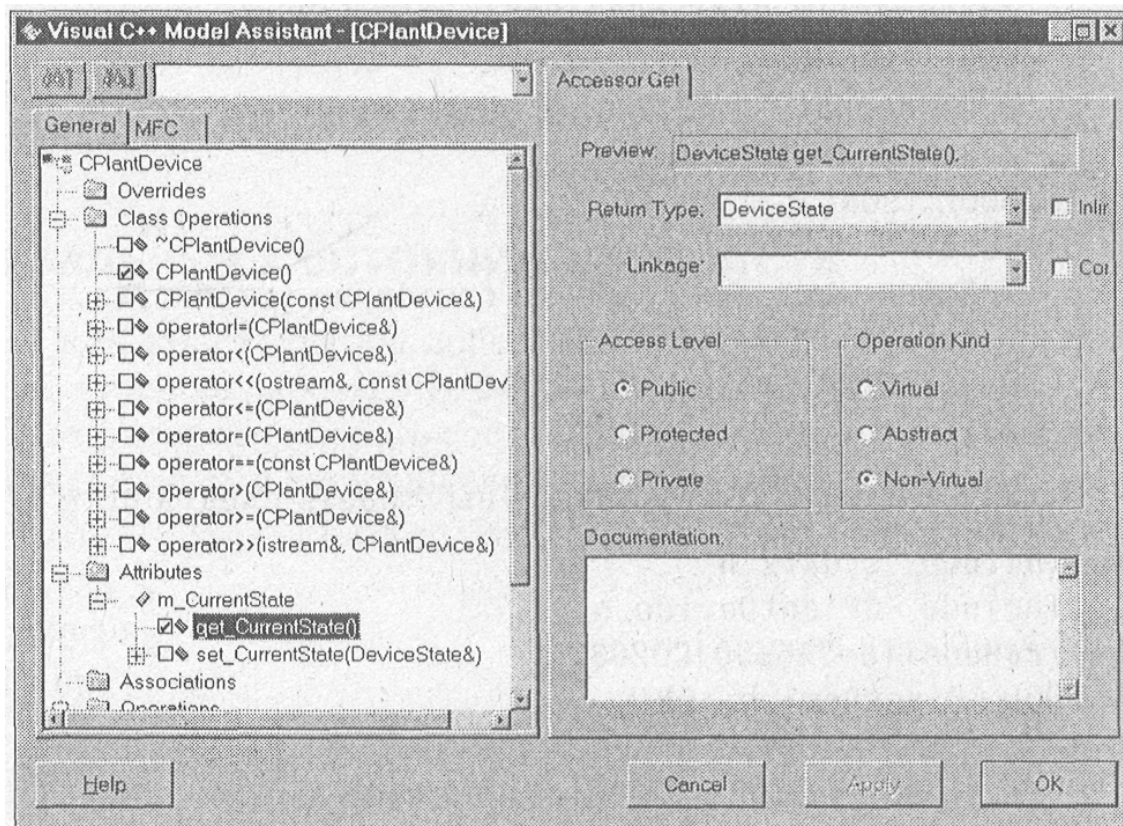


Рис. 19.6 Установка дополнительных операций класса

В пункте Attributes (атрибуты) можно увидеть, что Rational Rose уже позаботился об операциях, которые будут устанавливать значение переменной `m_CurrentState` и получать это значение. Пока нам необходима операция `get_CurrentState`. Установим отметку напротив этой операции и заполним необходимые установки так, как показано на рисунке.

Необходимо заметить, что по умолчанию Rational Rose создает тип возвращаемого значения (Return Type) как ссылку. В нашем случае в ссылке нет необходимости.

Не забудьте установить для операций `ON()` и `OFF()` поле Operation Kind (Тип операции) в `Virtual`, и можно закрывать это окно.

После генерации мы получим следующий код класса.

CPlantDevice. h:

```
class CPlantDevice
{
```

```
public:
```

```
    ///ModelId=3A73FAC302DO CPlantDeviceO;
```

```
    ///ModelId=3A73FAC3037A DeviceState get_CurrentState();
```

```
    //Включение устройства ///ModelId=3A65601C0208 virtual BOOL ON();
```

```
    //Выключение устройства ///ModelId=3A656044015E virtual BOOL
```

```

OFF(); //Проверка  выключено ли устройство ///ModelId=3A6560530122
BOOL IsOFF();
    //Проверка  включено ли устройство ///ModelId=3A6560640136
BOOL IsON(); private:
    //Переменная  для  хранения  состояния  устройства.
///ModelId=3A65607C0212 DeviceState m_CurrentState;
};
Файл CPlantDevice.cpp будет выглядеть следующим образом:
#include      "stdafx.h"      #include      "CPlantDevice.h"
///ModelId=3A65601C0208 BOOL CPlantDevice::ON()
{
    // TODO: Add your specialized code here.
    // NOTE: Requires a correct return value to compile.
    return TRUE;
}
///ModelId=3A656044015E BOOL CPlantDevice::OFF()
{
    // TODO: Add your specialized code here.
    // NOTE: Requires a correct return value to compile.
    return TRUE;
}
///ModelId=3A6560530122 BOOL CPlantDevice: : IsOFF()
{
    return m_CurrentState==Off;
}
///ModelId=3A6560640136
BOOL CPlantDevice::IsON()
{
    return m_CurrentState==On;
}
///ModelId=3A73FAC3'02DO CPlantDevice::CPlantDevice()
{
    // ToDo: Add your specialized code here and/or call the base class }
///ModelId=3A73FAC3037A
DeviceState CPlantDevice::get_CurrentState()
{
    return m_CurrentState;
}

```

Добавление начальных состояний устройств

Теперь в исходный код конструктора добавляем начальное состояние устройств. Все устройства в начале работы у нас будут выключены. А в операции ON() и OFF() добавляем присвоение нового состояния устройства в переменную m_CurrentState. Полученный код будет выглядеть следующим образом:

```

#include      "stdafx.h"      #include      "CPlantDevice.h"
///ModelId=3A65601C0208 BOOL CPlantDevice::ON()
{

```

```

m_CurrentState=On; return TRUE;
}
///

```

Создание операций ON/OFF для дочерних классов

Теперь для классов устройств, наследуемых из класса CPlantDevice, можно создать операции ON() и OFF(), которые должны быть переопределены. Для этого, опять же, необходимо активизировать окно Model Assistant на необходимом классе. Возьмем для примера класс Light (рис. 19.7).

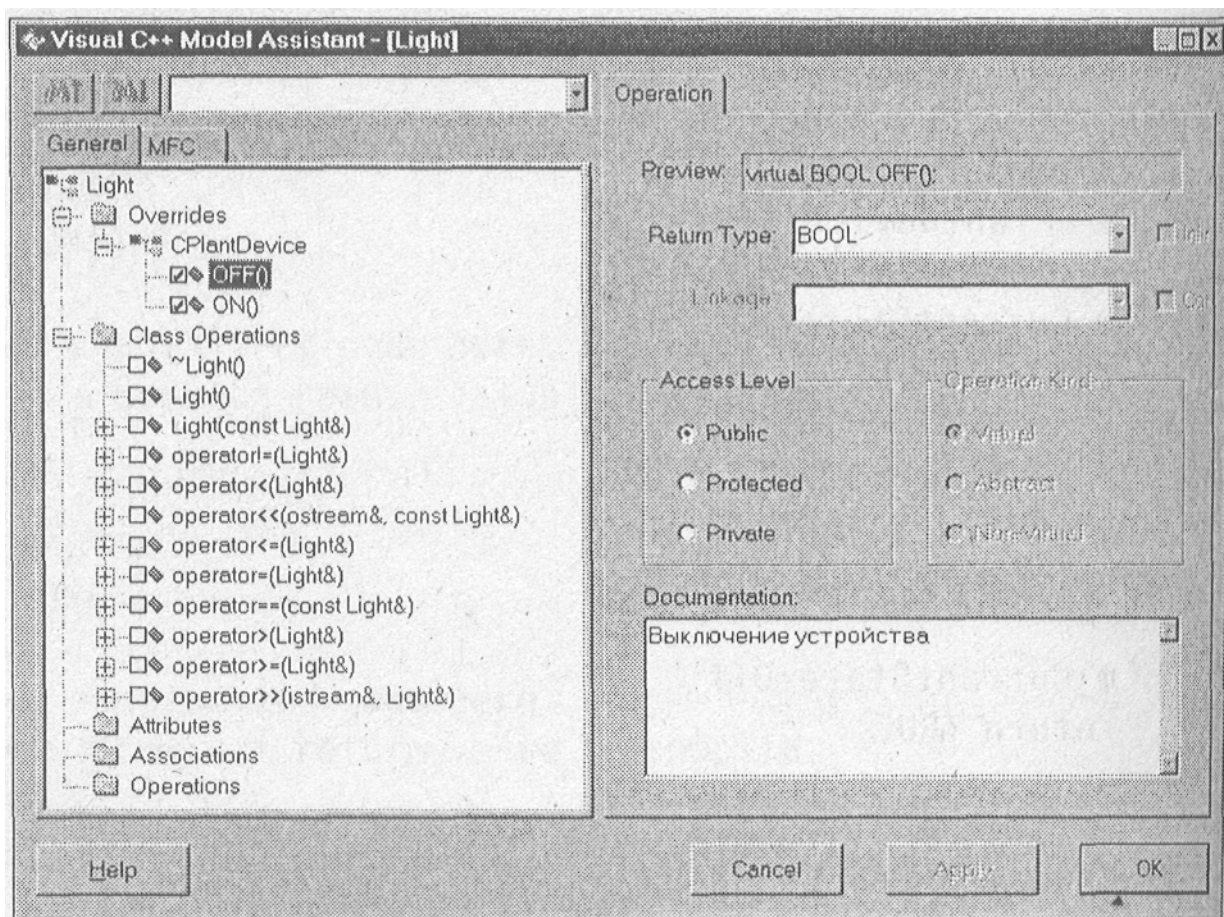


Рис. 19.7 Установки для переопределяемых операций класса Light

Здесь можно увидеть, что появилась папка Overrides (переопределение), в которой можно выбрать те операции, для которых необходимо переопределение. Поставим отметки на ON() и OFF() и обновим код, для того чтобы посмотреть, что получилось.

Файл Light.h:

```
#include "CPlantDevice.h"
///##ModelId=3A6746750352
class Light : public CPlantDevice
{
public:
//Включение устройства ///##ModelId=3A7401CA0280 virtual BOOL
ON();
//Выключение устройства ///##ModelId=3A7401CA0398 virtual BOOL
OFF();
}
```

Light.cpp

```
#include "stdafx.h" #include "Light.h"
///##ModelId=3A7401CA0280 BOOL Light::ON()
{
    return CPlantDevice::ON();
}
///##ModelId=3A7401CA0398 BOOL Light::OFF()
{
```

```
return CPlantDevice::OFF();
}
```

Заметьте, что в операциях, переопределенных из головного класса, в исходный код были включены даже комментарии, которые были введены в головном классе.

Совет

Вставляйте комментарии везде, где это возможно.

После этих действий диаграмма классов должна будет принять примерно следующий вид (рис. 19.8).

При создании настоящей, а не учебной программы, для каждого устройства в операциях ON(), OFF() необходимо описать взаимодействие с физическим устройством посредством последовательного или другого порта, но в нашем случае устройства виртуальны и не существуют на самом деле, поэтому эти операции не заполняются.

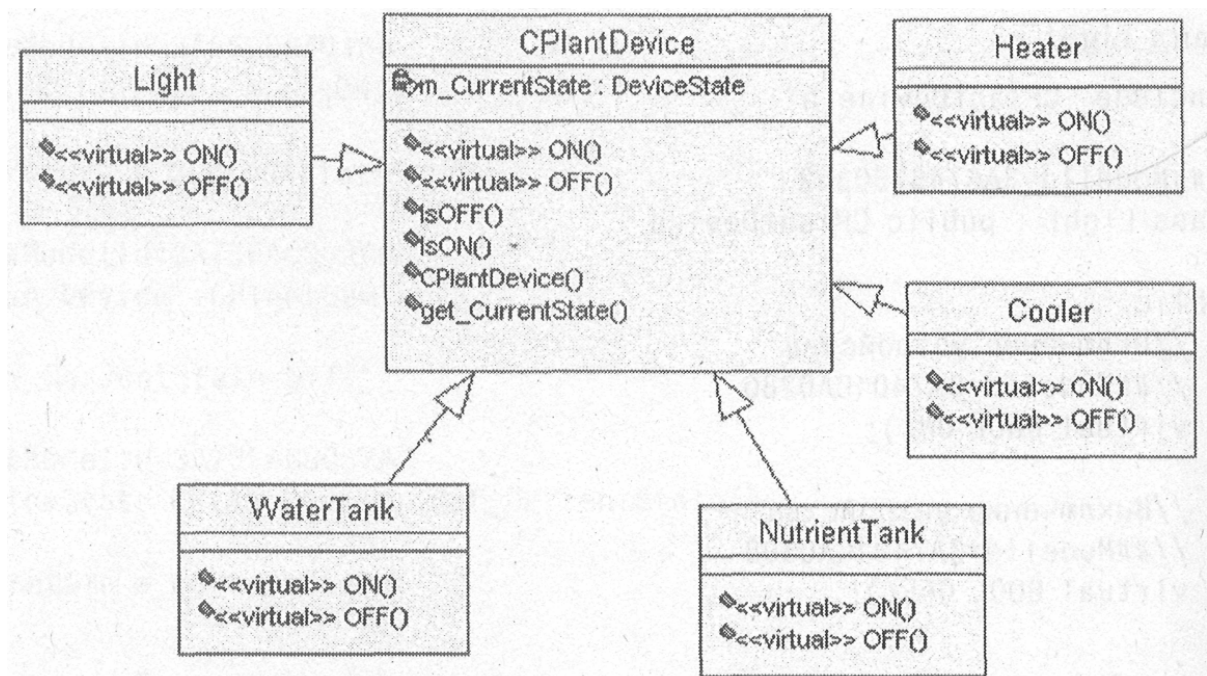


Рис. 19.8. Иерархия классов исполнительных устройств

Создание адреса расположения устройств

Вспомним, что если принимать каждое устройство как физическое, то оно должно характеризоваться своим местом расположения. Для задания месторасположения у нас уже создан класс Location, который используется для задания, расположения датчика температуры.

Первая мысль, которая приходит в голову - это включить в класс CPlant-Device переменную, хранящую точку расположения устройства. В данном случае это будет плохим решением.

Совет

Не бросайтесь реализовывать первое пришедшее в голову решение. Оно обычно бывает не оптимально.

Лучшим решением будет создание класса CDevices, из которого

будут наследоваться классы исполнительных устройств и датчиков. Этот класс и будет отвечать за обработку расположения устройств. При создании программ без применения Rational Rose вы, наверняка, отказались бы изменять иерархию уже готовых классов, вспомнив о достаточной трудоемкости такой операции, но с Rational Rose - это сущие пустяки.

Создадим новый класс CDevices. В этом классе создадим новый атрибут m_LastID с типом int и укажем, что атрибут должен быть Static.

Для чего нужен этот атрибут? Допустим, что каждое устройство в системе имеет уникальный номер, который его идентифицирует. Создадим атрибут m_ID с типом int, который будет хранить идентификатор. Для того чтобы при создании устройства каждый раз не задавать этот номер с большой вероятностью ошибиться, создадим статический элемент, который будет хранить последний ID. На рис. 19.9 показаны полученные атрибуты. Для того чтобы установить тип Static, воспользуйтесь окно Model Assistant.

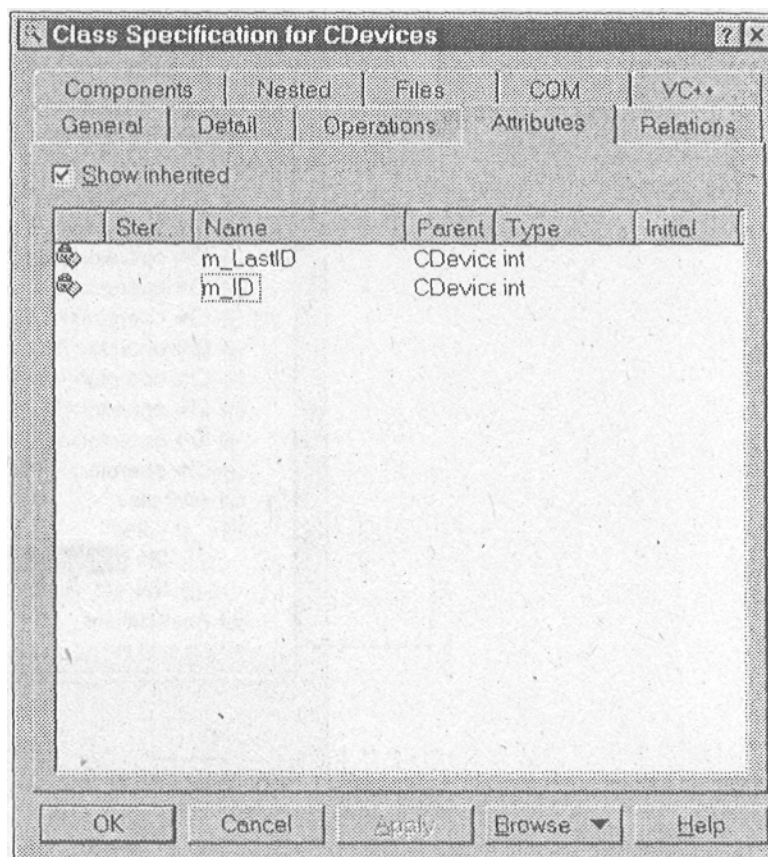


Рис. 19.9 Атрибуты класса CDevices

Для указания местоположение мы уже создали класс Location. Необходимо перенести связь Unidirectional Association с класса TemperatureSensor во вновь созданный класс CDevices. Впоследствии мы будем наследовать класс сенсоров из класса CDevices и возможность указания местоположение будет и у класса датчиков.

Совет

Не забудьте после переноса связи из одного класса в другой на

диаграмме классов удалить лишние связи в окне RClick==>OpenSpecification=>Relofion.

Обычно для работы с переменными класса создаются операции. Воспользуемся для этого окно ModelAssistant и создадим для класса Location операции get_value и set_value, как показано на рис. 19.10.

Теперь остается указать наследование ранее созданного класса CPlant-Device из класса CDevices при помощи стрелки Generalization. У вас должно получиться так, как на рис. 19.11.

Что мы должны получить теперь? В нашей гидропонной системе имеется некоторое количество исполнительных устройств, каждое из которых имеет свой идентификатор, уникальный в системе, и свое местоположение, которое можно устанавливать и считывать. Каждое устройство имеет два состояния включено/выключено. Можно проверить состояние устройства, включить его или выключить.

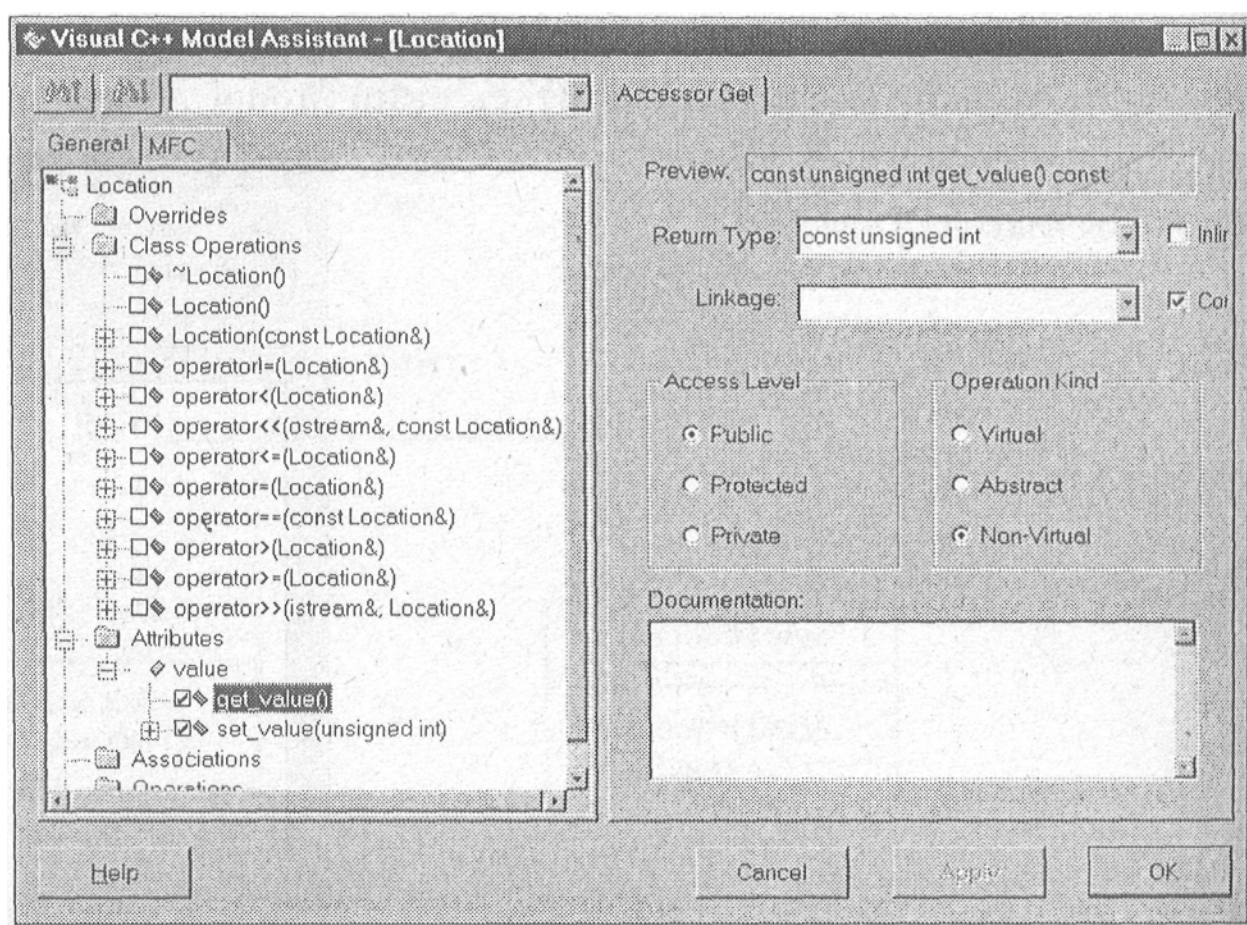


Рис. 19.10 Операции класса Location

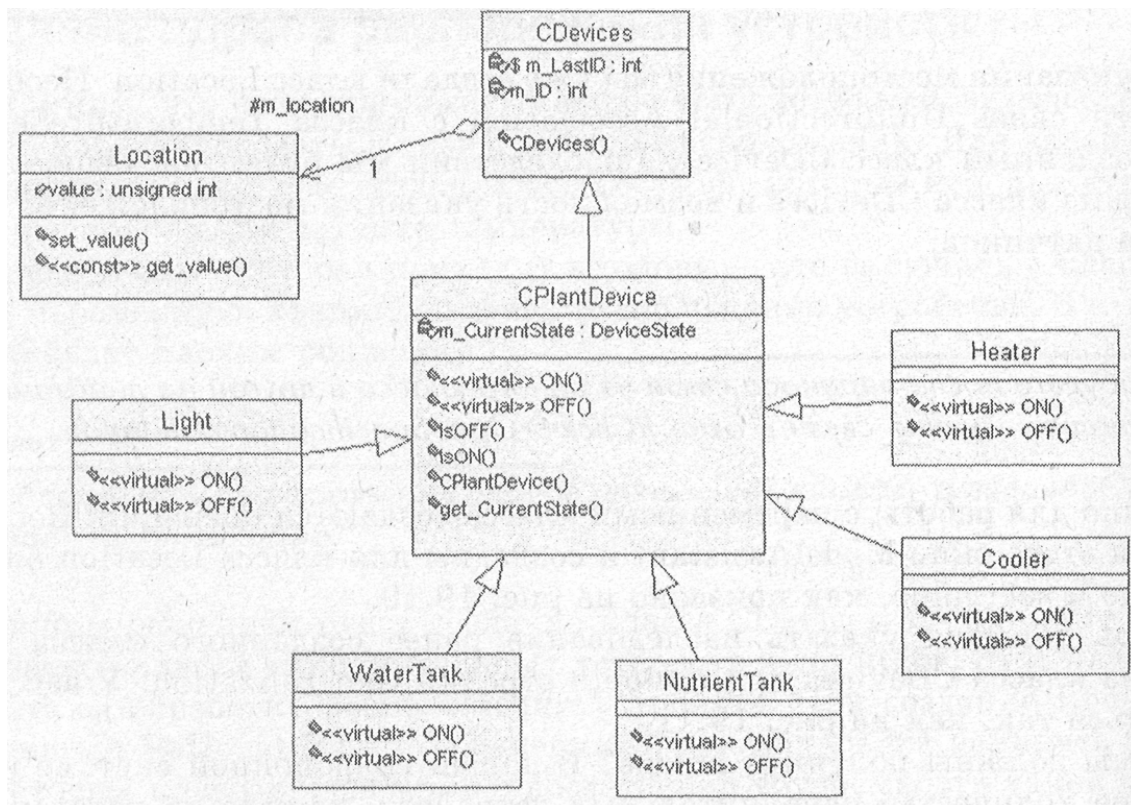


Рис. 19.11. Иерархия классов устройств после добавления CDevices

Посмотрим на исходный код головного класса CDevices, который получится после генерации: RClick=>Update Code. Однако не забудьте назначить новые классы в проект VC++ при помощи Tools=>Visual C++=> Component Assignment Tool. В противном случае можете получить примерно такое сообщение при генерации кода: «Warning: Class: CDevices; Skipped, supplier not assigned to VC++ component».

Заголовочный файл класса CDevices выглядит так:

```

class CDevices
{ protected:
  ///ModelId=3A581C8B01A4
  Location m_location; public:
  ///ModelId=3A752F8F012C
  CDevices(); private:
  ///ModelId=3A77C8EE0046
  int m_ID;
  ///ModelId=3A752F3F0064
  static int m_LastID; };
  
```

Агрегирование устройств

Условимся, что все устройства тепличного хозяйства входят как агрегаты в класс CEnvironmentalControiler (рис. 19.12).

Таким образом, код класса CEnvironmentalController будет следующим.

```

class EnvironmentalController
{
  
```

```

public:
//CGreenhouseView * pView;
public:
///##ModelId=3A81B6C602A8
Heater m_Heater;
///##ModelId=3A81B6C6023C
Cooler m_Cooler;
///##ModelId=3A81B6C601FE
Light m_Light;
///##ModelId=3A81B6C601CC
pHSensor m_pHSensor;
///##ModelId=3A81B6C60190
TemperatureSensor m_TemperatureSensor;
///##ModelId=3A81B6C60124
NutrientTank m_NutrientTank;
///##ModelId=3A81B6C600F2
WaterTank m_WaterTank;
//tfftModelId=3A81B6C600B4
GrowingPlan * m_GrowingPlan;
public:
///##ModelId=3A81B6C701F4 EnvironmentalControler();
///##ModelId=3A81B6C70154 ~Environmental Controller();
///##ModelId=3A201BD4032A OnTimer(Condition * condition);
};
А код тела класса должен быть 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_GrowingPlan->m_Condition->acidity){ if (m_WaterTank. IsON())
m_WaterTank.OFF(); if (m_NutrientTank. IsOFF())
m_NutrientTank.ON();
}
if (m_pHSensor.get_Value()>
m_GrowingPlan->m_Condition->acidity){ if (m_WaterTank. IsOFF(0)

```

```

        m_WaterTank.ON();          if          (m_NutrientTank.          IsON())
m_NutrientTank.OFF();
    }
    if (m_pHSensor.currentpH(=
m_GrowingPlan->m_Condition->acidity){ if (m_WaterTank.IsON())
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();
    }
    ///ModelId=3A81B6G701F4
    Environmental Controller::EnvironmentalController()
    {
        m_GrowingPlan          =          new          GrowingPlan(10);
m_TemperatureSensor.calibrate(); m_pHSensor.calibrate(); }

    ///ModelId=3A81B6C70154
    EnvironmentalController::~~Environmental.Controller()
    {
delete m_GrowingPlan;
    }

```

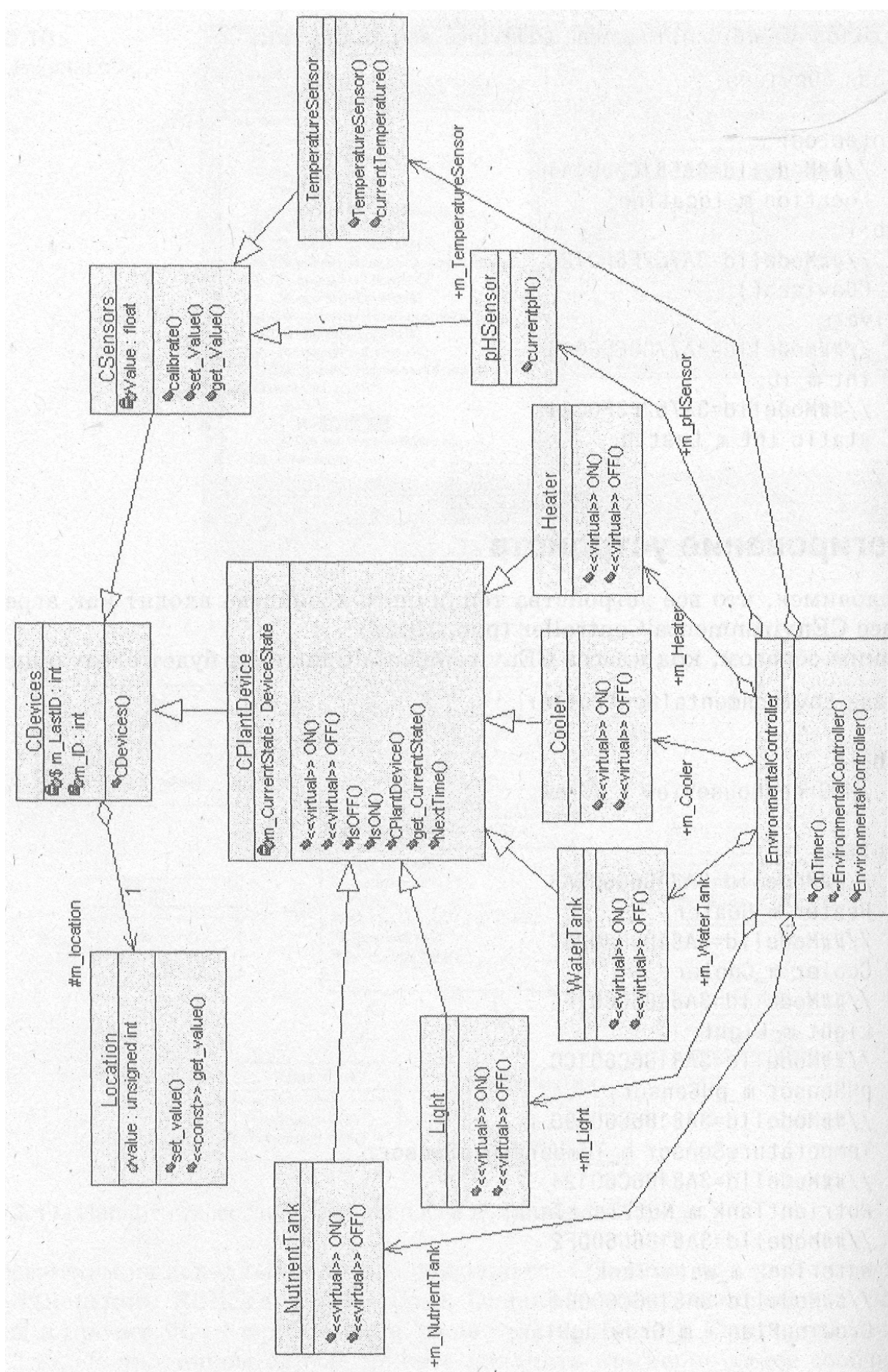


Рис. 19.12. Агрегирование классов устройств

Корректирование отклонений показателей

Для того чтобы контроллер начал корректировать отклонения показателей среды, необходимо изменить код обработчика таймера для класса CGreen-houseView, добавив в него вызов операции m_EnvironmentalController.On-TimerQ.

```
void CGreenhouseView::OnTimer(UINT nIDEvent)
{
    GetDocument()->m__Environment. Change(
    &GetDocument()->m_CurrentCondition,
    &GetDocument()->m_EnvironmentalController);          GetDocument()-
>m_EnvironmentalContrdller.OnTimer(
    &GetDocument()->m_CurrentCondit ion);
    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_TimeMD!=0){
    KillTimer(m_TimerlD); m_TimerlD = 0;
    }
    GetDocument()->m_Log.Process(
    &GetDocument()->m_CurrentCondition,this); Message("End plan");
    }
    CEditView::OnTimer(nIDEvent);
}
```

После запуска программы можно увидеть, как все отклонения в параметрах среды тут же начинают компенсироваться включением соответствующих исполнительных устройств (рис. 19.13).

По рисунку видно, что выполнение плана началось в девять часов утра. При этом температура и показатель pH уже были выше нормы. Для снижения температуры был включен вентилятор, а для снижения уровня pH открыт кран для поступления воды. При этом также было включено освещение теплицы. Постепенно температура и уровень pH пришли в норму, и в 9:10 исполнительные устройства были выключены. В 10:00 уровень pH был в норме, однако, температура опять поднялась на 1 градус, и снова потребовалось включение вентилятора.



Рис. 19.13 Регулировка показателей среды при помощи исполнительных устройств

Таким образом, тепличное хозяйство прекрасно работает. Устройства выполняют свои функции, что и требовалось от программы.

Вопросы для повторения

1. Как создать иерархию классов?
2. Какие особенности работы с Model Assistant при добавлении методов классов.
3. Как изменять иерархию классов?

Часть 5. Дополнительные возможности

Глава 20. Разработка Web приложений

Вступление

В предыдущих главах мы познакомились с тем, как при помощи Rational Rose создавать модели программных систем и генерировать по ним исходный код. Однако возможности программы на этом не ограничиваются. Подключаемые модули (Add-Ins) позволяют значительно расширить применение этого инструмента не только для моделирования поведения приложений, но и для создания модели баз данных и разработки структуры Web сайтов.

Это логично, поскольку приложение, работающее на Internet сервере, ничем не отличается от обычной программы. И даже простой Web сайт с линейной структурой страниц, который не использует средства Java или ASP (Active Server Pages) можно представить как приложение. Web сайту присущи все атрибуты настольного приложения. Здесь есть взаимодействие с пользователем, получение, хранение и передача информации, окна и меню.

Структура Web сайта также разрабатывается, затем кодируется и тестируется и, наконец, выставляется для всеобщего использования. Далее сайт сопровождается, добавляются и удаляются разделы, исправляются ошибки, расширяется информационное наполнение и функциональность.

Мы здесь не будем подробно останавливаться на принципах разработки Internet приложений, а попробуем создать небольшой сайт для продвижения в сети нашей только что разработанной программной системы. И в этом нам поможет Rational Rose. Конечно, это будет только первый шаг к созданию ныне модного приложения для е-коммерции, на примере которого мы изучим возможности инструмента для создания Internet приложений.

Для моделирования Internet приложений используется та же диаграмма классов, которая была рассмотрена в предыдущих главах.

Назначение Web Modeler

В Rational Rose для создания Web приложений включен Add In под названием Web Modeler (разработчик Web модели). Можно перевести этот термин как «модельер» Web, что, впрочем, не так далеко от истины, поскольку словарь Ожегова определяет слово «модельер» как специалист по изготовлению моделей одежды, в нашем случае «одежды» Web.

Для того чтобы начать работу с Web Modeler, необходимо установить его при помощи Add-Ins=>Add-Ins Manager=>Rose Web Modeler=ON.

После этого в меню Tools появится новый пункт Web Modeler (рис. 20.1).

Rational Rose Web Modeler позволяет моделировать приложения Web, создавать модель по существующему программному коду, а затем снова производить генерацию jsp, asp, и html файлов согласно изменениям в модели. Web Modeler позволяет разработчику все свое внимание уделить бизнес логике приложения и связям, не вникая в детали реализации, за которые отвечает этот подключаемый модуль.

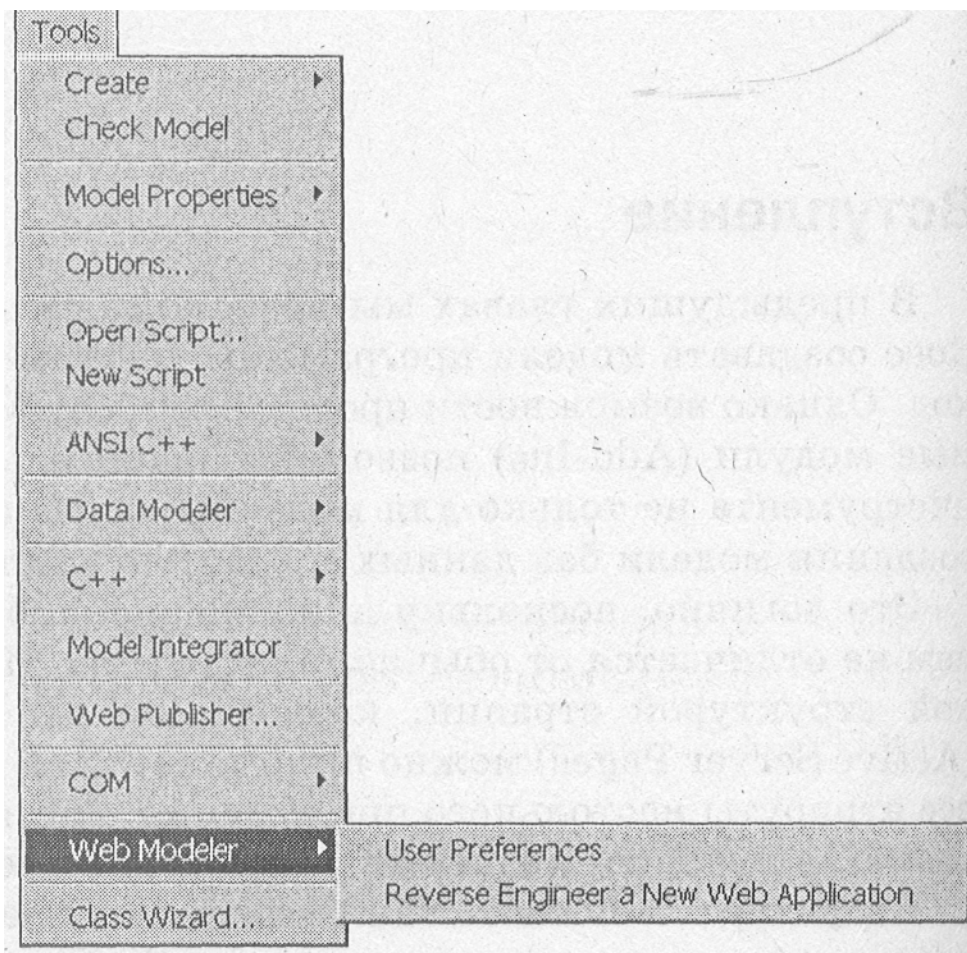


Рис. 20.1 Меню Web Modeler

Меню

Как вы уже видите на рисунке, меню состоит всего из двух пунктов: User Preferences (установки пользователя); Reverse Engineer a New Web Application (построение модели по коду).

User Preferences (установки пользователя)

Данный пункт предназначен для изменения установок модели, принятых по умолчанию для конкретного пользователя, что позволяет нескольким пользователям работать с одной Web моделью, но со своими настройками. Эти установки позволяют контролировать построение модели по коду, создание кода по модели, генерировать структуру каталога для ссылок, связей и HTML форм. При вызове этого пункта

активизируется окно (рис. 20.2) настройки параметров.

Установки позволяют настроить действия генератора для тегов HTML и скриптов. Возможен выбор между удалением или установкой как комментария операций и тегов, что необходимо для предохранения полученного кода от обработки внешними приложениями. Пункт *Resolve relative paths dynamically* (динамическое изменение связанных путей) разрешает генератору кода Rational Rose изменять пути к создаваемым файлам динамически в случае изменений в структуре пакетов модели.

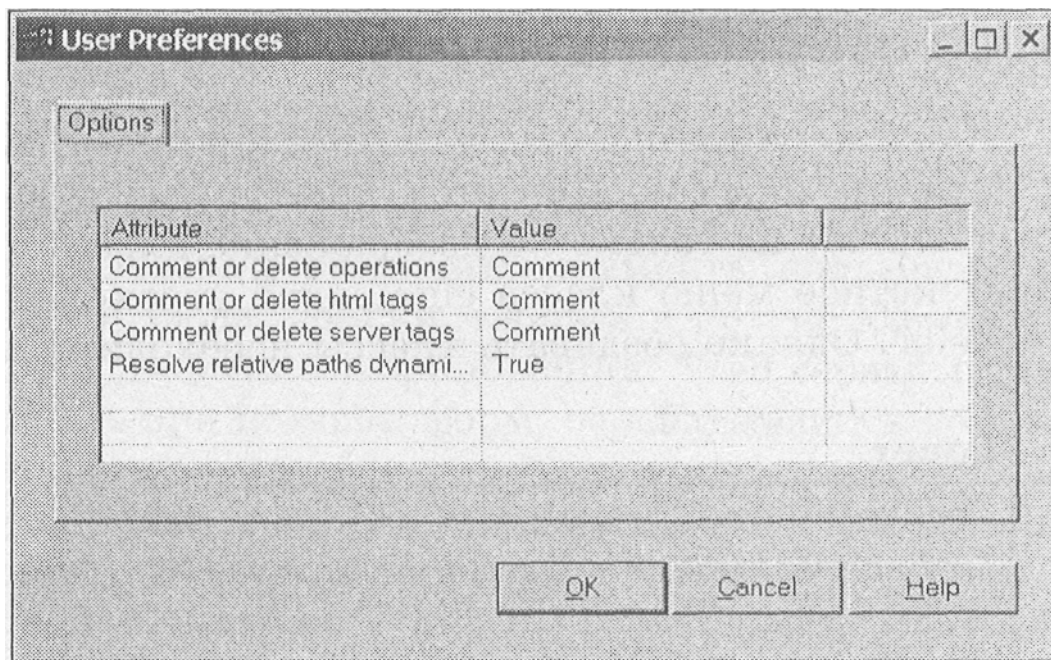


Рис. 20.2 User Preferences Web Modeler

Reverse Engineer a New Web Application (построение модели по коду приложения)

Данный пункт позволяет создавать модель по готовому коду приложения. При его активизации запускается мастер *Reverse Engineer*, который помогает осуществить использование готового кода для создания модели.

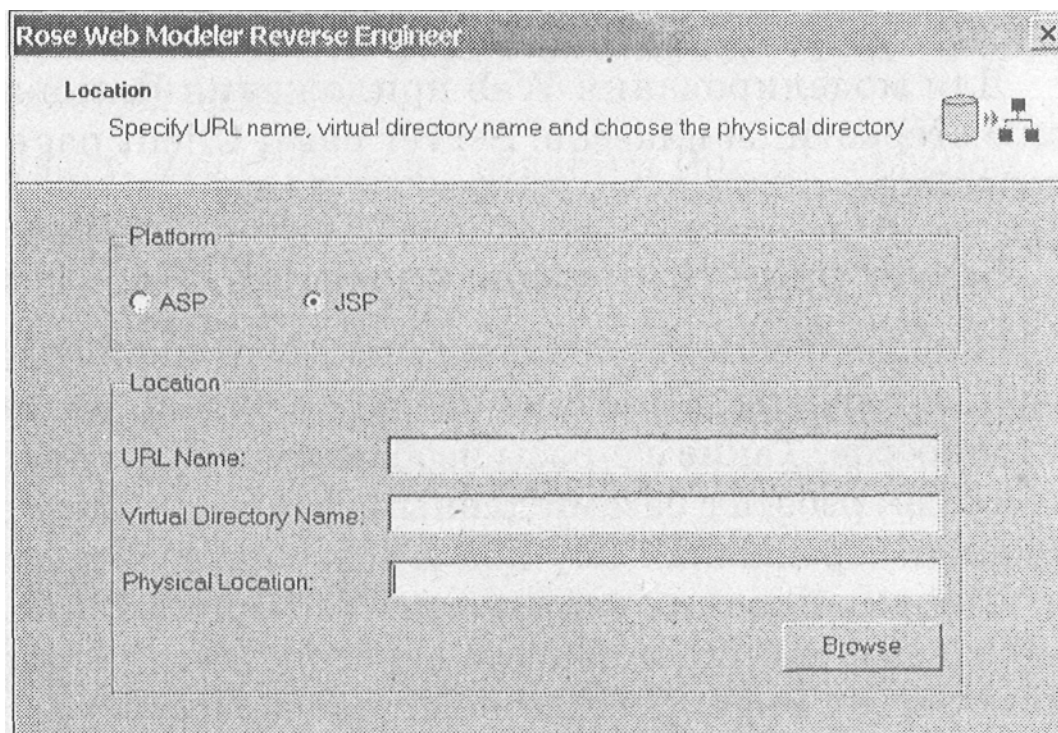


Рис. 20.3 Rose Web Modeler Reverse Engineer

Мастер позволяет задать платформу ASP (Active Server Pages) или JSP (Java Server Pages) и каталог, в котором находится исходный код. После того как мастер закончит свою работу в текущей модели, образуется структура выбранных классов, в которой отражаются связи, атрибуты и стереотипы.

Замечание

После внесения изменения в модель есть возможность перенести эти изменения в исходный код при помощи Web Modeler=>Generate Code из контекстного меню класса.

Web стереотипы

Как уже упоминалось, для моделирования Web приложения используется диаграмма классов. При этом классы имеют специальные стереотипы, которые влияют не только на изображение класса на диаграмме, но и дополняют контекстное меню класса еще одной строкой спецификаций и пунктом Web Modeler. Однако создавать классы необходимо в Web нотации.

Совет

Для того чтобы создавать классы в Web нотации, сделайте Menu:Tools=>Options=>Notation=>Default Language=Web Notation.

Для создания классов с определенными Web стереотипами можно воспользоваться панелью инструментов диаграммы классов. Конечно, ее необходимо предварительно настроить, чтобы нужные значки были доступны.

Замечание

Для настройки выберите пункт **Customize** из контекстного меню строки инструментов.

Для моделирования Web приложения Rational Rose предоставляет следующие стереотипы классов: **Server page**, **Client page**, **HTML Form**. Рассмотрим их подробнее.

Server page (страницы сервера)



Многие Web приложения используют скрипты, выполняемые на стороне сервера, и скрипты, выполняемые на стороне клиента, для ввода данных и обработки запросов. Такие запросы необходимы для работы с различными сервисами, такими как работа с базами данных, файлами, электронной почтой. Web сервер выполняет переданный ему код и возвращает форматированные данные клиенту.

Для уменьшения сложности Web приложения проектировщик может моделировать функциональность серверной части приложения при помощи стереотипа **Server Page**. Web Modeler поддерживает моделирование серверных страниц типа ASP и JSP.

Client page (страницы клиента)



Этот стереотип позволяет показать использование скриптов на стороне клиента. Также он отражает простые HTML страницы, текст, DHTML, графику, мультимедиа объекты. Страницы могут комбинировать статический HTML текст и динамические страницы, созданные при помощи JavaScript и VBScript. • Динамические страницы могут управляться программными событиями, содержать переменные, подпрограммы и функции.

Form (формы)



HTML формы позволяют пользователю взаимодействовать с Web-приложением. При помощи форм происходит обработка ввода пользователя и передача его серверу. Пользователи могут вводить данные, делать выбор посредством меню. Программисты могут использовать различные элементы пользовательского интерфейса, такие как поля ввода, списки, кнопки и другие элементы, которые могут понадобиться для разработки приложения. Web серверы принимают запросы, сформированные при помощи форм, обрабатывают и возвращают HTML страницы клиенту.

Когда создается модель Web приложения, формы используются для объединения полей ввода на клиентских страницах и не могут содержать собственных операций. Атрибуты в формах могут быть представлены только как поля ввода.

Связи

Для связей страниц и форм Web Modeler предоставляет дополнительные стереотипы, которые позволяют отразить характер связей. Кратко рассмотрим эти стереотипы:

Link. Показывает гиперссылку одной страницы на другую и может носить двунаправленный характер;

Submit. Тип связи, показывающий, что форма взаимодействует со страницей на сервере и передает ей данные;

Build. Тип связи показывает, что страница создается (build) сервером;

Redirect. Передача управления (redirect) одной серверной странице другой. Используется для страниц ASP;

Includes. Показывает, что одна страница включается в другую;

Forward. Связь похожа на Redirect, также обозначает передачу управления одной странице другой, однако используется для страниц JSP.

Шаги создания Web приложения

Теперь мы готовы для создания простого Web приложения. Допустим, мы хотим создать приложение, состоящее из главной страницы, которая создается сервером и включает меню. На этой же странице находится форма ввода, которая, например, позволяет отправлять пожелания пользователей авторам программы. Эта форма ввода будет взаимодействовать со страницей сервера, которая и будет непосредственно отправлять запросы. Для создания нашего приложения необходимо пройти несколько простых шагов.

Установите язык моделирования, как уже упоминалось ранее, Menu: Tools=>Options=>Notation=>Default Language=Web Notation.

Создайте виртуальный каталог. Из контекстного меню Logical View выберите Web Modeler=>New=>Virtual Directory Name и задайте каталоги, в которых будет находиться исходный текст.

Добавьте необходимые элементы в модель посредством строки инструментов диаграммы классов.

Установите свойства и добавьте атрибуты.

Создайте исходный текст.

Замечание

При создании элементов модели будьте внимательны: если элементы не будут входить в созданный виртуальный каталог, то свойства страниц могут быть недоступны.

Описанное приложение может выглядеть в Rational Rose примерно так, как представлено на рис. 20.4.

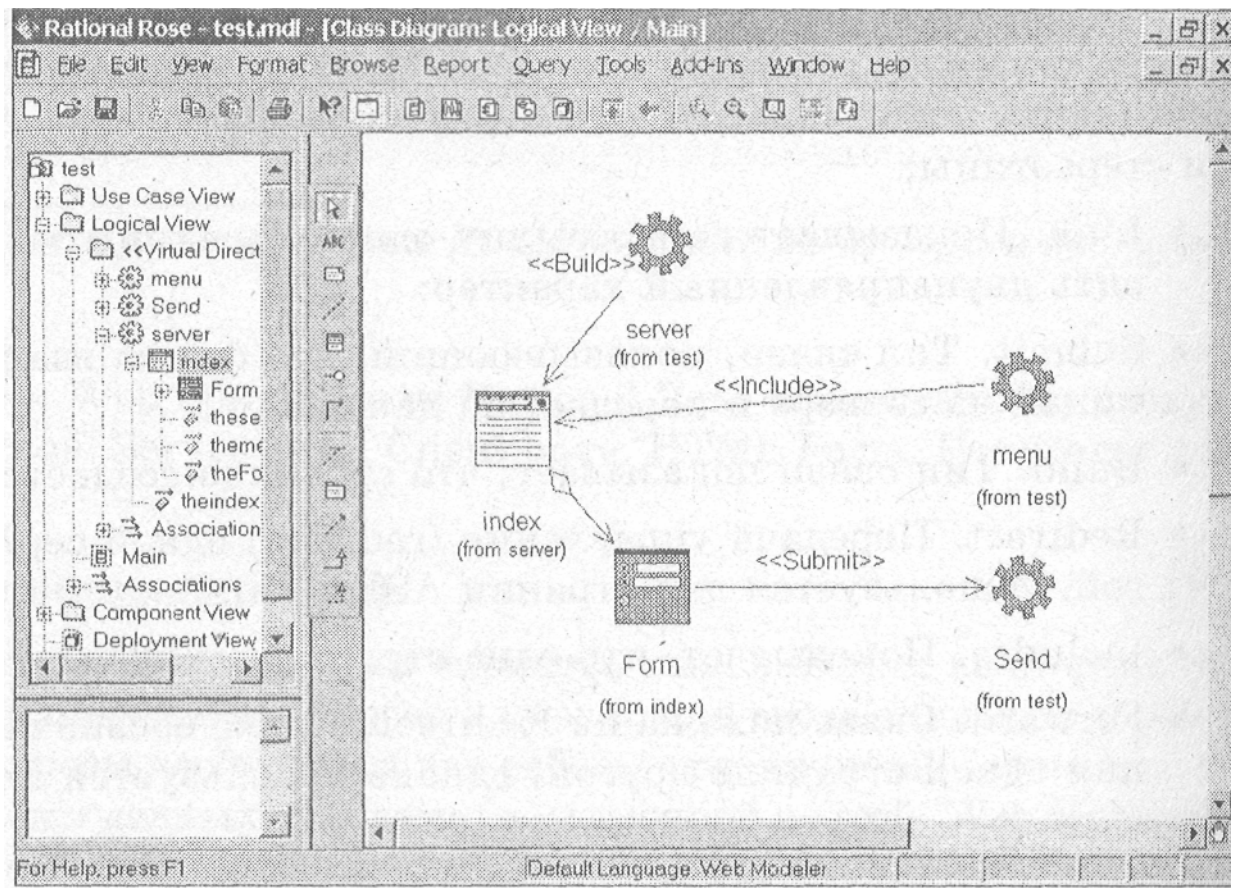


Рис. 20.4 Web модель

Используем эту модель для изучения атрибутов, а к созданию новых элементов и генерации исходного текста вернемся позднее.

Спецификации Client Page

При двойном клике по элементу модели активизируется окно Web-спецификаций. Набор спецификаций для страниц, форм и связей - различен. На рис 20.5 показана вкладка General для объекта Client Page.

На этой вкладке доступны следующие спецификации:

Alink - устанавливает цвет ссылок;

Background - устанавливает рисунок для фона страницы;

BgColor - устанавливает цвет фона;

Class - устанавливает имя класса;

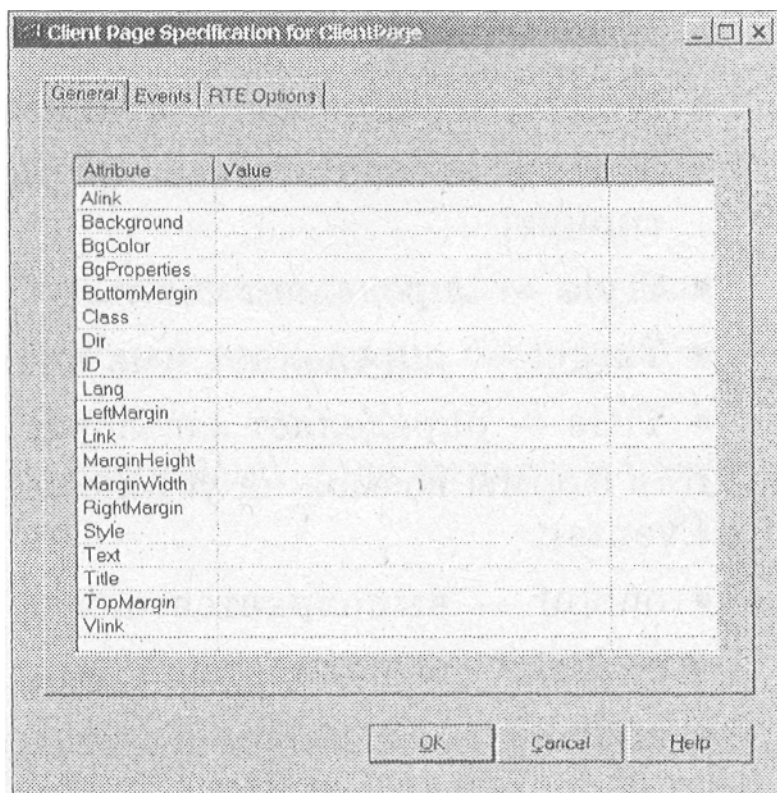


Рис. 20.5 Спецификации Client Page

Dir - устанавливает направление текста в элементе;

ID - идентификатор элемента;

Lang - установка языка элемента;

Link - установка цвета текста ссылок;

MarginHeight - установка высоты отступа фрейма;

MarginWidth - установка ширины отступа фрейма;

Style - установка стиля для элемента;

Text - установка цвета текста;

Title - установка дополнительной информации элемента;

Vlink - установка цвета ссылок, которые посещались.

Обработчики событий страницы устанавливаются на вкладке Events. Это onload - событие, отрабатывающее при загрузке, и onunload - событие при закрытии страницы. Вкладка Round Trip Engineering Options позволяют установить имя файла для генерации кода (File Name) и включить или отключить синхронизацию кода (RTE synchronization).

Спецификации Form

Вкладка General для объекта Form скромнее, зато обрабатываемых событий, представленных на вкладке Events значительно больше. Для спецификаций General доступна установка следующих полей:

Action - тип процесса, например имя подпрограммы обработки;

Class - имя класса;

Enctype - тип кодировки для данных формы;

ID - идентификатор;

Lang - установка языка для значений и текста формы;

Method - устанавливает HTTP метод для передачи данных (GET,

POST);

Name - устанавливает форму, ссылающуюся на страницу стилей или скрипт;

Style - определяет стиль;

Target - определяет имя целевого фрейма;

Title - определяет заголовок.

Для формы возможна установка обработчиков следующих событий (вкладка Events):

onblur - выполняется, когда элемент теряет фокус ввода;

onclick - выполняется, когда пользователь кликнул по элементу мышкой;

Ondblclick - выполняется, когда пользователь дважды кликнул мышкой;

onfocus - выполняется, когда элемент получает фокус ввода;

onkeydown - выполняется, в течение нажатия клавиши;

onkeypress - выполняется, когда пользователь нажимает клавишу;

onkeyup - выполняется, когда пользователь отпускает клавишу;

onmousedown - выполняется, когда пользователь нажимает клавишу мыши;

onmousemove - выполняется, когда пользователь двигает мышкой;

onmouseout - выполняется, когда указатель мыши вышел за границы элемента;

onmouseover - выполняется, когда указатель мыши проходит над элементом;

onmouseup - выполняется, когда кнопка мыши отпущена;

onreset - выполняется, когда форма уничтожается;

onsubmit - выполняется, когда форма передает данные (submit).

Спецификация Server Page

Для серверных страниц JSP доступна установка следующих атрибутов:

AutoFlush - установка автоматического сброса буфера;

Buffer - устанавливает размер буфера для клиентского браузера (по умолчанию 8 Кб);

Charset - устанавливает кодировку (по умолчанию ISO-8859-1);

ContentType - устанавливает тип кодировки MIME (по умолчанию текст);

ErrorMessage - устанавливает имя страницы обработчика ошибок;

Extends - устанавливает класс сервлета для страницы;

Import - список пакетов через запятую для импорта;

Info - комментарий, используется как информация для сервлета;

IsErrorMessage - индицирует, что окно сообщения об ошибке будет показываться (по умолчанию false);

IsThreadSafe - устанавливает, будет ли выполняться защита потока;

Language - устанавливает серверу использовать язык Java для этого файла;

Session - устанавливает, что клиент должен быть подключен с использованием JSP (по умолчанию true).

Создание формы ввода

Основа Web приложения - формы ввода. Именно с их помощью пользователь взаимодействует с программой, управляя работой приложения. Рассмотрим создание формы и генерацию исходного текста для нее. Создайте Client Page из контекстного меню созданного ранее виртуального каталога RClick=> Web Modeler=>New=Client Page.

Замечание

Если вы используете Server Роде для генерации страниц (Build), то достаточно создать страницу сервера, а клиентская страница будет создана в модели автоматически.

Для того чтобы посмотреть результат, не подключаясь к серверу, мы будем использовать форму, включенную в HTML страницу. Перейдите в окне Browse на созданную страницу Client Page и из контекстного меню создайте форму ввода. RClick=>Web Modeler=>New=>Form. Теперь, если перетащить мышкой созданные элементы на диаграмму, то связи будут показаны автоматически.

Допустим, форма должна будет содержать текст и два поля ввода и кнопку «Отправить». Встаньте в окне Browse на форму и добавьте новые поля через пункт контекстного меню Web Modeler.

Добавлять можно поля различных типов: HTML Input - для полей ввода: простых текстовых полей, паролей, кнопок, скрытых полей и всего того, что может понадобиться в формах для ввода данных; HTML Select - для списков; HTML Text Area для ввода многострочного текста.

Создайте исходный текст для клиентской страницы RClick=>Web Modeler =>Generate Code и просмотрите его при помощи пункта Browse Code. У вас должно получиться то, что показано на рис. 20.6.

Таким образом, у нас получилась полноценная форма ввода. Перейдите в каталог с исходным текстом и откройте файл Client Page.html чтобы убедиться, что созданная форма позволяет вводить и передавать данные, как показано на рис. 20.7.

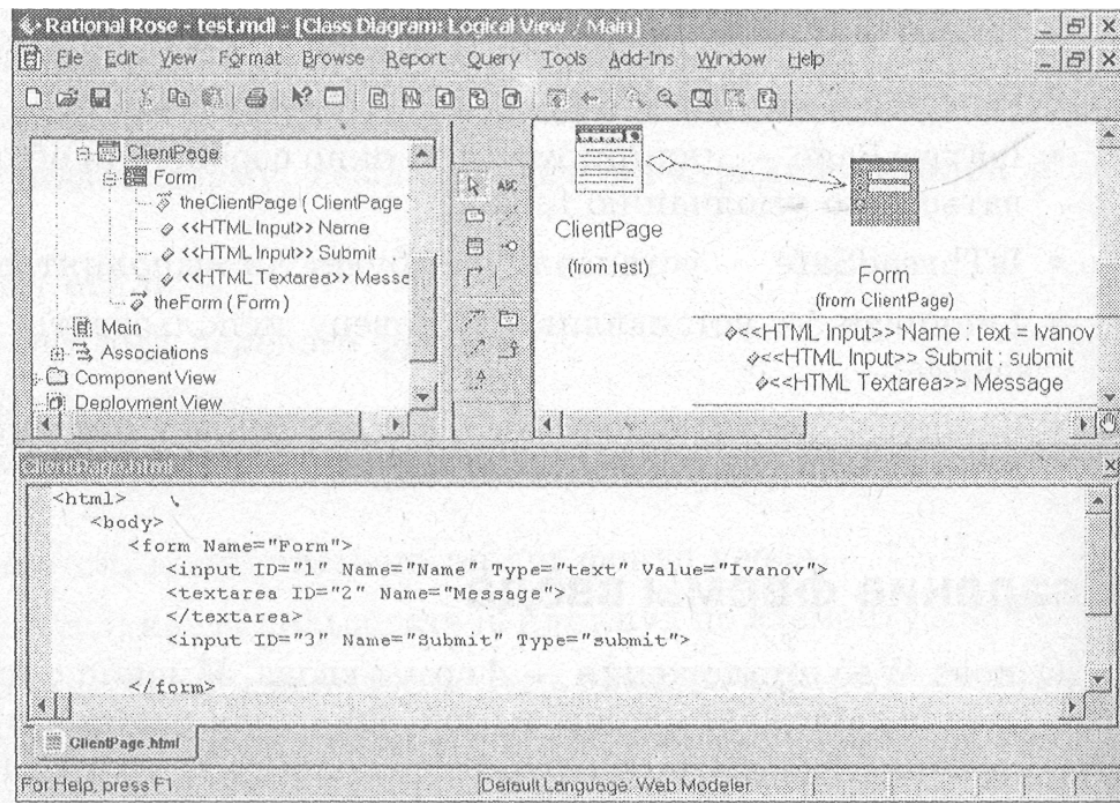


Рис. 20.7 Форма ввода в браузере

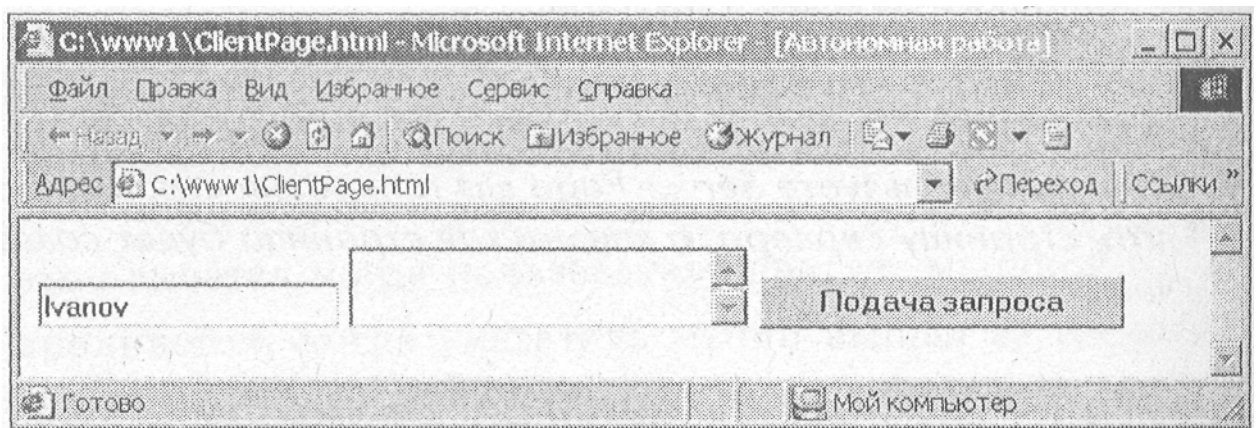


Рис. 20.6 Код формы

Конечно, это лишь скелет формы. Теперь необходимо эстетично расставить поля ввода, добавить текстовые сообщения и обработчик формы, стили, фон и много чего еще, для того чтобы получить полноценное Web приложение, но это тема отдельного разговора.

Вопросы для повторения

1. Для чего предназначен модуль Web Modeler?
2. Какие доступны элементы для создания Web модели?
3. Какие связи используются для Web модели?
4. Из каких шагов состоит создание Web модели?
5. Какие элементы можно создать в форме ввода при помощи Rational Rose?

Глава 21. Моделирование данных

Вступление

При создании программных систем процесс создания структуры данных (модели) является одним из важнейших этапов. Однако до недавнего времени аналитики-проектировщики, работающие с Rational Rose, должны были обращаться к другим CASE-средствам для автоматизации этого процесса, например, к ERwin компании PLATINUM. С появлением подключаемого модуля (Add-In) под названием Data Modeler у разработчиков появилась возможность не отказываться от своего любимого инструмента и использовать Rational Rose не только для создания логического представления системы, но и для моделирования физического представления данных.

К сожалению, язык UML не имеет в своем составе средств, позволяющих адекватно отображать физическую модель данных. Для ее моделирования компания Rational Rose включила дополнительные стереотипы классов, которые не имеют отражения в стандартном языке UML. Таким образом, Data Modeler не работает с новым подмножеством языка, а является только инструментом для моделирования физической структуры данных. Data Modeler позволяет создавать все необходимые объекты базы данных: таблицы, триггеры, хранимые процедуры и представления данных, поддерживает работу с основными системами обработки баз данных: IBM DB2 MVS, UDB, Oracle, Microsoft SQL Server, Sybase Adaptive Server.

Меню Data Modeler

После установки модуля при помощи Add-In Manager, в меню Tools появится дополнительный пункт Data Modeler (рис. 21.1).

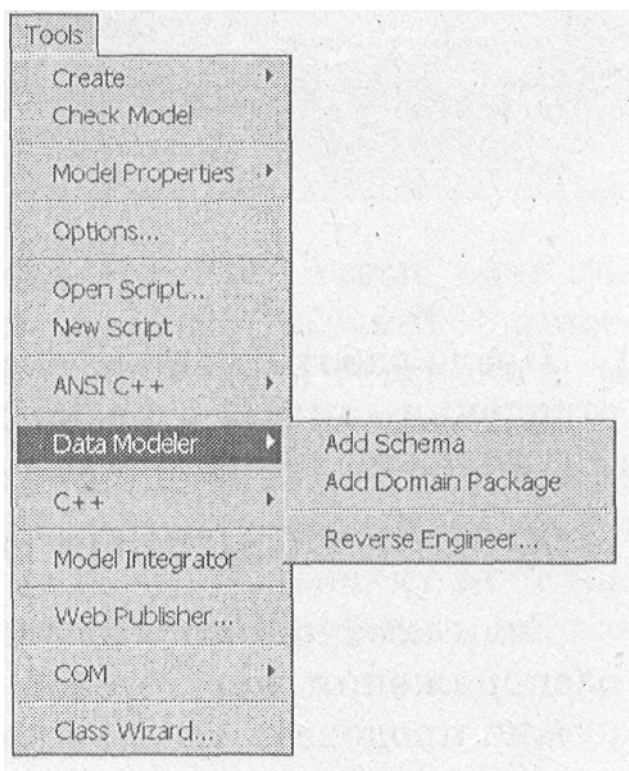


Рис.21.1 Меню Data Modeler

В контекстном меню объектов и диаграмм, где также появляется Data Modeler, этот пункт несет гораздо большую нагрузку, позволяя создавать новые таблицы, триггеры, хранимые процедуры, работать со схемами данных, выполнять другие функции. Возможности контекстного меню будут обсуждаться позднее. Меню состоит из трех пунктов:

Add Schema - добавить схему. Схема - основной контейнер, для создания модели данных. Первое, что потребуется для создания модели, - это создание схемы;

Add Domain Package - добавить доменный пакет. Домены позволяют определять типы данных, длины, точность, значения по умолчанию для колонок таблицы данных и могут использоваться для различных схем данных;

Reverse Engineering - обратное моделирование. Позволяет создать модель по готовой базе данных.

При запуске пункта Reverse Engineering запускается мастер, который позволяет провести обратное моделирование как из базы данных, так и из Data Definition Language script (скрипт на языке определения данных). Для базы данных достаточно задать тип базы данных и параметры соединения (рис. 21.2).

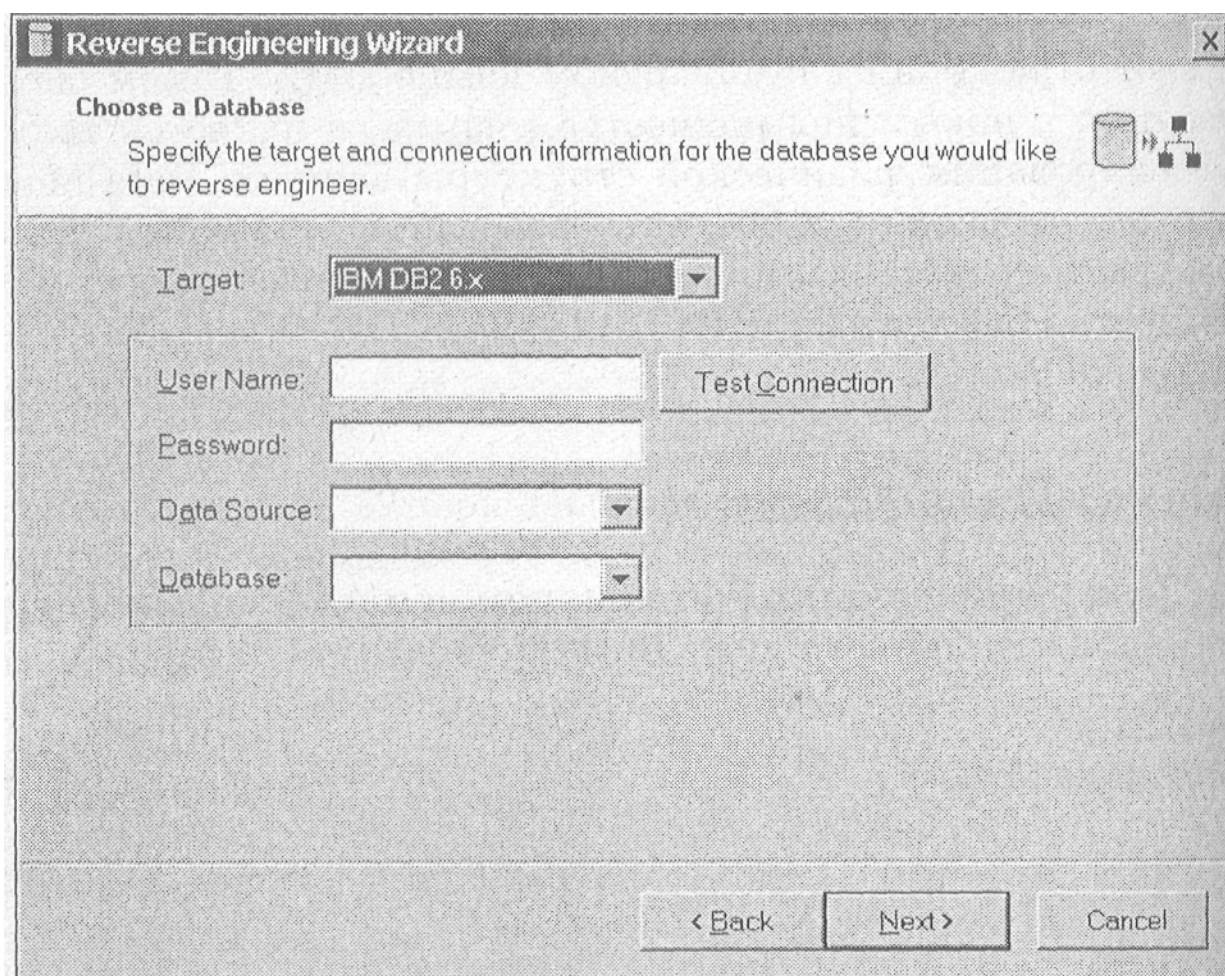


Рис. 21.2 Reverse Engineering Wizard

После работы мастера в модель будут перенесены таблицы, триггеры, представления данных и хранимые процедуры.

Порядок создания новой структуры данных

Если необходимо создать совершенно новую структуру данных, и в вашем распоряжении нет готовой базы данных для обратного моделирования, то нужно проделать следующие шаги:

Создать схему данных.

Создать таблицы данных в схеме.

Создать связи данных.

Перенести созданные объекты в базу данных.

Для начала мы разберем все эти действия по отдельности, а в конце главы создадим простую базу данных целиком.

Создание структуры данных

Первым шагом в создании структуры базы данных будет создание схемы, которая необходима для определения типа связи с конкретной базой данных. Схема данных позволяет трансформировать физическую модель в логическую и обратно и позволяет создавать диаграмму модели данных (Data Model Diagram).

Для создания схемы данных необходимо перейти в окно Browse и

из контекстного меню Logical View проделать RClick: Logical View=>Data Modeler=>New=>Schema.

У схемы данных пункт контекстного меню Data Modeler значительно полнее, чем одноименный пункт в меню Tools (рис. 21.3).

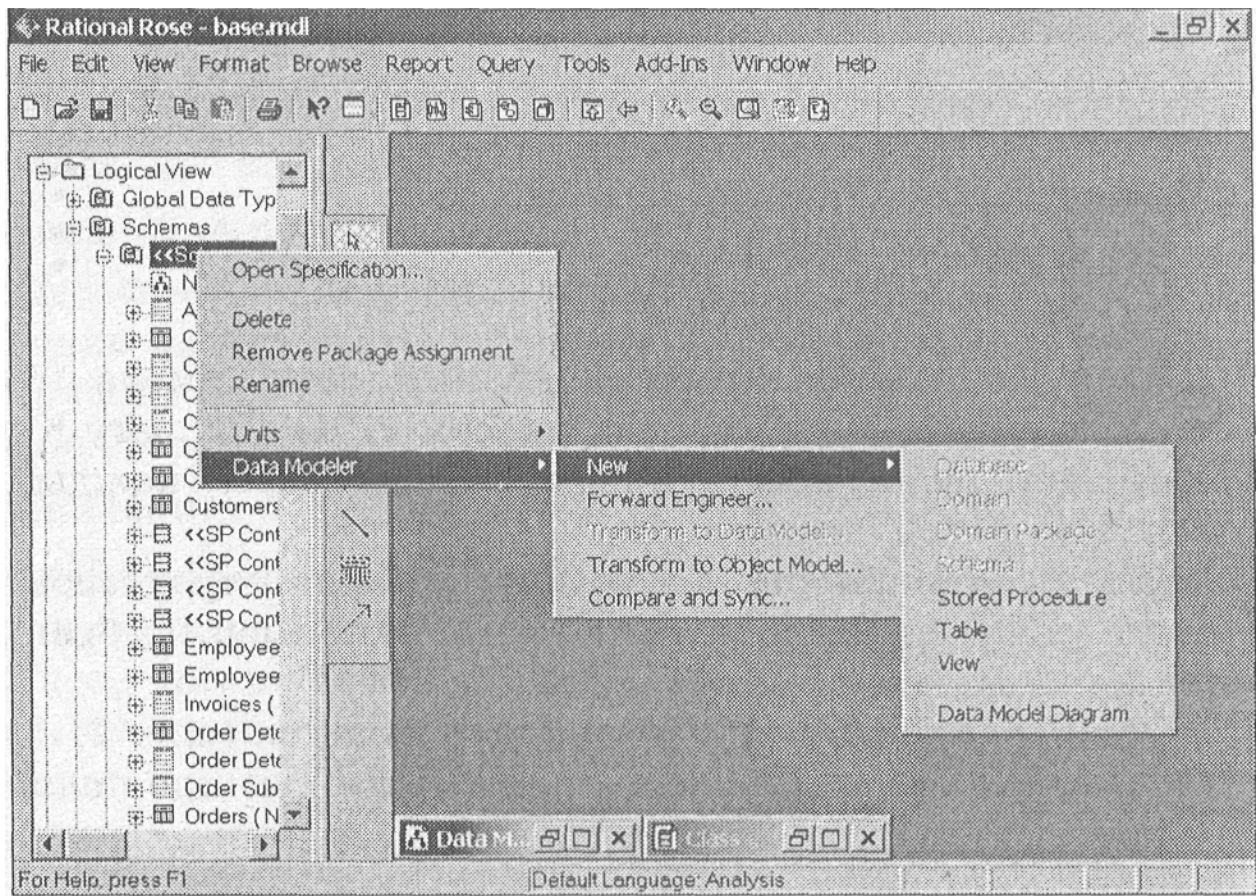


Рис. 21.3 Контекстное меню схемы данных

При помощи этого меню можно создать в схеме данных новые объекты (New), произвести перенос созданной модели в базу данных (Forward Engineering), трансформировать модель данных в объектную модель (Transform to Object Model) и провести синхронизацию модели данных с базой данных (Compare and Synchronizing). Работа мастера синхронизации практически ничем не отличается от мастера Reverse Engineering, однако результатом работы будет не перенос объектов данных в модель, а получение различий между моделью и реальной базой данных. Мастер также позволяет синхронизировать модель и базу данных.

Создание таблиц

После создания схемы данных появляется возможность создавать таблицы, представления и хранимые процедуры. В таблицы через окно спецификаций добавляются колонки, для которых определяются типы данных, размер, первичные ключи и т.д. Для часто используемых колонок возможно создание шаблонов, которые называются domain (домен).

Для создания таблицы данных необходимо из контекстного меню схемы данных выбрать Data Modeler=>New=>Table. Визуально таблица данных отображается как показано на рис 21.4.

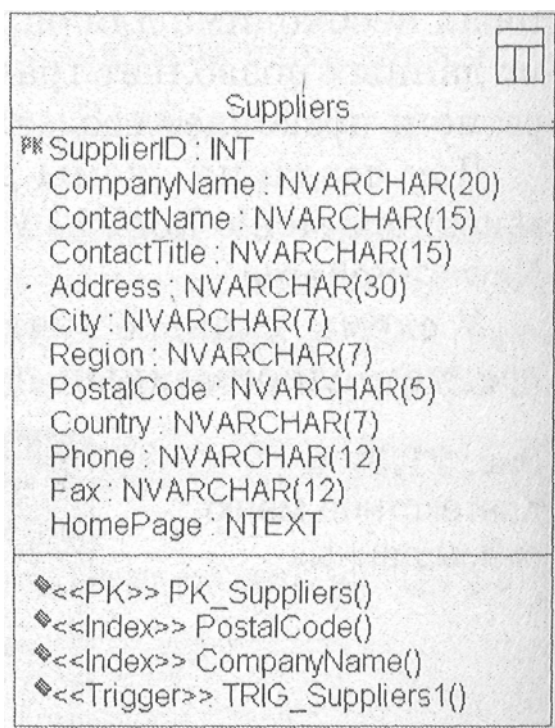


Рис. 21.4 Пример изображения таблицы данных

Вверху отражено название таблицы, затем имена и типы столбцов. Ключевое поле отмечено красным маркером «PK» (Primary Key), далее показаны ограничения целостности, индексы и триггеры, каждый из которых показан вместе со своим стереотипом.

Для задания колонок, ключей, индексов и триггеров необходимо активизировать спецификации для таблицы, например, выбрав пункт Open Specification из контекстного меню таблицы.

Окно спецификаций имеет шесть вкладок (рис. 21.5).

Закладка General позволяет задать следующие поля:

Name - имя таблицы;

Schema - схема данных, к которой принадлежит таблица;

Mapped - имя класса, с которым связана таблица;

Tablesapce - список имен табличных пространств в базе данных, позволяет выбрать табличное пространство к которому будет отнесена таблица данных.

Закладка Columns позволяет создать колонки в таблице данных, для чего устанавливать следующие поля:

Name - имя колонки;

PK - устанавливает, что колонка является первичным ключом;

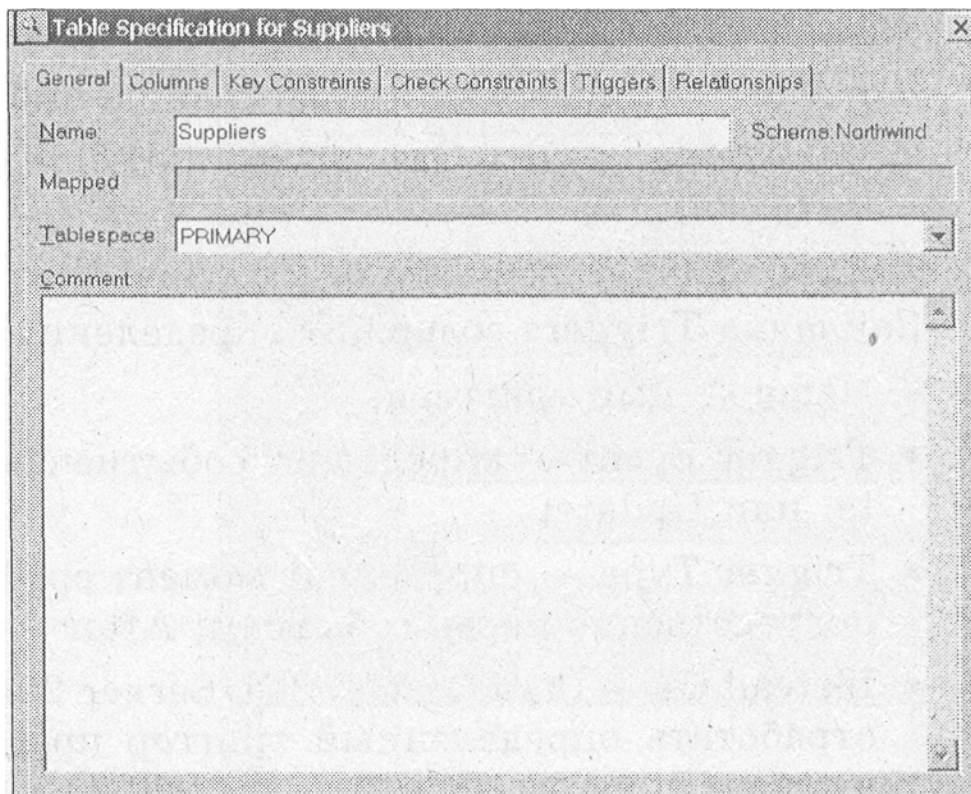


Рис. 21.5 Спецификации для таблицы данных

Domain - содержит имя домена, на который ссылается колонка;

Type - тип данных;

Not Null - определяет, что значения в колонке не могут быть NULL;

Unique - определяет, что значения должны быть уникальными;

Default - задает значение по умолчанию.

Закладка Key Constraints позволяет задать ограничения для колонок. Здесь можно задать ограничение на уникальность первичного ключа, ограничение на уникальность альтернативных ключей, а также просто определить индекс. На данной закладке присутствуют следующие поля:

Name - имя ограничения;

Type - тип ограничения, можно задать Primary Key Constraint, определяющее, что колонка является первичным ключом, Unique Key Constraint, определяющее, что значения в колонке будут уникальными, Index - определяет, что по колонке будет построен индекс, ускоряющий доступ к данным;

Deferrable - (для Oracle, SQL 92) определяет момент активизации ограничения;

Clustered - определяется, будут ли данные с близкими значениями ключа размещены в смежных блоках диска (clustered);

Unique - определяет уникальность индекса;

Fill Factor/PCTFree - числовое поле, определяющее фактор заполнения (fill factor) для поля;

Columns - список всех колонок;

Key Columns - список колонок, которые входят в ограничение

Закладка Check Constraints задает содержание ограничения. На этой

закладке присутствуют следующие поля:

Entire Table - определяет что ограничение действует для всей таблицы;

Name - список заданных ограничений;

Deferrable - (для Oracle, SQL 92) определяет момент активизации ограничения;

Expression - выражение ограничения. Закладка Triggers содержит определения триггеров:

Name - имя триггера;

Trigger Event - определяет событие срабатывания триггера Insert, Delete, или Update;

Trigger Type - определяет момент срабатывания триггера: Before - непосредственно перед событием; After - после события;

Instead Of - (для Oracle, SQL Server 2000 views) показывает, что должен отработать определенный триггер представления данных, а не триггер в таблице данных;

Action Body - определение триггера.

Закладка Relationship показывает установленные связи таблицы. Таким образом, спецификации позволяют указать все необходимое для определения таблицы данных от размера и типа колонок до триггеров.

Создание связей

После создания таблиц необходимо установить связи между ними. Для модели данных используются следующие виды связей: Identifying (идентифицирующая) сильная связь между родительской и дочерней базой данных; Non-identifying (неидентифицирующая) - независимая связь между таблицами. При этом каждый первичный или уникальный ключ помещается автоматически в дочернюю таблицу как foreign key и помечается красным маркером FK. Для идентифицирующей связи, в отличие от неидентифицирующей, в чения первичного ключа учитывается foreign key.

Для создания идентифицирующей связи между таблицами необходимо проделать следующее Menu: Tools => Create => Identifying Relationship. Когда курсор приобретет вид стрелки, соедините необходимые таблицы связью. Для создания неидентифицирующей связи используйте пункт Non-Identifying Relationship.

После создания связи необходимо настроить ее спецификации. Окно спецификаций активизируется в результате двойного нажатия мышью на связи или из контекстного меню связи (рис. 21.6).

Спецификации включают в себя три вкладки:

General - Основные свойства связи. Здесь задаются: имя связи; тип связи; наименования ролей (Parent, Child);

Migrated Key - Содержит список внешних ключей, образующихся в результате создания связи;

RI - Задание условий ссылочной целостности. Ссылочная целостность обеспечивается двумя способами: на основе триггеров; на основе декларативной ссылочной целостности (с использованием ограничений внешних ключей).

Рис. 21.6 Relationships Specification

Перенос структуры в базу

Последним шагом будет перенос созданной структуры в базу данных. Rational Rose позволяет создать скрипт на языке DDL и позволяет сразу же запустить его или сделать это позднее.

Для переноса из контекстного меню схемы данных выберите Data Modeler=>Forward Engineering. После этого активизируется окно мастера переноса структуры данных (рис. 21.7).

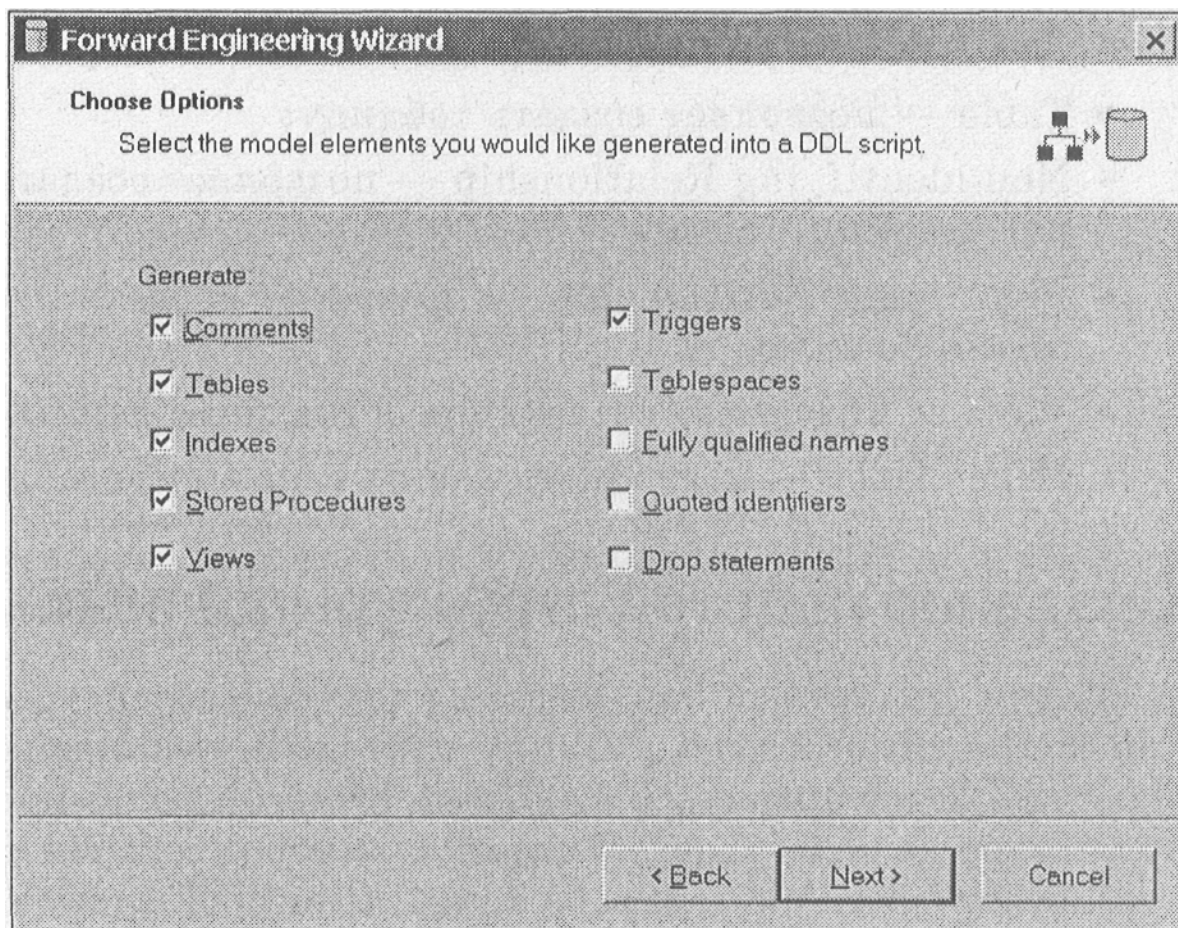


Рис. 21.7 Мастер переноса структуры данных

Вы можете установить отметки напротив необходимых для переноса элементов, которые и будут включены в создаваемый скрипт.

Диаграмма данных

Диаграммы классов для создания структуры данных оказалось недостаточно, поэтому разработчики включили в Rational Rose новую диаграмму Data Model Diagram. Однако не ищите ее в системной строке инструментов. Она создается из контекстного меню схемы базы данных при помощи пункта Data Modeler. Создайте новую схему данных при помощи RClick: Logical View=> Data Modeler=>New=>Schema и из нее создайте новую диаграмму данных. Для этого вызовите для вновь созданной схемы контекстное меню, выберите Data Modeler=>New=>Data Model.

Эта диаграмма отражает физическую модель данных, в отличие от логической модели, которая представлена, например, в диаграмме классов. При активизации диаграммы становится доступна строка инструментов.

Инструменты позволяют создавать объекты базы данных, такие как таблицы, представления, связи одним щелчком мыши.

В строке инструментов доступны следующие значки:

Selection Tool -позволяет выбрать элементы диаграммы;

Text Box -позволяет создавать надписи;

Note - создает комментарии;
 Anchor Note To Item - связывает комментарии с элементом;
 Table - позволяет создать таблицу;
 Non-identifying Relationship - позволяет создать неидентифицирующую реляционную связь;
 Identifying Relationship - позволяет создать идентифицирующую реляционную связь;
 View - позволяет создать представление данных;
 Dependency - позволяет указать связь представления данных и таблицы.

Создание простой структуры данных

Теперь мы готовы для создания простой структуры данных. Предположим, что необходимо создать таблицу протокола некоторых событий, при этом сам протокол будет храниться в таблице Protocol, полями которой будут ID - идентификатор записи, Time - время, в которое событие произошло. Таблица Protocol будет связана с таблицей Event, в которой будут храниться названия событий, а просмотр будет осуществляться через представление данных View_Pro-ocol. Описанная схема данных будет выглядеть, как показано на рис. 21.9.

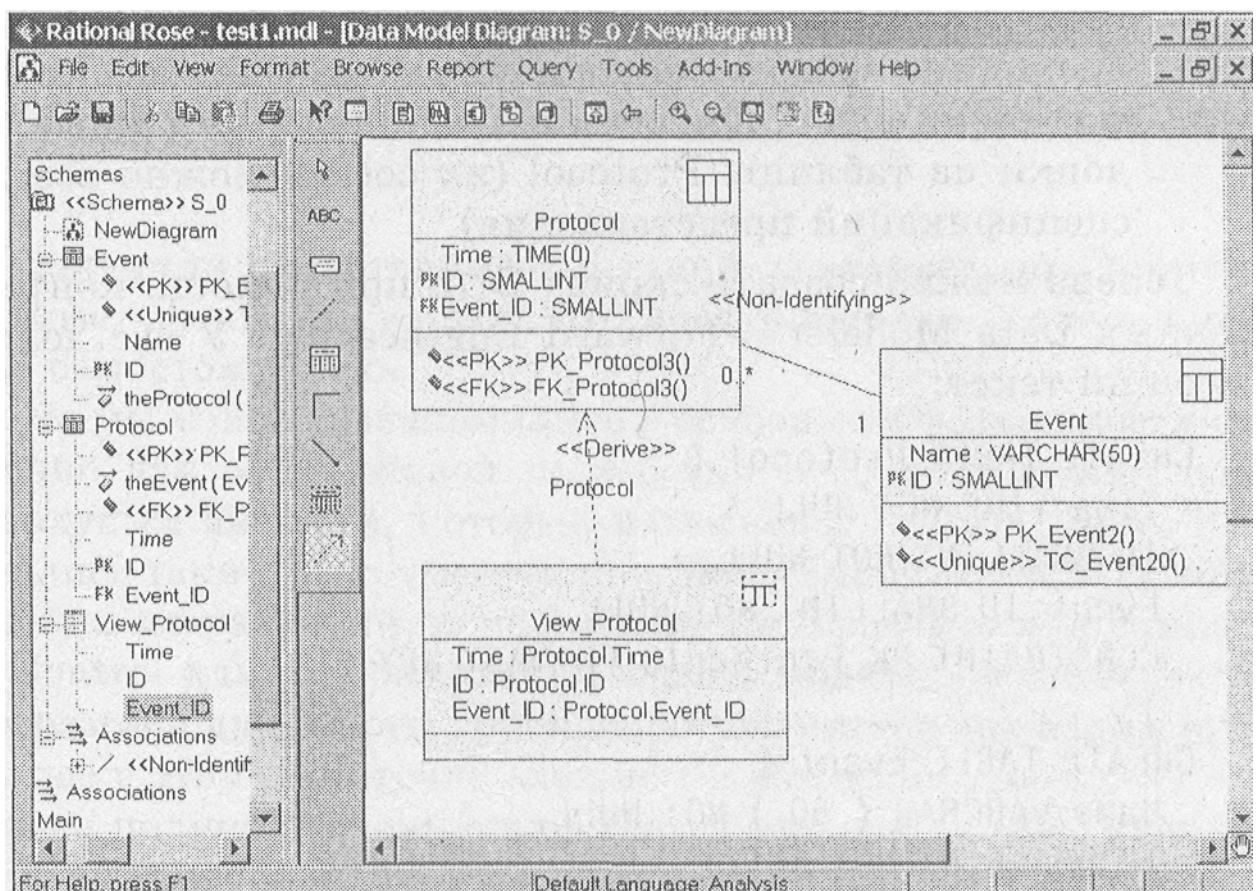


Рис. 21.9 Диаграмма данных

Для создания этой схемы проделайте следующие шаги:

1. Создайте новую схему данных: RClick:Logical View=>Data

Modeler=>New=>Schema.

2. Из контекстного меню схемы данных создайте новую диаграмму данных. Data Modeler=>New=>Data Model. Активизируйте диаграмму двойным щелчком мыши.

3. Из контекстного меню схемы данных или при помощи строки инструментов данных создайте таблицы и представления данных. Data Modeler=>New=>Table, Data Modeler=>New=>View.

Совет

Не забудьте «перетащить» вновь созданные элементы на диаграмму данных.

4. Из контекстного меню таблиц создайте новые колонки Data Modeler=>

New=>Columns. Замечание

Колонка Eventf_ID и все колонки представления данных будут созданы автоматически после создания связей.

5. Двойным щелчком мыши на колонке активизируйте окно свойств и настройте тип данных.

6. Соедините таблицы Event и Protocol неидентифицирующей связью. При этом в таблице Protocol будет создана новая колонка помеченная, маркером FK (Foreign Key) переименуйте ее в Event_ID.

7. Соедините представление данных View_Protocol и таблицу Protocol связью Dependency, при этом в представлении данных будут перенесены колонки из таблицы Protocol (их состав можно всегда изменить из окна спецификаций представления).

Теперь можно создать скрипт DDL при помощи контекстного меню схемы данных Data Modeler=>Forward Engineering. У вас должен получиться следующий текст:

```
CREATE TABLE Protocol ( Time TIME NOT NULL,\ ID SMALLINT NOT NULL, Event_ID SMALLINT NOT NULL, CONSTRAINT PK_Protocol3 PRIMARY KEY (ID) );
```

```
CREATE TABLE Event ( Name VARCHAR ( 50 ) NOT NULL, ID SMALLINT NOT NULL, CONSTRAINT PK_Event2 PRIMARY KEY (ID), CONSTRAINT TC_Event20 UNIQUE (ID) );
```

```
ALTER TABLE Protocol ADD CONSTRAINT FK_Protocol3 FOREIGN KEY (Event_ID) REFERENCES Event (ID) ON DELETE NO ACTION ON UPDATE NO ACTION; CREATE VIEW View_Protocol (Time, ID, Event_ID) AS SELECT Protocol.Time, Protocol.ID, Protocol.Event_ID FROM Protocol;
```

Теперь вы можете использовать полученный скрипт для генерации структуры базы данных.

Вопросы для повторения

1. Для чего используется Data Modeler?
2. Какие системы управления базами данных поддерживает Data

Modeler?

3. Какие возможности предоставляет меню Data Modeler в меню Tools?
4. Каков порядок создания новой структуры данных?
5. Для чего служит схема данных?
6. Как настроить ограничения в колонках данных?
7. Как перенести созданную структуру в базу данных?

Заключение

Теперь, когда вы получили представление о таком замечательном инструменте, как Rational Rose, возникает вопрос, что делать дальше. Конечно же, продолжать изучение самостоятельно.

Нельзя забывать, что мы только ознакомились с основными возможностями, которых уже достаточно для эффективной разработки и сопровождения программ. Это только верхушка айсберга, который называется Rational Rose. Вне нашего внимания остались такие инструменты, как Model Integrator для сравнения моделей и выяснения их различий, использование макросов для оптимизации работы. Version Control для сопровождения версий моделей, возможности, предоставляемые программой для работы группы программистов над одним проектом и другие, скрытые в этом достаточно сложном и объемном пакете.

Мощь Rational Rose в полной степени открывается при совместном использовании программы с другими продуктами компании Rational. Такими как Rational RequisitePro для управления требованиями. Rational Test, для создания сценариев тестирования, Rational SoDa для создания документации, Rational Clear Case для управления версиями.

Мне остается пожелать читателю не откладывая начать применять полученные знания на практике. И конечно же, продолжить дальнейшее изучение продуктов компании Rational Software.

Литература

Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++, 2-е изд./Пер с англ. М.: «Издательство Бином», СПб.: «Невский диалект», 1999 г. 560 с., ил.

Страуструп Б. Язык программирования С++: Пер. с англ. М. Радио и связь. 1991 352 с.: ил.

Шилдт. Г. МFC: основы программирования: Пер. с англ. - К.: Издательская группа BHV, 1997 560 с.

Шилдт Г. Теория и практика С++: пер. с англ. СПб.: BHV Санкт-Петербург, 1999. 416 с., ил.

Шилдт. Г. МFC: основы программирования: Пер. с англ. - Киев: Издательская группа BHV, 1997. - 560 с.

Шилдт. Г.; Программирование на С и С++ для Windows 95. - Киев: Торгово-издательское бюро BHV, 1996. 400 с., ил.

Вендров А.М. CASE-технологии. Современные методы и средства проектирования информационных систем. М.: Финансы и статистика, 1998. 176 с., ил.

Липаев В. В. Системное проектирование сложных программных средств для информационных систем. М.: СИНТЕГ, 1999. 224 с.

Кратчен Филипп. Введение в Rational Unified Process. 2-е изд.: Пер. с англ. - М.: Издательский дом «Вильяме», 2002. - • 240 с.: ил.

Ларман К. Применение UML и шаблонов проектирования. : Пер с англ. : Уч. Пос. М.: Издательский дом «Вильяме», 2001.- - 496 с.: ил.

Фраулер М., Скотт К. UML в кратком изложении. Применение стандартного языка объектного моделирования: Пер. с англ. - М.: Мир, 1999. 191 с., ил.

Приложения

Сочетания клавиш, которые действительны для всех окон

Команды редактирования	
Отменить	CTRL-Z
Забрать в буфер с удалением	CTRL-X
Скопировать в буфер	CTRL-C
Вставить	CTRL-V
Удалить	DELETE
Удалить из модели	CTRL-D
Переместить	CTRL-L
Отметить все	CTRL-A
Команды просмотра	
Увеличить до отмеченного	CTRL-M
Уменьшить	CTRL-I
Увеличить	CTRL-U
Обновить	F2
Команды Browse	
State Diagram	CTRL-T
Раскрыть	CTRL-E
Дочерняя диаграмма	CTRL-P
Свойства	CTRL-B
Ссылающийся элемент	CTRL-R
Предыдущая диаграмма	F3
Переключение Object/Interaction	F5
Пакет	F7
Помощь	F1