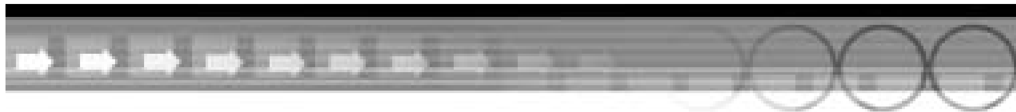

ЧАС 22

Объектно-ориентированный анализ и проектирование

На этом занятии вы узнаете:

- как анализировать проблемы и искать решения на основе объектно-ориентированного подхода;
- как на основе объектно-ориентированного подхода находить простые и надежные решения;
- о процессе анализа и реализации его результата в проекте C++;
- как обеспечить возможность расширяемости и многократного использования кода.

Цикл разработки

Теме цикла разработки посвящено множество книг и справочников. Некоторые предпочитают *каскадный* (waterfall) метод, который подразумевает, что сначала во всех деталях выясняют требования к готовому продукту, далее архитекторы определяют построение программы, используемые ею классы и т.д., а затем программисты реализуют дизайн и архитектуру. Т.е. программисту передается готовый проект и архитектура программы, а он должен лишь реализовать необходимые функциональные возможности.

Хотя каскадный метод вполне работоспособен, при разработке достаточно сложных программ его недостаточно. Во время работы программист неизбежно опирается на прежний опыт и постоянно обдумывает, что уже было написано ранее и что предстоит сделать далее. Хорошие программы на C++ способны самостоятельно продумать весь проект прежде, чем приступить к созданию его кода. Никакой гарантии того, что проект останется после этого неизменным, нет.

Прежде чем приступать к программированию, следует уяснить объем проекта. Для сложного проекта, реализация которого потребует работы множества программистов в течение нескольких месяцев, необходимо более полное и четкое формулирование архитектуры, чем для простой утилиты, написанной одним программистом за один день.

На этом занятии речь идет о проектировании больших и сложных программ, которые впоследствии будут дополняться и модифицироваться в течение многих лет. Немало программистов предпочитает работать на грани технологических возможностей и стараются писать программы на пределе сложности используемых инструментальных

средств и собственных знаний. Со временем разработчики на языке C++ находят множество способов, позволяющих расширять и усложнять программы, созданные одним или группой программистов.

Моделирование системы сигнализации

Моделирование (simulation) — это создание компьютерной модели реальной системы. Для создания модели существует множество причин, но хороший проект должен начинаться с осознания вопросов, на которые должна ответить модель.

Для начала рассмотрим следующую проблему: необходимо смоделировать систему сигнализации для дома. Дом имеет центральный зал с колоннами, четыре спальни, полуподвал и подземный гараж.

Внизу есть несколько окон: три на кухне, четыре в столовой, одно в ванной, по два в гостиной и прихожей, а также два маленьких окна рядом с входной дверью. Четыре спальни находятся наверху; каждая имеет по два окна, за исключением главной, в которой четыре окна. Наверху также есть две ванные комнаты, каждая с одним окном. И, наконец, в полуподвале и гараже находятся четыре полуокна.

Обычный вход в дом — через переднюю дверь. Кроме того, на кухне есть сдвигающаяся стеклянная дверь, а в гараже — две двери для автомобилей и одна — для входа в полуподвал. На заднем дворе также есть дверь в полуподвал.

Все окна и двери находятся под сигнализацией, на каждом телефоне и рядом с кроватями в главной спальне оборудованы "кнопки тревоги". Полуподвал тоже находится под сигнализацией, однако она настроена так, чтобы мелкие животные и птицы не могли стать причиной ее срабатывания.

При срабатывании расположенный в полуподвале центр системы сигнализации подает щебечущий звуковой сигнал, предупреждая о тревоге. Если тревога не будет отключена изнутри дома за определенный период времени, то будет вызвана полиция. Если нажата кнопка тревоги, полиция будет вызвана немедленно.

Система сигнализации связана также с детекторами огня, дыма и средствами разбрызгивания воды. Она обладает собственными средствами обеспечения бесперебойной работы, включая блок резервного питания и огнестойкий корпус.

Концептуализация

На этапе *концептуализации* (conceptualization) следует попытаться понять, что именно клиент ожидает от программы и для чего она вообще нужна? На какие вопросы должна отвечать эта модель? Например, используя моделирование, можно было бы получить ответ на такой вопрос: "Как долго датчик может оставаться неисправным, прежде чем это будет замечено?" или "Существует ли способ отключить сигнализацию окна без сообщения полиции?"

Этап концептуализации — это время, когда следует обдумать, что является для программы внутренним, а что — внешним. Следует ли моделировать действия полиции? Относится ли к системе сигнализации средство подачи звукового сигнала?

Анализ и требования

За концептуализацией следует этап анализа. При анализе необходимо помочь клиенту понять, чего именно он требует от программы, как она должна себя вести и какие виды взаимодействий отражать.

Как правило, эти требования фиксируют в наборе документов, которые могут содержать прецеденты. *Прецедент* (use case) — это подробное описание способов применения программы. Сюда относится описание взаимодействий и схемы применения, что помогает программисту лучше понять задачи проекта.

Знаете ли вы?

> **Унифицированный язык моделирования (Unified Modeling Language — UML)**
>
> Унифицированный язык моделирования — это способ представления и анализа требований. Представление информации в формате UML имеет два преимущества: во-первых, графическое представление наглядно, а во-вторых, оно является стандартным, что делает его понятным другим разработчикам. Хоть описание UML и выходит за рамки данной книги (этой теме посвящено много хороших изданий), следует заметить, что он обеспечивает способы представления прецедентов. Это особенно удобно при разработке объектно-ориентированных программ, поскольку позволяет непосредственно представить абстрактные и обычные классы.
>
> Очень хорошая книга по этой теме — *Освой самостоятельно UML за 24 часа, 3-е издание*, издательство *"Вильямс", 2005 г.*, автор *Джозеф Шмуллер* (Joseph Schmuller) (ISBN: 5-8459-0855-8).

Низкоуровневое и высокоуровневое проектирование

После выяснения, полного осознания и фиксации в соответствующей документации требований к программному продукту наступает время переходить к высокоуровневому проектированию. На этом этапе проектирования программист, ничуть не заботясь о проблемах платформы, операционной системы или языка программирования, выясняет, как именно будет работать система: каковы ее главные компоненты и как они будут взаимодействовать друг с другом.

Как вариант, можно отложить проблемы пользовательского интерфейса и сосредоточиться только на компонентах пространства проблем.

Пространство проблем (problem space) — это набор задач, которые должна решать разрабатываемая программа. *Пространство решений* (solution space) — это набор возможных решений поставленных задач.

На этапе высокоуровневого проектирования нужно обдумать обязанности объектов: что они делают и какую информацию содержат. Необходимо также обдумать способы взаимодействия объектов между собой.

В рассматриваемом примере присутствуют датчики разных типов, центральная система сигнализации, кнопки, провода и телефоны. Но здравый смысл убеждает, что в состав модели следует также включить помещения (возможно, и их полы) и, возможно, группы людей, например, жильцов и полицию.

Датчики могут быть разделены на детекторы движения, разрывные ленты, звуковые детекторы, детекторы дыма и т.д. Все это датчики разных типов, но пока их можно рассматривать как датчик вообще. Таким образом, понятие *датчик* (sensor) полностью соответствует абстрактному типу данных (Abstract Data Type — ADT).

Как абстрактный тип данных, класс `Sensor` (датчик) должен предоставлять полный интерфейс для датчиков всех типов, а каждый производный класс — обеспечивать его реализацию. Таким образом, клиенты смогут использовать разные датчики, ничуть не заботясь об их типе, и каждый из них будет решать именно свою задачу в соответствии с реальным типом.

Чтобы создать хороший класс ADT, необходимо иметь полное представление о том, что делают датчики (или как они работают). Например, являются датчики пассивными или активными устройствами? Ожидают ли они, что некий элемент нагреется или будет порвана проводящая ток лента, или расплавится контакт, или они исследуют окружающую их среду? Возможно, некоторые из датчиков имеют только два

состояния (сработал или нет), а другие, аналоговые, имеют большое количество состояний (например, текущая температура). Интерфейс абстрактного класса должен быть достаточно полным, чтобы соответствовать всем ожидаемым возможностям многочисленных производных классов.

Другие объекты

Далее следует выявить другие классы, которые будут необходимы для решения поставленных задач. Например, если предполагается сохранять записи в журнале, то, вероятно, понадобится таймер. Должна ли система записи в журнал опрашивать каждый датчик или каждый датчик должен периодически осуществлять запись в собственный файл отчета?

Пользователь должен быть способным включать, отключать и программировать систему сигнализации, поэтому понадобится некоторая разновидность пульта управления или терминала. Для этого в модели можно предусмотреть отдельный объект.

Какие классы нужны?

По мере решения этих проблем начинается разработка необходимых классов. Например, уже известно, что класс `HeatSensor` (датчик температуры) будет происходить от класса `Sensor`. Если этот датчик должен периодически осуществлять запись в отчет, то он мог бы также (при помощи множественного наследования) происходить от класса `Timer` (таймер) либо мог бы содержать таймер как переменную-член.

Класс `HeatSensor`, вероятно, будет обладать такими функциями-членами, как `CurrentTemp()` (текущая температура) и `SetTempLimit()` (установить предел температуры), а также, наверняка, унаследует от базового класса `Sensor` функцию `SoundAlarm()` (звуковой сигнал).

Одной их проблем объектно-ориентированного проектирования является инкапсуляция. Проектируемая система может обладать параметром `MaxTemp` (максимальная температура). Система сигнализации опрашивает датчик температуры, выясняет текущую температуру, сравнивает ее с максимально допустимой и подает звуковой сигнал, если она слишком велика. Можно сказать, что это нарушает принцип инкапсуляции. Возможно, было бы лучше, чтобы система сигнализации не знала и не заботилась о деталях анализа температуры, а все эти задачи взял на себя класс `HeatSensor`.

Знаете ли вы?

Дополнительные ресурсы

По этой теме есть очень хорошая книга: *The Object-Oriented Thought Process*, *Second Edition*, автор *Мэтт Вейсфилд* (Matt Weisfeld), издательство *Sams* (ISBN: 0-672-32611-6).

С аргументом, приведенным выше, можно согласиться, а можно не согласиться, но именно таким проблемам уделяют внимание на этапе анализа проблемы. Продолжая анализ, можно задаться вопросом, должны ли заниматься регистрацией только объекты датчика и класса `Log` (журнал), а объект класса `Alarm` (Тревога) — нет?

При хорошей инкапсуляции каждый класс имеет полный и достаточный набор средств, необходимых для выполнения его обязанностей, и никакие другие классы его не дублируют. Если класс `Sensor` несет ответственность за контроль текущей температуры, то никакой другой класс этой ответственности нести не должен.

С другой стороны, дополнительные классы могли бы помочь и предоставить необходимые функциональные возможности. Например, ответственность за регистрацию текущей температуры мог бы нести и класс `Sensor`, но эту задачу вполне можно делегировать классу `Log`, объект которого фактически осуществляет запись данных.

Жесткое разделение обязанностей между классами упростит саму программу, а также ее дальнейшее расширение и поддержку. Когда придет время изменить систему сигнализации, дополнив ее новыми модулями, интерфейс журнала и датчиков может остаться неизменен. Т.е. изменения в системе сигнализации не должны влиять на классы датчиков, и наоборот.

Должен ли класс `HeatSensor` иметь функцию `ReportAlarm()` (сообщение о тревоге)? Возможностью сообщать о тревоге должны обладать все датчики. Поэтому метод `ReportAlarm()` имеет смысл сделать виртуальным методом класса `Sensor`, а сам класс `Sensor` сделать абстрактным. Возможно, обобщенный метод `ReportAlarm()` класса `Sensor` придется переопределить в классе `HeatSensor`, чтобы реализовать детали, характерные только для данного типа датчиков.

Как передается сигнал тревоги?

Когда наступят условия срабатывания, датчики должны передать информацию объекту, который позвонит в полицию и сделает запись в журнале. Возможно, имеет смысл создать класс `Condition` (условие), конструктору которого будут передавать несколько параметров. В зависимости от сложности, параметры также могут быть объектами или переменными таких простых типов, как целые числа.

Возможно, объекты класса `Condition` будут переданы центральному объекту системы сигнализации, а, возможно, объекты класса `Condition` будут входить в состав объектов класса `Alarm`, которые сами способны предпринимать действия в случае тревоги. Возможно, никакого центрального объекта и не будет, может быть, датчики сами смогут создавать объекты класса `Condition`. В этом случае некоторые объекты класса `Condition` могли бы самостоятельно регистрировать информацию, а другие — вызывать полицию.

Хорошо разработанная управляемая событиями система не должна иметь центрального координатора. Все датчики должны быть независимыми объектами, получающими и посылающими сообщения друг другу, самостоятельно устанавливающими параметры и контролирующими дом. При обнаружении нарушения создается объект класса `Alarm`, который делает запись в журнале (или посылает сообщение объекту класса `Log`) и предпринимает соответствующее действие.

Цикл событий

Чтобы смоделировать такую управляемую событиями систему, разрабатываемая программа должна обладать циклом событий. Как правило, *цикл событий* (event loop) представляет собой бесконечный цикл, например, `while(1)`, который получает сообщения от операционной системы (щелчки мыши, нажатия клавиш и т.д.) и обрабатывает их один за другим, пока не возникнет условие выхода из цикла. Пример простого цикла событий приведен в листинге 22.1.

Листинг 22.1. Файл `simpleevent.cpp`. Простой цикл событий

```
 0:  // Листинг 22.1. Простой цикл событий
 1:  #include <iostream>
 2:
 3:  class Condition
 4:  {
 5:    public:
 6:      Condition() {}
 7:      virtual ~Condition() {}
 8:      virtual void Log() = 0;
 9:  };
10:
```

```
11:  class Normal : public Condition
12:  {
13:    public:
14:      Normal() { Log(); }
15:      virtual ~Normal() {}
16:      virtual void Log()
17:        { std::cout << "Logging normal conditions...\n"; }
18:  };
19:
20:  class Error : public Condition
21:  {
22:    public:
23:      Error() { Log(); }
24:      virtual ~Error() {}
25:      virtual void Log() { std::cout << "Logging error!\n"; }
26:  };
27:
28:  class Alarm : public Condition
29:  {
30:    public:
31:      Alarm ();
32:      virtual ~Alarm() {}
33:      virtual void Warn() { std::cout << "Warning!\n"; }
34:      virtual void Log() { std::cout << "General Alarm log\n"; }
35:      virtual void Call() = 0;
36:  };
37:
38:  Alarm::Alarm()
39:  {
40:    Log();
41:    Warn();
42:  }
43:
44:  class FireAlarm : public Alarm
45:  {
46:    public:
47:      FireAlarm() { Log(); };
48:      virtual ~FireAlarm() {}
49:      virtual void Call()
           { std::cout<< "Calling Fire Dept.!\n"; }
50:      virtual void Log()
           { std::cout << "Logging fire call.\n"; }
51:  };
52:
53:  int main()
54:  {
55:    int input;
56:    int okay = 1;
57:    Condition * pCondition;
58:    while (okay)
59:    {
60:      std::cout << "(0)Quit (1)Normal (2)Fire: ";
61:      std::cin >> input;
62:      okay = input;
63:      switch (input)
64:      {
65:        case 0:
66:          break;
67:        case 1:
```

```
68:          pCondition = new Normal;
69:          delete pCondition;
70:          break;
71:        case 2:
72:          pCondition = new FireAlarm;
73:          delete pCondition;
74:          break;
75:        default:
76:          pCondition = new Error;
77:          delete pCondition;
78:          okay = 0;
79:          break;
80:      }
81:    }
82:    return 0;
83:  }
```

Результат

```
(0)Quit (1)Normal (2)Fire: 1
Logging normal conditions...
(0)Quit (1)Normal (2)Fire: 2
General Alarm log
Warning!
Logging fire call.
(0)Quit (1)Normal (2)Fire: 0
```

Анализ

Расположенный в строках 58–81 простой цикл позволяет пользователю выбрать сигнал, имитирующий сообщение от датчика тревоги или пожарного датчика. В ответ на это сообщение создается объект класса `Condition`, конструктор которого вызывает различные функции-члены.

Вызов виртуальных функций-членов из конструктора может привести к непредсказуемым результатам, если нарушить порядок создания объектов. Например, когда объект класса `FireAlarm` создается в строке 72, порядок создания объектов будет таким: `Condition`, `Alarm`, `FireAlarm`. Конструктор класса `Alarm` вызывает метод `Log()`, но это метод класса `Alarm`, а не `FireAlarm`, несмотря на то что метод `Log()` объявлен виртуальным. Дело в том, что во время работы конструктора `Alarm()` никакого объекта класса `FireAlarm` еще не существует. Позже, когда объект класса `FireAlarm` уже будет создан, его конструктор вызовет метод `Log()` снова, и на сей раз это будет метод `FireAlarm::Log()`.

Проект системы PostMaster

Объектно-ориентированный анализ решает и другие проблемы. Предположим, компания *Acme Software, Inc.* решила начать новый проект и наняла группу программистов на C++ для его реализации. Босс компании хочет, чтобы группа разработала утилиту PostMaster, способную читать сообщения от нескольких разных провайдеров электронной почты. Потенциальный клиент — это деловой человек, использующий несколько систем электронной почты, таких, например, как CompuServe, America Online, Internet Mail, Lotus Notes и т.д.

Клиент должен быть способен настроить утилиту PostMaster так, чтобы она могла подключиться к каждому из провайдеров электронной почты, получить сообщения и предоставить их клиенту единообразно организованным способом, а также позволить послать ответ, переслать сообщение и т.д.

Впоследствии предполагается выпуск утилиты PostMaster версии Professional. В нее будет добавлен вспомогательный административный режим, который позволит пользователю назначить другое лицо, способное читать некоторую или всю почту клиента, обрабатывать рутинную корреспонденцию и т.д. Возможно, также потребуется компонент "искусственного интеллекта", способный сортировать почту по приоритетам на основании ключевых слов в тексте темы и содержимого.

Рассматривается возможность и других дополнений, например, способность работать не только с электронной почтой, но и группами новостей Internet, телеконференциями, списками рассылки. Вполне очевидно, что руководство компании *Acme Software, Inc.* возлагает большие надежды на программу PostMaster и горит нетерпением извлечь из ее продажи существенную коммерческую прибыль. Поэтому бюджет проекта практически неограничен (в разумных пределах), но сроки разработки крайне сжаты.

Дважды отмерь, один раз отрежь

На первом же совещании, пока еще не весь персонал нанят и оргтехника не расставлена по местам, ставится вопрос о полной спецификации на программный продукт. Исследовав рынок, маркетинговый отдел выбирает платформу (Unix, Macintosh, Windows), на которой будет работать система.

В ходе длительных и тяжелых совещаний у шефа компании становится ясно, что никакой спецификации на программный продукт нет, только общие пожелания. Поэтому принимается решение отделить пользовательскую часть (т.е. пользовательский интерфейс или UI) приложения от его внутренней структуры (системы связи и базы данных). Чтобы ускорить процесс, принимается решение разработать ее сначала для платформы Windows, а уже затем модифицировать продукт под платформы Unix и, возможно, Mac.

Это простое решение имеет для проекта серьезные последствия. Сразу становится очевидно, что понадобится одна или несколько библиотек классов для управления памятью, различных пользовательских интерфейсов, а также, возможно, для компонентов базы данных и связи.

Шеф компании абсолютно уверен, что жизнь или смерть проекта — в руках одного человека, который ясно понимает поставленные задачи, поэтому он требует, чтобы исходный анализ архитектуры и проектирование были завершены еще до того, как он наймет программистов для его реализации. Так что проблему придется анализировать.

Разделяй и властвуй

Очень быстро становится ясно, что в действительности придется решать не одну проблему, а несколько. Поэтому общую задачу следует разделить на несколько более простых.

- Связь. Возможность программного обеспечения связаться с провайдером электронной почты через модем или установить соединение по сети.
- База данных. Возможность сохранять данные на диске и обращаться к ним.
- Редактирование. Поддержка современных редакторов для создания и манипулирования сообщениями.
- Проблемы платформы. Проблемы UI, обусловленные различиями между платформами.
- Расширяемость. Планирование будущего расширения приложения.
- Организация и планирование. Организация работы разработчиков и обеспечение совместимости создаваемого ими кода. Каждая группа должна составить и оформить расписание и план работы. Руководство и отдел маркетинга должны точно знать, когда будет готов программный продукт.

Возможно, имеет смысл нанять менеджера, чтобы он взял на себя вопросы организации и планирования. Затем следует нанять старших разработчиков, чтобы они помогли провести анализ и проектирование, а затем приняли участие в реализации проекта. Эти старшие разработчики возглавят группы.

- Связь. Группа несет ответственность и за модемную связь, и за связь по сети. Им придется иметь дело с пакетами, потоками и битами, а не с сообщениями электронной почты как таковыми.
- Формат сообщения. Группа несет ответственность за преобразование сообщений от каждого провайдера электронной почты в формат, стандартный для системы PostMaster, и обратно. Эта же группа решит задачу записи сообщений на диск и их чтения при необходимости.
- Редактор сообщений. Эта группа несет ответственность за все разновидности UI для каждой платформы. Задачей группы будет также обеспечение интерфейса между внутренней структурой приложения и его пользовательским интерфейсом. Впоследствии это позволит распространить продукт на другие платформы без дублирования кода.

Формат сообщений

В первую очередь следует сосредоточиться на формате сообщений, оставив проблемы связи и пользовательского интерфейса на потом. Они будут исследованы уже после того, как появится более полное понимание того, с чем придется иметь дело. Пока не до конца ясно, какую именно информацию следует предоставлять пользователю, не стоит и беспокоиться об этом.

Исследования различных форматов электронной почты показывают, что они имеют много общего, но между ними есть и различия. Каждое сообщение электронной почты имеет отправителя, получателя и дату создания. Почти все сообщения имеют заголовок, тему и тело, которое может состоять из простого или форматированного текста, графики, а, возможно, даже звука или других дополнений. Большинство провайдеров электронной почты позволяют также передавать почтовые приложения, чтобы пользователи могли пересылать друг другу прикрепленные к письму файлы.

Как уже говорилось, ранее было принято решение преобразовывать каждое почтовое сообщение из его исходного формата в формат программы PostMaster. Это позволит не только хранить сообщения в унифицированном формате, но и существенно упростит остальные манипуляции с ними. Принято также решение отделить информацию заголовка (отправитель, получатель, дата, тема и т.д.) от тела сообщения. Зачастую пользователь просматривает только заголовки, не читая сразу содержимое всех сообщений. Возможно, пользователь захочет загрузить с хоста провайдера только заголовки сообщений, а их текст не получать вообще, однако первая версия программы PostMaster будет получать сообщения полностью, хотя отображать сможет и только заголовки.

Предварительное проектирование классов

Анализ сообщений наводит на мысль о необходимости разработать класс `Message` (сообщение). Предвидя модернизацию программы для работы с сообщениями, не относящимися к электронной почте, класс `EmailMessage` (сообщение электронной почты) имеет смысл получить как производный от абстрактного класса `Message`. Как производные от класса `EmailMessage` можно получить также такие классы, как `PostMasterMessage`, `InterchangeMessage`, `CISMessage`, `ProdigyMessage` и т.д.

Объект сообщения — это естественный выбор для программы обработки почтовых сообщений, но корректная унификация объектов в сложной системе представляет собой одну из сложнейших проблем объектно-ориентированного программирования.

В некоторых случаях, как с сообщениями, первичные классы, казалось бы, продиктованы задачами проекта. Однако чаще всего для выявления необходимых объектов приходится приложить намного больше усилий.

Но не впадайте в отчаяние. Большинство проектов сначала не совершенны. Имеет смысл обсудить проблемы вслух. Создайте список из всех существительных и глаголов, услышанных в ходе обсуждения проекта. Существительные — это хорошие кандидаты на объекты, а глаголы могли бы стать их методами (или самостоятельными специализированными объектами). Это не догма, а лишь один из опробованных на практике подходов начального этапа проектирования.

Простая часть позади. Теперь возникает вопрос: “Должен ли класс заголовка сообщения быть отделен от класса тела?”. Если да, то необходимо создать параллельные иерархии классов: `NewsGroupBody` и `NewsGroupHeader`, а также `EmailBody` и `EmailHeader`.

Параллельные иерархии — это зачастую свидетельство плохого проекта. Такая ошибка, когда набор подчиненных объектов в одной иерархии соответствует набору основных объектов в другой, является довольно распространенной при объектно-ориентированном проектировании. Дополнительные затраты на хранение и синхронизацию этих иерархий между собой очень скоро становятся чрезмерными — это классический кошмар для последующей поддержки.

Безусловно, это тоже не догма. Иногда такие параллельные иерархии являются наиболее эффективным способом решения некоторых проблем. Тем не менее, заметив смещение проекта в этом направлении, имеет смысл заново продумать проблему. Возможно существует более изящное решение.

Полученные от провайдера электронные сообщения не обязательно будут разделены на заголовок и тело, многие будут представлять собой один большой поток данных, который создаваемая программа должна будет обработать. Возможно, иерархия классов должна отражать именно эту концепцию?

Дальнейшая проработка задач приводит к необходимости составить перечень свойств сообщений, что позволит выявить их возможности, а также расположить хранимые ими данные на надлежащем уровне абстракции. Список свойств объектов — это наилучший способ определения необходимых им переменных-членов, а также выявления других объектов, которые могут понадобиться.

Почтовые сообщения должны будут сохраняться в порядке, указанном пользователем, например, по номерам телефонов и т.п. Хранение, безусловно, должно быть организовано на самом верхнем уровне иерархии. Но обязательно ли почтовые сообщения должны совместно использовать базовый класс с предпочтениями пользователя?

Корневые и некорневые иерархии

Существуют два общепринятых подхода создания иерархий наследования: можно все, или почти все, классы сделать производными от одного общего корневого базового класса либо создать несколько иерархий наследования. Преимуществом общего корневого класса является возможность избежать множественного наследования, а недостатком — вероятность переноса реализации в базовый класс.

Корневая иерархия удобна в случае, когда у всех классов набора есть общий предок.

Классы некорневых иерархий общего базового класса не имеют.

Поскольку известно, что разрабатываемый программный продукт предназначен для нескольких платформ, множественное наследование здесь не подходит, так как его поддерживают компиляторы не всех платформ. Таким образом, использование корневой иерархии и единого наследования является правильным решением. Но можно выявить и те области, где множественное наследование могло бы быть использовано в будущем. Впоследствии иерархию можно будет разделить и применить множественное наследование, если это не повредит проекту.

Для имен всех собственных внутренних классов применим префикс в виде символа p, чтобы можно было сразу выяснить, какие классы были разработаны самостоятельно, а какие принадлежат внешним библиотекам.

Корневым будет класс pObject. Каждый создаваемый класс будет происходить именно от него. Сам класс pObject останется довольно простым, он будет содержать только те данные, которые будут присутствовать абсолютно в каждом разрабатываемом классе.

Базовый класс корневой иерархии обладает, как правило, общепринятым именем (например pObject) и ограниченными возможностями. Задачей корневого класса является обеспечение возможности создания коллекций объектов любых производных от него классов так, как будто они являются экземплярами класса pObject. Недостаток корневой иерархии в том, что необходимо переносить интерфейс в корневой класс.

Следующими вероятными кандидатами в верхнюю часть иерархии являются классы pStored и pWired. Объекты класса pStored несут ответственность за сохранение данных на диске, а объекты класса pWired — за их передачу через модем или по сети. Поскольку почти все объекты придется сохранять на диске, имеет смысл разместить эти функциональные возможности довольно высоко в иерархии. Хотя все передаваемые через модем объекты подлежат сохранению, не все хранимые объекты следует передавать по сети. Таким образом, класс pWired имеет смысл сделать производным от класса pStored.

Каждый производный класс наследует все данные и функциональные возможности (методы) своего базового класса. Кроме того, он будет иметь и собственные дополнительные возможности. Таким образом, в класс pWired следует добавить набор методов, обеспечивающих передачу данных через модем.

Не исключено, что передаваемые объекты подлежат сохранению; не исключено, что все хранимые объекты можно передавать, однако не исключено и то, что ни одно из этих предположений не верно. Если сохранению подлежат только некоторые из передаваемых через модем объектов и только некоторые из хранимых объектов следует передавать, то придется использовать множественное наследование или как то обходить проблему. Потенциально в такой ситуации можно было бы, например, сделать класс pWired производным от класса pStored, а затем переделать методы сохранения так, чтобы они не делали ничего или возвращали сообщение об ошибке для тех объектов, которые передаются через модем, но не подлежат сохранению.

Таким образом, стало известно, что не все хранимые объекты подлежат передаче (например, объект предпочтений пользователя передаче не подлежит), но все объекты, передаваемые через модем, следует сохранять. На настоящий момент иерархия наследования выглядит так, как показано на рис. 22.1.

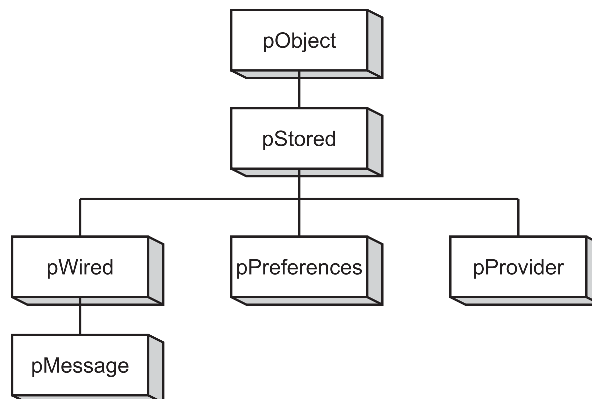

Рис. 22.1. *Предварительная схема иерархии наследования*

Купить или разработать?

Этот вопрос неизбежно возникает на этапе проектирования программы. Какие из функций имеет смысл реализовать самостоятельно, а какие можно купить. Для решения некоторых или даже всех проблем связи можно воспользоваться уже готовыми библиотеками. Существует множество коммерческих компаний (и некоммерческих источников), поставляющих подобные библиотеки.

Зачастую библиотеку дешевле купить, чем тратить время и силы на изобретение очередного колеса. Можно даже рассмотреть вопрос о покупке таких библиотек, которые не предназначены для использования именно в языке C++. Если они способны предоставить необходимые функциональные возможности, то встроить их в проект будет проще, чем разработать с нуля. Это могут быть также специальные инструментальные средства, облегчающие решение поставленной задачи.

Проектирование интерфейсов

На данном этапе проектирования очень важно избегать попыток реализации кода. Всю энергию следует сосредоточить на проектировании четкого и понятного интерфейса взаимодействия классов, а затем определить данные и методы, необходимые для каждого класса.

Прежде чем приступать к разработке остальных производных классов, имеет смысл полностью завершить список возможностей базовых классов. Поэтому на данном этапе основное внимание следует уделить классам `pObject`, `pStored` и `pWired`.

Корневой класс, `pObject`, будет иметь только те данные и методы, которые являются общими для *всех* объектов системы. Каждый объект, вероятно, должен иметь индивидуальный идентификационный номер. Таким образом, для класса `pObject` можно создать член `pID` (идентификатор), но сначала следует спросить себя, для каждого ли объекта, который даже не подлежит сохранению или передаче по сети, потребуется такой номер. Вопрос спорный, ведь даже если объект не подлежит сохранению, он остается частью иерархии.

Если же таких объектов нет, имеет смысл рассмотреть вопрос о слиянии классов `pObject` и `pStored` в один. В конце концов, если все объекты следует сохранять, то чем различаются эти классы? Обдумав ситуацию можно прийти к выводу, что некоторые классы, например объектов адресов, имеет смысл наследовать непосредственно от класса `pObject`, ведь их никогда не придется сохранять самостоятельно, поскольку они войдут в состав других объектов.

Теперь стало ясно, что наличие отдельного класса `pObject` имеет смысл. Нетрудно догадаться, что такая программа будет иметь книгу адресов, представляющую собой коллекцию объектов класса `pAddress`. Поскольку объекты класса `pAddress` никогда не будут сохраняться самостоятельно, индивидуальные идентификационные номера для каждого из них будут весьма полезны. Выяснив, что классу `pObject` необходима как минимум переменная-член `pID`, можно, наконец, предварительно определить его состав, который будет выглядеть следующим образом:

```
class pOjbect
{
  public:
    pObject();
    ~pObject();
    pID GetID() const;
    void SetID();
  private:
    pID itsID;
}
```

В этом объявлении класса есть на что обратить внимание. Во-первых, оно свидетельствует о том, что класс не происходит ни от одного другого класса, а, следовательно, является корневым. Во-вторых, здесь нет реализации его методов, даже таких, как метод `GetID()`, который, вероятно, будет иметь встраиваемую реализацию. В-третьих, уже определены постоянные методы — это часть интерфейса, а не реализация. И, наконец, здесь присутствует новый тип данных — `pID`. Для него, вероятнее всего, подойдет тип `unsigned long`, поскольку он вполне достаточен для номеров объектов.

Кроме того, если для переменной `pID` тип `unsigned long` окажется чересчур велик или мал, его можно будет централизовано изменить именно здесь, причем эта модификация повлияет абсолютно на все производные классы и в них не придется искать мест*а* применения переменной `pID`, чтобы изменить способ ее применения.

Теперь при помощи ключевого слова `typedef` объявим переменную `pID` как имеющую тип `ULONG`, который в свою очередь объявлен как тип `unsigned long`. Возникает вполне резонный вопрос: где именно расположить эти объявления?

При разработке большого проекта неизбежно совместное использование файлов. Демонстрируемый здесь стандартный подход подразумевает, что каждый класс объявлен в собственном файле заголовка, а реализация его методов расположена в соответствующем файле с расширением `.cpp`. Таким образом, будет создан файл `object.hpp` и соответствующий ему файл `object.cpp`. В проект войдут и другие файлы, например, `msg.hpp` и `msg.cpp`, содержащие объявление класса `pMessage` и реализацию его методов соответственно.

Создание прототипа

Маловероятно, что проект такой сложности, как PostMaster, сразу получится полностью законченным и работоспособным. Было бы просто наивным полагать, что столь масштабные проблемы будут безошибочно решены, а проектирование всех классов и их интерфейсов завершится еще до того, как разработчики приступят к созданию кода.

Для опробования создаваемого проекта на прототипе есть множество причин. Прототип позволит быстро проверить на практике основные идеи проекта. Существуют несколько разновидностей прототипов, однако каждый из них применяется для своих целей.

Прототип пользовательского интерфейса проекта позволяет наглядно проверить внешний вид программного продукта и его удобство для потенциальных пользователей.

Прототип функциональных возможностей не имеет законченного пользовательского интерфейса, но позволяет опробовать основные функции программы, такие, например, как передача сообщений или вложенных файлов.

И, наконец, прототип архитектуры поможет оценить на упрощенной версии программы степень улучшений при внесении изменений в проект еще до того, как он будет реализован в деталях.

Задачи прототипа должны быть предельно просты и понятны: исследование пользовательского интерфейса, эксперименты с функциональными возможностями, создание упрощенной модели финальной версии. Хороший прототип архитектуры может и не обладать совершенным пользовательским интерфейсом, и наоборот.

Кроме того, при создании прототипа будут отработаны фрагменты кода, которые впоследствии могут стать основой или элементами финального кода.

Правило 80/80

На этом этапе проектирования срабатывает эмпирическое правило, по которому 80% сотрудников способны выполнить 80% работ, поэтому основное внимание следует уделить оставшимся 20%. Вариации, конечно, возможны, но, как правило, проект осуществляется по принципу 80/80.

Следовательно, имеет смысл начинать с разработки основных классов, оставив второстепенные на потом. Когда основные классы будут определены почти полностью и потребуют лишь небольших финальных усовершенствований, можно будет выбрать один из главных классов и сосредоточиться на нем, оставив проектирование и реализацию похожих классов на потом.

Между прочим

Еще одно правило

Существует и другое правило, 80/20, которое гласит, что первые 20% кода программы потребуют 80% рабочего времени, а остальные 80% кола займут еще 80% времени!

Это вполне справедливо, что 20% работы займут 80% времени, но справедливо и то, что 80% прибыли компании принесут 20% клиентов. Это правило имеет многочисленные следствия.

Разработка класса `PostMasterMessage`

Помня о сказанном выше, сосредоточимся на классе `PostMasterMessage`. Этот класс будет под особым контролем ведущего разработчика.

Безусловно, в интерфейсе класса `PostMasterMessage` следует учесть его возможность работы с сообщениями различных типов. Поскольку предстоит взаимодействовать с разными провайдерами и получать от них сообщения в разном формате, понадобится специальный механизм, распознающий формат поступающих на компьютер сообщений.

Тем не менее, уже известно, что каждый объект класса `PostMasterMessage` будет содержать информацию об отправителе, получателе, дате и теме сообщения, а также тело сообщения и, возможно, присоединенные файлы. Следовательно, понадобятся методы доступа к каждому из этих атрибутов, а также методы, позволяющие выяснить размер присоединенных файлов, сообщений и т.д.

Некоторые службы, с которыми придется взаимодействовать, используют форматированный текст (т.е. текст, способный содержать команды для установки шрифта, размера символов и таких атрибутов, как полужирный шрифт и курсив). Другие службы подобные атрибуты не поддерживают, но могут существовать и такие, которые используют собственную систему форматирования текста. Таким образом, данный класс нуждается в методах преобразования форматированного текста в обычный текст ASCII, а, возможно, и в другой формат, принятый для программы PostMaster.

Интерфейс прикладных программ

Интерфейс прикладных программ (Application Programming Interface — API) — это набор документации и функций для использования служб. Большинство провайдеров без проблем предоставляют свои API, поэтому программа PostMaster сможет воспользоваться большинством дополнительных возможностей их сообщений (такими, например, как отформатированный текст и вложенные файлы). API программы PostMaster также имеет смысл опубликовать, чтобы в будущем другие провайдеры могли ориентироваться на работу с ними.

Класс `PostMasterMessage` должен иметь хорошо проработанный открытый интерфейс, и функции преобразования будут основным компонентом его API. Интерфейс класса `PostMasterMessage` выглядит на настоящий момент так, как показано в листинге 22.2.

Будьте осторожны!

Даже не пытайтесь!

Этот код не является определением базового класса (`MailMessage`) и не подлежит компиляции.

Листинг 22.2. Файл `postmasterinterface.cpp`. Интерфейс класса `PostMasterMessage`

```
 0:  // Листинг 22.2. Класс PostMasterMessage
 1:
 2:  class PostMasterMessage : public MailMessage
 3:  {
 4:    public:
 5:      PostMasterMessage();
 6:      PostMasterMessage(
 7:        pAddress Sender,
 8:        pAddress Recipient,
 9:        pString Subject,
10:        pDate creationDate);
11:
12:      // Здесь располагаются другие конструкторы.
13:      // Не забудьте включать конструктор копий,
14:      // конструктор для сохранения и конструктор
15:      // транспортного формата.
16:      // Включите также конструкторы для других форматов.
17:      ~PostMasterMessage();
18:      pAddress& GetSender() const;
19:      void SetSender(pAddress&);
20:      // Другие методы доступа.
21:      // Здесь будут расположены операторы, включая оператор
22:      // равенства и функции преобразования. Они должны
23:      // преобразовывать сообщения PostMaster в форматы
24:      // других сообщений.
25:
26:    private:
27:      pAddress itsSender;
28:      pAddress itsRecipient;
29:      pString  itsSubject;
30:      pDate itsCreationDate;
31:      pDate itsLastModDate;
32:      pDate itsReceiptDate;
33:      pDate itsFirstReadDate;
34:      pDate itsLastReadDate;
35:  };
```

Класс `PostMasterMessage` объявлен как производный от класса `MailMessage`. Наличие нескольких конструкторов облегчит создание экземпляров класса `PostMasterMessages` для других типов почтовых сообщений.

Понадобится набор методов доступа для чтения и установки значений данных-членов класса, а также операторы для преобразования всех или некоторых сообщений в другие форматы. Поскольку сообщения предстоит сохранять на диске и получать их по линии связи, понадобятся методы и для этих целей.

Программирование в больших группах

Даже этой предварительной архитектуры вполне достаточно, чтобы поставить задачи группам разработчиков. Над внутренней структурой той части приложения, которая относится к связи, может начинать работу группа связи тесно взаимодействуя с группой формата сообщений.

Группа формата сообщения, вероятно, займется общим интерфейсом класса `Message`, который был начат ранее и отложен в связи с вопросами чтения и записи данных на

диск. После того как взаимодействие с диском будет проработано достаточно хорошо, группа может переходить к завершению интерфейса на уровне передачи данных.

Весьма соблазнительно применить готовые редакторы сообщений, способные работать с сообщениями разнообразных форматов и внутренними классами сообщений, но это не самая лучшая идея. Группа должна разработать интерфейс для класса сообщения, ведь объекты редактора сообщений не должны знать много о внутренней структуре сообщений.

Продолжение проекта

По мере продолжения проекта будет неоднократно возникать вопрос: в какой класс необходимо поместить данный набор функциональных возможностей (или информацию)? Такую функцию должен содержать класс сообщения или класс адреса? Должен ли сохранять эту информацию редактор сообщений или само хранилище сообщений?

Объекты классов должны действовать в зависимости от обстоятельств, как секретные агенты. Они не должны совместно использовать больше информации, чем необходимо.

Решения в процессе проектирования

При разработке программы придется решать сотни проблем. Их следует расположить в порядке от более общих (в чем задача проекта?) к более определенным (как именно работает данная функция?).

Хотя детали реализации не будут завершены до тех пор, пока не окажутся воплощены в коде, а некоторые из интерфейсов продолжат перемещаться и изменяться по мере работы, необходимо удостовериться в полном понимании общей структуры и задач проекта прежде, чем приступать к созданию и компиляции кода. Одной из наиболее распространенных причин неудачи при разработке программного обеспечения является недостаточное понимание его задач.

Решения, решения и снова решения

Чтобы получить представление о задачах проекта, имеет смысл ответить на вопрос: какие пункты будет содержать меню приложения? Для программы PostMaster, вероятно, первым будет пункт New Mail Message (Новое почтовое сообщение), а это немедленно порождает следующую проблему: что случится, когда пользователь выбирает его? Должен ли сообщение создать редактор или созданный объект нового сообщения должен запустить редактор сообщений?

Задача пункта меню New Mail Message (Новое почтовое сообщение) вполне проста и очевидна — создание нового почтового сообщения. Но что случится, если пользователь щелкнет по кнопке Cancel (Отмена) уже после начала создания сообщения? Возможно, для простоты имело бы смысл сначала создать редактор и сделать так, чтобы он самостоятельно создавал новое сообщение.

Проблема такого подхода заключается в том, что редактор должен будет поразному действовать в случаях, когда сообщение создается и когда редактируется уже существующее. Если сообщение сначала создается, а затем передается редактору, то должен существовать только один набор кода, поскольку в обоих случаях редактируется уже существующее сообщение.

Если сообщение сначала создается, то кто это делает? Выполняет ли это код пункта меню? Если да, то должен ли он указывать объекту сообщения на необходимость его редактирования или это должно быть задачей конструктора объекта сообщения?

На первый взгляд это имеет смысл сделать в конструкторе, но, в конце концов, ведь не каждое созданное сообщение обязательно нужно редактировать. Так что это не самая лучшая идея. Во-первых, может понадобиться создавать "фиксированные" сообщения (например, сообщения об ошибках, отправляемые оператору системы по почте). Такие сообщения не нужно помещать в редактор. Во-вторых, и это важнее всего,

задача конструктора заключается в создании объекта: не больше, но и не меньше. После того как объект почтового сообщения будет создан, задача конструктора считается выполненной. Добавление обращения к методу редактирования только запутает конструктор и сделает объект почтового сообщения уязвимым для отказов редактора.

Однако хуже всего то, что метод передачи на редактирование обратится к другому классу — классу редактора, что приведет к вызову его конструктора. Класс редактора не является базовым для класса сообщения и не содержится внутри него. Было бы крайне неудачно, если бы создание объекта класса сообщения зависело от успеха работы конструктора класса редактора.

И, наконец, если сообщение не может быть создано успешно, то вызывать редактор не нужно вообще. Однако успех создания сообщения в этом случае зависел бы от успеха вызова редактора! Следовательно, перед вызовом метода `Message::Edit()` необходимо полностью выйти из конструктора класса сообщения.

Работа с управляющей программой

Один из подходов выявления проблем проекта подразумевает создание управляющей программы на раннем этапе процесса проектирования. *Управляющая программа* (driver program) — это пробная программа, которая предназначена только для проверки и демонстрации возможностей других функций. Например, управляющая программа для системы PostMaster могла бы предоставлять пользователю очень простое меню, позволяющее создавать объекты класса `PostMasterMessage` и управлять ими. Это позволит на практике опробовать основные элементы проекта.

Листинг 22.3 содержит несколько усовершенствованное определение класса `PostMasterMessage` и простую управляющую программу.

Листинг 22.3. Файл `driverprogram.cpp`. Управляющая программа для класса `PostMasterMessage`

```
 0:  // Листинг 22.3. Управляющая программа
 1:  #include <iostream>
 2:  #include <string.h>
 3:
 4:  typedef unsigned long pDate;
 5:  enum SERVICE { PostMaster, Interchange,
 6:                 CompuServe, Prodigy, AOL, Internet };
 7:
 8:  class String
 9:  {
10:    public:
11:      // Конструкторы
12:      String();
13:      String(const char *const);
14:      String(const String &);
15:      ~String();
16:
17:      // Перегруженные операторы
18:      char & operator[](int offset);
19:      char operator[](int offset) const;
20:      String operator+(const String&);
21:      void operator+=(const String&);
22:      String & operator= (const String &);
23:      friend std::ostream & operator<<
24:        (std::ostream& theStream, String& theString);
25:      // Общие методы доступа
26:      int GetLen() const { return itsLen; }
```

```
27:       const char * GetString() const { return itsString; }
28:       // static int ConstructorCount;
29:
30:     private:
31:       String (int);         // Закрытый конструктор
32:       char * itsString;
33:       int itsLen;
34:   };
35:
36:   // Стандартный конструктор создает строку нулевой длины
37:   String::String()
38:   {
39:     itsString = new char[1];
40:     itsString[0] = '\0';
41:     itsLen=0;
42:     // std::cout << "\tDefault string constructor\n";
43:     // ConstructorCount++;
44:   }
45:
46:   // Закрытый (вспомогательный) конструктор,
47:   // используемый только методами класса для создания
48:   // строк необходимой длины, заполненных символом null.
49:   String::String(int len)
50:   {
51:     itsString = new char[len+1];
52:     int i;
53:     for (i=0; i<=len; i++)
54:       itsString[1] = '\0';
55:     itsLen=len;
56:     // std::cout << "\tString(int) constructor\n";
57:     // ConstructorCount++;
58:   }
59:
60:   // Преобразует символьный массив в строку
61:   String::String(const char * const cString)
62:   {
63:     itsLen = strlen(cString);
64:     itsString = new char[itsLen+1];
65:     int i;
66:     for (i=0; i<itsLen; i++)
67:       itsString[i] = cString[i];
68:     itsString[itsLen]='\0';
69:     // std::cout << "\tString(char*) constructor\n";
70:     // ConstructorCount++;
71:   }
72:
73:   // Конструктор копий
74:   String::String (const String & rhs)
75:   {
76:     itsLen=rhs.GetLen();
77:     itsString = new char[itsLen+1];
78:     int i;
79:     for (i=0; i<itsLen; i++)
80:       itsString[i] = rhs[i];
81:     itsString[itsLen] = '\0';
82:     // std::cout << "\tString(String&) constructor\n";
83:     // ConstructorCount++;
84:   }
```

```
 85:
 86:  // Деструктор, освобождает выделенную память
 87:  String::~String ()
 88:  {
 89:    delete [] itsString;
 90:    itsLen = 0;
 91:    // std::cout << "\tString destructor\n";
 92:  }
 93:
 94:  String& String::operator=(const String & rhs)
 95:  {
 96:    if (this == &rhs)
 97:      return *this;
 98:    delete [] itsString;
 99:    itsLen=rhs.GetLen();
100:    itsString = new char[itsLen+1];
101:    int i;
102:    for (i=0; i<itsLen; i++)
103:      itsString[i] = rhs[i];
104:    itsString[itsLen] = '\0';
105:    return *this;
106:    // std::cout << "\tString operator=\n";
107:  }
108:
109:  // Непостоянный оператор индексирования, возвращает
110:  // ссылку на символ так, что ее можно изменить!
111:  char & String::operator[](int offset)
112:  {
113:    if (offset > itsLen)
114:      return itsString[itsLen-1];
115:    else
116:      return itsString[offset];
117:  }
118:
119:  // Постоянный оператор индексирования для использования
120:  // с постоянными объектами (см. конструктор копий)
121:  char String::operator[](int offset) const
122:  {
123:    if (offset>itsLen)
124:      return itsString[itsLen-1];
125:    else
126:      return itsString[offset];
127:  }
128:
129:  // Создает новую строку, добавляя текущую
130:  // строку к rhs
131:  String String::operator+(const String& rhs)
132:  {
133:    int  totalLen = itsLen + rhs.GetLen();
134:    String temp(totalLen);
135:    int i,j;
136:    for (i=0; i<itsLen; i++)
137:      temp[i] = itsString[i];
138:    for (j=0; j<rhs.GetLen(); j++, i++)
139:      temp[i] = rhs[j];
140:    temp[totalLen] = '\0';
141:    return temp;
142:  }
```

```
143:
144:  // Изменяет текущую строку, ничего не возвращая
145:  void String::operator+=(const String& rhs)
146:  {
147:    int rhsLen = rhs.GetLen();
148:    int totalLen = itsLen + rhsLen;
149:    String  temp(totalLen);
150:    int i,j;
151:    for (i=0; i<itsLen; i++)
152:      temp[i] = itsString[i];
153:    for (j=0; j<rhs.GetLen(); j++, i++)
154:      temp[i] = rhs[i-itsLen];
155:    temp[totalLen] = '\0';
156:    *this = temp;
157:  }
158:
159:  // int String::ConstructorCount = 0;
160:
161:  std::ostream& operator<<(
162:                            std::ostream& theStream,
163:                            String& theString)
164:  {
165:    theStream << theString.GetString();
166:    return theStream;
167:  }
168:
169:  class pAddress
170:  {
171:    public:
172:      pAddress(SERVICE theService,
173:                const String& theAddress,
174:                const String& theDisplay):
175:                itsService(theService),
176:                itsAddressString(theAddress),
177:                itsDisplayString(theDisplay)
178:      {}
179:      // pAddress(String, String);
180:      // pAddress();
181:      // pAddress (const pAddress&);
182:      ~pAddress() {}
183:      friend std::ostream& operator<<
184:        ( std::ostream& theStream, pAddress& theAddress);
185:      String& GetDisplayString()
186:        { return itsDisplayString; }
187:    private:
188:      SERVICE itsService;
189:      String itsAddressString;
190:      String itsDisplayString;
191:  };
192:
193:  std::ostream& operator<<
194:    ( std::ostream& theStream, pAddress& theAddress)
195:  {
196:    theStream << theAddress.GetDisplayString();
197:    return theStream;
198:  }
199:
200:  class PostMasterMessage
```

```
201:  {
202:    public:
203:      // PostMasterMessage();
204:
205:      PostMasterMessage(const pAddress& Sender,
206:                        const pAddress& Recipient,
207:                        const String& Subject,
208:                        const pDate& creationDate);
209:
210:      ~PostMasterMessage() {}
211:
212:      void Edit(); // invokes editor on this message
213:
214:      pAddress& GetSender()    { return itsSender; }
215:      pAddress& GetRecipient() { return itsRecipient; }
216:      String& GetSubject()     { return itsSubject; }
217:      // void SetSender(pAddress& );
218:      // Другие методы доступа
219:      // Здесь будут расположены операторы, включая оператор
220:      // равенства и функции преобразования. Они должны
221:      // преобразовывать сообщения PostMaster в форматы
222:      // других сообщений.
223:
224:    private:
225:      pAddress itsSender;
226:      pAddress itsRecipient;
227:      String  itsSubject;
228:      pDate itsCreationDate;
229:      pDate itsLastModDate;
230:      pDate itsReceiptDate;
231:      pDate itsFirstReadDate;
232:      pDate itsLastReadDate;
233:  };
234:
235:  PostMasterMessage::PostMasterMessage(
236:      const pAddress& Sender,
237:      const pAddress& Recipient,
238:      const String& Subject,
239:      const pDate& creationDate):
240:        itsSender(Sender),
241:        itsRecipient(Recipient),
242:        itsSubject(Subject),
243:        itsCreationDate(creationDate),
244:        itsLastModDate(creationDate),
245:        itsFirstReadDate(0),
246:        itsLastReadDate(0)
247:  {
248:    std::cout << "Post Master Message created. \n";
249:  }
250:
251:  void PostMasterMessage::Edit()
252:  {
253:    std::cout << "PostMasterMessage edit function called\n";
254:  }
255:
256:
257:  int main()
258:  {
```

```
259:    pAddress Sender
260:      (PostMaster, "jliberty@PostMaster", "Jesse Liberty");
261:    pAddress Recipient
262:        (PostMaster, "sliberty@PostMaster","Stacey Liberty");
263:    PostMasterMessage PostMasterMessage
264:      (Sender, Recipient, "Saying Hello", 0);
265:    std::cout << "Message review... \n";
266:    std::cout << "From:\t\t"
267:              << PostMasterMessage.GetSender() << std::endl;
268:    std::cout << "To:\t\t"
269:              << PostMasterMessage.GetRecipient() <<
std::endl;
270:    std::cout << "Subject:\t"
271:              << PostMasterMessage.GetSubject() << std::endl;
272:    return 0;
273:  }
```

Результат

```
Post Master Message created.
Message review...
From:      Jesse Liberty
To:        Stacey Liberty
Subject:   Saying Hello
```

Анализ

В строке 4 тип pDate определен как соответствующий типу unsigned long. Сохранение даты в переменной типа unsigned long является вполне обычным явлением. Как правило, оно содержит количество секунд, прошедших с такой произвольно назначенной даты, как 1 января 1900 года. В данной программе это лишь место для реального значения, которое будет определено впоследствии при завершении класса pDate.

В строке 5 определена перечисляемая константа SERVICE, позволяющая объектам адресов отслеживать типы адресов, включая PostMaster, CompuServe и т.д.

Строки 8–167 содержат интерфейс и реализацию класса String. Они аналогичны приведенным в предыдущих главах. Класс String применяется для нескольких переменных-членов всех классов сообщений, некоторых других классов, используемых классом сообщений, а также основной программой. Для завершения классов сообщений понадобится надежный полнофункциональный класс String.

В строках 169–191 объявлен класс pAddress. Здесь реализованы лишь основные функциональные возможности этого класса, которые будут дополнены впоследствии, по мере усовершенствования программы. Эти объекты представляют неотъемлемые компоненты каждого сообщения — адрес отправителя и адрес получателя. Объект полнофункционального класса pAddress будет способен переадресовать сообщение, ответить на сообщение и т.д.

В задачи объекта класса pAddress входит отслеживание отображаемой строки адреса и внутренней строки маршрутизации для ее службы. В проекте остается один открытый вопрос: должен ли существовать только один класс pAddress или для каждой службы следует создать собственный, производный от него? В настоящий момент служба отслеживает значение перечисляемой константы, хранимое в переменной-члене каждого объекта класса pAddress.

Строки 200–233 содержат интерфейс класса PostMasterMessage. В данном конкретном листинге этот класс является самостоятельным, но очень скоро его придется сделать частью иерархии наследования. При переделке, когда он окажется унаследован от

класса `Message`, некоторые из переменных-членов могут переместиться в базовые классы, а некоторые из функций-членов будут переопределять методы базового класса.

Чтобы сделать этот класс полнофункциональным, понадобится набор разных конструкторов, функций доступа и других функций-членов. Обратите внимание, этот листинг доказывает, что на момент создания простой управляющей программы, позволяющей проверить некоторые из сделанных предположений, рассматриваемый класс не обязан быть законченным на все 100%.

В строках 251–254 содержится функция `Edit()`. Она существенно упрощена и лишь отображает на экране сообщение. Ее функциональные возможности, обеспечивающие редактирование сообщения, будут реализованы позже, уже после того, как рассматриваемый класс станет полнофункциональным.

Строки 257–273 содержат управляющую программу. В настоящее время она проверяет лишь несколько функций доступа и перегруженный оператор `operator<<`. Тем не менее, это наглядно демонстрирует способы экспериментирования с классом `PostMasterMessage` и средой разработки, позволяющей изменять данные классы и исследовать влияние сделанных изменений.

Вопросы и ответы

- **Чем объектно-ориентированный анализ и проектирование отличаются от других подходов?**

  До появления объектно-ориентированных технологий аналитики и программисты рассматривали программы как группы функций для обработки данных. Объектно-ориентированное программирование объединило данные и функции в самостоятельные блоки, способные хранить и обрабатывать данные. Процедурное программирование в основном сосредоточено на функциях и их работе с данными. Как уже говорилось, программы на языках Pascal и C являются наборами процедур, а программы на языке C++ — набором классов.

- **Является ли объектно-ориентированное программирование той палочкой-выручалочкой, которая решит все проблемы программирования?**

  Нет, это и не предполагалось. Но для больших и сложных проектов объектно-ориентированный анализ, проектирование и программирование могут стать единственно возможными инструментальными средствами, способными справиться со сложной проблемой такими способами, которые ранее были недоступны.

Коллоквиум

Изучив возможности объектно-ориентированного анализа и проектирования, имеет смысл ответить на несколько вопросов и выполнить ряд упражнений, чтобы закрепить полученные знания.

Контрольные вопросы

1. Что необходимо сделать прежде, чем приступать к созданию кода?
2. Что такое интерфейс прикладных программ (API)?
3. Что делает управляющая программа?
4. В чем преимущество разделения больших приложений на меньшие фрагменты?

Упражнения

Не забывайте: решения находятся на прилагаемом CD.

1. Выберите небольшую проблему из повседневной или деловой жизни и выполните описанные на этом занятии действия для ее решения.
2. Измените файл `simpleevent.cpp` (листинг 22.1) и добавьте в каждый деструктор оператор `std::cout`, выводящий на экран сообщение. Запустите программу на выполнение и проследите, когда происходит вызов деструкторов. Как уже было продемонстрировано, вызов конструкторов произойдет при передаче подлежащих регистрации сообщений.
3. Исследуйте доступные по Internet классы и библиотеки C++. Рассмотрите их и решите, следует ли при случае их купить или пытаться реализовать самостоятельно.

Ответы на контрольные вопросы

1. Перед началом создания кода необходимо решить множество задач: выявить требования к программному продукту (что именно хочет пользователь?), провести анализ (что необходимо для выполнения этих требований?) и разработать проект (как выполнить требования проекта?). К сожалению, очень много компаний исповедуют совершенно неправильную философию: "Пусть все эти программисты садятся и сразу принимаются писать код! Не моя забота, что список требований не завершен!"
2. API — это набор документации и методов, необходимых для использования службы. Он не содержит информацию о работе службы, только о том, как ее использовать. Подробности реализации от клиента скрыты.
3. Управляющая программа позволяет проверять работу классов и функций по мере их разработки. Она может быть довольно простой, но вполне позволяющей выяснить поведение кода и взаимодействие компонентов.
4. Одним из самых больших преимуществ является возможность выполнения работ несколькими разработчиками или даже группами. В то время как одна группа работает над первым классом, никто не мешает второй группе работать над вторым.