ГЛАВА

2

ОБЪЕКТНАЯ МОДЕЛЬ

Объектно-ориентированная технология покоится на солидной технической основе, элементы которой образуют так называемую *объектную модель проектирования* (object model of development), или просто *объектную модель*. К основным принципам этой модели относятся абстракция, инкапсуляция, модульность, иерархия, контроль типов, параллелизм и персистентность. Сами по себе эти принципы не новы, но в рамках объектной модели они существуют в синергическом единстве.

Не подлежит никакому сомнению, что объектно-ориентированный анализ и проектирование в корне отличаются от традиционных методов структурного проектирования: они требуют совершенно иного подхода к декомпозиции и порождают архитектуру программного обеспечения, намного превосходящую возможности структурного программирования.

2.1 Эволюция объектной модели

Объектно-ориентированная технология не выросла спонтанно на развалинах бесчисленных проектов, потерпевших крах из-за устаревшей технологии проектирования, и не является радикальным пересмотром предшествующих подходов. На самом деле она вобрала в себя наилучшие идеи более ранних методологий. В главе описывается эволюция средств проектирования. Это поможет читателям понять основы и причины появления объектно-ориентированной технологии.

Оглядываясь на относительно недолгую, но яркую историю программирования, необходимо отметить две главные тенденции.

1. Перенос акцентов с детализированного на крупномасштабное программирование.

2. Эволюция языков программирования высокого уровня.

Большинство современных промышленных систем программного обеспечения намного масштабнее и сложнее своих предшественников, созданных всего несколько лет тому назад. Возрастание сложности стимулировало многочисленные прикладные исследования по методологии проектирования программного обеспечения, особенно в области декомпозиции, абстракции и иерархии. Это способствовало также появлению более выразительных языков программирования.

Поколения языков программирования

Вегнер (Wegner) [1] выделил четыре поколения наиболее популярных языков программирования высокого уровня, положив в основу классификации новаторские языковые конструкции, впервые появившиеся в этих языках. (Разумеется, это далеко не исчерпывающий список всех языков программирования.)

- Языки программирования первого поколения (1954–1958)

FORTRAN I	Математические формулы
ALGOL-58	Математические формулы
Flowmatic	Математические формулы
IPL V	Математические формулы

- Языки программирования второго поколения (1959–1961)

FORTRAN II	Подпрограммы, раздельная компиляция
ALGOL-60	Блочная структура, типы данных
COBOL	Описание данных, работа с файлами
Lisp	Обработка списков, указатели, сборка мусора

- Языки программирования первого поколения (1962–1970)

PL/I	FORTRAN+ALGOL+COBOL
ALGOL-68	Ближайший наследник ALGOL-60
Pascal	Простой наследник ALGOL-60
Simula	Классы, абстракция данных

- Разрыв преемственности (1970–1980)

  В этот период было изобретено много языков, но лишь немногие из них выдержали испытание временем. Среди них заслуживают упоминания следующие языки.

C	Эффективен; характеризуется малым размером исполняемых модулей
FORTRAN 77	Прошел стандартизацию ANSI

Продолжим классификацию Вегнера.

- Бум объектно-ориентированного программирования (1980–1990, проверку временем прошли лишь несколько языков)

Smalltalk 90	Чисто объектно-ориентированный язык
C++	Происходит от языков C и Simula
Ada83	Строгий контроль типов; сильное влияние языка Pascal
Eiffel	Происходит от языков Ada и Simula

- Появление интегрированных сред (1990 и до настоящего времени)

Visual Basic	Облегченное проектирование графического пользовательского интерфейса (graphical user interface — GUI) для приложений в операционной среде Windows
Java	Наследник языка Oak; разрабатывался для мобильных устройств
Python	Объектно-ориентированный язык сценариев
J2EE	Интегрированная среда на базе языка Java для промышленного применения
.NET	Интегрированная объектно-ориентированная среда, разработанная компанией Microsoft
Visual C#	Конкурент языка Java для среды Microsoft .NET
Visual Basic .NET	Вариант языка Visual Basic для среды Microsoft .NET

В каждом следующем поколении языков программирования механизмы абстракции изменялись. Языки первого поколения использовались, в основном, для научных и технических вычислений, и словарь этой предметной области был математическим. Такие языки, как FORTRAN I, позволяли программистам записывать математические формулы, освобождая их от сложностей, связанных с ассемблером или машинным кодом. Таким образом, первое поколение языков высокого уровня было шагом в направлении предметной области и отступало от технических деталей компьютеров.

Языки программирования второго поколения перенесли акцент на алгоритмические абстракции. В то время мощность компьютеров все больше увеличивалась, а область их применения расширялась, особенно в экономической сфере. Основная задача заключалась в инструктировании машины: например, сначала прочитать анкеты сотрудников, затем упорядочить и после этого распечатать результаты на принтере. Языки программирования этого поколения продолжали отдаляться от компьютеров и приближаться к предметной области.

В конце 1960-х годов, особенно после изобретения транзисторов и технологии интегральных схем, стоимость аппаратного обеспечения резко упала, а производительность компьютеров продолжала расти почти экспоненциально. Это позволяло

решать еще более сложные задачи, но вынуждало программистов манипулировать более разнообразными видами данных. В результате возникли языки третьего поколения, такие как ALGOL-68 и Pascal, поддерживающие абстракцию данных. Теперь разработчики получили возможность описывать свои собственные виды данных (т.е. создавать пользовательские типы) и реализовывать проектные решения на языке программирования. Это поколение языков программирования еще больше удалилось от машинной архитектуры и приблизилось к предметной области.

1970-е годы ознаменовались бурным всплеском активности в области разработки языков программирования. В результате было создано несколько тысяч разных языков программирования и их вариантов. Со временем необходимость разрабатывать все более крупные программы сделала очевидной неадекватность старых языков. Многие механизмы новых языков программирования разрабатывались именно для того, чтобы преодолеть эти ограничения. Лишь немногие из этих языков прошли проверку временем (вы когда-нибудь видели современный учебник по языкам Fred, Chaos или Tranquil?). Однако многие из концепций, воплощенных в этих языках, были внедрены в новых версиях более ранних языков.

Наибольший интерес представляет класс языков, называемых *объектными* (object-based) и *объектно-ориентированными* (object-oriented). Эти языки лучше всего поддерживают объектно-ориентированную декомпозицию программного обеспечения. Большинство объектно-новых языков этой категории (и объектно-ориентированных вариантов старых языков программирования) появились в 1980-х и начале 1990-х годов. После 1990-го года появилось немного объектно-ориентированных языков программирования, поддерживаемых поставщиками коммерческого программного обеспечения (например, языки Java и C++). Появление интегрированных сред (например, J2EE и .NET), предоставляющих в распоряжение программистов огромное количество компонентов и сервисов, которые упростили решение типичных и рутинных задач, резко увеличило производительность их труда и продемонстрировало преимущества повторного использования кода.

Топология языков первого поколения и ранних языков второго поколения

Рассмотрим структуру каждого поколения языков программирования. На рис. 2.1 продемонстрирована топология, характерная для большинства языков первого поколения и ранних языков второго поколения. Под *топологией* (topology) подразумеваются основные конструктивные блоки языка программирования и способы их взаимосвязи. Как показывает анализ рис. 2.1, в таких языках, как FORTRAN и COBOL, основным конструктивным блоком является подпрограмма (или параграф, если использовать терминологию языка COBOL).

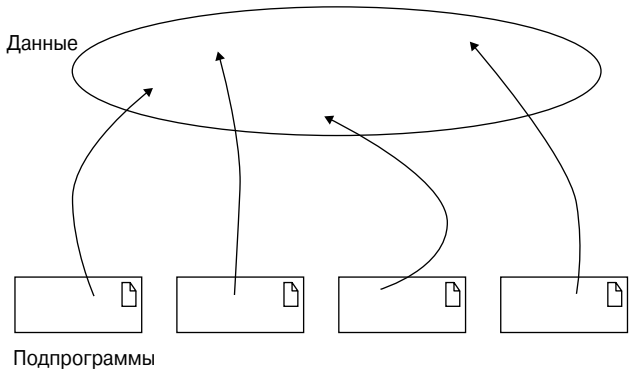


Рис. 2.1. Топология языков программирования первого поколения и первых языков второго поколения

Программы, написанные на этих языках, имеют относительно простую структуру, состоящую только из глобальных данных и подпрограмм. Стрелки на рис. 2.1 иллюстрируют зависимость подпрограмм от данных. В процессе проектирования обычно удается провести логическое разделение между разнотипными данными, но ранние языки программирования очень слабо поддерживают такое проектное решение. Ошибка, допущенная в одной части программы, может оказать разрушительное влияние на остальную часть системы, так как глобальные структуры данных открыты для всех подпрограмм. При внесении изменений в крупную систему эти языки не гарантируют поддержку целостности данных. Даже после короткого периода эксплуатации программа, написанная на ранних языках, содержит большое количество перекрестных связей между подпрограммами, неявно подразумеваемых данных и запутанных потоков управления. Все это снижает надежность системы и уменьшает ясность проектного решения.

Топология языков программирования позднего второго и раннего третьего поколения

К середине 1960-х годов программисты осознали роль подпрограмм как важного промежуточного звена, обеспечивающего связь между задачей и компьютером [2]. “Первая программная абстракция, которую теперь называют процедурной, является непосредственным следствием прагматического взгляда на программное обеспечение... Подпрограммы появились еще до 1950 года, но в то время они не воспринимались как абстракции... Их сначала рассматривали скорее как средства, позволяющие экономить силы... Однако очень скоро подпрограммы стали восприниматься как абстрактные функции” [3].

Понимание того, что подпрограммы являются механизмом абстрагирования, привело к трем важным последствиям. Во-первых, были разработаны языки, под-

держивавшие разнообразные механизмы передачи параметров. Во-вторых, были заложены основы структурного программирования, проявившиеся в поддержке вложенных подпрограмм и разработке теории управляющих конструкций и областей видимости. В-третьих, были изобретены методы структурного проектирования, позволившие разработчикам создавать крупные системы, используя подпрограммы в качестве конструктивных блоков. Нет ничего удивительно в том, что, как показано на рис. 2.2, топология языков программирования позднего второго и раннего третьего поколений оказалась вариантом топологии их предшественников. Эта топология устранила некоторые недостатки более ранних языков, в частности, усилила контроль над алгоритмическими абстракциями, но по-прежнему не позволяла создавать крупномасштабные системы.

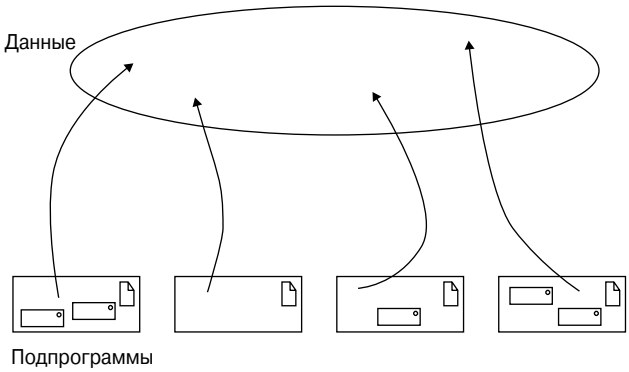


Рис. 2.2. Топология языков программирования позднего второго и раннего третьего поколений

Топология поздних языков третьего поколения

В языке FORTRAN II и других поздних языках программирования третьего поколения для создания крупномасштабных систем был предложен новый важный механизм структуризации. Укрупнение программных проектов привело к разрастанию коллективов программистов и породило необходимость независимо разрабатывать отдельные части одной и той же программы. Для решения этой проблемы была предложена концепция отдельно компилируемого модуля, который на первых порах интерпретировался как произвольный контейнер, содержащий набор данных и подпрограмм (см. рис. 2.3). На практике такие модули не рассматривались как механизмы абстрагирования. Их использовали просто как наборы подпрограмм, которые чаще всего модифицировались одновременно.

Большинство языков этого поколения поддерживало определенный вид модульной структуры, но практически не регламентировало семантическое согласование между интерфейсами модулей. Программист, сочиняющий подпрограмму

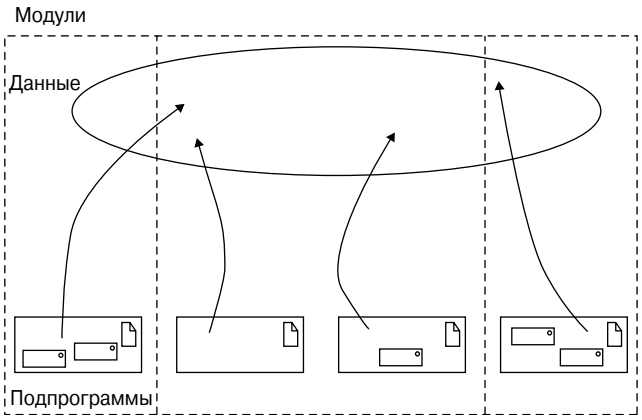


Рис. 2.3. Топология языков программирования позднего третьего поколения

для одного из модулей, мог, например, предполагать, что она будет вызываться с тремя параметрами: действительным числом, массивом, состоящим из десяти элементов, и целым числом, представляющим собой булеву величину. В то же время в другом модуле эта подпрограмма могла по ошибке вызываться с фактическими параметрами, противоречащими этому предположению, например, с целым числом, массивом, состоящим из пяти элементов, и отрицательного числа. Аналогично, автор одного из модулей мог предполагать, что определенный блок общих данных может изменяться только внутри данного модуля, а программист, создающий другой модуль, мог нарушить это предположение, непосредственно обращаясь к этим данным. К сожалению, поскольку большинство языков очень слабо поддерживало абстракцию данных и строгий контроль типов, такие ошибки можно было выявить только во время выполнения программы.

Топология объектных и объектно-ориентированных языков

Абстракция данных играет важную роль в борьбе со сложностью. “Природа абстракций, обеспечиваемых процедурами, хорошо подходит для описания абстрактных операций, но не подходит для описания абстрактных объектов. Это серьезный недостаток, поскольку во многих приложениях основную сложность составляет необходимость манипулирования именно объектами” [4]. Осознание этого факта привело к двум важным последствиям. Во-первых, возникли методы информационного проектирования (data-driven design), систематизировавшие способы абстракции в алгоритмических языках. Во-вторых, стала разрабатываться теория типов данных, в результате воплотившаяся в таких языках, как Pascal.

Естественным следствием этих идей стало появление языков Simula, Smalltalk, Object Pascal, C++, Ada, Eiffel и Java. По причинам, которые будут рассмотрены далее, эти языки называются *объектными* или *объектно-ориентированными*. Топология этих языков применительно к малым и средним по размеру приложениям представлена на рис. 2.4.

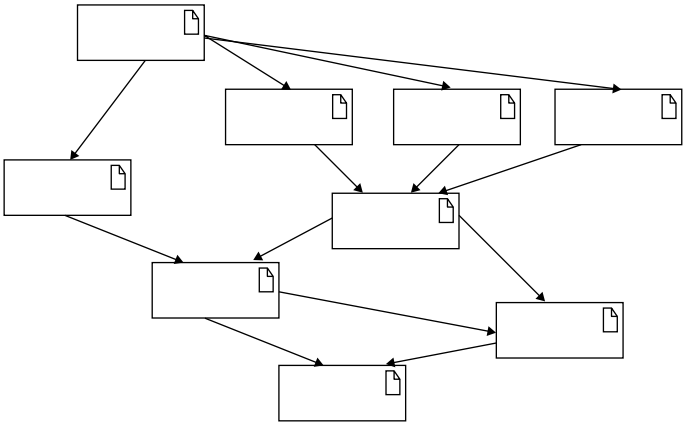


Рис. 2.4. Топология малых и средних языков программирования объектных и объектно-ориентированных языков программирования

Основным конструктивным элементом в этих языках является *модуль* (module), представляющий собой логически связанную совокупность классов и объектов, а не подпрограмм, как в более ранних языках. Иначе говоря, “если процедуры и функции — это глаголы, а данные — это существительные, то в основе процедурно-ориентированных программ лежат глаголы, а в основе объектно-ориентированных — существительные” [6]. По этой причине структура малых и средних по размеру объектно-ориентированных приложений изображается в виде графа, а не дерева, характерного для алгоритмических языков. Кроме того, в этих языках почти не используются или вообще исключены глобальные данные. Данные и операции объединяются таким образом, что основными логическими конструктивными элементами объектно-ориентированных систем теперь являются классы и объекты, а не алгоритмы.

В настоящее время наблюдается большой прогресс в программировании больших и даже огромных систем. Оказалось, что для очень сложных систем классы, объекты и модули являются необходимыми, но все же не достаточными средствами абстракции. К счастью, объектная модель допускает масштабирование. В крупных системах существуют кластеры абстракций, образующие слои. На каждом уровне абстракции можно выделить совокупности объектов, взаимодействующих друг с другом для реализации поведения, относящегося к более высокому уров-

ню абстракции. Если поинтересоваться устройством таких кластеров и заглянуть внутрь, можно обнаружить другой набор взаимодействующих абстракций. Именно эта организация сложности описана в главе 1. Соответствующая топология изображена на рис. 2.5.

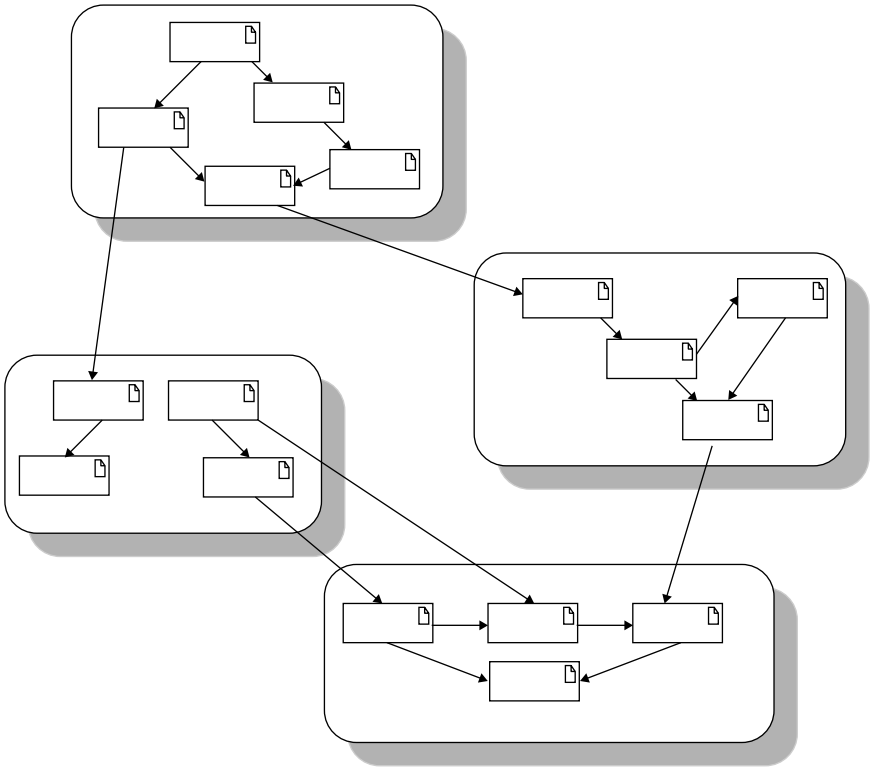


Рис. 2.5. Топология крупных приложений, созданных с помощью объектных и объектно-ориентированных языков программирования

2.2 Основные положения объектной модели

Методы структурного проектирования разрабатывались для того, чтобы помочь проектировщикам, создающим сложные системы из алгоритмов. Аналогично, методы объектно-ориентированного проектирования создавались, чтобы дать разработчикам мощные выразительные средства языков объектного и объектно-ориентированного программирования, в которых основными конструктивными элементами являются классы и объекты.

На развитие объектной модели влияло большое количество факторов, а не только объектно-ориентированное программирование. Как показано во врезке

“Основы объектной модели”, объектный подход признан объединяющей концепцией компьютерной науки, которую можно применять не только в языках программирования, но также при разработке пользовательских интерфейсов, баз данных и даже компьютерной архитектуры. Причина такой универсальности заключается в том, что объектная ориентация позволяет справиться со сложностью, характерной для самых разных систем.

Таким образом, объектно-ориентированный анализ и проектирование носят эволюционный, а не революционный характер. Они не отбрасывают достижений прежних методологий, а берут из них лучшее. К сожалению, большинство программистов плохо владеет методами объектно-ориентированного анализа и проектирования. Разумеется, многие инженеры успешно применяют методы структурного проектирования. Однако существуют пределы сложности, с которой можно справиться с помощью алгоритмической декомпозиции. Более сложные задачи можно решить только на основе объектно-ориентированной декомпозиции. Кроме того, пытаясь использовать такие языки, как C++ или Ada исключительно в качестве алгоритмических, программисты не только упускают большие возможности, но и получают менее эффективные программы по сравнению с программами, написанными на языках C или Pascal. Если дать мощную электродрель плотнику, который не знает о существовании электричества, то скорее всего он станет использовать ее в качестве молотка. В результате он согнет несколько гвоздей и разобьет пальцы, потому что электродрель — плохой молоток.

К сожалению, поскольку объектная модель является наследником нескольких концепций, ее терминология довольно запутанна. Например, для обозначения одного и того же понятия в языке Smalltalk используется термин *метод* (method), в языке C++ — *виртуальная функция-член* (virtual function-member), а в языке CLOS — *обобщенная функция* (generic function). Программисты, работающие на языке Object Pascal, C# и Java говорят о *приведении типов* (type coersion, or cast), а на языке Ada — о *преобразовании типов* (type conversion). Для того чтобы прояснить ситуацию, необходимо уточнить, что считается объектно-ориентированным, а что — нет.

Термин *объектно-ориентированный* используется так же легкомысленно и так же почтительно, как “материнство”, “яблочный пирог” и “структурное программирование” [6]. Разумеется, можно согласиться с тем, что понятие объекта является центральным понятием в любой объектно-ориентированной теории. В предыдущей главе объектом называлась материальная сущность с четко определенным поведением. Стефик (Stefik) и Боброу (Bobrow) дали следующее определение объектов: “сущности, обладающие свойствами процедур и данных, поскольку они производят вычисления и сохраняют свое локальное состояние” [7]. Определение объекта как сущности в некоторой мере является спорным, но основная идея заключается в сочетании алгоритмических идей и абстракции данных. Джонс (Jones) уточнил это понятие следующим образом: “В объектной модели акцент пе-

реносится на четкое описание компонентов физической или абстрактной системы, моделируемой с помощью программы... Объекты имеют некую “целостность”, которая не должна — а, в действительности, не может — быть нарушена. Объект может только изменять состояние, выполнять действия, допускать манипуляции и взаимодействовать с другими объектами приемлемым способом. Иначе говоря, существуют инвариантные свойства, характеризующие объект и его поведение. Например, к числу инвариантных свойств лифта относится его способность передвигаться по шахте вверх и вниз... Любая модель лифта должна учитывать эти инварианты, поскольку они являются его неотъемлемым атрибутом” [26].

Объектно-ориентированное программирование

Что же такое объектно-ориентированное программирование (object-oriented programming — OOP)? Наше определение выглядит следующим образом.

> Объектно-ориентированное программирование — это метод программирования, основанный на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определенного класса, а классы являются членами определенной иерархии наследования.

Необходимо обратить внимание на следующие важные части этого определения: 1) объектно-ориентированное программирование использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы (иерархия “целое/часть” определена в главе 1); 2) каждый объект является экземпляром (instance) определенного класса (class); 3) классы образуют иерархии (см. понятие об иерархии “общее/частное” в главе 1). Программа считается объектно-ориентированной, только если выполнены все три указанных требования. В частности, программирование, не использующее наследование, называется не объектно-ориентированным, а программированием с помощью абстрактных типов данных (programming with abstract data types).

В соответствии с этим определением одни языки программирования являются объектно-ориентированными, а другие — нет. Страуструп (Stroustrup) полагает: “Если термин “объектно-ориентированный язык” вообще имеет смысл, то он должен относиться к языку, хорошо поддерживающему объектно-ориентированный стиль программирования... Поддержка такого стиля программирования считается хорошей, если средства языка обеспечивают удобное использование этого стиля. Язык не поддерживает объектно-ориентированное программирование, если написание программ в этом стиле требует особых усилий или опыта; в этом случае говорят, что язык просто позволяет программистам использовать объектно-ориентированный подход” [27]. С теоретической точки зрения существует возможность имитации объектно-ориентированного программирования с помощью

Основы объектной модели

Йонезава (Yonezawa) и Токоро (Tokoro) указывают: “Термин “объект” почти одновременно возник в начале 70-х годов в разных независимых друг от друга областях компьютерных наук. Этот термин означал разные, хотя и взаимосвязанные понятия, изобретенные для того, чтобы уменьшить сложность систем программного обеспечения. Объекты использовались для представления компонентов системы, допускающей разложение на модули или фрагменты представляемых знании” [8]. Леви (Levy) отметил, что на эволюцию объектно-ориентированных концепций оказали влияние следующие события.

- Усовершенствование архитектуры компьютеров, включая повышение мощности систем и аппаратной поддержки операционных систем.
- Развитие языков программирования, таких как Simula, Smalltalk, CLU и Ada.
- Достижения методологии программирования, включая принципы модульности и сокрытия данных [10].

К этому еще следует добавить три момента, оказавшие влияние на становление объектного подхода:

- развитие теории баз данных;
- исследования в области искусственного интеллекта;
- достижения философии и теории познания.

Концепция объекта возникла более 20 лет назад при конструировании компьютеров с дескрипторной (descriptor-based) и мандатной (capability-baseds) архитектурой [10]. Эти разновидности архитектуры отличались от классической архитектуры фон Неймана и пытались преодолеть разрыв между высоким уровнем абстракций, присущих для языков программирования, и низким уровнем абстракций, характерных для компьютеров [11]. По мнению сторонников этих подходов преимущества новых разновидностей архитектуры позволяли лучше выявлять ошибки, повышали эффективность выполнения программ, сокращали количество инструкций, упрощали компиляцию и снижали объем требуемой памяти. Компьютеры тоже могут иметь объектно-ориентированную архитектуру.

С объектно-ориентированной архитектурой тесно связаны объектно-ориентированные операционные системы. Работая над мультипрограммной системой THE, Дейкстра (Dijkstra) впервые ввел понятие строительных систем (building systems) как машин с многоуровневыми состояниями [12]. К первым объектно-ориентированным операционным системам относятся Plessey/System 250 (для мультипроцессора Plessey 250), Hydra (для компьютера C.mmp, разработанного компанией CMU), CALTSS (для компьютера CDC 6400), CAP (для компьютера Cambridge CAP), UCLA Secure UNIX (для компьютеров PDP 11/45 и 11/70), StarOS (для компьютера Cm* компании CMU), Medusa (также для компьютера Cm*) и iMAX (для компьютера Intel 432) [13].

Вероятно, наиболее значительный вклад в объектную модель внесен объектными и объектно-ориентированными языками программирования. Впервые понятия

классов и объектов появились в языке Simula 67. Система Flex и последовавшие за ней диалекты Smalltalk-72, -74, -76 и, наконец, последняя версия Smalltalk-80, приняв парадигму языка Simula, довели ее до логического завершения, реализовав все возможные сущности в виде экземпляров классов. В 1970-х годах был разработан ряд языков — Alphard, CLU, Euclid, Gypsy, Mesa и Modula, — поддерживающих идею абстракции данных. Последующие исследования привели к внедрению идей языков Simula и Smalltalk в традиционные языки высокого уровня. Объединение объектно-ориентированного подхода с концепциями языка C привело к появлению языков C++ и Objective C. Впоследствии для предотвращения наиболее распространенных ошибок, характерных для программ на языке C++, был разработан язык Java. Внедрение концепций объектно-ориентированного программирования в язык Pascal породило языки Object Pascal, Eiffel и Ada. Кроме того, появились диалекты языка LISP, инкорпорировавшие свойства языков Simula и Smalltalk. Более подробно особенности этих языков описаны в приложении 1.

Первым, кто указал на важность построения систем в виде многоуровневых абстракций, был Дейкстра. Позднее Парнас (Parnas) ввел идею сокрытия информации [14]. В 1970-х годах ряд исследователей, главным образом Лисков (Liskov) и Жиль (Zilles) [15], Гуттаг (Guttag) [16], и Шоу (Shaw) [17], разработали механизмы абстрактных типов данных. Хоар (Hoare) дополнил эти подходы теорией типов и подклассов [18].

Несмотря на то что технологии построения баз данных развивались независимо от языков программирования, они также внесли свой вклад в разработку объектного подхода [19], в основном с помощью идей подхода “сущность-отношение” (entity-relationship — ER) [20] к моделированию данных. В модели ER, предложенной Ченом (Chen) [21], моделирование происходит в терминах сущностей, их атрибутов и взаимоотношений.

Свой вклад в развитие объектно-ориентированных абстракций внесли также исследователи в области искусственного интеллекта, разрабатывавшие методы представления знаний. В 1975 году Минский (Minsky) впервые выдвинул теорию фреймов (theory of frames) для представления реальных объектов в системах распознавания образов и естественных языков [22]. С тех пор фреймы стали использоваться в качестве архитектурной основы в различных интеллектуальных системах.

В заключение отметим, что объектная модель стала предметом исследования философов и специалистов по когнитологии. Идея о том, что мир можно рассматривать как совокупность объектов или процессов, была выдвинута еще в Древней Греции. В XVII веке Декарт высказал мысль, что объектно-ориентированный взгляд на мир является вполне естественным [23]. В XX веке эту тему развивала Рэнд (Rand) в своей философии объективистской эпистемологии [24]. Позднее Минский (Minsky) предложил модель человеческого мышления, в которой разум человека рассматривается как совокупность различно мыслящих агентов [25]. Он утверждает, что только взаимодействие таких агентов порождает то, что мы называем *интеллектом*.

обычных языков, таких как Pascal и даже COBOL или ассемблер, но это чрезвычайно трудно. Карделли (Cardelli) и Вегнер (Wegner) утверждают:

> "[A] Язык программирования является объектно-ориентированным тогда и только тогда, когда он удовлетворяет следующие условия.
>
> - Он поддерживает объекты, представляющие собой абстракции данных с интерфейсом в виде именованных операций и сокрытым локальным состоянием.
> - Объекты имеют ассоциированный с ними тип [класс].
> - Типы [классы] могут наследовать атрибуты супертипов [суперклассов]". [28]

Поддержка наследования в объектно-ориентированных языках означает возможность выражения отношения "is a" среди типов (например, красная роза — это цветок, а цветок — это растение). Если язык не поддерживает механизм наследования, то его нельзя считать объектно-ориентированным. Карделли и Вегнер предлагают называть такие языки *объектными* (object-based), а не *объектно-ориентированными* (object-oriented). Согласно этому определению языки Smalltalk, Object Pascal, C++, Eiffel, CLOS, C# и Java являются объектно-ориентированными, а Ada83 — объектным (позднее в язык Ada95 были добавлены объектно-ориентированные языки). Однако, поскольку объекты и классы являются элементами обеих групп языков, очень желательно и вполне возможно использовать объектно-ориентированные методы, работая на языках программирования как первой, так и второй группы.

Объектно-ориентированное проектирование

В программировании делается акцент на правильном и эффективном использовании механизмов конкретных языков программирования. В проектировании, наоборот, основное внимание уделяется правильному и эффективному структурированию сложных систем. Что же представляет собой объектно-ориентированное проектирование? Мы предлагаем следующее определение.

> Объектно-ориентированное проектирование — это метод проектирования, сочетающий процесс объектно-ориентированной декомпозиции и систему обозначения для представления логической и физической, а также статической и динамической моделей проектируемой системы.

В данном определении обращают на себя внимание следующие две важные части: объектно-ориентированное проектирование 1) основывается на объектно-ориентированной декомпозиции и 2) использует разные системы обозначений для

представления логических (классы и объекты) и физических (модули и процессы) моделей системы, а также ее статические и динамические аспекты.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного. В первом случае для выражения логической структуры системы используются классы и объекты, а во втором — алгоритмические абстракции. Иногда объектно-ориентированным проектированием называют любой метод, ведущий к объектной декомпозиции.

Объектно-ориентированный анализ

Объектная модель испытала влияние ранних моделей жизненного цикла программного обеспечения. Традиционные методы структурного анализа, прекрасно описанные в работах ДеМарко (DeMarko) [29], Йордона (Yordona) [30], Гейна (Gane) и Сарсона (Sarson) [31], а с приложением к системам реального времени — в работах Варда (Ward) и Меллора (Mellor) [32], а также Хатли (Hatley) и Пирбхаи (Pirbhai) [33], основное внимание уделяют потокам данных внутри системы. Объектно-ориентированный анализ (object-oriented analysis — OOA) главный акцент делает на создании моделей реальной действительности на основе объектно-ориентированного мировоззрения.

> Объектно-ориентированный анализ — это метод анализа, исследующий требования к системе с точки зрения классов и объектов, относящихся к словарю предметной области.

Как соотносятся между собой объектно-ориентированный анализ, проектирование и программирование? Результатами объектно-ориентированного анализа являются модели, лежащие в основе объектно-ориентированного проектирования, которое в свою очередь позволяет разработать схему полной реализации системы с использованием объектно-ориентированного программирования.

2.3 Составные части объектного подхода

Дженкинс (Jenkins) и Глазго (Glasgow) утверждают: “Большинство программистов работают только на одном языке и придерживаются только одного стиля. Они принимают парадигму, обусловленную используемым языком программирования. Часто они даже не рассматривают альтернативные точки зрения на проблему, а, следовательно, им трудно выбрать стиль, лучше соответствующий решаемой задаче” [34]. Боброу и Стефик считают, что стиль программирования — “это способ создания программ с помощью определенных принципов программирования и подходящего языка, позволяющего писать ясные программы” [35]. По их мнению, существуют пять основных разновидностей стилей программирования. Они перечислены ниже вместе с используемыми абстракциями.

1.	Процедурно-ориентированный	Алгоритмы
2.	Объектно-ориентированный	Классы и объекты
3.	Логико-ориентированный	Цели, часто выраженные в терминах исчисления предикатов
4.	Продукционный	Правила “если, то”
5.	Ориентированный на ограничения	Инвариантные соотношения

Не существует какого-то одного стиля программирования, который был бы наилучшим во всех возможных областях. Например, для проектирования баз знаний может оказаться наиболее удобным продукционный стиль, а для решения вычислительных задач — процедурно-ориентированный. Наш опыт подсказывает, что объектно-ориентированный стиль является наиболее приемлемым для самого широкого диапазона приложений. Действительно, эта парадигма часто служит архитектурным остовом, на котором реализуются другие парадигмы.

Каждый стиль программирования имеет свою концептуальную основу и требует уникального подхода к решению задачи. Основой объектно-ориентированного стиля является объектная модель.Эта модель состоит из следующих четырех главных элементов.

- Абстракция
- Инкапсуляция
- Модульность
- Иерархия

Термин *главный элемент* означает, что без любого из них модель не является объектно-ориентированной. Кроме главных, в этой модели существуют еще три дополнительных элемента.

- Контроль типов
- Параллелизм
- Персистентность

Термин *дополнительный элемент* означает полезный, но не существенный компонент объектной модели.

Разумеется, программировать на таких языках, как Smalltalk, Object Pascal, C++, Eiffel или Ada можно, и не применяя объектную модель, но в этом случае проект будет напоминать программы, написанные на языках FORTRAN, Pascal или C. Выразительная мощность объектно-ориентированного языка будет либо утеряна, либо искажена. Кроме того, что еще более важно, в этом случае программисты вряд ли справятся со сложностью задачи.

Абстрагирование

Абстрагирование — один из основных методов, позволяющих справиться со сложностью. Даль (Dahl), Дейкстра (Dijkstra) и Хоар (Hoare) утверждают: “Абстрагирование проистекает из распознавания сходства между определенными объектами, ситуациями или процессами в реальном мире, и является результатом решения сконцентрировать внимание на общих свойствах и проигнорировать различия” [36]. Шоу (Show) считает абстракцией “упрощенное описание, или спецификацию системы, выделяющую одни свойства и затеняющую другие, а хорошая абстракция подчеркивает значимые детали и отбрасывает несущественные на данный момент” [37]. Берзинс (Berzins), Грей (Gray) и Науман (Naumann) рекомендуют “признавать идею абстракцией только, если она может быть описана, понята и проанализирована независимо от механизма ее дальнейшей реализации” [38]. Суммируя эти разные точки зрения, приходим к следующему определению абстракции.

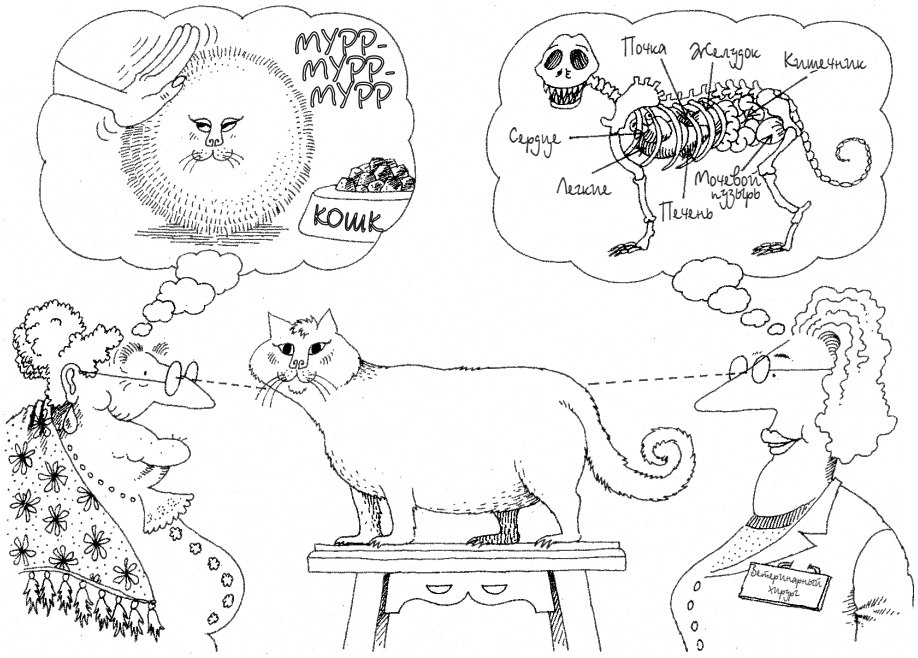
> Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко описывает его концептуальные границы с точки зрения наблюдателя.

Абстракция концентрирует внимание на внешнем представлении объекта и позволяет отделить существенные особенности поведения от их реализации. Абельсон (Abelson) и Суссман (Sussman) назвали такое разделение поведения от его реализации *принципом минимальных обязательств* (principle of least commitment) [39], в соответствии с которым интерфейс объекта должен обеспечивать только существенные аспекты его поведения и ничего больше [40]. Существует также дополнительный *принцип наименьшего удивления* (principle of least astonishment), согласно которому абстракция должна полностью описывать поведение объекта, ни больше и ни меньше, и не порождать сюрпризы или побочные эффекты, выходящие за пределы абстракции.

Выбор правильной совокупности абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Поскольку эта тема чрезвычайно важна, ей целиком посвящена глава 4.

“Существует целый спектр абстракций, начиная с объектов, почти точно соответствующих реалиям предметной области, и заканчивая совершенно излишними объектами” [41]. Перечислим абстракции в порядке убывания их полезности.

• Абстракция сущности	Объект, представляющий собой полезную модель некой сущности в предметной области
• Абстракция действия	Объект, состоящий из обобщенного множества операций, каждая из которых выполняет однотипные функции



Абстракция концентрирует внимание на существенных свойствах объекта с точки зрения наблюдателя

• Абстракция виртуальной машины	Объект, группирующий операции, которые либо вместе используются на более высоком уровне управления, либо сами используют некоторый набор операций более низкого уровня
• Произвольная абстракция	Объект, включающий в себя набор операций, не имеющих друг с другом ничего общего

Мы стараемся строить абстракции сущности, так как они прямо соответствуют сущностям предметной области.

Клиентом (client) называется любой объект, использующий ресурсы другого объекта (называемого сервером (server)). Поведение объекта характеризуется услугами, которые он оказывает другим объектам, и операциями, которые он может выполнять над другими объектами. Такой подход концентрирует внимание на внешних проявлениях объекта и приводит к идее, которую Мейер (Meyer) назвал *контрактной моделью* (contract model) программирования [42]: внешнее проявление объекта определяет его контракт с другими объектами, подлежащий выполнению его внутренним устройством (часто во взаимодействии с другими объектами). Контракт фиксирует все предположения, которые объект-клиент может формулировать относительно поведения объекта-сервера. Иначе говоря, контракт определяет ответственность объекта, т.е. его обязательное поведение [43].

Каждая операция, предусмотренная этим контрактом, однозначно определяется ее уникальной сигнатурой, состоящей из формальных параметров и типа возвращаемого значения. Полная совокупность операций, которые клиент может осуществлять над другим объектом, вместе с установленным порядком их вызова образуют *протокол* (protocol). Протокол указывает все возможные способы, которыми объект может действовать или реагировать на воздействие, полностью определяя внешнее статическое и динамическое представление абстракции.

Главной идеей абстракции является концепция инварианта. *Инвариант* (invariant) — это логическое условие (истинное или ложное), которое должно выполняться всегда. Для каждой операции, ассоциированной с объектом, можно задать *предусловия* (инварианты, предполагаемые операцией) и *постусловия* (инварианты, которым удовлетворяет операция). Нарушение инварианта противоречит контракту, связанному с абстракцией. Если нарушается предусловие, то это значит, что клиент не соблюдает свои обязательства и сервер не в состоянии надежно функционировать. Аналогично, если нарушается постусловие, то свои обязательства не выполняет сервер, так что клиент больше не может ему доверять. Признаком нарушения некоего инварианта является исключительная ситуация. Некоторые языки позволяют объектам генерировать исключительные ситуации, чтобы отменить дальнейшую обработку данных и предупредить о возникшей проблеме другие объекты, которые в свою очередь могут перехватывать и обрабатывать исключения.

Заметим, что понятия *операция, метод* и *функция-член* происходят из разных языков программирования (Ada, Smalltalk и C++ соответственно). По существу, они обозначают одно и то же и в дальнейшем будут использоваться как синонимы.

Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном запоминающем устройстве. Кроме того, у него есть имя и содержание. Все эти атрибуты являются статическими свойствами. Конкретные значения каждого из этих свойств носят динамический характер и зависят от жизненного цикла объекта. Например, файл можно увеличить или уменьшить, а также изменить его имя и содержимое. Суть всех программ, написанных в процедурном стиле, составляют действия, изменяющие динамические характеристики объектов. В таких программах события происходят только в результате вызова подпрограмм и выполнения операторов. С другой стороны, в программах, написанных в стиле, ориентированном на правила, события происходят только тогда, когда определенные условия приводят в действие установленные правила, которые, в свою очередь, активизируют другие правила, и т.д. В программах, написанных в объектно-ориентированном стиле, события связаны с воздействием на объекты (т.е. с передачей объектам определенных сообщений). Выполнение операции над объектом вызывает у него определенную реакцию. Поведение объекта полностью определяется операциями, которые над ним можно выполнять, и его реакцией на внешние воздействия.

Примеры абстракций

Проиллюстрируем сказанное несколькими примерами. Полное описание процесса выделения абстракций приведено в главе 4.

В теплице, использующей гидропонику, растения выращиваются в питательном растворе, а не в почве. Обеспечение соответствующей окружающей среды — тонкая работа, зависящая как от сорта выращиваемой культуры, так и ее возраста. Для этого необходимо контролировать разнообразные факторы: температуру, влажность, освещение, кислотность и концентрацию питательных веществ. В больших хозяйствах широко используются автоматические системы, контролирующие и регулирующие эти факторы. Проще говоря, цель автоматизации — добиться соблюдения режима выращивания разнообразных растений при минимальном вмешательстве человека.

Одна из ключевых абстракций в такой задаче — датчик. Существует несколько разновидностей датчиков. Все, что влияет на урожайность, должно быть измерено, поэтому необходимо иметь датчики температуры воды и воздуха, влажности, кислотности и концентрации питательного раствора и т.д. С точки зрения наблюдателя датчик температуры — это объект, предназначенный для измерения температуры в конкретном месте. Что такое температура? Это числовой параметр, изменяющийся в ограниченном диапазоне и измеряемый с определенной точностью по шкалам Фаренгейта, Цельсия или Кельвина. Что такое местоположение датчика? Это некоторое четко определенное место в теплице, температуру в котором нам необходимо измерить. Таких мест, как правило, немного. Для датчика температуры существенно, не где он расположен, а то, что он в принципе может быть размещен в конкретном месте, отдельно от других датчиков. Теперь выясним, что должен делать датчик температуры? Допустим, в соответствии с проектом датчик должен измерять температуру в точке своего размещения и сообщать ее по запросу. Уточним теперь, какие действия клиент может выполнять с датчиком? Пусть проектное решение предусматривает, что клиент может калибровать датчик и запрашивать у него текущее значение температуры. (См. рис. 2.6. Обратите внимание на то, что это представление похоже на представление класса в системе обозначений UML 2.0. Эта система обозначений будет рассмотрена в главе 5.)

Описанная выше абстракция была пассивной: клиент должен попросить объект `Temperature Sensor` измерить температуру. Однако есть и другая вполне разумная абстракция, расширяющая возможности системы. В частности, датчик мог бы быть активным и самостоятельно сообщать другим объектам об изменении температуры в определенном месте на заданное количество градусов. Эта абстракция мало отличается от предыдущей, за исключением новой формулировки обязанностей объекта: теперь датчик должен самостоятельно сообщать об изменении температуры, а не по запросу. Какие новые операции нужны ему в связи с этим?

Абстракция: Temperature Sensor
Важные свойства: температура местоположение
Обязанности: сообщать о текущей температуре калибровать

Рис. 2.6. Абстракция `Temperature Sensor`

Новая абстракция лишь ненамного сложнее первой (см. рис. 2.7). Клиент этой абстракции может изменять пороговое значение датчика температуры. Обязанность датчика — сообщать об изменении текущей температуры, когда ее текущее значение становится выше или ниже порогового. При вызове этой функции датчик сообщает свое местоположение и текущую температуру, снабжая клиента требуемой информацией.

Абстракция: Active Temperature Sensor
Важные свойства: температура местоположение пороговое значение
Обязанности: сообщать о текущей температуре калибровать установить пороговое значение

Рис. 2.7. Абстракция `Active Temperature Sensor`

Способ выполнения обязательств абстракции `Active Temperature Sensor` зависит от ее внутреннего представления и не должен интересовать внешних клиентов. Эта информация является секретом класса и реализуется его закрытой частью совместно с определениями функций-членов.

Рассмотрим теперь другую абстракцию. Для каждой культуры должен существовать производственный план, описывающий изменение во времени температуры, освещения, подкормки и ряда других факторов, обеспечивающих максимально высокий урожай. Этот план также можно абстрагировать, поскольку он является частью предметной области.

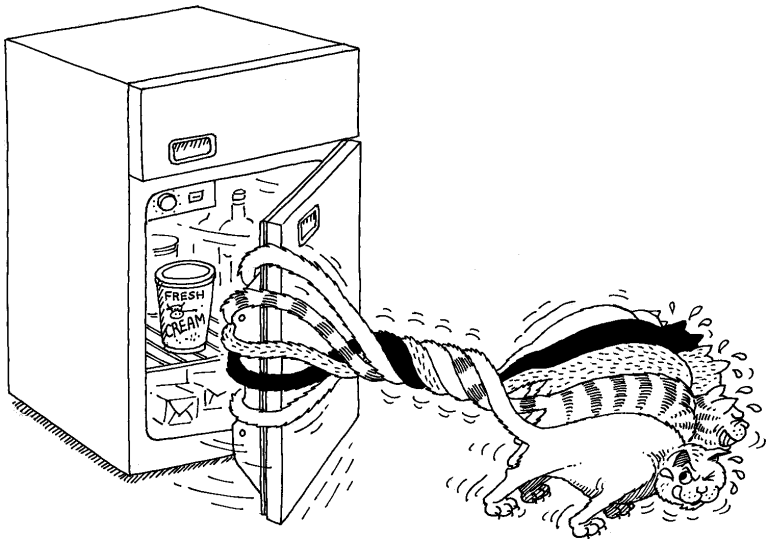
Производственный план должен предусматривать средства контроля всех необходимых действий по выращиванию растений и содержать их временнóе расписание. Например, для некоторой культуры на 15-е сутки роста план может требовать в течение 16 часов поддерживать температуру на уровне 25°C, причем 14 часов из них — с освещением, а затем понизить температуру до 18°C на остальное время суток. Кроме того, план может предусматривать внесение дополнительных удобрений в середине дня, чтобы поддержать слабую кислотность. С точки зрения внешнего наблюдателя клиент должен иметь возможность составлять план, модифицировать его и запрашивать его содержание, как показано на рис. 2.8. (Обратите внимание на то, что эта абстракция также эволюционирует с течением времени. По мере конкретизации деталей такая обязанность, как "задать план", может быть разделена на множество обязанностей, например, "установить температуру", "установить кислотность" и т.д. Вполне закономерно, что по мере уточнения потребностей клиента проект достигает зрелости и проектировщик начинает рассматривать разные подходы к его реализации.)

Абстракция: Growing Plan
Важные свойства: имя
Обязанности: задать план изменить план аннулировать план

Рис. 2.8. Абстракция `Growing Plan`

Проектное решение заключается в том, что в обязанности абстракции плана не входит выполнение самого плана, — это будет обязанностью другой абстракции (например, `Plan Controller`). Таким образом, можно *разделить понятия* (separation of concerns) между разными частями системы, уменьшив концептуальный размер каждой отдельной абстракции. Например, объект может предусматривать взаимодействие человека и компьютера, допуская ручное изменение плана. Объект, содержащий детали плана выращивания, должен уметь изменять состояние объекта `Growing Plan`. Кроме того, должен существовать объект, выполняющий план. Для этого он должен иметь возможность прочитать план в определенный момент времени.

Как следует из описанного примера, ни один объект не изолирован от других. Все объекты взаимодействуют друг с другом для обеспечения определенного



Объекты взаимодействуют между собой, обеспечивая требуемое поведение

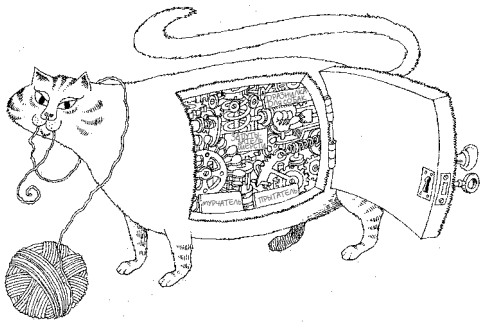
поведения[1]. Проектное решение, регламентирующее взаимодействие между объектами, определяет границы абстракций и протокол каждого объекта.

Инкапсуляция

Несмотря на то что абстракция `Growing Plan` описана как отображение время–действие (time/action mapping), она не обязательно должна быть реализована как таблица или ассоциативный массив (map). Действительно, выбор способа реализации абстракции `Growing Plan` не имеет никакого отношения к контракту клиента, пока ее представление соответствует контракту. Проще говоря, абстракция объекта должна предшествовать решению о ее реализации. Как только решение о реализации принято, оно должно рассматриваться как секрет абстракции и скрываться от большинства клиентов.

Абстракция и инкапсуляция дополняют друг друга. В центре внимания абстракции находится наблюдаемое поведение объекта, а инкапсуляция сосредоточена на реализации, обеспечивающей заданное поведение. Как правило, инкапсуляция осуществляется с помощью сокрытия информации (а не просто сокрытия данных), т.е. утаивания всех несущественных деталей объекта. Обычно скрываются как структура объекта, так и реализация его методов. “Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой ча-

[1]Иначе говоря, перефразируя поэта Джона Донна (John Donne), объект — не остров (хотя понятие острова можно абстрагировать в виде объекта).



Инкапсуляция скрывает детали
реализации объекта

сти” [44]. В то время как абстракция “помогает людям думать о том, что они делают”, инкапсуляция “позволяет легко перестраивать программы” [45].

Инкапсуляция определяет четкие границы между разными абстракциями и позволяет, таким образом, провести ясные различия между понятиями. Рассмотрим, например, структуру растения. Для того чтобы изучить механизм фотосинтеза, оставаясь на верхнем уровне абстракции, можно игнорировать такие детали, как функции корней растения или химические свойства стенок клеток. Аналогично, при проектировании баз данных принято писать программы так, чтобы они зависели не от физического представления данных, а от логической структуры данных [46]. В обоих случаях объекты, расположенные на одном уровне абстракции, защищены от деталей реализации объектов, относящихся к более низкому уровню.

“Для того чтобы абстракция работала, ее реализация должна быть инкапсулирована” [47]. На практике это означает, что каждый класс должен состоять из двух частей: интерфейса и реализации. Интерфейс класса фиксирует лишь внешнее представление объекта, описывая абстракцию поведения всех объектов данного класса. Реализация класса содержит как представление абстракции, так и механизмы достижения требуемого поведения объекта. Интерфейс класса содержит все предположения, которые клиент может сформулировать относительно объектов класса, а реализация скрывает от других объектов все детали, не имеющие отношения к процессу — все, что не касается предположений клиента.

Итак, инкапсуляцию можно определить следующим образом.

> Инкапсуляция — это процесс разделения элементов абстракции, определяющих ее структуру и поведение; инкапсуляция предназначена для изоляции контрактных обязательств абстракции от их реализации.

Бриттон (Britton) и Парнас (Parnas) назвали инкапсулированные детали “секретами абстракции” [48].

Примеры инкапсуляции

Для иллюстрации принципа инкапсуляции вернемся к примеру гидропонного тепличного хозяйства. Одной из важнейших абстракций данной предметной области является обогреватель. Обогреватель представляет собой абстракцию низкого уровня, поэтому существуют только три целесообразных действий с этим объектом: включение, выключение и запрос состояния.

Разделение понятий

Абстракция `Heater` не должна поддерживать постоянную температуру. Эту обязанность следует возложить на другую абстракцию (например, `Heater Controller`), которая должна взаимодействовать с датчиком температуры и обогревателем, обеспечивая поведение более высокого уровня. Это поведение относится к *более высокому уровню*, потому что оно основывается на простой семантике обогревателя и датчика, добавляя к ним дополнительные свойства, в частности, *гистерезис* предотвращает слишком быстрое включение и выключение нагревателя, когда температура приближается к критической. Разделив ответственность, мы делаем каждую абстракцию более цельной.

Интерфейс доступа (т.е. функции, которые класс может выполнить по требованию клиента, как показано на рис. 2.9) — это все, что клиенту следует знать о классе `Heater`.

Обращаясь к внутреннему представлению класса `Heater`, можно обнаружить совершенно другую картину. Предположим, что проектировщики решили разместить управляющие компьютеры вне теплицы (возможно, чтобы избежать слишком суровых климатических условий) и соединить их с датчиками и исполнительными механизмами с помощью линии последовательной передачи. Класс `Heater` естественно реализовать с помощью электромеханических реле, управляющих мощностью, поступающей на каждый обогреватель и получающих сообщения по линиям последовательной передачи. Например, для включения обогревателя можно передать специальную командную строку, номер обогревателя и индикатор включения.

Предположим, что по какой-либо причине проектировщики изменили решение и вместо последовательных линий передачи решили использовать для управления запоминающее устройство. Интерфейс класса `Heater` останется прежним, но его реализация кардинально изменится. В этом случае клиенты, которым доступен только интерфейс класса `Heater`, не обнаружат никаких изменений. Это главная особенность инкапсуляции. Фактически, клиента не должна интересовать реализация класса, пока класс `Heater` удовлетворяет его требованиям.

Абстракция: Heater
Важные свойства: местоположение состояние
Обязанности: включить выключить сохранять состояние

Рис. 2.9. Абстракция `Heater`

Рассмотрим теперь реализацию класса `Growing Plan`. Как указывалось ранее, временнóе расписание, по существу, является отображением времени в действие. Вероятно, наиболее эффективной реализацией такого отображения является словарь пар время/действие с открытой хеш-таблицей. Нет никакой необходимости хранить список действий для каждого часа, поскольку события не происходят настолько часто. Вместо этого можно хранить только моменты времени, в которых должны выполняться действия, изменяющие состояние системы, интерполируя реализацию между этими моментами времени.

Таким образом, реализация класса инкапсулирует два секрета: использование открытой хеш-таблицы (которая относится к словарю проектных решений, а не предметной области) и использование интерполяции для уменьшения требований к объему памяти (в противном случае пришлось бы хранить список пар время/действие для всего сезона). Клиентам этой абстракции совершенно не обязательно знать об этом решении, поскольку оно никак не влияет на внешнее представление класса.

Разумная инкапсуляция должна локализовать проектные решения, которые могут измениться. По мере эволюции системы ее разработчики могут обнаружить, что какие-то операции выполняются недопустимо долго, а какие-то объекты занимают слишком много памяти. В таких ситуациях внутреннее представление объекта, как правило, изменяется, чтобы реализовать более эффективные алгоритмы или оптимизировать использование памяти, заменяя хранение данных их вычислением. Возможность изменения представления абстракции без ведома клиентов является важным преимуществом инкапсуляции.

Сокрытие информации является относительным понятием. То, что скрыто на одном уровне абстракции, может оказаться открытым на другом уровне. В принципе, реализацию объектов можно раскрыть. Правда, в большинстве случаев это происходит в соответствии с явным замыслом разработчика абстракции и толь-

ко если разработчики классов-клиентов согласны на увеличение сложности. Инкапсуляция не способна предотвратить ошибочные решения. Как отметил Страуструп, “Инкапсуляция защищает от случайностей, но не от жульничества” [49]. Разумеется, ни один язык программирования не способен полностью закрыть доступ к реализации класса, хотя операционная система может перекрыть доступ к конкретному файлу, содержащему реализацию класса.

Модульность

“Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность... Однако гораздо важнее тот факт, что при этом в модульной программе возникает множество хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для понимания программы” [50]. В некоторых языках программирования, например в Smalltalk, модулей нет, и единственной физической единицей декомпозиции является класс. Во многих других языках, включая Object Pascal, C++ и Ada, модуль — это самостоятельная языковая конструкция, позволяющая создавать комплексы отдельных проектных решений. В этих языках логическую структуру системы образуют классы и объекты. Она помещается в модули, из которых состоит физическая структура системы. Это свойство особенно полезно в крупных системах, состоящих из многих сотен классов.

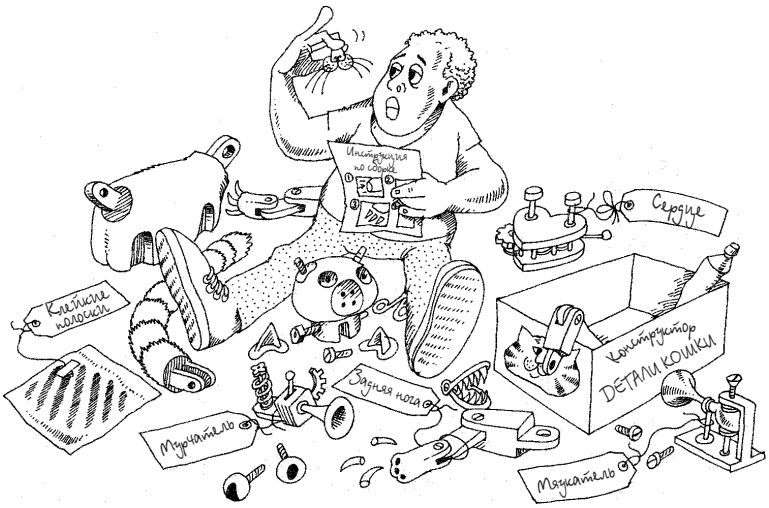
“Модульность — это разделение программы на фрагменты, которые компилируются по отдельности, но связаны между собой”. В соответствии с определением Парнаса (Parnas): “Связи между модулями — это их предположения о работе друг друга” [51]. В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, интерфейс модуля отделен от его реализации. Таким образом, можно утверждать, что модульность и инкапсуляция тесно связаны между собой.

Правильный выбор модулей для решения поставленной задачи является почти такой же сложной задачей, как выбор правильного набора абстракций. Зельковиц (Zelkowitz) был абсолютно прав, утверждая: “Поскольку в начале работы над проектом решение может быть неясным, декомпозиция на более мелкие модели может вызвать затруднения. Для традиционных приложений (например, разработки компиляторов) этот процесс можно стандартизировать, но для новых задач (например, оборонительных систем или систем управления космическими кораблями) задача может оказаться довольно трудной” [52].

Модули служат физическими контейнерами, содержащими классы и объекты, возникающие при логическом проектировании системы. Точно такая же ситуация возникает у проектировщиков материнских плат компьютеров. Требуемую логику можно создать с помощью элементарных схем типа НЕ-И (NAND), НЕ-ИЛИ (NOR), НЕ (NOR), но эти вентили необходимо упаковать в стандартные интеграль-

ные схемы. Однако программист обладает намного большей свободой, как если бы электротехник имел в своем распоряжении кремниевую мастерскую с неограниченными возможностями.

Разработчик, решающий небольшие задачи, может описать все классы и объекты в одном модуле. Однако при создании большинства программ (кроме самых простых) лучше объединить логически связанные классы и объекты, оставив открытыми лишь те элементы, которые нельзя скрывать от других модулей. Такой способ разбиения на модули хорош, но его можно довести до абсурда. Рассмотрим, например, программу, которая выполняется на многопроцессорном компьютере и использует механизм передачи сообщений. В больших системах, например, в системах управления, как правило, происходит обмен несколькими сотнями и даже тысячами видов сообщений. Было бы наивным определять каждый класс сообщения в отдельном модуле. Это крайне неудачное проектное решение. Оно не только чрезвычайно запутает документирование программы, но и усложнит поиск класса, необходимого пользователю. Более того, если потребуется изменить проект, то придется модифицировать и перекомпилировать сотни модулей. Этот пример показывает, что сокрытие информации имеет обратную сторону [53]. Произвольное разделение программы на модули иногда гораздо хуже, чем отсутствие модульности вообще.



Модульность позволяет упаковать абстракции в отдельные единицы программы

В традиционном структурном проектировании модульность, как правило, проявляется в группировке подпрограмм по логическим группам на основе критериев связности и целостности. В объектно-ориентированном программировании модульность имеет несколько иной характер, поскольку она предусматривает фи-

зическую упаковку классов и объектов, существенно отличающихся от подпрограмм.

Существует несколько эмпирических приемов и правил, позволяющих обеспечить разумную модульность. Бриттон (Britton) и Парнас (Parnas) утверждают: “Конечной целью декомпозиции программы на модули является снижение затрат на программирование, благодаря независимому проектированию и тестированию... Структура каждого модуля должна быть достаточно простой для понимания, допускать независимую реализацию других модулей и не влиять на поведение других модулей, а также позволять легкое изменение проектных решений” [54]. На практике эти принципы реализуются лишь частично. Повторное компилирование тела отдельного модуля не требует больших затрат, поскольку остальные модули остаются неизменными, а обновляются только связи между ними. Однако стоимость повторной компиляция интерфейса модуля относительно велика. В языках программирования со строгим контролем типов приходится повторно компилировать интерфейс и тело модифицированного модуля, затем все зависимые модули и т.д. В итоге для очень больших программ повторная компиляция может оказаться очень долгой (если только среда разработки не поддерживает инкрементную компиляцию), что явно нежелательно. Очевидно, руководитель проекта не должен допускать слишком частую повторную компиляцию. По этой причине интерфейс модуля должен быть как можно более узким (обеспечивая, тем не менее, необходимые связи с другими модулями). Правильный стиль программирования требует как можно больше скрывать реализацию модуля. Постепенный перенос описаний из реализации в интерфейс гораздо безопаснее, чем удаление излишнего интерфейсного кода.

Таким образом, проектировщик должен балансировать между двумя противоположными тенденциями: желанием инкапсулировать абстракции и необходимостью открыть доступ к тем или иным абстракциям для других модулей. “Изменяемые детали системы следует скрывать в отдельных модулях. Открытыми следует оставлять лишь элементы, обеспечивающие связь между модулями, вероятность изменения которых крайне мала. Все структуры данных в модуле должны быть закрыты. Они могут оставаться доступными для процедур внутри модуля, но должны быть недоступными для всех внешних подпрограмм. Доступ к информации, хранящейся внутри модуля, должен осуществляться с помощью вызова процедур данного модуля” [55]. Иначе говоря, следует создавать цельные (объединяющие логически связанные между собой абстракции) и слабо связанные между собой модули. Таким образом, можно сформулировать следующее определение.

> Модульность — это свойство системы, разложенной на цельные, но слабо связанные между собой модули.

Итак, принципы абстракции, инкапсуляции и модульности являются синергетическими. Объект позволяет определить четкие границы отдельной абстракции, а инкапсуляция и модульность создают барьеры между абстракциями.

Разделение системы на модули имеет два дополнительных аспекта. Во-первых, поскольку модули, как правило, являются элементарными и неделимыми структурными элементами программного обеспечения и могут использоваться в приложениях неоднократно, разделение программы на модули должно допускать повторное использование кода. Во-вторых, многие компиляторы создают отдельные сегменты кода для каждого модуля. Следовательно, могут возникнуть физические ограничения на размер модуля. Динамика вызовов подпрограмм и размещение объявлений внутри модулей могут существенно влиять на локальность ссылок и на управление страницами виртуальной памяти. Вызовы между несмежными сегментами приводят к “промахам кэша” (cashe miss) и порождают режим интенсивной подкачки страниц памяти (trashing), что в итоге тормозит работу всей системы.

Разбиение на модули должно учитывать дополнительные факторы, не носящие технического характера. Как правило, распределение работы между участниками проектной группы осуществляется по модульному принципу. Следовательно, систему необходимо разделять на модули так, чтобы минимизировать количество связей между участниками проекта. Опытные программисты обычно отвечают за разработку интерфейсов модулей, а менее опытные — за их реализацию. Аналогичная ситуация наблюдается в более крупном масштабе в отношениях между субподрядчиками. Распределение абстракций должно облегчать быстрое согласование интерфейсов модулей между компаниями, участвующими в проекте. Изменения интерфейсов, уже согласованных между компаниями, обычно сопровождаются воплями и зубовным скрежетом — помимо огромного расхода бумаги, — поэтому проектирование интерфейса является крайне консервативным делом. Кроме того, модуль служит отдельной единицей документации и администрирования. Десять модулей требуют в десять раз больше описаний, чем один, и поэтому, к сожалению, иногда на декомпозицию проекта влияют (как правило, негативно) побочные соображения, связанные с составлением проектной документации. На модульность проекта может влиять его секретность. Например, бóльшая часть кода может считаться открытой, а остальная часть — секретной. В таком случае секретную часть размещают в отдельных модулях.

Согласовать эти противоречивые требования довольно трудно, но следует помнить главное: выявление классов и объектов, с одной стороны, и организация их в виде отдельных модулей, с другой стороны, — это практически независимые проектные решения. Идентификация классов и объектов представляет собой часть логического проектирования системы, а идентификация модулей — часть ее физического проектирования. Разумеется, невозможно полностью осуществить

логическое проектирование системы до реализации физического проектирования, и наоборот. Два этих процесса носят итеративный характер.

Примеры модульности

Исследуем модульность гидропонной теплицы. Предположим, что проектировщик решил использовать для управления ее работой коммерческую рабочую станцию. С помощью этой рабочей станции оператор может создавать новые планы выращивания, модифицировать старые планы и контролировать их исполнение. Так как основной абстракцией в системе является план выращивания, необходимо создать модуль, содержащий все классы, связанные с отдельными планами выращивания (например, `FruitGrowmingPlan`, `GrainGrowingPlan`). В этом модуле можно собрать все классы, относящиеся к абстракции `GrowingPlan`. Кроме того, можно создать модуль, в котором будут собраны все функции, реализующие пользовательский интерфейс.

Проект, вероятно, будет содержать много других модулей. В конце концов необходимо определить главную программу, которая будет вызываться для запуска приложения. В объектно-ориентированном проектировании выбор главной программы часто является наименее важным решением, в то время как в традиционном структурном проектировании главная программа является краеугольным камнем, на который опирается весь проект. Объектно-ориентированный подход является более естественным. Мейер (Meyer) утверждает: “Реальные системы программного обеспечения естественно описывать как системы, предлагающие некоторый набор услуг. Сводить их к одной функции, в принципе, можно, но не естественно... Реальные системы не имеют крыши” [56].

Иерархия

Абстракция — вещь полезная, но их количество всегда, кроме самых тривиальных приложений, намного превышает человеческие возможности. Инкапсуляция позволяет справиться со сложностью, скрыв внутреннее представление абстракций. Модульность также упрощает понимание проблемы, позволяя объединить логически связанные абстракции в группы.

Определим иерархию следующим образом.

> Иерархия — это ранжирование, или упорядочение абстракций.

Наиболее важными видами иерархии в сложных системах являются структура классов (иерархия “общее/частное”) и структура объектов (иерархия “целое/часть”).

Примеры иерархии: одиночное наследование

Как указывалось выше, наследование представляет собой одну из наиболее важных иерархий, основанных на отношении “является” (“is a”). Оно лежит в ос-

нове многих объектно-ориентированных систем. Наследование означает такое отношение между классами, когда один класс заимствует структуру или поведение одного или нескольких других классов (одиночное и множественное наследование соответственно). Иначе говоря, наследование создает иерархию абстракций, в которой подклассы заимствуют свойства одного или нескольких суперклассов. Обычно подкласс наращивает или переопределяет существующую структуру и поведение суперкласса.

С семантической точки зрения, наследование описывает отношение типа "является". Например, медведь — это разновидность млекопитающего ("is a" kind of mammal"), дом — это разновидность материальных активов ("is a" kind of tangible asset), а метод быстрой сортировки — это разновидность алгоритма сортировки ("is a" kind of sorting algorithm). Таким образом, наследование порождает иерархию обобщения/специализации, в которой подкласс конкретизирует более общую структуру или поведение своего суперкласса. Действительно, существует безошибочный тест для наследования: если класс `B` не является разновидностью класса `A`, то класс `B` не должен быть наследником класса `A`.

Рассмотрим разные планы выращивания растений в гидропонной теплице. В предыдущем разделе описана абстракция очень общего плана выращивания растений. Однако разные культуры требуют разных планов выращивания. Например, планы выращивания всех фруктов похожи друг на друга, но отличаются от планов выращивания овощей или цветов. Учитывая эту кластеризацию абстракций, целесообразно определить стандартный план выращивания фруктов, а именно расписание поливов и уборки урожая. План `FruitGrowingPlan` можно считать разновидностью плана `GrowingPlan`.

Это означает, что план `FruitGrowingPlan` является разновидностью плана `GrowingPlan`. То же самое можно сказать о планах `GrainGrowingPlan` или `VegetableGrowingPlan`. В этом случае класс `GrowingPlan` является суперклассом, а остальные — подклассами.

По мере эволюции иерархии наследования структура и поведение, общие для разных классов, имеют тенденцию мигрировать в наиболее общий суперкласс. Именно поэтому наследование часто называют иерархией обобщение/специализация. Суперклассы представляют обобщенные абстракции, а подклассы — специализации, в которые добавляются, модифицируются и даже скрываются поля и методы из суперклассов. Наследование позволяет экономить выражения при описании абстракций. Пренебрежение иерархиями "общее/частное" может привести к громоздким проектным решениям. "Без наследования каждый класс представляет собой изолированную структурную единицу и должен разрабатываться "с нуля". Разные классы теряют общность, поскольку каждый программист реализует их методы по-своему. В этом случае согласованность классов системы может быть обеспечена только за счет дисциплинированности программистов.

Наследование позволяет создавать новые программы точно так же, как вводятся новые понятия — сравнивая новое с уже известным” [57].

Между принципами абстракции, инкапсуляции и иерархии существует здоровая конкуренция. “Абстракция данных создает непроницаемый барьер, скрывающий методы и состояние объекта. Принцип наследования требует открыть доступ и к состоянию, и к методам объекта, не прибегая к абстракции” [58]. У любого класса обычно существуют два вида клиентов: объекты, выполняющие операции над экземплярами данного класса, и подклассы, наследующие свойства класса. Лисков (Liskov) отмечает, что существуют три способа нарушения инкапсуляции с помощью наследования: “подкласс может получить доступ к переменным экземпляра своего суперкласса, вызвать закрытую функцию и, наконец, обратиться напрямую к суперклассу своего суперкласса” [59]. Разные языки программирования по-разному согласовывают наследование с инкапсуляцией. Наиболее гибкими в этом отношении являются C++ и Java. В частности, интерфейс класса может состоять из трех частей: закрытой (private), доступной только самому классу; защищенной (protected), в которой объявляются члены, доступные только для класса и его подклассов; и открытую (public), доступную всем клиентам.

Примеры иерархии: множественное наследование

В предыдущем примере продемонстрировано одиночное наследование: подкласс `FruitGrowingPlan` имел только один суперкласс `GrowingPlan`. Иногда полезно реализовать наследование от нескольких суперклассов. Предположим, например, что требуется определить класс, представляющий разновидности растений. Анализ предметной области показывает, что цветы, фрукты и овощи имеют свои особые свойства, существенные для технологии их выращивания. Например, для абстракции цветов важно знать периоды цветения и созревания семян. Аналогично, для абстракций фруктов и овощей важен момент сбора урожая. Исходя из этих соображений, можно создать два новых класса — `Flower` и `FruitVegetable` — наследники суперкласса `Plant`. Какую же модель следует выбрать, если растение и цветет, и плодоносит? Например, флористы часто используют для составления букетов цветки яблони, вишни и сливы. Для этой абстракции придется создать третий класс `FlowerFruitVegetable` — наследник классов `Flower` и `FruitVegetablePlant`.

Чтобы наилучшим образом выразить новую абстракцию и не допустить ошибку, следует создать классы, независимо друг от друга описывающие уникальные свойства цветов, а также фруктов и овощей соответственно. Эти два класса не имеют суперкласса. Они называются *классами-примесями* (mixin classes), поскольку смешиваются вместе с другими классами, создавая новые подклассы.

Например, можно описать класс `Rose` (см. рис. 2.10), наследующий свойства классов `Plant` и `FlowerMixin`. Таким образом, экземпляры подкласса `Rose` наследуют структуру и поведение классов `Plant` и `FlowerMixin`.

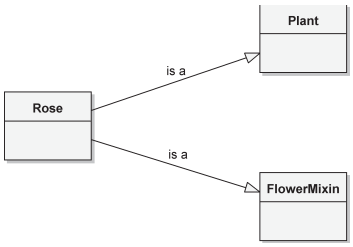


Рис. 2.10. Класс `Rose` — наследник нескольких суперклассов

Точно так же можно определить класс `Carrot`, продемонстрированный на рис. 2.11. В обоих случаях подкласс образуется путем наследования свойств двух суперклассов.

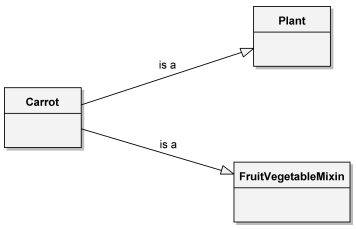


Рис. 2.11. Класс `Carrot` — наследник нескольких суперклассов

Определим теперь класс, описывающий свойства вишни, которая цветет и плодоносит (рис. 2.12).

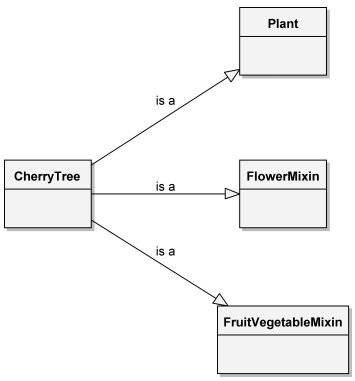


Рис. 2.12. Класс `CherryTree` — наследник нескольких суперклассов

Несмотря на то что множественное наследование представляет собой довольно простую концепцию, оно создает некоторые сложности для языков программирования — конфликты имен между различными суперклассами и повторное наследование. Конфликты имен возникают, когда в двух или большем числе суперклассов определены поля или операции с одинаковыми именами.

Повторное наследование возникает, когда несколько суперклассов наследуют свойства общего суперкласса. В таких ситуациях возникает ромбическая структура наследования, и проектировщик должен решить, сколько копий полей должен унаследовать класс самого нижнего уровня у суперкласса самого верхнего уровня — одну или две (см. рис. 2.13). В некоторых языках повторное наследование запрещено, в других конфликт разрешается однозначно, а в языке C++ решение должен принять сам программист. В языке C++ для запрещения дублирования повторяющихся структур используются виртуальные базовые классы, иначе в подклассе могут появиться дубликаты (для которых потребуется явное указание происхождения каждой).

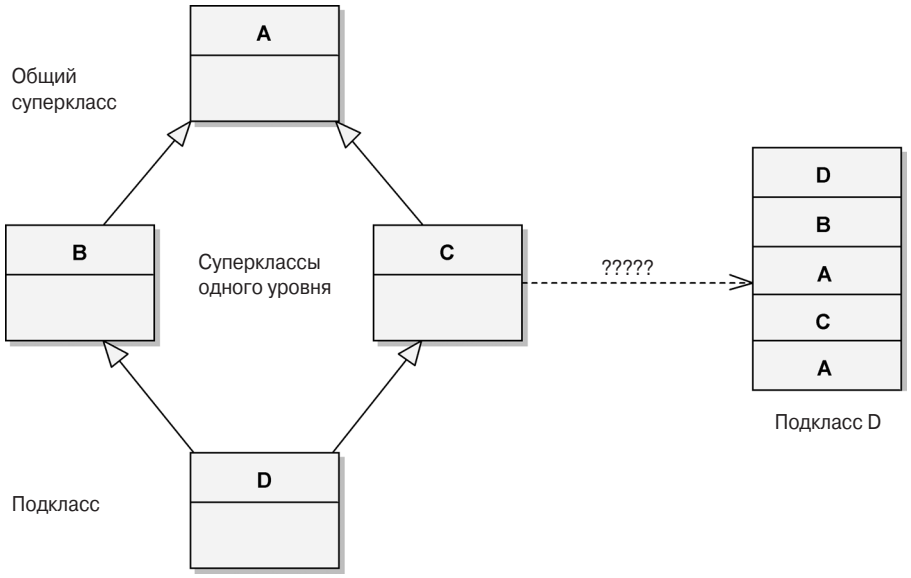


Рис. 2.13. Повторное наследование

Множественным наследованием часто злоупотребляют. Например, сладкая вата — это разновидность сладости, но никак не ваты. К этой ситуации можно применить описанный ранее критерий: если класс `B` не является разновидностью класса `A`, то он не должен быть наследником класса `A`. Часто плохо продуманные структуры множественного наследования следует свести к одному суперклассу, по возможности используя агрегацию других классов подклассами.

Примеры иерархии: агрегация

Если иерархии “общее/частное” определяют отношение “обобщение/специализация”, то иерархии “целое/часть” описывают отношения агрегации. Рассмотрим, например, абстракцию огорода. Можно утверждать, что огород — это совокупность растений и план их выращивания. Иначе говоря, растения и план выращивания — это части огорода. Это отношение “является частью” называется *агрегацией* (aggregation).

Агрегация есть во всех объектно-ориентированных проектах и языках программирования. Действительно, любой язык программирования, имеющий структуры, подобные записям, поддерживает агрегацию. Однако в сочетании с наследованием агрегация создает мощный инструмент: она позволяет физически объединить логически связанные структуры, а наследование — без труда повторно использовать эти общие группы в разных абстракциях.

Анализируя иерархии, подобные рассмотренным выше, часто говорят об уровнях абстракции, концепцию которых впервые предложил Дейкстра [60]. В терминах иерархии “общее/частное” абстракция верхнего уровня является обобщенной, а абстракция нижнего уровня — специализированной. Следовательно, можно сказать, что класс `Flower` относится к более высокому уровню абстракций, чем класс `Plant`. В терминах иерархии “целое/часть” класс относится к более высокому уровню абстракции по сравнению с любыми классами, входящими в его реализацию. Таким образом, класс `Garden` относится к более высокому уровню абстракции, чем класс `Plant`, на основе которого он создается.

Кроме того, агрегация поднимает проблему владения. Абстракция огорода допускает, что на огороде одновременно растет много растений, но вследствие замены одного растения другим огород не становится другим огородом, а уничтожение огорода не обязательно означает уничтожение всех растений (ведь их можно пересадить в другое место). Иначе говоря, жизненные циклы огорода и растений не зависят друг от друга. И наоборот, проектное решение предусматривает, что объект `GrowingPlan` неразрывно связан с объектом `Garden` и не существует независимо от него. Следовательно, план выращивания может храниться в каждом экземпляре класса `Garden` и уничтожается вместе с ним.

Контроль типов

Понятие типа происходит из теории абстрактных типов данных. Дейч (Deutsch) утверждает: “Тип — это точная характеристика структуры и поведения, присущих некоторой совокупности объектов” [61]. В дальнейшем будем считать, что тер-

мины тип и класс являются синонимами[2]. Несмотря на сходство понятий типа и класса, контроль типов следует обсудить как отдельный элемент объектной модели, поскольку концепция типа позволяет взглянуть на абстракцию с совершенно другой точки зрения.

> Контроль типов — это правила использования объектов, не допускающие или ограничивающие взаимную замену объектов разных классов.

Контроль типов заставляет проектировщиков выражать свои абстракции так, чтобы язык программирования, используемый для реализации системы, поддерживал принятые проектные решения.

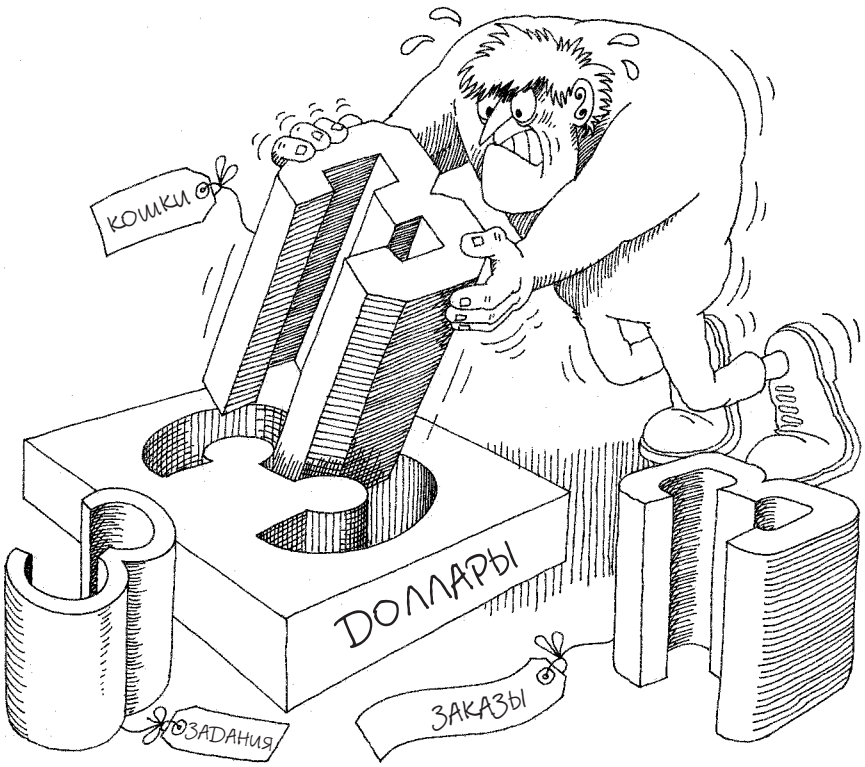
Конкретный язык программирования может иметь сильный или слабый контроль типов, и даже совсем не иметь такого свойства, оставаясь объектно-ориентированным. Например, язык Eiffel предусматривает строгий контроль типов: операция не может быть применена к объекту, если ее точная сигнатура не содержится в его классе или суперклассе.

Основным понятием контроля типов является соответствие (conformance). Рассмотрим, например, абстракцию физической единицы измерения [63]. Если разделить расстояние на время, мы получим число, означающее скорость, а не вес. Аналогично, деление единицы силы на температуру бессмысленно, а деление единицы силы на массу — нет. Эти примеры относятся к строгому контролю типов, поскольку правила исследуемой предметной области четко определены и регламентируют допустимые сочетания абстракций.

Строгий контроль позволяет использовать язык программирования для поддержки определенных проектных решений и позволяет справиться с нарастающей сложностью систем. Однако у строгого контроля типов есть обратная сторона. На практике она порождает семантические зависимости, при которых небольшое изменение интерфейса в базовом классе вынуждает повторную компиляцию всех его подклассов.

Существуют два общих решения этих проблем. Во-первых, можно использовать контейнерный класс, обладающий типовой безопасностью (type-safe). Это подход позволяет решить первую проблему, избегая смешивания объектов разных типов. Во-вторых, можно использовать идентификацию типов на этапе выполнения программы. Это позволяет решить вторую проблему, определяя тип объекта в определенный момент. Однако идентификацию типов на этапе выполнения про-

[2]Тип и класс не вполне одно и то же. В некоторых языках эти два понятия отличаются друг от друга. Например, в ранних версиях языка Trellis/Owl объект может иметь тип и принадлежать классу. Даже в языке Smalltalk объекты классов `SmallInteger`, `LargeNegativeInteger`, `LargePositiveInteger` относятся к одному типу `Integer`, но к разным классам [62]. Большинству смертных совершенно не обязательно различать типы и классы. Достаточно сказать, что класс реализует некий тип.



Строгий контроль типов предотвращает смешение абстракций

граммы следует использовать только, если для этого есть непреодолимая причина, поскольку ослабляется инкапсуляция. Как будет показано в следующем разделе, вместо идентификации типов на этапе выполнения программы можно (но не всегда) использовать полиморфные операции.

Как указывает Теслер (Tesler), языки программирования со строгим контролем типов имеют ряд преимуществ.

- В отсутствие контроля типов работа программ в большинстве языков может завершиться непредсказуемо.
- В большинстве систем цикл редактирование–компиляция–отладка настолько утомителен, что раннее обнаружение ошибок крайне желательно.
- Объявление типов облегчает документирование программ.
- Многие компиляторы способны генерировать более эффективный объектный код, если типы объявлены явно [64].

Языки, в которых контроль типов отсутствует, обладают большей гибкостью, но и в этом случае, по мнению Борнинга (Borning) и Ингаллса (Ingalls), “программисты практически всегда знают, какие объекты ожидаются в качестве аргументов

сообщения и какие подлежат возвращению" [65]. На практике безопасность, обеспечиваемая языками программирования со строгим контролем типов, с лихвой компенсирует потерю гибкости, присущей языкам без контроля типов, особенно при проектировании крупномасштабных систем.

Примеры контроля типов: статический и динамический

Сильный (strong) и *слабый* (weak) контроль типов, с одной стороны, и *статический* (static) и *динамический* (dynamic) контроль типов, с другой стороны, — это разные вещи. Строгий контроль типов означает *соответствие типов* (type consistency), а статический контроль типов (иначе называемый *статическим*, или *ранним связыванием* (static, or early binding)), определяет момент времени на этапе компиляции, в который фиксируется тип переменных или выражений. Динамический контроль типов (называемый также *поздним связыванием* (late binding)) означает, что типы переменных и выражений до момента выполнения программы остаются неизвестными. Язык программирования может одновременно предусматривать сильный и статический контроль типов (Ada), сильный и динамический контроль типов (C++, Object Pascal), или не предусматривать контроль типов, но допускать динамическое связывание (Smalltalk).

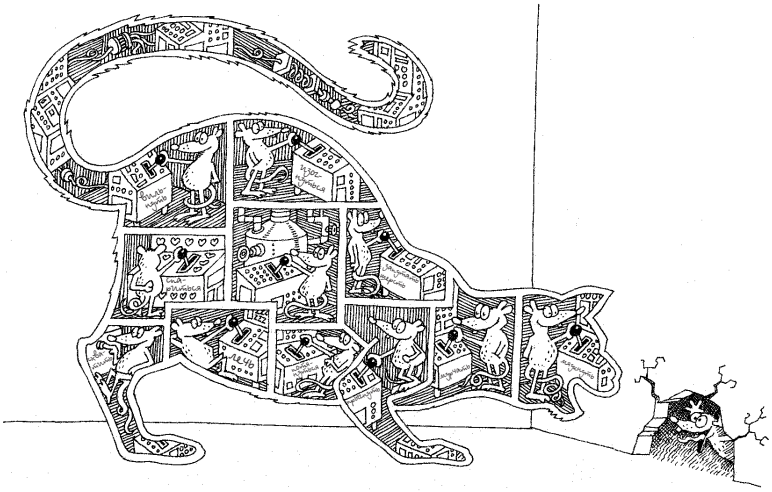
Наследование и динамическое связывание порождают *полиморфизм* (polymorphism). Эта концепция теории типов позволяет одним именем (например, одним объявлением переменной) обозначать объекты, относящиеся к разным классам, но связанные с общим суперклассом. Любой объект, обозначенный таким именем, может обладать общим набором операций [66]. Противоположностью полиморфизма является *мономорфизм* (monomorphism), присущий всем языкам программирования, предусматривающим сильный и статический контроль типов.

Полиморфизм является одним из наиболее мощных свойств объектно-ориентированных языков программирования, уступая по степени важности лишь поддержке абстракции. Именно полиморфизм отличает объектно-ориентированное программирование от традиционного программирования с абстрактными типами данных. Как будет показано в следующих главах, полиморфизм является центральным понятием в объектно-ориентированном проектировании.

Параллелизм

Существуют задачи, решая которые, автоматические системы должны обрабатывать много разных событий одновременно. Кроме того, решение некоторых задач требует вычислительной мощности, превышающей ресурсы одного процессора. В таких случаях естественно рассмотреть возможность использования сети распределенных компьютеров или перейти в многозадачный режим. Отдельный процесс — это источник независимого динамического действия внутри системы. Каждая программа имеет по крайней мере один поток управления, но в парал-

лельной системе имеется много таких потоков: как кратковременные, так и долгосрочные. Реальная параллельность достигается только на многопроцессорных системах, а системы с одним процессором могут лишь имитировать параллельность за счет алгоритма разделения времени.



Параллелизм позволяет различным объектам действовать одновременно

Кроме того, следует различать "*тяжеловесный*" (heavyweight) и "*легковесный*" (lightweight) параллелизм. Тяжеловесные процессы обычно управляются операционной системой независимо от других и могут иметь свое собственное адресное пространство. Легковесные процессы совместно существуют в рамках отдельной операционной системы, используя одно и то же адресное пространство. Связь между тяжеловесными процессами, как правило, реализуется посредством межпроцессорного взаимодействия, которое является слишком затратным. Связь между легковесными процессами требует меньшего расхода ресурсов и часто реализуется с помощью общего набора данных.

Создание крупного программного обеспечения — довольно трудная задача сама по себе, а разработка параллельной системы, в которой существуют несколько потоков управления, — еще сложнее, поскольку необходимо решать задачи, связанные с взаимной блокировкой (deadlock), активными тупиками (livelock), зависанием процессов (starvation), взаимным исключением доступа (mutual exclusion) и конкуренции между процессами (race condition). "На высшем уровне абстракции объектно-ориентированное программирование может ослабить проблему параллельности для большинства программистов, скрыв ее внутри повторно используемых абстракций" [67]. Блэк (Black) и соавторы развили эту мысль, заявив, что "объектная модель вполне приемлема для распределенных систем, поскольку

неявно она определяет 1) единицы распределения и движения и 2) взаимосвязанные сущности" [68].

В то время как объектно-ориентированное программирование сосредоточивает внимание на абстракции, инкапсуляции и наследовании данных, параллелизм делает акцент на абстракции и синхронизации процессов [69]. Объект — это понятие, объединяющее две точки зрения: каждый объект (построенный на основе абстракции реального мира) может представлять собой отдельный поток управления (абстракцию процесса). Такой объект называется *активным* (active). Система, построенная на основе объектно-ориентированного проектирования, может быть представлена как совокупность взаимодействующих объектов, часть из которых является активной и независимой от других. Следуя этому подходу, можно сформулировать такое определение параллелизма.

> Параллелизм — это свойство, отличающее активные объекты от пассивных.

Примеры параллелизма

Рассмотрим датчик `ActiveTemperatureSensor`, который периодически измеряет температуру и сообщает клиенту о том, что она отклонилась от установленного значения на определенное количество градусов. Несмотря на то что реализация этого класса пока неизвестна, очевидно, что здесь требуется параллелизм.

Существуют три подхода к параллелизму в объектно-ориентированном проектировании. Во-первых, параллелизм — это внутреннее свойство некоторых языков программирования, имеющих механизмы параллельности и синхронизации. В данном случае можно создать активный объект, выполняющий некий процесс параллельно с другими объектами.

Во-вторых, можно использовать библиотеку классов, реализующих какую-нибудь разновидность легковесных процессов. Естественно, такая реализация сильно зависит от платформы, хотя интерфейс библиотеки может быть универсальным. В этом случае параллелизм не является частью языка программирования (благодаря этому он не перегружает непаралелльные системы), а лишь выглядит таким, поскольку присутствует в стандартных классах.

В-третьих, параллелизм можно имитировать с помощью прерываний. Разумеется, для этого необходимы определенные знания низкоуровневого аппаратного обеспечения. Например, в реализации класса `ActiveTemperatureSensor` можно предусмотреть аппаратный таймер, периодически прерывающий выполнение приложения для того, чтобы все датчики измерили текущую температуру и активизировали свои функции обратного вызова.

Независимо от выбранного подхода включение параллелизма порождает проблему синхронизации работы как активных, так и последовательных объектов.

Например, если два объекта пытаются послать сообщения третьему, необходимо использовать определенный механизм взаимной блокировки, гарантирующий, что объект, на который направлено действие, не будет разрушен при одновременной попытке двух активных объектов изменить его состояние. Для решения этой проблемы необходимо использовать идеи абстракции, инкапсуляции и параллелизма. В параллельных системах недостаточно определить методы объекта — необходимо также принять меры для сбережения семантики этих методов в присутствии нескольких потоков управления.

Персистентность

Любой программный объект занимает определенный объем памяти и существует определенное время. Аткинсон (Atkinson) и соавторы считают, что множество способов существования объектов неисчерпаемо: от промежуточных объектов, возникающих лишь во время вычисления выражений, до постоянных баз данных, существующих даже после завершения работы программы. Персистентность [3] объектов может проявляться в разном виде.

Промежуточные результаты вычисления выражений.

- Локальные переменные в вызове процедур.
- Собственные переменные (как в ALGOL-60), глобальные переменные и динамические данные, экстент (extent) которых не совпадает с их областью видимости.
- Данные, сохраняющиеся между разными сеансами выполнения программы.
- Данные, сохраняющиеся при переходе от одной версии программы к другой.
- Данные, существующие после исчезновения программы [70].

Первые три вида персистентности характерны для традиционных языков программирования, а последние — для баз данных. Этот конфликт технологий иногда приводит к неожиданным архитектурным решениям: программисты разрабатывают специальные схемы для хранения объектов, состояние которых должно сберегаться в период между запусками программы, а конструкторы баз данных вынуждены использовать свою технологию для управления промежуточными объектами [71].

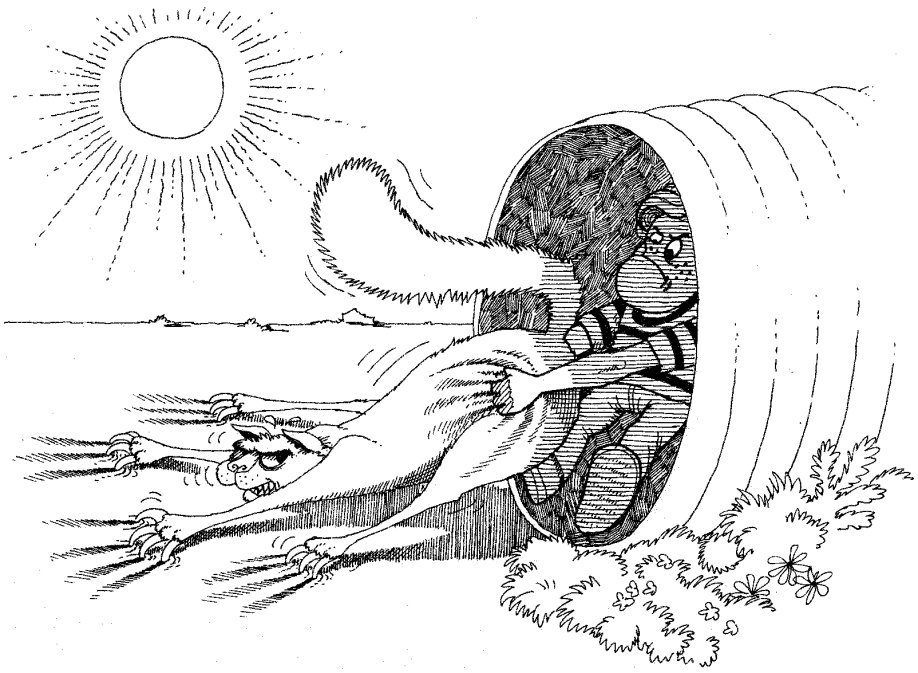
[3] Слово *persistence* очень многозначно, поэтому часто переводится по-разному — то как *устойчивость*, то как *сохраняемость*. И то, и другое не совсем верно, поскольку в первом случае возникает иллюзия, что речь идет о прочности объектов, а во втором — о возможности его сохранения на запоминающих устройствах. Данное в тексте определение абсолютно точно указывает на то, что это слово означает сохранение эффекта после устранения вызвавшей его причины. Это явление называется *персистентностью. — Примеч. ред.*

Объекты, которые Аткинсон и его соавторы назвали “данными, существующими дольше программы”, нашли интересное применение при создании Web-приложений, которые могут не иметь связи с источником данных, используемых на протяжении всего периода выполнения транзакции. Какие изменения могут произойти с данными, предоставленными клиентскому приложению или Web-сервису, не имеющими связи с источником данных, и как решить эту задачу? Для реализации таких распределенных и несвязных сценариев были разработаны особые технологии, такие как ActiveXData Object для среды .NET (ADO .NET) компании Microsoft.

Сочетание параллелизма и объектов привело к созданию параллельных объектно-ориентированных языков программирования. Точно также внедрение принципа персистентности в объектную модель привело к появлению объектно-ориентированных баз данных. На практике подобные базы данных создаются с помощью проверенных временем технологий — последовательных, индексированных, иерархических, сетевых или реляционных моделей, но наряду с этим программисты могут использовать абстракцию объектно-ориентированного интерфейса, позволяющего выполнять запросы к базе данных и другие операции над объектами объектов, время жизни которых превосходит время жизни отдельной программы. Такое сочетание разных подходов значительно упрощает разработку отдельных видов приложений. В частности, это позволяет применить одни и те же методы проектирования к сегментам программы, одни из которых связаны с базами данных, а другие — нет.

Некоторые объектно-ориентированные языки программирования непосредственно поддерживают персистентность. Например, язык Java поддерживает технологии Enterprise Java Beans (EJBs) и Java Data Objects. В языке Smalltalk существуют протоколы для записи объектов на диск и считывания их с диска. Однако записывать объекты в обычные файлы довольно наивно. Этот способ плохо масштабируется. Иногда персистентность обеспечивается за счет коммерческих объектно-ориентированных баз данных [72]. Однако намного чаще для этого используются объектно-ориентированные оболочки реляционных баз данных. Объектно-реляционные базы данных могут создавать также индивидуальные разработчики. Однако это очень непростая задача. Для облегчения ее решения появились специальные интегрированные среды, такие как Hibernate, распространяемые с открытым текстом [76].

Персистентность относится не только к продолжительности существования данных. В объектно-ориентированных базах данных после завершения работы сохраняются не только объекты, но и классы, чтобы все программы могли интерпретировать сохраненное состояние единообразно. Очевидно, что это затрудняет поддержку целостности базы данных по мере ее увеличения, особенно если класс объекта необходимо изменить.



Персистентность — это свойство сохранять состояние и класс объекта во времени и пространстве

До сих пор персистентность рассматривалась как способ существования объектов во времени. Это объясняется тем, что в большинстве систем однажды созданный объект занимает постоянный объем памяти, пока не будет уничтожен. Однако в распределенных системах персистентность можно рассматривать как способ существования объектов в пространстве. В таких системах полезно рассматривать объекты, которые могут перемещаться с одной машины на другую, изменяя свое представление.

Итак, можно сформулировать следующее определение персистентности.

> Персистентность — это способность объекта преодолевать временны́е рамки (т.е. продолжать свое существование после исчезновения своего создателя) или пространственные пределы (т.е. выходить за пределы своего первоначального адресного пространства).

2.4 Применение объектной модели

Как указывалось ранее, объектная модель принципиально отличается от традиционных методов структурного анализа, проектирования и программирования. Это не означает, что объектная модель отбрасывает все испытанные временем

принципы и накопленный опыт. Скорее, она обогащает существующие модели новыми элементами. Таким образом, объектная модель обеспечивает ряд существенных преимуществ, которые невозможно достичь в рамках других моделей. Наиболее важно то, что объектная модель позволяет создавать системы, у которых есть пять признаков хорошо структурированных сложных систем, перечисленных в главе 1: иерархическая структура, относительность выбора элементарных компонентов (например, несколько уровней абстракций), разделение функций, общая структура и устойчивые промежуточные формы. Опыт показывает, что объектная модель имеет еще пять практических преимуществ.

Преимущества объектной модели

Во-первых, объектная модель позволяет эффективно использовать выразительные возможности объектных и объектно-ориентированных языков программирования. Страуструп (Stroustrup) утверждает: “Не всегда понятно, как наилучшим образом использовать преимущества такого языка, как C++. Существенно повысить эффективность и качество кода можно просто за счет использования C++ в качестве “улучшенного C” с элементами абстракции данных. Однако гораздо более значительным достижением является использование иерархии классов в процессе проектирования. Именно это называется объектно-ориентированным проектированием и именно здесь язык C++ позволяет достичь наибольших результатов” [73]. Опыт показывает, что отказ от объектной модели приводит к игнорированию или неправильному использованию наиболее мощных средств таких языков программирования, как Smalltalk, C++ и Java.

Во-вторых, использование объектного подхода стимулирует повторное использование не только кода, но и проектных решений [74]. Объектно-ориентированные системы компактнее, чем их традиционные эквиваленты. Компактность означает не только уменьшение размера программ, но и удешевление и ускорение проекта за счет повторного использования проектных решений. Однако повторное использование кода, как правило, не возникает само по себе, если оно не было основной целью проекта. Кроме того, первоначальная разработка кода, предназначенного для повторного использования, может оказаться более дорогой, чем обычно. Однако эти первоначальные затраты впоследствии будут компенсированы за счет экономии средств и времени при использовании в новых проектах.

В-третьих, использование объектной модели приводит к созданию систем с устойчивыми промежуточными формами, что упрощает их изменение. Это позволяет улучшать систему со временем, не прибегая к ее полной переделке при первой же серьезной модификации.

В-четвертых, как показано в главе 7, объектная модель уменьшает риски, связанные с проектированием сложных систем. Прежде всего, этому способствует то, что процесс объединения компонентов системы в одно целое растягивается на

весь период ее разработки, а не является единовременным событием. Объектный подход к проектированию подразумевает разумное разделение функций, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

Наконец, объектная модель учитывает особенности процесса познания. Робсон (Robson) утверждает: “Многие люди, не имеющие представления о работе компьютера, считают объектно-ориентированные системы вполне естественными” [75].

Нерешенные вопросы

Для того чтобы эффективно использовать объектный подход, необходимо ответить на следующие вопросы.

- Что такое классы и объекты?
- Как правильно идентифицировать классы и объекты, относящиеся к конкретному приложению?
- Как выбрать правильную систему обозначений для описания проекта объектно-ориентированной системы?
- Как создать хорошо структурированные объектно-ориентированные системы?
- Как организовать управление объектно-ориентированным проектом?

Резюме

- Развитие технологии привело к созданию методов объектно-ориентированного анализа, проектирования и программирования, позволяющих создавать крупномасштабное программное обеспечение.
- Существует несколько парадигм программирования: процедурно-ориентированная, объектно-ориентированная, логико-ориентированная, ориентированная на правила и ориентированная на ограничения.
- Абстракция выделяет существенные характеристики объекта, отличающие его от всех других видов объектов и, таким образом, проводит четкие концептуальные границы объекта с точки зрения наблюдателя.
- Инкапсуляция — это процесс выделения элементов абстракции, образующих ее структуру и поведение; инкапсуляция позволяет отделить контрактный интерфейс абстракции от ее реализации.
- Модульность — это свойство системы, разложенной на целостные и слабо связанные между собой модули.

- Иерархия — это ранжирование или упорядочение абстракций.
- Контроль типов — это система правил, предотвращающих или сильно ограничивающих взаимную замену объектов разных классов.
- Параллелизм — это свойство, отличающее активные объекты от пассивных.
- Персистентность — это способность объекта преодолевать временны́е или пространственные пределы.