1

Основные понятия

Мои объекты вечны,
Их нужно сохранить.

В. С. Гилберт. *Микадо*

В 1990-х годах словосочетание “объектно-ориентированный” в контексте информационных технологий стало синонимом слов “современность”, “высокое качество”, “ценность”. Некоторое время спустя слово “объект” стало заменяться словом “компонент”, хотя и в несколько ином значении. Поскольку мы склонны к преувеличению достоинств этой технологии, желательно было бы добиться более объективной точки зрения. Эта книга призвана сформулировать принципы объектно-ориентированной и модульной разработок в корректных терминах. Кроме того, она должна показать, как эти абстрактные идеи могут превратиться в реальные, практические советы по построению полезных компьютерных систем. Описываемый подход не зависит от конкретного языка программирования и не охватывает вопросы синтаксических предпочтений и выбора среды разработки. При использовании объектно-ориентированных сущностей нужно также понимать, что мы вступаем в область, уже имеющую некоторые стандарты. Однако в настоящее время ее границы еще не определены, а исследования остаются незавершенными. Непременно нужно рассмотреть множество других сфер, связанных с данной, таких как архитектура программного обеспечения, распределенные открытые системы, системы баз данных, CASE-технологии, экспертные системы и т.д. Придется также столкнуться и со сравнительно неизвестными сферами.

Ключевые преимущества объектно-ориентированного подхода — это возможность повторного использования и расширяемость, т.е. объектно-ориентированные системы могут быть легко собраны из ранее написанных компонентов. Эти системы будут легко расширяться без какой-либо модернизации повторно используемых компонентов. В данной главе рассмотрены оба указанных преимущества, а также область применения объектно-ориентированных систем.

Объектно-ориентированным методом в этой книге называется нечто большее, чем просто объектно-ориентированное программирование. В это понятие включается философия разработки систем программирования, извлечение знаний, анализ требований, моделирование предметной области, проектирование систем, проектирование баз данных и многие другие вопросы. Основное внимание будет обращаться на указанную философию и ее использование для решения проблем, возникающих при создании информационных систем. Например, такие преимущества, как возможность повторного использования и расширяемость, не ограничиваются повторным использованием или дополнением частей кода. Потенциально документы по проектированию и анализу могут храниться в библиотеках и повторно использоваться или расширяться снова и снова, если, разумеется, пользователи смогут их легко отыскать.

Как уже отмечалось, в термин "объектно-ориентированный" вкладывают слишком общий смысл. Чтобы лучше понять данный вопрос, в этой книге будут различаться понятия "объектно-ориентированное" (object-oriented), "компонентно-ориентированное" (component-oriented), "основанное на объектах" (object-based) и "основанное на классах" (class-based) программирование, а также их разработка и анализ. В следующих главах будет показано, что некоторые коммерческие системы действуют согласно объектно-ориентированным принципам, но имеют некоторые недостатки. Тем не менее объектно-ориентированные системы на самом деле существуют, и важно не то, является ли система или язык объектно-ориентированным, а то, *как* они реализованы.

В этой главе вводятся основные понятия и терминология объектно-ориентированных методов. Вначале приводится краткая историческая справка. Мотивация и преимущества объектно-ориентированных методов и программирования будут изложены в главе 2.

1.1. Историческая справка

История развития объектно-ориентированного подхода отражает и повторяет историю вычислительной техники в целом. Начнем с 1940-х годов, когда первые работы по вычислительной технике были связаны исключительно с тем, что в настоящее время называют программированием. Только позже выделились проектирование и анализ. Точно так же первым привлекло к себе внимание объектно-ориентированное программирование, позже появилось объектно-ориентированное проектирование, а еще позже объектно-ориентированный анализ. Таким образом, эта книга начинается с описания объектно-ориентированного программирования, затем рассматриваются вопросы разработки и анализа, хотя далее основное внимание будет уделено именно последнему.

Хотя Тен Дайк (Ten Dyke) и Канц (Kunz) объявили, что разработчики ракет Minuteman использовали элементарные объектно-ориентированные методы еще в 1957 году, история объектно-ориентированного программирования на самом деле началась в Норвегии в 1967 году. Однако с развития языка программирования Simula, основанного на языке ALGOL и более раннем языке моделирования дискретных событий Simula 1, и продолжался использоваться в 1970-х годах объектный подход параллельно с языком Smalltalk, который сделал понятие "объект" объектом поклонения. Важными промежуточными этапами были языки Alphard [810] и CLU [490]. Ст*о*ит отметить, что в объектно-ориентированном языке Simula были представлены все понятия структурного программирования. С тех пор было создано много языков, которые были порождены этими разработками и получили название "объектно-ориентированных".

Имитационное моделирование — трудная задача для программистов, использующих обычные языки третьего поколения. От программистов требуется адаптировать функциональный поток управления, обычный для таких языков, к потоку управления, который естественнее описывается через составные объекты, изменяющие свое состояние, и происходящие события. В объектно-ориентированном программировании функциональный поток заменяется передачей сообщений между объектами, которые вызывают изменения состояния. Таким образом, объектно-ориентированное программирование — это крайне естественный подход, поскольку структура программ непосредственно отражает структуру задачи. Более того, в моделируемых задачах обычно понятно, что является объектами. В частности, это могут быть машины на улице, механизмы производственной линии и т.д. Они обычно являются "реальными", а не абстрактными объектами, и как таковые, их легче определить. К сожалению, как будет показано далее, это не всегда справедливо для коммерческих приложений.

Smalltalk и GUI

Термин "объектно-ориентированный" окончательно утвердился с появлением языка программирования Smalltalk, который был создан в исследовательском центре Xerox в Пало Альмо. Он основывался не только на языке Simula, но и на диссертации Алана Кая (Alan Кау), выполненной в университете штата Юта. Как отмечено в работе [659], эти исследования были основаны на представлении маленького, но универсального персонального компьютера, способного решать любые задачи управления информацией. Первой версией подобного устройства была машина Flex, которая получила название Dynabook. Smalltalk — это, по сути, программное обеспечение для Dynabook, сильно зависимое от понятия классов языка Simula, а также понятия наследования и структурных особенностей языка Lisp[1]. Smalltalk объединил понятие класса из Simula с набором функциональных абстракций, подобных Lisp, хотя, как языки программирования, Smalltalk и Lisp достаточно не похожи.

Следующий этап, который пришелся на 1980-е годы, продемонстрировал рост интереса к интерфейсу пользователя (UI). Самый известный пионер в области коммерческих продуктов, компания Xerox, а позже и компания Apple заинтересовали мир удобным WIMP[2]-интерфейсом, и много идей в Smalltalk прочно связано с этими разработками. С одной стороны, объектно-ориентированное программирование способствовало разработке таких интерфейсов — особенно это касается Lisa и Macintosh от компании Apple — а с другой, стиль объектно-ориентированных языков находился под влиянием идеологии WIMP. Наиболее очевидное следствие этого — это большое (по сравнению с другими сферами) множество библиотечных объектов для разработки интерфейса. Одной из основных причин успеха объектно-ориентированного программирования была сложность таких интерфейсов и сопутствующая этому высокая цена реализации. Можно предположить, что без присущей объектно-ориентированному коду возможности повторного использования было бы невозможно обеспечить столь широкое распространение таких интерфейсов. Например, известно, что для разработки Apple Lisa, предшественника Macintosh, потребовалось более 200 человеко-лет

[1] Lisp (LISt Processing) означает "обработка списков". Этот язык, изначально разработанный Джоном Маккарти приблизительно в 1958 году, стал основным языком реализации для многих ранних работ в области искусственного интеллекта.

[2] WIMP — это сокращение от Windows, Icons, Menus (иногда Mice), Pointers, которое обозначает стиль графического интерфейса пользователя (Graphical User Interface — GUI).

работы, большая часть которой ушла на разработку интерфейса. На протяжении этого периода влияние WIMP-стиля было настолько сильным, что все терминалы были постепенно оборудованы внешними интерфейсами WIMP, такими как Microsoft Windows, или им подобными. С учетом курса на стандартизацию в открытых системах на основе UNIX также учитывались аспекты UI, причем в качестве немаловажного фактора выступала конкурентная борьба OpenLook, OSF Motif и других подобных программ. В этом смысле объектно-ориентированный подход наложил свой отпечаток на отрасль компьютеризации и определил тот внешний вид экрана, какой мы привыкли видеть с 1993 года.

Влияние искусственного интеллекта

Начиная с середины 1970-х годов наблюдается значительная взаимосвязь между объектно-ориентированным программированием и исследованием и разработкой систем искусственного интеллекта (ИИ), что привело к появлению нескольких важных языков искусственного интеллекта, в частности Lisp. Именно благодаря этому появился Lisp со средствами Flavors, Loops и CLOS (Common Lisp Object System). Идеи объектно-ориентированного подхода оказали свое влияние на среду программирования ИИ (в том числе расширения Lisp, такие как KEE и ART). На эти системы также сильно повлияли методы представления информации, основанные на семантических сетях и фреймах. Эти структуры выражают знания о реальных объектах и понятиях в виде сетей стандартных объектов, которые могут наследовать свойства от своих родителей. Таким образом, основным вкладом этого расширения объектно-ориентированного подхода стала строгая теория наследования. Объектно-ориентированные методы до сих пор основываются на этих результатах. Языки искусственного интеллекта будут обсуждаться позже, в главе 3.

Еще одно направление исследований в области ИИ, наряду с работами в сфере параллельных вычислений, привело к появлению понятия *исполнитель* (actor). Системы с использованием исполнителей [6], в том числе системы доски объявлений [261], — это попытка смоделировать работу групп сотрудников или экспертов. Исполнитель — это более антропоморфное понятие, чем объект, для которого определены обязанности, потребности и знания о взаимодействии с другими исполнителями. Языки, использующие понятие исполнителя, обычно предназначены для параллельных приложений реального времени. Родственное современное понятие — это интеллектуальные исполнители (или агенты). Понятно, что современные компонентно-ориентированные средства разработки, такие как технология COM+, требуют знания компонентов о возможных участниках взаимодействия. Более подробно об этом будет сказано в главах 6 и 7.

Новые языки

По сравнению с коммерческими приложениями, разработка пользовательских интерфейсов не порождает серьезных проблем, связанных с управлением данными. Вследствие того, что основное внимание обращалось на моделирование проблем и проектирование пользовательского интерфейса, а не на, скажем, управление базами данных, характерным свойством ранних языков программирования была проблема производительности. Это привело к разработке новых языков, таких как Eiffel, и расширению имеющихся, удобных и эффективных языков, таких как Ada, C и PASCAL. Такое смещение внимания также означало, что языки объектно-ориентированного программирования часто не предоставляли средств работы с постоянными объектами, параллелизмом и т.д. Одним из средств решения данной проблемы

стала разработка систем объектно-ориентированных баз данных. Детально этот вопрос будет обсуждаться в главе 5. Некоторые объектно-ориентированные и объектные языки программирования обзорно рассмотрены в главе 3.

Требования многих пользователей, а особенно таких финансово влиятельных, как министерство обороны (DoD) Соединенных Штатов Америки, часто приводят к тому, что промышленные компании вынуждены менять курс. Требования DoD на протяжении 1960–1970-х годов затрагивали три основных аспекта: инженерный подход к практической разработке программного обеспечения, который выразился в так называемых "структурных" методах; возможность повторного использования компонентов программного обеспечения (модульность); и открытость систем. Разнообразные методы разработки систем, принятые многими основными поставщиками информационных технологий, — это реакция на первое требование. В настоящее время (после бума, произведенного CASE-средствами в конце 1980-х) в этой области наблюдается некоторое затишье. Системы UNIX, X/Windows и Ada — все это в некотором роде реакция на требование DoD относительно "открытости" систем. Это требование означает минимизацию затрат на взаимодействие с любым программным обеспечением или аппаратными средствами. Язык Ada также характеризуется улучшенной модульностью. Позже мы оценим, насколько язык Ada является объектно-ориентированным. А сейчас нас интересует, вносит ли он что-то (и объектно-ориентированное программирование вообще) в перечисленные три ключевых вопроса. Объектно-ориентированное программирование также неявно входит в перечисленные выше ключевые требования DoD. Оно дает возможность строить системы с использованием повторно используемых компонентов, объединяя таким образом в единое целое модульность и программную инженерию. С появлением разного рода промежуточного программного обеспечения, ориентированного на сообщения, и стандартов передачи сообщений, таких как CORBA (см. главу 4), возможность взаимодействия в рамках действительно распределенных систем расширяется еще больше.

В процессе становления объектно-ориентированного программирования интерес сместился к объектно-ориентированным методам проектирования и анализа (или описания). Преимущества повторного использования и расширяемости можно рассматривать в ракурсе проектирования и описания, а не только программирования. Авторы работ [76], [643] и [723] соглашаются, что в общем контексте программной инженерии, чем выше уровень повторного использования, тем лучше. Это приводит к возникновению важных вопросов. Должно ли объектно-ориентированное проектирование реализовываться в объектно-ориентированном языке? Должны ли методы проектирования быть связаны с определенными языками?

Новые базы данных и CASE-средства

В начале 1990-х годов, когда реляционные базы данных стали общепринятой (если не необходимой) технологией реализации коммерческих продуктов, их основные производители стали разрабатывать различные "постреляционные расширения" своих продуктов, предназначенные для таких отраслей, как экспертные системы, функциональное программирование, а позже и объектно-ориентированное программирование. Появились коммерческие продукты объектно-ориентированных и полуобъектно-ориентированных баз данных, а теоретические вопросы объектно-ориентированного подхода сместились в сторону традиционных вопросов баз данных. Например, стали актуальными задачи эффективного управления перманентными объектами, кэширования и контроля версий объектов. Эти разработки можно интерпретировать как процесс упорядочения объектно-ориентированного подхода. Возникло также

множество вопросов, касающихся относительной эффективности декларативных языков реляционных запросов по сравнению с подходами, основанными на объектной идентичности. По иронии судьбы последнее свойство относится не только к объектно-ориентированным базам данных, но и к ранним сетевым и иерархическим системам. Недавно стало ясно, что существует фундаментальное различие между двумя подходами: чисто объектно-ориентированными базами данных и смешанными объектно-реляционными системами. Фактически это различие породило два стандарта языков запросов: OQL и SQL3. В главе 5 будет подробнее рассмотрена теория баз данных, а также будут изучены объектно-ориентированные базы данных и другие вопросы.

CASE-средства (computer-aided software engineering) автоматизированного проектирования и создания программ становятся все больше необходимы в разработке коммерческих систем. Одни на это смотрят с восторгом, другие — скептически. Появление многочисленных методов объектно-ориентированного анализа и проектирования и инструментальных CASE-средств, поддерживающих их, заставляет задуматься о преимуществах их использования. В главах 6–8 рассмотрены вопросы объектно-ориентированного анализа и методов проектирования. CASE-средства, поддерживающие эти методы, также описаны в главах 6 и 7.

Распределенные системы и Web

1990-е годы характеризуются возросшими насущными потребностями рынка в разработке нового программного обеспечения, а также появлением более дешевых и более мощных компьютеров. Это привело к новому витку развития объектно-ориентированного подхода и появлению ряда новых его применений, помимо разработки GUI и систем ИИ. Приобрели важное значение распределенные вычисления и архитектура "клиент/сервер", а объектная технология стала основой многих разработок, особенно с появлением так называемых трехуровневых систем на платформе "клиент/сервер". Продолжают играть важную роль и реляционные базы данных. Новые области применения и улучшенные аппаратные средства способствовали массовому использованию объектно-ориентированного программирования и потребовали соответствующего внимания к вопросам объектно-ориентированного проектирования и (позже) анализа. Теория объектно-ориентированных баз данных также была сформирована на протяжении этого десятилетия. Сейчас эти технологии начинают использоваться в широких масштабах. Появление и популяризация World Wide Web привели к возникновению новой проблемы. Поскольку в Web передается самая разнообразная информация — текст, графика, звук или видео — реляционные базы данных не обеспечивают качества, необходимого для приложений, требующих хранения и извлечения сложных структур данных. Для управления этими структурами естественно использовать объектно-ориентированный подход. Первым широко известным языком с поддержкой технологии Web был объектно-ориентированный язык Java. Компании, поддерживающие активно используемые Web-узлы, такие как Microsoft и IBM, для обеспечения репликации, контроля версий, скорости и способности к восстановлению были вынуждены применять объектно-ориентированные базы данных, такие как Versant и ObjectStore. Сетевые вычисления, "тонкие" клиенты и агенты требуют развития общего теоретического подхода и методов разработки программного обеспечения. Перспективы объектной технологии самые обнадеживающие. Развитие данного решения будет представлено далее в этой книге.

Анализ и проектирование

С начала 1990-х основное внимание сместилось с проектирования на анализ. Первая книга под названием *Object-Oriented Systems Analysis* была написана Шлеер и Меллором [707] в 1988 году. Как и в ранней работе Буча (Booch), в ней непосредственно не был описан объектно-ориентированный метод. Основное внимание уделялось описанию расширенной модели "сущность-связь", основанной на представлении задачи в терминах сущностей и отношений между ними. При этом поведенческие аспекты объектов игнорировались. В 1992 году Шлеер и Меллор опубликовали вторую книгу, посвященную своим исследованиям. В этой книге утверждалось, что данное поведение может моделироваться с использованием обычных диаграмм переходов из состояния в состояние. Между тем Питер Коад (Peter Coad) объединил идеи поведения в рамках простого, но объектно-ориентированного метода [171, 172]. Это породило живой интерес к объектно-ориентированному анализу и проектированию и привело к появлению публикаций по этим вопросам.

В последнее время акцент сместился в сторону стандартов. Объектная технология может преуспеть в конкуренции с существующими подходами, только если пользователи получат уверенность в том, что они смогут получить от объектно-ориентированных приложений все необходимые свойства открытых систем. Если объектно-ориентированные приложения взаимно несовместимы, если объектно-ориентированные базы данных не могут взаимодействовать между собой и с реляционными базами данных и если не существует стандартных форм записи и терминологии для объектно-ориентированного анализа, на это надежды мало. Главный сторонник стандартизации — это рабочая группа по развитию стандартов объектного программирования OMG (Object Management Group). OMG — это очень большая группа влиятельных компаний (около 700), призванная добиться согласованности терминологии объектно-ориентированного подхода и стандартов интерфейсов различных поставщиков. Такой уровень сотрудничества — редкость для компьютерной индустрии. Встречи технического комитета OMG проводятся в Европе, США и на Дальнем Востоке на международной основе. Группа OMG призвана быстро публиковать стандарты, быстрее чем какие-либо другие организации по стандартизации. Она уже опубликовала ряд стандартов, начиная с нескольких версий широко используемой технологии построения распределенных объектных приложений CORBA (Common Object Request Broker Architecture) и многоуровневой архитектуры для взаимодействия распределенных объектно-ориентированных приложений, аналогичной, в некотором смысле, семиуровневой модели ISO, и заканчивая стандартизованным определением понятия обмена. Первая версия CORBA определяла принципы построения продуктов, скрывающих сложность стратегии распределения на базе RPC. Стандарт CORBA 2 допускает существование брокеров объектных запросов от разных производителей и их взаимодействие, а в CORBA 3 добавлены языки сценариев и поддержка асинхронной передачи сообщений для их гарантированной доставки. Появилась компонентная модель CORBA Component Model, подобная EJB (Enterprise Java Beans). Это серверные компоненты, которые рассматриваются ниже, в главе 7. Некоторые поставщики предлагают продукты ORB (Object Request Broker), совместимые со стандартами CORBA. Официально ISO признано только несколько стандартов OMG, но все рабочие версии стандартов группы получили широкое распространение. Конкурирующий набор популярных стандартов *де-факто* был разработан в "лагере" Microsoft; к ним относятся компоненты Active X, COM+ и DCOM. Эти архитектурные модели, CORBA и DCOM, будут обсуждаться позже, в главе 4. Группа OMG также одобрила язык UML как стандартную форму обозначений для

объектно-ориентированного проектирования, основанную на объединении систем обозначений Буча, Якобсона и ОМТ. Эти вопросы будут рассматриваться детальнее в главе 6 и приложении Б.

Компоненты

На момент написания книги основными вопросами, связанными с объектной технологией, были компонентная разработка CBD (component-based development), шаблоны, стандартизация формы записи для объектно-ориентированного анализа и проектирования, процесс разработки и архитектура. Эти вопросы рассматриваются в главах 6 и 9. Поскольку объектная технология стала широко распространенной, для организаций, работающих в рамках данного подхода (например, в области электронной коммерции, Web-технологий и промежуточного программного обеспечения), на первый план вышли вопросы совместимости и миграции. Эти вопросы рассматриваются в главах 4 и 5.

Таким образом, самая поздняя фаза развития объектно-ориентированных методов характеризуется смещением акцента с программирования на проектирование и анализ, а также на вопросы распределенных систем и стандартов. К тому же внимание стало уделяться приложениям, требующим серьезных вычислений и обработки сложных данных. Это привело к появлению объектно-ориентированных баз данных. Практически все коммерческие версии программных продуктов созданы на основе объектно-ориентированных методов. Во многих современных проектах по разработке программного обеспечения среднего уровня сложности, а также в больших и важных проектах уже используют объектно-ориентированное программирование и соответствующие методы. В главах 6—9 будет глубже рассмотрена разработка объектно-ориентированного программного обеспечения. В главе 10 описывается несколько приложений объектно-ориентированного подхода и программирования. В табл. 1.1 подводятся итоги краткого описания истории первых трех десятилетий развития объектной технологии.

Таблица 1.1. Три десятилетия развития объектно-ориентированных методов

Фаза I: 1971–1980 гг.	**Фаза II: 1981–1990 гг.**	**Фаза III: 1991–2000 гг.**
Период изобретения	**Период замешательства**	**Период созревания**
Моделирование дискретных событий	WIMP-интерфейсы Xerox & Apple	Акцент на анализ, проектирование, архитектуру и бизнес-модели
Язык Simula	Расширения Lisp	Коммерческие приложения
Кау: машина Flex	Среда ИИ	Распределенные и Web-системы
PARC: Dynabook	Новые языки: Eiffel, C++, ...	Объектно-ориентированные базы данных
Smalltalk		Стандарты, шаблоны, Java, компоненты, миграция

Объектно-ориентированные методы являются частью культуры разработки программного обеспечения и все еще остаются незамеченными многими основными компаниями-разработчиками. Со смещением фокуса разработки в область распределенных систем объектная концепция стала наиболее подходящей парадигмой разработки, акцентирующей внимание на инкапсуляции и передаче сообщений. Возрастающее значение стоимости поддержки системы может привести к осознанию, что возможность повторного использования — это *ключевой* вопрос программирования, проектирования и анализа. Однако это все кажется слишком простым и редко используется в спорах между сторонниками и противниками объектно-ориентированного подхода. Только несколько коммерческих проектов, которые используют объектно-ориентированные технологии, обеспечивают достаточную степень повторного использования кода. Участники лишь некоторых из них, как описывается в главе 2, сообщали о таком преимуществе. Однако развивающийся рынок компонентов требует повторного использования в довольно больших масштабах. Большинство экспертов верят, что для успеха CBD требуется больше внимания уделять архитектуре программного обеспечения и увеличению роли моделирования объектов. К тому же с уходом в историю проблемы 2000 года большие организации возвращаются к истокам и переходят к использованию согласованных процессов программной инженерии. Можно надеяться, что это послужит отличительным признаком первого десятилетия двадцать первого столетия. Эти годы будут отмечены как золотой век архитектуры и совершенствования процесса. Более подробно это показано в табл. 1.2.

Таблица 1.2. Непосредственное будущее объектно-ориентированных методов

Фаза IV: 2001–2010 гг.
Золотой век архитектуры и совершенствования процесса
Концентрация внимания на архитектуре и шаблонах (микроархитектура)
Формирование ОО процессов разработки
Широкое распространение распределенных систем
Переход к компонентно-организованным системам и оболочкам для ранее созданных систем
Компонентно-ориентированная разработка обеспечит настоящее повторное использование
Особое внимание к моделированию бизнес-процессов и инженерии требований
Смещение от C++ к Java и другим безопасным языкам программирования

С моей точки зрения в недалеком будущем будут необходимы не только лучшие, более чистые и эффективные объектно-ориентированные языки, но и лучшие методы для проектирования объектно-ориентированного программного обеспечения и инженерии требований. Результативная смесь языков, такая как C++, с соответствующими инструментальными средствами UI, такими как Microsoft Visual Studio, уже существует и обеспечивает создание множества полезных объектных библиотек низкого уровня. Чтобы сделать эти библиотеки применимыми для коммерческого использования, требуются соответствующие методы и

средства. Есть твердая уверенность, что это — важное предварительное условие для широкого принятия объектно-ориентированного подхода. UML будет основой для этой работы, но он должен развиваться, чтобы отражать реальные потребности бизнеса. Кроме того, необходимы лучшие методы инженерии требований. Методы и модели процессов, связанные с этим вопросом, будут обсуждаться в главах 8 и 9, где основное внимание будет обращено на раннее тестирование, которое должно получить широкое распространение. Конечно, самые полезные библиотеки компонентов будут разрабатываться вместе с реальными программами. Преимущества и потенциальные ловушки объектно-ориентированного подхода проанализированы в главе 2, а в этой главе читатель ознакомится с необходимыми понятиями и терминологией.

Объектная технология подвергается регуляризации, подобно экспертным системам и технологиям, основанным на правилах. Объектно-ориентированные методы составляют часть основных инструментальных средств разработчика программного обеспечения, в состав которых входят языки четвертого поколения 4GL, базы данных, графическое программное обеспечение и т.д. Однако объектная технология все еще впитывает новые идеи, связанные с человеческим фактором, моделированием данных, искусственным интеллектом и другими областями компьютерных наук. Исследования в данной области продолжаются. Однако если брать в целом, объектно-ориентированное программирование можно рассматривать как сформировавшуюся дисциплину, постоянно используемую коммерческими организациями. Технологии объектно-ориентированных баз данных и брокеры объектных запросов также могут рассматриваться как относительно сформировавшиеся области, как будет показано далее в этой главе. Однако если говорить о несовершенстве данной технологии, то в первую очередь следует обратить внимание на область методов. Чтобы понять, почему, надо ознакомиться с основными принципами объектного подхода, чем мы и займемся далее.

1.2. Что такое объектно-ориентированные методы

Как уже говорилось, понятие *объектно-ориентированные методы* является очень обширным, как, собственно, и "объектно-ориентированный" (ОО) и "объектная технология" (ОТ). В частности, оно означает объектно-ориентированное программирование, проектирование, анализ и базы данных, т.е. фактически целую философию разработки систем и представления знаний на базе мощного подхода.

Исторически, как было показано ранее, развитие этой области началось с объектно-ориентированного программирования, и только совсем недавно появился большой интерес к другим вопросам. С административной точки зрения вопросы программирования, возможно, являются наименее интересными, но, чтобы понять основные концепции и терминологию, мы начнем с обзора объектно-ориентированного программирования и связанной с ним терминологии и не будем возвращаться к данному вопросу. Далее, особенно в главах 6, 7 (в которых рассматриваются методы, CASE-средства и сопутствующие вопросы) и 9, будет показано, как эти основные концепции применяются в жизненном цикле разработки системы в качестве метода анализа. Другими словами, мы начнем с конкретных вопросов программирования, а затем перейдем к абстрактным методам проектирования и анализа, после чего рассмотрим вопросы управления процессом. Объектно-ориентированная разработка во многом сводится к так называемой компонентной разработке, хотя, строго говоря, можно создавать компоненты и без использования объектно-ориентированного программирования. Просто готовый

продукт может напоминать объекты своим поведением. Мы разберемся с этим в главе 7. Для меня объектная технология — это нечто большее, чем программирование и даже проектирование. Она обеспечивает общий подход к представлению знаний, который можно применять как для бизнес-моделирования, так и для построения систем. Это будет рассмотрено позже в главах 6–8.

В следующем разделе вводится терминология объектно-ориентированных методов, которая включена в словарь терминов, где также указаны определения некоторых других, возможно незнакомых, терминов. Читатель должен сознавать, что различные авторы иногда используют эти термины совершенно по-разному. Это не удивительно, поскольку данный подход вызывает очень живой интерес. К счастью, в значительной степени благодаря усилиям Object Management Group, появилась некоторая стандартная терминология, которая и будет использоваться в этой книге. Использованные в этой книге термины согласованы, что, в свою очередь, наводит на мысль, что такие сферы, как моделирование бизнес-процессов, искусственный интеллект, семантическое моделирование данных, управление знаниями и объектно-ориентированный подход, рано или поздно должны объединиться.

1.3. Основная терминология и идеи

В следующей главе будет показано, что на изменение структур данных приходится порядка 16% расходов в области информационных технологий. Чтобы понять основы объектной технологии (ОТ), попытаемся разобраться, почему это происходит в традиционных компьютерных системах и как, при надлежащем применении, ОТ помогает снизить эти расходы. В дальнейшем это послужит основой для понимания основных терминов.

Традиционная компьютерная система, основанная на так называемой аппаратной архитектуре фон Неймана (Von Neumann architecture), может рассматриваться как множество функций или процессов, работающих с набором данных, хранимых либо в памяти, либо на диске — это не принципиально. Эта статическая архитектурная модель показана на рис. 1.1. Из рисунка видно, что в процессе работы системы динамика состоит в вызове некоторой функции f(1), которая считывает соответствующие данные А, преобразовывает их и записывает в В. Потом вызывается некоторая другая функция f(2), которая тоже считывает некоторые данные (возможно, те же), использует их по назначению и записывает в С. Подобный перекрывающийся доступ к данным порождает проблемы параллельной обработки и целостности, которые могут быть разрешены с помощью систем управления базами данных. Перед тем как двигаться дальше, ст*о*ит рассмотреть вопрос, что нужно сделать, чтобы изменить часть структуры данных.

Рассматривая это с точки зрения специалиста по поддержке программы, можно сделать единственный вывод: необходимо проверить, затрагивают ли эти изменения каждую отдельную функцию. В этом вопросе может помочь качественная документация, но на практике она доступна редко. Отчасти это объясняется тем, что хорошая документация сама должна состоять из объектно-ориентированного описания системы, и ее невозможно представить в отрыве от объектно-ориентированной реализации или, по крайней мере, проектного решения. Кроме того, изменение каждой функции в соответствии с новыми структурами данных может иметь побочный эффект в других частях системы. Возможно, это объясняет необычайно высокую стоимость поддержки программных систем.

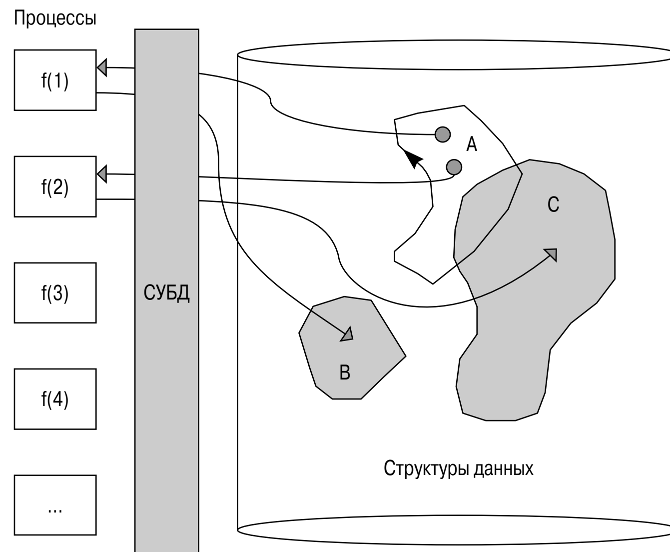

Рис. 1.1. Архитектура обычной компьютерной системы

На рис. 1.2 показан совершенно иной архитектурный подход к системам. Все данные, доступ к которым нужен функции, инкапсулируются в один пакет с этой функцией — так называемый *объект* (object) — таким образом, чтобы другой объект не имел доступа к этим данным. Используя аналогию, предложенную Стивом Куком (Steve Cook), можно рассматривать эти объекты как яйца. Желток — это структура данных, белки состоят из функций, которые имеют доступ к этим данным, а скорлупа представляет сигнатуру общедоступных операций. Оболочка интерфейса скрывает реализацию и самих функций, и структур данных. Предположим, что в яйце, изображенном "в разрезе" на рис. 1.2, изменилась структура данных. Тогда специалисты по сопровождению должны проверить только факт влияния изменений на белок этого яйца; сфера вмешательства локализована. Изменение реализации одного объекта не может затронуть "внутренность" другого. В этом и состоит **инкапсуляция**: данные и процессы **объединяются** и **скрываются** за интерфейсом.

Однако при использовании этой модели в чистом виде возникает проблема. Допустим, каждый объект содержит функции, которым нужны одинаковые данные. В таком случае появляется необходимость дублирования данных, и подход становится довольно непрактичным. Решение состоит в передаче сообщений между объектами. Тогда объект X сможет использовать данные A, но не инкапсулировать их. При условии, что в желтке X хранится идентичность другого объекта Y, который содержит необходимые данные, он может передать сообщение, запрашивающее эти данные или даже несколько преобразованную их версию. Это показано на рис. 1.3, где маленькая черная точка символизирует идентичность целевого объекта, а стрелки показывают направление передачи сообщения. Можно сказать, что это — половина принципа объектной технологии. Вторая половина — это возможность классификации объектов и их связывания различными способами. Обратите внимание на то, что этот подход локализует и таким образом сильно упрощает исходную задачу.

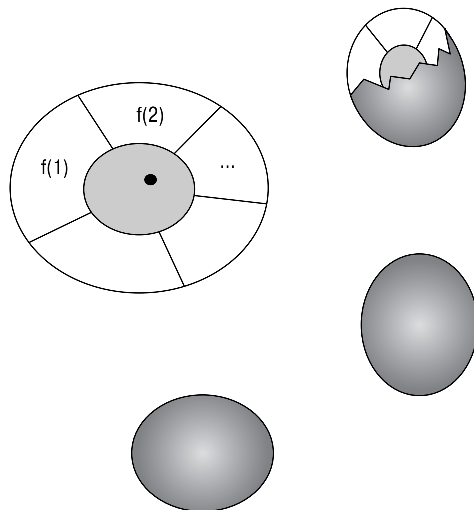

Рис. 1.2. Архитектура объектно-ориентированной системы

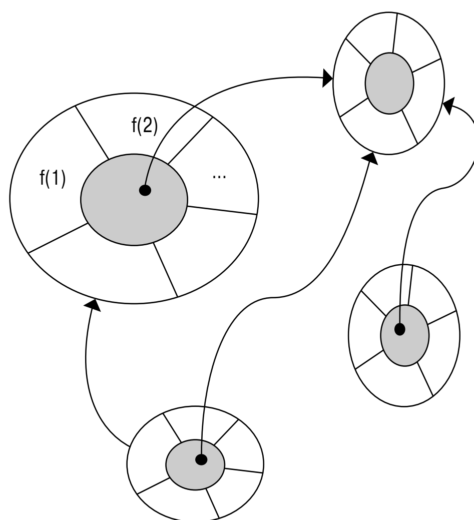

Рис. 1.3. Передача сообщений исключает дублирование данных

При изменении структуры данных программисту нужно всего лишь проверить функции в белке яйца, инкапсулирующего эти данные. Это изменение не может повлиять на другие части системы, если оболочка не повреждается или не деформируется, т.е. если не меняется интерфейс. Таким образом, если необходимо на порядок уменьшить проблемы сопровождения, нужно очень упорно поработать, чтобы обеспечить корректные, полные и устойчивые интерфейсы объектов. Из этого следует, что в данном случае согласованный анализ и проектирование еще более эффективны и необходимы, чем для традиционных систем.

Этот дополнительный объем работ дает результат, потому что объектная технология ведет к некоторым очень существенным преимуществам. Любопытно, что этот принцип инкапсуляции часто игнорируется разработчиками методов объектно-ориентированного анализа, как будет показано позже.

С этими несомненно понятными основными идеями можно переходить к расшифровке “жаргона” объектно-ориентированных методов. Разработка объектно-ориентированного программного обеспечения обычно описывается с использованием указанных ниже терминов и концепций.

Объекты

Основными единицами построения системы на этапе концептуализации, проектирования или программирования являются **объекты** (object) или экземпляры, организованные в классы с общими **свойствами** (features). Эти свойства могут быть трех видов.

- **Атрибуты** (attribute), такие как объем, положение или цвет, символизируют связи с другими объектами и состояние самого объекта.
- Процедуры или услуги, предоставляемые объектом, такие как перемещение или расширение. Их называют **операциями** (operation) или **методами** (method).
- Правила, которые устанавливают взаимосвязи свойств объекта или определяют условия его жизнеспособности. Их иногда называют **инвариантами** (invariant).

Строго говоря, методы являются функциями, которые реализуют операции, а операции — это абстрактные спецификации методов. В традиционном объектно-ориентированном программировании атрибуты не являются видимыми для других объектов, но ниже и в главе 6 будет показано, что это условие можно ослабить, что даст существенные преимущества. Идея пакетирования функций с данными тесно связана с понятием типа данных в традиционных языках программирования, где 3 — это экземпляр “типа” `integer` (целое число), а целые числа характеризуются только одним атрибутом (их значением) и несколькими допустимыми арифметическими операциями. Действительные числа характеризуются похожими операциями, но реальное осуществление умножения для чисел с плавающей точкой совершенно отличается от способа реализации этой операции для целых чисел. Аналогично задавая параметры и методы, можно трактовать более сложные объекты, часто встречающиеся в моделях данных. Слово “метод” употребляется в его первоначальном значении, пришедшем из языка Smalltalk, и никак не связано с иными (объектно-ориентированными) методами. В последнем случае так называются методы разработки системы, которые подробнее обсуждаются в главах 6–8. В данном же контексте **метод** (method) определяется как процедура или функция, которая изменяет состояние объекта или заставляет объект отправить сообщение. Описание или сигнатура метода называется **операцией** (operation). Операции показывают, какие сообщения объект может успешно обрабатывать.

Класс (class), в смысле объектно-ориентированного программирования, — это совокупность объектов, которые имеют общие свойства и методы. В этой книге термин “класс” обычно используется во множественном числе. Класс может рассматриваться как шаблон для построения экземпляров. **Тип** (type) объекта — это спецификация класса, а класс — это реализация типа. Тип объекта отражает идею (*содержание* класса), а не коллекцию свойств (*расширение*), поэтому имеет уникальное название. Атрибуты и методы типа объекта часто

рассматриваются как его **свойства** (features) или **обязанности** (responsibilities). Атрибут представляет обязанность знания чего-либо, а метод — обязанность выполнения.

По мере возможности объекты должны основываться на реальных сущностях и понятиях приложения или предметной области. Объекты могут быть или классами, или экземплярами, хотя некоторые авторы и стандарты, такие как UML, используют термин *объект* как синоним слова *экземпляр*. Как отмечено в работе [71], в некоторых языках класс может быть экземпляром класса более высокого уровня или метакласса. В этой книге везде будет использоваться (несколько жаргонное) значение термина *объект* для обозначения класса или экземпляра. Точные термины будут применяться только там, где различие существенно. Термин *объект* можно четко сформулировать как "нечто идентифицируемое", но на этом внимание будет остановлено не раньше главы 5, которая посвящена обсуждению именно этого вопроса.

Идентичность

На уровне концептуального моделирования объекты (типы, классы или экземпляры) имеют уникальную идентичность на протяжении всей своей жизни. Это отличает объектно-ориентированные модели от, скажем, реляционных. Это чрезвычайно важно в объектно-ориентированных базах данных. На уровне языка программирования можно возразить, что класс не имеет идентичности, но пока этот вопрос обсуждаться не будет.

Инкапсуляция

Структуры данных и элементы реализации метода объекта являются невидимыми для других объектов в системе. Единственный путь получения доступа к состоянию объекта — это передача сообщения, вызывающего один из его методов. Строго говоря, доступ к атрибутам осуществляется через методы, которые считывают и устанавливают их значения. Другими словами, атрибуты — это словарь, с помощью которого можно обсуждать видимое извне поведение объекта. Вообще говоря, в программировании это обеспечивает эквивалентность объектного и абстрактного типов данных. Однако для полного описания объекта необходимо также четко определить его интерфейс. Некоторые методы объекта могут быть скрыты за интерфейсом — это **закрытые** (private) методы. Интерфейс лучше всего рассматривать как общедоступное описание **обязанностей** (responsibility) объекта. Атрибуты могут быть рассмотрены как *обязанность знания*, а операции — как *обязанность действия*. Другими словами, можно сказать, что общедоступный интерфейс определяет *вопросы, которые можно задать объекту*, и *программы действия, которые можно ему предложить*. Такое представление объекта — иногда его называют "очеловечиванием" — позволяет рассматривать данный объект как маленького искусственного человечка, способного поддерживать диалог с другими особями и размышлять о собственных характеристиках. Это представление очень удобно на этапе выделения требований и анализа систем.

Сообщения

Объекты (классы и их экземпляры) общаются посредством передачи сообщений. Это, по большей части, исключает дублирование данных и гарантирует, что изменение структур данных и процедур, инкапсулированных в пределах объектов, не распространяет свое влияние на другие части системы. Сообщения реализуются как вызовы функций. Сообщения всегда возвращают данные, отправленные объектом. Объект может отправить сообщение другому объекту только в том случае, если в нем хранится идентичность другого объекта. Это

может рассматриваться как недостаток объектно-ориентированного модельного представления, когда существует необходимость передачи сообщений многим объектам. Однако позже будет показано, что существуют пути устранения этой проблемы.

Наследование

Причину важности наследования можно понять, рассмотрев аналогию с вареным яйцом. Выше было показано, что сопровождение системы может быть упрощено, а необходимые изменения локализованы, потому что реализация скрыта от других объектов за интерфейсом. Следовательно, это преимущество предполагает, что интерфейс никогда не изменится. Однако наш мир не идеален. Люди — даже разработчики — могут ошибаться. Технические требования меняются. Таким образом, интерфейс может подвергаться изменениям и (в связи с обновлением технологии) даже бо́льшим, чем при традиционных методах разработки. Ответ — это издание законов, запрещающих изменение интерфейсов и настойчиво требующих, чтобы видоизменения затрагивали только подклассы, которые расширяют или, возможно, заменяют свойства имеющихся объектов. Любое отклонение от этого режима должно рассматриваться как базовая перестройка архитектуры системы. Это правило предполагает, что классы предназначены для расширения, и укрепляет позиции объектно-ориентированного анализа и проектирования. Экземпляры (обычно) наследуют все свойства классов (и только их), которым они принадлежат. Это называется **классификацией** (classification). Но в объектно-ориентированной системе можно дать возможность классам наследовать свойства более общих суперклассов. В этом случае унаследованные свойства могут перекрываться, кроме того, для обработки исключительных ситуаций могут вводиться дополнительные свойства. Наследование реализует идеи **специализации** (specialization) и **абстракции** (abstraction) и представляет частный случай структурной взаимосвязи между группами классов. Наследование — это только одна из абстрактных структур, с помощью которой мы упорядочиваем мир; но она чрезвычайно важна, и ее важность соответствует важности глагола *to be* в английском языке. Позже будет рассмотрена еще одна ключевая структура — композиция (соответствующая глаголу *to have*).

Полиморфизм

Возможность использовать одинаковые выражения для обозначения разных операций называется полиморфизмом. Это, например, использование знака + для обозначения сложения в классе вещественных или целых чисел, применение сообщения “add 1” (“прибавить 1”) и к счету в банке, и к списку памяток: похожие сообщения дают совершенно разные результаты. Полиморфизм обеспечивает возможность абстрагирования общих свойств. Он часто реализуется через *динамическое связывание* (dynamic binding). Наследование — это частный случай полиморфизма, который характеризует объектно-ориентированные системы. Некоторые специалисты утверждают, что полиморфизм является *центральным* понятием в объектно-ориентированных системах, но существуют не объектно-ориентированные модельные представления, в которых тоже присутствует данное понятие. Это можно видеть на примере таких языков, как ML или Miranda.

Подстановки

Чтобы воспользоваться преимуществами повторного использования программного обеспечения, должна существовать возможность замены родительских классов их подклассами. Например, если для представления французов создается новый подкласс класса `People`

(люди), то любое сообщение, которое “понимает” класс `People`, должно быть понятно новому подклассу. Из этого следуют определенные проектные ограничения, которые на уровне абстрактного моделирования часто идут вразрез со здравым смыслом: например, задание многоугольников как подкласса прямоугольников. Это будет обсуждаться позже, в главах 6 и 9.

Делегирование: бесклассовое наследование

Объектная технология является **классово-ориентированной** (class-oriented) в том смысле, что экземпляры извлекают свои свойства из классов, которые в свою очередь могут извлекать свои свойства из более абстрактных классов, находящихся выше по иерархии, или из группы классов. Еще один способ достижения преимуществ объектно-ориентированного подхода — использование бесклассовой парадигмы, где каждый объект рассматривается как прототип в следующем смысле. Каждый объект — это экземпляр, который рассматривается как типичный. Другие экземпляры могут быть порождены путем незначительных изменений свойств существующих экземпляров. Таким образом, стандартная собака (по кличке Шарик или Дружок) имеет четыре лапы. Существует несколько экземпляров собак (с другими именами), порожденных на основе шаблона или прототипа, определенного собакой Шарик. Пес моего друга (по имени Спок) потерял лапу в результате несчастного случая. Таким образом, эта собака полностью подобна Шарику, за исключением того, что имеет три лапы. Языки, поддерживающие эту модель наследования, такие как SELF [767], называются **языками прототипов** (prototype language). Говорят, что они поддерживают бесклассовое **наследование** или **делегирование** (delegation). Фреймы и сценарии ИИ тоже могут рассматриваться как прототипы, хотя они также позволяют использовать наследование классов. **Системы на основе исполнителей** также используют идею передачи (делегирования) функций. Они позволяют объектам передавать другим объектам разрешение выполнять операции от их имени. Языки реализации таких систем — это обычно языки очень низкого уровня, по сравнению с языками объектно-ориентированного программирования.

В двух словах

Итак, структуры данных и детали реализации объекта являются невидимыми для других объектов в системе. Единственный способ получения доступа к состоянию объекта — передать сообщение, которое инициирует выполнение некоторого метода. Интерфейс можно рассматривать как общедоступное объявление *обязанностей* объекта. В целом, реализация объектно-ориентированного подхода в языке программирования, проектном решении или в компьютерной системе характеризуется двумя ключевыми свойствами.

- Инкапсуляция
- Наследование

Как всегда, когда мы имеем дело с дихотомией, инкапсуляция и наследование не являются четко разделенными понятиями; между этими абстракциями существует некоторое противостояние и взаимное притяжение. Наследование может нарушать принцип повторного использования, потому что подобъекты могут иметь привилегированный доступ к реализации методов других подобъектов. Таким образом, некоторые термины, такие как полиморфизм, можно отнести к обеим категориям, а наследование можно даже рассматривать как форму абстракции. Общие и отличные свойства абстракции и наследования рассмотрены в разделе 1.3.3.

Другие авторы при описании общей концепции предпочитают использовать термин "абстракция", а не "инкапсуляция". В данной книге эти термины взаимозаменяемы, и трудно сказать, какая точка зрения является более предпочтительной[3]. Оба слова включают множество важных концепций. Стоит помнить, что инкапсуляция — это только один способ поддержки принципа сокрытия информации.

В двух последующих разделах рассматривается значение этих двух основных принципов и вводится терминология для дальнейшего использования.

1.3.1. Абстракция и инкапсуляция

Абстракция (которая является ключевым средством инженерии программного обеспечения) включает различные вопросы, и ее трудно охватить целиком и сразу. Часто говорят, что абстракция — это *критический* инструмент объектно-ориентированного проектирования. Наиболее подходящим по смыслу является описание, данное в оксфордском словаре английского языка (Oxford English Dictionary — OED): "результат мысленного отделения". Еще одно близкое определение: "представление основных свойств чего-либо без упоминания предпосылок или несущественных подробностей". С понятием абстракции тесно связана идея полноты в том смысле, что она должна инкапсулировать *все* существенные свойства предмета. В объектно-ориентированном программировании этот термин обозначает, что объекты должны абстрагировать и инкапсулировать как данные, так и процессы. Познание осуществляется не только через свойства, но и через поведение: *по делам узнаю я их*. Абстракция является инкапсулированной, если она состоит из интерфейса, видимого для внешнего мира, и невидимой реализации. Интерфейс может рассматриваться либо как общедоступное "лицо" класса, либо как нечто обособленное от него, которое определяет, что реализует класс.

Одно важное различие между стандартными типами и интерфейсами, типами и классами в объектной технологии состоит в том, что последние не полностью определяются атрибутами и операциями (сигнатурой типа). Объектные интерфейсы могут также содержать **описание семантики** (assertion): утверждения о сигнатуре. Характерный элемент семантики — это пред- или постусловие метода или инвариант класса. Это будет очень важным при изучении объектно-ориентированного анализа и проектирования в главе 6, но в данный момент мы не будем акцентировать внимание на данном вопросе.

Абстрактный тип данных, АТД (abstract data type — ADT), — это абстракция (подобная классу), которая описывает совокупность объектов в терминах инкапсулированных или скрытых структур данных и операций над этими структурами. Абстрактные типы данных, в отличие от базовых, подобных `Integer`, могут быть описаны пользователем в создаваемом приложении, а не разработчиками базового языка программирования. Следует отметить, что идея абстрактных типов данных очень напоминает "типы объектов", используемые в методах моделирования данных. Это не случайно. На этом строится материал главы 6. Однако ключевая разница между АТД или классами в объектно-ориентированном программировании и типами объектов состоит в том, что АТД содержат в себе методы. Например, абстрактный тип, представляющий длину в неметрической системе единиц, должен включать методы сложения футов и дюймов.

---

[3] Подробнее см. [71].

На рис. 1.4, *а* показан класс, называемый Employees, вместе с некоторыми его атрибутами и методами. Однажды описанная, эта абстракция доступна разработчику непосредственно и просто. На рис. 1.4, *б* показан этот же тип в более ранней объектно-ориентированной системе обозначений, в которой не учитываются переменные-члены экземпляра. В этой книге будут использованы обозначения UML, которые показаны на рис. 1.4, *а*.

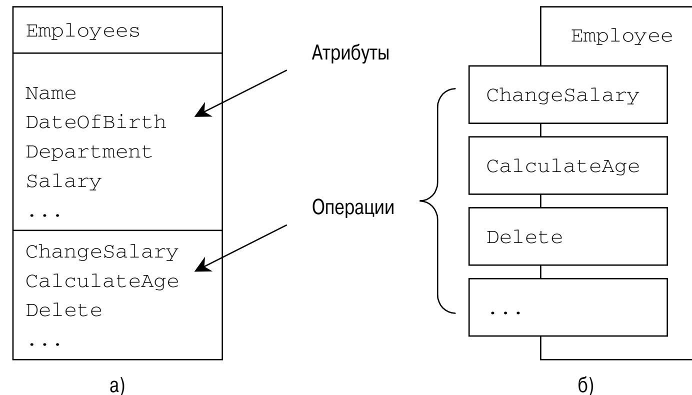

Рис. 1.4. Класс Employees, описывающий понятие сотрудника в системе обозначений UML (а), и объект типа Employee в другой простой системе обозначений, не учитывающей атрибуты (б)

С точки зрения программиста между классами и типами существуют отличия, потому что информация о типе — это только спецификация объекта; класс, которому он принадлежит, может быть определен только во время выполнения программы. Различие состоит в том, что классы описывают спецификации, которые могут совместно использоваться совокупностями экземпляров или другими классами, а не только экземплярами. Однако с точки зрения аналитика классы и типы — по сути, одно и то же.

Выше было указано, что в объектно-ориентированном программировании принято считать, что уникальная идентичность присуща только экземплярам, что отражает инстанцирование экземпляров в компьютерной системе во время выполнения. Вне контекста программирования это является ошибкой, поскольку класс также имеет уникальную идентичность. Как-никак, в концептуальном мире существует только один класс Apple. В этой книге **объект** — это нечто, имеющее уникальную идентичность. Он может быть конкретным субъектом, реальным или вымышленным, концепцией, абстракцией или чем-то конкретным. Это противоречит многим определениям, содержащимся в книгах и статьях, где термин "объект" используется как синоним слова "экземпляр". Поскольку обсуждается объектная технология, "объект" должен представлять собой общий термин в пределах предметной области. Таким образом, если имеется в виду класс, то слово "объект" можно рассматривать просто как сокращенную форму термина "тип объекта".

Абстрактные классы (abstract class) не могут иметь экземпляров; они существуют только для обеспечения основных концепций, которые будут использоваться подклассами. Когда различие между классом и экземпляром становится значительным, термин "объект" уже нельзя использовать. **Реальные классы** (concrete class) могут иметь экземпляры.

Объекты имеют два аспекта: внутренний и внешний. Внутренний аспект описывает состояние, реализацию и инстанцирование объекта. Внешне представление, в чисто объектно-ориентированном стиле, отражает только имена методов и типы их параметров. Оно показывает, что может делать объект (как на рис. 1.4, *б*). Автор немного подкорректировал эту "чистую" точку зрения, показав атрибуты объектов, как на рис. 1.4, *а*. Чтобы вернуться к чистой точке зрения, можно идентифицировать эти атрибуты не с внутренним состоянием, а со стандартными методами, которые обеспечивают доступ и обновление. Однако если это не приводит к путанице, атрибуты удобно рассматривать непосредственно. Во многих книгах и статьях по объектно-ориентированному программированию на такой точке зрения настаивают очень строго. Однако приведенные примеры обычно описывают программные абстракции очень низкого уровня, такие как множества, множества с повторяющимися элементами, коллекции, стеки и т.д. Например, стек описывается как структура данных с четырьмя методами: добавление `push`, выталкивание данных `pop`, проверка на пустоту `empty` и получение верхушки стека `top`. Это говорит о том, что стек — это упорядоченный список данных, элементы которого добавляются и удаляются только с одного конца (рис. 1.5). Доступ к данным стека осуществляется только через эти методы, и нет необходимости описывать их реализацию. Стек может быть реализован как связный список или массив с указателем на верхушку стека. Стек не имеет видимых атрибутов, хотя в качестве таковых могут рассматриваться методы `top` и `empty`.

В коммерческих системах объекты гораздо сложнее, и невозможно представить себе абстрактное понятие сотрудника или счета без описания их атрибутов. Приверженцы старых подходов могут продолжать представлять каждый атрибут А парой стандартных методов доступа: получения и установки значения А.

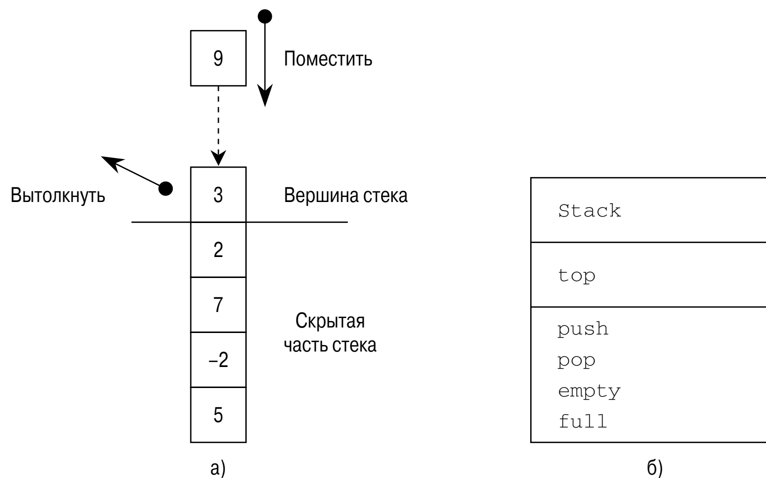

Рис. 1.5. Физическое представление конкретного стека целых чисел (а) и протокол объектного типа стека на языке UML (б)

Экземпляры объектов (строго говоря, классов или "объектных типов") аналогичны записям в базе данных. Они содержат конкретные данные и обладают свойствами объекта. Все экземпляры объекта имеют одинаковое множество атрибутов и методов. Это может не выполняться для более общих классов, потому что наследование класса обеспечивает возможность

специализации классов с удалением или добавлением дополнительных атрибутов и методов. Например, стек — это специальный вид списка, имеющий множество методов конкатенации. Еще одним примером может быть домашняя кошка, которая наследует свойства кошки в целом, но дополнительно имеет хозяина.

Атрибуты экземпляра иногда называются **переменными экземпляра** (instance variable), а атрибуты класса — **переменными класса** (class variable). Переменные класса совместно используются *всеми* экземплярами класса. Например, число экземпляров класса — это подходящий кандидат для переменной класса. Кроме того, различают **методы класса** (class method) и **методы экземпляра** (instance method). Методом класса может быть вычисление суммы или среднего значения некоторого атрибута среди всех экземпляров класса.

Под общим понятием абстракции понимают много малопонятных технических терминов, таких как инкапсуляция, полиморфизм и обобщение. Рассмотрим смысл некоторых из этих терминов. Поступая таким образом, мы обнаружим новые основные свойства объектно-ориентированного стиля.

Инкапсуляцией (которая поддерживает принцип **сокрытия информации**) называется включение в объект всей необходимой ему информации таким образом, чтобы другим объектам не требовалось знаний об этой внутренней структуре. Так, в примере, показанном на рис. 1.4, детали алгоритма изменения оклада `ChangeSalary` могут быть скрыты внутри объекта `Employees`, при этом другие объекты или даже пользователи не смогут получить эту информацию. Подобным образом можно использовать закрытые данные объекта, такие как зарплата. Реализация хранилища данных тоже всегда должна быть закрытой в рамках объекта. Для доступа к данным о зарплате (или для их изменения) не обязательно знать, в каком виде они хранятся: как целое число или число с плавающей точкой. То же можно сказать и о методах; их детальная реализация должна быть скрыта от других объектов, а видимым должно быть только поведение. Помимо идентичности, объекты могут иметь внутреннее состояние, но оно не должно быть доступно напрямую. Одно из следствий этого принципа — клиенты объектов не подвергаются опасности при изменении реализации, если при этом не меняется интерфейс. Невидимые или инкапсулированные части объекта — это его **закрытая реализация** (private implementation); упомянутые выше видимые атрибуты и методы составляют **открытый интерфейс** (public interface) объекта.

В [71] предлагаемый в данной книге подход (как и в работах множества других авторов) подвергается критике за смешение понятий абстракции, инкапсуляции и сокрытия информации. Впрочем, автор это делал умышленно. Сокрытие информации [617] — это принцип модульности и закрытости проектного решения одних объектов для других. Берард определил *абстракцию* как процесс, посредством которого мы решаем, какая информация должна быть видимой и какая нет. Инкапсуляция — это только стратегия упаковки, используемая для реализации этих решений. Он утверждает, что сокрытие информации — это не обязательно хорошо, и, конечно, это не уникальная идея для объектно-ориентированного программирования. Хотя я до некоторой степени и поддерживаю этот подход, все же хочу заметить (в чем убежден), что смешение этих понятий во вводном тексте не создает путаницу, а облегчает понимание.

Стратегии связывания

В программировании возможность определять класс объекта во время выполнения программы и выделять для него память называется **динамическим связыванием** (dynamic binding). В языках, поддерживающих статическое связывание, память под объекты и их типы

распределяет компилятор, определяя их класс раз и навсегда. Поскольку эта книга посвящена далеко не только программированию, различия между понятиями класса и типа учитываться на будут. Однако важно помнить, что класс — это не просто множество. Класс включает переменные-члены и действия, а множество — только члены. Динамическое связывание противоположно **раннему** или **статическому связыванию** (static binding), когда распределение памяти для разных типов осуществляется компилятором. Динамическое связывание называют также **поздним** (late binding). **Динамическое связывание** — это методика программирования, которая реализует принцип полиморфизма в языках объектно-ориентированного программирования. Платой за повышение гибкости при динамическом связывании является снижение быстродействия и более низкая производительность, по сравнению со статически связываемыми системами.

На рис. 1.6 показаны способы реализации динамического связывания и полиморфизма.

Снова сообщения

В объектно-ориентированных системах данные извлекаются из объекта одним единственным способом — путем передачи объекту **сообщения** (message). Сообщение содержит адрес (объекта или объектов, которым оно передается) и инструкцию, состоящую из имени метода и списка (возможно, пустого) параметров.

Если адресат содержит метод, для которого инструкция имеет смысл, то ответ возвращается передающему объекту. Таким образом, целочисленному объекту 3 можно передать сообщение `Add(5)` (прибавить 5) и ожидать ответа `8`. В ответ на сообщение "сообщите оклад сотрудника с именем Эрика", возвращается либо запрашиваемая информация, либо что-то похожее на "У вас нет прав доступа к данным об окладе Эрики", если при обращении к некоторой таблице ограниченного доступа используется определенная процедура сокрытия информации. Если адресат не содержит метода, который может обработать сообщение, возвращается стандартное сообщение об ошибке. Например, такая ошибка имела бы место, если сообщение `CalculateAge` (вычислить возраст) передать объекту `3`, экземпляру типа `Integer`. При использовании компилируемого языка программирования, ошибки такого вида должен обнаруживать компилятор еще до того, как в процессе выполнения программы может возникнуть эта ситуация. Напомним, что методы, инкапсулированные внутри объекта, точно определяют, какие сообщения этот объект может успешно обработать. Совокупность сообщений, на которые может отвечать объект, иногда называется **протоколом** (protocol). Имя сообщения иногда называют его **селектором** (selector). Различные получатели могут поразному интерпретировать одно и то же сообщение.

В строго (статически) типизированных языках программирования компилятор гарантирует, что объекты не смогут получать несанкционированные сообщения. При динамическом связывании каждый объект сам отвечает за защиту от "нелегальных" сообщений. Отметим, что сообщения поступают с интерфейса объекта (поскольку реализация является скрытой).

Если классы рассматриваются как компоненты, протокол часто делится на (возможно, перекрывающиеся) множества сообщений, называемые **интерфейсами** (interface). Обычно, однако, под интерфейсом подразумевают полное множество сообщений.

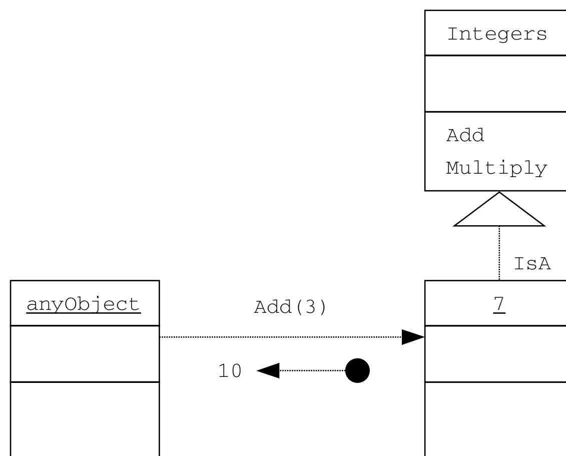

Рис. 1.6, а. Сообщение `Add(3)` передается целому числу 7, которое наследует процедуру сложения от класса целых чисел

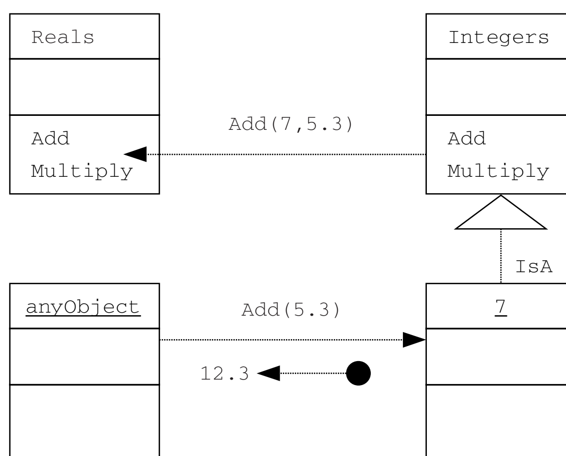

Рис. 1.6, б. Сообщение `Add(5.3)` не может быть обработано в классе целых чисел, поэтому вызывается соответствующий метод класса вещественных чисел. Это называется **перенаправлением**

Множественная абстракция

Итак, термины инкапсуляция, абстракция данных и сокрытие информации означают практически одно и то же. Однако некоторые специалисты различают простое сокрытие информации и так называемую множественную абстракцию. Под этим понимается следующее. Конкретные экземпляры рассматриваются как объекты, принадлежащие множеству с отдельно определенными свойствами и наследующие эти свойства. Например,

Фред — это экземпляр мужчины, у него карие глаза, но он наследует большинство своих свойств от абстрактного множества мужчин, включающего атрибут “цвет глаз”.

Еще одна особенность объектов — они уникально идентифицируются на все время жизни. В системах баз данных идентичность объекта порождает некоторые существенные проблемы и обеспечивает некоторые значительные преимущества. Об этом говорится позже, в главе 5.

Еще о полиморфизме

В различных контекстах возможность использовать один и тот же символ для разных целей называется по-разному: **полиморфизм** (polymorphism), **перегрузка** (overloading) и **перегрузка операций** (operator overloading). Например, передача сообщения `Add` целому и действительному числу фактически активизирует совершенно разные процедуры (рис. 1.7), но в обоих случаях удобно использовать одинаковую форму записи +; это облегчает восприятие и делает язык легче для изучения и запоминания. Аналогично команда `delete` может быть необходимой в различных случаях для разных объектов, особенно если при удалении записи из базы данных должен обеспечиваться контроль целостности. Формально, полиморфизм — имеющий много форм — означает возможность переменной или функции принимать разные формы во время выполнения программы или, более точно, возможность обращаться к экземплярам разных классов. Перегрузка — это особый случай, при котором для обозначения двух разных операций просто используется одно имя, как, например, операция `open` (открыть) применяется к файлам и окнам.

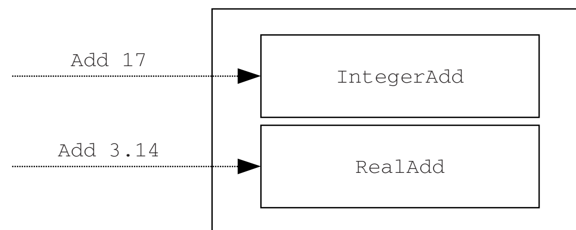

Рис. 1.7. Перегрузка операции (полиморфизм) суммирования

На самом деле в языке Smalltalk понятие полиморфизма используется несколько некорректно. Чистый принцип лучше представлен в функциональных языках, таких как ML, где функции могут иметь аргументы разных типов. Хотя понятие полиморфизма, как оно представлено выше, вполне подходит для последующего изложения материала данной книги, по этому вопросу существует много теоретической литературы, в которой внимание акцентируется на множестве тонких моментов.

В [141] различается несколько видов полиморфизма. На самом верхнем уровне выделяется специальный и универсальный полиморфизм. Специальный полиморфизм — это использование одного и того же символа для семантически несвязанных операций. К этой категории относится перегрузка операций, например использование символа + для суммирования целых чисел и матриц. Еще одна разновидность специального полиморфизма, так называемое приведение, позволяет реализовать операции для входных данных комбинированного типа, например складывать целые числа с вещественными. Универсальный полиморфизм делится на

параметрический и полиморфизм включения (или наследования); последний рассматривается в разделе 1.3.2. Параметрический полиморфизм — возможность выбора аргументов в вызове функции из некоторого диапазона типов. В нашем изложении его можно считать обобщением. Типы полиморфизма представлены в табл. 1.3.

Таблица 1.3. Классификация типов полиморфизма по Вегнеру

Специальный полиморфизм	Универсальный полиморфизм
перегрузка	параметрический
приведение	полиморфизм включения

Обобщение

Возможность определять параметризованные модули называется **обобщением** (genericity); данная возможность используется во многих языках, которые обеспечивают инкапсуляцию, таких как Ada. Пример родового типа — это список, который может быть списком имен, целых чисел или неких специфических объектов, например имен сотрудников (рис. 1.8). Реальный тип определяется только по контексту. В таких языках, как Ada, Modula-2 и даже ALGOL, существует возможность определять параметризованные модули. Обычно параметры являются типами. Родовые сообщения обеспечивают создание повторно используемых компонентов и каркасов за счет устранения зависимости вызова процедур от реального содержимого вызова.

Обобщение и наследование можно рассматривать как *альтернативные* методики создания расширяемых и повторно используемых модулей, но, как будет показано далее, обобщение более ограничено на практике. Тем не менее это свойство полезно иметь в языке программирования, потому что непосредственно повторно используемые компоненты должны применяться к нескольким типам. Например, механизм поиска или сортировки, который работает только с числами, практически бесполезен.

Идентификация объектов

Важный вопрос — как идентифицировать объекты в реальном приложении. Этот вопрос порождает глубокие философские дебаты и часто вызывает большое замешательство. С точки зрения эмпирика, объекты — это нечто, обнаруживаемое органами восприятия, и их выделить очень легко. С феноменологической точки зрения, восприятие — это активный, созидательный процесс, и объекты возникают как из нашего понимания, так и из окружающего мира посредством диалектического процесса. Еще более распространенная точка зрения предполагает, что реальные абстракции — это отражение общественных отношений и человеческой созидательной деятельности, для которой существует объективная основа в реальном мире. Мы вернемся к этой противоречивой задаче в следующих главах. Пока просто определим, что объекты могут быть по-разному представлены аналитиком, знакомым с приложением, и что выбор "лучшего" представления — это фундаментальный вклад человеческого разума в вычислительные системы.

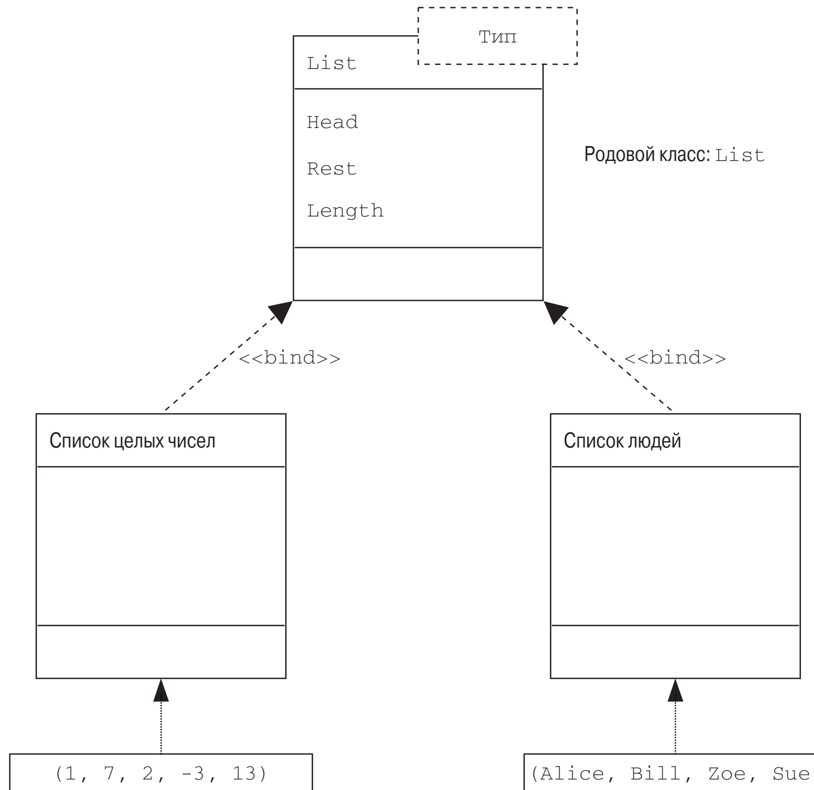

Рис. 1.8. Использование обобщения при моделировании наследования для родового класса `List` (список)

Некоторые объекты соответствуют непосредственно реальным объектам, таким как `Employees`, а другие, такие как стеки, соответствуют придуманным абстракциям. Абстракции отражают реальное разделение предметной области. Уровень абстракции может зависеть от приложения, но это ставит под угрозу повторное использование абстракции, и проектировщик должен определять абстракцию максимально широко. С другой стороны, нужно следить, чтобы не потерять эффективность, создав абсурдные и неоправданные общие объекты. Так, при определении класса `Employees` мы не включаем в число его атрибутов ни "родную планету", ни курсы галактических валют, необходимые для процедуры выдачи заработной платы, хотя некоторые будущие разработчики систем могут упрекнуть нас за такую недальновидность. Что в этом случае должен делать разработчик? Использовать концепцию наследования, к которой мы сейчас и перейдем.

В следующей главе будет показано, что первая причина для выделения абстракции и инкапсуляции — это получение кода, пригодного для повторного использования. Использование объекта посредством его спецификации как абстрактного типа или класса подразумевает, что если его внутренний аспект подвергается изменению, то остальные части системы не будут затронуты. Точно так же, если изменится реализация других объектов системы, то этот

объект тоже не должен быть затронут. Единственная проблема — не должен меняться интерфейс. Таким образом, фиксация "верных" абстракций — это очень важно. Этот вопрос занимает существенное место в объектно-ориентированном анализе и в этой книге.

1.3.2. Наследование

Еще одна характерная особенность объектно-ориентированной системы — это работа со структурными и семантическими отношениями между экземплярами и классами (или типами) и устранение чрезмерной избыточности при хранении одинаковых данных или процедур. Ключевое понятие — это понятие наследования, обобщения или классификационной структуры.

В объектно-ориентированном программировании класс может порождать свои экземпляры в памяти. Эти экземпляры в точности "наследуют" все особенности класса: его методы и атрибуты. Говорят, что класс **классифицирует** его экземпляры, и отношение IsA между экземпляром и его классом называется отношением классификации. Экземпляр может принадлежать только одному классу; другими словами, **множественная классификация** невозможна. Также невозможна и **динамическая классификация**: способность экземпляра менять свой класс во время выполнения программы. Также классы могут наследовать все свойства более общих классов. На рис. 1.9 показан пример ситуации, в которой классы `Cars` (машины) и `Cycles` (велосипеды) являются специализацией (подклассами) класса `Vehicles` (транспортные средства). Для обозначения отношения AKO (A Kind Of — "вид") используется символ UML — маленький равнобедренный треугольник или стрелка. Этот пример иллюстрирует **одиночное наследование** (single inheritance): каждый класс имеет не более одного обобщения. Дальше будет показано, что некоторые языки допускают **множественное наследование** (multiple inheritance), при котором класс может иметь более одного суперкласса. Такие структуры, собственно, называются структурами **обобщения**. Когда речь будет идти о структурах, относящихся к классам и экземплярам, или реализации обобщения и классификации, будем их называть **структурами наследования**. Если экземпляры *в точности* наследуют свойства своего класса, то в подклассах могут добавляться новые свойства. Таким образом, при создании подкласса `MotorizedCycle` (моторизованный велосипед) класса `Cycle` ему необходимо добавить некоторые другие свойства, такие как `FuelType` (тип горючего). Кроме того, в некоторых языках свойства могут перекрываться. Несомненно, такое перекрытие дает возможность подстановки, так что оно запрещено в некоторых языках и проектных решениях.

Если объект имеет тип, тогда он может считаться экземпляром класса. Например, `Fido` — это экземпляр класса собак. Можно пойти дальше и указать, что собака — это частный случай млекопитающего; млекопитающее относится к классу позвоночных животных; и т.д. В итоге получается полная структурная классификация. Формально в число атрибутов каждого экземпляра включается особый атрибут `IsA` (или *слот* в терминологии искусственного интеллекта, где объекты часто называются *фреймами*). Этот атрибут содержит имя класса, которому соответствует экземпляр, или родительский элемент в иерархии. Классы тоже могут иметь родительские элементы, которые будем называть атрибутами **AKO** (A Kind Of). Атрибут AKO содержит список суперклассов, обобщающих рассматриваемый класс. Преимущество такой формализации состоит в том, что класс или объект более низкого уровня может наследовать совместно используемые свойства и методы. При этом исключается необходимость хранения их в каждом экземпляре. Например, атрибут `BirthDate` (дата рождения) совместно используется всеми подтипами класса `Person` (человек); таким образом, нет

необходимости упоминать его явно для класса Employee, если в спецификацию типа включен атрибут (AKO: Person).

Наследуемые методы при определенных обстоятельствах могут перекрываться. Примером может служить метод Delete, унаследованный Employees от базового класса Objects (объекты), содержащего методы для наиболее типичных операций с объектами всех типов. Если поддерживать некоторую безопасность авторизации, для удаления перекрытого стандартного метода может потребоваться особый метод. Хотя перекрытие устраняет возможность подстановки, эта цена часто невелика, если учесть получение более естественных и понятных моделей.

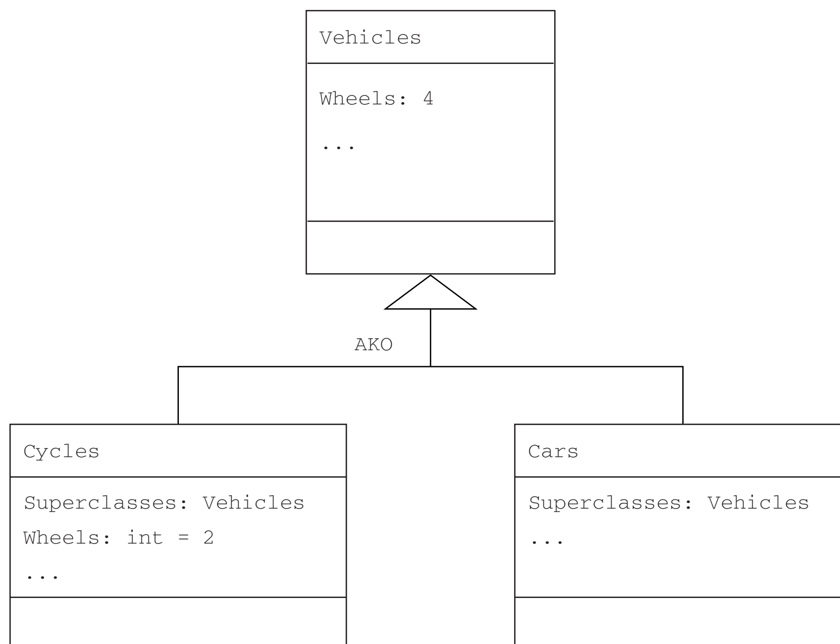

Рис. 1.9. Классы велосипедов Cycles и автомобилей Cars являются специализацией класса транспортных средств Vehicles (отношение AKO). В классе Cycles (велосипеды) перекрывается используемое по умолчанию значение параметра Wheels (количество колес). Классы Cars и Cycles и, возможно, другие классы являются подклассами Vehicles

Если идеи абстракции, рассмотренные в предыдущем разделе, исходят из работ по теории и применению языков программирования, то идеи наследования берут начало в изучении искусственного интеллекта (ИИ), где были разработаны семантические сети [645] и фреймы [562] — методики представления знаний о стереотипных понятиях и объектах. Зависимость между базовыми и более специализированными понятиями в таких сетях реализуется посредством наследования свойств и процедур. Концепции наследования в настоящее время используются в моделировании данных, где широко применяется понятие подтипов логического объекта. В ИИ возможно все: фреймы дают возможность нарушать инкапсуляцию, допускается

множественная и динамическая классификация, а параметры по умолчанию могут передаваться вглубь по структуре наследования. В книге [60] приведены доводы в пользу того, что основанные на фреймовом представлении системы гораздо больше подходят для повторного использования, чем традиционные объектно-ориентированные. Обычно в процессе объектно-ориентированного программирования объекты наследуют методы и свойства от классов, расположенных выше по иерархии, но не наследуют значения атрибутов. Экземпляр наследует только способность иметь определенный тип значений. С другой стороны, в ИИ и в некоторых приложениях баз данных поддерживается возможность наследовать значения. Проиллюстрируем это на примере, к которому иногда будем возвращаться на протяжении всей книги.

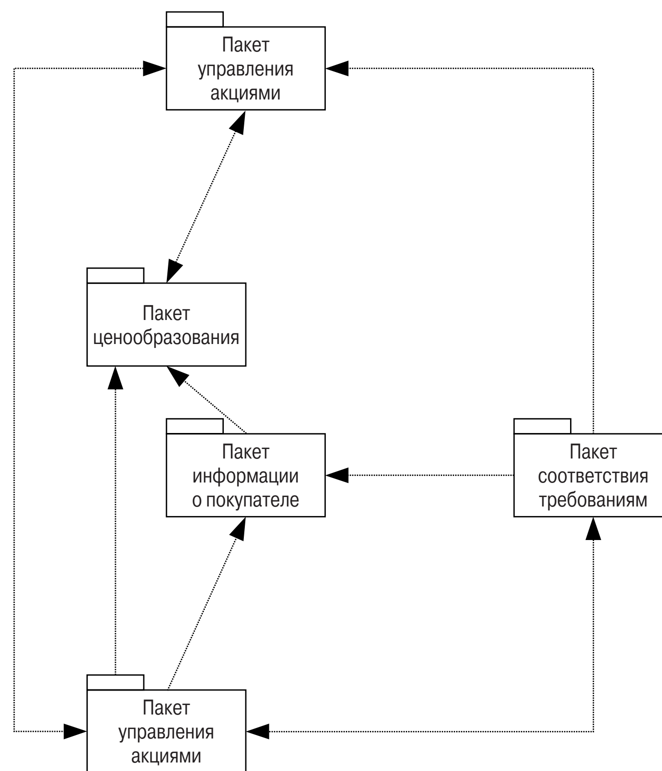

Рис. 1.10. Высокоуровневая структура SACIS

Aardvark в часы досуга

Aardvark Leisure Product — это компания, которая распространяет игрушки и товары досуга для широкой публики. Для этой компании создается информационная система складского учета SACIS (рис. 1.10). Компания Aardvark желает быть первой не только в телефонном справочнике, поэтому руководство решает принять объектно-ориентированный подход к разработке SACIS. Кроме того, система должна быть простой в использовании, чтобы потребители и персонал магазина могли применять ее при распродажах. Это означает, что очень

важную роль играет интерфейс пользователя, и в систему потребуется ввести определенный объем логики, чтобы она могла помочь в поиске товаров, соответствующих нуждам пользователей. Таким образом, SACIS будет экспертной объектно-ориентированной системой баз данных.

Рассмотрим некоторые классы, требуемые в SACIS. В их число по крайней мере должны входить: `People` (люди), `Customers` (потребители), `Adults` (взрослые), `Children` (дети), `StockItems` (товары на складе), `SportItems` (спортивные элементы), `Toys` (игрушки), `Shops` (магазины), `Employees` (служащие) и, возможно, `Suppliers` (поставщики). Классы `Toys` и `SportItems` относятся к классу `StockItems`. Предположим, что существует также класс `Frisbees` и что `F123` принадлежит этому классу. В `StockItems` имеется метод класса, обеспечивающий уведомление об уровне запасов, который наследуется классами `Toys` и `Frisbee`. Если сообщение на уведомление об уровне запасов будет послано `Frisbee`, унаследованный метод проверит значение атрибута `StockLevel` (уровень запасов) и вернет число, представляющее число единиц данного товара на складе. Несомненно, не имеет смысла передавать это сообщение экземпляру `F123`, поскольку отдельные экземпляры класса могут использовать сообщения, направленные всему классу. В этом случае сообщение `delete` (удалить) может быть послано `F123` при продаже этого объекта или лучше (с точки зрения последующей поддержки потребителей) передать сообщение "пометить себя как проданный" с идентификатором покупателя в качестве параметра. Объект `F123` в традиционном объектно-ориентированном программировании наследует все атрибуты `Frisbees`, но не их значения. Аналогично `Frisbees` наследует свойства `Toys`. В системе ИИ мы бы разрешили наследование значений, в частности значений, присваиваемых по умолчанию. Вот почему. Игрушки обладают атрибутом "безопасно для детей", и разумно предположить, что большинство игрушек действительно безопасны. Таким образом, по умолчанию это свойство может принимать значение "да" для всего множества объектов. По мнению автора, удобно наследовать это значение и для класса `Frisbees`, и для экземпляра `F123`, чтобы потребитель мог недвусмысленно запрашивать "это безопасно?" и получать однозначный ответ "да". Для экземпляра `F124`, изготовленного по техническим требованиям заказчика и содержащего цепочки, прикрепленные по периметру, это значение будет перекрываться противоположным. В таких языках, как Smalltalk и C++, все это должно делаться в приложении. В системах ИИ некоторая часть этих действий выполняется автоматически.

Классификационные структуры этого типа реализуют частичный полиморфизм включения, т.е. сообщения, посланные `Toys`, будут поняты `Frisbees` и `F123`, если они не перекрываются.

Композиция и агрегирование

Не все структурные зависимости имеют семантику наследования, однако практические объектно-ориентированные языки должны обеспечивать обработку композиционных структур. В дополнение к наследованию, обобщению и классификации, тут будет рассмотрен другой вид структур: **композиция** (composition), **агрегирование** (aggregation) или структура **APO** (a-part-of). Самый типичный пример — это составной объект, такой как машина, состоящая из корпуса, колес и двигателя. В свою очередь, колесо может состоять из оси, спиц, диска и покрышки.

На рис. 1.11 для обозначения агрегирования используется ромбик. Это обозначение будет более детально объяснено в главе 6. Неоднозначных фраз, подобных "имеет", следует избегать всеми средствами. Аналогичным способом могут быть смоделированы другие ассоциативные структуры, такие как принадлежность, подобие, долг или даже аналогии. В главе 6 будет показано, что эти структуры лучше рассматривать как ассоциации.

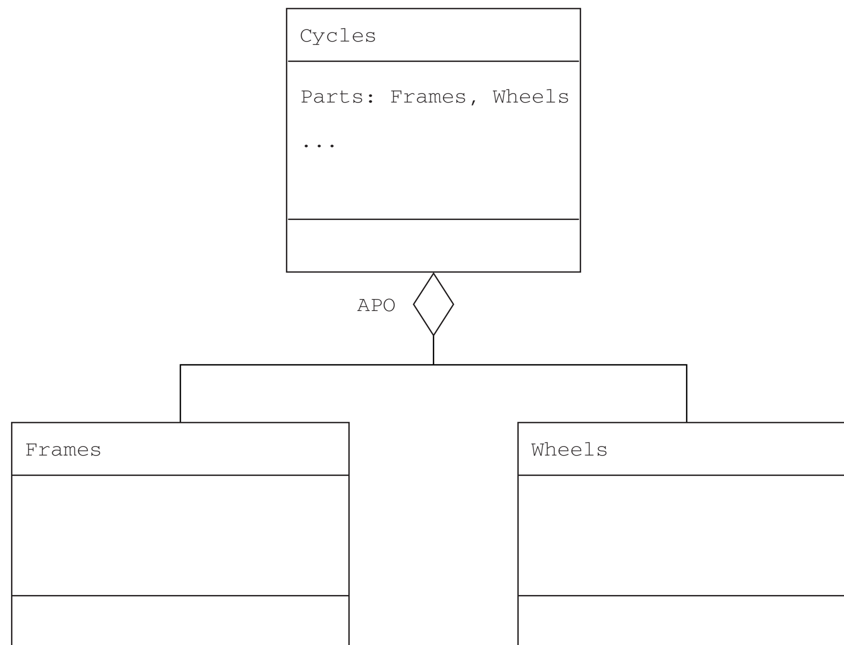

Рис. 1.11. Иерархия композиции, показывающая, что объект `Cycles` состоит из `Frames` (корпуса) и `Wheels` (колес) (и, возможно, из других составляющих). Аббревиатура APO означает A Part Of

Множественное наследование

Возвращаясь к семантике наследования, следует рассмотреть случай, в котором объект или класс может иметь два (или более) родительских объекта. Хорошим примером является рыбка гуппи, которая, как типичная аквариумная рыбка, является экземпляром и класса `Fish` (Рыб), и класса `Pet` (Домашних животных). Гуппи должна наследовать свойства обоих классов. Эта возможность называется множественным наследованием. Проблемой множественного наследования является то, что свойства, унаследованные от двух (или более) родительских классов, часто бывают полностью или частично противоречивыми. Вернемся к нашему примеру: рыба живет в море, а домашнее животное — в доме хозяина, так где должна жить рыбка гуппи? Существует много методов решения этой задачи в каждом конкретном случае. Наиболее привычный метод — это дать возможность системе сообщить о противоречии пользователю и запросить у него значение спорного атрибута. В качестве альтернативы некоторые системы позволяют проектировщику создать процедуру, разрешающую такие противоречия автоматически. Имеются также предложения об объединении ответов в составное

или компромиссное решение (см., например, интерпретацию в приложении А). Другие предложения включают максимальный скептицизм и предположение, что значения являются неизвестными [396].

Возможны два типа конфликтов: имен и значений. В литературе об объектно-ориентированном программировании обычно упоминается только конфликт имен, несмотря на то что в литературе по искусственному интеллекту большее значение придается конфликтам значений, описанным в предыдущем абзаце. Конфликт значений происходит тогда, когда атрибут наследует от порождающих классов два разных значения. Это могут быть значения по умолчанию, если объект сам является классом, или реальные значения экземпляра, если речь идет о наследовании свойств экземпляром (IsA). Конфликт имен происходит тогда, когда два родительских класса содержат разные свойства или методы с одним именем. Конфликты имен обычно возникают в связи с наследованием метода, и в работе [778] перечислены следующие семь стратегий разрешения конфликта, используемые в системе Flavors.

- Вызов наиболее конкретного метода.
- Вызов всех методов в порядке следования или в обратном порядке.
- Выполнение первого метода, возвращающего ненулевое значение.
- Выполнение всех методов и возвращение списка данных.
- Вычисление суммы всех возвращаемых значений.
- Вызов всех демонов `before` (до), а затем вызов всех демонов `after` (после).
- Использование второго аргумента для выбора одного метода или подмножества методов.

К вопросу стратегий разрешения конфликтов мы вернемся в главе 6. Понятие демонов будет объяснено позже по тексту. Можем сказать, демон — это процедура, которая активизируется при изменении данных, а термины `before` и `after` означают, запускается ли процедура в начале изменений или после их завершения; подобно пред- и постусловиям.

Еще один способ избежать некоторых опасностей множественного наследования — отделить наследование интерфейсов от наследования реализации. Этот способ иллюстрируется в языке Java и обсуждается в главе 3.

Роли

И множественное наследование, и отсутствие динамической классификации обнажают вопрос, который до сих пор не обсуждался, — проблема роли. На рис. 1.4 было показано, что `Employees` является классом. Это было бы справедливым в платежной системе, но в целом пребывание в роли сотрудника — это вопрос преходящий. Другими словами, `Employees` означает **роль** (role), а не класс. Поскольку в объектно-ориентированных языках программирования экземпляры не могут изменять свой класс, Анна Арбейтер не может перейти из категории `Students` (студенты) в категорию `Employees` (сотрудники) или из класса `Employees` в `WelfareClaimants` (клиенты департамента социального обеспечения). Чтобы сохранить идентичность Анны, надо сделать ее экземпляром класса `People` и связать с этим классом атрибут экземпляра, отвечающий за сохранение статуса занятости. В качестве альтернативы можно смоделировать роль как класс и использовать шаблоны проектирования `State` (состояние) или `Visitor` (посетитель) (см. главу 7) для перемещения экземпляра из

одного класса в другой. Самая большая опасность в моделировании ролей как классов состоит в том, что это может привести к сверхсложности и созданию огромных структур множественного наследования.

Жизнь без конфликтов в Aardvark

В SACIS необходимость множественного наследования может возникнуть для класса `YoungCustomer` (юный потребитель), который должен наследовать свойства и линию поведения как `Customers`, так и `Children`. Отметим, что некоторые дети могут еще не быть потребителями и что некоторые потребители — взрослые. Класс `YoungCustomer` наследует свойства и методы у класса детей `Children`, которым не продают опасные товары, и это свойство должно перекрывать более слабые значения, унаследованные от класса `Customer`. Это еще раз показывает необходимость стратегий разрешения конфликтов в системах, которые поддерживают множественное наследование. Еще более замысловатая стратегия требуется при продаже таких товаров, как велосипеды, которые являются и игрушками, и спортивным инвентарем. Типичная игрушка — это недорогой товар с коротким временем жизни, не подлежащий техническому обслуживанию. Экземпляры класса `SportItems`, с другой стороны, часто дорого ст*о*ят и нуждаются в систематическом профилактическом обслуживании.

Отметим (рис. 1.12), что `CheckValidItem` (проверка соответствия товара) — это разная операция для классов `Customers` и `Children`, потому что первая проверяет легальность товара, а вторая — легальность и безопасность.

Как было показано, в классическом объектно-ориентированном программировании классы наследуют методы и атрибуты (способность иметь значение некоторого типа), а в искусственном интеллекте экземпляры могут наследовать даже значения. В этой книге иногда предпочтение отдается последнему. Это позволяет лучше оперировать семантическими понятиями, такими как значения по умолчанию. Такой подход также выявляет различия между зависимостями АКО и IsA. Наследование классом свойств интерпретируется как отношение АКО (A Kind Of), а наследование экземпляров — как зависимость IsA. Это отличие аналогично различию между включением и принадлежностью в теории множеств. Напомним, что объектно-ориентированное программирование запрещает множественную классификацию. На рис. 1.13 иллюстрируется естественность множественного наследования и множественной классификации.

Наследование обеспечивает расширяемость. В систему можно добавить новую разновидность объекта без необходимости модификации существующего кода. Таким образом, наш гипотетический сотрудник с Ганимеда (Ganymede), спутника Юпитера, может относиться к новому классу `AlienEmployees`, наследующему свойства и методы `Employees` и имеющему дополнительные свойства. Некоторые функции в этом новом классе могут перекрывать функции базового класса. Таким образом, наследование гарантирует, что функции кодируются только однажды.

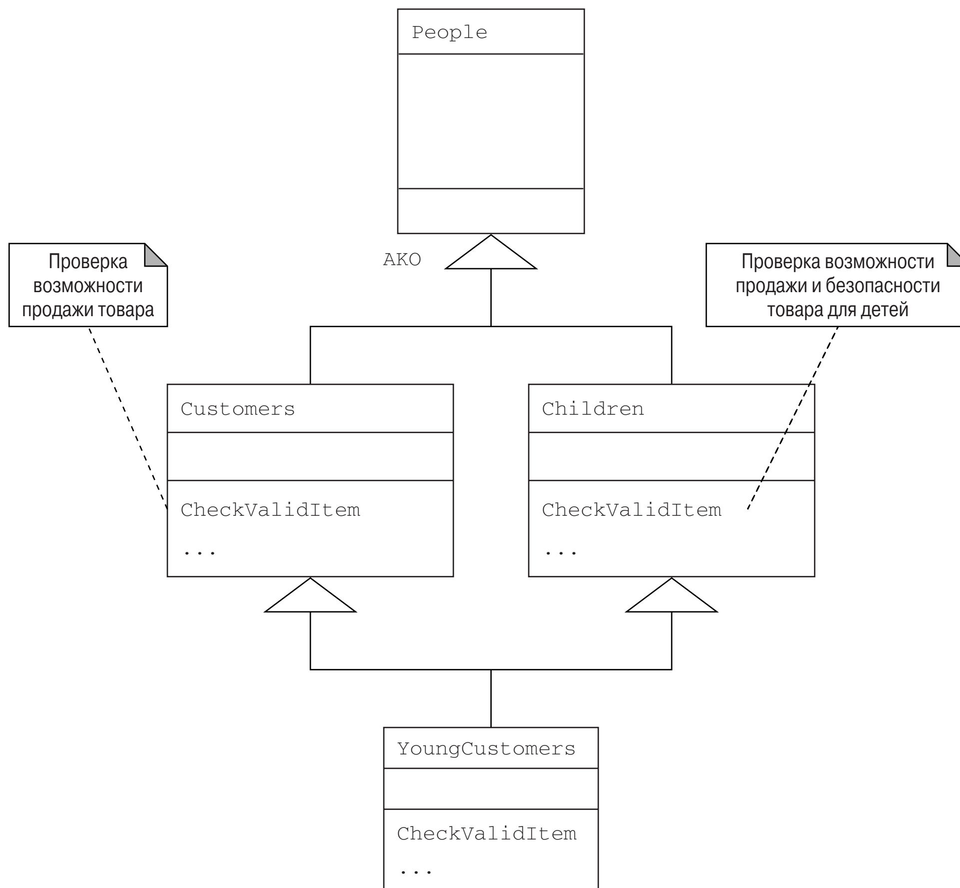

Рис. 1.12. Множественное наследование в SACIS

1.3.3. Инкапсуляция, наследование и объектная ориентация

Умный читатель должен был обратить внимание на то, что в разделе 1.3.1 не было возможности избежать преждевременного упоминания о наследовании. К сожалению, это свидетельствует о том, насколько близки концепции инкапсуляции и наследования. Связь обеспечивается через такие концепции, как полиморфизм, перекрытие, объектная идентичность и передача сообщений.

Наследование часто представляется просто как особый случай полиморфизма, но эта концепция настолько богата и естественна, что данная точка зрения отражает ее не полностью. Пока дело касается языков программирования, эта типично теоретическая точка зрения является обоснованной и полезной. В контексте спецификации и проектирования она более ограничена. Как обсуждалось выше, с этой точкой зрения не совсем согласуется наследование значений.

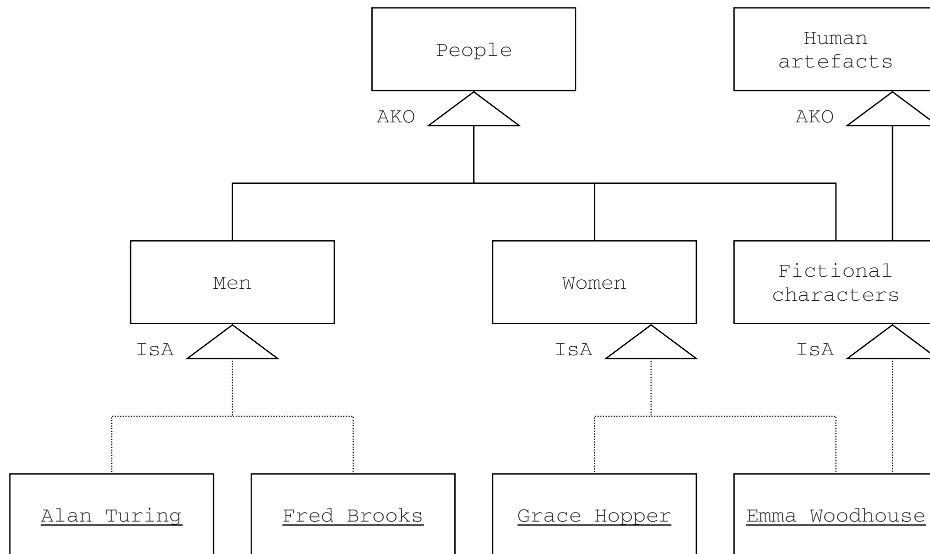

Рис. 1.13. Связи AKO и IsA в структуре классификации

Как будет показано в следующей главе, одно из ключевых преимуществ объектно-ориентированного подхода — возможность повторного использования. Однако с этим требованием необходимо быть предельно осторожным. Безусловно, абстракция обеспечивает возможность повторного использования, но существуют и другие точки зрения [720], [745], согласно которым наследование и делегирование функций могут помешать достижению этой цели. Причина состоит в том, что при наследовании детали реализации иногда открываются клиентам объекта. Кроме того, может быть открыта сама иерархия, так что ее нельзя будет безопасно изменять. Например, если стек определен как частный случай класса `List` (список), то он может унаследовать реализацию операции `head` (голова) как метод `top` и просто исключить несоответствующие операции, такие как `length` (длина). Если реализация стека изменится и станет более эффективной, существует опасность, что клиенты будут зависеть от старой реализации, как частного случая списка. Эта проблема становится еще более существенной, если разрешено множественное наследование. К счастью, Снайдер (Snyder) показал, что аккуратное проектирование позволяет избежать этой проблемы. С другой стороны, наследование частично обеспечивает возможность повторного использования. Без наследования повторное использование многих классов в чистом виде маловероятно.

Применение сложных схем наследования в определенном роде усложняет систему и может свести на нет возможность повторного использования. Компенсирующее преимущество расширяемости — это улучшенная структурная семантика приложения. Чем сложнее система, тем труднее ее поддерживать, тем богаче ее семантика и тем специфичнее сама система. Следовательно, снижается вероятность многократного использования ее компонентов. Однако существуют некоторые приложения, где простое решение — это неудачное решение. Использование сложных схем наследования может быть обоснованным, например, в приложениях из области ИИ и экспертных систем. Именно для таких приложений разрабатываются подобные схемы. По мере того как методология экспертных систем находит свое применение в

традиционных системах, с этим вопросом все чаще сталкиваются аналитики и проектировщики. Например, в системе SACIS задача соответствия товара клиенту требует решения в классе экспертных систем.

Еще несколько определений

Перед тем как завершить раздел основных понятий, нужно добавить несколько определений.

Ключевым понятием объектно-ориентированного подхода является **само-рекурсия** (self-recursion) или ссылка на себя. Это подразумевает, что объекты могут рекурсивно передавать сообщения собственным методам или направлять сообщения себе. В Smalltalk ключевое слово `self` используется для обозначения экземпляра, от имени которого совершается операция, а не класса, который содержит описание этой операции. Это подразумевает, что операция относится к объекту только во время выполнения программы. Еще одна точка зрения на этот вопрос — объект должен знать о собственной уникальной идентичности и должен иметь возможность хранить себя как значение одного из своих собственных атрибутов. Например, в классе сотрудников атрибут `ManagedBy` (руководитель) для экземпляра `J. Smith` может содержать значение `J. Smith`, если J. Smith случайно окажется управляющим директором компании Aardvark. Фактически каждый объект должен содержать ссылку на любой объект, которому он может передать сообщение, если подобная ссылка не передается как параметр.

Таким образом, объектно-ориентированная система программирования, проектирования или анализа должна предоставлять средства инкапсуляции и наследования, а также обеспечивать понятие идентичности объекта и само-идентифицируемости. Для каждого объекта необходимо объявить, какие внутренние данные — аспекты его состояния — можно изменять явно. Его внешнее поведение полностью описывается передачей сообщения.

Теперь можно дать определение **объектно-базированному** программированию как средству разработки программ, поддерживающему инкапсуляцию и идентичность объектов. Иными словами, методы и атрибуты сокрыты внутри и являются закрытыми для объектов, а сами объекты имеют уникальные идентификаторы. Поддержка классов отсутствует или обеспечивается в минимальном объеме, т.е. не поддерживается множественная абстракция. Другими словами, объекты не принадлежат абстрактным классам с отдельной идентичностью. Кроме того, не поддерживается наследование. Ada — это типичный объектно-базированный язык.

Классово-базированные языки, такие как CLU, включают понятие множественной абстракции на уровне экземпляр/класс, но не поддерживают наследование абстрактных классов, которые не могут иметь конкретных экземпляров. Классово-базированные системы обладают всеми свойствами объектно-ориентированных систем; они наследуют их.

Объектно-ориентированные системы обеспечивают наследование свойств и объектно-базированных, и классово-базированных систем, а также дополнительно поддерживают полное наследование между классами, т.е. и экземпляры, и классы наследуют методы и свойства класса (классов), к которому они относятся. Некоторые объектно-ориентированные системы допускают наследование значений атрибутов экземплярами на уровне класса. Объектно-ориентированные системы также обеспечивают саморекурсию.

Компонентно-ориентированные системы — это (тавтология) системы, построенные на использовании компонентов. Компоненты — это выполняемый модуль, который производится, приобретается и развертывается независимо. Это элемент композиции с четко заданным интерфейсом и явной контекстной зависимостью. Такие интерфейсы обычно соответствуют

стандарту языка определения интерфейсов. Компонент играет свою роль в общей композиции и соответствует классу или коллекции нескольких классов.

Итак, мы рассмотрели всю терминологию, необходимую для рассмотрения практических преимуществ и ловушек объектно-ориентированного подхода. Теперь можно приступать к изучению некоторых конкретных языков, чтобы оценить, насколько они соответствуют теоретической идее, описанной выше. Приведенная в главе терминология описана в словаре терминов. Его можно использовать как терминологическую *памятку*.

1.4. Резюме

В этой главе представлена терминология объектно-ориентированных методов и объектно-ориентированного программирования. Обсуждение начато с терминологии в ее историческом и коммерческом контексте, рассмотрены ее истоки, а также связанные с ней ключевые понятия разработки программного обеспечения. К ним относятся следующие.

- Обеспечение возможности повторного использования и расширяемость модулей
- Индустриализация процесса разработки программного обеспечения
- Создание систем с открытыми интерфейсами и совместным использованием ресурсов
- Понимание значения спецификации

Объектно-ориентированный язык (или система) обеспечивает две характерные особенности: инкапсуляцию и наследование.

Абстракция — это процесс выявления существенных объектов в приложении и пренебрежения несущественными свойствами. Абстракция дает возможность повторного использования и сокрытия информации за счет инкапсуляции. Инкапсуляция заключается в сокрытии реализации объектов и открытом провозглашении спецификации их поведения посредством множества атрибутов и операций. Структуры данных и методы, которые обеспечивают выполнение этого принципа, являются закрытыми для объекта.

Тип или класс объекта аналогичен типу данных или типу сущностей с инкапсулированными методами. Данные и методы инкапсулированы и скрыты внутри объектов. Классы могут иметь конкретные экземпляры. Термин *объект* в этой книге означает или класс, или экземпляр.

Наследование — это возможность обобщения и конкретизации понятия или его классификации. Подклассы наследуют свойства и методы от суперклассов и могут добавлять собственные или перекрывать унаследованные. В большинстве объектно-ориентированных языков программирования экземпляры наследуют все без исключения свойства своих классов и только их. Множественное наследование имеет место, когда класс является подклассом более чем одного класса. Наследование обеспечивает расширяемость, но может затруднить возможность повторного использования. Множественное наследование — это мощный инструмент. Однако оно вызывает дополнительные проблемы, и поэтому его нужно использовать с осторожностью.

Объекты взаимодействуют только путем передачи сообщений. Полиморфизм или перегрузка улучшает удобочитаемость текста и производительность программиста, но может привести к снижению эффективности самой программы.

```
объектно-базированный = инкапсуляция + объектная идентичность
классово-базированный = объектно-базированный + множественная
абстракция
объектно-ориентированный = классово-базированный + наследова-
ние + саморекурсия
компонентно-ориентированный = объектно-ориентированный + внеш-
ние интерфейсы
```

1.5. Дополнительная литература

Первыми работами по объектно-ориентированному программированию, возможно, являются [207], [435], [408] и [303].

Существует множество хороших вводных книг по объектно-ориентированному программированию и его преимуществам, хотя в них и не рассказывается об объектно-ориентированных методах вообще. На мой взгляд, наилучшая книга — это [544], основная задача которой — описание языка Eiffel. Однако в этой книге также на высоком уровне представлено введение в основы объектно-ориентированного подхода. Во втором издании книги [549] изложена современная версия тех же идей, а также представлен значительно более завершенный обзор методов из данной научной области. Однако теперь эта книга стала очень объемной и несколько утомительной. Еще одна чудесная книга такого же типа [198] представляет описание языка Objective C. Ее можно назвать хорошим высокоуровневым введением в объектно-ориентированные концепции, кроме того, она весьма хорошо написана. Ни одна книга не содержит достаточно материала об объектно-ориентированном анализе и базах данных, но в [549] изложены интересные идеи, относящиеся к вопросам объектно-ориентированного проектирования. Книга [98] — это самое первое введение в объектно-ориентированное программирование, его концепции и проектирование. Лучшими введениями в объектную технологию на уровне менеджмента проектов являются [750] и [548].

Те читатели, которые желают изучить этот вопрос глубже, могут рассмотреть огромный объем исследовательских статей, часть из которых приведена в списке литературы в конце книги. Две старые, но полезные и доступные работы [712] и [448] содержат некоторое количество подлинно технического и чисто теоретического материала. Работа [516] — это образец совсем недавних исследований этого типа. Книга [83] объединяет несколько хороших вводных статей, а [828] — это огромная коллекция статей по всем аспектам объектно-ориентированных методов. Труды ежегодных конференций OOPSLA и TOOLS всегда включают много полезного и современного материала.

В качестве источников прикладной и теоретической информации следует упомянуть два журнала — это *Journal of Object-Oriented Programming*, JOOP (журнал по объектно-ориентированному программированию) и *Object-Oriented Systems* (Объектно-ориентированные системы), который содержит больше теоретического материала.

Различие между объектно-базированными, классово-базированными и объектно-ориентированными системами изначально было сделано в статье, содержащейся в уже упомянутой ранее книге [712].

В этой книге под объектами понимаются и классы, и экземпляры, несмотря на то что в современных работах объекты и экземпляры различаются. Автор решил не нарушать этой негласной договоренности, что, надеюсь, покажет остальная часть книги.

1.6. Упражнения

1. Выберите **два** характерных свойства ОО подхода.
   - **а)** полиморфизм
   - **б)** наследование
   - **в)** возможность повторного использования
   - **г)** абстракция
   - **д)** инкапсуляция
   - **е)** обобщение
   - **ж)** сокрытие информации
   - **з)** объектная идентичность
   - **и)** динамическое связывание
2. Какая концепция искусственного интеллекта является наиболее близкой к понятию объекта?
   - **а)** слот
   - **б)** механизм вывода
   - **в)** база знаний
   - **г)** фрейм
   - **д)** фацет
   - **е)** правило
3. Какая разница между перечисленными ниже понятиями (сформулируйте по одному предложению для каждого пункта)?
   - **а)** экземпляр и класс
   - **б)** тип данных и класс
   - **в)** класс и роль
   - **г)** тип объекта и тип сущности
   - **д)** класс и компонент
   - **е)** динамическое связывание и полиморфизм
   - **ж)** обобщение и наследование
   - **з)** наследование и классификация
4. Обоснуйте включение атрибутов в описание класса.

5. Укажите различия между чисто ОО стилем наследования и наследованием, применяемым в ИИ-системах. Проследите, поддерживают ли они перечисленные ниже концепции.
   - **а)** динамическая классификация
   - **б)** динамическая специализация
   - **в)** множественное наследование
   - **г)** множественная классификация
   - **д)** наследования значений
   - **е)** перекрытие
   - **ж)** заменяемость
6. Дайте определение идентичности объекта.
7. Сформулируйте определения и приведите примеры следующих понятий.
   - **а)** атрибут/метод класса
   - **б)** атрибут/метод экземпляра
8. Что такое множественное наследование? Когда оно используется?
9. Прокомментируйте следующее высказывание. *“Множественное наследование чрезвычайно опасно; его необходимо запретить. Даже одиночное наследование необходимо устранить из наших языков и методов.”*
10. Прокомментируйте следующее высказывание. *“Объектная технология умерла. Ей на смену пришла компонентно-ориентированная разработка.”*
11. Постройте диаграммы структур наследования и композиции для классов, упомянутых при обсуждении системы SACIS выше в этой главе.