2.5. Классы

Понятия класса и объекта тесно связаны: невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие между этими понятиями. Объект обозначает конкретную сущность, определенную во времени и в пространстве. класс определяет данные и поведение, которыми должен обладать объект. Например, для объекта дом №2 по улице Лесной можно создать класс House (Дом), который будет определять, что у дома должны быть высота, ширина, длинна и количество квартир. Тогда у объекта класса House - дома №2 по улице Лесной - могут быть высота 15 метров, ширина 10 метров, длинна 50 метров и 60 квартир. Класс - более общий термин, являющийся, по существу, шаблоном для объектов. Класс можно сравнить с проектом дома, а объекты - с построенными по проекту домами. Таким образом, ***класс*** - это некое множество объектов, имеющих общую структуру и общее поведение. Любой конкретный объект является экземпляром класса. Объект не является классом, хотя класс может быть объектом.

Класс изображают в виде прямоугольника, разделенного на три части (рис. 2.10). В верхней части прямоугольника содержится имя и (необязательно) стереотип класса. Средний раздел включает в себя атрибуты класса. В нижней секции описываются операции или поведение класса.

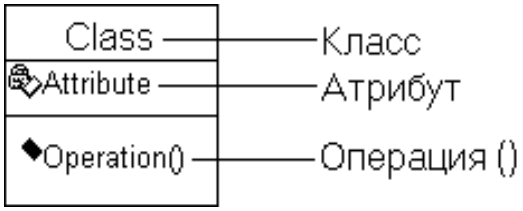


Рис. 2.10. Графическая нотация для изображения класса

<<Entity>> Employee
- Employee_ID:integer=0 # $ Salary:float + Address:string + State:string
Hire(new_dept:String, new_salary:float=50):integer Fire() + Transfer() + GetSalary()

Рис. 2.11. Пример класса Employee (Сотрудник)

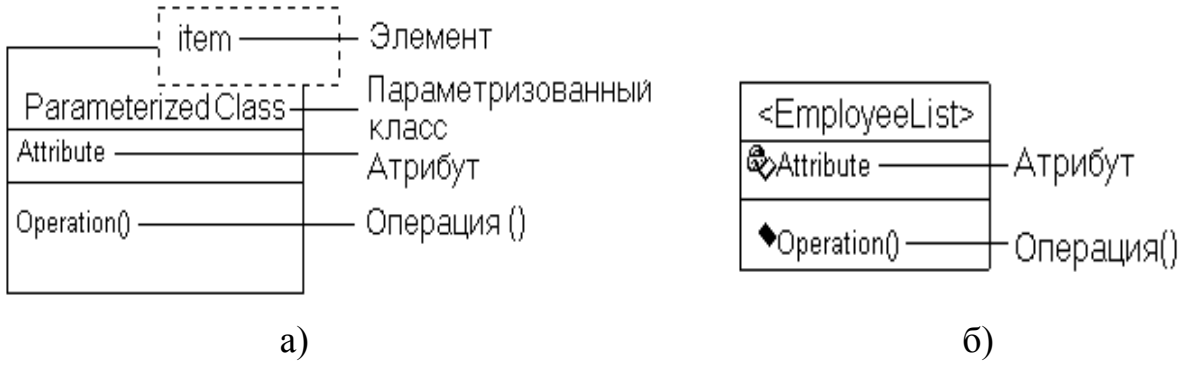
Атрибут – это некоторая информация, характеризующая класс. Например, класс Employee (Сотрудник) (рис. 2.11) имеет четыре атрибута: Employee_ID (Идентификационный номер), Salary (Оклад), Address (Адрес), State (Статус). ***Операции*** класса отражают его поведение (действия, выполняемые классом). Для класса Account определены четыре операции: Hire (Нанять), Fire (Уволить), Transfer (Перевести), GetSalary (Установить оклад).

2.5.1. Типы и стереотипы классов

Кроме регулярных классов доступны классы следующих типов: параметризованные, классы-наполнители, утилиты классов, утилиты параметризованных классов, утилиты классов-наполнителей, метаклассы.

Параметризованный класс (parameterized class) применяется для создания семейства других классов, его еще называют шаблоном (контейнером).

Параметризованный класс изображается с помощью нотации показанной на рис. 2.12, а. В прямоугольнике, выделенном пунктирными линиями, указываются аргументы параметризованного класса. Изменяя значения аргументов, мы получаем стандартные классы. Например, параметризованный класс List (Список) позволяет создать такие классы, как EmployeeList (Список сотрудников), OrderList (Список заказов) и AccountList (Список счетов). Для этого в приведенной выше нотации нужно заменить параметр «Элемент» на соответствующий специфический элемент Employee (Сотрудник), Order (Заказ) или Account (Счет).



а) б)

Рис. 2.12. Графическая нотация для изображения:
а) параметризованного класса; б) класса-наполнителя

Аргументом параметризованного класса может быть другой класс, тип данных или выражение-константа. Можно задавать неограниченное количество аргументов.

Класс-наполнитель (instantiated class) является параметризованным классом, аргументы которого имеют фактические значения. Для рассмотренного выше примера это может быть класс EmployeeList (Спосок сотрудников). В соответствии с нотацией UML, название аргумента класса-наполнителя заключается в угловые скобки (< >) (рис. 2.12, б).

Если в системе есть совокупность функций, которые используются всей системой и не слишком хорошо подходят для какого-либо конкретного класса, эти функции можно объединить в ***утилиту класса*** (class utility), которая будет использоваться всеми классами системы. Утилиты классов часто применяют для расширения функциональных возможностей языка программирования или для хранения общих элементов функциональности многократного использования, необходимых в нескольких системах. На диаграмме утилита класса выглядит как класс с тенью (рис. 2.13).

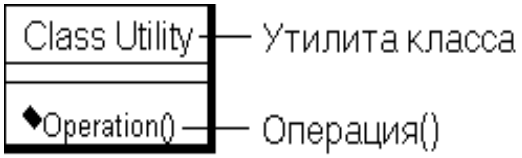
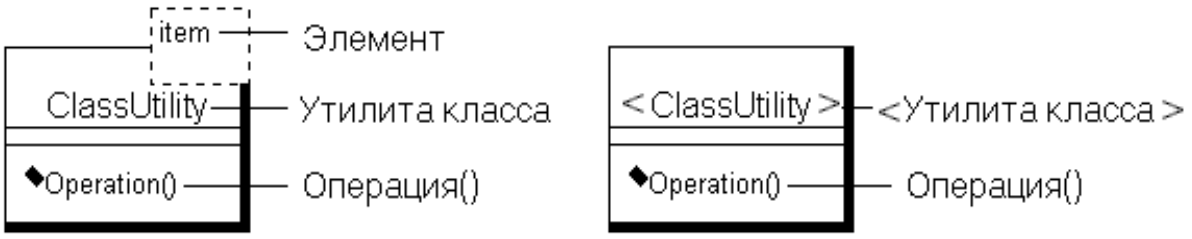


Рис. 2.13. Графическая нотация для изображения утилиты класса

Утилитой параметризованного класса (parameterized class utility) является параметризованный класс, содержащий только набор операций. Это шаблон для создания утилит класса. Утилита параметризованного класса изображается с помощью нотации показанной на рис. 2.14, а.



а) б)

Рис. 2.14. Графическая нотация для изображения:
а) утилиты параметризованного класса; б) утилиты класса-наполнителя

Утилитой класса-наполнителя (instantiated class utility) называется утилита параметризованного класса, параметры которой имеют фактические значения. Утилита класса-наполнителя изображается с помощью нотации показанной на рис. 2.14, б.

Метакласс (metaclass) - это класс, экземпляры которого являются классами, а не объектами. К числу метаклассов относятся параметризован-

ные классы и утилиты параметризованных классов. Метаклассы изображают с помощью нотации показанной на рис.2.15.

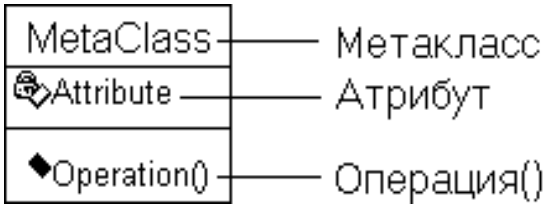


Рис. 2.15. Графическая нотация для изображения метакласса

Абстрактным называется класс, который не наполняется конкретным содержанием (не инстанцируется). Иными словами, если класс А абстрактный, в памяти никогда не будет объектов типа А.

Обычно абстрактные классы применяют при работе с наследованием. В них содержатся данные и поведение, общие для нескольких других классов. Например, у нас может быть класс Животное с атрибутами Рост, Цвет и Разновидность. От этого класса наследуют три других класса: Кошка, Собака и Птица. Каждый из них наследует свойства Рост, Цвет и Разновидность от класса Животное, а также имеет свои собственные уникальные атрибуты и операции. Объекты класса Животное не создаются во время работы приложения - все объекты являются только кошками, собаками и птицами. Класс Животное является абстрактным, он описывает, что общего есть у кошек, собак и птиц. В нотации UML название абстрактного класса пишут курсивом (рис.2.16).

Рис. 2.16. Графическая нотация для изображения абстрактного класса

Стереотип - это механизм, позволяющий классифицировать классы. Допустим, Вы хотите найти в модели все формы. Для этого можно создать стереотип Form (Форма) и назначить его всем окнам приложения. В дальнейшем при поиске форм нужно искать только классы с этим стереотипом. На языке UML определены три основных стереотипа: Boundary (Граница), Entity (Объект) и Control (Управление).

Пограничными классами (boundary classes) называются такие классы, которые расположены на границе системы со всем остальным миром. Они включают в себя формы, отчеты, интерфейсы с аппаратурой (принтер,

сканер и т.п.) и интерфейсы с другими системами. Пограничные классы изображают с помощью нотации показанной на рис. 2.17, а.

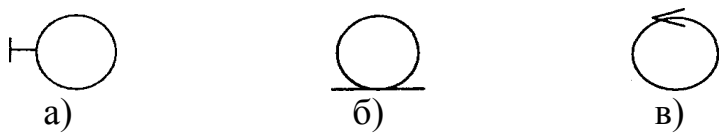


Рис 2.17. Графическая нотация для изображения:
а) пограничного класса; б) класса-сущности; в) управляющего класса

Для выявления пограничных классов необходимо исследовать диаграммы Вариантов Использования. Для каждого взаимодействия между действующим лицом и вариантом использования должен существовать хотя бы один пограничный класс (рис. 2.18, а). Именно он позволяет действующему лицу взаимодействовать с системой. Необязательно создавать уникальные пограничные классы для каждой пары «действующее лицо - вариант использования». Например, если два действующих лица инициируют один и тот же вариант использования, они могут применять для взаимодействия с системой общий пограничный класс (рис. 2.18, б).

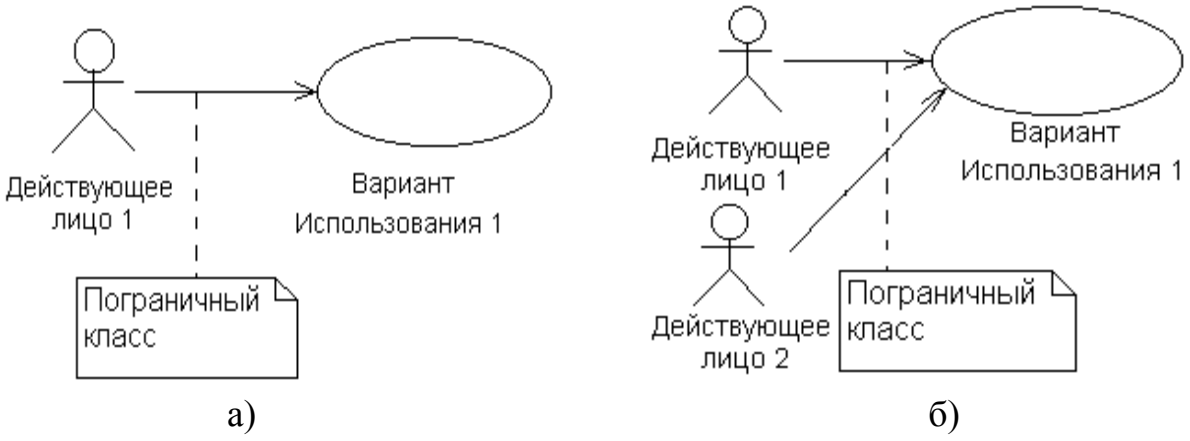


Рис. 2.18. Графическая нотация для изображения:
а) пограничного класса одного варианта использования; б) общего пограничного класса

Классы-сущности (entity classes) содержат информацию, хранимую постоянно. Классы-сущности можно обнаружить в потоке событий и на диаграммах Взаимодействия. Они имеют наибольшее значение для пользователя, и потому в их названиях часто применяют термины из предметной области. В системе работы с данными о сотрудниках примером такого класса является класс Employee (Сотрудник). Классы-сущности изображают с помощью нотации показанной на рис. 2.17, б.

Объектно-ориентированный подход может применяться для создании таблиц баз данных. Вместо того чтобы с самого начала задавать струк-

туру базы данных, ее разрабатывают на основе информации, получаемой из объектной модели. Из требований к системе определяют потоки событий. Из потоков событий определяют объекты, классы и атрибуты классов. Каждый атрибут класса-сущности становится полем в базе данных. Применяя такой подход, легко отслеживать соответствие между полями базы данных и требованиями к системе, что уменьшает вероятность сбора ненужной информации.

Управляющие классы (control classes) отвечают за координацию действий других классов. Обычно у каждого варианта использования имеется один управляющий класс, контролирующий последовательность событий этого варианта использования. Сам управляющий класс не несет в себе никакой функциональности, другие классы посылают ему мало сообщений, но сам он посылает множество сообщений. Управляющий класс делегирует ответственность другим классам. По этой причине управляющий класс часто называют классом-менеджером. Управляющие классы изображают с помощью нотации показанной на рис. 2.17, в.

На рис. 2.19 приведен пример диаграммы Взаимодействия с управляющим классом ControlObject, который отвечает за координацию.

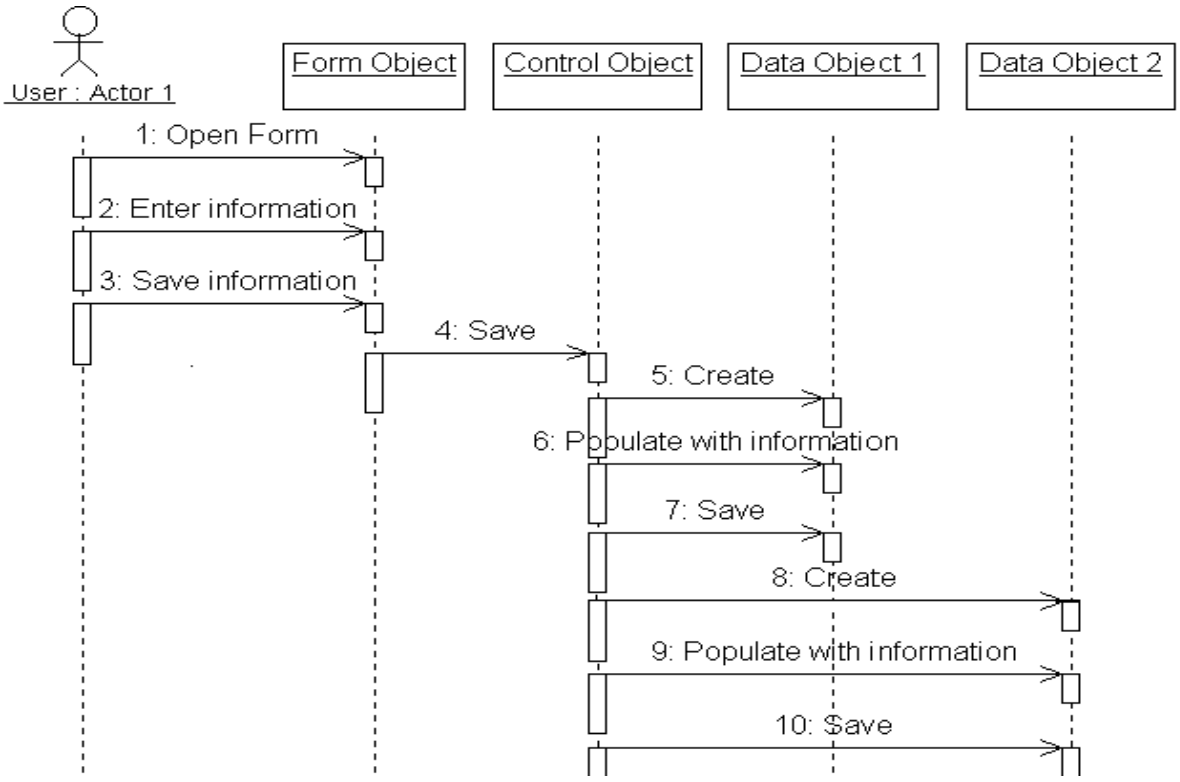


Рис. 2.19. Управляющий класс на диаграмме Последовательности

В системе могут применяться управляющие классы, общие для нескольких вариантов использования. Например, класс SecurityManager (Менеджер безопасности) может отвечать за контроль событий, связанных с

безопасностью, а класс TransactionManager (Менеджер транзакций) - заниматься координацией сообщений, относящихся к транзакциям с базой данных. Могут быть и другие менеджеры для работы с такими элементами функционирования системы, как разделение ресурсов, распределенная обработка данных и обработка ошибок. С помощью управляющих классов можно изолировать функциональность системы. Инкапсуляция в один класс, например координации безопасности, минимизирует последствия вносимых изменений. Любые изменения логики системы отвечающей за безопасность затронут только менеджера безопасности.

2.5.2. Подходы к выявлению ключевых абстракций системы

Рассмотрим несколько подходов к анализу объектно-ориентированных систем с целью выявления ключевых абстракций.

1. Уэнди и Майкл Боггс в книге «Mastering UML with Rational Rose» предлагают ***подход к выявлению ключевых абстракций системы, основанный на использовании UML.***

Выявление классов можно начать с изучения имен существительных в описании потоков событий [2]. В общем случае имена существительные могут соответствовать следующим элементам: действующее лицо, класс и атрибут класса.

Классы также можно обнаружить, анализируя диаграммы Взаимодействия. На диаграммах нужно определить похожие объекты и создать для них общий класс. Допустим, диаграмма Последовательности описывает процесс выплаты зарплаты Джону и Фреду. Атрибуты обоих объектов Джон и Фред сходны: имя, адрес и номер телефона. Операции тоже одинаковы: нанять или уволить сотрудника. Таким образом, можно создать класс Employee (Сотрудник), который будет шаблоном для объектов Джон и Фред.

Не все классы можно обнаружить в потоках событий и на диаграммах Взаимодействия. При выявлении классов следует также рассмотреть три возможных стереотипа: сущность (entity), границу (boundary) и управление (control).

2. Разные ученые предлагают свои источники классов и объектов. Такие подходы считаются ***классическими подходами к выявлению ключевых абстракций системы.***

Шлеер и Меллор предлагают следующих кандидатов в классы и объекты:

1) осязаемые предметы (автомобили, телеметрические данные, датчики давления и т.п.);
2) роли (мать, учитель, политик и т.п.);
3) события (посадка, прерывание, запрос и т.п.);

4) взаимодействие (перечисление, пересечение, заем и т.п.).

Роусс, исходя из моделирования баз данных, предлагает следующих кандидатов в классы и объекты:

1) люди (человеческие существа, выполняющие некоторые функции);
2) места (области связанные с людьми или предметами);
3) предметы (осязаемый материальный объект или группы объектов);
4) организации (формально организованная совокупность людей, ресурсов, оборудования, которая имеет определенную цель и существование которой от индивидуумов не зависит);
5) концепции (принципы и идеи, не осязаемые сами по себе, но предназначенные для организации деятельности, общения или для наблюдения за ними);
6) события (нечто случающееся с чем-то в заданное время или последовательно).

Коад, Йордан предлагают следующих кандидатов в классы и объекты:

1) структуры (отношение «целое-часть» и «общее-частное»);
2) другие системы (внешние системы, с которыми взаимодействует приложение);
3) устройства, с которыми взаимодействует приложение;
4) события (происшествия, которые должны быть запомнены);
5) разыгрываемые роли (роли, которые исполняют пользователи, работающие с приложением);
6) места (здания, офисы и другие места, существенные для работы приложения);
7) организационная единица (группы, к которым принадлежат пользователи).

На более высоком уровне абстракции Коад вводит понятие предметной области, которая в сущности является логически связанной группой классов, относящейся к высокоуровневым функциям системы.

3. В то время как классические подходы концентрируют внимание на осязаемых элементах предметной области, ***анализ поведения*** рассматривает в качестве первоисточника объектов и классов динамическое поведение системы. Классы формируют, основываясь на группах объектов, демонстрирующих сходное поведение.

Вирфс-Брок предлагает следующую схему формирования классов. Объекты, которые имеют сходные ответственности, объединяются в общий класс. В иерархии классов каждый подкласс, выполняя обязательства суперкласса, может привносить свои дополнительные услуги.

Рубин и Гольдберг предлагают идентифицировать классы и объекты, анализируя функционирование системы: «Наш подход основан на изуче-

нии поведения системы. Мы сопоставляем формы поведения с частями системы и пытаемся понять, какая часть инициирует поведение, и какие части в нем участвуют... Инициаторы и участники, играющие существенные роли, опознаются как объекты и делаются ответственными за эти роли» [3].

Идеи Рубина тесно связаны с подходом, предложенным в 1979 году Альбрехтом. Данный проход ориентирован на функции. По Альбрехту, функция определяется как «отдельное бизнес-действие конечного пользователя», то есть: ввод/вывод, запрос, файл или интерфейс [3]. Эта концепция происходит из области информационных систем, однако, может быть применена к любой автоматизированной системе. По существу, функция - это любое достоверно видимое извне и имеющее отношение к делу поведение системы.

4. Еще один подход к выявлению ключевых абстракций системы основан на ***анализе предметной области***.

Иногда в поисках полезных и доказавших свою работоспособность идей имеет смысл обратиться сразу к нескольким приложениям в рамках рассматриваемой предметной области. Если Вы испытываете затруднения в процессе разработки ПО, исследуйте уже имеющиеся подобные системы. Это поможет понять, какие ключевые абстракции и механизмы, использованные в них, будут полезны, а какие нет. В качестве примера рассмотрим систему бухгалтерского учета. Допустим, она должна представлять различные виды отчетов. Если считать работу с отчетами предметной областью, ее анализ может привести разработчика к пониманию ключевых абстракций и механизмов, которые обслуживают все виды отчетов. Полученные таким образом классы и объекты будут представлять собой множество ключевых абстракций и механизмов, отобранных с учетом одной из целей исходной задачи - создания системы отчетов.

Идею анализа предметной области впервые предложил Нейборс. Такой анализ определяют еще, как «попытку выделить те объекты, операции и связи, которые эксперты данной области считают наиболее важными» [3]. Мур и Байлин определяют следующие этапы анализа предметной области:

1) «построение скелетной модели предметной области при консультации с экспертами в этой области;
2) изучение существующих в данной области систем и представление результатов в стандартном виде;
3) определение сходства и различий между системами при участии экспертов;
4) уточнение общей модели для приспособления к нуждам конкретной системы» [3].

В роли эксперта часто выступают обычные пользователи системы, например, инженер или диспетчер. Эксперт не обязательно должен быть

программистом, он должен быть близко знаком с исследуемой областью и «разговаривать» на ее языке. Сотрудничество пользователей и разработчиков системы имеет важное значение.

Анализ предметной области лучше вести попеременно с проектированием - немного поанализировать, затем немного попроектировать и т.д. Данный подход достаточно эффективен, исключая лишь узко специализированные области, но такие уникальные программные системы встречаются редко.

5. В отдельности классический подход, поведенческий подход и изучение предметной области, рассмотренные выше, сильно зависят от индивидуальных способностей и опыта аналитика. Для большинства реальных проектов одновременное применение всех трех подходов неприемлемо, так как процесс анализа становится недетерминированным и непредсказуемым. ***Анализ вариантов*** - это подход, который можно успешно сочетать с перечисленными, делая их применение более упорядоченным. Впервые его формализовал Джекобсон, определивший вариант применения, как «частный пример или образец использования, сценарий, начинающийся с того, что пользователь системы инициирует операцию или последовательность взаимосвязанных событий» [3].

Опишем этапы проведения анализа вариантов. Этот вид анализа можно начинать вместе с анализом требований, когда пользователи, эксперты и разработчики перечисляют сценарии, наиболее существенные для работы системы (пока не углубляясь в детали). Затем сценарии тщательно прорабатывают, раскладывая их по кадрам, как это делают телевизионщики и кинематографисты [3]. При этом устанавливается, какие объекты участвуют в сценарии, каковы обязанности каждого объекта и как они взаимодействуют в терминах операций. Тем самым четко распределяются области влияния абстракций. Далее набор сценариев расширяется, чтобы учесть исключительные ситуации и вторичное поведение. В результате появляются новые или уточняются существующие абстракции.

6. Следующий подход к выявлению ключевых абстракций системы использует ***CRC-карточки***. CRC обозначает Class-Responsibilities-Collaborators (Класс/Ответственности/Участники). Это простой и эффективный способ анализа сценариев. Карты CRC впервые предложили Бек и Каннингхэм для обучения объектно-ориентированному программированию, но такие карточки оказались хорошим инструментом для «мозгового штурма» и общения разработчиков между собой.

CRC-карты – это обычные библиографические карточки размером 3 на 5 или 5 на 7 дюймов. Хорошо, если карточки будут линованными и разноцветными. На карточках записывают карандашом сверху - название класса, снизу в левой половине - за что он отвечает, а в правой половине - с кем он взаимодействует. Проходя по сценарию, заводят по карточке на каждый обнаруженный класс. Полученные абстракции анализируют, при

этом можно либо выделить излишек ответственности в новый класс, либо перенести ответственности с одного большого класса на несколько более детальных классов, либо передать часть обязанностей другому классу.

Карточки позволяют представить разные формы взаимодействия объектов. С точки зрения динамики сценария их расположение может показать поток сообщений между объектами, с точки зрения статики они представляют иерархии классов.

7. ***Метод использования неформального описания задачи*** с целью выявления ключевых абстракций системы впервые был предложен Абботом. Согласно ему, нужно описать задачу или ее часть на простом русском языке, а потом подчеркнуть имена существительные и глаголы [3]. Имена существительные - кандидаты на роль классов, а глаголы могут стать именами операций.

Данный метод прост и заставляет разработчиков заниматься словарем предметной области. Однако он весьма приблизителен и не подходит для сложных проблем. Человеческий язык – довольно неточное средство выражения, поэтому список объектов и операций зависит от умения разработчика записывать свои мысли. Кроме этого для многих существительных можно найти соответствующую глагольную форму и наоборот.

8. Еще один способ использует в качестве источника ключевых абстракций системы продукты ***структурного анализа***. После проведения структурного анализа уже существует модель системы, описанная диаграммами потоков данных и другими продуктами структурного анализа. На основе этих моделей можно определить классы и объекты тремя различными способами.

Мак Менамин и Палмер предлагают сначала сформировать словарь данных, а затем приступить к анализу контекстных диаграмм модели. Они утверждают следующее: «Рассматривая список основных структур данных, следует подумать, что они описывают. Например, если они имена прилагательные, то какие имена существительные они описывают? Ответы на такие вопросы могут пополнить список объектов» [3]. Источниками объектов в этом случае являются: предметная область, входные и выходные данные, услуги и других ресурсы.

Следующий способ основан на анализе диаграмм потоков данных. В этом случае кандидатами в объекты могут быть:

- внешние сущности;
- хранилища данных;
- хранилища управляющих сущностей;
- управляющие преобразования.

Кандидатами в классы являются потоки данных и потоки управления. Преобразование данных можно рассматривать как операции над объектами или как поведение некоторого объекта.

Зайдевиц и Старк предлагают еще один метод, который они называют анализом абстракций. В структурном анализе входные и выходные данные изучаются до тех пор, пока не достигнут наивысшего уровня абстракции. Процесс преобразования входных данных в выходные рассматривается как основное преобразование. В абстрактном анализе разработчик изучает это преобразование для того, чтобы определить, какие процессы и состояния являются самыми важными. Так определяются основные сущности. Затем с помощью диаграмм потоков данных изучают всю инфраструктуру, прослеживая входящие и исходящие из центра потоки данных, и группируют встречающиеся на пути процессы и состояния.

Следует отметить, что принципы структурного проектирования, который основан на структурном анализе, полностью ортогональны принципам объектно-ориентированного проектирования. Диаграммы потоков данных представляют собой скорее описание проекта, чем модель существа системы. Кроме этого трудно построить объектно-ориентированную систему, если модель ориентирована на алгоритмическую декомпозицию. Поэтому лучше использовать как подготовительный этап для объектно-ориентированного проектирования объектно-ориентированный анализ.

2.5.3. Атрибуты

Атрибут - это фрагмент информации связанный с классом. Рассмотрим основные типы, характеристики и способы выявления атрибутов.

С атрибутами можно связать три основных фрагмента информации: имя атрибута, тип его данных и первоначальное значение.

Тип данных атрибута специфичен для используемого языка. Это может быть, например, тип string, integer, long или boolean.

Рассмотрим пример определения первоначального значения (значения по умолчанию). Допустим, атрибут TaxRate (Ставка налога с покупки) класса Order (Заказ) для какого-либо города равняется 7.5%. Следовательно, для него можно определить значение по умолчанию, равное 0.075. Значение по умолчанию атрибута задавать не обязательно. В описании атрибута оно указывается после знака «=» (рис. 2.11).

При добавлении атрибута к классу каждый экземпляр класса получит свою собственную копию этого атрибута. Рассмотрим, например, класс Employee. В процессе работы приложения можно создать экземпляры двух сотрудников Джона и Билла. Каждый из этих объектов получит свою собственную копию атрибута Salary (Оклад). ***Статичный атрибут*** (static) - это такой атрибут, который используется всеми экземплярами класса. Если бы атрибут Salary был статичным, он был бы общим для Джона и Билла. На языке UML статичный атрибут помечают символом «$». В нашем примере Salary станет «$Salary» (рис. 2.11).

Производным (derived) называется атрибут, полученный из одного или нескольких других атрибутов. Например, класс Rectangle (Прямоугольник) может иметь атрибуты Width (Ширина) и Height (Высота). У него также может быть атрибут Area (Площадь), вычисляемый как произведение ширины и высоты. Так как Area получается из этих двух атрибутов, он считается производным атрибутом. В нотации UML производные атрибуты помечают символом «/». В описанном примере атрибут Area следует написать как «/Area» (рис. 2.20).

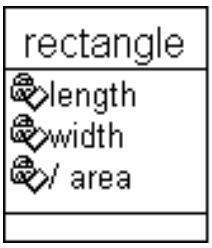


Рис. 2.20. Графическая нотация для изображения производного атрибута

Рассмотрим способы выявления атрибутов.

Один из источников атрибутов - имена существительные в потоке событий. Некоторые из них являются классами или объектами, другие - действующими лицами, третьи - атрибутами. Например, в потоке событий может быть написано: «Пользователь вводит имя сотрудника, его адрес, номер социальной страховки и номер телефона». Это означает, что у класса Сотрудник имеются атрибуты Имя, Адрес, Номер страховки и Номер телефона.

Атрибуты можно также выявить, изучая документацию, описывающую требования к системе. Нужно изучить такие требования, которые определяют собираемые системой данные. Любой элемент такой информации может быть атрибутом класса.

Еще одним источником атрибутов может стать структура базы данных. Если она уже определена, поля в ее таблицах дадут хорошее представление об атрибутах. Часто имеется однозначное соответствие между таблицами базы данных и классами-сущностями. Если вернуться к предыдущему примеру, то таблица Сотрудник будет иметь поля: Имя, Адрес, Номер телефона и Номер страховки. Тогда соответствующий класс Сотрудник будет иметь атрибуты: Имя, Адрес, Номер телефона и Номер страховки. Однако не всегда существует такое однозначное соответствие. Подходы к проектированию баз данных и классов могут различаться. В частности, реляционные базы данных не поддерживают наследование.

Определяя атрибуты, нужно следить за тем, чтобы каждый из них можно было соотнести с требованиями к системе. Если это не удается сделать, нельзя быть уверенным в том, что данное требование нужно за-

казчику. Такой подход помогает решить классическую проблему приложения, собирающего огромный объем никому не нужной информации.

После определения атрибутов их нужно соотнести с классами. Например, класс Сотрудник может содержать имя и адрес, но не должен включать в себя сведения о выпускаемой компанией продукции. Для последних подошел бы класс Продукция.

Обратите внимание на классы, у которых слишком много атрибутов. Возможно, такой класс следует разделить на два меньших. Так, класс с десятью или пятнадцатью атрибутами может быть вполне приемлемым. Только нужно убедиться, что все его атрибуты нужны и действительно должны принадлежать этому классу. Будьте осторожны с классами, у которых слишком мало атрибутов. Вполне возможно, что все нормально - например, управляющий класс имеет мало атрибутов. Однако это может быть и признаком необходимости в объединении нескольких классов.

Иногда могут возникнуть сомнения, соответствует ли обнаруженная Вами информация атрибуту или классу. Рассмотрим, например, такой атрибут, как название компании. Является ли он атрибутом класса Person (Человек), или лучше создать отдельный класс Company (Компания)? Ответ зависит от того, какое приложение Вы разрабатываете. Если Вы собираете сведения о компании и имеется связанное с ней поведение, она может быть классом. Допустим, что Вы проектируете систему работы с заказчиками. В таком случае может потребоваться информация о компаниях, которым поставляются товары или услуги. Тогда нужно знать, сколько сотрудников работает в компании, ее имя и адрес, контактный телефон и т.д. С другой стороны, специфическая информация о компании может и не требоваться. Допустим, приложение должно генерировать письма людям, работающим в других организациях. При этом достаточно знать названия их фирм. В таком случае имя компании будет атрибутом класса Контакт. Кроме того, нужно рассмотреть, есть ли поведение у подозрительной информации. Если в вашем приложении компания имеет некоторое выраженное поведение, лучше моделировать ее как класс. Если поведения нет, то это скорее всего атрибут.

2.5.4. Операции

Операцией называется связанное с классом поведение. Операции определяют ответственности классов.

Описание операции состоит из трех частей: имени, параметров и типа возвращаемого значения. Аргументы или параметры операции - это получаемые ею входные данные. Тип возвращаемого значения относится к результату действия операции.

В языке UML для операций принята следующая нотация:

имя операции (аргумент1:тип данных аргумента1, аргумент2:тип данных аргумента2, ...):тип возвращаемого значения операции

Для каждого аргумента должны быть заданы имя и тип данных. На диаграмме Классов аргументы и их типы указываются в скобках после имени операции (рис. 2.11). При желании для аргументов можно задавать их значения по умолчанию (рис. 2.11). В таком случае нотация UML будет иметь вид:

имя операции (аргумент1:тип данных аргумента1=значение по умолчанию аргумента1, …):тип возвращаемого значения операции

Выделяют четыре типа операций: операции реализации, операции управления, операции доступа и вспомогательные операции.

Операции реализации (implementor operations) реализуют некоторую бизнес-функциональность. Такие операции можно найти, исследуя диаграммы Взаимодействия. Диаграммы этого типа фокусируются на бизнес-функциональности. Каждое сообщение такой диаграммы можно соотнести с операцией реализации.

Необходимо, чтобы каждую операцию реализации можно было проследить до соответствующего требования. Это достигается на различных этапах моделирования. Операция выводится из сообщения на диаграмме Взаимодействия, сообщения выделяются из подробного описания потока событий, который создается на основе варианта использования, а последний - на основе требований. Возможность проследить всю эту цепочку гарантирует, что каждое требование будет воплощено в коде, а каждый фрагмент кода реализует какое-то требование.

Операции управления (manager operations) управляют созданием и разрушением объектов. В эту категорию попадают конструкторы и деструкторы классов.

Атрибуты обычно бывают закрытыми или защищенными. Тем не менее другие классы иногда должны просматривать или изменять их значения. Для этого предназначены ***операции доступа*** (access operations). Допустим, имеется атрибут Salary (Оклад) класса Employee (Сотрудник). Было бы не желательно, чтобы другие классы могли изменять этот атрибут. Для выполнения этого условия добавим к классу Employee две операции доступа: GetSalary и SetSalary. К первой из них, являющейся общей, могут обращаться все классы. Она получает значение атрибута Salary и возвращает его вызвавшему ее классу. Операция SetSalary также является общей, она позволяет вызвавшему ее классу установить новое значение атрибута Salary. Эта операция может содержать любые правила и условия проверки, которые необходимо выполнить, прежде чем изменить атрибут. Операции доступа дают возможность безопасно инкапсулировать атрибуты внутри

класса, защищая их от других классов, но при этом позволяет осуществлять контролируемый доступ к ним.

Создание операций Get и Set (получения и изменения значения) для каждого атрибута класса является промышленным стандартом.

Вспомогательными (helper operations) называются такие операции класса, которые необходимы ему для выполнения его ответственностей, но о которых другие классы не должны ничего знать. Это закрытые и защищенные операции класса.

Как и операции реализации, вспомогательные операции можно обнаружить на диаграммах Последовательности и Кооперативных диаграммах. Часто такие операции являются рефлексивными сообщениями.

При рассмотрении отдельных типов операций уже описывались способы их выявления, обобщим эти сведения.

Создав диаграммы Взаимодействия, Вы проделаете основную часть работы, требуемой для выявления операций.

Для идентификации операций выполните следующие действия.

1. Изучите диаграммы Взаимодействия. Большая часть сообщений на этих диаграммах является операциями реализации. Рефлексивные сообщения будут вспомогательными операциями.

2. Рассмотрите управляющие операции. Возможно, требуется добавить конструкторы и деструкторы.

3. Рассмотрите операции доступа. Для каждого атрибута класса, с которым будут работать другие классы, необходимо создать операции Get и Set.

При идентификации операций и анализе классов следует иметь в виду следующее.

1. Относитесь с подозрением к любому классу, имеющему только одну или две операции. Возможно, класс написан совершенно правильно, но его следует объединить с каким-нибудь другим классом.

2. С большим подозрением относитесь к классу без операций. Как правило, класс инкапсулирует не только данные, но и поведение. Класс без поведения лучше моделировать как один или несколько атрибутов.

3. С осторожностью относитесь к классу со слишком большим числом операций. Набор ответственностей класса должен быть управляем. Если класс очень большой, им будет трудно управлять. В такой ситуации лучше разделить класс на два меньших.

Как и в случае других элементов модели, для классификации операций создаются их ***стереотипы***. Существуют четыре наиболее распространенных стереотипа операций:

1) Implementor (операции реализации) - операции, реализующие некоторую бизнес-логику;

2) Manager (операции управления) - конструкторы, деструкторы и операции управления памятью;

3) Access (операции доступа) - операции, позволяющие другим классам просматривать или редактировать атрибуты данного класса. Как правило, такие операции называют GetИмя_атрибута или SetИмя_атрибута;

4) Helper (вспомогательные операции) - закрытые или защищенные операции, которые используются классом, но не видны другим классам.

Назначение операциям стереотипов облегчает понимание модели. Кроме того, стереотипы помогают убедиться в том, что ни одна операция не была пропущена. Стереотип указывается перед именем операции в двойных угловых скобках (<< >>), например, <<Entity>> (рис. 2.11).

2.5.5. Понятие видимости

Разделяют внутренне устройство классов - реализацию и внешнее устройство класса интерфейс. В ***интерфейсе*** главное - объявление операций, которые поддерживаются экземплярами классов. К интерфейсу также относятся объявление других классов, переменных, констант и ситуаций, уточняющих абстракцию. ***Реализация*** класса никому кроме самого класса недоступна, она состоит из реализации объявленных в интерфейсе операций.

С интерфейсом класса связана такая характеристика, как видимость. ***Видимость*** показывает, каким образом данные (атрибуты) и поведение (операции) инкапсулируются в класс. В таблице 2.1 приведены допустимые значения этого параметра (см. также рис. 2.11).

Таблица 2.1. Допустимые значения видимости атрибутов и операций

Тип видимости	Обозначение	Описание
public (общий, открытый)	«+»	Атрибут или операция доступные всем классам
private (закрытый)	«-»	Атрибут или операция не доступные другим классам
protected (защищенный)	«#»	Атрибут или операция доступные только самому классу и его потомкам
package or implementation (пакетный)		Атрибут или операция являются общими, но только в пределах своего пакета