МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ЛЕСОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

КАФЕДРА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И МОДЕЛИРОВАНИЯ

Объектно-ориентированный анализ и программирование

Конспект лекций для студентов направления подготовки
38.03.05 Бизнес-информатика всех форм обучения

Составитель: Т.С. Крайнова

ЕКАТЕРИНБУРГ

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Объектно-ориентированный подход в последнее десятилетие стал одним из наиболее интенсивно развивающихся направлений в программировании и наиболее популярным средством разработки программного обеспечения.

Начало развитию объектно-ориентированного подхода положил язык Simula 67, который был разработан в конце 60-х гг. в Норвегии. Несмотря на то, что язык намного опередил свое время, современники (программисты 60-х гг.) оказались не готовы воспринять ценности языка Simula 67, и он не выдержал конкуренции с другими языками программирования (прежде всего, с языком Fortran).

Но достоинства языка Simula 67 были замечены некоторыми программистами, и в 70-е гг. было разработано большое число экспериментальных объектно-ориентированных языков программирования. В результате исследования этих языков были разработаны современные объектно-ориентированные языки программирования: C++, Ada, Smalltalk и др.

Наиболее распространенным объектно-ориентированным языком программирования является язык C++ [1, 6, 8]. Он возник на базе соединения языков C и Simula. C++ был разработан в начале 80-х Бьерном Страуструпом, сотрудником компании AT&T. Все эти годы язык интенсивно развивался, и, наконец, в августе 1998 г. был принят международный стандарт языка C++.

Разработка новых объектно-ориентированных языков программирования продолжается и в настоящее время. Например, с 1995 г. стал широко распространяться объектно-ориентированный язык программирования Java, ориентированный на сети компьютеров и, прежде всего, на Internet. В настоящее время компанией Microsoft разрабатывается новый объектно-ориентированный язык C# (C Sharp), который во многом базируется на языке C++ и также ориентирован на разработку Internet-приложений.

Вместе с развитием объектно-ориентированного программирования стали развиваться и объектно-ориентированные методы разработки программного обеспечения, охватывающие стадии анализа и проектирования. Среди общепризнанных объектно-ориентированных подходов к анализу и проектированию следует выделить методы Г. Буча [3, 4], Д. Рамбо, А. Джекобсона, Шлеера-Меллора и Коуда-Йордона. В результате объединения усилий первых трех авторов появился на свет унифицированный язык моделирования UML [2, 5, 7, 9], который в 1997 г. был принят в качестве стандарта консорциумом Object Management Group и получил широкое распространение в сфере производства программного обеспечения.

Основные идеи объектно-ориентированного подхода опираются на следующие положения:

– программа представляет собой *модель* некоторого реального процесса, части реального мира; модель содержит не все признаки и свойства представ-

ляемой ею части реального мира, а только те, которые существенны для разрабатываемой программной системы;

– модель реального мира или его части может быть описана как совокупность взаимодействующих между собой *объектов*;

– объект описывается набором *атрибутов* (свойств), значения которых определяют состояние объекта, и набором *операций* (действий), которые может выполнять объект;

– взаимодействие между объектами осуществляется посылкой специальных *сообщений* от одного объекта к другому; сообщение, полученное объектом, может потребовать выполнения определенных действий, например изменения состояния объекта;

– объекты, описанные одним и тем же набором атрибутов и способные выполнять один и тот же набор операций, представляют собой *класс* однотипных объектов.

С точки зрения языка программирования класс объектов можно рассматривать как тип данных, а отдельные объекты – как данные этого типа. Определение программистом собственных классов объектов должно позволить описывать конкретную задачу в терминах ее предметной области (при соответствующем выборе имен типов и имен объектов, их атрибутов и выполняемых действий).

Объектно-ориентированный подход дает следующие основные преимущества:

– уменьшение *сложности* программного обеспечения;

– повышение его *надежности*;

– обеспечение возможности *модификации* отдельных компонент программ без изменения остальных компонент;

– обеспечение возможности *повторного использования* отдельных компонент программного обеспечения.

Систематическое применение объектно-ориентированного подхода позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы. Этим объясняется интерес программистов к объектно-ориентированному подходу и объектно-ориентированным языкам программирования.

Целью данного курса лекций является введение в объектно-ориентированный подход к разработке программного обеспечения. В рамках курса рассмотрены концепции и понятия объектно-ориентированного подхода (на основе [4]), а также их выражение на унифицированном языке моделирования UML (на основе [5]).

1. СЛОЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Объектно-ориентированный подход возник в первую очередь в ответ на растущую ***сложность*** программного обеспечения. На заре компьютерной эры

возможности компьютеров были ограничены и было очень трудно написать большую программу. В 60–70-е гг. эффективность применения компьютеров резко возросла, и стало все больше создаваться прикладных программ повышенной сложности. Наибольшее распространение в это время получило структурное проектирование по методу сверху вниз. Однако через некоторое время оказалось, что структурный подход не работает, если объем программы превышает приблизительно 100 тыс. строк. Как результат – выход проектов за рамки установленных сроков и бюджетов и, более того, их несоответствие начальным требованиям. Для решения этих проблем и стали применять объектно-ориентированный подход.

Справедливости ради отметим, что объектно-ориентированный подход является достаточно универсальным инструментом. Он может применяться и для разработки программ малой и средней сложности. Однако, именно для сложных систем использование объектно-ориентированного подхода является критичным. Таким образом, основной областью использования объектно-ориентированного подхода и основным объектом нашего интереса являются ***сложные промышленные программные продукты***.

Подобные системы могут применяться для решения самых разных задач. В качестве примеров можно привести:

– системы с обратной связью (интеллектуальные, самообучающиеся системы), которые активно взаимодействуют или управляются событиями физического мира и для которых ресурсы времени и памяти ограничены;

– задачи поддержания целостности информации объемом в сотни тысяч записей при параллельном доступе к ней с обновлениями и запросами;

– системы управления и контроля над реальными процессами (например, диспетчеризация воздушного и железнодорожного транспорта).

Системы подобного типа обычно имеют большое время жизни, и большое количество пользователей оказывается в зависимости от их нормального функционирования. Глобальные системы национального или даже мирового масштаба являются яркими примерами таких систем (системы управления ядерными объектами, Интернет).

Сложность является существенной чертой промышленной программы: один разработчик практически не в состоянии охватить все аспекты такой системы. Фактически сложность промышленных программ превышает возможности интеллекта одного человека.

Сложность, присущая программному обеспечению, определяется следующими ***основными причинами***:

– сложностью реального мира;

– сложностью управления процессом разработки;

– гибкостью программного обеспечения;

– сложностью описания поведения систем.

Сложность реального мира

Проблемы, которые мы пытаемся решить с помощью разрабатываемого программного обеспечения, часто неизбежно содержат сложные элементы, к которым предъявляется множество различных и нередко противоположных требований.

Но даже в простых проблемах сложность может возникнуть из-за языковой и понятийной "нестыковки" между заказчиком системы и ее разработчиком: пользователи обычно с трудом могут внятно объяснить разработчикам, **что** на самом деле нужно сделать. Часто пользователь лишь смутно представляет, **что** ему нужно от будущей программной системы. С другой стороны, разработчик, являясь экспертом лишь в своей области знаний, недостаточно квалифицирован в предметной области.

Дополнительные сложности возникают в результате изменения требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Использование системы, после того как она разработана и установлена, заставляет пользователей лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

Сложность управления процессом разработки

Большое число требований к системе неизбежно приводит либо к созданию нового программного продукта значительных размеров, либо к модификации существующего, что также не делает его проще. Сегодня обычными стали системы размером в десятки тысяч и даже миллионы строк на языках высокого уровня. Ни один человек не в состоянии понять и разработать полностью такую систему. Поэтому возникает необходимость разбиения системы на подсистемы и модули и коллективная разработка систем.

В идеале для успеха разработки команда разработчиков должна быть как можно меньше. Но какой бы она ни была, всегда возникнут трудности, связанные с координацией работ над проектом, и проблема взаимопонимания требований и спецификаций системы.

Гибкость программного обеспечения

Программирование обладает максимальной гибкостью среди технических наук. Программист, как и писатель, работает со словом, и всеми базовыми элементами, необходимыми для создания программ, он может обеспечить себя сам, зачастую пренебрегая уже существующими разработками. Такая гибкость – чрезвычайно привлекательное, но опасное качество: пользователь, осознав эту возможность, постоянно изменяет свои требования; разработчик увлекается украшательством своей системы во вред основному ее назначению. Поэтому программные разработки остаются очень кропотливым и "бесконечным" делом, а программные системы потенциально незавершенными.

Сложность описания поведения системы

Сложные программные системы содержат сотни и тысячи переменных, текущие значения которых в каждый момент времени описывают состояние программы. Кроме того, они имеют большое количество точек ветвления, которые определяют множество зависящих от ситуации путей решения задачи. Все это разработчик должен продумать, зафиксировать в программах, протестировать и отладить.

Любая сложная система, в том числе и сложная программная система, обладает следующими ***общими признаками***:

1. Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые, в свою очередь, также могут быть разделены на подсистемы и т.д.

Сложная система состоит не просто из отдельных компонент, между ними имеются определенные иерархические отношения.

Например, большинство персональных компьютеров состоит из одних и тех же основных элементов: системного блока, монитора, клавиатуры и манипулятора "мышь". Мы можем взять любую из этих частей и разложить ее, в свою очередь, на составляющие. Системный блок, например, содержит материнскую плату, платы оперативной памяти, центральный процессор, жесткий диск и т.д.

Продолжая, мы можем разложить на составляющие центральный процессор. Он состоит из регистров и схем управления, которые сами состоят из еще более простых деталей: диодов, транзисторов и т.д. Возникает вопрос, что же считать простейшим элементом системы? Ответ дает второй признак.

2. Выбор того, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя.

Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого. Если пользователю достаточно выделить системный блок, монитор и клавиатуру, то для разработчика компьютера этого явно недостаточно.

3. Внутрикомпонентная связь обычно сильнее, чем связь между компонентами.

Это обстоятельство позволяет отделять высокочастотные взаимодействия внутри компонентов от низкочастотных взаимодействий между компонентами и дает возможность относительно изолированно изучать каждую компоненту.

4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.

Иными словами, разные сложные системы содержат одинаковые структурные части. Эти части, в свою очередь, могут использовать общие более мелкие компоненты. Например, и у растений, и у животных имеются крупные подсистемы типа сосудистых систем, и клетки как более мелкие компоненты.

5. Любая работающая сложная система является результатом развития работавшей более простой системы.

В качестве примера назовем теорию эволюции живой природы.

Сложная система, спроектированная с нуля, вряд ли заработает. Следует начать с работающей простой системы.

В процессе развития системы объекты, первоначально рассматривавшиеся как сложные, становятся элементарными, и из них строятся более сложные системы.

2. ОБЪЕКТНАЯ МОДЕЛЬ

Объектно-ориентированный подход основывается на совокупности ряда принципов, называемой ***объектной моделью***. Главными принципами являются

– абстрагирование;

– инкапсуляция;

– модульность;

– иерархичность.

Эти принципы являются главными в том смысле, что без них модель не будет объектно-ориентированной. Кроме главных, назовем еще три дополнительных принципа:

– типизация;

– параллелизм;

– сохраняемость.

Называя их дополнительными, мы имеем в виду, что они полезны в объектной модели, но не обязательны.

2.1. Абстрагирование

Люди развили чрезвычайно эффективную технологию преодоления сложности. Мы абстрагируемся от нее. Если мы не в состоянии полностью воссоздать сложный объект, то приходится игнорировать не слишком важные детали. В результате мы имеем дело с обобщенной, идеализированной моделью объекта.

Например, изучая процесс фотосинтеза у растений, мы концентрируем внимание на химических реакциях в определенных клетках листа и не обращаем внимание на остальные части – черенки, жилки и т.д.

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

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных. Такое разделение смысла и реализации называют ***барьером абстракции***. Установление того или иного барьера абстракции порождает

множество различных абстракций для одного и того же предмета или явления реального мира. Абстрагируясь в большей или меньшей степени от различных аспектов проявления реальности, мы находимся на разных ***уровнях абстракции***.

Для примера рассмотрим системный блок компьютера. Пользователю, использующему компьютер для набора текста, не важно, из каких частей состоит этот блок. Для него это – коробка белого цвета с кнопками и емкостью для дискеты. Он абстрагируется от таких понятий, как "процессор" или "оперативная память". С другой стороны, у программиста, пишущего программы в машинных кодах, барьер абстракции лежит намного ниже. Ему необходимо знать устройство процессора и команды, понимаемые им.

Является полезным еще один дополнительный принцип, называемый ***принципом наименьшего удивления***. Согласно ему абстракция должна охватывать все поведение объекта, но не больше и не меньше, и не привносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.

Например, нам необходимо использовать структуру данных, аналогичную стеку (с доступом, осуществляемым по правилу "первым вошел, последним вышел"), однако требуется проверять наличие в "стеке" некоторого элемента. Если мы назовем эту структуру данных стеком и предложим постороннему программисту, он очень удивится, заметив "лишнюю" операцию.

Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном устройстве, имеет имя и содержимое. Эти атрибуты являются статическими свойствами. Конкретные же значения каждого из перечисленных свойств динамичны и изменяются в процессе использования объекта: файл можно увеличить или уменьшить, изменить его имя и содержимое.

Будем называть ***клиентом*** любой объект, использующий ресурсы другого объекта, называемого ***сервером***. Мы будем характеризовать поведение объекта ***услугами***, которые он оказывает другим объектам, и ***операциями***, которые он выполняет над другими объектами. Этот подход концентрирует внимание на внешних проявлениях объекта и реализует так называемую ***контрактную модель программирования***. Эта модель заключается в следующем: внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство (часто – во взаимодействии с другими объектами). Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом. Другими словами, этот контракт определяет ***ответственность*** объекта – то поведение, за которое он отвечает.

Каждая операция, предусмотренная контрактом, однозначно определяется ее ***сигнатурой*** – списком типов формальных параметров и типом возвращаемого значения. Полный набор операций, которые клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются, называется ***протоколом***. Протокол отражает все возможные спо-

собы, которыми объект может действовать или подвергаться воздействию. Тем самым протокол полностью определяет внешнее поведение абстракции.

Пример. В тепличном хозяйстве, использующем гидропонику, растения выращиваются на питательном растворе без песка, гравия и другой почвы. Управление режимом работы парниковой установки – очень ответственное дело. Оно зависит как от вида выращиваемых культур, так и от стадии выращивания. Нужно контролировать целый ряд факторов: температуру, влажность, освещение, кислотность и концентрацию питательных веществ. В больших хозяйствах для решения этой задачи часто используют автоматические системы, которые контролируют и регулируют указанные факторы. Цель автоматизации состоит здесь в том, чтобы при минимальном вмешательстве человека добиться соблюдения режима выращивания.

Одна из ключевых абстракций в данной задаче – ***датчик***. Известно несколько разновидностей датчиков. Все, что влияет на урожай, должно быть измерено. Таким образом, нужны датчики температуры воды, температуры воздуха, влажности, кислотности, освещения и концентрации питательных веществ.

С внешней точки зрения ***датчик температуры*** – это объект, который способен измерять температуру там, где он расположен. Температура – это числовой параметр, имеющий ограниченный диапазон значений и определенную точность и означающий число градусов по Цельсию.

Местоположение датчика – это некоторое однозначно определенное место в теплице, температуру в котором необходимо знать. Таких мест, вероятно, немного. Для датчика температуры при этом существенно не само местоположение, а только то, что данный датчик расположен именно в данном месте.

Рассмотрим обязанности датчика температуры. Датчик должен знать значение температуры в своем местонахождении и сообщать ее по запросу. Клиент по отношению к датчику может выполнить такие действия: калибровать датчик и получать от него значение текущей температуры. Таким образом, объект "Датчик температуры" имеет две операции: "Калибровать" и "Текущая температура".

Функции, объявленные внутри описания, называются ***функциями-членами***. Их можно вызывать только для переменной соответствующего типа.

Центральной идеей абстракции является понятие инварианта. ***Инвариант*** – это некоторое логическое условие, значение которого (истина или ложь) должно сохраняться. Для каждой операции объекта можно задать предусловия (т.е. инварианты, предполагаемые операцией) и постусловия (т.е. инварианты, которым удовлетворяет операция).

Рассмотрим инварианты, связанные с операцией currentTemperature. Предусловие включает предположение, что датчик установлен в правильном

месте в теплице, а постусловие – что датчик возвращает значение температуры в градусах Цельсия.

Изменение инварианта нарушает контракт, связанный с абстракцией. Если нарушено предусловие, то клиент не соблюдает свои обязательства и сервер не может выполнить задачу правильно. Если нарушено постусловие, то свои обязательства нарушил сервер, и клиент не может ему больше доверять.

В случае нарушения какого-либо условия ***возбуждается исключительная ситуация***. Объекты могут возбуждать исключения, чтобы запретить дальнейшее выполнение операции и предупредить о проблеме другие объекты, которые в свою очередь могут принять на себя перехват исключения и справиться с проблемой.

2.2. Инкапсуляция

Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

На самом деле клиента не интересует, и не должно интересовать то, как реализовано выполнение контрактных обязательств. По крайней мере, пока сервер соблюдает свои обязательства.

Пример. Для программиста, *использующего стек*, важно только то, что он может помещать и извлекать нужные ему объекты с помощью вызова данных операций. Как реализован стек он может не знать, и детали реализации для него не всегда важны. Стек может быть реализован с использованием массива, имеющего фиксированное количество элементов, или посредством списковой структуры. Однако все эти детали скрыты от пользователя.

Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Инкапсуляция выполняется посредством скрытия информации, т.е. маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта, и реализация его операций. Для скрытия информации многие объектно-ориентированные языки программирования имеют соответствующие механизмы.

В результате всего сказанного мы можем ввести понятия интерфейса и реализации. ***Интерфейс*** – это набор операций, используемый для специфицирования услуг, предоставляемых классом. Интерфейс отражает внешнее поведение объекта. Внутренняя ***реализация*** описывает представление этой абстракции и механизмы достижения желаемого поведения объекта.

Интерфейс стека – это его операции pop и push, а реализация – это конкретное представление стека.

Друзьями класса называются классы или операции, имеющие доступ к закрытым операциям или данным некоторого класса.

Таким образом, введение ограничения доступа к элементам класса на практике реализует понятие инкапсуляции.

Инкапсуляция локализует те особенности проекта, которые могут подвергнуться изменениям. По мере развития системы разработчики могут решить, что какие-то операции выполняются несколько дольше, чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют внутреннее представление объекта. В результате становится возможным реализовать более эффективные алгоритмы, либо оптимизировать алгоритм по критерию памяти, заменяя хранение данных их вычислением. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов.

Сокрытие информации – понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. Кроме того, на практике иногда необходимо ознакомиться с реализацией класса, чтобы понять его назначение.

2.3. Модульность

Модулем называют набор связанных процедур вместе с данными, которые они обрабатывают.

В большинстве языков, поддерживающих принцип модульности, интерфейс модуля отделен от его реализации. Таким образом, принципы модульности и инкапсуляции являются взаимосвязанными.

Пример. В качестве примера рассмотрим модульную структуру программы, использующей стек.

Реализация стека и код пользователя будут находиться в раздельно компилируемых частях программы.

Как правило, объявления, описывающие интерфейс модуля, помещаются в так называемый ***заголовочный файл***, имеющий характерное имя, которое отражает его использование. Заголовочный файл обычно включается и в файл с пользовательским кодом, и в файл с реализацией модуля. Это достаточно простой и эффективный способ обеспечить идентичность представления интерфейса в обоих файлах. Включение информации об интерфейсе в файл с пользовательским кодом обусловлено необходимостью проверки типов используемых в нем интерфейсных функций во время компиляции.

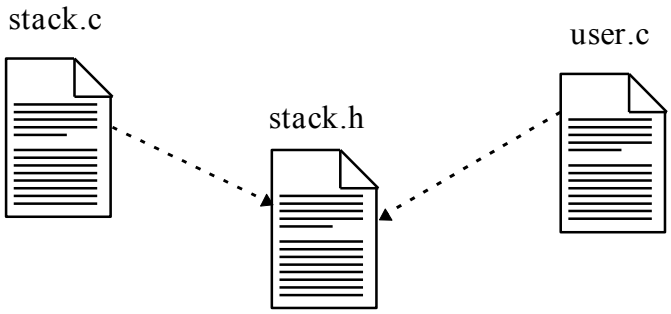

Рис. 2.1 Структура модулей программы, использующей стек

Компилятор должен знать, сколько отвести под него памяти. Если бы эта информация содержалась только в реализации класса, нам пришлось бы написать ее полностью, прежде чем мы смогли бы задействовать клиентов класса. То есть весь смысл отделения интерфейса от реализации был бы потерян.

При традиционном структурном подходе модульность – это искусство раскладывать подпрограммы по кучкам так, чтобы в одну кучку попадали подпрограммы, использующие друг друга или изменяемые вместе.

В объектно-ориентированном программировании по модулям необходимо распределить классы и объекты.

Правильное разделение программы на модули является сложной проблемой. Для небольших задач допустимо наличие одного модуля. Однако для большинства программ лучшим решением будет сгруппировать логически связанные элементы в отдельный модуль. При этом следует оставить открытыми только те элементы, которые совершенно необходимо видеть другим модулям. Заметим, что деление программы на модули бессистемным образом иногда гораздо хуже, чем отсутствие модульности вообще.

Рассмотрим приемы и правила, которые позволяют составлять модули наиболее эффективным образом:

– конечной целью разбиения программы на модули является снижение затрат на программирование за счет независимой разработки и тестирования;

– структура модуля должна быть достаточно простой для восприятия;

– реализация каждого модуля не должна зависеть от реализации других модулей;

– должны быть приняты меры для облегчения процесса внесения изменений там, где они наиболее вероятны.

Программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях. Для этого используют следующие правила:

– особенности системы, подверженные изменениям, следует скрывать в отдельных модулях;

– в качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала;

– все структуры данных должны быть обособлены в модуле; доступ к ним будет возможен для всех процедур этого модуля и закрыт для всех других;

– доступ к данным из модуля должен осуществляться только через процедуры данного модуля.

Следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями.

На выбор разбиения на модули могут влиять и некоторые внешние обстоятельства. При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками. Абстракции можно распределить так, чтобы быстро установить интерфейсы модулей по соглашению между группами, участвующими в работе. Внесение изменений в интерфейс одной подсистемы приводит к необходимости модификации других подсистем и изменений в их документации, все эти факторы требуют от интерфейса консерватизма.

Могут сказываться и требования секретности: одна часть кода может быть несекретной, а другая – секретной, тогда последняя выполняется в виде отдельного модуля (модулей).

В результате всего сказанного сформулируем следующее определение модульности:

Модульность – это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

Большие системы могут быть разложены на несколько сотен, если не тысяч, модулей. Пытаться разобраться в физической архитектуре такой системы без ее дополнительного структурирования почти безнадежно. По этой причине удобно ввести понятие подсистемы. Подсистемы представляют собой совокупности логически связанных модулей.

Подсистема – это агрегат, содержащий другие модули и другие подсистемы. Каждый модуль в системе должен располагаться в одной подсистеме или находиться на самом верхнем уровне.

Некоторые модули подсистемы могут быть общедоступны, т.е. экспортированы из системы и видимы снаружи. Другие модули могут быть частью реализации подсистемы и не использоваться внешними модулями.

2.4. Иерархичность

Абстракция является полезным инструментом. Однако всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. Инкапсуляция позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстрак-

ций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры.

Иерархия – это упорядочение абстракций, расположение их по уровням.

Основными видами иерархических структур применительно к сложным системам являются иерархии типа "является" и иерархии типа "имеет".

Иерархия ***"является"*** подразумевает, что элемент, стоящий на нижнем уровне абстракции, является разновидностью элемента, стоящего на верхнем уровне.

Например, лазерный принтер *является* разновидностью принтеров (лазерный принтер *является* принтером), принтер Хьюлетт-Паккард 6L *является* разновидностью лазерных принтеров (принтер Хьюлетт-Паккард 6L *является* лазерным принтером). Понятие "принтер" обобщает свойства, присущие всем принтерам, а лазерный принтер – это просто особый тип принтера со свойствами, которые отличают его, например, от матричного или струйного принтера.

Важный элемент объектно-ориентированных систем и основной вид иерархии "является" – иерархия обобщения (наследования) (отношение родитель-потомок).

Обобщение означает такое отношение между абстракциями, когда абстракция-потомок заимствует структурную или функциональную часть одной или нескольких абстракций-родителей. Если абстракция-потомок заимствует часть одной абстракции-родителя, то говорят об ***одиночном наследовании.*** Если же потомок заимствует часть нескольких родителей, то говорят о ***множественном наследовании.*** Часто потомок достраивает или переписывает компоненты родителя.

"Лакмусовой бумажкой" обобщения является обратная проверка. Если В не есть А, то В не стоит производить от А.

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общей абстракции. Потомок представляет собой специализированный частный случай своего предка. По этой причине говорят о наследовании как об иерархии *обобщение-специализация.* Таким образом, абстракция, стоящая на верхнем уровне, является обобщением для нижестоящей, а нижестоящая – специализацией вышестоящей.

Принцип наследования позволяет упростить выражение абстракций, делает проект менее громоздким и более выразительным. В отсутствие наследования каждая часть сложной системы становится самостоятельным блоком и должна разрабатываться "с нуля". Абстракции лишаются общности, поскольку каждый программист реализует их по-своему. Стройность системы достигается тогда только за счет дисциплинированности программистов.

С другой стороны, принципы абстрагирования, инкапсуляции и иерархичности находятся между собой в некоем здоровом конфликте. Абстрагирование данных создает непрозрачный барьер, скрывающий состояние и функции

объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных объектов.

Пример. Одиночное наследование. Вернемся к иерархии "принтер – лазерный принтер" (лазерный принтер является разновидностью принтеров).

Абстракция "лазерный принтер" строится на основе родительской абстракции "принтер". "Лазерный принтер" наследует от "принтера" свойства, определяющие все принтера. Кроме того, лазерный принтер имеет структурные и функциональные части, реализующие свойства, которые характерны именно для лазерных принтеров.

Пример. Множественное наследование. Введем абстракции "временный работник" и "секретарь". Секретарь может быть постоянным работником или временным. В последнем случае абстракция "временно работающий секретарь" наследует компоненты обеих абстракций. Временно работающий секретарь выполняет обязанности секретаря и имеет правовой статус временного работника.

Множественным наследованием часто злоупотребляют. Например, сладкая вата – это частный случай сладости, но никак не ваты. Следует применять ту же "лакмусовую бумажку": если В не есть А, то ему не стоит наследовать от А.

Иерархия ***"имеет"*** вводит отношение агрегации (*целое/часть*)**.** В иерархии "имеет" некоторая абстракция находится на более высоком уровне, чем любая из использовавшихся при ее реализации.

Агрегация есть во всех языках, использующих структуры или записи, состоящие из разнотипных данных. Но в объектно-ориентированном программировании она обретает новую мощь: агрегация позволяет физически сгруппировать логически связанные структуры, а наследование с легкостью копирует эти общие группы в различные абстракции.

Пример. Компьютер *имеет* системный блок (системный блок является частью компьютера). Системный блок компьютера одновременно имеет (агрегирует) материнскую плату, платы оперативной памяти, центральный процессор и множество других компонент. Заметим, что от замены процессора на более мощный, от добавления нескольких плат оперативной памяти или второго жесткого диска системный блок не становится другим системным блоком. Если же мы разбираем системный блок, мы уничтожает его как объект, однако его компоненты остаются и могут быть использованы в других системных блоках. Другими словами, системный блок и его компоненты имеют свои отдельные и независимые сроки жизни.

2.5. Типизация

Типизация – это способ защититься от использования объектов одного класса (типа) вместо другого, или, по крайней мере, управлять таким использованием.

Идея согласования типов занимает в понятии типизации центральное место. Возьмем, к примеру, физические единицы измерения. Разделив расстояние на время, мы ожидаем получить скорость, а не вес. В умножении температуры на силу смысла нет, а в умножении расстояния на силу есть. Все это примеры сильной типизации, когда прикладная область диктует правила и ограничения на использование и сочетание абстракций.

Важным понятием объектно-ориентированного подхода в целом является полиморфизм.

Полиморфизм – это способ присваивать различные значения (смыслы) одному и тому же сообщению. Смысл зависит от типа обрабатываемых данных.

Имеется несколько типов полиморфизма.

Принудительное приведение. Функция или оператор работает с несколькими различными типами, преобразуя их значения к требуемому типу.

Перегрузка. Функция или оператор вызывается на основе сигнатуры.

Если в описание класса ввести определение функции-члена с именем типа "operator *оператор*", то это означает, что данный *оператор* может быть применен к объектам или объекту данного класса, так же как и к переменным стандартных типов. При этом тело данной функции определяет смысл оператора.

Другие типы полиморфизма **– *включение и параметрический полиморфизм*** – мы рассмотрим в п. 4.3 и 4.5 соответственно.

Для осуществления явных преобразований переменных одного типа к другому типу имеются специальные ***операторы приведения***.

2.6. Параллелизм

Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере.

Процесс (поток управления) – это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере один поток управления, в параллельной системе таких потоков много.

Век одних потоков недолог, а другие живут в течение всего сеанса работы системы.

Параллелизм главное внимание уделяет абстрагированию и синхронизации процессов.

Объект, полученный из абстракции реального мира, может представлять собой отдельный поток управления (т.е. абстракцию процесса). Такой объект называется ***активным.***

Для систем, построенных на основе объектно-ориентированного проектирования, мир может быть представлен как совокупность взаимодействующих объектов, часть из которых является активной и выступает в роли независимых вычислительных центров. На этой основе дадим следующее определение параллелизма.

Параллелизм – это свойство, отличающее активные объекты от неактивных.

2.7. Сохраняемость

Любой программный объект существует в памяти и живет во времени.

Существуют объекты, которые присутствуют лишь во время вычисления выражения. Но есть и такие (например, как базы данных), которые существуют независимо от программы. Временной спектр сохраняемости объектов охватывает следующее:

– промежуточные результаты вычисления выражений;

– локальные переменные в вызове процедур;

– глобальные переменные и динамически создаваемые данные;

– данные, сохраняющиеся между сеансами выполнения программы;

– данные, сохраняемые при переходе на новую версию программы;

– данные, которые *вообще* переживают программу.

По традиции, первыми тремя уровнями занимаются языки программирования, а последними – базы данных. Языки программирования, как правило, не поддерживают понятия сохраняемости. Можно записывать объекты в неструктурированные файлы, но этот подход пригоден только для небольших систем. Как правило, сохраняемость достигается применением специальных объектно-ориентированных баз данных.

До сих пор мы говорили о сохранении объектов во времени. В большинстве систем объектам при их создании отводится место в памяти, которое не изменяется и в котором объект находится всю свою жизнь. Однако иногда необходимо обеспечивать возможность перемещения объектов в пространстве так, чтобы их можно было переносить с машины на машину и изменять форму

представления объекта в памяти. Это касается систем, распределенных в пространстве.

В результате получим следующее определение.

Сохраняемость – это способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

3. ОБЪЕКТЫ

Объект можно определить как осязаемую реальность, проявляющую четко наблюдаемое поведение. Объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве. Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины "экземпляр класса" и "объект" взаимозаменяемы.

3.1. Состояние

Пример. Рассмотрим торговый автомат, продающий напитки. Поведение такого объекта состоит в том, что после опускания в него монеты и нажатия кнопки автомат выдает выбранный напиток. Предположим, что сначала нажата кнопка выбора напитка, а потом уже опущена монета. Большинство автоматов при этом просто ничего не сделают, так как пользователь нарушил их основные правила. То есть автомат играл роль (ожидание монеты), которую пользователь игнорировал, нажав сначала кнопку. Предположим далее, что пользователь автомата не обратил внимание на предупреждающий сигнал "Бросьте столько мелочи, сколько стоит напиток" и опустил в автомат лишнюю монету. В большинстве случаев автоматы не дружественны к пользователю и радостно заглатывают все деньги.

В каждой из таких ситуаций поведение объекта определяется его историей: важна последовательность совершаемых над объектом действий. Такая зависимость поведения от событий и от времени объясняется тем, что у объекта есть внутреннее состояние. Для торгового автомата, например, состояние определяется суммой денег, опущенных до нажатия кнопки выбора. Другая важная информация – это набор воспринимаемых монет и запас напитков. На основе этого примера дадим следующее определение:

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств. В число свойств входят атрибуты объекта и атрибуты всех его агрегированных частей.

Одним из свойств торгового автомата является способность принимать монеты. Это статическое (фиксированное) свойство, в том смысле, что оно – существенная характеристика торгового автомата. С другой стороны, этому

свойству соответствует динамическое значение, характеризующее количество принятых монет. Сумма увеличивается по мере опускания монет в автомат и уменьшается, когда продавец забирает деньги из автомата.

В некоторых случаях значения свойств объекта могут быть статическими (например, заводской номер автомата), поэтому в данном определении использован термин "обычно динамическими".

К числу свойств объекта относятся присущие ему или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально.

Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Мы говорим "как правило", потому что в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом модель мира.

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. Состояние торгового автомата описывается в терминах других объектов, например имеющихся в наличии напитков. Конкретные напитки – это самостоятельные объекты, отличные от торгового автомата.

3.2. Поведение

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

Поведение – это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений. Поведение объекта – это его наблюдаемая и проверяемая извне деятельность.

Операция – это услуга, которую можно запросить у любого объекта класса для воздействия на его поведение.

Например, клиент может активизировать операции push и pop для того, чтобы управлять объектом-стеком (добавить или изъять элемент).

В чисто объектно-ориентированном языке принято говорить о передаче ***сообщений*** между объектами. В C++ мы говорим, что один объект вызывает функцию-член другого. В основном понятие сообщение совпадает с понятием операции над объектами.

Передача сообщений – это один уровень, задающий поведение. Из нашего определения следует, что состояние объекта также влияет на его поведение.

Рассмотрим торговый автомат. Мы можем сделать выбор, но поведение автомата будет зависеть от его состояния. Если мы не опустили в него достаточную сумму, скорее всего ничего не произойдет. Если же денег достаточно, автомат выдаст нам желаемое (и тем самым изменит свое состояние).

Некоторые операции изменяют состояние. В связи с вышесказанным можно заключить, что состояние объекта представляет суммарный результат его поведения.

Операция – это услуга, которую класс может предоставить своим клиентам. На практике типичный клиент совершает над объектами операции следующих видов:

– ***модификатор*** – это операция, которая изменяет состояние объекта;

– ***селектор*** – это операция, считывающая состояние объекта, но не меняющая состояния;

– ***конструктор*** – это операция создания объекта и/или его инициализации;

– ***деструктор*** – это операция, освобождающая ресурсы, которые использует объект, и/или разрушающая сам объект;

Две последние операции являются универсальными. Они обеспечивают инфраструктуру, необходимую для создания и уничтожения экземпляров класса. Если у класса есть конструктор, то он вызывается всегда, когда создается объект класса. Если у класса есть деструктор, то он вызывается всегда, когда объект класса уничтожается.

Объекты могут создаваться следующим образом:

– *автоматический объект* создается каждый раз, когда его описание встречается при выполнении программы, и уничтожается каждый раз при выходе из блока, в котором оно появилось;

– *статический объект* создается один раз, при запуске программы, и уничтожается один раз, при ее завершении;

– *объект в свободной памяти* создается с помощью операции new и уничтожается с помощью операции delete;

– *объект-член* создается как подобъект другого класса.

Для инициализации отдельных частей объекта с помощью конструктора служат ***инициализаторы конструктора***. Важность инициализаторов в том, что только с их помощью можно инициализировать константные члены и члены, являющиеся ссылками.

В чисто объектно-ориентированных языках определять процедуры и функции вне классов не допускается. В гибридных языках допускается описывать операции как независимые от объектов подпрограммы.

Операции, определенные вне классов, называют ***свободными подпрограммами.***

Свободные подпрограммы – это процедуры и функции, которые выполняют роль операций высокого уровня над объектом или объектами одного или разных классов. Свободные процедуры обычно группируются в соответствии с классами, для которых они создаются.

3.3. Идентичность

Идентичность – это такое свойство объекта, которое отличает его от всех других объектов.

Источником ошибок в объектно-ориентированном программировании является неумение отличать имя объекта от самого объекта.

Ситуацию, когда объект именуется более чем одним способом несколькими синонимичными именами, называют ***структурной зависимостью***.

Структурная зависимость порождает в объектно-ориентированном программировании много проблем. Трудность распознания побочных эффектов при действиях с синонимичными объектами часто приводит к утечкам памяти, неправильному доступу к памяти и, хуже того, непрогнозируемому изменению состояния.

3.4. Отношения между объектами

Сами по себе объекты не представляют никакого интереса: только в процессе их взаимодействия реализуется система. Например, самолет – это "совокупность элементов, каждый из которых по своей природе стремится упасть на землю, но за счет совместных непрерывных усилий преодолевающих эту тенденцию". Он летит только благодаря согласованным усилиям своих компонентов.

Отношения двух любых объектов основываются на предположениях, которыми один обладает относительно другого: об операциях, которые можно выполнять, и об ожидаемом поведении. Особый интерес для объектно-ориентированного анализа и проектирования представляют два типа отношений между объектами: связь и агрегация.

Связь – это семантическое соединение между объектами. Объект сотрудничает с другими объектами, посылая сообщения через связи, соединяющие его с ними. Связь – это специфическое сопоставление, через которое клиент запрашивает у объекта-сервера услугу или через которое один объект находит путь к другому.

Пусть есть два объекта A и B и связь между ними. Чтобы A мог послать B сообщение, B должен быть в каком-то смысле видим для A.

Перечислим следующие четыре способа обеспечить ***видимость***:

– сервер глобален по отношению к клиенту;

– сервер (или указатель на него) передан клиенту в качестве параметра операции;

– сервер является частью клиента;

– сервер локально порождается клиентом в ходе выполнения какой-либо операции.

Если связи обозначают равноправные или "клиент-серверные" отношения между объектами, то агрегация описывает отношения целого и части, приводящие к соответствующей иерархии объектов, причем, идя от целого (агрегата), мы можем прийти к его частям (атрибутам).

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно. Самолет состоит из крыльев, двигателей, шасси и прочих частей. С другой стороны, отношения акционера с его акциями – это агрегация, которая не предусматривает физического включения. Акционер монопольно владеет своими акциями, но они в него не входят физически.

4. КЛАССЫ

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте.

Класс – это множество объектов, имеющих общую структуру и общее поведение. Любой конкретный объект является экземпляром класса.

Пример. Рассмотрим сходства и различия между следующими классами: цветы, маргаритки, красные розы, желтые розы, лепестки и пчелы. Мы можем заметить следующее:

– маргаритка – цветок;

– роза – (другой) цветок;

– красная и желтая розы – розы;

– лепесток является частью обоих видов цветов;

– пчелы опыляют цветы и питаются их нектаром.

Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. В каждой проблемной области абстракции, описывающие ее, взаимодействуют различными способами.

Известны три основных типа отношений между классами. Во-первых, это отношение "обобщение/специализация" (общее и частное), т.е. иерархия "является". Розы являются цветами, что значит: розы являются специализированным частным случаем, подклассом более общего класса "цветы". Во-вторых, это отношение "целое/часть", т.е. иерархия "имеет". Например, лепестки являются частью цветов. В-третьих, это семантические, смысловые отношения, ассоциации. Например, пчелы ассоциируются с цветами.

Языки программирования выработали несколько общих подходов к выражению таких отношений. В частности, большинство объектно-ориентированных языков непосредственно поддерживает следующие виды отношений:

– ассоциация;
– агрегация;
– обобщение;
– зависимость;
– инстанцирование.

4.1. Ассоциация

Ассоциация – смысловая связь. По умолчанию, она не имеет направления и не объясняет, как классы общаются друг с другом. Мы можем только отметить семантическую зависимость, указав, какие роли играют классы друг для друга.

Так, ассоциация "Product – Sale" – двустороннее отношение: задавшись товаром, можно выйти на сделку, в которой он был продан, а пойдя от сделки, найти, **что** было продано.

Кратность (мощность) ассоциации – это количество ее участников. Различают три случая кратности ассоциации:

– один-к-одному;
– один-ко-многим;
– многие-ко-многим.

Рассмотренная в примере ассоциация имеет тип один-ко-многим: каждый экземпляр товара относится только к одной последней продаже, в то время как каждый экземпляр Sale может указывать на совокупность проданных товаров.

Отношение один-к-одному обозначает очень узкую ассоциацию. Например, в розничной системе продаж примером могла бы быть связь между классом "Продажа" и классом "Снятие денег с кредитной карточки": каждая продажа соответствует ровно одному снятию денег с данной кредитной карточки.

Отношение многие-ко-многим тоже нередки. Например, каждый объект класса "Покупатель" может инициировать сделку с несколькими объектами класса "Торговый агент", и каждый "Торговый агент" может взаимодействовать с несколькими объектами класса "Покупатель".

Класс может иметь ассоциацию с самим собой. Такая ассоциация называется ***рефлексивной***.

4.2. Агрегация

Агрегация является частным случаем ассоциации. Отношение агрегации между классами имеет непосредственное отношение к агрегации между их экземплярами.

Пример. Вернемся к классу Controller, который является абстракцией объектов, управляющих температурой в теплице (см. п. 3.4).

Класс Controller – это целое, а экземпляр класса Heater (нагреватель) – одна из его частей. В рассмотренном случае мы имеем специальный случай агрегации – композицию.

Композиция – форма агрегирования, в которой целое владеет своими частями, имеющими одинаковое с ним время жизни. Части с нефиксированной кратностью могут быть созданы после создания агрегата, но, будучи созданными, живут и умирают вместе с ним. Такие части могут также быть явно удалены перед уничтожением агрегата. В случае композитного агрегирования объект в любой момент времени может быть частью только одного композита.

4.3. Обобщение

4.3.1. Наследственная иерархия

Обобщение (наследование) – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других классов (множественное наследование).

Для описания полиморфизма используются два различных понятия: операция и метод. У класса есть ***операции***, которые определяют его поведение, и ***методы*** – реализации данных операций. При этом каждый потомок класса может предоставить метод, реализующий любую унаследованную операцию, отличный от соответствующего метода предка. Чисто виртуальная функция (абстрактная операция) не имеет соответствующего метода.

Самый общий класс в иерархии классов называется ***базовым***. В большинстве приложений базовых классов бывает несколько, и они отражают наиболее общие абстракции предметной области.

У класса обычно бывает два вида клиентов: экземпляры классов и подклассы. Часто полезно иметь для них разные интерфейсы. В частности, мы хотим показать только внешне видимое поведение для клиентов-экземпляров, но нам нужно открыть служебные функции и представления клиентам-подклассам.

Для этой цели описание класса разделяется на три части:

– открытую (public), видимую всем клиентам;

– защищенную (protected), видимую самому классу, его подклассам и друзьям (friend);

– закрытую (private), видимую только самому классу и его друзьям.

4.3.2. Обобщение и типизация

Вопросы построения наследственных иерархий тесно связаны с типизацией (в языках с сильной типизацией), поскольку при использовании наследования формируется и система типов.

При определении класса его суперкласс можно объявить public (как в нашем примере). В этом случае открытые и защищенные члены суперкласса становятся открытыми и защищенными членами подкласса. Таким образом, подкласс считается также и подтипом, то есть обязуется выполнять все обязательства суперкласса. В частности, он обеспечивает совместимое с суперклассом подмножество интерфейса и обладает неразличимым с точки зрения клиентов суперкласса поведением.

Если при определении класса объявить его суперкласс как private, это будет означать, что, наследуя структуру и поведение суперкласса, подкласс уже не будет его подтипом. Открытые и защищенные члены суперкласса станут закрытыми членами подкласса, и, следовательно, они будут недоступны подклассам более низкого уровня. В этом случае подкласс и суперкласс обладают несовместимыми интерфейсами с точки зрения клиента.

Если объявить суперкласс protected, то открытые и защищенные элементы такого суперкласса станут защищенными элементами подкласса. Однако, с точки зрения клиента интерфейсы класса и суперкласса несовместимы.

С наследованием связан особый тип полиморфизма – ***включение (чистый полиморфизм)***. Данный тип полиморфизма реализуется при вызове виртуальных функций для указателей (ссылок) на объекты. При открытом наследовании указатель родительского класса может указывать на объекты всех подклассов. Если виртуальная функция имеет различные реализации в подклассах, то выбор, какую ее реализацию вызывать, определяется с учетом выяснения подтипа на этапе выполнения. То есть виртуальная функция вызывается в зависимости не от *типа указателя*, а от реального *типа объекта*, на который он указывает. Данная ситуация называется механизмом позднего связывания.

4.3.3. Множественное наследование

Рассмотрим две проблемы, которые возникают при множественном наследовании: конфликт имен между суперклассами и повторное наследование.

Конфликт имен происходит тогда, когда в двух или более суперклассах случайно оказывается элемент (переменная или операция) с одинаковым именем.

Повторное наследование возникает тогда, когда при задании более чем одного базового класса какой-либо класс дважды является базовым для другого класса.

Продолжим пример с работающим студентом. Анализируя глубже полученную иерархию наследования, мы обнаружим, что и работник, и студент имеют ряд общих признаков, в частности, имя. Разумно ввести еще более общую абстракцию "Человек".

Одним из механизмов задания такого совместного использования является виртуальный базовый класс. ***Виртуальный базовый класс*** в производном классе представлен одним и тем же (совместно используемым) объектом.

4.4. Зависимость

Отношение ***зависимости (использования)*** между классами означает, что изменение в спецификации одного класса может повлиять на другой класс, который его использует, причем обратное в общем случае неверно. Можно сказать, что один из классов (клиент) пользуется услугами другого (сервера).

Один класс может использовать другой по-разному. В нашем примере это происходит при описании интерфейсной функции. Отношение использования также имеет место, если в реализации какой-либо операции происходит объявление локального объекта используемого класса.

4.5. Инстанцирование

Инстанцирование – подстановка фактических параметров шаблона вместо формальных. В результате создается конкретный класс, который может иметь экземпляры. Инстанцирование безопасно с точки зрения типов.

4.6. Переменные и операции класса

Обычно для каждого объекта необходима своя копия переменных, описанных в классе. Однако в некоторых ситуациях требуется, чтобы в классе были данные, общие для всех его экземпляров – переменные класса.

Утилитами называют совокупность глобальных переменных и свободных подпрограмм, сгруппированных в форме объявления класса. В этом случае глобальные переменные и свободные подпрограммы рассматриваются как члены класса, причем именно как статические. Введение утилит позволяет приблизить реализацию системы на языке C++ к набору классов и взаимодействующих объектов, как в чисто объектно-ориентированных языках.

4.7. Интерфейсы

Когда с помощью объектно-ориентированного подхода начали разрабатывать крупные программные системы, выяснилось, что кроме классов нужны дополнительные уровни абстракции. В частности, если сложный объект имеет

разное поведение, в зависимости от того, с кем он взаимодействует, то бывает удобно скрыть все функции, не нужные в данный момент. А точнее: на время данного взаимодействия сделать доступными все необходимые функции и только их. Для описания таких групп функций удобно использовать понятие ***интерфейса***. В данном контексте интерфейс удобно рассматривать как абстрактный класс, не имеющий собственных данных.

Пример. Все элементы управления телевизора можно разделить на несколько групп: пользовательские (громкость, номер канала), специальные (частота канала) и аппаратные (параметры электрических цепей). При этом пользователь работает с пользовательскими органами управления, настройщик – со специальными, а телемастер – с аппаратными. При этом, если телевизор исправен и настроен, пользователю нет необходимости видеть и менять состояние специальных и аппаратных органов управления. Поэтому пользовательские элементы управления обычно выносятся на переднюю панель телевизора, специальные закрыты небольшой дверцей, а аппаратные вообще погружены внутрь корпуса. Если бы все было на поверхности, пользователь мог бы сделать все то же, что и раньше, но для него оказались бы доступными специальные и аппаратные органы управления, и он мог бы случайно испортить настройки. Кроме того, передняя панель была бы загромождена настолько, что мало кто смог бы ориентироваться в обилии кнопок, ручек и т.п.

Между интерфейсами могут существовать отношения обобщения, ассоциации и зависимости, аналогичные одноименным отношениям между классами. Между классом и интерфейсом могут существовать отношения реализации и зависимости. Будем говорить, что класс ***реализует*** (или ***поддерживает***) интерфейс, если он содержит методы, реализующие все операции интерфейса. Интерфейс может реализовываться несколькими классами, а класс может реализовывать несколько интерфейсов. С другой стороны, класс может зависеть от нескольких интерфейсов, при этом предполагается, что какие-то классы эти интерфейсы реализуют.

4.8. Группирование классов

Когда система разрастается до десятка классов, можно заметить группы классов, связанные внутри и слабо зацепляющиеся с другими. Такие группы классов образуют пакет. ***Пакетом*** в области объектно-ориентированных технологий называют общий механизм организации элементов в группы. В данном контексте мы будем говорить только о группировании классов и называть пакетом группы, содержащие классы и другие пакеты.

Пакет не имеет операций или состояний в явном виде, они содержатся в нем неявно в описаниях агрегированных классов.

Некоторые классы в пакете могут быть открытыми, то есть экспортироваться для использования за пределы пакета. Остальные классы могут быть ча-

стью реализации, то есть не использоваться никакими классами, внешними к этому пакету.

5. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ

Одной из основных задач, решаемых на этапе анализа и ранних стадиях проектирования, является выявление ***ключевых абстракций*** задачи – классов и объектов, составляющих словарь предметной области.

На ранних стадиях внимание проектировщика сосредоточивается на внешних проявлениях ключевых абстракций. Такой подход создает логический каркас системы: структуры классов и объектов. На последующих фазах внимание переключается на внутреннее поведение ключевых абстракций, а также их физическое представление.

Выделим следующие способы проведения ***объектно-ориентированного анализа*:**

– классический подход;
– анализ поведения;
– анализ предметной области;
– анализ вариантов;
– CRC-карточки;
– неформальное описание.

Классический подход к классификации предполагает, что все вещи, обладающие некоторым свойством или совокупностью свойств, формируют некоторую категорию. Причем наличие этих свойств является необходимым и достаточным условием, определяющим категорию.

Например, студент – это категория: каждый человек или является студентом, или не является, и этого признака достаточно для решения вопроса, к какой категории принадлежит тот или иной индивидуум. С другой стороны, высокие люди не определяют категории, если, конечно, мы специально не уточним критерий, позволяющий четко отличать высоких людей от невысоких.

Таким образом, классический подход в качестве критерия похожести объектов использует родственность их свойств. В частности, объекты можно разбивать на непересекающиеся множества в зависимости от наличия или отсутствия некоторого признака.

Какие конкретно свойства надо принимать во внимание? Это зависит от цели классификации. Например, цвет автомобиля надо зафиксировать в задаче учета продукции автозавода, но он не интересен программе, которая управляет уличным светофором. Поэтому нет абсолютного критерия классификации, одна и та же структура классов может подходить для одной задачи и не годиться для другой.

При данном подходе кандидатами в классы и объекты могут быть выбраны:

– осязаемые предметы (автомобили, датчики);

– роли (учитель, политик);

– события (посадка на Марс, запрос);

– взаимодействие (заем, встреча).

Анализ поведения сосредоточивается на динамическом поведении как на первоисточнике объектов и классов. Классы формируются на основе групп объектов, демонстрирующих сходное поведение.

Напомним, что ответственностью объекта называют совокупность всех услуг, которые он может предоставлять по всем его контрактам. В результате анализа объединяют объекты, имеющие сходные ответственности и строят иерархию классов, в которую каждый подкласс, выполняя обязательства суперкласса, привносит свои дополнительные услуги.

До сих пор мы неявно имели в виду единственное разрабатываемое нами приложение. Но иногда в поисках полезных и уже доказавших свою работоспособность идей полезно обратиться сразу ко всем приложениям в рамках данной предметной области. Анализ данной предметной области может указать на ключевые абстракции, оказавшиеся полезными в сходных системах. ***Анализ предметной области*** – это попытка выделить те объекты, операции и связи, которые эксперты данной области считают наиболее важными.

Анализ включает следующие этапы:

– построение скелетной модели предметной области при консультациях с экспертами в этой области;

– изучение существующих в данной области систем и представление результатов в стандартном виде;

– определение сходства и различий между системами при участии экспертов;

– уточнение общей модели для приспособления к нуждам конкретной системы.

Анализ области можно вести относительно аналогичных приложений (вертикально) или относительно аналогичных частей одного и того же приложения (горизонтально).

В роли эксперта часто выступает пользователь системы, например, инженер или диспетчер. Он не обязательно должен быть программистом, но ему должны быть хорошо знакомы исследуемая проблема и ее язык.

Анализ вариантов – это подход, который можно успешно сочетать с первыми тремя, делая их применение более упорядоченным.

Вариант применения – это частный пример, сценарий или образец использования, начинающийся с того, что пользователь системы инициирует операцию или последовательность взаимосвязанных событий.

Пользователи, эксперты и разработчики перечисляют сценарии, наиболее существенные для работы системы (пока не углубляясь в детали). Затем они

тщательно прорабатывают сценарии, раскладывая их по кадрам. При этом устанавливается, какие объекты участвуют в сценарии, каковы обязанности каждого объекта и как они взаимодействуют в терминах операций. Далее набор сценариев расширяется, чтобы учесть исключительные ситуации и вторичное поведение.

CRC-карточки – Class-Responsibilities-Collaborators (Класс-Ответственности-Сотрудники) – это простой и эффективный способ анализа сценариев.

Сотрудник – это другой класс, который взаимодействует с данным для обеспечения некоего общего набора поведений.

На обычных карточках небольшого размера сверху пишут название класса, снизу в левой половине – за что он отвечает, снизу в правой половине – с кем он сотрудничает.

На рис. 5.1 приведен вид CRC-карточки для класса Stack.

Имя класса: Stack	
Ответсвенность: push pop	Сотрудники: нет

Рис. 5.1. CRC-карточка для класса Stack

Разработчики по ходу анализа сценария заводят по карточке на каждый обнаруженный класс и дописывают в нее новые пункты.

Карточки можно раскладывать так, чтобы представить формы сотрудничества объектов. С точки зрения динамики сценария их расположение может показать поток сообщений между объектами, с точки зрения статики они представляют иерархии классов.

Согласно методу ***неформального описания*** надо описать задачу или ее часть на обычном разговорном языке, а потом подчеркнуть существительные и глаголы. Существительные – кандидаты на роль классов, а глаголы могут стать именами операций.

Подход прост, однако он весьма приблизителен и непригоден для сколько-нибудь сложных проблем. Кроме того, человеческий язык не является точным средством выражения.

Некоторым существительным больше соответствуют не классы, а, например, признаки или свойства объектов (имя, возраст, вес, адрес и т.п.) или даже имена операций ("телефонный вызов" вряд ли означает какой-либо класс).

Рассмотрим вопросы ***поиска, выбора и уточнения ключевых абстракций.***

Самая главная ценность ключевых абстракций заключается в том, что они определяют границы нашей проблемы: выделяют то, что входит в нашу систему и поэтому важно для нас, и устраняют лишнее. Задача выделения таких абстракций специфична для проблемной области. Правильный выбор объектов зависит от назначения приложения и степени детальности обрабатываемой информации.

Определение ключевых абстракций включает в себя два процесса: открытие и изобретение. Мы *открываем* абстракции, слушая специалистов по предметной области: если эксперт про нее говорит, то эта абстракция обычно действительно важна. *Изобретая*, мы создаем новые классы и объекты, не обязательно являющиеся частью предметной области, но полезные при проектировании или реализации системы. Например, пользователь банкомата говорит "счет, снять, положить"; эти термины – часть словаря предметной области. Разработчик системы использует их, но добавляет свои, такие, как "база данных", "список", "очередь" и т.д. Эти ключевые абстракции созданы уже не предметной областью, а проектированием. Наиболее мощный способ выделения ключевых абстракций – сведение задачи к уже известным классам и объектам.

Определив новые абстракции, мы должны найти их место в контексте уже существующих классов и объектов. Не стоит пытаться делать это строго сверху вниз или снизу вверх, поскольку трудно сразу расположить классы и объекты на правильных уровнях абстракции. Иногда, найдя важный класс, мы можем передвинуть его вверх в иерархии классов, тем самым увеличивая степень повторности использования кода. Аналогично, можно прийти к выводу, что класс слишком обобщен, и это затрудняет наследование.

Кроме того, по ходу работы возможно следующее:

– выделить излишек ответственности в новый класс;

– перенести ответственность с одного большого класса на несколько более детальных классов;

– передать часть обязанностей другому классу.

6. ОСНОВНЫЕ КОНСТРУКЦИИ ЯЗЫКА UML

Поскольку использование объектно-ориентированного подхода особое значение имеет при разработке сложных программных продуктов, модели предметной области, которые приходится строить в этих случаях, тоже будут сложны. Удержать в голове эту модель целиком или по частям разработчик (или коллектив разработчиков) обычно не в состоянии. Поэтому особое значение при объектно-ориентированном подходе имеют средства, позволяющие визуализировать, сохранять и документировать принимаемые решения. Одним из таких средств является унифицированный язык моделирования UML. Разработка системы средствами UML происходит в виде построения набора диаграмм, позволяющих описать определенные части моделируемой реальности в определенных аспектах.

В данном разделе мы рассмотрим основные конструкции языка UML. Наше обсуждение ни в коей мере не будет претендовать на полноту изложения. Мы лишь рассмотрим некоторые типы диаграмм и отражаемых на них элементов. В основном мы ограничимся отдельными способами представления структурных отношений между классами, интерфейсами и объектами (статический аспект) и их поведения (динамический аспект) в логической структуре проекта, не затрагивая таких вопросов моделирования, как выработка требований к системе и физическая реализация.

6.1. Диаграмма классов

Основными элементами, отображаемыми на ***диаграмме классов***, являются классы, интерфейсы и отношения между ними.

Графическое изображение класса представлено на рис. 6.1.

Каждый класс должен иметь имя. На некоторых значках классов полезно перечислять несколько атрибутов и операций класса. "На некоторых", потому что для большинства тривиальных классов это не нужно. Если мы не хотим видеть на диаграмме атрибуты и операции класса, мы удаляем разделяющие черты и пишем только имя класса.

Рис. 6.1. Значок класса

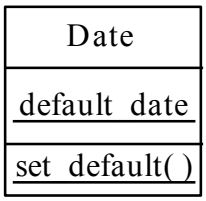

Рис. 6.2. Статические члены

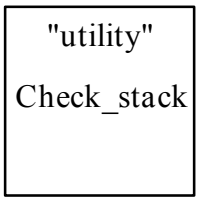

Рис. 6.3. Утилита класса

Атрибут может обозначаться именем, также могут быть указаны тип (класс), видимость, кратность и значение по умолчанию в соответствии со следующим синтаксисом:

видимость имя [кратность] : тип = значение_по_умолчанию.

Видимость обозначается путем добавления в качестве префикса к имени следующих символов: открытый(+), защищенный (#), закрытый(–).

Кратность (количество элементов указанного типа, составляющих атрибут) показывается в виде последовательности разделенных запятой спецификаций интервалов в формате: нижняя граница .. верхняя граница и конкретных значений. Для обозначения неограниченного количества используется символ *. Например:

1	в точности один;
*	ноль или больше;
0 .. *	ноль или больше;
1 .. *	один или больше;

3 .. 7 указанный интервал;
1 .. 3, 7 указанный интервал или точное число.

Приведем примеры описания атрибутов:

A только имя;
+A : C видимость, имя и тип;
A[10] : C имя, кратность и тип;
A : C = E имя, тип и значение по умолчанию.

Операции обычно изображаются внутри значка класса только своим именем. Возможно также указывать дополнительные элементы в формате: видимость имя (формальные_параметры) : тип_возвращаемого_значения.

Формальные параметры записываются в формате:

вид имя : тип = значение_по_умолчанию,

где вид принимает одно из следующих значений:

in – входной параметр (значение по умолчанию);
out – выходной параметр;
inout – смешанный, т.е. и входной, и выходной.

Если параметр помечен как входной, то функция не может изменять его значения, но может его использовать (передача параметров по значению); если параметр помечен как выходной, то функция не может использовать его значение, но может присвоить ему некоторое значение; наконец, если параметр помечен как смешанный, то функция может и читать его значение, и изменять (передача параметров по ссылке).

Приведем примеры описания операций:

display ( ) – только имя;
display ( ): bool – имя и тип возвращаемого значения;
+set (n: Name, s: String) – видимость, имя и формальные параметры.

Атрибуты и операции класса (статические члены) изображаются подчеркнутыми. Пример на рис. 6.2 показывает изображение класса Data (см. п. 4.6).

Утилита изображается обычным значком класса со стереотипом "utility" (рис. 6.3). Добавление к стандартному элементу языка ***стереотипа*** позволяет приспособить данный элемента для решения специальной проблемы.

Интерфейс на диаграмме классов можно изобразить двумя способами: развернутым и сокращенным. В случае развернутого способа интерфейс изображается на диаграмме как класс со стереотипом "interface" и без секции атрибутов (рис. 6.4). Сокращенное изображение интерфейса представляет небольшой кружок с именем интерфейса возле него (рис. 6.5).

Рис. 6.4. Развернутое изображение интерфейса

Рис. 6.5. Сокращенное изображение интерфейса

Классы редко бывают изолированными; они вступают в отношения друг с другом.

Значок ассоциации изображается в виде линии, соединяющей два класса (рис. 6.6). При изображении ассоциации ей можно сопоставить текстовую пометку, документирующую имя этой связи или подсказывающую ее роль. Ассоциации часто отмечаются существительными, например "Место работы", описывающими природу связи. К значку также можно добавить роли, которые классы играют в ассоциации. Одна пара классов может иметь более одной ассоциативной связи. Возле значка ассоциации можно указать ее кратность.

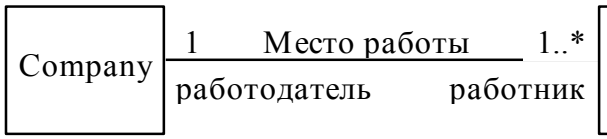

Рис. 6.6. Ассоциация

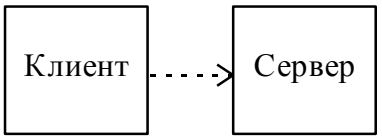

Рис. 6.7. Зависимость

Указывая кратность на одном конце, показывают, что на этом конце именно столько объектов должно соответствовать каждому объекту на другом конце. Если кратность явно не указана, то подразумевается, что она не определена.

На рис. 6.6 изображена ассоциация "Компания" – "Человек", которой дано имя "Место работы". Классы, участвующие в ней, играют роли "Работодатель" и "Работник" соответственно.

Отношение зависимости изображается пунктирной линией со стрелкой, направленной от клиента к серверу (рис. 6.7). Стереотип "friend" на линии зависимости указывает, что клиент является дружественным классом для сервера.

Значок обобщения (наследования) включает большую незакрашенную стрелку, которая указывает от подкласса к суперклассу (рис. 6.8). Место классов в наследственной иерархии можно уточнить следующим образом. Имена абстрактных классов указываются курсивом. К имени базового класса добавляется свойство root, а к имени листового класса – свойство leaf. Аналогично, имена чисто виртуальных функций-членов указываются курсивом, а к имени операции, которая не может быть виртуальной, добавляется свойство leaf.

Пример. На рис. 6.8 показана наследственная иерархия, связанная с геометрическими фигурами (см. п. 4.3.1). Абстрактный класс Shape отмечен как базовый класс, имеющий чисто виртуальные функции draw, rotate и листовую функцию get_center. Функция draw в протоколе класса Circle является полиморфной. Класс SolidCircle объявлен как листовой.

Значок агрегации – линия с добавлением незакрашенного ромба на конце, обозначающем агрегат (рис. 6.9). Экземпляры класса на другом конце стрелки будут частями экземпляров класса-агрегата. Отношение композиции обозначается линией с закрашенным ромбом на конце (рис. 6.10).

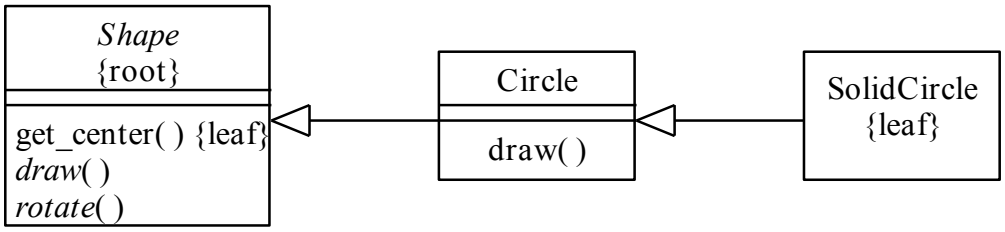

Рис. 6.8. Обобщение

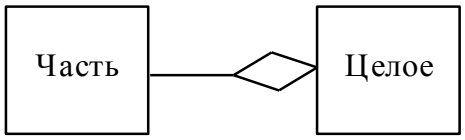

Рис. 6.9. Агрегация

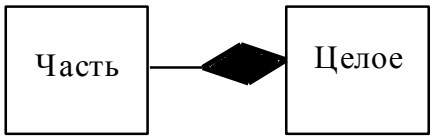

Рис. 6.10. Композиция

Параметризованный класс изображается значком обычного класса с пунктирным прямоугольником в правом верхнем углу, в котором указаны параметры (рис. 6.11). Моделировать инстанцирование можно двумя способами. Во-первых, можно явным образом определить зависимость со стереотипом "bind", показывающую, что источник инстанцирует целевой шаблон с заданными фактическими параметрами. Во-вторых моделировать можно неявно, для чего требуется объявить класс, имя которого обеспечивает инстанцирование. Пример использования данных способов представлен на рис. 6.11–6.12 для стека контроллеров, рассмотренного в п. 4.5.

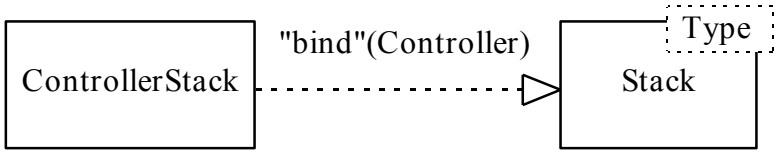

Рис. 6.11. Явное связывание

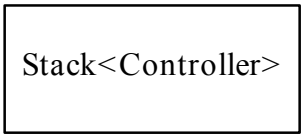

Рис. 6.12. Неявное связывание

Пример. На рис. 6.13 представлена система классов, служащая для моделирования процесса обслуживания системы управления климатом в теплице. Класс Plan включает правила управления климатом и имеет операцию execute

(выполнить). Имеется ассоциация между этим классом и классом Controller (контроллер, управляющий климатом): экземпляры класса Plan задают климат, который должны поддерживать экземпляры класса Controller.

Эта диаграмма также показывает, что класс Controller является агрегатом: его экземпляры содержат в точности по одному экземпляру классов Heater (нагреватель) и Cooler (охлаждающее устройство), и любое число экземпляров класса Light (лампочка). Оба класса Heater и Cooler являются подклассами абстрактного запускающего процесс класса Actuator, который предоставляет протоколы startUp и shutDown (начать и прекратить соответственно) и использует класс Temperature.

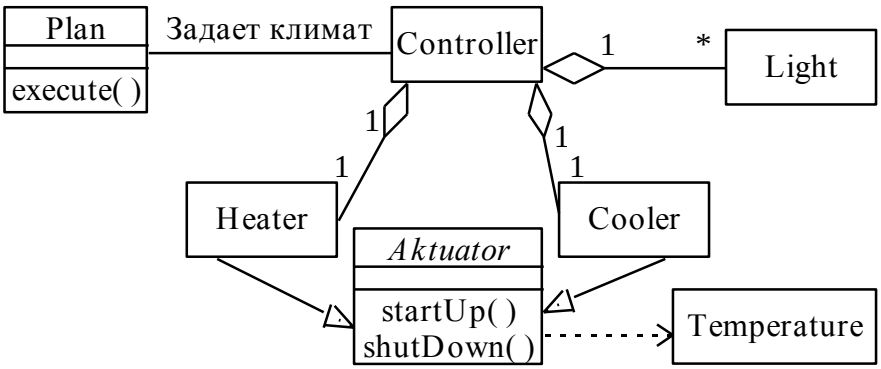

Рис. 6.13. Классы для описания процесса обслуживания системы управления климатом

Связь интерфейса с реализующим его элементом можно графически представить двумя способами. Простая форма позволяет изобразить отношения между интерфейсом и реализующим его классом в виде кружка с одной стороны класса (рис. 4.3–4.4). Если интерфейс представлен в расширенной форме, то отношение реализации изображают в виде пунктирной линии с большой незакрашенной стрелкой, направленной в сторону интерфейса (рис. 6.14). Зависимость между классом и интерфейсом показывается аналогично зависимости между классами.

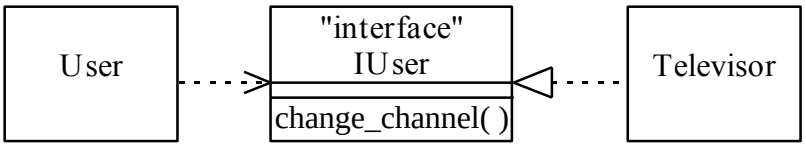

Рис. 6.14. Отношения между классами и интерфейсом

6.2. Диаграмма объектов

Диаграмма объектов показывает существующие объекты и их связи в некоторый момент времени.

Диаграмма объектов может рассматриваться как прототип: она представляет отношения, которые могут возникнуть у данного множества экземпляров

классов, безотносительно к тому, какие конкретно экземпляры участвуют в этом взаимодействии.

Объект на диаграмме объектов изображается значком, показанным на рис. 6.15. Он совпадает со значком класса, но имя объекта подчеркивается. Горизонтальная линия разделяет текст внутри значка объекта на две части: имя объекта и его атрибуты.

Имя объекта следует синтаксису для атрибутов и может быть записано в одной из следующих форм:

Рис. 6.15. Значок объекта

Рис. 6.16. Значок связи

A – только имя объекта;
: C – только класс объектов (анонимный экземпляр);
A : – экземпляр, класс которого пока неизвестен;
A : C – имя объекта и класса.

Каждый значок без имени объекта обозначает на диаграмме отдельный анонимный объект.

Объекты взаимодействуют с другими объектами через связи, которые изображаются на диаграмме прямыми линиями (см. рис 6.16).

На значках объектов бывает полезно указать несколько их атрибутов. Синтаксис атрибутов совпадает с синтаксисом атрибутов класса и позволяет указать их текущее значение (см. рис 6.17). Имена атрибутов объектов должны соответствовать атрибутам, определенным в классе объекта, или в любом из его суперклассов.

На рис. 6.17 приведен пример диаграммы объектов, соответствующий диаграмме классов на рис. 6.6.

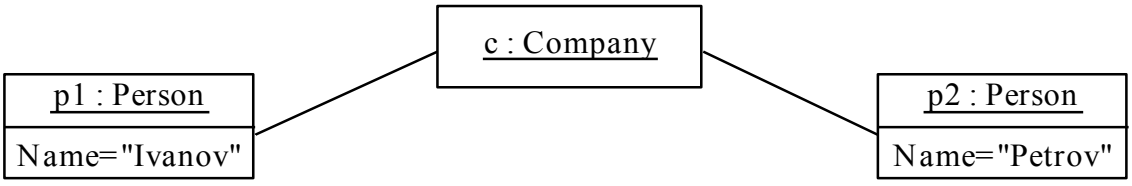

Рис. 6.17. Диаграмма объектов

6.3. Диаграммы взаимодействий

На ***диаграммах взаимодействий*** отражаются множество объектов и отношения между ними, включая сообщения, которыми они обмениваются. К

диаграммам взаимодействий относятся диаграммы последовательностей и диаграммы кооперации.

На ***диаграммах последовательностей*** внимание акцентируется прежде всего на временной упорядоченности сообщений. Преимущество диаграммы последовательностей в том, что на ней легче читается порядок посылки сообщений.

Диаграмма последовательностей внешне напоминает таблицу (рис. 6.18–6.19). Изображения объектов на диаграмме последовательностей помещают в верхней ее строке. Обычно инициирующий взаимодействие объект размещается слева, а остальные правее (тем дальше, чем более подчиненным является объект).

Отправления сообщений (вызовы операций) показываются горизонтальными стрелками. Линия, обозначающая посылку сообщения, проводится от вертикали клиента к вертикали сервера. Первое сообщение показывается на самом высоком уровне, второе – ниже и т.д. Возвращаемое значение можно показать пунктирной стрелкой от сервера к клиенту.

Вызов операции обозначается ее именем, кроме того, здесь могут быть приведены возвращаемое значение и фактические параметры, подходящие к сигнатуре операции:

N ( ) – только имя операции;

r := N (a, c) – возвращаемое значение (r), имя и фактические параметры операции – a и c.

Пример. Диаграмма последовательностей приведена на рис. 6.18. На диаграмме отражено сотрудничество трех объектов. Сценарий начинается с вызова объектом A операции f1 над объектом B. Это порождает вызов объектом B операции f2 над объектом C, что потребует вызова объектом C операции f3 над собой. Когда эта операция будет выполнена, объект B возвратит значение r объекту A, который затем вызывает операцию f4 над объектом C.

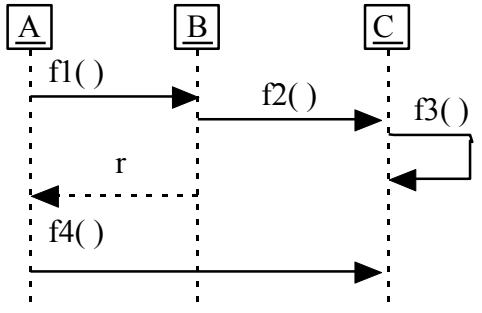

Рис. 6.18. Диаграмма последовательностей

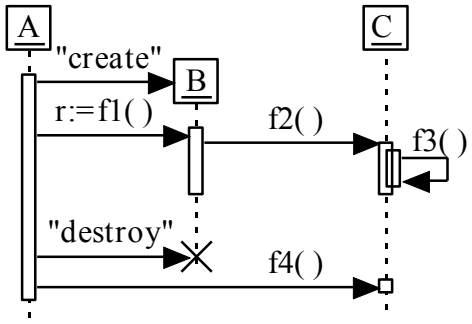

Рис. 6.19. Расширенная диаграмма последовательностей

Диаграммы последовательностей характеризуются двумя особенностями, отличающими их от диаграмм кооперации.

Во-первых, на них показана ***линия жизни*** объекта. Это вертикальная пунктирная линия, отражающая существование объекта во времени. Большая

часть объектов, представленных на диаграмме взаимодействий, существует на протяжении всего взаимодействия, поэтому их изображают в верхней части диаграммы, а их линии жизни прорисованы сверху до низу. Объекты могут создаваться и во время взаимодействий. Линии жизни таких объектов начинаются с получения сообщения со стереотипом "create". Объекты могут также уничтожаться во время взаимодействий; в таком случае их линии жизни заканчиваются получением сообщения со стереотипом "destroy", а в качестве визуального образа используется большая буква X.

Во-вторых, на этих диаграммам может быть изображен ***фокус управления***. Он изображается в виде вытянутого прямоугольника, показывающего промежуток времени, в течение которого объект выполняет какое-либо действие, непосредственно или с помощью подчиненной процедуры (рис. 6.19). Верхняя грань прямоугольника выравнивается по временной оси с моментом начала действия, нижняя – с моментом его завершения. Вложенность фокуса управления, вызванную рекурсией (т.е. обращением к собственной операции) или обратным вызовом со стороны другого объекта, можно показать, расположив другой фокус управления чуть правее своего родителя.

Рис. 6.19 показан пример, аналогичный предыдущему, в котором, однако, объект B создается и уничтожается во время взаимодействия.

Диаграмма кооперации акцентирует внимание на структурной организации объектов, принимающих и отправляющих сообщения.

На диаграмме изображаются объекты и связи между ними, как на диаграмме объектов. Рядом с соответствующей связью на диаграмме можно записать набор сообщений. Каждое сообщение состоит из следующих трех элементов:

– направление вызова;

– вызов операции;

– порядковый номер.

Направление сообщения показывается стрелкой, указывающей на объект-сервер.

Вызов операции обозначается с использованием того же самого синтаксиса, что и на диаграмме кооперации.

Порядковый номер показывает относительный порядок посылки сообщений. Сообщение с меньшим порядковым номером посылается до сообщения с большим номером. Нумерация начинается с единицы и добавляется как префикс к вызову операции. Для отображения вложенных сообщений используется следующая нотация: 1 – первое сообщение; 1.1 – первое сообщение, вложенное в сообщение 1; 1.2 – второе сообщение, вложенное в сообщение 1, и т.д.

Для описания связи одного объекта с другим к дальней концевой точке этой связи можно присоединить один из следующих стереотипов пути:

"association" – соответствующий объект видим для ассоциации;

"self" – соответствующий объект видим, потому что является диспетчером для операции (обозначение связи объекта с самим собой);

"global" – соответствующий объект видим, т.к. находится в объемлющей области видимости;

"local" – соответствующий объект видим, поскольку находится в локальной области действия операции;

"parameter" – соответствующий объект видим, т.к. является параметром операции.

Пример. Диаграмма кооперации, соответствующая диаграмме взаимодействий с рис. 6.19, показана на рис. 6.20. Дополнительная информация на диаграмме отражает тот факт, что объект C является глобальным по отношению к остальным объектам.

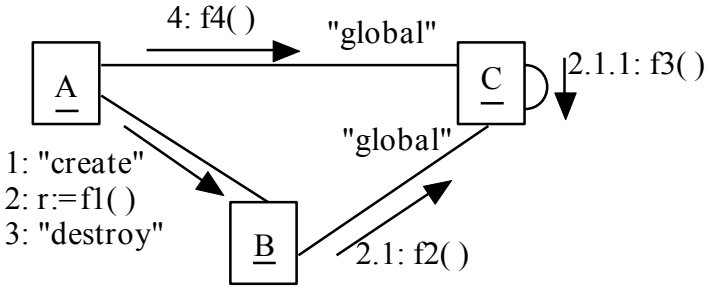

Рис. 6.20. Диаграмма кооперации

Чаще всего приходится моделировать неветвящиеся последовательные потоки управления. Однако можно моделировать и более сложные потоки, содержащие итерации и ветвления.

Итерация представляет собой повторяющуюся последовательность сообщений. Ее изображение приведено на рис. 6.21: перед номером сообщения в последовательности ставится выражение итерации, например *[i := 1 .. n] (или просто *, если надо обозначить итерацию без дальнейшей детализации). Итерация показывает, что сообщение (и все вложенные в него сообщения) будет повторяться в соответствии со значением заданного выражения.

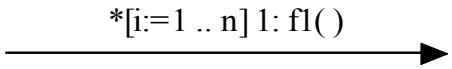

Рис. 6.21. Итерации

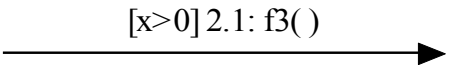

Рис. 6.22. Условие

Условие представляет собой сообщение, выполнение которого зависит от результатов вычисления некоторого булевского выражения. Для моделирования условия перед порядковым номером сообщения ставится выражение, например [x>0] (рис. 6.22). У всех альтернативных ветвей будет один и тот же порядковый номер, но условия на каждой ветви должны быть заданы так, чтобы два из них не выполнялись одновременно (не перекрывались).

6.4. Диаграмма состояний

Диаграмма состояний включает:

–состояния;

– переходы вместе с ассоциированными событиями и действиями.

Состоянием в контексте диаграммы состояний будем называть ситуацию в жизни объекта, на протяжении которой он удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает какого-то события.

Событием в контексте диаграммы состояний будем называть любое происшествие, которое может быть причиной изменения состояния системы.

Деятельность – это продолжающийся во времени неатомарный шаг вычислений при реализации поведения объекта. ***Действие*** – атомарное вычисление, которое приводит к изменению состояния системы или возврату значения. Действие может заключаться в вызове операции, создании или уничтожении объекта либо в простом вычислении – скажем, значения выражения. Поскольку действие атомарно, оно не может быть прервано другим событием, и, следовательно, выполняется до полного завершения. В отличие деятельности, которая может быть прервана другим событием.

Обозначение отдельного состояния приведено на рис. 6.23. Каждое состояние должно иметь имя, указываемое на значке.

Рис. 6.23. Значок состояния

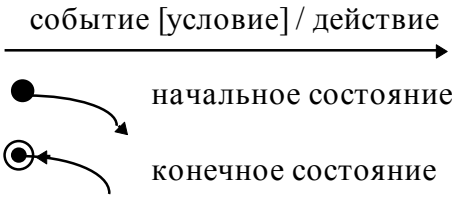

Рис. 6.24. Значки перехода

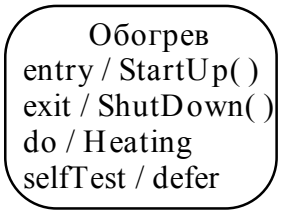

Рис. 6.25. Состояние "Обогрев"

Изменение состояний осуществляется в результате перехода. ***Переход*** – это отношение между двумя состояниями, показывающее, что объект, находящийся в первом состоянии, должен выполнить определенные действия и перейти во второе состояние, как только произойдет указанное событие и будут удовлетворены указанные условия.

Говорят, что при таком изменении состояния переход *срабатывает.* Пока переход не сработал, объект находится в исходном состоянии; после срабатывания он находится в целевом состоянии. Например, Обогреватель может перейти из состояния Ожидание в состояние Активация при возникновении события tooCold (слишкомХолодно) с параметром desiredTemp (желаемаяТемпература).

Переход определяют пять элементов:

исходное состояние – состояние, из которого происходит переход; если объект находится в исходном состоянии, то исходящий переход может сработать тогда, когда объект получит событие-триггер, инициирующее этот переход, причем должно быть выполнено сторожевое условие (если оно задано);

событие-триггер – событие, при получении которого объектом, находящимся в исходном состоянии, может сработать переход (при этом должно быть выполнено сторожевое условие);

сторожевое условие – булевское выражение, которое вычисляется при получении события-триггера; если значение истинно, то переходу разрешено сработать, если ложно – переход не срабатывает; если при этом не задано никакого другого перехода, инициируемого тем же самым событием, то событие теряется;

действие – атомарное вычисление, которое может непосредственно воздействовать на рассматриваемый объект или оказать косвенное воздействие на другие объекты, находящиеся в области видимости;

целевое состояние – состояние, которое становится активным после завершения перехода; состояние может иметь переход само в себя.

Переход на диаграмме состояний изображается линией со стрелкой, направленной от исходного состояния к целевому. Рядом с линией указываются событие, вызвавшее переход, условие, при выполнении которого происходит переход, и ассоциированные с переходом действия (рис. 6.24).

На каждой диаграмме состояний должно присутствовать ровно одно стартовое состояние: оно обозначается немаркированным переходом в него из специального значка, изображаемого в виде закрашенного кружка (см. рис. 6.24). Иногда бывает нужно указать также конечное состояние, которое обозначают, рисуя немаркированный переход от него к специальному значку, изображаемому как кружок с закрашенной серединой (см. рис. 6.24).

Допускаются "переходы без события" – ***нетриггерные переходы***. В этом случае переход совершается сразу после завершения действия, связанного с состоянием. Если переход условный, он состоится только в случае, если условие выполнено.

На значках состояний можно указывать некоторые действия, деятельности и внутренние переходы.

Можно назначить выполнение некоторого ***действия на входе или выходе из состояния***. Для этого используются ключевые слова entry и exit (вход и выход). Например, на рис. 6.25 приведен значок состояния "Обогрев" объекта, управляющего температурой. При каждом входе в данное состояние выполняется операции StartUp (запустить обогреватель), а при каждом при выходе из состояния – операции ShutDown (отключить обогреватель).

Когда объект находится в некотором состоянии, он обычно бездействует и ожидает возникновения какого-либо события. Но иногда приходится моделировать и ситуацию непрерывно продолжающейся ***деятельности***. Находясь в таком состоянии, объект чем-нибудь занимается до тех пор, пока эта деятель-

ность не будет прервана событием. Ключевое слово do служит для явных указаний: "Начать деятельность при входе в состояние (после того, как было отработано действие при входе) и окончить при выходе из него". Например, пока объект находится в состоянии "Обогрев", осуществляется деятельность Heating (рис. 6.25). Допустимо специфицировать и последовательность действий, которые отделяются друг от друга точкой с запятой, например

do / op1(a); op2(b); op3(c).

Единичное действие не может быть прервано, но к последовательности действий это не относится. Между выполнением соседних действий возможно обрабатывать события, и не исключено, что в результате этого произойдет выход из состояния.

В любой моделируемой ситуации бывает необходимо одни события распознавать, а другие игнорировать. Между тем иногда необходимо распознать событие, но отложить его обработку на будущее. Такое поведение специфицируется с помощью отложенных событий. ***Отложенное событие*** – это список событий, возникновение которых в конкретном состоянии отложено до перехода в состояние, где эти события не являются отложенными. В этот момент события могут быть обработаны и инициировать те или иные переходы, как будто они произошли только что. На рис. 6.25 показано, что событие описывается как отложенное путем связывания с ним специального действия defer (отложить). Находясь в состоянии "Обогрев", объект, управляющий температурой, может отложить обработку сигнала самопроверки, посылаемого некоторым внешним объектом, отвечающим за периодическое обслуживание.

Находясь в некотором состоянии, объект может получить события, которые желательно обработать, не покидая состояния. Такая обработка называется ***внутренним переходом***. Между внутренним переходом и переходом в себя имеется некоторое различие. При переходе в себя событие инициирует переход, происходит выход из данного состояния и выполняется действие при выходе (exit). Затем, поскольку при переходе в себя происходит выход из состояния и повторный вход в него же, выполняется действие, ассоциированное с переходом, и, кроме того, действие при входе в состояние (entry). Предположим, однако, что необходимо обработать событие, не возбуждая действия при входе и выходе. Для этого в значок, обозначающий состояние, можно поместить внутренний переход в виде

событие / действие.

Если объект находится в таком состоянии и происходит указанное событие, то соответствующее действие выполняется без выхода и повторного входа в состояние, т.е. событие обрабатывается, не возбуждая действий при входе и выходе.

Пример. На рис. 6.26 изображена диаграмма состояний для класса Controller, служащего для управления температурой и освещением в системе управления климатом теплицы.

Мы видим, что все объекты этого класса начинают свою жизнь в начальном состоянии "Ожидание"; затем они изменяют свое состояние по событию "Ввод климатического задания", для которого не предполагается явных действий. Дальше динамическое поведение этого класса состоит в переключении между состояниями "День" и "Ночь"; оно определяется событиями "Восход" и "Закат" соответственно; с этими событиями связаны действия по изменению освещения. В обоих состояниях событие понижения или повышения температуры в теплице вызывает обратную реакцию – операцию "Изменить температуру". Мы возвращаемся в состояние "Ожидание", когда поступит событие "Отмена климатического задания".

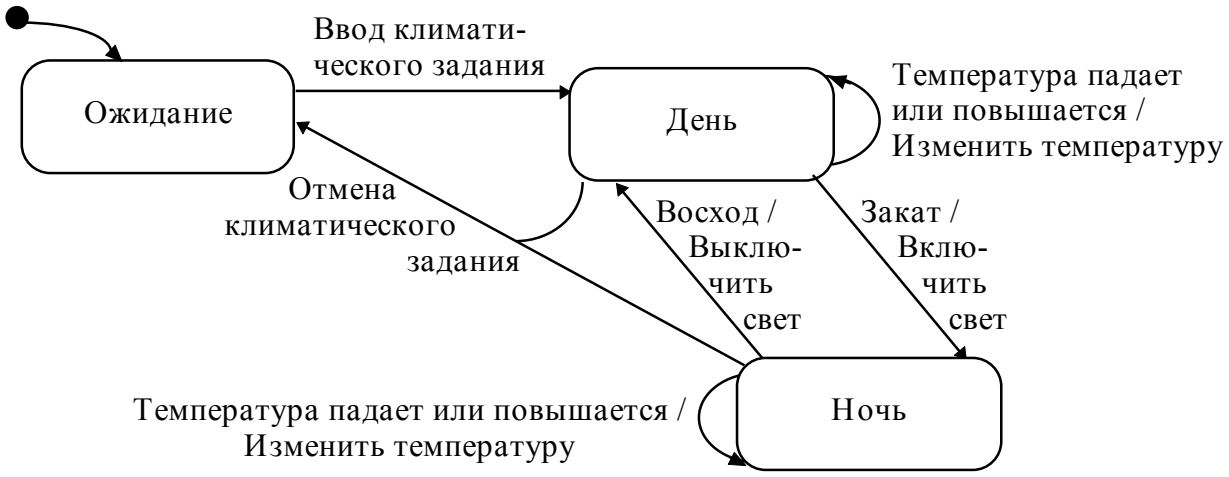

Рис. 6.26. Диаграмма состояний для объекта класса Controller

На диаграмме состояний возможно показать подсостояния – состояния, являющиеся частью других состояний. ***Простым*** называется такое состояние, которое не имеет внутренней структуры. Состояние, у которого есть подсостояния, называется ***составным***. Составное состояние изображается так же, как и простое, но имеет дополнительный графический раздел, в котором показаны подсостояния. Глубина вложенности состояний не ограничена. Если в объемлющем составном состоянии имеется несколько подсостояний, то говорят, что объект одновременно находится в составном состоянии и в одном из подсостояний.

Пример. На рис. 6.27 показаны внутренние детали составного состояния "Охлаждение", то есть вложенные в него состояния.

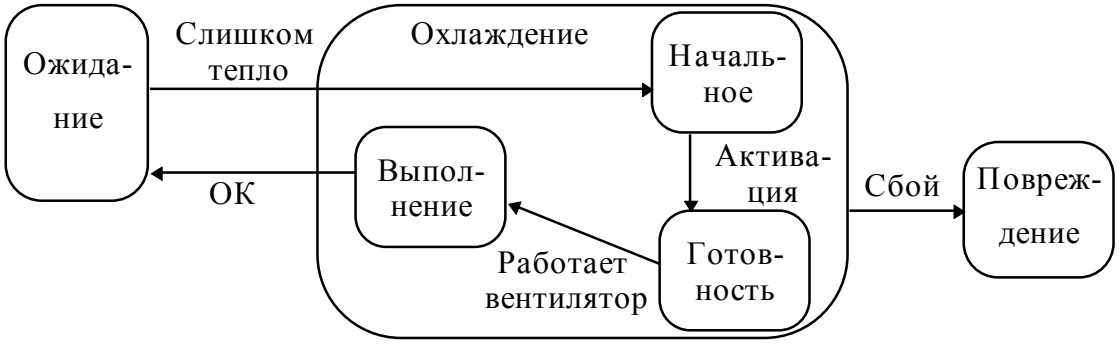

Рис. 6.27. Вложенные состояния

Переходам между состояниями разрешено начинаться и кончаться на любом уровне. Рассмотрим различные формы переходов:

– переход между одноуровневыми состояниями (такой, как из "Готовность" в "Выполнение") – простейшая форма перехода;

– переход непосредственно в подсостояние (как переход из "Ожидание" в "Начальное"), или непосредственно из подсостояния (как из "Выполнение" в "Ожидание"), или одновременно и то, и другое;

– переход из составного состояния (как из состояния "Охлаждение" через событие "Сбой" в состояние "Повреждение");

– переход в состояние с вложенными подсостояниями (например, предыдущий переход в состояние "Повреждение").

Если целевое состояние является составным, то оно должно иметь некоторое начальное подсостояние, куда управление попадает при входе в составное состояние после выполнения ассоциированного с ним действия при входе (если таковое определено). Если же целевым является одно из вложенных состояний, то управление попадает в него, но опять-таки после выполнения действий при входе в объемлющее составное состояние и в подсостояние (если таковые определены).

Указание перехода из составного состояния означает, что он осуществляется из каждого подсостояния этого суперсостояния. Для перехода, исходящего из составного состояния, когда исходным является либо оно само, либо какое-либо из его подсостояний, управление сначала покидает вложенное подсостояние (тогда выполняется его действие при выходе, если оно определено), а затем – составное состояние (тогда также выполняется действие при выходе).

6.5. Диаграмма деятельности

Диаграмма деятельности показывает поток переходов от одной деятельности к другой. Она включает состояния деятельности, состояния действия и переходы.

Диаграмма деятельности, как и диаграмма состояний, отражает процесс перехода из состояния в состояние. При этом все или большинство состояний

являются состояниями деятельности, а все или большинство переходов обусловлены завершением деятельности в состоянии-источнике.

Состояния системы, представляющие собой выполнение некоторого действия, называют ***состояниями действия***. Состояния действия не могут быть подвергнуты декомпозиции, как и сами действия.

Состояние деятельности можно представлять себе как составное состояние, поток управления которого включает только другие состояния деятельности и действий. Состояния деятельности могут быть подвергнуты дальнейшей декомпозиции, вследствие чего выполняемую деятельность можно представить с помощью других диаграмм деятельности. Состояния деятельности не являются атомарными, то есть могут быть прерваны. Предполагается, что для их завершения требуется заметное время.

Можно считать, что состояние действия – это частный вид состояния деятельности, а конкретнее – такое состояние, которое не может быть подвергнуто дальнейшей декомпозиции.

Состояния деятельности и действия изображаются прямоугольниками с закругленными краями (рис. 6.28). Внутри такого значка можно записывать произвольное выражение. При этом у состояния деятельности могут быть дополнительные части, такие как действия входа и выхода (то есть выполняемые соответственно при входе в состояние и выходе из него) и подсостояния.

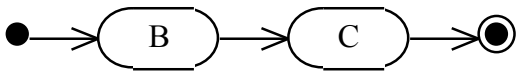

Рис. 6.28. Состояния и переходы

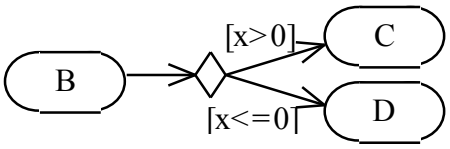

Рис. 6.29. Ветвление

Когда действие или деятельность в некотором состоянии завершается, поток управления сразу переходит в следующее состояние действия или деятельности. Для описания этого потока используются ***переходы***, показывающие путь из одного состояния действия или деятельности в другое. Переход изображается простой линией со стрелкой, как показано на рис. 6.28. Такие переходы являются нетриггерными (переходами по завершении), поскольку управление по завершении работы в исходном состоянии немедленно передается дальше.

После того как действие в данном исходном состоянии закончилось, выполняется определенное для него действие выхода (exit). Далее, безо всякой задержки, поток управления по переходу попадает в очередное состояние действия или деятельности. При этом выполняется определенное для нового состояния действие входа (entry), затем – действие или деятельность самого состояния и следующий переход. Поток управление должен где-то начинаться и заканчиваться. Как показано на рис. 6.28, можно задать как начальное состояние, так и конечное.

Если последовательных переходов недостаточно для моделирования потока управления, можно включить в модель ветвление, которое описывает различные пути выполнения в зависимости от значения некоторого булевского выражения. Как видно из рис. 6.29, точка ветвления представляется ромбом. В точку ветвления может входить ровно один переход, а выходить – два или более. Для каждого исходящего перехода задается булевское выражение, которое вычисляется только один раз при входе в точку ветвления. Ни для каких двух исходящих переходов эти сторожевые условия не должны одновременно принимать значение "истина", иначе поток управления окажется неоднозначным. Но эти условия должны покрывать все возможные варианты, иначе поток остановится. Для удобства разрешается использовать ключевое слово else для пометки того из исходящих переходов, который должен быть выбран в случае, если условия, заданные для всех остальных переходов, не выполнены.

Диаграммы деятельности могут служить в качестве блок-схем, причем внутри состояний можно записывать действия, применяя синтаксис используемого языка программирования. Например, на рис. 6.30 показана часть диаграммы деятельности, изображающая итерационный процесс.

Пример. На рис 6.31 изображена диаграмма деятельности, моделирующая структуру работ по возведению здания.

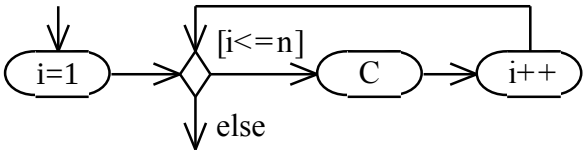

Рис. 6.30. Итерационный процесс

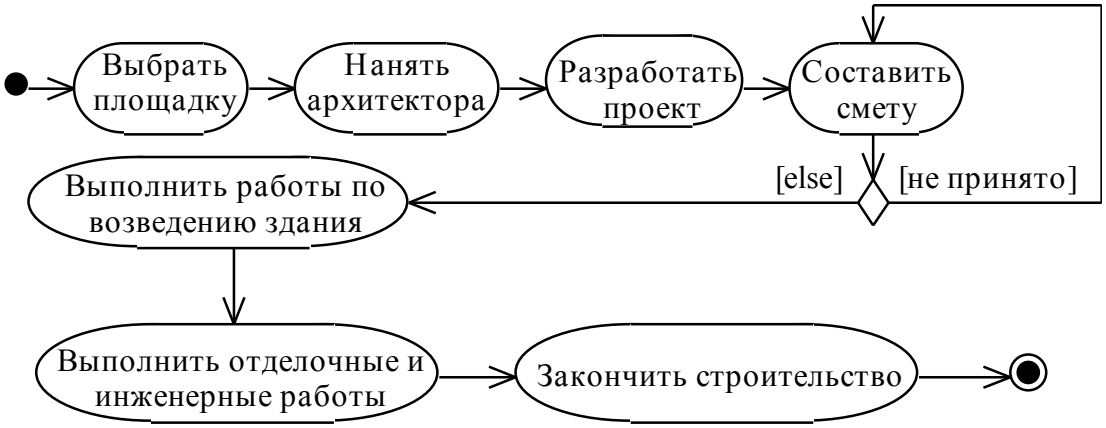

Рис. 6.31. Диаграмма деятельности для работ по возведению здания

ЛИТЕРАТУРА

1. *Бадд Т.* Объектно-ориентированное программирование в действии. – СПб.: "Питер", 1997.

2. *Боггс У., Боггс М.* UML и Rational rose. – М.: Лори, 2000.

3. *Буч Г.* Объектно-ориентированное проектирование с примерами применения. – М.: Конкорд, 1992.

4. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. – М.: "Издательство Бином", СПб.: "Невский диалект", 1998.

5. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя. – М.: ДМК, 2000.

6. *Пол А.* Объектно-ориентированное программирование на C++. – СПб.; М.: "Невский диалект" – "Издательство Бином", 1999.

7. *Романовский К.Ю., Кузнецов С.В., Кознов Д.В.* Объектно-ориентированный подход и диаграммы классов в UML // Объектно-ориентированное визуальное моделирование. – СПб: Изд-во С.-Петербургского ун-та, 1999.

8. *Страуструп Б.* Язык программирования C++. – СПб.; М.: "Невский диалект" – "Издательство Бином", 1999.

9. *Фаулер М., Скотт К.* UML в кратком изложении. Применение стандартного языка объектного моделирования. – М.: Мир, 1999.

10. *Фридман А.Л.* Основы объектно-ориентированной разработки программных систем. – М.: Финансы и статистика, 2000.